

Mapping to Coarse Grain Reconfigurable Arrays Using Evolutionary Algorithms

by

Fred (Shing-Fat) Ma

A thesis submitted to the Faculty of Graduate Studies and Research in partial
fulfillment of the requirements for the degree of Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Carleton University

Copyright © 2005 by Fred (Shing-Fat) Ma



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 0-494-00803-2

Our file *Notre référence*

ISBN: 0-494-00803-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

This thesis presents a genetic algorithm (GA) to place circuit components on to an array of reconfigurable algorithmic logic units (ALUs), known as datapath units (DPUs). Word-oriented reconfigurable arrays are relatively new, and design flows for them are still developing. One of the challenges of such “coarse grain” arrays is the lesser flexibility with which their resources can be arranged and recruited. Placement methodologies for application-specific integrated circuits (ASICs) and bit-wise “fine grain” arrays are not directly applicable. The greater predefined structure in a coarse grain array results in placement restrictions on some components.

In the array used in this thesis, the coarse granularity resulted in such a high level of abstraction that it was not necessary to separate placement and routing. The routing was simple enough that routability could be accurately determined by the placement GA. Hence, the GA could simultaneously deal with placement, routing, and heterogeneous position constraints. The flexibility in formulating problems for a GA lent itself very well to a multifaceted approach to enforcing the various architecturally imposed constraints.

A comprehensive design case was undertaken to establish the difficulties which automated placement has to deal with. In addition to developing the GA and partially designing the test suite of circuits for its characterization, a preliminary analysis was made of evolutionary approaches to circuit synthesis for coarse grain reconfigurable arrays.

Acknowledgements

I would like to acknowledge the generous funding that made my degree possible, from Micro-net, NSERC, and Nortel.

Of the people I am indebted to, I would like to first mention Rui Wang at Nortel who, took me on as an intern. Regrettably, due to a tragic failure in personal health, I am not able to share this accomplishment with him. Others there were also instrumental in my finding my way to a solid research topic. Gregory Carleton, who always found the time to come by for a friendly chat, and who put me in touch with the Chameleon activity that would drive my research. Mark Rollins, who was actively engaged in that technology, and who helped with access to the technology when times got tough. David Starks and Eddie Hum, who not only helped tremendously with funding, but spearheaded the research group and cultivated such an open atmosphere that I was never at a loss for opportunity to be actively involved. Despite their tight schedules as managers at Nortel, they still found time to provide feedback on reports and paper drafts.

From Chameleon, I have Raj Karamchedu, Charlie Cump, and Mark (again, since he went there) to thank. It was extremely helpful of them to make their compiler available at Carleton.

I have my parents, relatives, and friends to thank for their understanding of my absence when I had to dig in, especially on the home stretch. And I have them to thank for their company (Ted for the midnight instant messaging), their encouragement, and the reminder that there is life outside of my degree.

Special thanks are due to committee members John Chinneck, Anthony White, Franz Oppacher, Pavan Gunupudi, and Mark Rollins for their critical advice in ensuring a polished thesis whose contributions show through clearly.

To my supervisors Calvin Plett and John Knight, I owe my degree, and no small amount of personal development. Aside from the funding that allowed me to pursue the Ph.D., I will never be able to thank them properly for their endless positive encouragement in pursuing the research goals of my degree, for their tireless help and support, for sharing their technical knowledge and enthusiasm, and for somehow knowing when I needed the space and time to tackle my own problems. And at least as importantly, I have them to thank for their often-times unreserved sharing of their time on non-Ph.D. issues in general.

Table of Contents

Abstract	ii
Acknowledgements	iii
Glossary	xviii
Acronyms	xxii
1. Introduction	1
1.1 Platform Characteristics for Telecom DSP	1
1.2 Characterizing Reconfigurability	2
1.3 Design Flow	4
1.4 Tool Support.....	5
1.5 Genetic Algorithms (GAs) for Place-and-Route	6
1.6 Problem Statement	6
1.7 Thesis Organization.....	6
2. Solution Methods for Combinatorial Problems	8
2.1 Optimal Methods Versus Heuristics.....	9
2.2 Search Space Methods.....	10
2.3 Simulated Annealing	12
2.4 Tabu Search.....	13
2.5 Genetic Algorithms	15
2.5.1 Hyperplane Resampling.....	16
2.5.2 Diversity.....	16
2.5.3 Global Search and Parallelism.....	19
2.5.4 Selection of Parents for Mating	20
2.5.5 Crossover	21
2.5.6 Preservation of Building Blocks	21
2.5.7 Population Replacement Scheme.....	24
2.6 Take-Away Points	26

3. Existing Reconfigurable Platforms and Physical Design Algorithms.....	27
3.1 Architecture-Dependent Factors.....	27
3.1.1 Granularity	27
3.1.2 Reconfiguration Speed.....	29
3.1.3 Distributed Control	31
3.1.4 Mapping and Algorithms	31
3.2 Fine Grain Platforms	32
3.2.1 DISC: Dynamic Instruction Set Computer (1995-1996)	32
3.2.2 Garp: Gate Array Processor (1997-2000).....	32
3.2.2.1 Comparison of Garp with DISC.....	32
3.2.2.2 Garp Interconnect.....	34
3.2.2.3 Garp Technology Mapping and Placement.....	35
3.2.3 Chimaera (1997-present)	36
3.3 Coarse Grain Platforms	36
3.3.1 DeHon (1994-1996): DPGA and MATRIX	37
3.3.2 RaPiD: Reconfigurable Pipelined Datapath (1996-Present).....	38
3.3.3 Pleiades (1996-2000)	41
3.3.4 PipeRench (1997-2002)	42
3.3.5 Wormhole (1997-1998)	43
3.3.6 Xputer (1990's).....	45
3.3.7 MorphoSys (1999-present)	45
3.3.8 Chameleon (2001+)	46
3.4 Conclusions/Summary.....	50
3.4.1 General Observations.....	50
3.4.2 Take-Away Points.....	53
4. Kernel Design Case	55
4.1 The FFT Kernel	56
4.2 Design Case Generalizability	58
4.2.1 Chameleon Fabric as a Representative Case	58
4.2.2 Generalizable Features of the FFT Relevant to Mapping.....	59
4.3 Kernel Design Flow.....	60
4.4 Datapath Design	62
4.4.1 Dynamic Range Considerations.....	63
4.4.2 Complex Arithmetic	64
4.4.3 Twiddle Address Generation and Stage Tracking	69
4.4.4 Data Set/Memory Organization, Butterfly Scheduling, and Kernel Architecture	69
4.4.5 Delay Lines	72

4.4.6	Data Set Access Pattern and Address Generation	73
4.4.7	Retiming Butterfly I/Os	78
4.5	Control Path Design	82
4.5.1	Main State Machine	82
4.5.2	Shift Register Substate Machines	82
4.5.2.1	Walking Bit State Machine (Modified Johnson Counter).....	86
4.5.2.2	Walking Window State Machine	86
4.6	Functional Verification and Resource Utilization.....	88
4.7	Placement	90
4.7.1	Floorplanning the Datapath	90
4.7.2	Control Path Placement	92
4.8	Summary and Conclusions	95
4.8.1	Generalizability of Design Case	95
4.8.2	Demonstrated Viability of New Paradigm.....	96
4.8.3	General Observations and Summary Points	96
4.8.4	Platform Architecture	97
4.8.5	Design Methodology.....	98
4.8.5.1	Datapath	98
4.8.5.2	Control Path.....	98
4.8.6	Take-Away Points.....	99
5.	The Placement Problem and Comparison with Current Approaches.....	101
5.1	Problem Formulation.....	101
5.1.1	Challenges Due to Interconnect.....	101
5.1.2	The Placement Problem	102
5.1.3	Mathematical formulation.....	103
5.1.4	Simplified Archetypal Placement Problem.....	104
5.2	Distinguishing Features of Placement for Chameleon-Like Architecture	106
5.3	Choosing GAs for the Iterative Search.....	106
5.4	Implications for Cost Structuring Against Clustering DPUs .	107
5.5	Comparison With Placement for Surveyed Platforms	107
5.6	Comparison with GAs in Other Placement Problems	108
5.7	Summary and Conclusions	110
6.	Developing the GA for Placement.....	111
6.1	Simplified GA for the Basic Ordering Problem.....	113

6.1.1	Gene Encoding and Crossover.....	114
6.1.1.1	Partially Matched Crossover	115
6.1.1.2	Random Keys Representation	116
6.1.1.3	Uniform Crossover.....	118
6.1.2	Infeasibilities.....	118
6.1.2.1	Local Search and Lamarckian Evolution	120
6.1.3	Cost Structure	121
6.1.4	Fitness Scaling	122
6.1.4.1	Simple Linear Fitness Scaling.....	124
6.1.4.2	Sigma Truncation	125
6.1.4.3	Rank Based Fitness	126
6.1.5	Measuring Diversity	132
6.1.5.1	Challenges to Quantifying Diversity	133
6.1.5.2	New Intuitive Basis for Quantifying Scatter	135
6.1.5.3	Intuitive Normalization for Metric.....	140
6.1.5.4	Generating Trial Data.....	140
6.1.5.5	Reference Points for Normalization.....	141
6.1.5.6	The Reference Measure for Comparison	147
6.1.6	Implementation Environment	147
6.1.6.1	Tool Flow Transition.....	149
6.1.7	Example: Simple DPU Ordering for FFT	149
6.1.7.1	PMX	149
6.1.7.2	RKR.....	153
6.1.7.3	The Importance of Mutation	156
6.2	Full Placement Problem	158
6.2.1	Insights from C++/STL Implementation	158
6.2.1.1	Artifacts Due to Ranking and Cost Compression	158
6.2.1.2	Migration of Basic Ordering Problem.....	159
6.2.2	Heterogeneous Constraints and Features	159
6.2.2.1	Binding Datapath Units to Tiles.....	160
6.2.2.2	Repair of Tile Binding Violations.....	167
6.2.2.3	“Polarity”: Even/Odd Row Constraints.....	171
6.2.2.4	Global Vertical Wires.....	173
6.2.2.5	Placement of Multipliers	173
6.2.2.6	Scheduling Repairs/Penalization of Violations.....	174
6.3	Overall GA	175
6.4	Summary/Conclusions.....	175
7.	Experimental Results and Discussion.....	182
7.1	Algorithm Performance	182
7.1.1	GA Characterization Metrics Developed on the FFT	182
7.1.1.1	GA Performance for the FFT With All Constraint Enforcement Mechanisms Enabled.....	183
7.1.1.2	Testing the Individual Constraint Mechanisms.....	184
7.1.2	Other Kernels	187
7.1.2.1	The Case Against Random Kernels	187

7.1.2.2	Datapath Design of Real Kernels	187
7.1.2.3	GA Performance Across Kernels	190
7.2	Calibration of Diversity Metric	192
7.2.1	Comparison of Metrics	192
7.2.2	Adjustment of Normalization	194
7.3	Summary and Conclusions	194
8	Conclusions	197
8.1	Discussion	197
8.2	Conclusions	198
8.3	Contributions	199
8.4	Future Work	202
8.4.1	Diversity Feedback	202
8.4.2	A GA Approach for Synthesis	202
8.4.3	Other Approaches to Placement	204
	Appendix A:Generic Design Flows	207
A.1	ASIC Flow.....	207
A.2	Coarse Grain Reconfigurable Array Flow.....	207
	Appendix B:Details of Surveyed Platforms.....	211
B.1	Caching Kernels and Configuration Bits.....	211
B.2	Interconnect and Scalability	211
B.3	Dynamic Instruction Set Computer	214
B.4	Garp Details.....	216
B.4.1	Replaced Ultrasparc in Garp.....	216
B.4.2	Garp Array Details.....	217
B.4.3	Garp Performance Evaluation.....	219
B.4.4	Garp Low Level Mapping.....	219
B.4.5	Garp Compiler	219
B.4.6	Gama: Garp's Technology Mapping and Placement.....	223
B.5	Chimaera Details	224
B.6	MATRIX Details	225
B.7	Pathfinder Routing	225
B.8	Pleiades Details	227
B.8.1	Pleiades Interconnect	227

B.8.2	Pleiades Power Reduction	227
B.8.3	Pleiades Prototype Part	228
B.8.4	CAD Development	229
B.9	PipeRench Details	230
B.9.1	Technology Migration	230
B.9.2	PipeRench Computational Model	230
B.9.3	PipeRench Compiling	230
B.9.4	Exploring PipeRench Architectural Parameters and Implementa- tion	231
B.9.5	PipeRench Interface to Host Processor.....	232
B.10	Wormhole Details.....	233
B.10.1	Wormhole's Datapath-Heavy Logic Resources	233
B.10.2	Wormhole Configuration Loading.....	233
B.10.3	Wormhole's Allocation of Device Resources	234
B.10.4	Wormhole Implementations	234
B.10.4.1	Wormhole's First Prototype.....	234
B.10.4.2	Stallion: The Second Wormhole Design.....	234
B.11	Xputer Details.....	235
B.11.1	Pre-mid-1990's [HHW90,HHR+91,HKR94]	235
B.11.2	Mid-1990's and Beyond [HK95,HHHN99,HHHN00a,HHHN00b].....	236
B.12	MorphoSys Kernel Scheduler	237
B.13	Chameleon Details	237
B.14	Other Platforms	238
B.15	Interconnect Networks	239
B.15.1	H-Tree Networks	239
B.15.2	aSOC.....	241
B.15.2.1	The aSOC Architectural Concept.....	241
B.15.2.2	Evaluation of aSOC.....	243
Appendix C:Survey of CAD Efforts.....		247
C.1	Input Specification and Typical Compilation Flows.....	247
C.2	Verification.....	251
C.3	Tools	251
C.3.1	DEFACTO	251
C.3.2	Handel-C	251
C.3.2.1	Hardware Inference	253
C.3.2.2	Timing Control	253
C.3.2.3	Non-Partial Configurability.....	256
C.3.2.4	Handel-C Versus HDLs	257

C.3.2.5	Commercialization and Secrecy.....	257
C.3.3	JHDL.....	258
C.3.4	Streams-C.....	260
C.3.5	C/Fortran Compilation to RTL for a Custom MIMD.....	262
C.4	Algorithms and Methodologies.....	262
C.4.1	HySAM.....	262
C.4.2	Partitioning of netlist for FPGA.....	264
C.4.3	Compilation of C Targeting Coarse Grain Array.....	265
C.4.4	Other Mapping Algorithms.....	266
C.5	Conclusions/Summary.....	267
Appendix D:GA Details.....		269
D.1	GA Convergence, FFT Kernel Design Case.....	269
D.1.1	Simple DPU Ordering Problem.....	269
D.1.1.1	RKR: Searching for Good Search Conditions.....	269
D.1.1.2	Population Replacement Scheme.....	276
D.1.2	Full Placement Constraints.....	278
D.1.2.1	Selection of Mutatees from Population.....	278
D.1.2.2	Data from Testing Constraint Mechanisms Separately.....	283
D.2	The Case for a Simple Linear Diversity Measure.....	283
Appendix E:Genetic Algorithms for Synthesis.....		295
E.1	Problem at First Glance.....	297
E.1.1	Implications of Input Specification on Mapping.....	297
E.1.2	Challenges Beyond FPGA Mapping.....	298
E.2	Traditional or Apparent Approaches.....	299
E.2.1	CAD assisted microcoding.....	299
E.2.2	Vertex Clustering and DPU Packing.....	299
E.3	Using GAs to Determine DPU Personalities.....	300
E.3.1	Combinational Explosion of DPU Personalities.....	301
E.3.2	Combinational Explosion due to Time Dimension.....	301
E.3.3	Chromosome Evaluation.....	302
E.4	Using GAs to Discover Good Subcircuits.....	304
E.4.1	Genome Representation for Netlists of Multiport Components.....	304
E.4.2	Vector Chromosomes.....	305
E.4.2.1	Fixed Length Chromosomes.....	305
E.4.2.2	Variable Length Vector Chromosomes.....	308
E.4.3	Nonvector Chromosomes.....	313
E.4.3.1	Genetic Programming.....	313
E.4.3.2	EGG.....	316

E.5 Summary	316
References	321

List of Figures

Figure 1.1.	Flexibility and breadth of application domain trades off with performance/efficiency due to optimizations afforded by narrowing the focus of targeted applications.....	3
Figure 2.1.	A taxonomy of combinatorial algorithms, and examples.....	8
Figure 2.2.	Search space optimizers seek a better solution by taking a “step” in the solution space from a given solution.....	10
Figure 2.3.	Framework for simulated annealing algorithm.....	13
Figure 2.4.	Framework for tabu search algorithm.....	14
Figure 2.5.	Framework for genetic algorithm.....	15
Figure 2.6.	Simple crossover creates offspring in the hyperplanes of the parents.....	16
Figure 2.7.	Roulette wheel selection of parents for mating.....	20
Figure 2.8.	Two-point crossover. Crossover points are randomly determined.....	22
Figure 3.1.	Garp architecture.....	33
Figure 3.2.	Garp array details from [HW97].....	34
Figure 3.3.	Garp Interconnect detail [Gar97].....	35
Figure 3.4.	Preliminary ideas on reconfiguration from DeHon.....	37
Figure 3.5.	MATRIX [MD96].....	39
Figure 3.6.	A RaPiD cell [ECFF96].....	40
Figure 3.7.	Pleiades architecture [Rab97].....	41
Figure 3.8.	PipeRench operation [GSB+00].....	43
Figure 3.9.	Colt prototype consisting of 4x4 coarse grain blocks.....	44
Figure 3.10.	Chameleon reconfigurable fabric.....	47

Figure 3.11. Example of using Verilog mnemonics to define the DPU personalities and associated index values for a DPU named BfyIPadrsMasker.	49
Figure 3.12. Control path view of tile: state bits consist of PLA output registers that are fed back to the PLA input.	50
Figure 4.1. A small, illustrative 8-point FFT ($N=8$).....	57
Figure 4.2. Design flow targeting Chameleon platform.....	61
Figure 4.3. The FFT of a $2N$ -point time-domain record can have magnitudes that are 2^N times greater than the magnitude in the time-domain record.	64
Figure 4.4. Complex multiplier with one multiplication per cycle, broken down by DPUs and real multipliers.	65
Figure 4.5. Long hand multiplication of N -bit multiplicands yields an $(2N-1)$ -bit product.	66
Figure 4.6. Heading off an anticipated overflow problem.	67
Figure 4.7. Uniform attenuation of both butterfly outputs pulls data points and Twiddle factors away from the unit circle.	68
Figure 4.8. Twiddle factor address generation in the lower 16 bits by modulo-512 accumulation of $1 \ll i-1$ for stage i	70
Figure 4.9. Architecture of FFT datapath. Delay	71
Figure 4.10. Animation of Chameleon's method to squeeze 3 delay cycles from 1 DPU, as described in [Cha01b].	72
Figure 4.11. Repeated butterfly for N -point decimation-in-frequency FFT with input-normal ordering, $N=16$	74
Figure 4.12. The addresses of a butterfly's inputs can be generated by inserting a bit into the i^{th} bit position, where i is the FFT stage being processed.	76
Figure 4.13. Address generator for butterfly input data.	77
Figure 4.14. Interleaved butterflies for top and bottom data banks.	78
Figure 4.15. Determine the muxes by identifying when ports and registers source and sink which pieces of data.	79

Figure 4.16. Sharing registers at butterfly input.....	80
Figure 4.17. Merging of delays at the butterfly output.	81
Figure 4.18. Example of a main state machine.	83
Figure 4.19. Shift register state machines.	85
Figure 4.20. Example of coding for DPU personality control.	87
Figure 4.21. DPU personality changes at different times.	89
Figure 4.22. Floorplan view of the single-slice FFT.....	91
Figure 4.23. Supporting signals for SRB bits already in each tile, as well as unplaced SRB bits and associated dependencies being marked off as they become placed. ...	94
Figure 5.1. DPUs in a netlist are ordered onto a slice.....	104
Figure 6.1. GA framework of Figure 2.5 in greater detail.	112
Figure 6.2. Decoding of a chromosome to a solution, and its evaluation.	113
Figure 6.3. Chromosome for the basic ordering problem.	113
Figure 6.4. Two-point crossover applied to an ordering problem.....	114
Figure 6.5. Obtaining an ordering from a chromosome under RKR.....	116
Figure 6.6. Uniform crossover.	118
Figure 6.7. Handling infeasibilities.	120
Figure 6.8. Typical convergence for different interconnect cost schemes.....	123
Figure 6.9. Linear fitness scaling.	124
Figure 6.10. Cost distributions for small and large populations. Large populations have greater chance of greater outliers with larger costs.	125
Figure 6.11. Linear fitness scaling with C_{\max} dynamically taking the value of the greatest cost in each generation.....	126

Figure 6.12. Scaling by sigma truncation.....	127
Figure 6.13. Ranked based fitness. Chromosome index assumes that chromosomes are sorted according to descending cost.	128
Figure 6.14. Selection of parents by unrolling the roulette wheel of ranked based fitnesses. 131	
Figure 6.15. Spinning the roulette wheel arm.	132
Figure 6.16. Some methods of measuring diversity of solutions in a search space.	134
Figure 6.17. Intuitive recognition of scatter in a gene by histogramming the gene's values for the entire population.....	137
Figure 6.18. Distribution of values for a single gene over the population. The gene's value range is [0.0,1.0].	139
Figure 6.19. Shaping the PDF of randomly generated gene values.	142
Figure 6.20. The diversity metric for gene value distributions that have coalesced into varying degrees of nonuniformity.....	143
Figure 6.21. The simple gene diversity metric.....	146
Figure 6.22. The basic ordering problem in Matlab using PMX and steady-state evolution. 151	
Figure 6.23. RKR, steady-state evolution, 2-point crossover, mutation of general population members.....	154
Figure 6.24. Controlling selective pressure in ranked fitness.	156
Figure 6.25. Simple DPU ordering problem: Comparison of different mating/mutation schemes for PMX and RKR.....	157
Figure 6.26. Comparison of differing mating/mutation schemes using C++/STL implementation.	160
Figure 6.27. Initial approach to binding DPUs to tiles, assuming 4 DPUs/tile and $N_{\text{tiles}}=3$ tiles.....	162
Figure 6.28. Tile subranges delineated by interpolating between unbound DPU.....	164

Figure 6.29. Floating key-based tile boundaries and physical tile boundaries for the case of Figure 6.27(a), after bound DPUs have their keys remapped to the fixed sub-ranges of their home tiles, and all DPUs are combined and sorted.	165
Figure 6.30. Repair heuristic for tile binding violation.....	169
Figure 6.31. Scheduling of repairs.	176
Figure 6.32. Final genetic algorithm, as an elaboration of Figure 2.5.	177
Figure 7.1. GA performance for various PFK (shown here as “Population Change/Generation”) and KFO.....	185
Figure 7.2. Distribution of convergence times for placement for different kernels.....	191
Figure 7.3. Sample FFT placement to demonstrate diversity measures, KFO=0.5=PFK... 193	
Figure 8.1. A possible way to feed diversity back to the GA.	203
Figure 8.2. Fully interconnected DPUs. The embedded time-varying personalities determine which wires are used by which muxes in each cycle.....	204
Figure 8.3. GA and simulated annealing perform similarly.....	204
Figure B.1. Conceptual design hierarchy as design increases in size.	213
Figure B.3. DISC implementations.....	214
Figure B.2. DISC caching of hardware kernels corresponding to opcodes of an FPGA-based processor.	215
Figure B.4. DISC-II’s recursive object thinning algorithm [WH96].....	216
Figure B.5. Garp cell architecture [Gar97].	218
Figure B.6. Garp evaluation.....	220
Figure B.7. Hyperblock formation for VLIW compilation [CW98].	221
Figure B.8. MATRIX configured to perform convolution in four different ways.	226
Figure B.9. Hierarchical 2-level mesh for Pleiades [ZWGR99].....	228

Figure B.10. PipeRench architecture [GSB+00].....	231
Figure B.11. Optimizing PipeRench[GSB+00].	232
Figure B.12. HSRA's hierarchical interconnect network takes the form of a nested "H" shaped binary tree [TMJ+99].....	240
Figure B.13. HSRA deskewing.....	241
Figure B.15. aSOC crossbar switch and control circuitry [LST00].....	242
Figure B.14. aSOC's mesh to integrate disparate cores. Details taken from [LST00]. ..	243
Figure B.16. aSOC arrays	244
Figure B.17. aSOC results for testbench algorithms applied to the two cores compared with a single MIPS R4000, as well as the same cores with arbitrated bus and packet switch data communication [LST00].....	246
Figure C.1. Typical flows for mapping of functionality to kernels.	249
Figure C.2. Defacto flow to map high-level software code to configurable parallel proces- sors.	252
Figure C.3. Handel-C logic that results when a variable is assigned different expressions in different places in a linear sequence of assignments.	254
Figure C.4. Handel-C's token passing handshake for timing control.....	255
Figure C.5. Parallel sequences of Handel-C code require parallel token paths, and logic to wait for all sequences to complete [Pag96].	256
Figure C.6. JHDL's binding of Java constructors/destructors with the loading/freeing of kernels to/from the Xilinx FPGA cache on Virtual Computer Corporation's configurable computing machine (CCM). Schematic from [BH98].	259
Figure C.7. Streams-C architecture.....	261
Figure C.8. Target for the compilation of high-level software to MIMD RTL for an ASIC flow. Processor detail from [BRM+99]	263
Figure D.1. Captured statistics from a nonconvergent case of initial RKR implementation, 50% mutation rate applied to each offspring i.e. 2 offspring and up to 2 mu- tants per generation.	270

Figure D.2. Initial scheme for quantifying diversity.....	271
Figure D.3. A gradual transition of the mean violation count towards the new minimum as the genes proliferate through the population.	273
Figure D.4. Initial RKR implementation: 2-point crossover, mutation of offspring, unconditional acceptance of unique kids into population	274
Figure D.5. Rank of kids relative to the population, for a search with 10% chance of mutating each offspring.	275
Figure D.6. (Next page). Simple DPU order problem: Comparison of unconditional and competitive acceptance of kids for PMX and RKR.....	276
Figure D.7. (Next page): Comparing the selection of mutates. Mutates were selected either with uniform likeliness or according to ranked fitness.....	278
Figure D.8. Comparison of placement for the FFT netlist with extra tile/polarity bindings. The two variations correspond to selection of mutates uniformly and based on ranked fitness.	280
Figure D.9. Extra hard placement problem with uniform selection of mutates. Extra tile and polarity constraints were added to the FFT kernel.....	281
Figure D.10. Extra hard placement problem with ranked selection of mutates. Extra tile and polarity constraints were added to the FFT kernel.....	282
Figure D.11. The FFT kernel placement with only valid constraints (no arbitrary extra constraints). All constraint mechanisms were enabled.....	285
Figure D.12. FFT kernel placement. Disable all tile binding mechanisms. Only polarity binding mechanisms enabled.	286
Figure D.13. A second data set for FFT kernel placement with only polarity binding mechanisms enabled. Compliance with tile bindings are completely random...	287
Figure D.14. FFT kernel placement. Disable repair of tile violations. Compliance with tile bindings rely entirely on key remapping and penalization of tile edge offsets.	288
Figure D.15. FFT kernel placement. Disable penalization of tile edge misalignments. Compliance with tile bindings rely entirely on key remapping and repairs of violations.....	289

Figure D.16. FFT kernel placement. Disable key remapping for tile-bound DPUs. Gross infeasibilities are thus possible.	290
Figure D.17. FFT kernel placement. Disable key remapping for tile-bound DPUs and penalization of tile edge misalignments.	291
Figure D.18. FFT kernel placement. Disable all polarity binding mechanisms. Only tile binding mechanisms enabled. Compliance with polarity bindings are completely random.	292
Figure D.19. FFT kernel placement. Disable repair of polarity violations. Compliance with polarity bindings rely entirely on penalization.	293
Figure E.1. Y-chart for ASIC design, as described in [GDWL92]. Corresponding elements for targeting coarse grain reconfigurable platforms are shown.....	296
Figure E.2. An extremely simple vector genotype for a DPU netlist.	309

List of Tables

Table 1.1: Design flows for ASICs and reconfigurable logic.....	4
Table 4.1: Generalizable features of design case	58
Table 4.2: Butterfly upper input address as a function of FFT stage and butterfly count. An 8-point 4-butterfly FFT is assumed.	75
Table 4.3: Resource usage for FFT kernel.....	94
Table 7.1: Kernels tested.....	190

Glossary

Term	Meaning
allele	The particular value attached to a gene
bijjective	A mapping which is one-to-one and onto (surjective)
biquad	A second-order block from which most IIR filters are constructed
building blocks	A cluster of genes which give beneficial elements to a solution
butterfly	A basic 2-input 2-output circuit in the FFT which does a complex multiply and two additions
chromosome	An ordered collection of genes identifying the parameters of a problem solution
clones	Kids that are identical to existing population members
coadapted alleles	A favourable characteristic of a solution which depends on several interacting alleles
collision	An unallowed duplication of gene values in a chromosome, particularly for an ordering problem
constraints	Limitations on the feasibility of solutions
crossbar	A circuit that allows its input leads to be connected to its output leads by any desired permutation
delay line	Here, a digital delay line in which bits are delayed by a certain number of clock cycles
design space	Used interchangeably with search space and solution space
diversity	The difference in the genetic material in a breeding population
epistasis	A nonlinear interaction of genes which produces an exceptional increase in fitness
evolutionary algorithms	Algorithms that evolve solutions to given problems. Includes these variations: genetic algorithms, genetic programming, evolutionary programming, and evolution strategy
exploitation	Favouring regions of the search space in which to search, according to current information of where is good. Strong bias in favouring "good" solutions based on their cost.

Term	Meaning
fabric	The array of reconfigurable elements available for computation, esp for Chameleon
fitness	A figure of merit assigned to each chromosome
gene	An element in a chromosome corresponding to some parameter in the problem (see allele)
generational replacement	Replacing all the members of a population by new ones at once
genome	Synonym for chromosome
genotype	The definition of a genome. The meanings assigned to the genes.
Hamming distance	The number of elements by which two vectors differ.
high-level synthesis	In digital hardware design, an approach to convert potentially untimed functional specification to a set of hardware resources, and a schedule of their use.
implicit parallelism	One offspring samples a large number of hyperplanes in the chromosome
inversion	A method of genetic reordering which helps preserve beneficial building blocks.
kids	New population members generated by mating, mutation or both (synonym newcomers)
kids fraction offspring	The fraction of the kids that are offspring rather than mutated "adults"
Lamarckian evolution	Modifying the solutions generated by genotype-phenotype decoding e.g. by local search and optimization, but not modifying the underlying genotype.
latency	The delay between a circuit's input signal and its processed output (see throughput)
linkage	The tendency of beneficial clusters of genes to be propagated to children
locus	The position of a gene in a chromosome
marginal overflow	Wrap-around in a 2's-complement number. Leads to a large error.
Mealy machine	A finite state machine in which input signals can affect some outputs without passing through storage

Term	Meaning
Moore machine	A finite state machine in which all inputs must pass through a storage register before exiting
newcomers	New population members generated by mating, mutation or both (synonym kids)
nonblocking	In parallel programming, execution of a code line is not blocked until the completion of its predecessor
offspring	New population members generated by mating as opposed to mutation
one hot	A finite state machine in which all the bits in the state storing register are 0, except for one 1.
phenotype	A solution represented by a genotype, in terms of original problem parameters rather than gene-encoded values.
pipeline	A signal processing system where the delayed output comes out at exactly the input rate
polarity	A constraint where only odd numbered DPUs can write to memory and only even ones can read
population fraction kids	The fraction of the population that is replaced by kids in each generation
product term	Variables logically ANDed together. PLA size is often designated by the number of product terms
rank based fitness (ranked fitness)	For a population of N , the highest cost genome has fitness $f=1$, next highest has $f=2$, ... , lowest has $f=N$
retiming	The shifting of delays around in a digital circuit or dataflow graph.
search space	Used interchangeably with design space and solution space
sigma truncation	The maximum allowed cost C_{Max} is some number of standard deviations from the mean cost. Fitness is $Cost - C_{Max}$.
size (of tile)	The number of DPUs in the tile, 7 for Chameleon, 4 in some examples in Chapter 5
slice	For Chameleon, a vertical set of 3 tiles. The whole chip contains 4 slices.
solution space	Used interchangeably with design space and search space

Term	Meaning
steady-state replacement	Replacing only 1 pair of population members per generation.
sum of products	Logic in the form of the OR of sets of variables ANDed together
surjective	A mapping which is onto. If $y = f(x)$, for every point in the domain of y , there is a x such that $y = f^{-1}(x)$
swapper/ swappee	A swapper is a constraint violating DPU which will be interchanged with another, the swappee
synthesis	The conversion of logic-level functional specifications to a circuit composed of library modules (gates); consists of converting functional specifications to generic logic, technology independent logic optimization, and technology mapping
technology mapping	Optimally mapping logic-level functional specifications to library modules that are optimized for the target platform
throughput	The rate (often maximum rate) at which a circuit processes continuous (streaming) input signals
tile	A computation unit in Chameleon, it contains 7 DPUs, 2 multipliers. 3 tiles make a slice
twiddle factors	The unity magnitude complex exponential constants used in the FFT
twins	Nonunique kids
uniform crossover	Alleles of parents are randomly interchanged
Verilog	A programming language used to describe digital circuits (see VHDL)

Acronyms

Term	Meaning
&&	Logical AND
	Logical OR
3GL	Third Generation Language
AC	A signal or data set that has been uniformly offset to have zero mean.
ALU	An arithmetic unit usually which can add, subtract, compare and do logic operations
ASIC	Application Specific Integrated Circuit. One designed for a specific application, not a processor
CAD	Computer Aided Design
CDF	Cumulative probability Distribution Function
CDFG	Control/Data Flow Diagram
CISC	Complex Instruction Set Computer
CLB	Control Logic Block, a basic element in Xilinx FPGAs containing logic and bit storage
CMOS	A technology for fabricating integrated circuits.
CRC	Cyclic Redundancy Check, used for checking for errors in data
DC	Direct Current. Used for any signal of frequency zero, or the zero frequency part of a signal. Also, the mean of a signal or data set.
DCT	Discrete Cosine Transform, a transform used in picture compression
DFD	Data Flow Diagram
DFG	Data Flow Graph
DPU	Data Processing Unit
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ERC/DRC	Electrical Rule Checking/Design Rule Checking. Two checks run on all integrated circuits
FFT	Fast Fourier Transform

Term	Meaning
FIR (filter)	Finite Impulse Response filter. Previous inputs are completely forgotten in a finite time.
FPGA	Field Programmable Gate Array.(see FPGA-)
FPGA-	A circuit that can generate any synchronous logic function within its capacity
FSM	Finite state machine, often used for control in digital circuits.
FSMD	FSM and associated datapath.
GA	Genetic Algorithm. In this thesis, we allow the meaning to include features from various <i>evolutionary algorithms</i> (a superset of GAs).
HDL	Hardware Description Language. A programming language to describe digital circuits
HLS	High-level synthesis.
HW	Hardware, particularly electronic circuits
IC	Integrated circuit.
ID	Identification
IDDQ	A method of production testing of integrated circuits based on the quiescent current they use.
IFU	The name "Wormhole" gives to its ALUs
IIR (filter)	Infinite Impulse Response filter. The output is a weighted average of past input and output values.
ILP	Integer Linear Programming
KFO	Kids Fraction that are Offspring (by breeding)
LSB	Least Significant Bit
LSM	Local Memory Storage. A block of word SRAM within a reconfigurable array.
LUT	Lookup table, used to do combinational logic in many FPGAs and in Chameleon
MAC	Multiply and Accumulate. A compound operation commonly used in taking the inner product of two numerical vectors.
MCM	Multi-chip module. An IC package with more than one IC die.

Term	Meaning
MIMD	A kind of microprocessor that simultaneously applies multiple instructions to multiple pieces of data.
MOO	Mutation of offspring only
MUX	A circuit that can select between multiple inputs to send to the output.
MPGM	Mutation of General Population Members
MSB	Most Significant Bit
O_MGPM	Only Mutation of General Population Members
OFDM	Orthogonal Frequency Division Multiplexing, used in modems and wireless LANs
PDF	Probability Distribution Function
PE	Processing Element
PFK	Population Fraction that are Kids
PLA	Programmable Logic Array. A matrix like circuit to implement combinational logic
PMX	Partially Matched Crossover
QAM	Quadrature Amplitude Modulation, a modulation method for wireless communication
RISC	Reduced Instruction Set Computer
RKR	Random Keys Representation
RTL	Register Transfer Language. A description of digital circuits as registers with logic between them
SA	Simulated annealing
SDC	Synopsys Design Compiler, here used to translate Verilog code into a netlist
SIMD	Single Instruction Multiple Data. A type of microprocessor that applies the same instruction to multiple pieces of data.
SOC	System on Chip. Systematizing the integration of various circuits into a chip in such a way as to avoid many of the complications in an ad hoc approach.
SRAM	Static Random Access Memory. (as opposed to DRAM, which forgets in milliseconds)

Term	Meaning
SRB	State Register Block. Contains registers for the control logic in the Chameleon reconfigurable array.
STL	Standard Template Library for C++
SW	Software
TSP	Travelling salesman problems
UV	Ultraviolet
VDD	Drain voltage. The positive voltage source for an IC.
VLIW	Very Long Instruction Word. A variation of superscalar processors by scheduling parallel operations at compile time rather than run time.
VSS	Source voltage. The ground voltage for an IC.
VHDL	A programming language used to describe digital circuits (see Verilog)
X+	Crossover plus some mutation scheme like MOO or MGMP

1. Introduction

Programmable logic has come a long way since the days of classic once-programmable logic devices (PLDs). They have evolved through stages of reprogrammability by erasing previous circuits with UV light and high voltage. The current trend of SRAM based FPGAs allows configuration bits to be clocked into the device and stored in much the same way as data, thus simplifying system integration of these devices. Until recently, the process of programming such devices has been relegated to system bootup; programming speed was not as important as data processing.

The current interest in reconfigurable hardware for telecom is rooted partly in the need for high throughput and the *repetitiveness* of DSP. It is beneficial to setup a pipelined datapath that can operate over a lengthy sequence of data samples. This characteristic of DSP has been referred to as *stream data*. Also associated with stream data is the *predictability* of the sequence length; most standards today delineate blocks or frames of prespecified lengths at different levels of data aggregation. This allows the control logic to be very simple, without speculative computing and conditional flushing of pipelines. By tailoring a platform for pipelined operation with simple control flow, programmable hardware can have high throughput and area efficiency, yet still retain reconfigurability.

1.1 Platform Characteristics for Telecom DSP

Reconfigurable hardware is a trade-off between the two extremes of microprocessors and ASICs in a continuum of reconfigurability. Compared to DSP processors (DSPs), ASICs are faster, consume less power, and have high computational density. However, they have very involved design flow, require expensive fabrication masks, and incur fabrication costs and delays. These liabilities discourage redesigns for bug fixes and feature upgrades. They also impede adaptability to changing standards. In addition to avoiding these liabilities, reconfigurable devices open up the possibility of remote, automated upgrades in a manner similar to software.

In contrast to an ASIC's specialization, microprocessors are the most reconfigurable of platforms [DeH96] in that the processor's hardware resources are utilized differently in each clock cycle, as dictated by the machine code. The overhead for this reconfigurability is the entire infra-

structure for streaming in and decoding instructions. Only a small portion of the silicon is performing computation at any time. However, it is possible to sequence through operations with very intricate, possibly nonrepetitive control flow.

The reconfigurable middle ground has traditionally been fine grain FPGAs, in which bit-wise logic can be independently configured. There has been growing interest in word oriented “coarse grain” configurable logic arrays. This is because, in addition to throughput, much of modern day DSP for telecom deals with computationally intense, word-oriented data. Optimizing the logic for word data can yield greater speed and functional density. The coarse grain array elements take the form of configurable ALUs, or *datapath units* (DPUs).

1.2 Characterizing Reconfigurability

How reconfigurable a device is has been associated with how little time there is between programming the device and its usage [DeH96]. In order of reconfigurability, this applies to processors, FPGAs that can be partially reconfigured (possibly without interrupting execution of running circuits on-chip), FPGAs that have to be fully programmed, devices that are programmed with extraordinary voltages, older FPGAs that require UV erasure, once-programmable devices, sea-of-gates arrays requiring only the final steps of fabrication, and ASICs.

There are some ambiguities with this “measure” of reconfigurability. For example, it is not atypical for ASICs to perform widely varying functions, depending on control signals. An alternative characterization is the platform’s breadth of applications, which depends on granularity as well as programming lead time (Figure 1.1). Between the extremes of a processor’s flexibility and an ASIC’s narrow focus are platforms that favour certain applications or algorithms, to varying degrees, by design. Narrowly focused architectures are very efficient for their target application domain.

For algorithms having operations that are not well aligned with the specialized architecture, more general purpose platforms can suffer less inefficiency. For example, coarse grain logic specialized for dot-product operations might not implement encryption or compression as efficiently as a general-purpose fine grain FPGA, which may be able to build more suitable macro elements from fundamental LUTs.

When the continuum is defined in terms of breadth of application versus specialization for efficiency in a narrow application domain, an ASIC with a great number of configurable operating

modes actually moves left in the trade-off spectrum of Figure 1.1(b), approaching microprocessors.

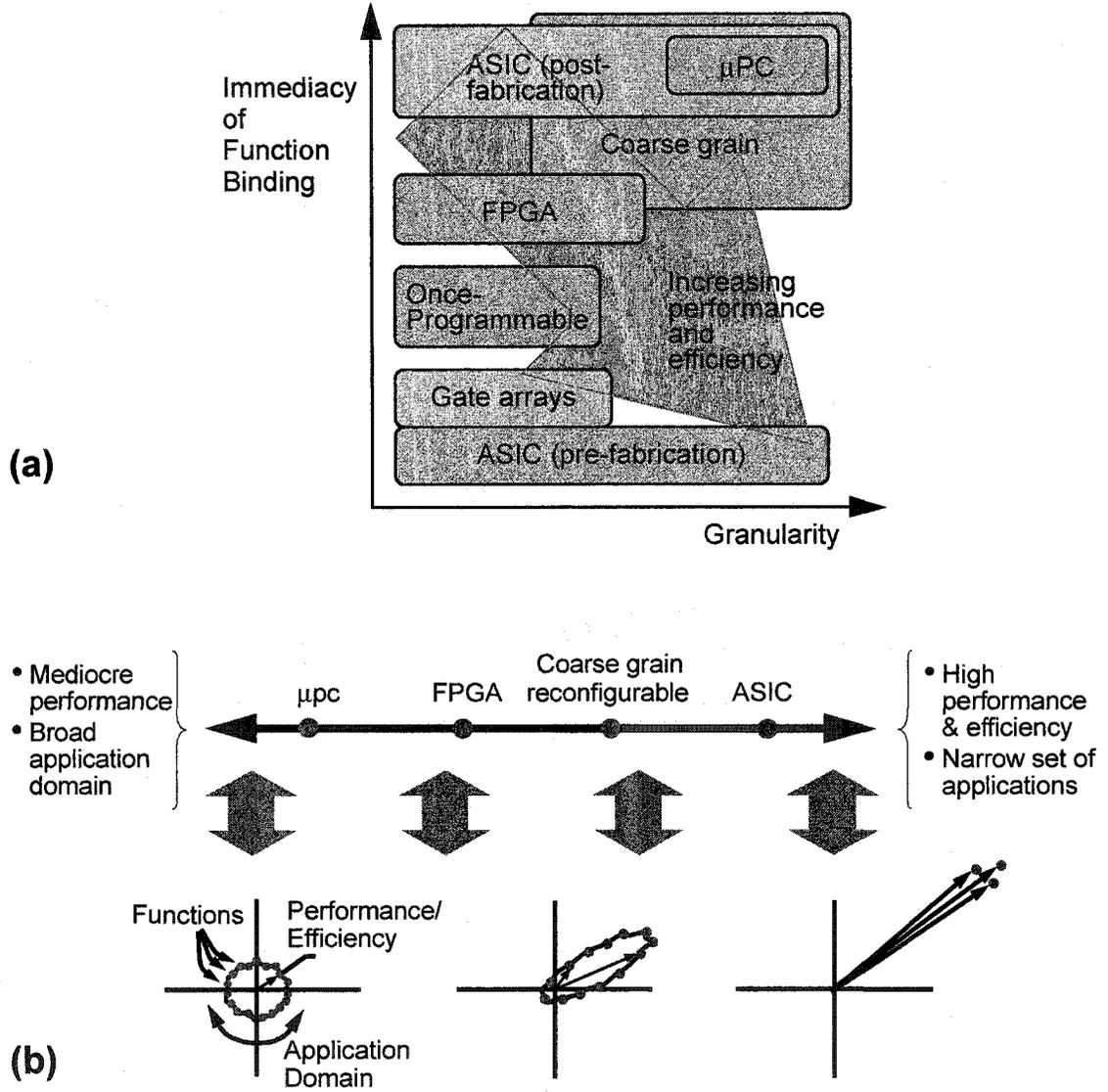


Figure 1.1. Flexibility and breadth of application domain trades off with performance/efficiency due to optimizations afforded by narrowing the focus of targeted applications.

1.3 Design Flow

It is helpful to have a global picture of the design flow for reconfigurable logic as compared to traditional hardware design flows. Table 1.1 summarizes typical ASIC and reconfigurable design flows detailed in Appendix A. Many of the fabrication, packaging, and testing phases for ASICs are avoided by reconfigurable arrays that are off-the-shelf parts.

Table 1.1: Design flows for ASICs and reconfigurable logic.

ASIC flow	Reconfigurable logic flow
Hardware/software partitioning. Hardware specification	
HDL design.	Kernel design
Functional verification through simulation	
Synthesis	
Functional and partial timing verification through simulation	
Physical design (placement and routing)	
Functional & timing verification	Functional & timing verification
Masks & fabrication	
Testing	
Packaging	
Testing	
In-system testing	

The *kernel design* stage consists of specifying the internal operations of the kernel at a high level, but with sufficient detail that it can be converted to a netlist of platform-specific array resources. This latter conversion is the *synthesis* stage. The subsequent *physical design* phase consists of a *placement* step, which binds the synthesized resources to location-specific resources on the array in such a way that the interconnecting wires will not likely be too long and slow; it also consists of a *routing* step, which allocates wiring resources to realize the interconnections in a way that meets timing requirements.

1.4 Tool Support

Due to the newness of coarse grain arrays, the design paradigms in each stage of the design flow are rudimentary, and constantly changing to find more effective forms. The supporting tools are thus limited in the functionality that they successfully automate. There are many manually intensive bottlenecks in the end-to-end design of a kernel, as will be demonstrated in the design case of Chapter 4.

The newness of coarse grain arrays also leads to a wide variety of array architectures, which further exacerbates the tool immaturity. This lack of a defacto standard platform removes a major conducive factor in the environment for the maturing of tools -- that of a large user/vendor base working under consolidated design methodologies.

At this stage, therefore, there is a need not only for automation tools at the various design flow phases, but an *easily extensible* framework from which a tool is customized for the constraints of different architectures. Not only would such an approach meet the plethora of architectures, but it would also allow an automation effort to keep up with the evolution of a single base architecture, as it changes to exploit new fabrication technologies.

The advantage of adaptable automation applies to all platforms, but is especially important for coarse grain arrays. This is because of the greater degree to which the functionality is prepackaged into larger function units with more completely defined operations. The lesser flexibility in the target resources necessitates greater imaginativeness in the mapping methodology to get an efficient mapping, and greater flexibility in the tool that will provides the automation.

In the place and route phase, for example, architecturally imposed inflexibility can manifest itself as hard constraints on where certain array elements can be bound. This may be in order to access the required memory banks, to share control logic with other array elements (which is determined by how control logic resources are grouped up), or simply because a functionality is needed that is only supported by certain array elements. A computer-aided design (CAD) approach should be able to *easily* adapt to such heterogeneous constraints.

1.5 Genetic Algorithms (GAs) for Place-and-Route

While investigating the design flow, two bottlenecks that were in dire need of address by CAD were the synthesis phase and physical design phase. This thesis focuses on the physical design phase. For the example architecture studied in this thesis (Section 3.3.8), the vendor tools were rarely able to find placements that respected architectural constraints, while manual efforts can take days for complex circuits.

Section 2.1 and Chapter 5 elaborate on reasons for exploring GAs to handle the seemingly arbitrary constraints associated with architecture. Briefly, their heuristic and stochastic nature is good for complex constraint-driven problems. Their generate-solution-and-evaluate approach simplifies incorporation of architectural constraints into the problem encoding, while their iterative nature makes their performance less sensitive to the vagaries of encoding the problem. Their population-based approach makes them easily parallelizable to accelerate the search for a solution, and provides a certain protection against the explosion of complexity with the problem size.

1.6 Problem Statement

For coarse grain arrays, conventional placement methods face difficulties due to (1) the diverse heterogeneous placement constraints, (2) the likelihood of rapid changes in platform architecture, and (3) the constant changes of a nascent tool flow. Hence, automation algorithms must be easily adaptable to the mapping problem.

This thesis studies the use of evolutionary algorithms for physical design targeting coarse grain reconfigurable hardware arrays. It investigates whether their flexibility and adaptability can tackle the seemingly arbitrary heterogeneous constraints that result from the array architecture, and still yield advantageous and computationally tractable CAD engines.

1.7 Thesis Organization

This thesis is organized as follows.

Chapter 2 provides background on solution methods for combinatorial problems, with emphasis on stochastic iterative search, and evolutionary algorithms in particular. Many of the difficulties in designing for coarse grain arrays can be cast into combinatorial problems.

A comprehensive survey is first performed on novel reconfigurable hardware platforms and associated CAD algorithms (Chapter 3).

A FFT design case is conducted in Chapter 4, that identifies efficient kernel design practices for the control path and datapath, bottlenecks, and refinements of the platform architecture.

Having identified the instantiation and interconnection problem, which is quite different from that for fine-grained FPGAs or ASICS, Chapter 5 restates the problem, discusses characteristics of place and route that distinguish it from corresponding design phases in ASICs and FPGAs, and presents reasons why evolutionary algorithms seem to be a good choice.

Chapter 6 presents the abstract part of the thesis work and shows the adaptations and innovations made to apply evolutionary algorithms to the physical design phase. It also presents some original work on diversity, developed to help identify premature convergence.

Chapter 7 gives the results of applying the algorithms to the example of Chapter 2, and several other examples. It also summarizes the diversity experiments.

Chapter 8 contains summarizing statements, conclusions, contributions, and future work

2. Solution Methods for Combinatorial Problems

As the coming chapters will show, many of the challenging tasks in the design flow can be cast as combinatorial problems. This chapter provides background on solution methods for such problems. Problem-specific implementation details are described in Chapter 6.

Figure 2.1 provides a bird's eye view of this section. We will focus on the heuristic, iterative quadrant. In particular, evolutionary algorithms are discussed in detail, primarily based on genetic algorithms.

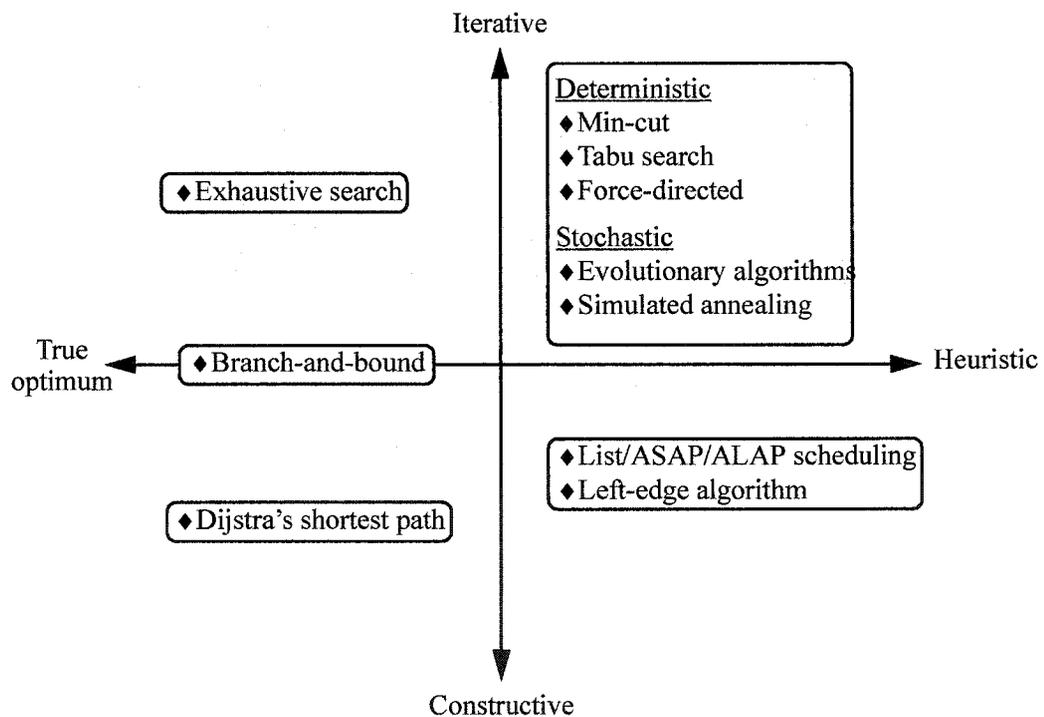


Figure 2.1. A taxonomy of combinatorial algorithms, and examples.

Branch-and-bound can be a cross between iterative and constructive because it solves a simplified problem to get a lower bound on the cost of a partial solution. Solutions to the simplified problem may also solve the full problem; in this way, solutions to the full problem are tried.

2.1 Optimal Methods Versus Heuristics

Classical combinatorial methods such as integer linear programming and dynamic programming find the *best* solution i.e. one that minimizes cost or maximizes profit. However, they become computationally demanding when the solution space becomes large. More importantly, they require a careful casting of the problem constraints into a specific form. The seemingly arbitrary and heterogeneous constraints that arise from coarse grain arrays can pose serious challenges to the applicability of such optimization algorithms. Architectural changes in successive generations of a platform may render useless the work in casting constraints into the right form.

Note that methods like integer-linear programming can be modified to apply to constraint based problems. One may also find their inherent complexity is not an overwhelming issue in the problem attacked here. However they were not considered in this work.

In engineering, the challenge is often to find a *good* solution. Whether the solution is optimal is less important than to have a method to find *a* solution. In many cases, it is at least as important that the method can be quickly adaptable as the details of the problem change. For such potentially suboptimal solutions, a range of heuristic algorithms are available [Fou84], particularly relating to problems in graphs, networks, and scheduling/ordering.

Constructive Algorithms

As a basis for selecting a heuristic approach, heuristic solutions may be classified in two broad categories: constructive and iterative. Constructive approaches consist of a set of rules to build up a solution one piece at a time e.g. one parameter at a time. Iterative approaches use heuristics to repeatedly find better solutions based on older solutions (possibly initial guesses from a constructive algorithm). Basically, each iteration visits another potential solution in the “space” of all possible solutions; the iterative process is a search of the solution space. Except where noted, we will refer to iterative solutions and search space methods synonymously. Search space methods are a generalization of gradient-based methods for continuous parameter problems. We use the terms *search space*, *design space*, and *solution space* interchangeably.

Iterative Algorithms

Constructive approaches can be contrasted with iterative approaches in that they typically repeat a series of construction steps that depend very much on the problem. On the other hand, the framework of a search space method as described above is quite generic; however, the problem

specific information is embedded in the evaluation of each solution's cost, which in turn depends on how the problem parameters are encoded as search space variables (basically, the axes of the solution space). The various search space methods differ in how they synthesize new solutions from old solutions.

Because of its iterative nature, search space methods contain a feedback which provides some robustness against vagaries in the problem dependent encoding scheme. This allows very flexible encoding of heterogeneous constraints such as those found in coarse grain reconfigurable processing arrays. Consequently, a constrained optimization problem can be modelled very naturally; it is more likely that changes in problem definition due to evolving platform architecture can be incorporated by modifying the existing work for the original problem.

2.2 Search Space Methods

As the name implies, search space methods look for solutions in an N -dimensional space of solution points, where N is the number of parameters used to encode the problem. Each axis of the solution space corresponds to one of the parameters used to encode the problem. The solution points in the solution space correspond to all the possible combinations of values for the N parameters. A (hopefully) better solution point is found from a current solution point by taking a "step" in the solution space (Figure 2.2). The new solution is then evaluated, and another step is taken. Different search space algorithms have different mechanisms to avoid being trapped in a local cost minimum. There is generally no way to determine if a good solution is a global minimum; however the question is often not of concern.

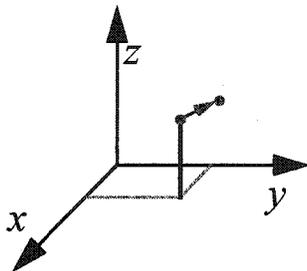


Figure 2.2. Search space optimizers seek a better solution by taking a "step" in the solution space from a given solution.

Here, solutions consist of 3 parameters: x , y , and z . The cost of the solution as a function of (x,y,z) is not shown.

In the following sections, three popular search space methods will be described briefly: simulated annealing, tabu search, and genetic algorithms (or more generally, evolutionary algorithms).

They get out of local minima by allowing the search to include poorer solutions, with the hope of reaching better regions in the solution space.

Notions of Locality

Even though a local gradient is not used to guide the search, as they are for continuous parameter problems, various notions of locality can be used to help adapt the search algorithms to different problems. Since the basis of the search space methods is to explore new candidate solution points based on knowledge of current and past solution points, a notion of similarity between solutions is useful. For this purpose, we will distinguish between *Hamming* locality and *Euclidean* locality.

Hamming locality refers to the fact that two solutions are similar in some sense if they differ in the values of only a few parameters. The quantity by which a parameter differs between the two solutions is not considered. This is appropriate when a parameter merely enumerates a design option, and the value does not represent a physical quantity. For example, consider the task of arranging N numbered cars into a row of N parking spots. For each arrangement, each parking spot can be assigned a number corresponding to a car (the “car ID”). It is not meaningful to say how much arrangements differ based on the difference in car ID numbers in corresponding parking spots. It is meaningful, however to say that two solutions differ in 3 parameters (or 6, or 19). Thus, we are using the Hamming distance as a measure of closeness between solution points in the search space. Consequently, a set of solution points may be considered local to each other, even though they may appear widely scattered when viewed in the search space using conventional Euclidean distance.

Euclidean locality refers to the closeness of solutions based on the Euclidean distance between points in the solution space. This is meaningful when the parameters correspond to physical quantities, which is more often the case in continuous parameter problems. One of the problems with this distance which is avoided by the Hamming distance is how to scale the different axial components before combining them, if they represent different kinds of physical quantities. In scaling, the contribution from each parameter should be commensurate with the importance of that parameter.

2.3 Simulated Annealing

Simulated annealing (SA) is a search space method that can be used for both continuous parameter problems and combinatorial optimization problems. It attempts to lower a solution's cost in a way that mimics the lowering of energy in a solid as it is annealed. In this analogy, the material state can achieve a global minimum in energy if the temperature is lowered slowly enough. This state corresponds to the uniform propagation of the most stable crystalline structure throughout the material. If cooled too quickly, the state is trapped in a local minimum, which corresponds to defects or disturbances captured in the crystal structure.

In SA optimization (Figure 2.3), the current state $P_{Current}$ is a point in the solution space. A new state is obtained by perturbing the current state by a random vector; this corresponds to the thermal agitation of the crystallizing material, which allows the molecules to jostle around and find a stable arrangement. The step size within the solution space is randomly distributed, but statistically larger for higher temperatures. The material is able to get out of local minima because the nonzero thermal energy allows it to temporarily assume states that increase its energy, with a probability that is limited by the temperature. In SA, this corresponds to accepting poorer solutions with some probability, thus allowing the solution space to be thoroughly explored if the temperature is lowered slowly. As the search converges to a (hopefully) global minimum, the temperature (and thermal energy) approaches absolute zero; this narrows the breadth of the search and makes it more "greedy" by accepting fewer inferior solutions.

Exploratory Search Vs. Fine-Tuning

Because the random step size and probability of accepting poorer solutions, SA may be viewed as a stochastic search. In the beginning, high temperatures effectively make it a global random search, while low temperatures at the end make it a local directed search. In between, it stochastically explores the neighbourhood of the current solution. The effectiveness of SA in finding a global minimum is highly sensitive to the rate at which the temperature is lowered. The thoroughness of exploration due to the diversity of stochastic search can be contrasted with greedy directed search; the requirement for both will be apparent in the following algorithms as well.

1. Initialize temperature $C_{CurrentTemp}$ to a large enough value that the search is basically random. It may be necessary to randomly sample a number of solutions to estimate the variance in cost.
2. Randomly select a point $P_{Current}$ in the search space and evaluate its cost $C_{Current}$.
3. Loop
4. Get a new P_{New} by perturbing $P_{Current}$ and evaluate its cost C_{New} . Perturbation scheme may depend on temperature e.g. such that likelihood of large perturbations are greater for higher temperature.
5. If $C_{New} < C_{Current}$, then P_{New} replaces $P_{Current}$;
otherwise, P_{New} replaces $P_{Current}$ with probability $\exp\left(\frac{C_{Current} - C_{New}}{C_{CurrentTemp}}\right)$.
6. Update $C_{CurrentTemp}$ according to cooling scheme.
7. Stop if an acceptable solution is found, if the maximum search time exceeded, or some metric shows the search to be stuck in a minimum for a long time.

Figure 2.3. Framework for simulated annealing algorithm.

2.4 Tabu Search

Tabu search is largely used for combinatorial problems. It performs a *Hamming local search* by point changes in the solution parameters. It exhibits limited greediness by choosing the best move that doesn't repeat a recent move, or recent solution. This prevents the search from cycling around. This behaviour is effected by maintaining a "tabu" list of recent moves and/or solutions.

The rules encapsulated by the tabu list can be more sophisticated. Therefore, the effectiveness of the search is sensitive to the length of the list, or alternatively, the lifetime of the list items. Being flexible, the contents of the list can specify problem-dependent details of the solution rather than merely tabu solutions. A tabu list item can also stipulate that new solutions retain certain attributes rather than avoid recently encountered attributes. The framework for a tabu search algorithm is given in Figure 2.4.

1. Randomly select a point $P_{Current}$ in the search space.
2. Initiate the tabu list by adding to it the features of interest. To avoid repetition, for example, this might be the change(s) which generated $P_{Current}$ (not applicable to 1st $P_{Current}$) e.g. the addition of a particular edge in a graph solution. It may also contain solution attributes that the next point must have, or must not have. Because of computation overhead, it often does not contain the whole of $P_{Current}$ itself.
3. Loop
4. Get a new point P_{New} by choosing the best of a subsample of neighbouring points. How the neighbourhood is defined is problem dependent e.g. neighbouring points may be generated by perturbing $P_{Current}$.
5. If P_{New} doesn't violate the tabu list, or if it does but also satisfies an "aspiration criteria", then do the following. (The aspiration criteria is a problem-dependent criteria that overrides the tabu list e.g. if the cost is much less than the best solution so far.)
6. P_{New} replaces $P_{Current}$
7. Update the tabu list and aspiration criteria.
8. Trim away old tabu list items to maintain the list length limit.
9. Stop if an acceptable solution is found, if the maximum search time exceeded, or some metric shows the search to be stuck in a minimum for a long time.

Figure 2.4. Framework for tabu search algorithm.

It is possible for the tabu search to encounter a local minimum which its list of rules cannot cope with i.e. it is trapped in the local minimum. The common way to handle this is to restart the search from a new point in the solution space, if no improvement in the search has been observed for a long time. In fact, it is a common approach to dealing with local minima for algorithms in general e.g. it is an integral part of applying an algorithm known as "min-cut", discussed in Section 5.1.4 (multiple initial points are typically tried in multiple applications of min-cut).

2.5 Genetic Algorithms

Genetic algorithms (GAs) are used for continuous-parameter and combinatorial problems. The method is modelled after the principles of evolution. Through a number of generations, genes representing favourable attributes are proliferated through a population of candidate solutions by being given greater mating opportunities to chromosomes that benefit from such genes.

In the case of GAs, the genes are the problem parameters, and the population is a set of solution points, or *chromosomes*, that sample the solution space. Their distribution is initially random. In a mimicry of evolution, mating occurs when parent solutions are selected from the population, with a bias toward good solutions; their genes are then combined to form potentially better offspring. The new solutions displace old solutions, thus shifting the distribution of sampling points toward regions with good gene values and increasing the average quality of the population. If the new solutions are *not* good, the bias toward selecting good parents ensures that the poor solutions have limited opportunity to create more sampling points in the region of bad solutions. Thus, more exploration effort is focused on promising regions, less is devoted to unpromising regions.

Figure 2.5 shows the basic framework of a genetic algorithm, which the following sections elaborate on.

1. Initialize random population of solutions (genomes).
2. Loop
3. Generate offspring (select parents and mix their attributes)
4. For diversity, generate mutants by perturbing some solutions.
5. Replace current population members with offspring and mutants using suitable scheme.
6. Stop when a valid solution is found or if the maximum search time is exceeded.

Figure 2.5. Framework for genetic algorithm.

Genotype Vs. Phenotype

Despite GA's flexible handling of problem parameters and constraints, the information contained in the genes are not always a direct encoding of the parameters in the domain of the optimization problem. A translation may be needed to get a candidate solution from the chromosome. In

the form of a chromosome, the solution is known as a *genotype*; in the problem domain, the solution is referred to as a *phenotype*. One might represent the problem parameters differently for the GA for the following reasons:

- ◆ To improve the chances that information about good attributes gets passed from parents to offspring intact (Section 2.5.6)
- ◆ To simplify crossover (Section 6.1.1.2)
- ◆ To reduce the proportion of infeasible solution points in the search space (Section 6.1.2).

2.5.1 Hyperplane Resampling

It is illuminating to consider exactly how the sampling points shift to new positions in the search space. Consider an optimization problem with three variables, x , y , and z . Figure 2.6 shows two parent solutions P_1 and P_2 combining to form a child solution. The child inherits x from P_1 , and (y, z) from P_2 . This is the most basic form of genetic *crossover*. Thus, the evolutionary redistribution of sampling points corresponds to putting more points at the intersection of hyperplanes in which good points reside. These hyperplanes are the neighbourhoods of Hamming locality of good solutions. As indicated earlier, the offspring are not necessarily close to the parents in a Euclidean sense.

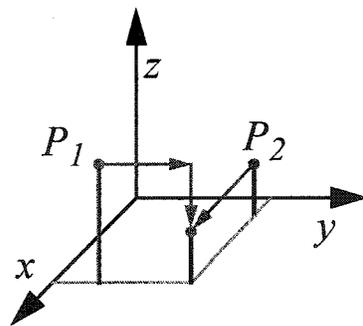


Figure 2.6. Simple crossover creates offspring in the hyperplanes of the parents.

2.5.2 Diversity

As in evolution, diversity is maintained in the GA population by retaining suboptimal solutions. This is akin to not wanting many copies of a few human beings, since there would not be enough diversity in genetic material to explore different make-ups. In GAs, the lack of diversity in genetic

material curtails the ability to thoroughly explore the solution space. This is generally not desirable, except for cases in which the solutions are converging to an optimum after a thorough search. Even then, it is not necessary for the whole population to converge so long as one solution achieves the optimum.

Selective Pressure and Mutation

Since the natural tendency is for sampling points to collect in promising regions, a fundamental issue in GAs is to avoid premature convergence before the design space is adequately explored. Besides controlling the bias with which good chromosomes are selected for mating, diversity can be maintained by constantly mutating a small number of solutions i.e. randomizing one of their genes. This has the effect of pushing a sample point along one axis in the search space for some random distance. Thus, it has a Hamming local scattering effect as a way to constantly look for new genetic material. It effectively provides a stochastic local search in the Hamming neighbourhood of the existing solution points.

Mutation is preferable to generating a random solution from scratch. The random global search has already been performed when the search space was initially seeded with random solutions. We already know that the existing sample points have congregated around the good regions (or hyperplanes) of the design space, so a purely random newcomer will likely represent a much poorer solution. It will have negligible opportunity to mate and pass on new genetic information. Instead, mutation explores new “twist” on an existing reasonable solution, one for which the genetic material doesn’t exist in the population. Because the gene value doesn’t exist, it cannot be acquired by the offspring, now or in the future, without mutation.

There can be various ways to incorporate mutation into the evolution. In nature, a creature’s genetic material can be modified over an individual’s lifetime simply by exposure to environmental factors. This random targeting of individuals is not biased toward fitter or less fit individuals. But the effect on breeding is most pronounced when offspring acquire a mutation i.e. mutation is coupled with crossover. Therefore, mutation in nature can be mimicked by either mutating randomly selected population members with uniform probability, or mutating offspring. Mutations in offspring can be induced by mutating the offspring directly, or mutating copies of the parent before mating.

Mutation Rate

The mutation rate is interpreted differently depending on whether offspring or general population members are mutated. If the offspring are being mutated, the mutation rate simply refers to the likelihood of subjecting the offspring to mutation. Whether each offspring is given the same opportunity to mutate is just an implementation detail [Whi94]. The mutation rate can just as easily refer to the likelihood of mutating one of the two offspring. This just translates into a factor of two difference in the average number of mutations per generation. Note that if copies of the parents are mutated prior to mating, both offspring inherit the mutation regardless of whether both parents are mutated.

If general population members are being mutated, we took the mutation rate as the percentage of the general population to be mutated. If this number is noninteger, the fractional portion refers to the likelihood that one more chromosome will be mutated.

For simplicity, we synonymously refer to the new solutions for the current generation as *kids* or *newcomers*, regardless of whether they are generated by mating or mutation.

Uniquification

In addition to controlled selectivity and mutation, diversity can also be enhanced by eliminating *duplication among kids*, or “twins”. Likewise, *duplication between kids and the general population* (“clones”) can also be eliminated. Detecting the presence of twins is computationally cheaper than detecting clones because the latter requires comparison of each kid with the entire population.

Diversity Versus Selective Pressure

Genetic diversity and selective pressure are essential for the timely convergence of the GA to a near-optimum solution (e.g. see introduction/background in [OW99,Urs02,HLP+02,HLV97,Whi89]). These diametrically opposite needs are often referred to as *exploration* and *exploitation*. Since they must be carefully balanced for an efficient yet thorough search, the ability to measure diversity can be very important. Not only does it help track down the source of premature convergence, it is also useful for GA implementations that monitor diversity and adjust the degree of diversity-enhancing mechanisms.

The conceptual basis for quantifying diversity, however, is rather ill-defined. How does one quantify the amount of scatter in a population of solution points in N -space? Section 6.1.5 presents some challenges in doing this, surveys some current methods, and then develops some intuitive

ideas for measuring diversity, motivated partly by the uncertainty about the effectiveness of various diversity metrics tried.

2.5.3 Global Search and Parallelism

One of the distinguishing features of GAs is that it maintains a semblance of global search throughout the search. In other search space methods, a number of solution points may be tried in a sequential manner, each new point being based on the last point (or last few points). In GAs, the entire population of solutions is available for selection as parents, so new solutions are not restricted to be in a single neighbourhood. Depending on the strategy for offspring to displace population members, the simultaneous availability of all solution points for mating can also be exploited to parallelize the algorithm.

GA's inherent hyperplane resampling is also touted as aiding in global search and parallelism. This is illustrated in a limited way in Figure 2.6, where a single offspring samples 2 hyperplanes. In fact, the offspring occupies many hyperplanes of different dimensionalities. On average, though, the hyperplanes containing good solutions will receive more sampling points with time. This relies on the effect of fitness on the *distribution* of chromosomes among the hyperplanes in successive generations; therefore, it is a statistical effect that can only be realized by search methods that work with entire populations of solutions.

Implicit Parallelism

The fact that one offspring simultaneously resamples a great number of hyperplanes is known as *implicit parallelism*. It helps contend with the combinatorial explosion of solutions in typical combinatorial problems as the problem size increases. Empirical studies indicate that the order of complexity to find a “good” solution is much less than exponential, though of course, it is not guaranteed to be optimal. In a landmark theorem known as the *Schema Theorem* [Hol75], implicit parallelism was intuitively analyzed for the rudimentary case of binary genes and simple schemes for crossover and population.

One should beware of interpreting GA theory literally. The theory mentioned above gives some intuition on some of the mechanisms at work in GAs, but it is based on the most basic form of GA. Furthermore, it only applies at the start of the GA algorithm, when the search space is randomly seeded [Whi94]. Despite its intuitive usefulness, its analytical applicability has been widely con-

tested, and alternative models have emerged. In practice, GAs are often viewed heuristically and tested empirically.

2.5.4 Selection of Parents for Mating

In order for parental selection to favour good chromosomes, we need a sensible way to assign a figure of merit to each chromosome, referred to as its *fitness*. The selection scheme then favours a chromosome with a probability that is proportional to its fitness. For constraint driven problems, it is more natural to tally up a *cost* for a chromosome, based on constraint violations, then convert this to fitness (Sections 6.1.2-6.1.4). However the fitness is determined, the simplest way to perform the selection is to create a fitness pie chart out for all the chromosomes in the population, then pretend the pie chart is a roulette wheel and spin the dial. This is shown in Figure 2.7(a) for a population of 6. A simple way to implement this is to map the wheel to the real range [0.0,1.0] and generate a random number within that range.

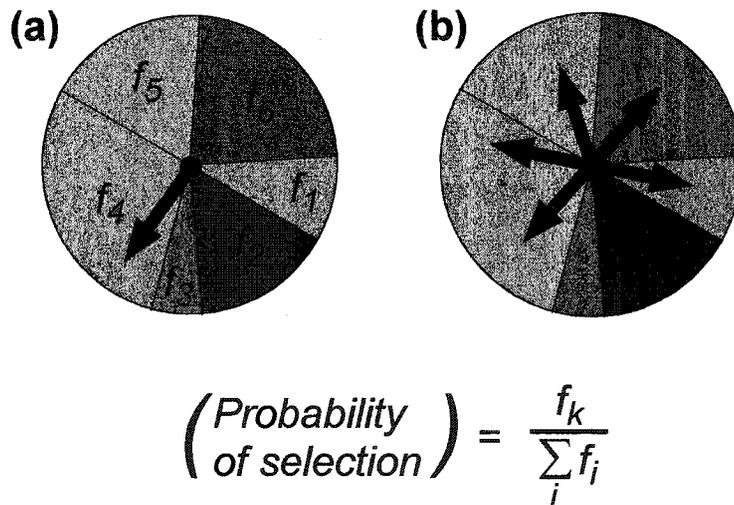


Figure 2.7. Roulette wheel selection of parents for mating. The population contains 6 chromosomes with fitnesses f_1 through f_6 . (a) Single-arm. (b) A multi-arm roulette wheel.

For evolution schemes in which N chromosomes are to be replaced at once, a single-arm roulette wheel must be spun N times to select N parents, which can then be paired off to mate. Alternatively, an N -arm roulette wheel can be spun once in a method known as “stochastic universal

sampling". Figure 2.7(b) shows this for the case where $N=6$ is the same as the population size i.e. the entire population is replaced by a new population of offspring. This can be implemented by adding a common random number in the range $[0, 1/N]$ to the set of numbers $\{0, 1/N, 2/N, \dots, (N-1)/N\}$. The N -arm roulette wheel gives the tightest variance in the number of times a chromosome is selected. For example, it is possible for the 1-arm roulette wheel to select f_i six times, in complete disproportion to its fitness. In contrast, the equidistant spacing of the arms in a 6-arm roulette wheel ensures that f_i is never selected more than once. Thus the distribution of selected parents more closely reflects their relative fitnesses.

Other standard schemes exist for selecting parents, such as pairwise competition between candidate parents (see "Ranked Selection and Tournament Selection" on page 130). For simplicity, we start with the simple roulette wheel schemes.

2.5.5 Crossover

Problem-specific methods abound to swap the values of corresponding genes between parents, thus creating offspring. The basic crossover schemes are 1-point and 2-point crossover. In 1-point crossover, values of genes to the left of some random position in the chromosome are exchanged between parents, thus yielding two children (the same two children are created if the gene values to the right of the random position are swapped). We distinguish between a gene and its value; a gene merely identifies the problem parameter, whereas a value is a numerical value taken on by the gene. A gene that has been assigned a particular value is called an *allele*.

Figure 2.8 shows a 2-point crossover. It is clear that 1-point crossover is a special case of 2-point crossover, with one of the crossover points at the left or right edge of the chromosome. If the chromosome is drawn as a ring rather than a straight chain, the 2-point crossover segment is just a randomly determined arc. For 1-point crossover, one end of the arc always is anchored to the same spot on the ring. This affects the transmission of useful information from the parents to children, as is now discussed.

2.5.6 Preservation of Building Blocks

Crossover preserves segments of genes from the parent chromosomes and passes them to the children. The hope is that the right combination of good gene values will generate better solutions. It is recognized that favourable attributes of a solution are often due to combinations of several

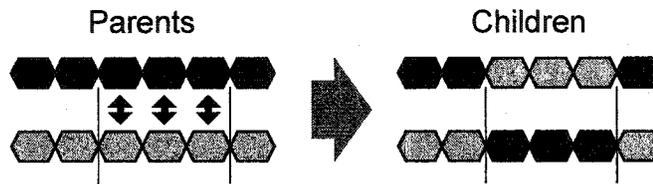


Figure 2.8. Two-point crossover. Crossover points are randomly determined.

values simultaneously taken by several genes, known as *coadapted alleles*. If these combinations contribute to exceptionally high quality in the solution, the nonlinear effect of interacting genes is called *epistasis*. When the beneficial elements of a solution come in clusters of genes, the genes are called *building blocks*. The tendency for building blocks to be propagated to the children intact (most likely by design) is known as *linkage*.

Since the simple crossovers preserve segments of adjacent genes, the chromosome should be defined so that the natural building blocks of a problem are close together. The more distributed or longer the building blocks, the less likely they will be passed to the children intact. This is because if a crossover point happens in the middle of a building block, it is *disrupted*. It is not always easy to keep crossover points out of the building blocks because building blocks are a vague intuitive groupings of genes. Building blocks may be nested, overlap, and have varying levels of subjective importance; the fact that certain genes form natural building blocks may not even be known. For the building blocks which have been recognized, gene proximity is the only way to preserve them. The performance of the GA becomes sensitive to the ordering of the genes.

Disruption in 1- and 2-Point Crossover

A comparison can now be made between 1- and 2-point crossover. The fact that one edge of the crossover region is permanently fixed in the 1-point crossover region is a liability, or at least an inflexibility in the implemented GA. Building blocks that cross that point in the chromosome will never be propagated. 1-point crossover is potentially beneficial if the problem can be coded such that building blocks never form across the fixed point, since it is one less crossover point that can disrupt other building blocks. However, it still prevents crossovers that would otherwise take place, thus limiting the ways in which good regions in the solution space can be sought. For this reason, a 2-point crossover was initially used.

The next two subsections discuss two simple ways of improving the preservation of building blocks through the use of nonbinary genes and a genetic re-ordering known as *inversion*.

Nonbinary Genes

Putting a greater amount of information into a nonbinary gene makes the information impervious to disruption, since crossover points only occur between genes. This nonbinary encoding can be as simple as using an integer for nonbinary problem parameters. Conceptually, there is no limit to amount of data that can be encoded into a single gene. Even with nonbinary genes, however, proximity can still be important to preserve the higher level multi-gene building blocks.

Arguments have been put forth against nonbinary genes (see [Whi94] as a start), citing the greater number of genes for binary encoding, thus exposing more hyperplanes for simultaneous resampling. To impart some intuition to this, consider an integer parameter x that is encoded as a series of N bits within a chromosome. The cluster of MSBs can be regarded as a building block. Now what happens when we view the search space with x as an axis (rather than the bits of x)? If a crossover point occurs within the N bits, it may be viewed as the MSBs staying fixed and the LSBs being mutated i.e. a local search in vicinity of the parents. If the LSBs are taken as fixed and the MSBs viewed as mutated, this becomes a distributed search among a sections of the x axis. This detailed exploration of new values along x would not be possible if x was not broken out into bits. The diversity of x values would decrease with evolution. This highlights the greater necessity of mutation in nonbinary genes.

Despite the case against nonbinary encoding, it has yielded better results when used in a natural way for nonbinary parameters ([Mic96] for starters). We believe that much of the premise of hyperplane resampling holds even for nonbinary genes, despite the restriction of crossover points to certain positions in the in the equivalent binary chromosome. This causes the statistics to change, but one cannot ignore the fact that hyperplanes of different dimensionalities are still being sampled when using nonbinary genes [Ant89]. However, the missing disruption for generating gene new values in the mating process makes mutation more critical in maintaining the role of exploration of new regions in the design space.

Inversion

Inversion refers to randomly exploring different gene orders by reversing the gene order within a randomly determined segment of the chromosome. It is basically a mutation in the gene ordering rather than a gene value. The motivation is that some gene orderings will respect some building blocks, while others will scatter them far apart in the chromosome. If reordering is allowed, then the gene for a particular problem parameter can no longer be identified by its position in the chro-

mosome i.e. the gene's *locus*. A scheme is required to associate each gene with its problem parameter, irrespective of position e.g. tagging each gene with an index number. Moreover, a scheme is needed to resolve the different gene orderings of mating parents. For example, a mating that yields 2 offspring could be performed under each parent's gene ordering, possibly with the same crossover points, thus yielding 4 offspring. The combined process of preferential selection and 2-point crossover then favours good gene values *and* good gene order.

2.5.7 Population Replacement Scheme

The question of how to replace existing population members with newly generated solutions can be posed at two levels: (1) What proportion of the population to change at once, and (2) Which individuals in the population to replace with kids.

Proportion of Population to Change at Once

Assuming that simple mating is performed, resulting in 2 offsprings per pair of parents, the question at the high level is what proportion of the population to replace at once. There are two extremes, *generational* replacement and *steady-state* replacement. In the *generational* scheme, N selections are performed on a population size of N to get N parents (with repetition). The parents are paired off to generate N offspring, which then replace the original population in its entirety. In *steady-state* replacement, one pair of parents are selected, the offspring displace the least fit population members, and the cycle is repeated. Generational replacement can be viewed as less greedy than steady state because the good solutions from the original population are lost.

The combined effect of mutation and steady-state replacement can lead to some pathological situations. If general population members are mutated, a mutation rate can easily lead to more than two mutants per generation. That is, the kids can consist almost entirely of mutants rather than offspring.

Generational and steady-state replacement can be generalized by borrowing concepts from *evolution strategy* [SR95,HB91]. A prespecified fraction of the population may be replaced in each "generation". In this generalized view of a generation, steady-state replacement performs one mating per "generation". Note that *evolution strategy* carries with it many other specifications of combining and selecting population members which sets it apart from GAs; for the GA developed in this work, we base the policies on intuitive understanding of their effects on exploitation versus exploration.

Which Population Members to Replace

The following are examples of how to select which population members to replace.

- ◆ Offspring can replace their parents. This yields a very ungreedy search, since the lost parents are likely to be good population members.
- ◆ Kids can be merged with the population, and the N best retained (N being the population size). This is very greedy, since the diversity of new kids may not be permitted to enter the population due to inadequate fitness.
- ◆ As in the steady-state injection scheme, kids unconditionally displace the population's worst members. Kids with poor fitness can still make it into the population to contribute genetic material even if they are worse than the population's worst members.
- ◆ Diversity can also be enhanced by replacing randomly selected population members. The degree of diversity can be controlled by selecting these "skid row" candidates according to their fitness.

If direct replacement of parents is used, it may still be necessary to the other replacement schemes in cases where more than 2 kids are generated per mating. For example, mating can be performed under each of the parents' gene orderings, if they are permitted to have different orderings for the purpose of inversion; this doubles the number of offspring. Extra kids can also be generated by mutation, if mutation is performed on parents and/or offspring during mating.

Note that all the replacement schemes can be used if mutations are applied to randomly selected population members rather than parents/offspring. In this case, the "parent" would be the underlying chromosome that gives rise to the mutant.

In this work, the initial replacement strategy was to replace the entire population with kids in a single generation. We also tried a variation of steady-state in which the best of the kids and population were retained. To combat premature convergence with greater diversity, we later tried unconditionally displacing the poorest population members with kids; this ensured that the population was actually being impacted by the diversity of new kids. Outside of the initial attempt at generation replacement, we avoided direct replacement of parents by their offspring, since the parents are likely to be favourable members of the population. When general population members were mutated, we did not displace the source chromosomes with the mutants.

2.6 Take-Away Points

- ◆ Tabu search, simulated annealing, and GAs represent different schemes with which to iterate over candidate solutions.
- ◆ In GAs, selective pressure contributes to a greedy, directed search, while population diversity causes the search to be more exploratory. Both must be balanced to avoid unnecessarily searches and premature convergence.
- ◆ Diversity and nongreediness can be enhanced in GAs by mutation, uniquification, generational replacement, and unconditional acceptance of kids.

3. Existing Reconfigurable Platforms and Physical Design Algorithms

This chapter surveys some prominent reconfigurable computing arrays. We discuss how computational models and architectural features impact their scalability, usability for various applications, and the ease with which they can be designed for. Electronic design automation (EDA) efforts are also presented.

In addition to fine grain and coarse grain arrays, examples of interconnect infrastructure are presented in Appendix B.15.

The Chameleon coarse grain FPGA is the last array presented. It is presented in more detail because, due to its availability, it is used as a representative platform for the exploration of mapping and automation in the remainder of this thesis.

Chapter 4 is a nontrivial design using Chameleon. It is advisable to read it directly after reading the Chameleon description in this chapter.

The following is not a complete survey; other platforms worthy of investigation include REACT, Sonic, PAM (Programmable Active Memories) and PAMETTE, Splash and Splash-2 for historical perspective, Paddi, Remarc, and Chess. Brief details are presented in Appendix B.14.

3.1 Architecture-Dependent Factors

The architectural features can be categorized into 3 classes: (1) granularity, (2) reconfiguration speed, and (3) distributed control. Architectural details often drive the methodologies and heuristic algorithms for computer assisted design; hence, mapping and algorithms are covered as a fourth factor.

3.1.1 Granularity

Reconfigurable arrays can be roughly divided into fine grain logic and coarse grain logic.

Fine Grain Arrays

All fine grain reconfigurable arrays take the form of 2D arrays of logic blocks. The logic blocks can be multiple-output, with a number of logic or data forwarding pathways from input to output. One or more lookup tables (LUTs) invariably make up the core logic block, and many arrays have a fast carry chain along the rows or columns of logic blocks. For hardware caching, it is typical to design kernels and intra-kernel operators as row-wise macros of logic blocks, thus simplifying the positioning of various kernels into a 1D placement problem. Dataflow is downward, across the operators. Often, the bit slices of a datapath align with the array columns. Extra logic that enhances speed for word-oriented operations (e.g. fast carry chain) is not considered as costly as in the past, since interconnect is consuming a greater proportion of the area in modern designs.

Coarse Grain Datapath vs. Fine Grain Datapath

Coarse grain arrays differ from fine grain arrays in their high-level array elements. Since the datapath is already word-oriented, many coarse grain arrays do not have to be 2D. Instead, some arrays consist of a linear arrangement of word-oriented computing elements.

Since the coarse grain array elements are already at a high level of hierarchical abstraction (typically similar to ALUs), the operational paradigm can depart from that of using a fine grain array to accelerate complex opcodes in a processor instruction stream. For the latter, specialized *opcodes* often imply some low-level customization, which fine grain arrays are better suited for. For coarse grain platforms, this acceleration of special opcodes is mirrored at a higher level of abstraction; kernel tasks are dispatched at the level of jobs or function calls in a high-level language.

Coarse Grain Datapath, Fine Grain Control

One of the consequences of tailoring the computation resources to the word-based datapath of DSP is that it is less suitable for control logic, which often uses 1-bit boolean logic. It may be possible to implement some of the required state machines and boolean logic in the datapath; however, it would not be the most efficient use of those resources. In order to support circuits with complex behaviour, the platform architecture should have some provisions specifically for control. One might design DPUs that allow for fine grain use, but that defeats the purpose of coarse grain DPUs. It is more realistic to augment the coarse grain resources with fine grain logic in the form

of digital logic lookup tables (LUTs) or programmable logic arrays (PLAs). Each have their advantages. LUTs require a lot of interconnect to form wide-fanin logic, but are faster than PLAs; however, PLA delays are predictable and only have to be as fast as the DPUs. In either case, the device resources are split up between coarse grain DPUs and fine grain control logic.

The Specialization Trade-Off: Misalignment with Application

As an example of misalignment between platform and application, consider the division between datapath and control path logic. In contrast to fine grain FPGAs, designing circuit configurations for coarse grain platforms is complicated by the separation of resources for the datapath and control path. In a fine grain platform, a common pool of resources is used for both. Since the division is arbitrary, it can be driven by design simplicity. For a coarse grain device, the split is fixed by the device architecture, and does not necessarily match the requirements of a circuit design. The circuit implementation must be modified to accommodate the device architecture. This is merely a manifestation of the greater potential for misalignment between application and platform whenever the resources are more specialized. In this case the reconfigurable logic is specialized for datapath and control path.

As mentioned in the introduction, similar misalignment can occur by the mere fact of fixing a bit-width for the datapath. In contrast, a datapath can be of arbitrary width in an FPGA. Even there, however, there is the need to specialize to some degree (e.g. fast carry chains) to improve performance.

3.1.2 Reconfiguration Speed

Reconfiguration Speed and Flexibility in Kernel Scheduling

Reconfiguration speed is extremely important in order for a device to do more than just reconfigure on bootup and emulate an ASIC thereafter. Fast reconfiguration uses up less of the real-time budget and increases the freedom to download different circuits more frequently. In the process of mapping algorithms to programmable circuit configurations (or *kernels*), this leads to a greater number of ways in which an algorithm can be decomposed to either make the most efficient use of on-chip resources and/or meet real-time requirements. For example, fast reconfiguration makes it appealing to apply a number of successive kernels to a single segment of an input data stream rather than applying one configured operation to a number of segments of input data.

The first operational paradigm is “data-centric” because kernels are repeatedly being renewed from off-chip while data stays on chip after transformation by each successive kernel. In contrast, the “kernel-centric” paradigm fixes the on-chip kernel(s) to transform as much data from off-chip as possible before loading the next kernel for the next operations in the dataflow. These are two extremes in a range of operational models [MKF+01].

Run-Time Reconfiguration

Many modern devices are reconfigurable in-place with standard data voltages, without powering down the device or the host system. Within this regime, there are various degrees of reconfigurability, with different names: “dynamic reconfiguration”, “run-time reconfiguration”, and “on-the-fly reconfiguration”. We use these terms synonymously and simply point out differences in reconfiguration abilities. For example, classical Xilinx devices reconfigure slowly via serial port, while Chameleon clocks in reconfiguration bits at the same bandwidth as data.

Granularity Versus Reconfiguration Time

Since fine grain arrays allow more control over bit-wise resources, many more configuration bits are needed to specify a piece of functionality compared to a coarse grain array. This lengthens configuration time, and reduces the flexibility with which streamed data computations can be organized i.e. only long-running kernels are worthwhile running, over a large enough data set. Hence, data-centric operation is preferred for fine grain devices.

Partial Reconfigurability

If a kernel uses only a fraction of the array, reconfiguration time can be reduced if the device does not require that a full set of configuration bits be loaded at once. Additionally, some devices are segmented in a way that allows one region to reconfigure without interrupting a task that is running on a different region. This further mitigates the run-time cost of reconfiguration, since reconfiguration can be overlapped with execution.

Configuration Caching

By caching configurations on-chip, reconfiguration time can be amortized over many executions of the loaded kernel. This happens at two levels. On one hand, entire kernels can be cached. On the other hand, an array element may cache several operating modes, which the kernel toggles between as it executes. At both levels, the fewer configuration bits required for coarse grain con-

figuration makes it possible to cache more configurations than in fine grain arrays, for the same amount of configuration memory. Appendix B.1 describes further subtleties in configuration caching.

Another drawback in caching fine grain configuration bits is that the greater detailed control of fine grain resources places a heavy burden on the kernel designer and the mapping methodology to ensure that kernels to be cached are relocatable and that communications lines don't collide.

3.1.3 Distributed Control

Distributed control addresses scalability of interconnect both in terms of congestion and delay. By avoiding global signalling, it avoids congestion around the center of the design as more pieces have to talk to each other in a larger system. As well, local control signalling has less propagation delay.

Distributed control costs more logic, since there is duplication of functionality in distributed control logic. In a typical hierarchical design, however, the use of interconnect increases more quickly with system size than the use of logic (See Appendix B.2 for an illustrative rationale). Distributed control uses the resource for which there is a slowly increasing demand (logic) to mitigate the more quickly increasing demand for interconnect.

3.1.4 Mapping and Algorithms

Compiling and Resource Binding

The compiling and physical resource binding of a kernel depends heavily on the array architecture, including granularity, dimensionality of the array, operations supported by the array elements, interconnect network, memory interface, and computational model. For example, fine grain FPGAs with data words that flow downward across row-wise operators dictate a certain approach in technology mapping a dataflow diagram. Coarse grain arrays meant for systolic algorithms, such as RaPiD and Morphosys, are suited to highly vectorized operations for efficient mapping in space across the array elements. This may be more readily handled using vector-friendly language such as RaPiD-C and SA-C. Arrays not meant specifically for systolic operations may require dataflow diagrams to undergo architecture-specific technology mapping, followed by place and route.

3.2 Fine Grain Platforms

This section surveys the following arrays: DISC, Garp, and Chimaera. In DISC, the reconfigurable array is used to construct a host processor, as well as to cache kernels for handling complex opcodes. In Garp, a MIPS processor is integrated with a medium grain FPGA; methods are also developed to compile from C. Chimaera encompasses compiler development, kernel caching management, and strategies of multichip design.

3.2.1 DISC: Dynamic Instruction Set Computer (1995-1996)

Appendix B.3 describes a commercial FPGA tightly coupled to a processor. The FPGA is used as a hardware cache for acceleration of specialized opcodes. Due to conceptual similarities with the following Garp array, we merely point out similarities and differences with the hardware architecture here. No such differences are made regarding EDA, since the mapping for DISC appeared to be manual.

3.2.2 Garp: Gate Array Processor (1997-2000)

Garp was developed by Berkeley Reconfigurable Architectures, Systems, and Software (BRASS) group [CHW00][HW97][Gar97][CW98]. Garp is detailed in Appendix B.4, including low level mapping and compiler development.

In the Garp effort, the computationally specialized portions of an Ultrasparc 1/170 was replaced with a MIPS-II processor and a custom FPGA (Figure 3.1). The Ultrasparc is a 4-way superscalar processor. The portions displaced by the array were the superscalar integer unit and the floating point unit. The reduced MIPS core was extended with opcodes to configure the array, initiate kernel execution, and synchronize operations. The array was 24 columns by 32 rows (Figure 3.2(a)). The resulting hybrid made use of the UltraSPARC memory system.

3.2.2.1 Comparison of Garp with DISC

Garp's Similarities with DISC

- ◆ Row based kernels (Figure 3.2(b))

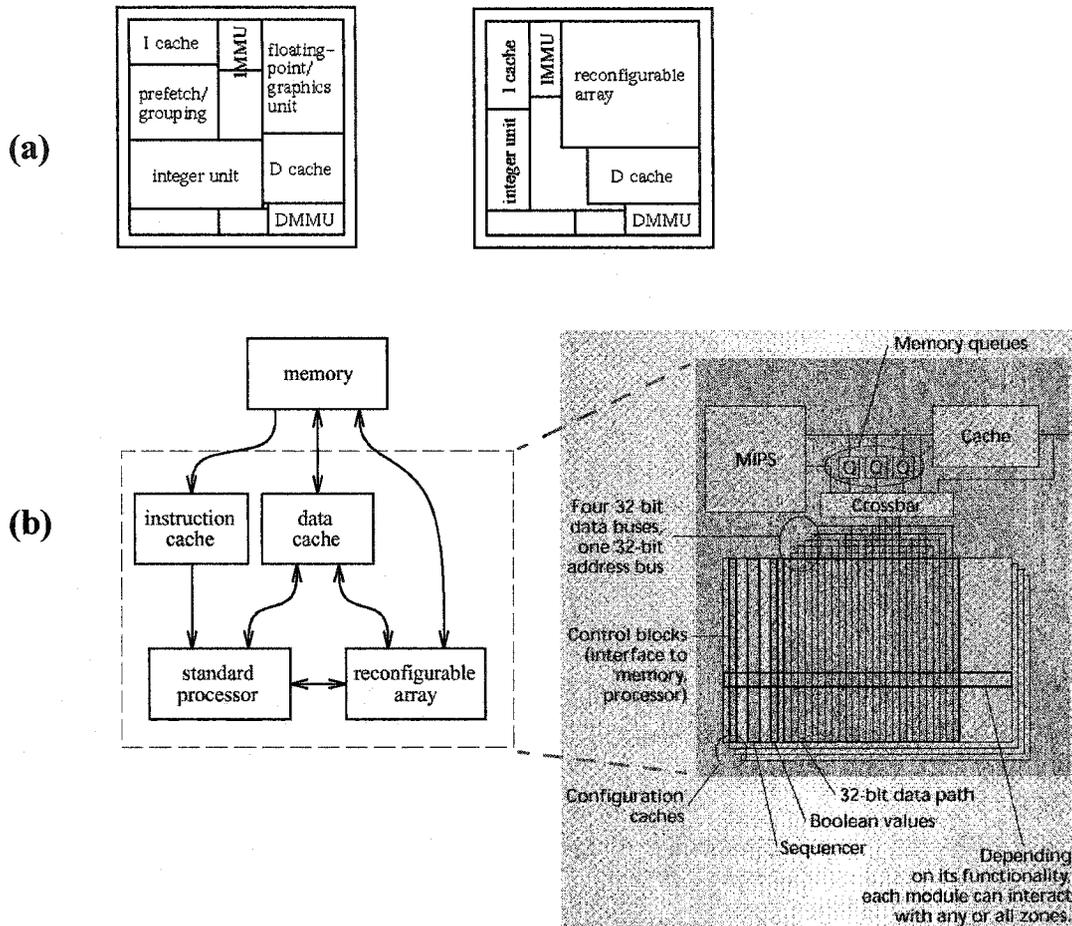


Figure 3.1. Garp architecture.

(a) Floorplan of UltraSparc (left) and Garp (right) [HW97].

(b) Garp internal architecture [CHW00].

◆ Vertical global buses (Figure 3.2(a))

The vertical buses in Garp are for data, configuration loading, and saving/restoring state information about the array.

◆ Processor controls the main instruction stream

◆ Data cache and main memory is accessible by the array

◆ The processor can be stalled until kernels execution finishes

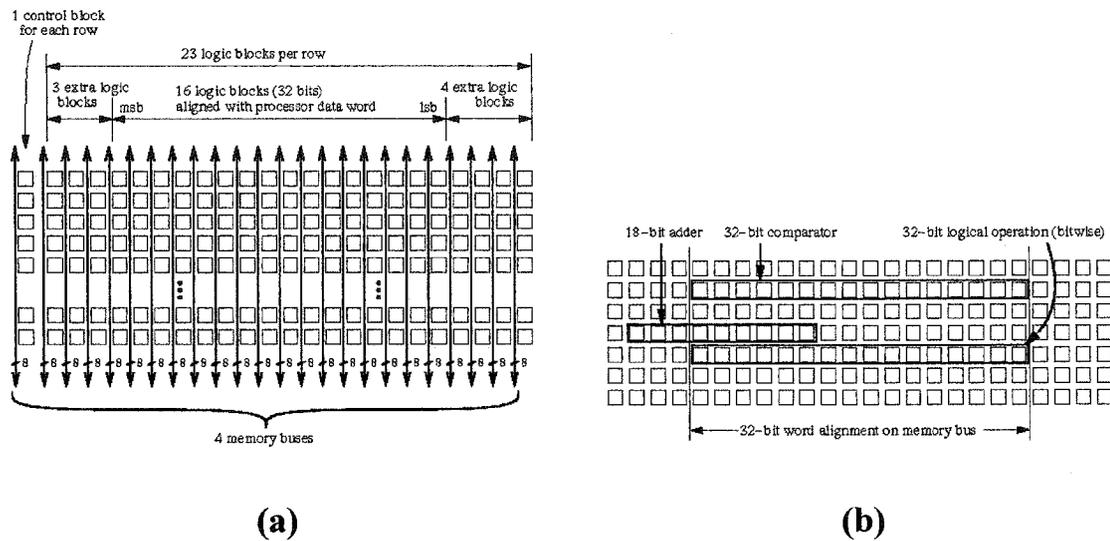


Figure 3.2. Garp array details from [HW97].

(a) Functional organization of columns. (b) Example macro placement.

Garp's Differences with DISC

- ◆ The MIPS processor is prebuilt rather than assembled from array resources.
- ◆ The granularity is 2-bits, which halves the number of configuration bits to load
- ◆ Limited partial reconfigurability. Portions of the array that are not configured during reconfiguration are cleared
- ◆ Entire array configurations are cached, rather than using different parts of the array to cache different kernels
- ◆ The custom design of the array allowed word-oriented specializations for fast arithmetic e.g. add, shift, multiply (Figure B.5).

3.2.2.2 Garp Interconnect

The interconnect within the array is shown in Figure 3.3. The wires run in the cartesian channels between rows and columns of logic blocks. Connections between wires are made via a special path within each logic block which bypasses most of the logic. The vertical wires are of various lengths, while the horizontal wires are either local or global. Local wires span 8 columns

(apparently not reflected in Figure 3.3), are skewed with unit stride, and can be driven from the center or the ends.

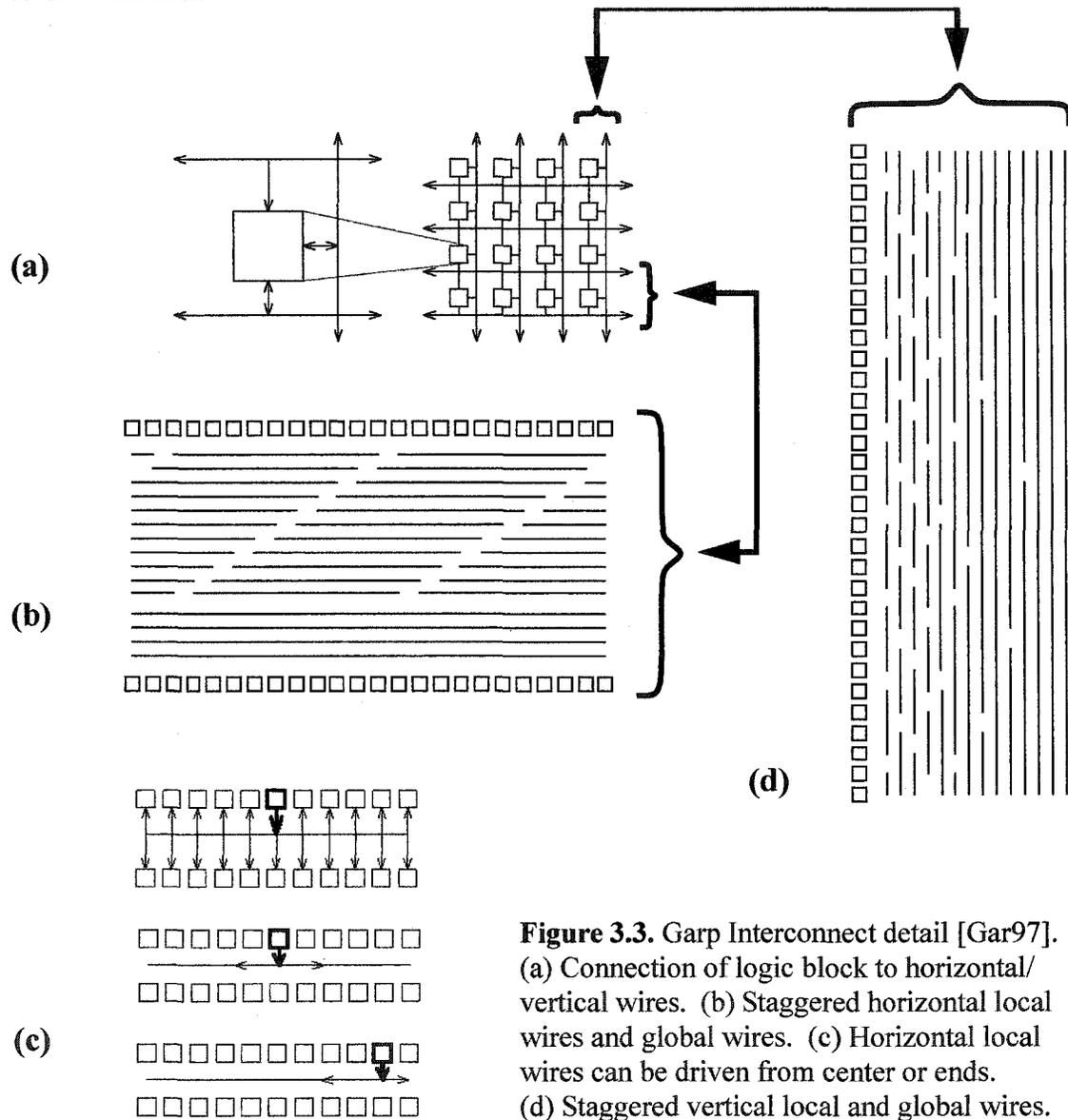


Figure 3.3. Garp Interconnect detail [Gar97].
 (a) Connection of logic block to horizontal/vertical wires. (b) Staggered horizontal local wires and global wires. (c) Horizontal local wires can be driven from center or ends. (d) Staggered vertical local and global wires.

3.2.2.3 Garp Technology Mapping and Placement

Appendix B.4.6 describes a dynamic programming approach for simultaneous technology mapping and placement. A dataflow graph (DFG) is clustered in a way such that each cluster has one output, which drives one destination. Starting from the DFG inputs, the clusters are matched with

lowest-cost library macros. Macros that make up the DFG occupy array rows, and are stacked vertically. The cost of each match takes into account the stacking order of the logic that feeds macro's inputs. Thus, placement and delays are also accounted for. In this work, there did not seem to be a shortage of vertical wires to support the interconnection of an arbitrary stacking of macros.

3.2.3 Chimaera (1997-present)

The Chimaera architecture from Northwestern University [HFHK97] consists of tightly coupled FPGA and microprocessor for hardware caching. This integration was intended to eliminate the a communications bottleneck between the two, and allow the acceleration of a broad class of functions as opcodes. Functions loaded into the FPGA by the host processor operate speculatively on every clock cycle, with the results written back to a register file only for those functions explicitly invoked by the processor.

Chimaera Array Architecture

The execution model is that of bit-slice data spread horizontally across the array, propagating downward as row-wise operations are applied. Kernels corresponding to opcodes take up entire contiguous rows. Hence, the array is partially reconfigurable by the row. Logic blocks have multiple LUTs and multiple inputs/outputs, allowing data forwarding at the same time as computation. Special carry-propagation logic propagates critical paths along each row. Horizontal wires of various reaches are available, but limited. In order to simplify context switching, there are no pipeline registers aside from the logic block's ability to read/write the register file of the host processor.

Compiling, Placement, and Routing for Chimaera

Appendix B.5 describes efforts in mapping algorithms. The placement exploits the fine grain nature of the array elements. Rows are packed in a manner similar to list scheduling. Heterogeneous constraints were not mentioned.

3.3 Coarse Grain Platforms

We present some initial efforts by DeHon, which laid down some of the fundamental ideas about configuration caching and the coupling of reconfigurable computing engines with microprocessors. His 2D grid of ALUs is described.

The 2D array architectures are still quite generic. We next present architectures that are more dedicated for pipelined processing of stream data. These include RaPiD, a linear array that emphasizes systolic communication; PipeRench, whose architecture accepts the mapping of virtual pipelines of any length; and Wormhole, which realizes high bandwidth configuration loading by prepending configuration bits to the data stream and having the DPUs set up a data processing path through the array.

In addition to pure reconfigurable platforms, we also present a more general-purpose SoC paradigm. Pleiades targets low power through voltage control, and includes studies of interconnect schemes and CAD.

The platform used as the primary example in this thesis is the Chameleon 2D array for telecom which is presented last.

3.3.1 DeHon (1994-1996): DPGA and MATRIX

DeHon was one of the earliest proponents of caching FPGA configurations on-chip to avoid the configuration bottleneck of dynamic reconfiguration [DeH94] (Figure 3.4). According to his thesis [DeH96], such a *dynamic programmable gate array* (DPGA) was fabricated, and its efficiency was supported by paper analysis.

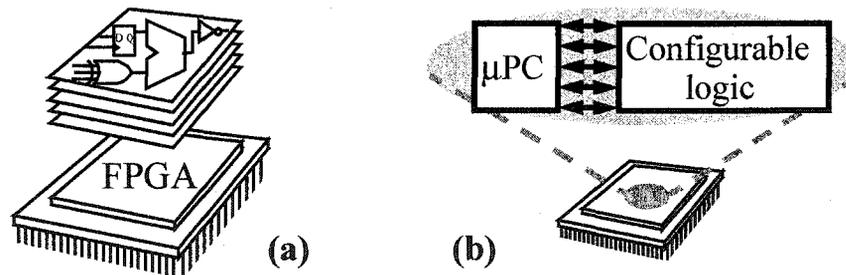


Figure 3.4. Preliminary ideas on reconfiguration from DeHon. (a) Caching multiple configurations to avoid configuration bottleneck. (b) Tightly coupling a reconfigurable array to a microprocessor to accelerate complex and evolvable opcodes.

DeHon also proposed an architecture in which a reconfigurable array resided on the same chip as a microprocessor [DeH95]. The array would be configured to accelerate complex operations

for user-defined opcodes. Examples of such tasks include adapting an on-chip bus to an off-chip bus, adapting to different off-chip memory configurations, and data conditioning tasks such as reversing the order of bytes, encoding/decoding, encryption/decryption, and [de]compression. For network processors, it could extract fields/headers from packets.

The coupling between the microprocessor and the array would be “tight” in the sense that the array would access the same registers. This allows the array to monitor opcodes, detect assertion violations and out-of-range addresses indices/addresses. It would also be able to perform running calculations in the background, such as accumulation, multiply-and-accumulate (MAC), and various statistics. If the tight coupling allowed the array to control the resources within the host processor, the array could perform code expansion and thus enable custom microcode.

The study culminated in a device known as MATRIX, an array of 6x6 8-bit computing cells known as *basic functional units* (BFUs, Figure 3.5)[MD96]. Each BFU contains 256x8 bits of memory, an ALU, a multiplier, and control logic that can depend on the data. The interconnect consists of local wires (Figure 3.5) and global wires. A FIR filter was paper mapped to the array in a number of ways to illustrate the array’s ability to operate as a systolic array, a microcoded processor, a VLIW processor, and a VLIW/multiple-SIMD processor (Appendix B.6).

3.3.2 RaPiD: Reconfigurable Pipelined Datapath (1996-Present)

University of Washington’s RaPiD architecture [ECFF96,CFF+96] is a reconfigurable linear array of computing resources optimized for DSP, i.e. easily pipelined and optimized for regular, repetitive computations requiring little control. It was intended to be closely coupled with a host processor, with access to its cache. Each cell in the array contains a 16-bit multiplier, ALUs, registers, and small local memories (Figure 3.6). Cells are connected by segmented buses with built-in pipeline registers and FIFOs at each end. Nearest-neighbour communications is favoured, thus allowing good mappings from systolic algorithms. This permits the leveraging of compiler algorithms for systolic arrays.

RaPiD’s Pipelining of Opcodes in Systolic Arrays

Most configuration is static, but it was 27% of the control signals can dynamically change from cycle to cycle. It was found that some control signals needed to change in tandem with data propagating through the array, so the dynamic control signals were pipelined to mirror the movement

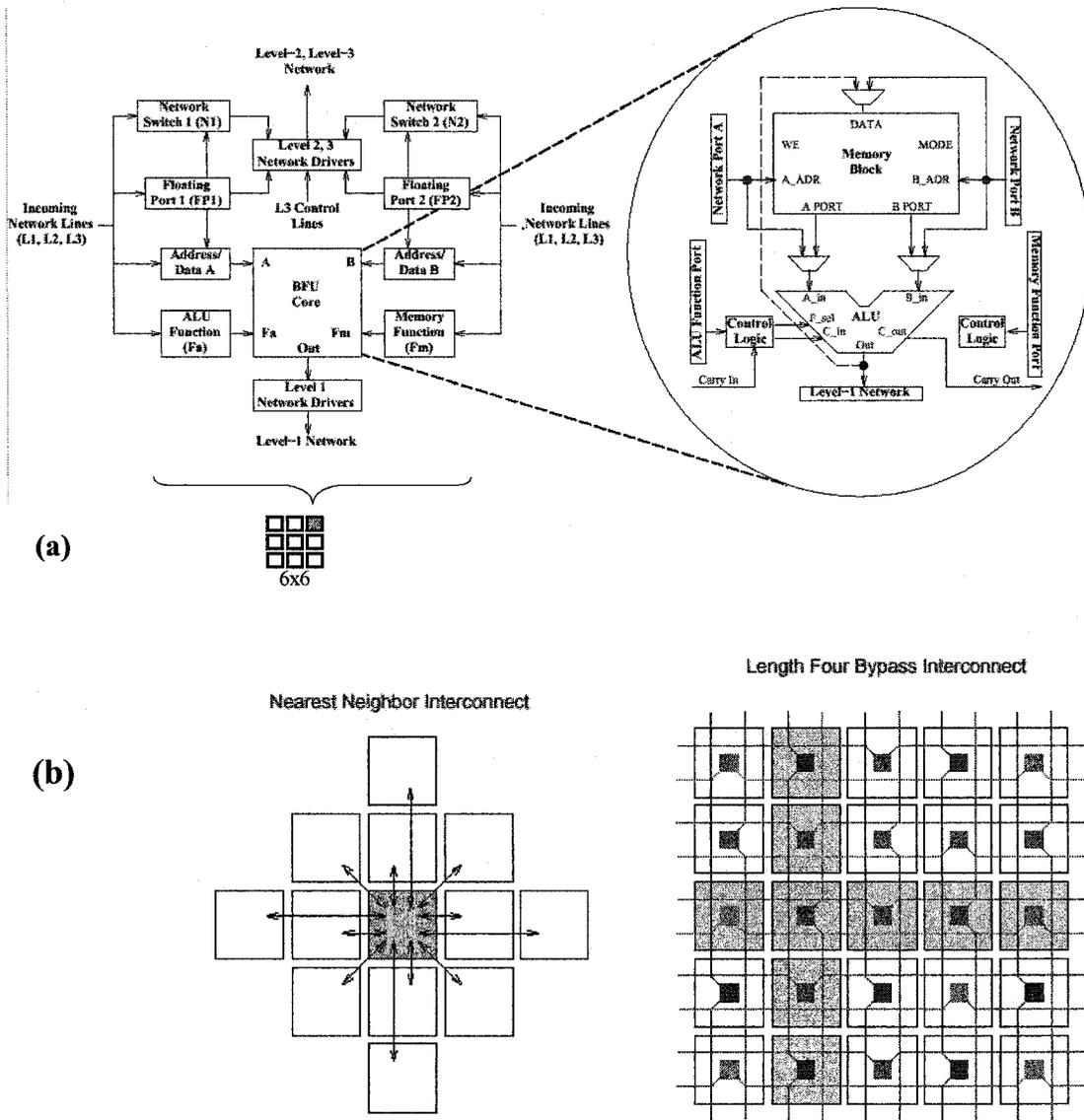


Figure 3.5. MATRIX [MD96].

(a) Functional unit (BFU).

(b) Local interconnect consists of nearest neighbour and 4-cell bypass.

of data through the device. LUTs were provided to decode the limited number of control signals to a more complex set of configuration bits. Registers within the LUTs facilitate the construction of simple finite state machines (FSMs) for nonpipelined operation.

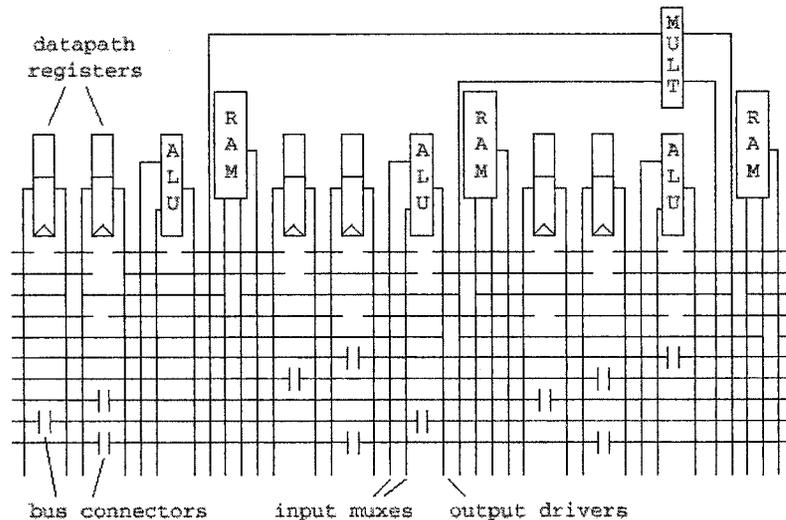


Figure 3.6. A RaPiD cell [ECFF96].

RaPiD Kernel Mapping

[ECF+97] presents mappings of FIR, 1D & 2D DCT, and motion estimation to the RaPiD architecture. It was acknowledged that automatic compilation from generic HDL would be very difficult for designs done without knowledge of the RaPiD architecture [ECF97]. [CFBE98] develops a variation of C to simplify the mapping of the iterations of the innermost loops across the pipeline stages of the array. Matrix multiplication was used as the guiding application. This highly vectorized application made it possible to infer a netlist for RaPiD from the time-to-space mapping.

More recent work [Sha01] automates the generation of a custom RaPiD architecture, optimized for a given set of applications. Simulated annealing is then used for the placement of compiled netlists, followed by routing using an academically popular algorithm called Pathfinder (Appendix B.7). Pathfinder models the switches between wires, and routes nets by iteratively applying the graph-based shortest path algorithm to all nets of a graph, continuously adjusting edge weights according to congestion.

RaPiD is also starting to incorporate recent routing approaches that perform retiming in order to accommodate interconnect networks that contain pipeline registers for long nets [Sha03].

3.3.3 Pleiades (1996-2000)

Pleiades is a vision for “infopad”, which was worked toward by Jan Rabaey’s group at Berkeley’s Wireless Research Center [Rab97]. As a platform, it targeted wireless multimedia on the mobile unit, in which the driving need was to perform many different computationally intense tasks with low power. Reconfigurability was seen as a way to achieve this by dynamically matching the hardware to the immediate computational need. It was recognized that most of the heavy computation was concentrated in a few inner loops of software. These were ideal candidates for the hardwired functions, with enough configurability to support similar operations across the entire application domain. Thus, the configurability is less than that of a fine or coarse grain array; it was referred to as “domain specific” configurability; algorithms for wireless multimedia was seen to have “intermediate specificity”.

Because the configurable modules were function-specific, the paradigm is based on an ASIC flow. However, it uses an architectural template consisting of a core processor linked by a reconfigurable communication network to satellite processors of varying degrees of specialization (Figure 3.7). Dynamically configurable fine grain programmable gate array (PGA) satellites could accelerate functions that did not warrant special-purpose satellites; these would obviously be less efficient than specialized circuits in the satellites. The satellites were parameterized to provide limited reconfigurability.

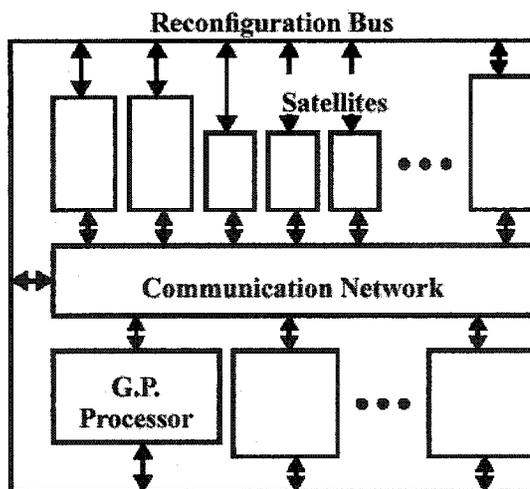


Figure 3.7. Pleiades architecture [Rab97].

Pleiades is detailed in Appendix B.8, including interconnect, power reduction, a prototype part, and vague EDA efforts. After looking at several interconnect topologies, a 2-level channel mesh

was adopted. There was strong interdependence between the targeted applications and the parameters for a good mesh. EDA consisted of a highly eclectic approach for mapping, but the detailed algorithms are not very clear.

3.3.4 PipeRench (1997-2002)

Carnegie Mellon's Reconfigurable Computer Project approaches the pipelined DSP platform with a coarse grain that is finer than most, recognizing that DSP for telecommunications often does not make full use of a 32-bit datapath. Their PipeRench device [GSB+00] consists of a fixed number of reconfigurable pipeline stages (Figure 3.8). Pipelines longer than the number of stages on the device are segmented to fit the device in a process referred to as *virtualization*. Data that would otherwise exit the device after flowing through the first segment of pipeline stages is wrapped around to the input as successive pipeline stages on the device are configured with functionality of successive pipeline stages in the second segment. This process continues until the data flows through all the segments of the entire virtual pipeline. The throughput is just the clock rate reduced by the fraction of the entire virtual pipeline that can fit onto the device at any one time. Virtualization is discussed more thoroughly in [Sch97,CWG+98].

PipeRench is detailed in Appendix B.9, including technology independence, computational model, compiling from C-like code, and exploration of physical array parameters.

PipeRench Technology Mapping, Placement, and Routing

The logic generated by the PipeRench compiler is submitted as a DFG to pattern matching to map it to predesigned parameterized modules that exploit architecture-specific routing and PE functions [CG99]. The algorithm for placing/routing the final logic emphasizes short compile times rather than exhaustive compaction, due to the liberty of virtualizing long pipelines onto the device. The DFG, mapped to library modules, is topologically sorted and scheduled across clock cycles (which correspond to FPGA stripes/rows) using list-based scheduling. Note the use of a full crossbar switch between stripes to facilitate this. The compiler development is elaborated on in [BG99].

PipeRench Similarities in Placement with Chimaera

Despite differences in granularity, both PipeRench and Chimaera rely on downward flowing data across operators. For both, placement seems to be patterned after the approach for scheduling

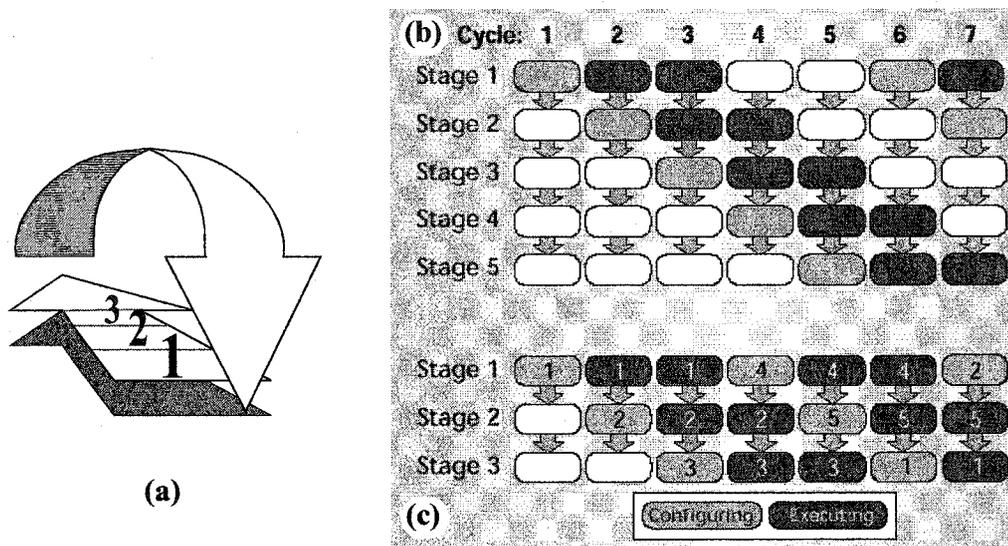


Figure 3.8. PipeRench operation [GSB+00].

(a) Conceptual depiction of a 5-stage logical pipeline mapped into a 3-stage physical pipeline by wrapping from the end to the beginning. (b) Desired 5-stage pipeline operation. (c) Resulting pipeline operation when mapped onto a device with 3 pipeline stages.

in high-level synthesis, with the role of successive clock cycles being replaced by the rows of the array. Hence, DFG vertices are packed into rows.

3.3.5 Wormhole (1997-1998)

Wormhole is an architectural concept that originates from Virginia Tech [BA97,BAM96,Bit97,He98,SRA+00], a part of the ongoing direction of Dr. Peter Athanas. Works [BA97,BAM96,Bit97] describe a Ph.D. design of the Colt 2D array of interconnected function units (IFUs, Figure 3.9) meant specifically to support stream-based computation. It is a preliminary attempt to explore wormhole architectural ideas for distributed configuration load, systolic-like local data transfer between datapath operators for speed, and fast local control circuitry instead of slow global control lines from a central location. The motivation for distributing configuration and control functionality into local autonomous circuits goes beyond high speed; the aim is to have the system scale well as the system gets bigger.

Appendix B.10 details the wormhole effort, including its paucity in control logic, use of decentralized logic for fast configuration loading via the datapath, and the allocation of array resources

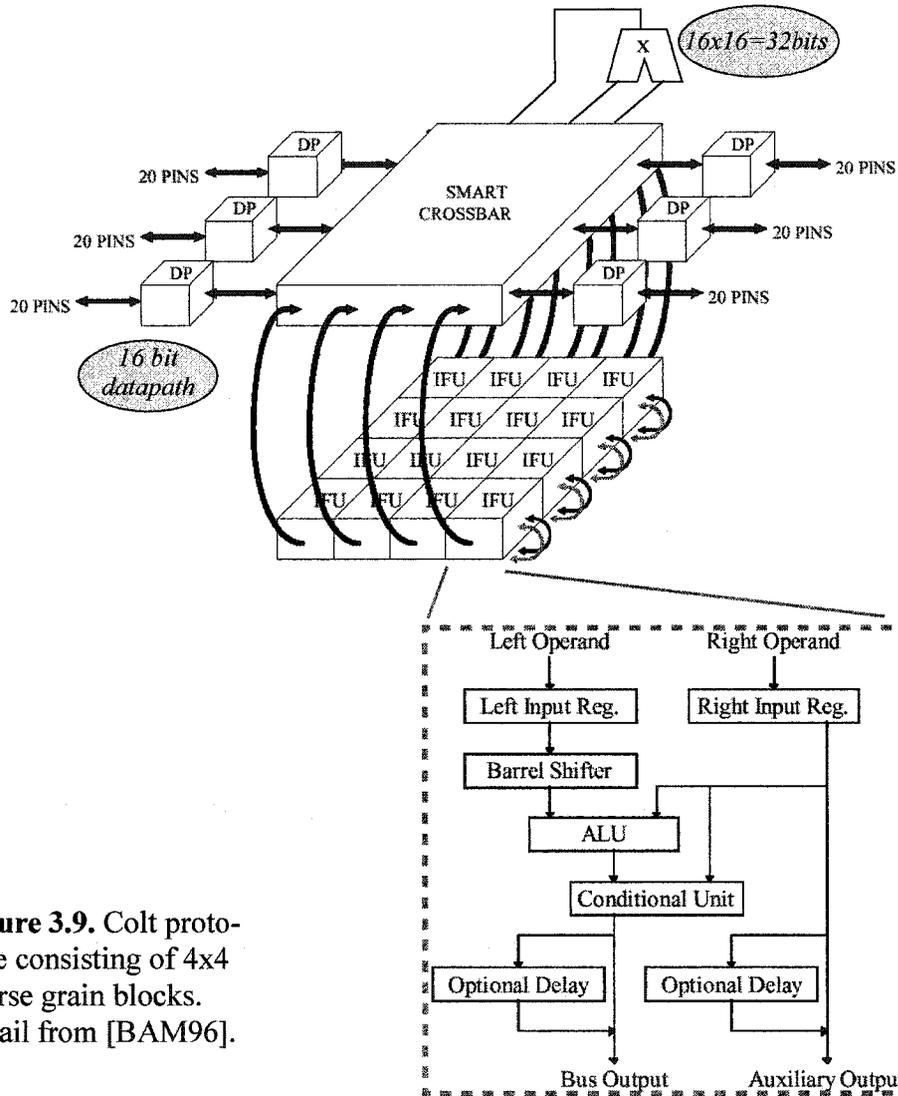


Figure 3.9. Colt prototype consisting of 4x4 coarse grain blocks. Detail from [BAM96].

to streams. Also described are experiences with 2 wormhole implementations, including paper mappings, stellar performance, and the eventual need for global control to synchronize the array with the incoming data.

Wormhole Interconnect Network

The crossbar switch connects all of the top and bottom most IFUs to a set of I/O ports and the multiplier, suggesting that datapaths are expected to flow somewhat vertically and use local neighbour interconnects in doing so. Each possible input/output pair has a switch point and a compact

state machine that monitors the input for any header information that specifies a connection with the output. The local neighbour connections along the left and right edges wrap around so that those borders are “locally” connected. A cartesian “skip” bus allows connections from the cross-bar to the edge of the IFU matrix to skip over peripheral IFUs to connect with interior IFUs.

Wormhole Configuration Loading

For speed and scalability, wormhole avoids a centrally controlled global system for loading configuration bits (Appendix B.10.2). Configuration information is prepended to a data stream and injected into the array. Each piece of array resource (e.g. I/O, switch, IFU) reads the appropriate configuration information and forwards the rest of the stream to the next piece of resource. This distributed handling of configuration sets up a pipeline in the array, with data flowing through the pipeline right after the configuration information.

3.3.6 Xputer (1990's)

Dr. Reiner Hartenstein's involvement with his Xputer architecture spans the 1990's (Appendix B.11). Xputer is not to be confused with the parallel computing concept of *transputers* [IC99]. The Xputer is premised on a 2D coarse grain array of ALUs for streamed data applications. The architecture is geared to, but apparently not limited to, systolic-like calculations. Control flow is assumed to be of such regularity that the only control information enters the array in the form of specially tagged data samples. The timing of array operations is not clear; there are indications of asynchronous data-driven behaviours, as well as later mention of high-level scheduling (HLS). FPGA and ASIC prototypes were designed. Mapping consists of technology-independent optimizations, followed by use of simulated annealing (SA) for placement and routing. The place-and-route description is extremely general.

3.3.7 MorphoSys (1999-present)

MorphoSys [SLL+00] is a 2D array of ALUS that operate in SIMD manner along one dimension. Systolic interconnection is emphasize, though nonneighbouring connections are possible. The architecture is motivated by regular, repetitive (i.e. vectorizable) 2D operations such as image processing.

While C++ and VHDL models exist to simulate the array, support of functional specification of kernels at a high level is developed in an independent effort [VNK+03]. This project develops a

compiler for a derivative of C, dubbed “SA-C” [HRB+99], which simplifies common coding structures for 2D image processing. Among other features, it supports high-level coding for sliding 2D window operations. In its scheduling, [VNK+03] accounts for bandwidth limits in MorphoSys’s interface to external memory (there are no local memories), and compiles SA-C to fully scheduled instructions on the MorphoSys array. As with RaPiD, the regularity of the operations allow an elegant mapping of operations to array elements, thus obviating the conventional place-and-route of circuit design.

Reference [LCD02] is another effort that targets MorphoSys-like architectures, with SIMD operation. Combinational expressions within loops are mapped to coarse grain array processors. The algorithm takes advantage of linearly ramped array accesses as the loop iterates. In the manner of a hard macro, it analyzes a computation expression and mixes transformation of expression trees with relative placement of recruited array elements to re-use data from memory reads, thus maximizing throughput in the face of limited memory bandwidth. Like [VNK+03], it is meant for a MorphoSys-like memory interface rather than locally distributed memories. In particular, it minimizes memory accesses for 2D coarse grain process arrays which interface to memory via global horizontal or vertical buses. Remarc and MorphoSys are mentioned as candidate targets.

3.3.8 Chameleon (2001+)

This is the representative architecture for the rest of the thesis.

Figure 3.10 shows the architecture of the Chameleon reconfigurable array [SC01,Cha,Cha01b], also known as a “fabric”, consisting of ALUs DPUs, local memories, and PLAs for the control path. Supporting the fabric is infrastructure for high-bandwidth transport of data and configuration bits between the fabric and the off-chip system, and a microprocessor to coordinate transactions. Chameleon’s commercially realized device, CS2112, is able to achieve 100MHz operation by parts binning.

DPU Hardware

The main computing engines in the fabric are 32-bit DPUs (Figure 3.10(a)). These DPUs are also capable of parallel 16-bit operations. The main computational block within the DPU is an ALU capable of two’s complement arithmetic and bitwise boolean operations. It can simultaneously monitor and flag a number of relational and arithmetic conditions. The ALU is fed by two operand paths with optional boolean masking, shifting, and registers for pipelining or storing con-

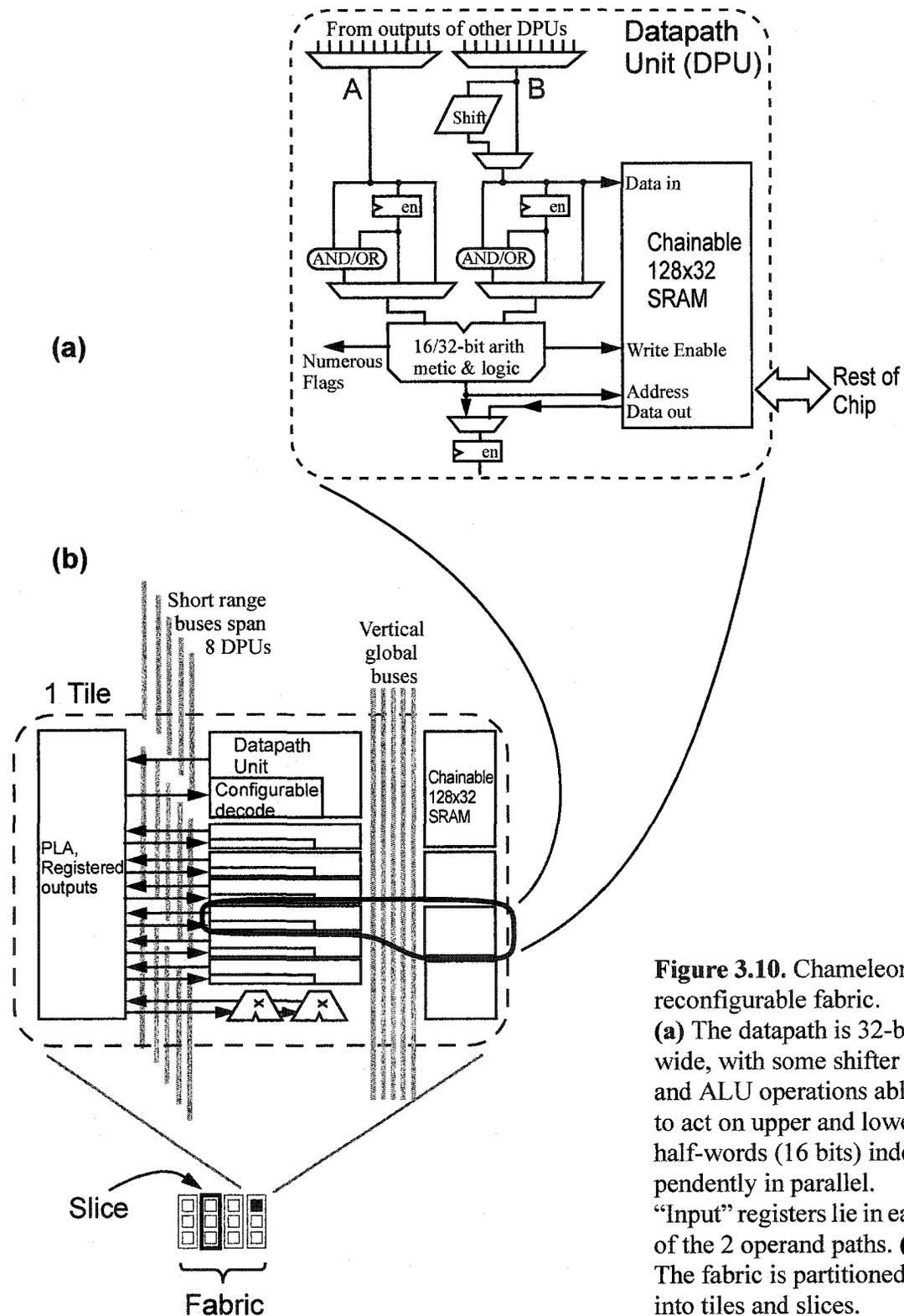


Figure 3.10. Chameleon reconfigurable fabric. **(a)** The datapath is 32-bits wide, with some shifter and ALU operations able to act on upper and lower half-words (16 bits) independently in parallel. “Input” registers lie in each of the 2 operand paths. **(b)** The fabric is partitioned into tiles and slices.

starts. Input muxes select the operands from buses driven by other DPUs. Data can be read/written to/from the adjacent SRAM; each SRAM memory has one read port and one write port on the fabric side which can be used by the executing kernel (Figure 3.10(a)). The SRAM memories can be chained together into a contiguous address space.

DPU Personalities

The designer specifies configuration information that controls the operating modes for all the components in the DPU. One set of such configuration bits make up a DPU “personality”. The DPU can cache up to 8 personality definitions. In each clock cycle, the adjacent PLA selects from the preloaded “personalities” (Figure 3.10(b)). A lookup table (LUT) in the DPU (“configuration decode” in the figure) translates the select lines into the individual control signals for the functional blocks in the DPU. Preloading of these personalities and programming of the PLA occur on a per-kernel bases. Figure 3.11 shows an example of constructing DPU personalities from the operating modes for the logic units of a DPU (Figure 3.10(a)).

Interconnect

Vertical datapath buses run along each slice. Local buses can source signals up 8 DPUs and down 7. Each tile can drive up to 3 vertical global buses spanning the whole slice. Horizontal global buses also cross all 4 slices.

Control Path

The control path is shown in Figure 3.12. It consists of a PLA with registered outputs and a wide input muxing plane. The PLA has 16 inputs, 32 product terms, and 32 outputs. There is one such PLA per slice, 3 per tile.

Some of the PLA’s registered outputs are fed back to the input as state-bits for building state-machines. Thus, the PLA acts as the control path. Each PLA is grouped with 7 DPUs and 2 multipliers to form a “tile”, of which there are 3 in a slice. Some control its feed back to all the tiles of a slice, while others only feed back to their own tiles. Hence, control is both global and distributed local.

```

//      BfyIPadrsMasker
//-----

defparam BfyIPadrsMasker.B_REG_INITIAL_VALUE = 32'b100 ;

CS2112_DPU BfyIPadrsMasker ( .clk(clk), .rst(rst),
    .a_in0 ( AdrsMask ) ,
    .b_in0 ( BfyCnt00 ) ,
    .dpu_output ( MaskedAdrs ) ,
    .csm_addr ( BfyIPadrsMasker_Sel )
);

`define BfyIPadrsMasker_Or 3'd0
defparam BfyIPadrsMasker.INSTRUCTION_0 =
    `A0_IN | `OPA_NO_REG |
    `B0_IN | `DUPL | `OPB_NO_REG |
    `ALU_OR | `LOAD_O_REG ;

`define BfyIPadrsMasker_And 3'd1
defparam BfyIPadrsMasker.INSTRUCTION_1 =
    `A0_IN | `OPA_NO_REG |
    `B0_IN | `DUPL | `OPB_NO_REG |
    `ALU_AND | `LOAD_O_REG ;

`define BfyIPadrsMasker_St1A 3'd2
defparam BfyIPadrsMasker.INSTRUCTION_2 =
    `B0_IN | `LSL | `SHFT_AMT_1 | `OPB_NO_REG |
    `ALU_PASSB | `LOAD_O_REG ;

`define BfyIPadrsMasker_St1B 3'd3
defparam BfyIPadrsMasker.INSTRUCTION_3 =
    `B0_IN | `LSL | `SHFT_AMT_1 | `OPB_OR_MASK |
    `ALU_PASSB | `LOAD_O_REG ;

```

Figure 3.11. Example of using Verilog mnemonics to define the DPU personalities and associated index values for a DPU named *BfyIPadrsMasker*.

The first *defparam* statement defines initializing value of the input register in the *B* arm. The *define* statements give friendly names to *INSTRUCTION_0* through *INSTRUCTION_1*. The associated *defparam* statements assemble the word for each personality. Each mnemonic in the word specifies an operating mode for a subcircuit within the DPU. Each mnemonic occupies a distinct field in the personality's word.

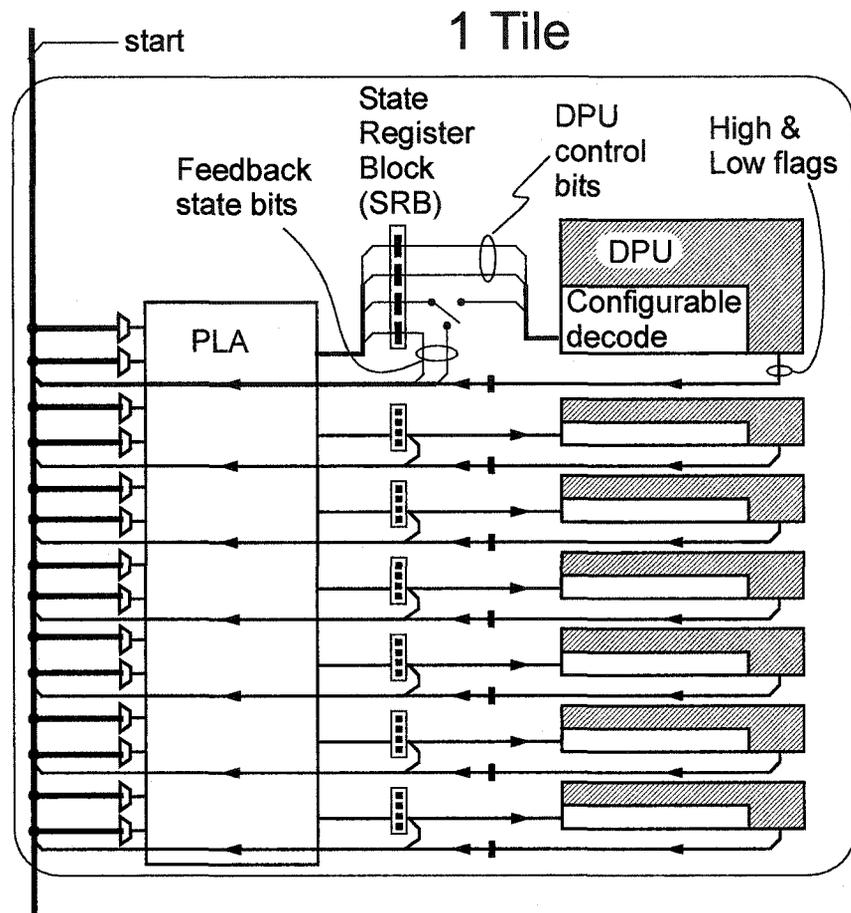


Figure 3.12. Control path view of tile: state bits consist of PLA output registers that are fed back to the PLA input.

3.4 Conclusions/Summary

3.4.1 General Observations

Flexibility Versus Efficiency

To varying degrees, devices can be more optimized for greater efficiency and performance over a narrower application domain. The trade-off is the greater inefficiencies for misaligned functions. ASICs are at the specialized end of the spectrum, while processors are the most flexible. FPGAs

are in between, and are good for the limited control complexity and speed requirements of stream data. Coarse grain FPGAs are further specialized for fast and efficient processing of word-oriented data.

Beyond coarse granularity, examples of specializing arrays for narrower application domains include the characterization of PipeRench performance over a range of architectural parameters, for a set of applications. CAD environments have been developed for this purpose for RaPiD and Xputer (Appendix B.11.2). Similarly, Pleiades was meant to be specialized for specific applications prior to fabrication, based on extensive simulations. Chameleon targets telecom applications.

Kernel Caching and Real Time Usage

Coarse grain arrays require fewer configuration bits, are faster to configure, and can cache more configurations. This broadens the options for scheduling kernels to minimize the overall usage of real time.

From the DISC and Garp projects, it is seen that fine grain kernel caching requires special attention to location independence in the design of the kernels, as well as in communications with the host processor.

Interconnect and Scalability

Scalability issues include both routability (or congestion) as well as speed of signal propagation over greater distances. In large designs, interconnect delay was a primary speed limitation, addressable by pipelining.

For the 2D arrays, the HSRA study (Appendix B.15.1) found that a hierarchical tree should be flatter and wider than a binary tree. This is supported to some degree by the Pleiades study of 2-level hierarchical mesh. The cluster points in moving from the lower mesh to the upper mesh is highlighted as sensitive to bottlenecks if not properly chosen.

Both Pleiades and aSOC (Appendix B.15.2) discounted the more global interconnect networks, such as bus-based and cross-connect networks, as not scalable.

Off-chip Bandwidth

Memory interface and buffering schemes were identified as important to keep the hardware from stalling, especially in architectures which do not contain local memories within the array e.g. Garp's memory queue. Such external memory schemes typically access an external memory (viewed as a single memory by the array) via a global bus in the array.

PipeRench demonstrates the feasibility of a data-centric scheme to alleviate data bandwidth. Data is kept on chip while lower bandwidth configurations are loaded to change the processing environment for different stages of an algorithm.

For Morphosys and Remarc, memory bandwidth is reduced by re-using loaded data that is utilized in more than one loop iteration of high-level sequential code.

Chameleon's local memory blocks create an on-chip workspace, thus avoiding the off-chip bandwidth limits.

Distributed Control

There are numerous examples of distributed control to alleviate the need for global control signals.

- ◆ Both the Wormhole and aSOC projects indicate that some kind of distributed control is required to prevent bottlenecks due to poor scaling of centralized control and its associated global signalling. The crossbars in both projects operate autonomously once programmed. Wormhole takes this further by having distributed intelligence in the IFUs to read off the configuration data in the stream header, and set up the pathway to the next IFU for the same thing. As mentioned earlier (Section 3.1.3), this requires the duplication of control logic throughout the array (or switch).
- ◆ Appendix C describes an illustrative hardware-to-software translation called Handel-C, which uses a token passing handshake between neighbouring logic. As opposed to using a centralized controller, this duplicates handshaking logic in all computing subcircuits.
- ◆ HSRA combats the increasing wire/logic ratio with system size by shifting delay-line logic from the source driver to the signal's various destinations, despite the duplication of delay-line resources.
- ◆ Half of Chameleon's control bits feed back to their own tiles rather than all tiles in the slice.

Central control and global signalling cannot be completely avoided. Wormhole's second Stallion chip had to resort to global signalling to properly handle pipeline stalls. The lack of such central control caused problems in the first realization.

Vectorizability and Mapping

Mapping for a number of coarse grain arrays rely on the vectorizability of array operations to simplify the mapping of scalar operations to DPUs e.g. RaPiD and Morphosys. This is implicit in the architectures that are overtly geared to pipelines for stream data e.g. PipeRench and Wormhole.

The vectorizability of algorithmic operations is implicitly assumed when mapping is applied to regular, inner-most loops of sequential, high-level code. For Morphosys, SA-C hides these assumed looping structures for 2D window operations in image processing. Special languages have been similarly developed for RaPiD and Xputer (Appendix B.11.2).

Resource Binding

Approaches to resource binding seem quite diverse and dependent on array architecture. Fine grain FPGAs come in 2D arrays while coarse grain arrays are both linear and 2D. For reconfigurability, fine grain operators are typically oriented row-wise and stacked vertically (or vice-versa).

- ◆ Fine grain arrays Garp and Chimaera have row-wise operators and heuristically explore a subspace of all possible placement solutions. Chimaera exploits fine granularity by packing operators into rows.
- ◆ DISC orientates cached kernels row-wise. Intrakernel relative placement seemed to be manual. Kernels loaded on demand into predefined placement sites selected at run time.
- ◆ RaPiD uses SA for placement and Pathfinder (iterative shortest path) for routing
- ◆ PipeRench uses list-scheduling to pack operators into rows of DPUs. A full cross-bar between rows seems to trivialize routing.
- ◆ Xputer uses SA for placement and routing. The description is vague.
- ◆ MATRIX, Wormhole, and Chameleon are placed and routed manually.

3.4.2 Take-Away Points

- ◆ The limited flexibility and high performance of coarse grain arrays are suitable for stream data.
- ◆ Coarse grain arrays can have fast reconfiguration, which open up options for scheduling kernels to save overall real time.

-
- ◆ Mapping to coarse grain arrays is significantly simplified for vectorizable algorithms.
 - ◆ Scalability can be facilitated by a hierarchical interconnect network and distributed control.
 - ◆ Data bandwidth can be alleviated by (1) sophisticated queuing of memory transactions, (2) data-centric computation, (3) clever re-use of loaded data where applicable, and (4) including local memories in the array.
 - ◆ Physical binding methodologies are extremely eclectic and platform dependent. A generalizable approach should be able to easily incorporate diverse architectural features.

4. Kernel Design Case

In this chapter, bottlenecks in the kernel design process are identified as a motivation for the development of CAD algorithms. This is done by manually mapping an example application, the FFT, to a coarse grain reconfigurable platform. Due to its availability, the platform used for these designs is the Chameleon CS2112; however, we also identify the features of the CS2112 that represent generalizable architectural concepts. The goals of this chapter are three-fold:

- ◆ Provide a view of the mapping process for a reconfigurable array
- ◆ Establish design practices that are both resource efficient and ease the complexity of mapping
- ◆ Identify difficulties which can be addressed by CAD algorithms

This exploration and developmental process leads to the formulation of the place and route problem for the coarse grain array in chapter 5, as well as a comparison with GAs for more general placement and routing.

Throughout this effort, we attempt to generalize findings beyond the FFT. Differences from designing for ASIC/FPGA are highlighted where they occur. A basic familiarity with digital design principles and methodology is assumed.

Design Case Mapping Tasks

The following tasks are elaborated upon in section 4.3 (“Kernel Design Flow” on page 60).

- ◆ Identifying kernel subcircuits to map to DPUs
- ◆ Packing functionality into DPUs to conserve resources. This includes piggy-backing control functionality onto already recruited DPUs where possible.
- ◆ Assembling DPU configurations and their time sequence in order to realize the required functionality
- ◆ Delineating control functionality into resource-efficient state machines. They must work together to invoke the various DPU configurations at the right times
- ◆ Physically arranging the DPUs so that the required interconnections are realizable
- ◆ Physically locating the control logic to maximize resource sharing and avoid the resource limits of each subsection (each tile’s PLA)

In both the datapath and control path, it is not enough to answer the question of whether there are enough resources. CAD algorithm development will also be driven by how easy it is to map to the resources, both manually and automatically. This design case provides an idea of the degree to which the design process can be systematized with efficient practices. The rest of the thesis gives an idea of the degree to which the physical design phase lends itself to automation.

4.1 The FFT Kernel

The FFT (Figure 4.1) is a key part of OFDM [Bur01], and can also reduce the complexity of very large FIR filtering by realizing the time-domain convolution as frequency-domain multiplication [JGP96]. For this design case, the FFT is a 1024-point FFT, arbitrarily chosen to be decimation-in-frequency (DIF) with input-normal ordering [OS89]. Chameleon has internally developed a decimation-in-time (DIT) FFT kernel [Cha00] from which some design parameters were taken.

The 1024-point FFT was chosen because the design pushes the limits of complexity that must be managed in the kernel design process. The dataflow diagram is far larger than can be mapped to a static pipeline; extensive time sequencing of operations is required to reuse device resources for various operations. This not only exercises the reconfiguration of the DPUs, but also the ability of the control path to realize such sequencing. Since the sequence of operations changes at various stages in the FFT, even greater demands are placed on the ability of the control path to handle complex behaviour.

Limiting the kernel to 1 slice of the array (Figure 3.10(b)) motivates design practices for efficient packing of the datapath. There is exactly enough memory to hold the 1024 data points and the 512 twiddle factors (Figure 4.1(b)), so long as in-place processing [JGP96] is observed.

For an upper bound on throughput, consider the FFT's basic computation, the butterfly (Figure 4.1). A 1024-point FFT requires 512 butterflies per stage over 10 stages (5120 butterflies). With 6 multipliers/slice and 4 real multiplies per complex multiplication, each slice can process 1 butterfly/cycle or 51.2 μ s/FFT at a clock rate of 100MHz. With all 4 slices running, the average kernel execution time for the device is 12.8 μ s/FFT.

Interested readers are referred to [ZB00] for performance comparison of the FFT in various technologies. In this investigation, the throughput per area of the Chameleon platform was several times that of FPGAs and DSP processors for large data sets. However, it was many times less than an ASIC that was architecturally tailored for a specific functionality and subjected to low-level

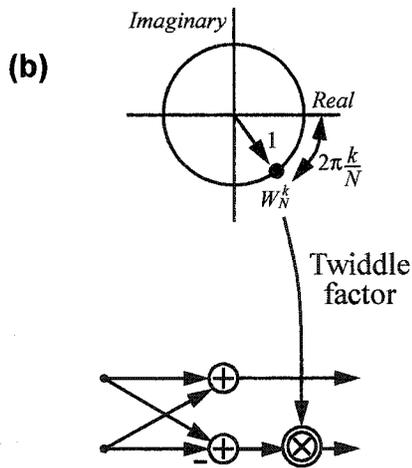
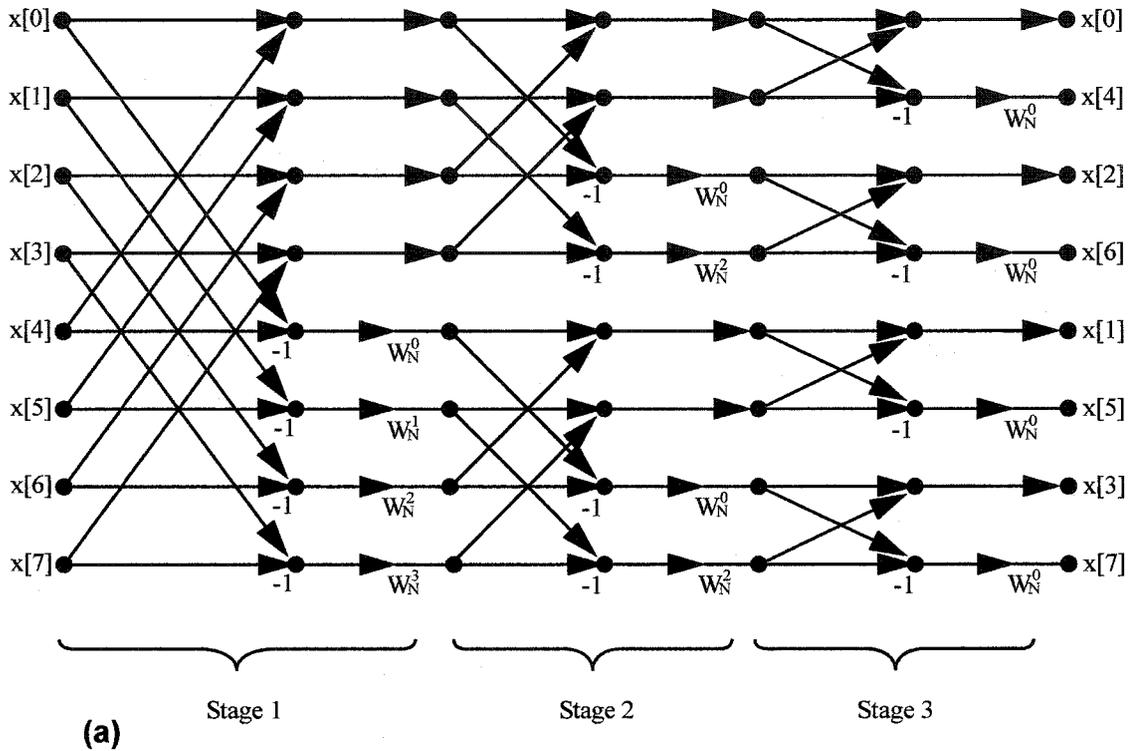


Figure 4.1. A small, illustrative 8-point FFT ($N=8$).
(a) Repeated butterfly, decimation-in-frequency FFT with input-normal ordering. **(b)** Butterfly detail.

device and circuit optimizations. This is an expected trade-off for the more general purpose nature of the reconfigurable platform, and the freedom from a host of nonrecoverable engineering costs/delays. It is possible that this disparity can be lessened by applying a similar degree of device and circuit level optimization in designing a reconfigurable platform.

4.2 Design Case Generalizability

This section describes the generalizable features of this design case. Section 4.2.1 presents the generalizability of the Chameleon platform, while Section 4.2.2 presents the generalizable features of the FFT for mapping. These are summarized in Table 4.1.

Table 4.1: Generalizable features of design case

(Chameleon) Platform	(FFT) kernel
<ul style="list-style-type: none"> ◆ Hierarchical partitioning <ul style="list-style-type: none"> • Datapath vs. control • Slices and tiles • Global vs. local buses ◆ Distributed chainable memories ◆ All types of configurability <ul style="list-style-type: none"> • Fast partial configurability by slice • “Run-time” configurability • Cycle-by-cycle control of configurable DPU modes 	<ul style="list-style-type: none"> ◆ Deterministic nested looping <ul style="list-style-type: none"> • Butterflies and FFT stages ◆ Computationally intense ◆ Highly repetitive ◆ Algorithmic parameters change with outer loop <ul style="list-style-type: none"> ⇒ Control complexity ◆ Complex numbers

4.2.1 Chameleon Fabric as a Representative Case

Many features of the Chameleon fabric are representative of coarse grain reconfigurable logic in general, as presented in section 3.1.

All Types of Configurability

The Chameleon array has many of the different types of reconfigurability mentioned in Section 3.1.2. Configuration bits can be loaded into a subset of slices without interrupting kernels that are already running in those slices, thereby overlapping configuration loading time with kernel execution [TA00]. The loaded configurations can be activated without interrupting executing kernels in adjacent slices. Kernel execution can also overlap in time with data transport between external memory and the fabric SRAMs.

Specialization: Coarse Grain Datapath, Fine Grain Control Path

The fabric is divided assuming a very typical separation of the kernel design; logic is optimized for datapath and control path portions of the design, with the PLA being a very typical realization of programmable control. Furthermore, the PLA consists of global control signals broadcasted to all PLAs in the slice, as well as distributed locally control signals fed back only to their own PLAs.

Partitioning to Limit Interconnect and Control Path Complexity

The hierarchical breakdown of slices and tiles to limit complexity in realizing the fabric is typical of what can be expected in programmable platforms. For example, separation into slices prevents the explosion of required interconnect, as does the segregation of wires into local and global within slices; one does not expect excessive communications between partitions in a design, thus the natural segmentation of the partition.

The breakdown of PLAs into tiles is another reasonable assumption for reconfigurable platforms in order to limit the complexity of PLAs and prevent them from running slowly.

Distributed Chainable Memories

The distributed chunks of memory and their finite chainability is also a generalizable feature of programmable logic platforms. Data organization in accordance with prebuilt memories is a prevalent task in platform-based design, as is planning how to integrate them with the datapath to realize the desired functionality and throughput. Chameleon's fabric memory can be chained into 4 segments of 512 words.

4.2.2 Generalizable Features of the FFT Relevant to Mapping

Many aspects of the FFT are representative of DSP for stream data.

For the butterflies to be processed sequentially, there are two levels of nested loops, one to loop through the stages (1, 2, and 3 in the figure) and one to loop through the butterflies within each stage. These are in fact the generalizable features i.e. two-level nested loops are common in stream data processing, as well as repetitive application of a core operation such as the butterfly. Furthermore, the pervasive use of complex numbers mirrors baseband DSP for modems.

Many parameters change between stages, including the data retrieval pattern for the butterfly inputs, the butterfly height, and the twiddle factor retrieval pattern. Together with the nested loop-

ing, the complicated stage-dependent memory access makes this a good test case to identify issues in mapping complex control to similarly architected reconfigurable platforms.

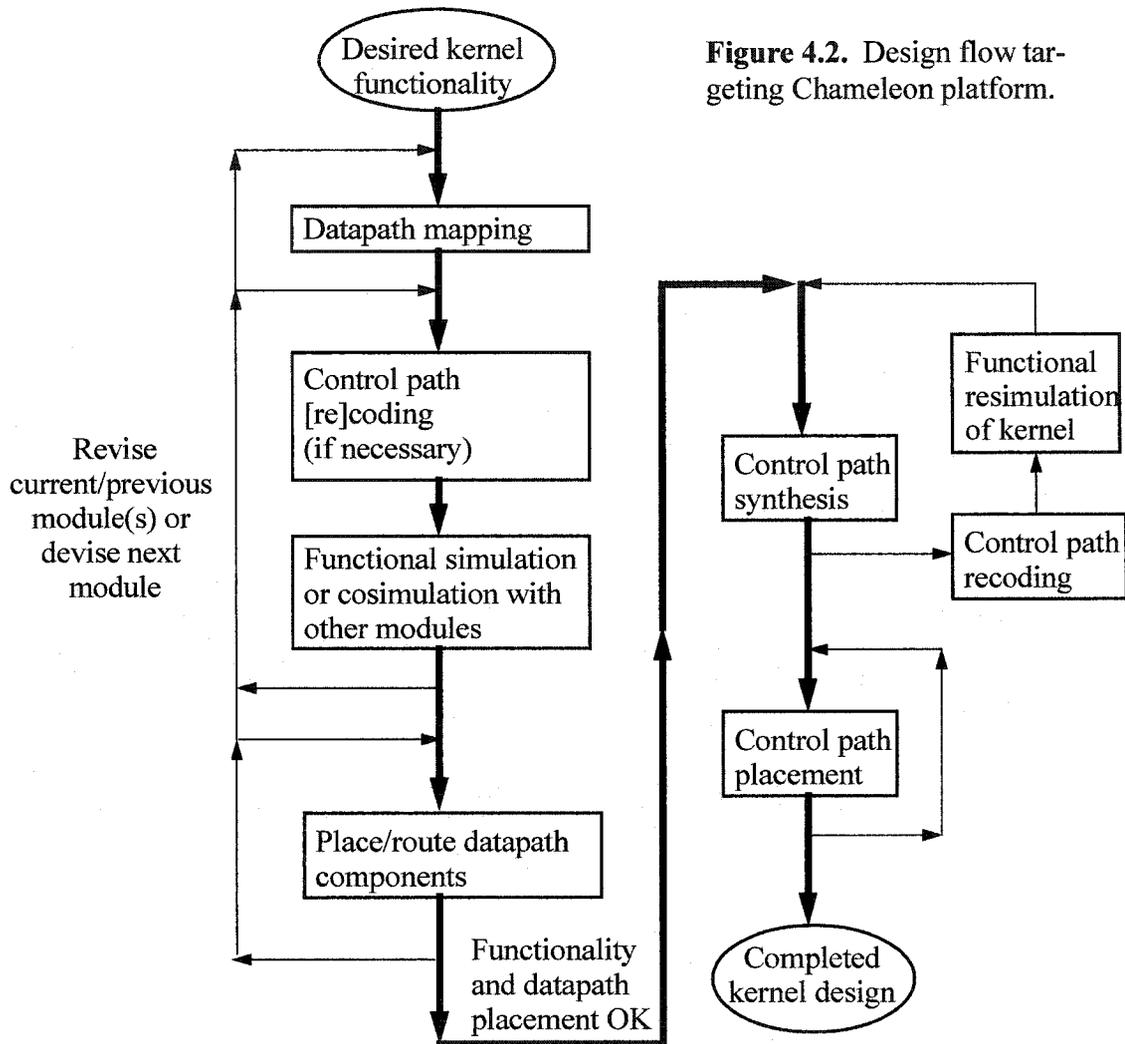
4.3 Kernel Design Flow

Figure 4.2 shows the flow used for this design case. Kernel design is implemented module by module. For each module, the datapath functionality is first mapped to DPUs and concatenated memories using netlist-like “structural” Verilog. DPU personalities are encoded as parameters of the DPU instances. To orchestrate the timing of these personalities, the control path logic is then specified using a C-like “procedural” Verilog that maps to the registered PLA. One or more modules are then functionally verified by simulating with behavioral models for datapath and memory components. As in digital design in general, the design revision and functional verification step is iterative. Until the results are as desired, it may be necessary to change the functionality of the datapath, and the control flow and/or timing of the control path.

The next step is to place datapath components so that the required interconnections are realizable. This step is the start of the Chameleon tool flow. It is performed on a regular basis starting from an early point in the kernel design because it checks code synthesizability, routability, and timing for the specified configurations. It may be necessary to loop back and revise the design further in order to improve its placability.

Synthesis of the control logic is then attempted. In this phase, the vendor tools generate the sum-of-products logic for the control signals in each tile, based on the Verilog code. They also ensure that where products overlap between multiple signals in a tile, they are shared. PLA resource usage information is then generated. However, the default assignment of control signals to tiles is not optimized. Therefore, overlap and resource usage is not initially representative of the final control resource binding. Hence, the synthesis phase is initially used only to check synthesizability i.e. that control logic code can indeed be rendered into sum-of-products form, as well as to get an idea of the logic complexity for individual control signals. It remains for the designer to specify bindings of control signals to the PLAs of specific tiles (next step). Before doing so, it may be necessary to iterate with different control path designs depending on the logic complexity of the signals.

The control binding takes place in the final step, control path placement, most of which happens at the very end of the kernel design. That is when the datapath placement is finalized, thereby fix-



ing the location of SRB registers that drive the DPU personalities (Figure 3.10(c)). In this phase, the designer tries to assign SRB bits that are free to move to specific tiles so that as much PLA logic is shared between bits as possible, while at the same time avoiding the overloading of any tile.

Because the synthesis tool runs through the entire flow, the control path placement results are ignored until such time that control path placement constraints can be meaningfully specified i.e. after the datapath is placed. Before successful placement of the datapath, the control path synthe-

sis step is still somewhat useful, since it generates resource usage statistics as a rough sanity check of kernel realizability.

To find a good placement for the control bits, they can be divided into two groups. *DPU control bits* drive the personality select lines of a DPU. *State bits* feed back to the PLA, thus acting as memory registers for state machines in the control path. The tile assignment of DPU control bits are determined by the DPU placement. Therefore, control path placement constraints consist primarily of designer-provided tile bindings for the state bits, specified in such a way as to balance resource usage between PLAs and avoid overrunning their resource limits. As with PLAs in general, these limits are: the number of state bits, the number of product terms, and the number of input signals feeding the product terms.

Duplication of product terms can be minimized by placing a state bit in a tile where many of the required product terms are already being used. Before any state bits have been placed, a PLA's product terms consist only of those for the DPU control bits. Therefore, resource sharing can be maximized by attempting to place the state bits after the datapath components have been placed for the entire design.

4.4 Datapath Design

Complexity Overhead in Serializing

The algorithmic illustration of Figure 4.1 does not show issues related to implementation. Serializing the butterflies does more than simply extend the time it takes to perform all the arithmetic operations. The apparent interconnect complexity in Figure 4.1 is converted into address generation for memory access. The amount of resources for the explicit butterfly computations is rivaled by that required for complex addressing scheme, and for synchronizing data flows.

Parallelism and Cycle-Accurate Timing of DPU Personalities

In contrast to DSPs, the fabric allows the address generation to be pipelined with the entire butterfly processing chain. Thus it runs in parallel with butterfly computation. The cost of this capability is that the designer must time all operations and dataflows to the clock cycle. In contrast to ASIC/FPGA design, this is done by designing the control path to properly time the changes in each DPU's user-defined personalities. Assembling configurations at the level of DPU cycles bears resemblance to DSP programming, except that the need for cycle-accurate timing is not abstracted

out. This extra degree of difficulty in programming the DPUs is also the extra degree of low-level control that enables hardware performance.

Note that there are 2 sources of parallelism that allows the entire fabric to be utilized at every clock cycle: (1) pipelining and parallelizing operations to achieve 1 butterfly/cycle, and (2) running 4 parallel FFTs on the 4 slices of the array.

Coarse Granularity and DPU Packing

Packing operations in the datapath takes on a special importance in targeting coarse grain platforms. Resource requirements increase in big chunks, and conservation is viewed at the level of DPUs rather than gates. If this was an ASIC design, for example, the low-level control of how to assemble logic makes it trivial to realize bit-wise operations like boolean masking and bit shifting. In contrast, these operations require entire DPU cycles in a processor array. Methods to conserve DPUs include (1) using upper and lower 16 bits of the datapath for different instances of the same operation; (2) multiplexing nonsimultaneous operations onto a common DPU; and (3) shuffling the order of datapath operations along data flows (especially delays) so that more elements can be packed into a DPU.

Design of the FFT Kernel

The following subsections illustrate the paradigm for mapping to datapath resources. We first present the computational core in Figure 4.1(b), including arithmetic artifacts, then the added functionality arising from serialization and memory organization.

4.4.1 Dynamic Range Considerations

For the real and imaginary parts of complex numbers, we use a fixed point two's complement representation, with the binary point taken to be on the left i.e. the representable range is $-1 \leq x < +1$, or $-1.00..0_2 \leq x < +1.00..0_2$ in binary. Numbers that marginally overflow beyond this range “wrap around” and cause the maximum possible error [OS89]. Therefore, occurrences of $+1.00...0_2$ in the data or twiddle factors must be reduced to $+0.11..11_2$. For the twiddle factors, this was done by negligibly attenuating all twiddle factors by $0.11..11_2/1.00..0_2$. For the data, appropriate scaling can be achieved by simply taking the binary point to be at the extreme left.

To avoid overflow during the computation of the FFT, each of the 10 stages must halve the data set since the input additions generate results in the range $-2 \leq x < +2$. This doubling of dynamic

range is inherent in the transform; consider an 1024-point FFT on a signal of all ones. The spectrum will be a single DC tone of amplitude 1024 (Figure 4.3), or a doubling of amplitude in each of the 10 stages. Since energy is conserved in Fourier Transform (Parseval's theorem), this example is the most extreme example of taking energy that is uniformly distributed in time, and focusing it into a single tone (of which there are 1024).

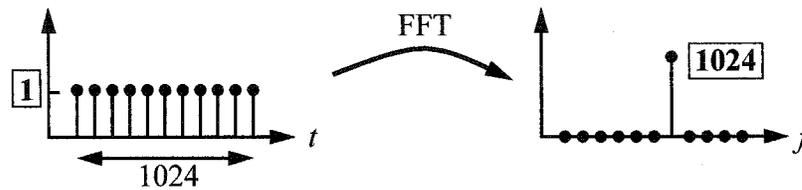


Figure 4.3. The FFT of a $2N$ -point time-domain record can have magnitudes that are 2^N times greater than the magnitude in the time-domain record.

In an ASIC design, this increase in required dynamic range manifests itself as bit-width expansion. The obvious solution of dropping the lowest bit by shifting right is not suitable for the DPUs because only one operand path contains a shifter. Section 4.4.2 shows that fixed point multiplication provides this halving for free.

4.4.2 Complex Arithmetic

Figure 4.4 shows the complex multiplication that dominates the butterfly. Since the functionality does not vary with time, the mapping is simple, though it uses much logic. It is a direct mapping of long-hand complex multiplication, with 4 real multiplications and 2 real additions. Logic is conserved by using upper and lower 16-bits of a 32-bit datapath for real and imaginary parts of complex data. This fits well with the fact that the multipliers take 16-bit multiplicands; multiplier personalities determine whether multiplicands are taken from upper or lower half words. Each real multiplier generates a 32-bit two's complement sign-extended product from 16-bit two's complement multiplicands.

The butterfly logic can be simplified by having two numerical artifacts cancel each other out. Since 16-bit multiplicands have only 15 numeric bits and one sign bit, the product has only 30

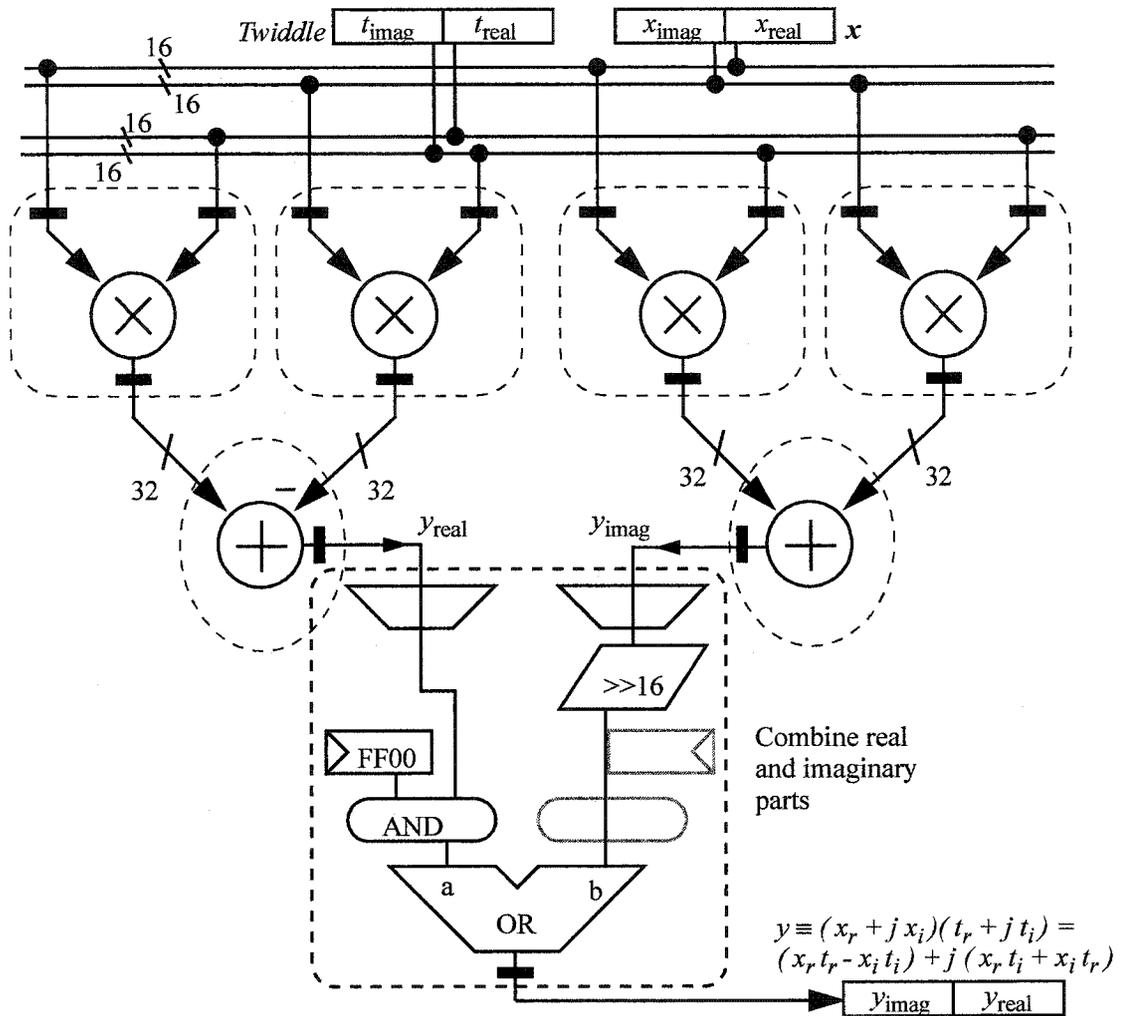


Figure 4.4. Complex multiplier with one multiplication per cycle, broken down by DPUs and real multipliers.

Four of the six multipliers are used and one DPU at the end truncates real/imaginary parts to 16 bits and concatenates them. The pipelining is the minimum required for 100MHz clocking.

meaningful numeric bits; the 31st bit is a sign bit, and the 32nd bit is a 1-bit sign extension. If the binary point is taken to be at the extreme left, this effectively halves the number, which provides the required per-stage attenuation of 1/2. This is a rare case where two artifacts of implementation cancel each other. An example of this halving is shown in Figure 4.5 using 5-bit multiplicands and a 10-bit product.

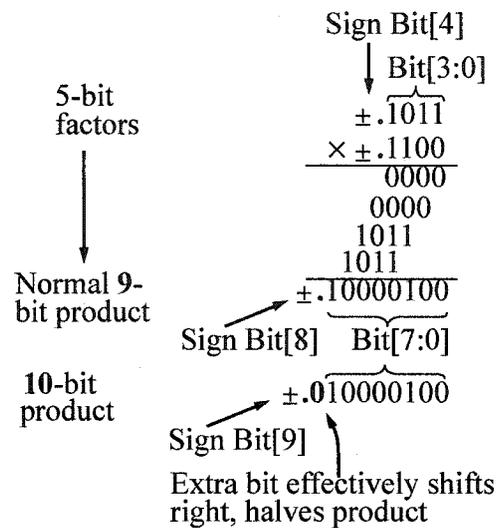


Figure 4.5. Long hand multiplication of N -bit multiplicands yields an $(2N-1)$ -bit product. Extending this to $2N$ -bits has a halving effect if the binary point is taken to be at the extreme left.

Despite this automatic halving there is still an overflow hazard from truncating the product to 16 bits. This is because it always discretizes the number toward $-\infty$ - $j\infty$. Normally, this quantization error potentially moves the product just outside the unity circle on the complex number plane. However, due to the natural halving in multiplication, the real/imaginary parts potentially lie just outside of $-1/2 \leq x < +1/2$ (Figure 4.6). This postpones the risk of marginal overflow to the next stage, when two such numbers are added. The risk presents itself when the two operands have been rotated (by multiplication) to have similar phase, almost aligned with the real or imaginary axes. To avoid overflow due to quantization error, the ALUs can perform saturated addition; this detects overflow and returns $-1.00..0_2$ or $+0.11..1_2$ as appropriate. The error due to this beneficial clipping is minimal, since it is in the order of quantization noise.

The marginal overflow that is avoided by saturated addition can be generalized beyond the FFT. For example, if -1 undergoes complex multiplication by itself, the result is $+1$. The natural halving provided by the above sign-extension yields $+1/2$ and prevents overflow, but two such numbers added together would marginally overflow. If it wasn't for the marginal clipping from saturated addition, the value would need to be scaled down further by a full half, thus degrading the signal to quantization noise by 3dB.

In addition to avoiding overflow, the butterfly subcircuit must incorporate measures to equalize the delay between the twiddled and untwiddled outputs. To do this, the untwiddled arm was subjected to a dummy multiplication by $+0.11..1_2 \approx 1$; this also equalizes the attenuation in the untwid-

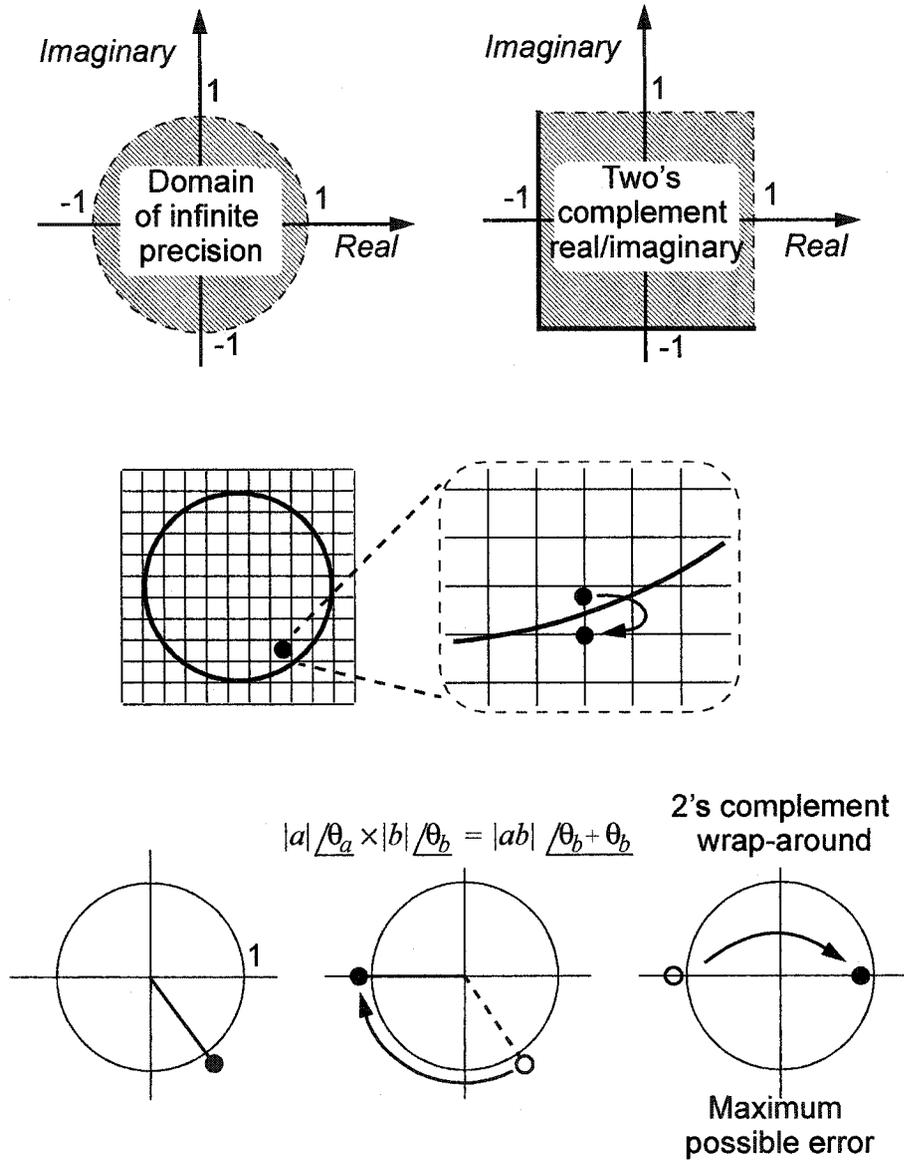


Figure 4.6. Heading off an anticipated overflow problem. Because the input data is normalized and each stage introduces an attenuation by $\frac{1}{2}$, the domain of all arithmetic results should be bounded by the unit circle. Two's complement representation of the real/imaginary parts of a complex number can represent any point in the unit *square*. Truncating the 32-bit real/imaginary part of a complex product always causes value to change toward $-\infty$ and potentially generates a point outside the unit circle. If multiplied by a similarly large number, the result could lie outside the unit square and wrap around, causing the maximum possible error.

dled arm with the attenuation in the twiddle factors. In contrast to DSPs and ASIC design, using multipliers for delays when they are available is more efficient than wasting high-powered DPUs on a 32-bit delay line.

Some additional benefits result from uniformly attenuating both butterfly outputs by $0.11..1_2$. The first is that there is no distortion. The second is that it reduces the likelihood of overflow, since all datapoints are retracted inward from the unit circle (Figure 4.7).

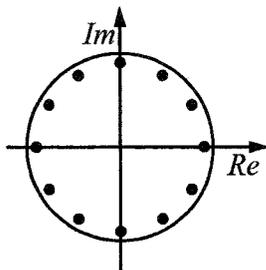


Figure 4.7. Uniform attenuation of both butterfly outputs pulls data points and Twiddle factors away from the unit circle.

The final benefit of uniformly attenuating both butterfly outputs is related to the freedom from distortion. Since the attenuation is systematic, it is easy to analyze the effect of the attenuation over the 10 FFT stages. For each stage, the error due to attenuation is minimal since it is in the order of quantization noise. Over 10 stages, the accumulated attenuation is 0.9969, or 0.3% error. This is too small to bother compensating. The computation time is not justifiable to do this in floating point outside the kernel. If done in the kernel, it represents 10% more kernel run time, as well as much more control complexity; since multiplicands have amplitudes ≤ 1 , an 11th stage would be needed to multiply by $\frac{1/2}{0.9969}$ and perform a quadrupling shift to undo the $1/2$ numerator and the natural halving of multiplication. Furthermore, since the per-stage attenuation by 1 LSB prevented quantization error from pushing data points outside the unit circle, it is possible that undoing that attenuation could send some data points beyond the unit circle and risk overflow.

For delay equalization of the butterfly outputs, some alternatives were considered and discarded. It is not possible to use multipliers as delays without attenuating because factor $+1$ is not representable. Neither are there enough free multipliers in the slice to multiply by -1 twice. It is also not an option to use one multiplier to multiply an entire 32-bit complex number by -1 because multiplicands are 16 bits.

Another option that was ruled out is to attenuate by $1/2^{1/10}$ per stage. After 10 stages, the total attenuation of $1/2$ can conceivably be undone by a doubling shift. As above, this also risks sending data beyond the unit circle and incurring wrap-around.

4.4.3 Twiddle Address Generation and Stage Tracking

The twiddle factors are stored in a block of chained memories from one tile. These complex constants are ordered according their exponents k (Figure 4.1). If the butterflies are processed top down (Figure 4.11) within each stage, it is clear that the twiddle factor addresses can be generated by a modulo- $N/2$ accumulation of 2^{i-1} in stage i . Figure 4.8 shows a subcircuit to do this. The twiddle address step undergoes a doubling shift-left once per stage.

Since the DPU's lower 16-bits is more than adequate for this, the same shifting action in the upper half word can be used for the additional task of counting the FFT stages and flagging the last stage for shutdown activities. This is done by shifting a 1-bit toward the MSB; the ALU flag must be configured to mirror the MSB.

This contrasts with ASIC design, where stage tracking may be efficiently realized using a tiny binary counter and comparator (which may be just the counter's overflow bit). In this array of 32-bit DPUs, using the available upper half word of a DPU that already shifts once per stage eliminates the need for two DPUs to count stages and compare. The stage counter and comparator is not implemented in the PLA because it uses too many product terms. Thus the manner of counting is driven by resource conservation.

4.4.4 Data Set/Memory Organization, Butterfly Scheduling, and Kernel Architecture

The partitioning of the application data plays a central role in determining the kernel architecture because it determines how the datapath subcircuits are time shared. In this design, the main consideration in memory organization is to enable data retrieval to support 1 butterfly/second. Since each butterfly requires 2 data-reads (and data-writes), the 1024-point data is broken into 2 banks of 512 words (Figure 4.11). Since chained memories share common address and data lines, each bank can only be read by only one DPU at a time, and written by only one DPU at a time. Breaking the data set into 2 banks enables the retrieval of the 2 data points/cycle. A major design

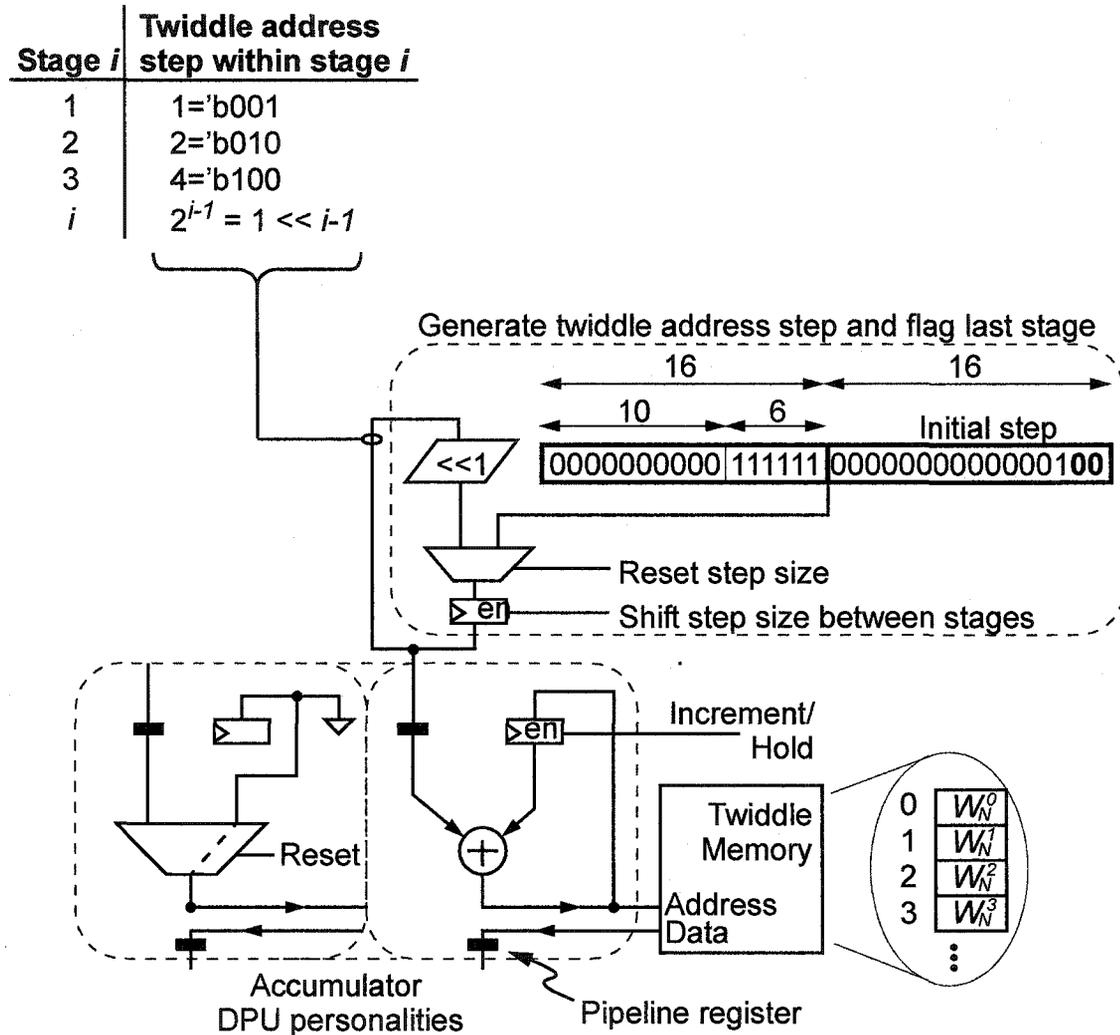


Figure 4.8. Twiddle factor address generation in the lower 16 bits by modulo-512 accumulation of $1 \ll i-1$ for stage i . Accumulator DPU personalities: Reset, Increment, and Hold. Address step size generator personalities: Reset, doubling Shift, and a Hold mode that is in effect for the processing of an entire stage. The 2 LSB in bold are dummy bits because memory is addressed as 8-bit words rather than 32-bit words. Modulo-512 wrap-around is realized by configuring the memory to mask out higher order address bits.

challenge is to interleave butterflies acting on upper bank data with those acting on lower bank data to sustain one butterfly per cycle.

Figure 4.11 shows this memory partitioning for a smaller 16-point FFT. Stage 1 presents no problem to achieving one butterfly/cycle because each butterfly's input comes from a different databank. Beyond stage 1, however, each butterfly needs 2 cycles to read both inputs from the *same* databank. Therefore, corresponding butterflies from the two databanks are interleaved to achieve 1 butterfly/cycle. The two sequentially retrieved data points bound for the each butterfly must also be time aligned before entering the butterfly (Section 4.4.7).

Figure 4.11 also shows that the data memory is scanned in a stage-dependent order. Since upper and lower databanks have the same data access patterns, it is only necessary to generate the memory addresses once. A further simplification results from the use of in-place processing i.e. each butterfly output is written back to the same location as its corresponding input. Therefore, the write-addresses for the butterfly outputs are the same as the read-addresses for corresponding inputs. Hence, memory-writes can use the addresses from the memory-reads, though the addresses must be delayed by the latency of the butterfly and the memory fetch.

The resulting architecture is shown in Figure 4.9. The 9-cycle delay line was arrived at after the detailed design of the butterfly. A tenth cycle of delay is obtained by using the input registers of the memory write DPUs (the write address is taken from the DPU after the ALU).

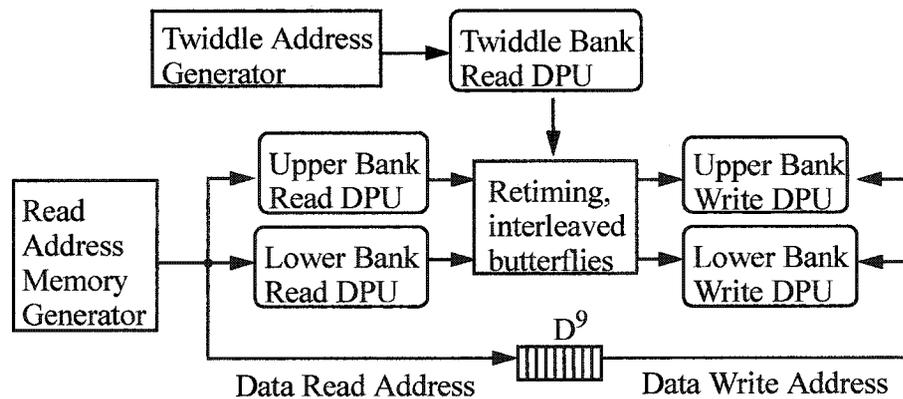


Figure 4.9. Architecture of FFT datapath. Delay $D \equiv Z^{-1}$.

4.4.5 Delay Lines

In this architecture, delay lines in the datapath can be costly. This is because datapath registers are meant for pipelining, and are interspersed with logic. There only 3 registers embedded in each DPU; since two registers lie in parallel input paths, only two registers can ever be in series. To extend the delay that can be provided, a DPU can emulate a 3-stage shift provided by a simple sequencing of personalities (Figure 4.10).

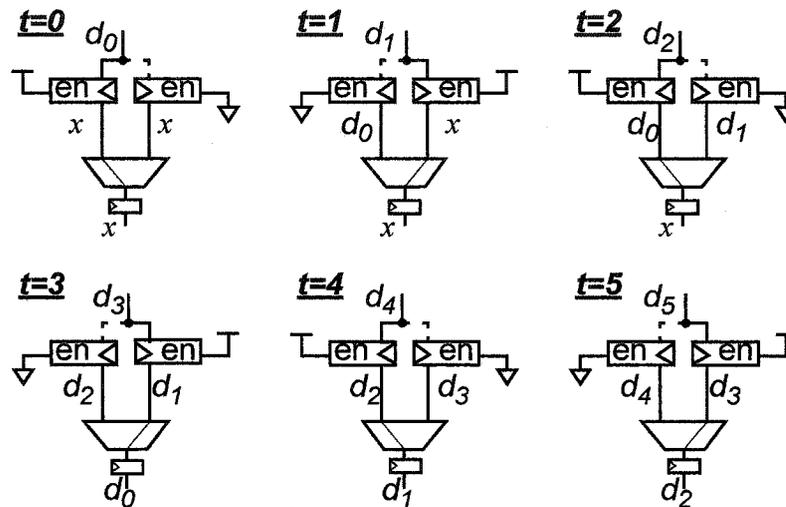


Figure 4.10. Animation of Chameleon's method to squeeze 3 delay cycles from 1 DPU, as described in [Cha01b].

Some restrictions may limit the use of this 3-cycle DPU. A delay line formed from several such DPUs can only be tapped in between DPUs. This requires that some DPUs be used for less than 3 cycles of delay if several tapping points are not a multiple of 3 cycles apart. Since the required delay times may not be known exactly until the entire kernel is completed, this represents uncertainty in DPU usage.

An change in platform architecture that could simplify the use of datapath delay lines is to provide more registers in a DPU that can be serially linked into a delay line. It may also be possible to design the DPU to use these registers as a register file. Delay lines can still be tapped only between DPUs, but fewer DPUs needed. This alleviates the uncertainty in DPU usage. With enough delays within a DPU, it is possible that only one DPU is needed between taps in many situations; thus, the DPU usage may be very predictable based on the number of taps required.

Alternatives to using a delay line should be considered for platforms that intersperse registers with logic. The cost of a delay line should be compared to that of duplicating the logic which generates the data stream to be delayed (the databank memory addresses in this case). This is only an option if the data stream is generated by the kernel from scratch, rather than by performing operations on another data stream. For this design, the number of DPUs to generate the write-address was similar to that for delaying the read-address. The delay line approach was chosen because it did not require duplicating the control logic for the address generator.

4.4.6 Data Set Access Pattern and Address Generation

Figure 4.11 shows the data set access pattern required to process the butterflies sequentially. Beyond stage 1, the access pattern is stage dependent in a convoluted way. To simplify the pattern, we seek an address scanning pattern for the *upper* inputs to the butterflies. It is especially good if we identify a dependence of the address on the butterfly count, since that is the only signal that varies uniquely with the butterflies within the stage. We also know that the relationship must have a dependence on the stage count. When a simple enough pattern is found that it can be synthesized using datapath logic, the *lower* input to the butterfly is obtained by adding the stage-dependent butterfly height. The memory-read DPUs must then alternate between the butterfly upper input and lower input addresses so that the upper and lower inputs are retrieved consecutively.

As can be seen from Figure 4.11, this complicated addressing pattern does not apply to stage 1 because the addressing pattern is trivially simple in the first stage. Therefore, additional DPU personalities are required to generate the butterfly input addresses for stage 1. This translates into more control path resources because there are more DPU personalities to control and more conditions have to be monitored to properly time them.

For all stages beyond stage 1, the identifiable pattern in the butterflies' upper input addresses is shown in Table 4.2 for a smaller 8-point example. It is obvious that the address can be generated by inserting a binary zero into the butterfly count at the bit position corresponding to the stage count (starting from the left). In stage i_{stage} , a binary zero is inserted at bit $[N_{\text{stage}} - i_{\text{stage}}]$, where the LSB is bit[0].

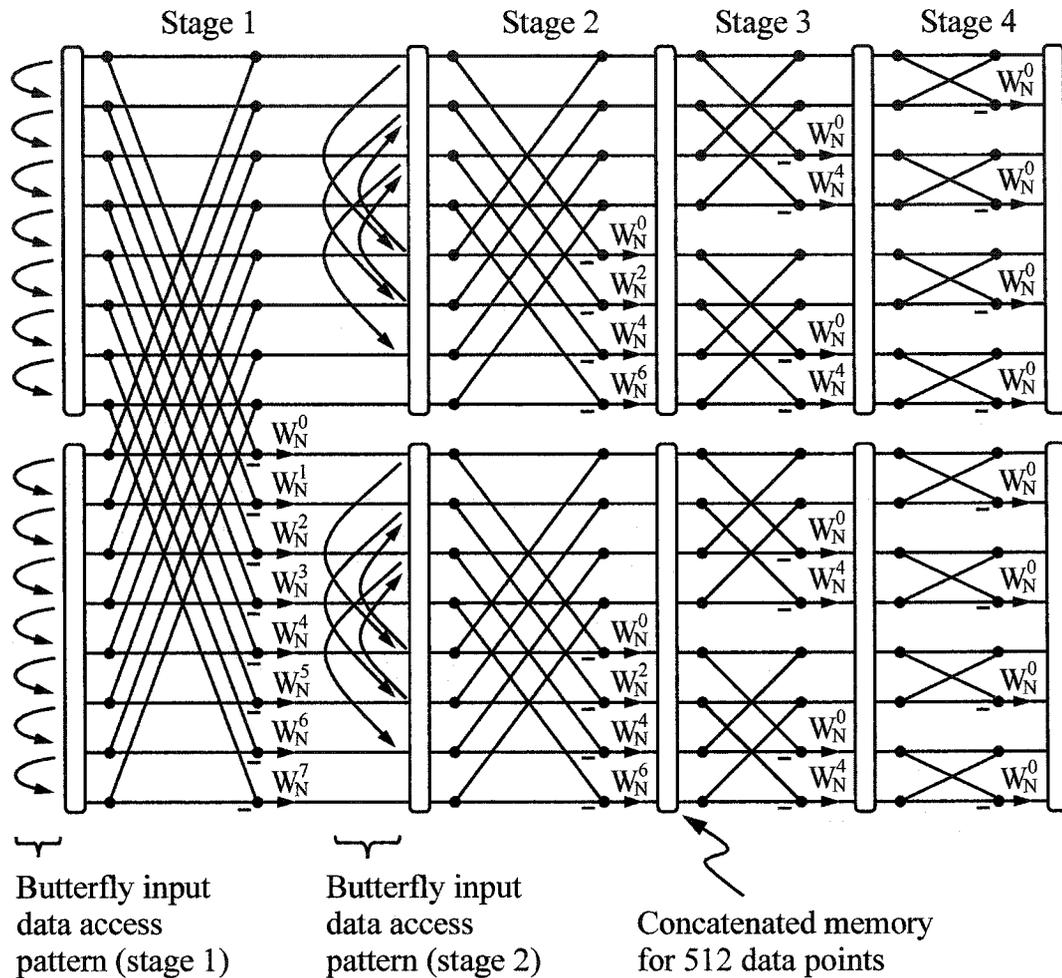


Figure 4.11. Repeated butterfly for N -point decimation-in-frequency FFT with input-normal ordering, $N=16$.

Stages 1 and 2 show the data set access pattern to process the butterflies sequentially. For in-place processing, each of the top memory blocks represent the same physical memory, as do the lower memory blocks.

Since the butterfly height is also $1 \ll (N_{stage} - i_{stage})$, the lower input address is generated by inserting a binary 1 into the butterfly count at the same place that the binary zero was inserted for the upper input address:

$$\left[\begin{array}{l} \text{Butterfly lower} \\ \text{input address} \end{array} \right] = \left[\begin{array}{l} \text{Butterfly upper} \\ \text{input address} \end{array} \right] + 1 \ll (N_{stage} - i_{stage})$$

Butterfly count	Butterfly's Upper Input Data		
	Address (Either Data Bank)		
	Stage 2	Stage 3	Stage 4
0= 'b 00	0= 'b 000	0= 'b 000	0= 'b 000
1= 'b 01	1= 'b 001	1= 'b 001	2= 'b 010
2= 'b 10	2= 'b 010	4= 'b 100	4= 'b 100
3= 'b 11	3= 'b 011	5= 'b 101	6= 'b 110

Table 4.2: Butterfly upper input address as a function of FFT stage and butterfly count. An 8-point 4-butterfly FFT is assumed.

In order for the address generator to perform insertions for the butterfly's upper or lower input addresses, the butterfly count must be split up into two pieces, one to left of the inserted 0 or 1, and another to the right of it (Figure 4.12). In contrast to ASIC design, this is not just a wire routing exercise; this is to be expected in a prebuilt coarse grain platform. These pieces must be extricated from the butterfly count using masks, handled as separate datapath flows, shifted appropriately, then recombined using Boolean operators. As in the twiddle address step generator, it is desirable to design the logic using 16-bit half words to handle each of the two pieces rather than the full 32-bits of the datapath. Nine bits are required to address 512 twiddle factors, plus 2 appended zeros for byte-wise addressing; these 11 bits fit easily into a half word.

Figure 4.13 shows a way to separate the left and right portions of the butterfly count and insert the binary zero for the address of the butterfly's upper input. DPU (a) generates the butterfly count while DPU (b) generates the masks to extract the left and right parts of the butterfly count. A feedback path through the shifter allows these masks to shift between stages as the position of the inserted bit shifts. DPU (c) performs the extraction by ANDing with the left and right portions of butterfly count. DPU (d) recombines left and right pieces by shifting the left piece from the upper 16 bits to the lower 16 bits, then ORing with the right piece. The left piece is only shifted down by 15 bits in order to leave room for the inserted bit.

The butterfly counter holds each count for 2 cycles so that both the addresses for upper and lower input to the butterfly can be generated. To pack the DPU with functionality, the otherwise unused upper half word is used for the related task of testing the count to generate an end-of-stage pulse *BfyCntDone*. Therefore, the butterfly count must be duplicated from the lower half word to

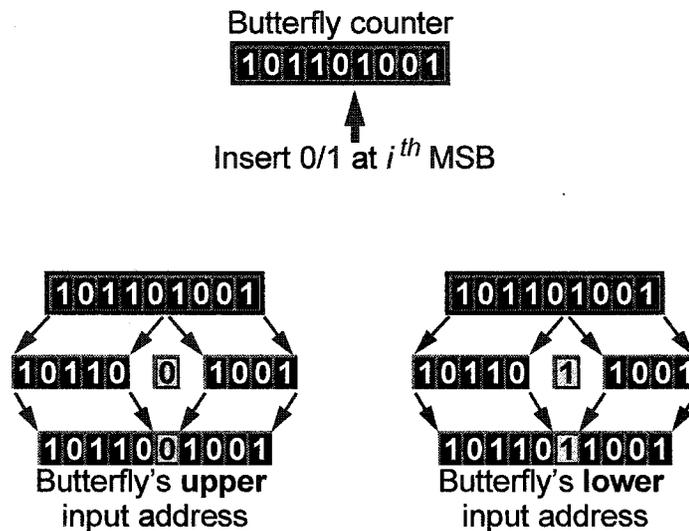


Figure 4.12. The addresses of a butterfly's inputs can be generated by inserting a bit into the i^{th} bit position, where i is the FFT stage being processed.

A zero is inserted for get the upper input's address. A one is inserted to get the lower input's address.

the upper half word in the counter loop, and the comparison value must be stored in the upper half word of the input register in the right operand path; this register also contains the counter's unit step size in the lower half word. Unlike typical DSPs, the ALU can be configured to test for equivalence between operands regardless of what computation is being performed; this eliminates the need for a separate comparator DPU.

In the rest of this circuit, the left/right portion of the butterfly count is processed in the upper/lower half words of the datapath. The mask generator's initial mask is set up to extract the minimal left portion of *ButterflyCount* (zero out of the eleven bits) and the maximum right portion (all of the 11 bits). These masks shift right 1 position each stage to allow the inserted bit to shift right, as required. Note that the butterfly count only has 10 bits, including the two unused LSBs; the masks will be properly aligned in stage 2. Addresses before stage 2 are not generated by bit insertion.

To insert a binary one for the address of the butterfly's *lower* input, it is straightforward to show that the masks must be inverted, and the AND/OR operations of DPUs (c) and (d) must be interchanged. Therefore, alternating between upper and lower input addresses requires a modified

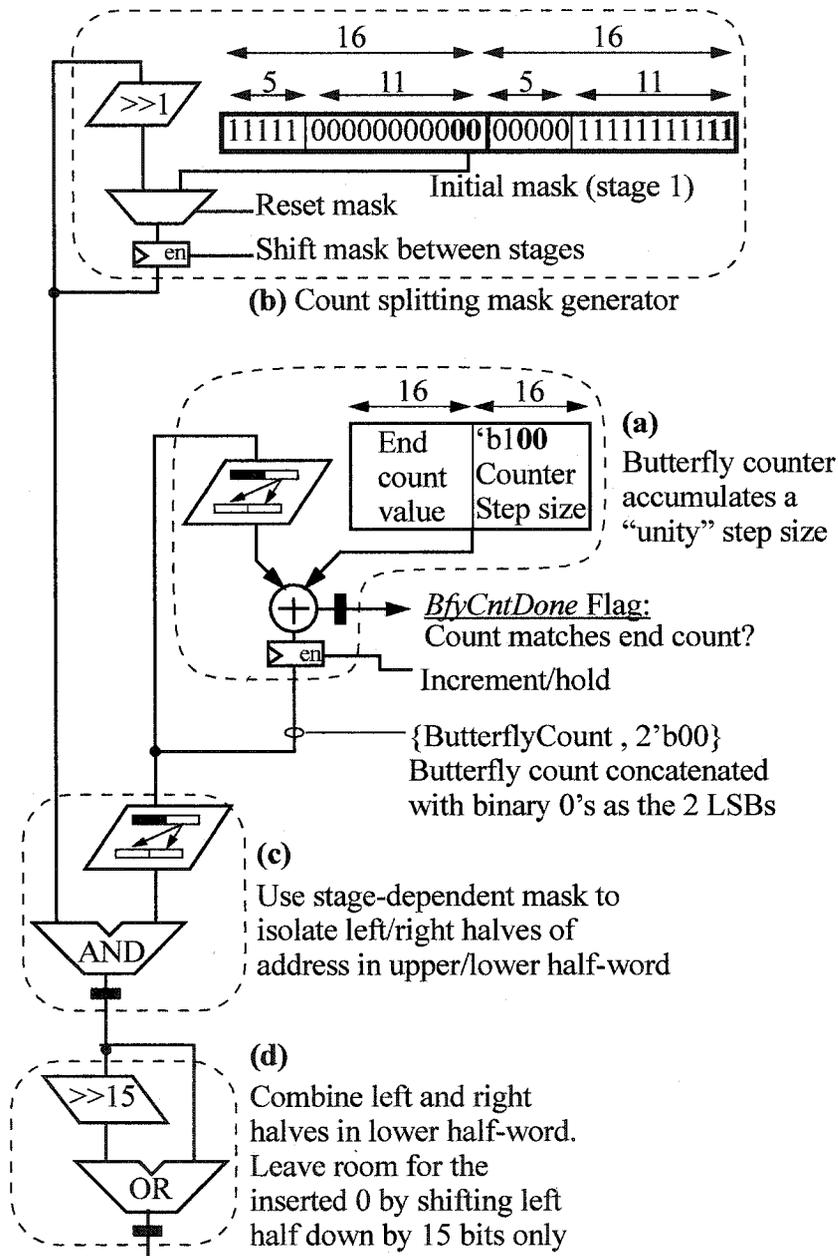


Figure 4.13. Address generator for butterfly input data. Inserting a binary 0 into the butterfly count generates the address for the data that feeds the butterfly's upper input. These DPU personalities must alternate with those for generating the address for the butterfly lower input in order to get both inputs one after another.

mask generator DPU (b) that uses the ALU to invert the mask every cycle. A further DPU personality is needed to combine the shifting of the mask with the appropriate inversion during stage transitions. Similarly, DPUs (c) and (d) do the extraction and combining, and must alternate the personalities for upper and lower input addresses every cycle in order to retrieve upper and lower input data.

4.4.7 Retiming Butterfly I/Os

Figure 4.14 shows the butterflies for the top and bottom data banks, as well as the retiming registers required to offset them in time for interleaving. Output registers resynchronize data bound for the top and bottom data banks so that they can use the same write address coming from the delay line (Figure 4.9). These 6 register delays “D” will consume 5 DPUs; this must be whittled down by time-sharing physical registers.

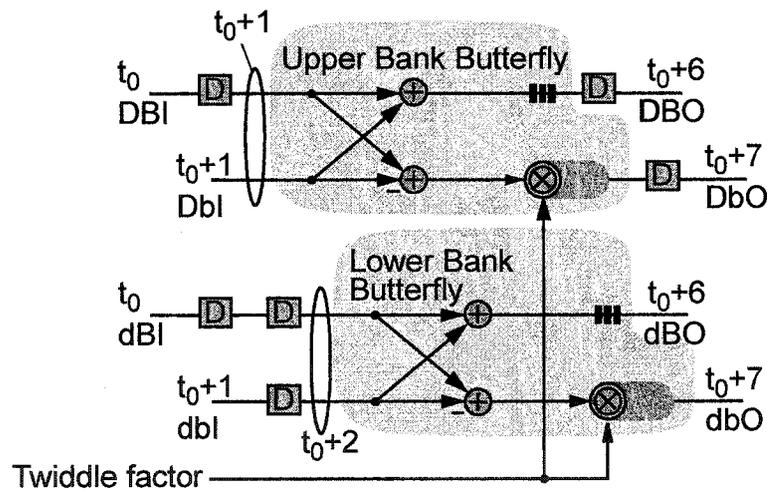


Figure 4.14. Interleaved butterflies for top and bottom data banks. The per-stage attenuation of the untwiddled datapath balances the attenuation in the twiddle factor.

Figure 4.15 shows demuxes from the data banks sending data to the 4 input arms of Figure 4.14, followed by a mux feeding the common butterfly. In Figure 4.16(a), register usage is reduced by pushing register pairs through [de]muxes to merge them. The final simplified circuit is shown in Figure 4.16(b). Similar sharing is possible for the butterfly output registers.

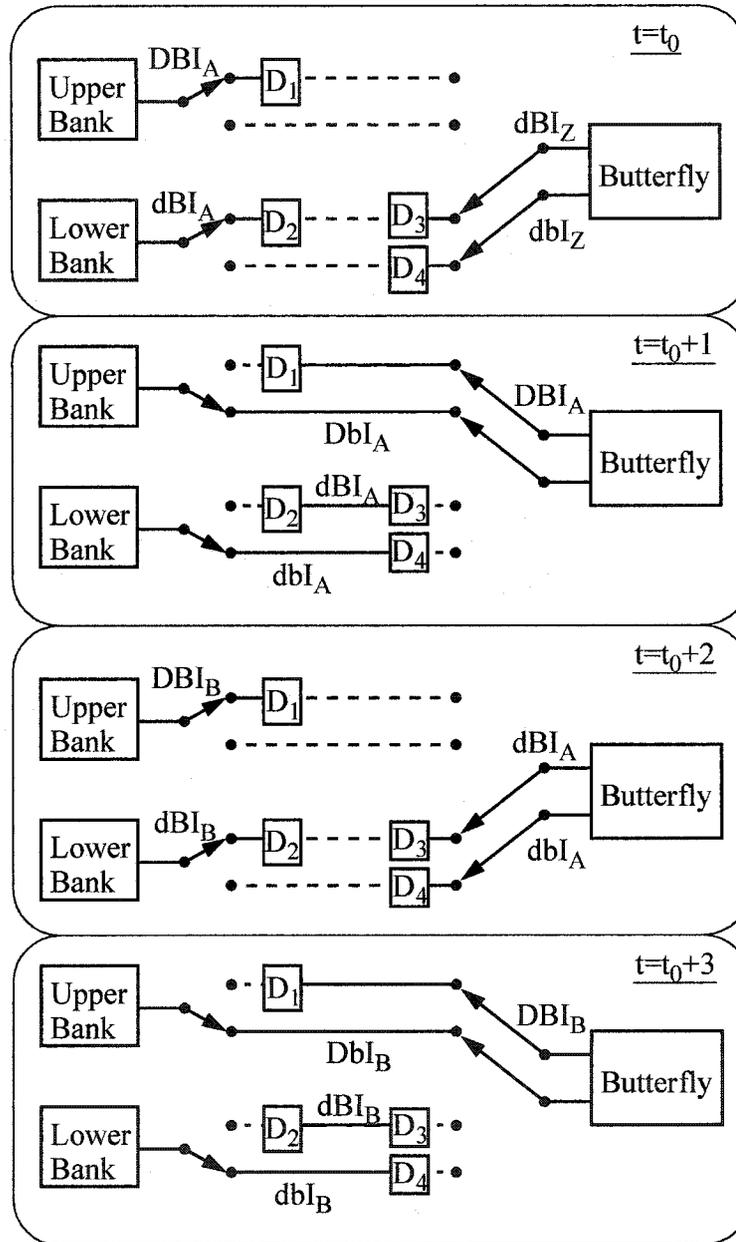


Figure 4.15. Determine the muxes by identifying when ports and registers source and sink which pieces of data.

In the following examples, delays are also pushed along their data flows to find locations where they coincide with readily available DPU input registers, or where equivalent timing offsets can be

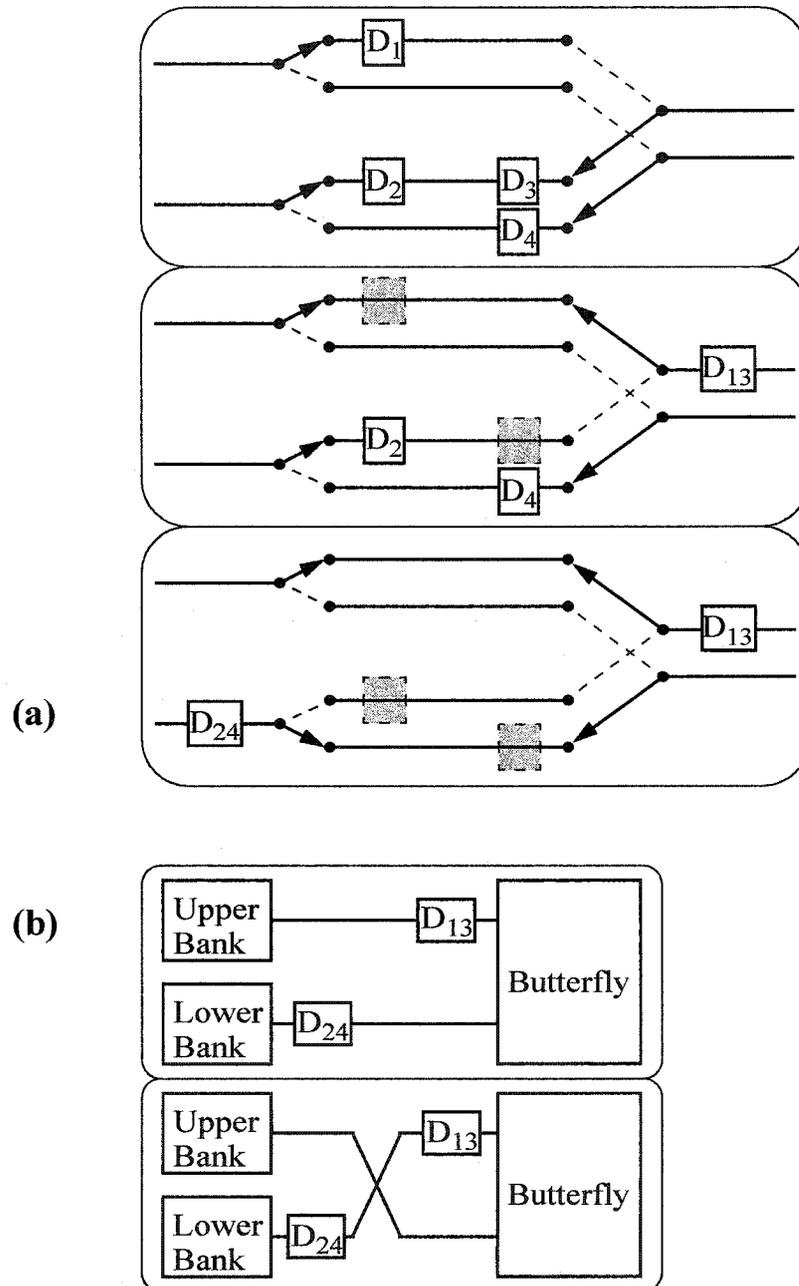


Figure 4.16. Sharing registers at butterfly input.
(a) Each time registers push through a [de]mux and merge, the [de]mux changes state. After merging the delays, the top 2 wires are never used simultaneously and can be merged; same with the bottom two wires, yielding the alternating phases in **(b)**.

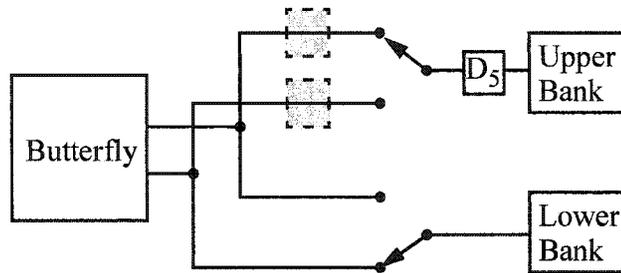


Figure 4.17. Merging of delays at the butterfly output.

realized without using up a DPU. This is easily generalizable to other kernel designs and platforms.

D_5 cannot be incorporated into the input register of the upper bank's write DPU because memory write *data* is taken from the DPU input before the input register (Figure 3.10(a)). However, the equivalent delay of write data relative to write address can be achieved by advancing the write *address* to the upper bank's write DPU i.e. *bypassing* the write DPU's input register, which was originally required as a tenth delay after the 9-cycle delay of Figure 4.9; this can be seen as removing a single delay from each of the of the inputs to the upper bank write DPU i.e. pushing a delay through the DPU, except it has no output.

D_{13} can be incorporated into the input registers of the DPUs that sink the butterfly's upper input i.e. each of the adder and subtractor at the butterfly input will register the upper operand only.

D_{24} in Figure 4.16(b) can still be folded into the lower bank's read DPU and eliminated by equivalently delaying the read address to the lower bank. This is equivalent to pushing a delay through the read DPU from output to input in Figure 4.9; the delay is realized by registering the input of the read DPU (lower bank only).

Similar merging can be performed at the butterfly output Figure 4.17.

In addition to the requirement for retiming delays, interleaving the butterflies also entails a number of [de]muxes. Demuxing is realized by distributed muxing i.e. broadcasting a signal to several DPUs and having those DPUs either select that signal or not. Thus, muxing and "demuxing" is done at the cost of DPU personalities to select signals; the number of DPU personalities get multiplied by the number of signals feeding the mux, which affects control complexity.

4.5 Control Path Design

The control path consists of a main state machine and cooperating auxiliary state machines, or substate machines. The main state machine takes care of the coarse modes i.e. the ones relating to few major changes in general state over a great lengths of time, such as the entire duration of kernel execution. The substate machines provide fine timing control of DPU personalities.

4.5.1 Main State Machine

As shown in Figure 4.18, the main state machine can be appear very simple for several reasons. First, it is essentially a Mealy machine, which allows many combinations of output values without requiring separate states for each. Since PLA output registers that drive DPU personality selection are not visible to the control path, they should be regarded as output delays only. Even though a Moore machine has separate states for each possible output combination, the overall combinational logic for the Mealy machine might not be less than that of a Moore machine because Mealy output logic has more complex dependence on inputs; however fewer main state bits are needed for the Mealy machine, which generates fewer product terms when ANDing main state bits with substate bits in control logic expressions.

Simplicity of the main state machine is also facilitated by offloading much of the fine timing complexity to the substate machines. This allows one to select PLA-efficient substate machine structures that are suitable for pipelined datapath control for the detailed control. Structures like single bit-width shift registers and flag registers can be subjected to different behavioral specifications within each main state.

Another factor contributing to simplicity of the main state machine is the global routing of state bits and DPU flags to all the PLAs within the slice. The control path can use state information from other modules and PLAs, greatly alleviating the state bit requirement to keep track of this state information.

4.5.2 Shift Register Substate Machines

During the start of kernel execution, main state transitions, and outer-loop (stage) transitions, states are needed to track data percolating through the latency of the datapath and perform control changes at the right times. This increases control complexity. Events to be timed include:

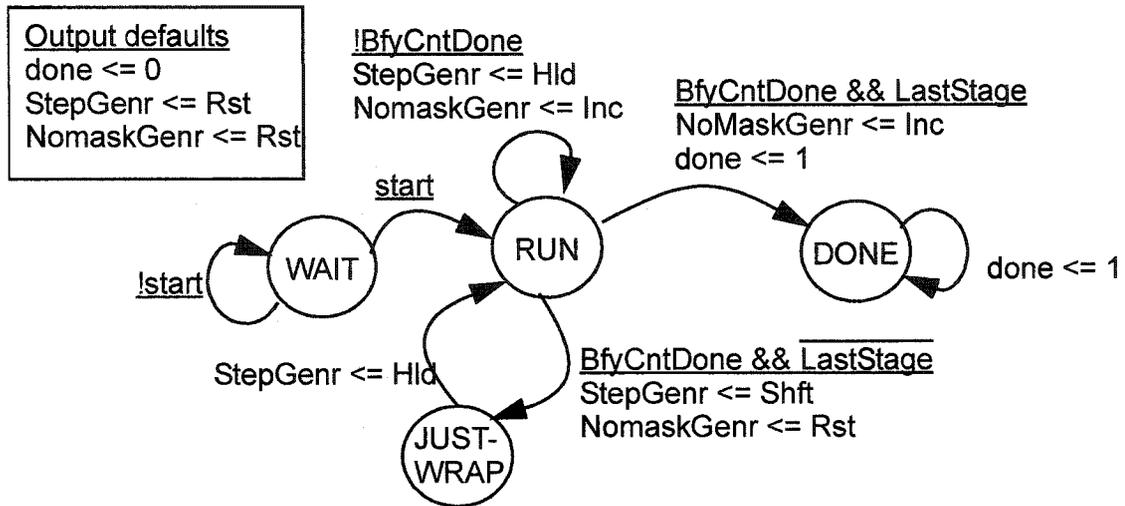


Figure 4.18. Example of a main state machine.

Verilog nonblocking assignments are used to emphasize the pipeline delays in the output. The *JUSTWRAP* state is an example of a transitional state to help time DPU personality changes as the first data percolates through the datapath in a new stage or major operational mode change. The datapath can typically have a high latency, making the use of shift register substate machines more efficient for a sequencing personality changes during changes in main state, or changes in outer loop iterations (Section 4.5.2).

- ◆ Start of butterfly count and twiddle address count
- ◆ Shifting of butterfly address generation mask and twiddle address step size at the stage transitions
- ◆ Transition from stage 1 to non-stage-1 behavior at various DPUs along the datapath e.g. butterfly address generation, whether to alternate between cross/through routing at the butterfly input/output (Figure 4.16b)
- ◆ Enabling memory writes only when the write data is valid.

Complication of control logic by main state transitions is a general digital design problem that is made prominent by high latency datapaths (due to pipelining). This is because the timing of different control signals needs to take place over more intermediate cycles between one major state and the next. The complexity is amplified by nested loops, or if one or more outer loops is handled

in a different manner than the rest (as stage 1 is here), because the control logic must accommodate more contingencies.

The structures used for substate machines must be PLA-friendly for processing streamed data i.e. they must *efficiently* use the limited resources of the control path's registered PLA to perform the specific task of mirroring the percolation of data through the datapath. This differs from classical state machines for ASIC design, which uses more general purpose binary counters to minimize state bit registers; there is much flexibility to reduce state decoding complexity during ASIC synthesis e.g. using more than 2-level logic. For the PLA, however, the logic must be rendered into sum-of-product form, which uses many product terms, which in turn seem to be in higher demand than registers. Thus, the programmability of the PLA precludes the optimization exploited by ASIC design for a fixed function.

Shift registers state machines can trade off product terms for registers (Figure 4.19(a)). Each wire between adjacent bits in a shift register requires one feedback to the PLA, taking one input and one product term i.e. the logic feeding each bit is simple; all logic complexity feeds the first bit to determine what enters the shift register. The idea is that this simple state encoding reduces the complexity for state decoding. This approach is applicable in general to platforms with PLAs in the control path, particularly if product terms and inputs are more precious than output registers. Various shift register machines were considered for use with the registered PLA.

In FPGAs, a similar departure from binary counters is the use of one-hot state machines, in which only 1 state bit is active. One-hot state machines make greater use of readily available registers, but have simpler combinational logic requirements. The simplest possible one-hot state machine is also a shift-register machine, one where a solitary 1 is sent through the shift register. However, the one-hot state machine's use of registers is somewhat excessive for the PLA (1 bit per state). This is due to a general characteristic of PLAs; the sum-of-product form proliferates product terms when mixing logic conditions. We use the "one-hot shift register" to illustrate the problem for DPU personality control, then discuss two state machine structures that mitigate the problem by requiring fewer product terms to encode entire swaths of consecutive state bits: Chameleon's walking bit state machine, and a walking window state machine.

Consider the frequent situation where certain DPU personalities take effect after a state i has passed. Control logic must then recognize the swath of consecutive states k that follow state i . For the one-hot shift register, this uses up one product term per state. To see why, consider that bit $k+1$ in the shift register is fed by the product term consisting of just bit k , which is high during state k ;

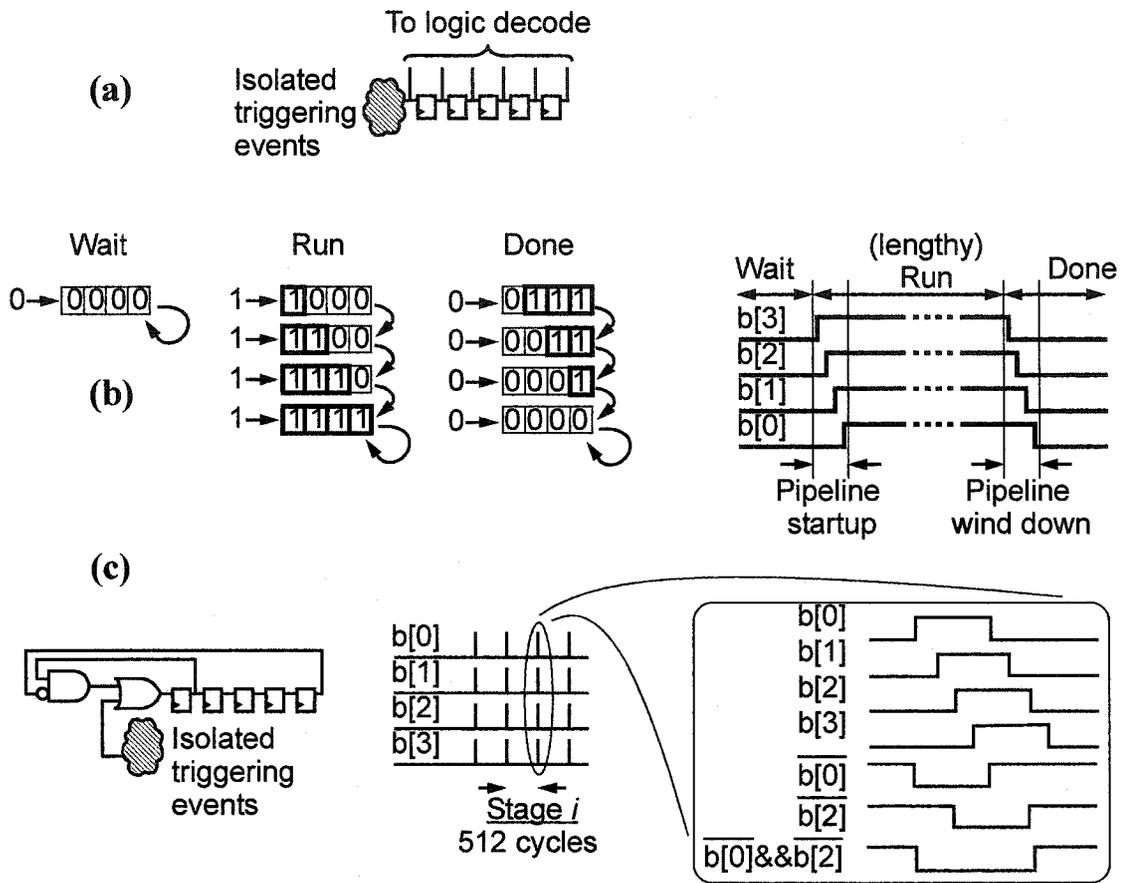


Figure 4.19. Shift register state machines.

(a) Basic shift register as a state machine.

(b) Chameleon’s walking bit substate machine. Binary ones are injected upon onset of kernel execution (*RUN* state in Figure 4.18) and binary zeros are inserted after entering *DONE* state. Single level loop iterations occur during the *RUN* state

(c) Walking window state machine. At each stage transition or start of outer loop iteration, the triggering logic injects a single binary one. The OR’d product term from the feedback keeps feeding ones only until the register is full. The pattern passing through the *N*-bit shift register is a window of *N* ones.

logic expressions that include this *exact* product term get it for free because it is already required by the one-hot state machine. Since this is a substate machine, however, expressions in state bit *k* almost always ANDs with specific main states (or other qualifying flags and logic conditions), thereby changing the required product term to one not already used by the shift register (Figure 4.20). Likewise, logic expressions that reduce to product terms involving \bar{k} cannot be

share those product terms with those making up the shift register. This leads to a large product term requirement.

4.5.2.1 Walking Bit State Machine (Modified Johnson Counter)

Figure 4.19 (b) shows Chameleon’s walking bit state machine, which uses fewer product terms than the one-hot shift register for single-level loops. Consider a DPU that must change from a passive or reset personality to a steady state run personality in the 3rd startup cycle at startup, then to back to a harmless “done” personality at 4th wind down cycle. The personality changes would be based on $b[2]||b[1]$. ANDing these two terms with qualifying conditions creates far fewer product terms than ANDing a swath of bits in a one-hot machine. Alternatively, if the changes happen in the 4th startup cycle and the 3rd wind down cycle, they will depend on $b[2]\&\&b[1]$, which is one extra product term; of course, multiple terms in the qualifying AND expression could lead to more product terms.

Like the one-hot state machine, this state machine requires one register per substate, or per latency cycle in the datapath, which can be quite expensive for high latencies. It is good for one level of looping because there is only one major transition into a steady state operational mode, and one transition out of it; the logic that feeds that shift register has a very simple dependence on the main state.

4.5.2.2 Walking Window State Machine

Figure 4.19 (c) shows the “walking window” substate machine used in this design, good for the 2 levels of nested looping. PLA logic injects a 1 upon any triggering event of interest, and feedback logic adds an extra product term to ensure that the whole shift register fills up with ones before refilling with zeros. This is suitable for outer loop transitions because nonzero disturbance is temporary i.e. it returns to the all-zero ready state so that exactly the same thing can happen on the next outer loop iteration. The one-hot shift register also has this property, but note that a 5-bit walking window machine gives 10 edges per triggering event, whereas a 10-bit one-hot state machine would be needed for the same. As in the walking bit machine, a swath of consecutive states can be monitored by a small number of product terms; in Figure 4.19, a DPU personality that should remain in effect for the duration of the outer loop iteration starting from the 7th transition cycle would depend on the single product term $\overline{b[0]}\&\&\overline{b[2]}$.

```

`TwidAdrsGenr_STATE_RUN : begin
//-----

// StagePipe
//-----

if ( BfyCntDone && !BfyCnt00b1 )
    StagePipe[0] <= 1'b1 ;

// TwidAdrsNomaskGenr_Sel
//-----

if (
    !StagePipe[0] || StagePipe[4]
    ||
    BfyIO_state != `BfyIO_STATE_St1
)
    TwidAdrsNomaskGenr_Sel <= `TwidAdrsNomaskGenr_Inc ;

if (
    !BfyCnt00b1 &&
    BfyIO_state != `BfyIO_STATE_St1 &&
    ( StagePipe[3] || StagePipe[6] || WayPastSt1 )
)
    TwidAdrsNomaskGenr_Sel <= `TwidAdrsNomaskGenr_Hld;

// TwidAdrsStepGenr_Sel & state
//-----

TwidAdrsStepGenr_Sel <= `TwidAdrsStepGenr_Hld ;

if ( BfyCntDone && LastStage )
    state <= `TwidAdrsGenr_STATE_DONE ;

if (
    BfyIO_state != `BfyIO_STATE_St1
    &&
    !StagePipe[4] && StagePipe[3]
)
    TwidAdrsStepGenr_Sel <= `TwidAdrsStepGenr_Shft ;

//-----
end // `TwidAdrsGenr_STATE_RUN

```

Figure 4.20. Example of coding for DPU personality control.

StagePipe is a walking window substate machine. For the logic expressions that govern the DPU personality selection (suffixed with “_Sel”), main state bits and substate bits are rarely used in isolation. State bits are usually combined with each other, or other condition flags, to form product terms that differ from those needed to form the state machines.

Another benefit of the walking window state machine is that its sequence of edges are triggered by solitary events, like a discrete-time one-shot. It is easy to specify solitary triggering events with logical expressions. Consequently, there is flexibility in the exact clock cycle in which the series of edges begin, upon a change in main state or outer loop iteration. In contrast, the walking bit state machine does not have feed back. It relies on cycle-by-cycle output from combinational logic which feeds it to ensure that the proper pattern of edges occur upon a change in main state or outer loop iteration. This is more arduous to ensure, and the more complex logic expressions makes it harder to fine tune the exact cycle in which the edges begin. It often makes more sense to put feedback around it and make it into a walking window state machine.

It may appear to conserve product terms to use a one-hot state machine because, in the simple case, every possible product term already exists to build the delay line. From the walking bit state machine example, however, it is seen that state encoding can easily require a complemented expression, which introduces product terms not used to realize the delay line. Therefore, the simple case cannot be trusted. More importantly, the expressions involving the state bits are rarely used in isolation. *Most* of the time, expressions in the state bits are ANDed with qualifying expressions (4.20) e.g. the main state bits and/or state bits to distinguish between first stage and non-first stage operation. This gives rise to product terms that contain these other signals and therefore differ from the solitary substate bits used to build the delay line. In this case, one-hot machine's need to use many state bits to specify a swath of time may actually be very harmful, if the logic is such that they are OR'd together (different product terms) rather than ANDed together (same product term).

4.6 Functional Verification and Resource Utilization

The final design before floorplanning took 19 out of 21 DPUs in the slice. Figure 4.21 shows the DPU personality changes at various times during the kernel execution. The twiddle address step is held constant throughout each stage and shifted at the stage transitions (*TwidAdrsStepGenr*). The twiddle address generator *TwidAdrsNomaskGenr* increments linearly in stage 1, then alternates between increment and hold because of the interleaved butterflies.

The mask that extricates the left and right half of the butterfly input data address (*BfyIPadrsMaskGen_Sel*) inverts every cycle after stage 1 and also shifts simultaneously between stages. Similarly, the masking operations (*BfyIPadrsMasker* and *BfyIPadrsMerge*) to insert binary 0 or 1 alternate every cycle after stage 1.

Interstage transition activity is initiated by *BfyCntDone*. The *Thru* configuration of Figure 4.16(b) is maintained in stage 1, then alternates between *Cross* and *Thru* because of the interleaved butterflies starting in stage 2.

The FFT kernel output was verified against Matlab's floating point FFT.

4.7 Placement

4.7.1 Floorplanning the Datapath

Figure 4.22 (right side) is an example of a floorplan for the 1-slice FFT. The tools may not always find a workable placement for complex and highly packed designs. This may be due to lack of local interconnect, or because changes are necessary to use global interconnect. The following are typical design challenges in trying to resolve this.

To run at 100MHz, the use of global interconnect in the datapath requires that the input of the destination DPU be registered, which is not a problem if the input register is already being used e.g. to permit a hold mode. Otherwise, the latency along that dataflow increases, leading to design changes to equalize delays, as well as nontrivial changes in the control path. The retiming required to pipeline global interconnect is not specific to programmable hardware, but it is more important for coarse grain platforms because of the lesser flexibility and potentially greater cost of retiming delays. Since a slice is a 1D array of processors, the datapath placement constraints in a 1 slice design make it all the more likely for signals to need global interconnect.

Sometimes, the input cannot be registered at the destination e.g. memory write DPUs tap the write data before the input register (Figure 3.10(b)). The input register might also be used to store a constant for use with the mask. In these situations, placement constraints must be specified to avoid the use of global lines for those connections.

If these DPUs are not free to move e.g. due to distances of other connections, then an additional forwarding DPU may be required to provide a halfway point between source and destination,

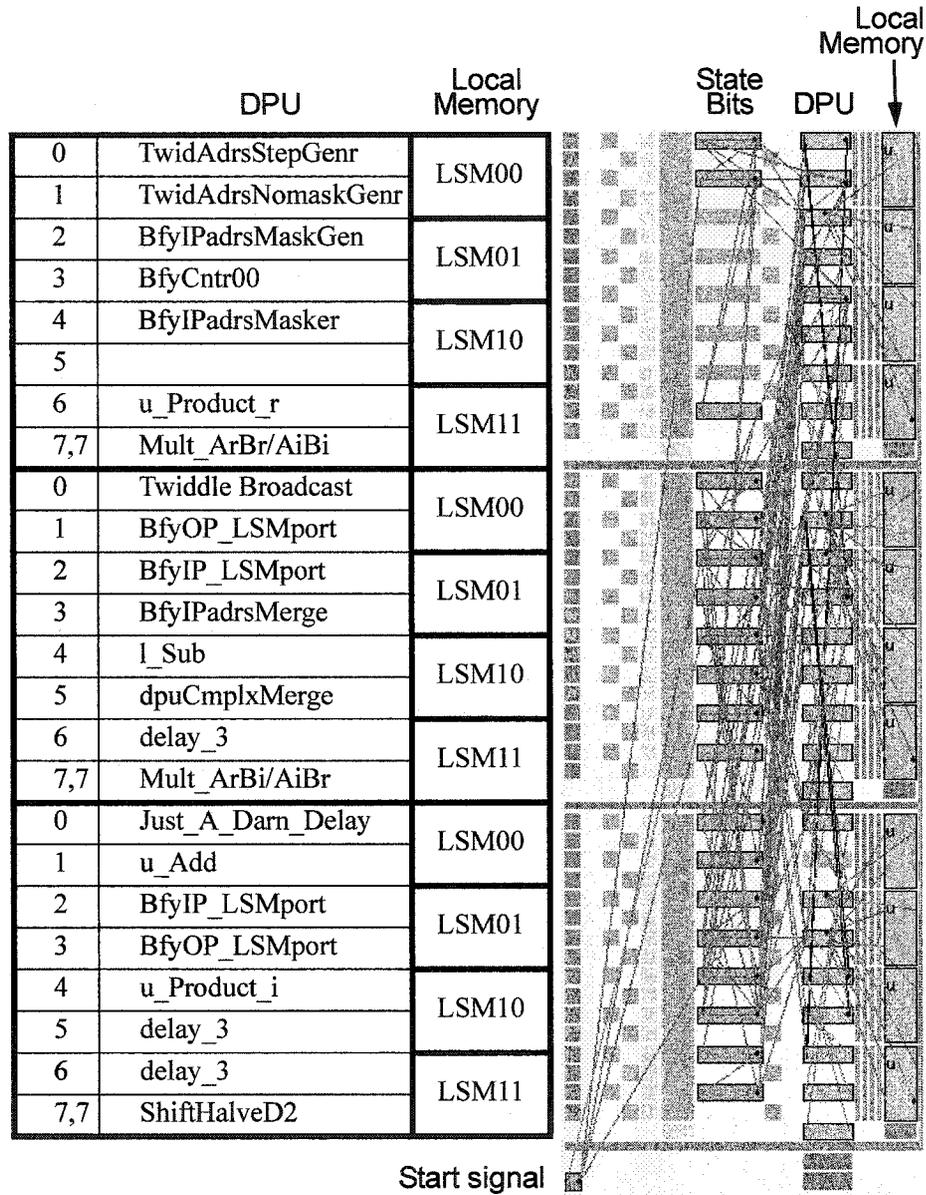


Figure 4.22. Floorplan view of the single-slice FFT.

which also introduces an extra cycle of delay and its associated rework. This is also subject to the availability of DPUs and positions in the vicinity of the halfway point.

As one might expect from a linear processor array, the middle area in Figure 4.22 is the most congested. DPUs in the middle tile are valuable because they can connect with most of the DPUs in the top and bottom tiles. Some of the DPUs in the top/bottom tile are there because of the mem-

ory e.g. read/write DPUs; DPUs that connect to DPUs in both top *and* bottom tiles have no choice but to be in the middle tile.

One may try to alleviate central crowding by placing DPUs that are fed by central DPUs away from the center, but they must still be within reach of the center to avoid using a global bus. Thus, DPUs *close* to the center also become precious. These DPUs vie for closeness to the center with read/write DPUs, which must also exchange data with processing DPUs near the center. Alternatively, DPUs fed by memory read/write DPUs may be pushed toward the top/bottom edge, assuming that their other input connections allow it, but those far-flung DPUs would be even less likely to be able to connect with the computational machinery in the middle via local wires.

Because forwarding DPUs are meant to bridge widely separated DPUs located in the upper and lower vicinity of the slice, they also need to be in the middle of the slice. Congestion may make this impossible, so forwarding DPUs are expensive and are not always an alternative to global wires.

The multitude of considerations make it attractive to have an automatable algorithm for placement. Unlike the much greater place and route task for ASIC design, the problem is small enough to manually find a placement in a matter of days or even hours. However, this cost would be incurred for every design modification, some of which may be required by rework to find a workable control path synthesis and placement.

Slice-Wise Partitioning and Central Crowding

Note that a fabric is not as restrictive as a linear array processor because kernels are not restricted to 1 slice. However, the more abundant local wires run along the slice, making it a natural partitioning element. Therefore, some central crowding is to be expected.

Final Resource Usage

In the datapath, the final design used 19 out of 21 DPUs, plus one wasted broadcast DPU -- 5 out of 9 vertical global buses were available.

4.7.2 Control Path Placement

Control placement follows control synthesis, which generates the minimized sum-of-products for the SRB bits (Figure 3.12). The latter is a fairly mechanical step performed by the vendor tools; it is the control placement which the designer must handle in such a way as to minimize usage of control logic. In this phase, a state bit is (ideally) placed by assigning it to a tile in such a

way that the PLA's inputs and product terms are not exceeded. There is more flexibility in control placement than datapath placement because half the state bits are globally fed to all the PLAs in the tile. In contrast to datapath placement, the assignment of state bits to tiles is determined more by the availability of PLA inputs and product terms rather than the reach of local interconnect and the availability of global interconnect. Furthermore, state-bit placement need only be specified to the level of a tile because PLAs are distributed by nature.

If the assignment of state bits to tiles was to be automated, one criteria is to choose a tile where the unplaced state bit shares the most product terms and inputs with state bits and personality selection bits already in the tile. For the unplaced state bit, inputs refer to the signals in its sum-of-products expression; inputs for a tile's PLA refer to all the inputs to all the SRB bits already in the tile. Before any state bits are placed, the tiles contain only the (nonmobile) SRB bits that select DPU personalities, since DPU locations are fixed before control placement.

Since the state bits were being placed manually, it was not tractable to tally up common product terms between SRB bits with potentially long boolean expressions; a state bit was put into a tile based only on commonality of input signals. However, commonality of input signals are a probabilistic indicator of commonality in product terms, which can be shared. Figure 4.23 highlights the differences in input signals between tiles using a text editor [Oua01], since input signals common to all tiles should have no bearing on attracting a state bit to be placed in a particular tile.

Because product term usage is not being directly considered, it is likely that initial state bit placements will cause resource overflow in one or more tiles after control synthesis and placement. Some state bits must then be migrated to other tiles. The criteria for selecting a state bit to migrate was similar to that for selecting a tile to assign to i.e. a state bit in the crowded PLA is considered for emigration based on commonality in inputs with its current PLA versus that of the destination PLA, ignoring inputs that occur in both PLAs.

This manual procedure still required that quite a few state bits be migrated before PLA resource limits were met. The most obvious reason is that commonality in inputs is only a probabilistic indicator of sharable product terms. A sanity check on the product terms showed that "bloated" main state bits in the twiddle retrieval module consumed an inordinate number of product terms in tile 0; emigrating other state bits one at a time had a piecemeal effect on reducing product term count. However, emigrating the bloated state bits would have overwhelmed the destination tile. They rightfully belong in tile 0 even though most of the remaining state bits had to emigrate, since

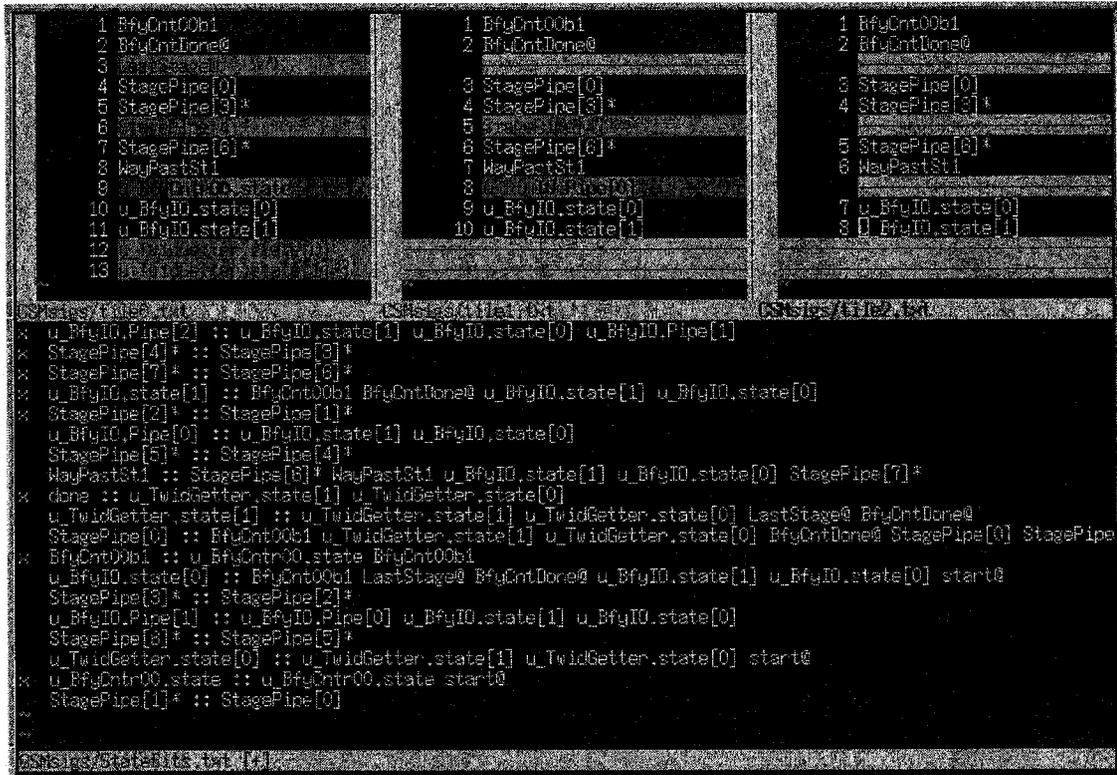


Figure 4.23. Supporting signals for SRB bits already in each tile, as well as unplaced SRB bits and associated dependencies being marked off as they become placed.

that is where twiddle retrieval circuitry resides. This suggests that bloated state bits should be placed first in automated placement.

This rearranging of state bits between tiles is a generalizable task in that any platform with PLAs in the control path can be expected to partition the PLAs into smaller, manageable size. This final balancing of PLA resources allowed the FFT kernel to successfully compile to fabric configuration bits. The slice resource utilization is shown in Table 4.3.

Table 4.3: Resource usage for FFT kernel.

Tile	DPU's (max 7)	PLA inputs (max 16)	Product terms (max 32)	Global Vertical Tracks (max 3)
0	7	14	31	1

Table 4.3: Resource usage for FFT kernel.

Tile	DPU (max 7)	PLA inputs (max 16)	Product terms (max 32)	Global Vertical Tracks (max 3)
1	7 ¹	14	29	1
2	6	15	30	2

¹One unnecessary broadcasting DPU
(plenty of global tracks to use instead)

4.8 Summary and Conclusions

4.8.1 Generalizability of Design Case

Chameleon's CS2112 can be representative of coarse reconfigurable platforms for DSP because it has all the different types of reconfigurability discussed. Further generalizable architectural features include separation of datapath and control path, hierarchal partitioning in each, local and pipelined global interconnect, and local and global control signalling.

Design tasks that can be general to coarse grain platforms include:

- ◆ DPU packing,
- ◆ DPU binding,
- ◆ retiming due to use of global interconnect, and
- ◆ placement of state bits among partitioned PLAs.

The greater importance of datapath frugality represents a shift away from ASICs because DPUs are conserved rather than gates. Trivial ASIC operations can consume DPUs nontrivially.

The features of the FFT which make it a good test case include 2-level nested loops and complex arithmetic. Furthermore, operations between nearest neighbor and widely separated data in memory exercises the platforms ability to accommodate a wide variety of data access patterns in one kernel through several means:

- (1) the capacity for control path complexity,
- (2) the flexibility with which local memories can be combined width-wise and depth-wise,
and

- (3) the flexibility with which array logic can be recruited to interleave access to different data banks.

4.8.2 Demonstrated Viability of New Paradigm

For many new computing architectures, the FFT is viewed as a proving case due to the broad spectrum of micro-functionality that it encompasses. Platforms and design methodologies can spring from elegant ideas, but their practicality for general designs is never certain until serious designs are undertaken by those who are expected to use them. Due to the novelty of the platform and the design paradigm, this design case required the establishment of numerous design practices. The fact that it was completed by people aside from the platform architects and designers demonstrates the viability of the array architecture, and the envisioned computational model and design flow.

4.8.3 General Observations and Summary Points

The final design used almost all the DPUs, PLA inputs, and product terms in each tile of the slice. PLA output registers and global vertical buses were not a bottleneck (local state registers were unused). With 4 slices running at 100MHz, the average kernel execution time over the whole device is 12.8 μ s/FFT. The control path design was of the same order of effort as the datapath.

Avoiding Overflow for Real and Complex Arithmetic

For a scaling in which the binary point is at the complete left, generating an $2N$ -bit product from N -bit multiplicands provides halving that combats marginal overflow. This has the same effect as bit-width expansion, or dropping LSBs in other scalings. This same benefit is realized when multiplying 2 complex numbers within, or on, the unity circle

As in real numbers, *marginal* overflow from addition is easily avoided if DPUs have saturated addition. Otherwise, a halving binary shift is required, which decreases dynamic range.

Alternative Kernel Implementation

The complexity of the FFT kernel was due in part to the nested loops i.e. all 10 stages were handled in one kernel. If each stage was a kernel, the datapath could be slightly simplified as the twiddle address step could be hard coded into a register rather than generated by a DPU. The bigger savings is expected in the control logic, since there would not have to be contingent logic to handle

the data addressing in stage 1 differently. The trade-off would be the reconfiguration time between FFT stages.

Central Crowding in Placement

In DPU placement, DPUs closer to the center of the slice are more valuable in a highly packed kernel because of limited reach of local interconnect. Central crowding can be expected for linear processors arrays in general when used for pipelined datapaths (as opposed to vector processing), though the fabric is not quite as restrictive. The high demand for central positions, can be alleviated by using vertical global buses and forwarding DPUs in some situations.

Disadvantage of Doing Two Real FFTs Simultaneously

Since the input record can contain complex data points, each kernel can simultaneously perform two FFTs on real records. However, to combine the data in order to do this, and to separate the data afterward, would cost a lot of real time, compared to the time it takes to run an FFT. This is because a kernel may need to be loaded to do this (software would certainly take too much time) and different data import/export calls may be needed. This is an example of how the speed of coarse grain arrays changes the cost/benefit balance. A “trick” that is worthwhile in software is not worthwhile in hardware due to speed differences.

4.8.4 Platform Architecture

Paucity of Datapath Registers

Datapath registers for delay lines are expensive in this architecture, which seems to favor only pipelining delays by embedding registers among the DPU logic. This points toward possible architectural enhancements by making available resources that can easily be used for delay lines¹. Such registers can also double as a register file within the DPU.

PLA-LUT Hybrid for Control Path

Because crossing state information from different state machines leads quickly to high product term usage, the fabric can benefit from the presence of control logic other than PLA logic. For example, register based LUTs can realize combinational logic of a given number of inputs with

1. Chameleon documentation provides a scheme to implement a circular buffer for this purpose using local memory and DPUs.

arbitrary complex expressions, as well as doubling as shift registers. Furthermore, approaches already exist to optimize the partitioning of combinational logic between PLAs and LUTs [KT03].

4.8.5 Design Methodology

4.8.5.1 Datapath

DPU Packing

To save DPUs, similar operations should be combined by using the upper and lower half-words of the datapath whenever possible. This fits well with the rectangular representation of complex numbers that are often found in modem baseband DSP.

Control path functionality should be piggy-backed onto DPUs where possible. Since the datapath logic aggregates many bits, this can save considerable amount of single-bit control logic.

Retiming

Delays can be conserved by pushing them through DPUs and [de]muxes so that they can be incorporated into a DPU's input register or so that they can be shared.

Memory banks, which are normally considered data sources/sinks, can also be regarded as DFG nodes through which delays can be pushed. In this case, the address is the input, while the data is the output (or input, in the case of a memory-write).

If there is a shortage of DPUs, unused multipliers also provide 2-cycle delays.

4.8.5.2 Control Path

Partitioning into Main State and Substate Machines

A generalizable approach to designing a PLA-based control path is to partition the control path into a state-bit-efficient main Mealy machine for gross time delineation, and PLA-efficient substate machines suitable for finely timed control of pipelined datapaths.

PLA-Efficient Substate Machines

Shift register substate machines were examined to mitigate product term count for the frequent case where swaths of substages need to be recognized. Chameleons walking bit state machine is good for 1 level of looping. For this design case, a walking window state machine was devised for 2 level nested looping.

Strategies for Control Path Placement

Unlike datapath placement, the global routing of every other state bit throughout the slice ensures that control path placement is not limited by routability. The resource limit is likely to be product terms or PLA inputs. A manual approach is described to place state bits into tiles based on maximizing the sharable PLA resources, with an eye toward automation. For the manual method, product term overlap between PLA output bits is estimated based on overlap in PLA inputs in the expression for each output. A similar method is used to select PLA output bits to migrate to other tiles if a particular PLA's resources are exceeded.

Based on manual placement of the control path, we suspect that a good starting strategy is to place complex sum-of-product expressions first.

4.8.6 Take-Away Points

- ◆ As a coarse grain platform with many generalizable features, the Chameleon architecture and design paradigm has been proven based on a realistic kernel that has many of the complexities and characteristics of stream data in telecom.
- ◆ As with other DSP hardware, designers must take care to avoid numerical artifacts
- ◆ The high speed of reconfigurable hardware changes some assumptions of what is advantageous e.g. doing 2 real FFTs at once is not advantageous
- ◆ The targetability of coarse grain arrays can be improved by incorporating more datapath registers than just those for pipelining, and by complementing PLAs in the control path with LUTs.
- ◆ Coarse granularity makes DPU packing critical. In addition to using both upper and lower half-words e.g. for complex numbers (architecture permitting), much of this packing involves pushing delays through DFG operators and memories.
- ◆ For pipeline realizations of nested loops, the control path can be efficiently designed for a PLA by partitioning the logic into a simple main state machine, and several substate machines based on delay lines.
- ◆ A heuristic approach was devised to place control bits so that product terms are shared. For the manual approach only, sharable product terms are estimated from commonality in inputs.

-
- ◆ Due to the many positional constraints encountered in manual placement, an automatable algorithm would be useful to explore DPU placements. Ideally, it would be able to handle the any design modifications that may be necessary in order to use global buses or forwarding DPUs. Such a tool would also be very useful if the kernel is subject to revisions in functional specification.

5. The Placement Problem and Comparison with Current Approaches

The FFT kernel design case illustrates the synthesis and physical design phases of mapping. The phases occur in the parallel development of the datapath and control path. For the datapath, the task can be viewed as the transformation of a dataflow diagram to a netlist of DPUs, followed by the binding of DPUs to site-specific locations. In the control path, the task is to devise logic to control the datapath behaviour, followed by the binding of this control logic to site-specific resources on the platform. For the Chameleon platform, this reduces to the assignment of control bits to the PLAs of the different tiles.

In this design flow, the mapping of an algorithmic dataflow diagram (DFD) to a netlist is significantly more difficult than the physical design problem. The place/route problem is a constraint satisfaction problem due to the limited degrees of freedom in a prebuilt platform. This chapter introduces the use of *genetic algorithms* (GAs) for the simpler problem of placement. In the Chameleon platform, there is full connectivity between DPUs within 8 rows of each other. Hence, much of the place and route problem reduces to a placement problem. If a good placement can be found, a connection scheme is guaranteed.

This chapter also discusses how placement for Chameleon-like architectures differs from the highly eclectic platforms and placement approaches surveyed in Chapter 3. As well, we discuss how the use of GAs found in general placement problems are not suitable.

5.1 Problem Formulation

5.1.1 Challenges Due to Interconnect

There are cases in which a good placement cannot be found. As noted in the FFT design case, central crowding is a problem. The vertical global buses exist to permit placements in which connected DPUs are not close enough to each other. This feature comes with a price; data that travels along these buses are delayed by a cycle because registers are required at the input of the destina-

tion DPUs in order to meet timing constraints. Since the kernel is a pipeline of parallel datapaths, this could necessitate nontrivial changes in other dataflows in order to maintain synchronicity. It may also involve nontrivial changes in the control logic to accommodate the retiming.

One way to avoid the retiming is to have the placement algorithm favour the use of vertical global interconnect in cases where the input register is already being used at the destination. By doing this, the connected DPUs may have more freedom to be placed far from each other, thus freeing up central positions for connected DPUs which do not use input registers at the destination.

5.1.2 The Placement Problem

Given that the fabric is partitioned into linear arrays (slices), placement of the kernel or subcircuit within a slice can be regarded as a problem in ordering the DPUs into a linear array. This linear DPU ordering determines the routability in a much more direct manner than placement in ASIC or FPGA design. This is because of the complete interconnection of DPUs within 8 positions of each other; therefore, whether two DPUs are connectable is determined only by whether they are positioned within 8 DPUs of each other. (We initially ignore the use of global lines or “do-nothing” DPUs for forwarding signals over greater distances).

Note that the predefined infrastructure of the coarse grain platform reduces the place and route problem to a problem of constraint satisfaction. However, this is a simplification of the entire placement problem, for the purpose of establishing a baseline algorithm. Other platform characteristics must eventually be incorporated. Specifically, an arrangement of DPUs is sought such that features at each site meet the requirements of the DPU bound to that site, and the required interconnect is available. Site-specific requirements are now discussed.

Proximity to Resources

The candidate sites for a particular DPU may be subject to a number of requirements. In the FFT example, the 2 data banks for the butterfly data and the single data bank for the twiddle factors occupy contiguous memory blocks that naturally fit into a tile. The DPUs that read and write these memory banks must reside in the same tile as the data banks. This can be generalized to platforms beyond the Chameleon CS2112. That is, resources that provide access to distributed local memories should be placed in physical proximity to the local memories they access.

Heterogeneous DPUs

Other constraints on DPU placement may depend on the functions available within each DPU, rather than its proximity to resources like memory. For example, only DPUs on even rows can read from local memory; conversely, only DPUs on odd rows can write to local memory. Likewise, only the last row in each tile can perform multiplication (in fact, they contain two multipliers rather than a DPU). This can be generalized as follows; not all functionality is replicated in all function units of a reconfigurable platform.

If this heterogeneity of DPUs is accommodated in the placement algorithm, it has tremendous practical implications for the platform architecture. It is prohibitively expensive to replicate *all* possible functionality in *all* DPUs. It is far preferable to intersperse DPUs with complementary functions and rely on local fine-tuning of the placement to match an operation with an appropriate DPU. This approach is attractive for coarse grain function units because the place and route problem (and thus the fine-tuning of the placement) can be an order of magnitude smaller than for a fine grain. Furthermore, the specialization of coarse grain DPUs for specific functions occur at a higher level, thus representing a larger commitment of resources.

5.1.3 Mathematical formulation

Mathematically, the problem is as follows (refer to Figure 5.1). A DPU netlist is a directed hypergraph (V, E) , where each net $e_i \in E$ consists of source DPU $v_i^S \in V$ and one or more destinations $\{v_i^{D1}, v_i^{D2}, \dots\} \subseteq V$. The coarse grain array is a vector of DPU positions $(p_1, p_2, \dots, p_{N_{\text{DPUs/slice}}})$. To allow gross position constraints, contiguous p_m are further grouped into abutted tiles $T = \{t_1, t_2, \dots\}$ such that $t_k = \{p_{l+1}, p_{l+2}, \dots, p_{N_{\text{DPUs/tile}}}\}$, where $l = (k-1)N_{\text{DPUs/tile}}$. Each v_i may be required to reside within a subset of adjacent tiles (1-tile subsets in this case). We assume that the array interleaves N_{Classes} classes of DPUs with varying capabilities ($N_{\text{Classes}} = 2$ for Chameleon). A v_i may also be required to reside in certain classes; hence $\{p_m\}$ are also grouped into classes $C = \{c_1, c_2, \dots, c_{N_{\text{Classes}}}\}$, where $c_k = \{p_{l \cdot N_{\text{Classes}} + k} \mid 0 \leq l \leq N_{\text{DPUs/slice}} / N_{\text{Classes}}\}$. Under this notation, the problem is then to arrange the v_i 's onto the p_m 's such that (1) there are no more than $N_{\text{GlobalWires/tile}}$ nets being sourced from each tile to destinations more than $l_{\text{LocalWire}}$ away, and (2) any memberships of each v_i in T and C are met.

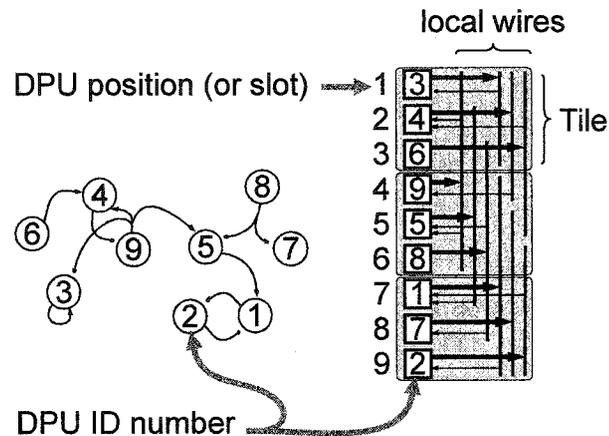


Figure 5.1. DPUs in a netlist are ordered onto a slice.

The required connections should be mostly realizable with the limited reach of local wires. *Example architecture* has 3 DPUs/tile and 3 tiles/slice. Local wires extend up 3 rows and down 2 rows, except near the top and bottom. *Actual slice is much larger* (see Figure 4.22).

5.1.4 Simplified Archetypal Placement Problem

To motivate the discussion on solution methods, we consider the generic, reduced ordering problem. Only the availability of local wires is considered. A DPU's position does not depend on proximity to memory or whether or not there is multiplication. Therefore, connected DPUs need only be within 8 rows of each other. The availability of global wires is ignored. The placement problem becomes a rudimentary ordering problem.

Combinational Problems

The ordering problem falls in the category of combinatorial optimization problems. The distinguishing characteristic of these problems is that the solution space consists of a finite number of discrete solutions; there are only so many arrangements of DPUs that are possible. However the finite number is often much too large to exhaustively try each solution. For example, consider Chameleon's architecture with 9 DPUs (including 2 multipliers) per tile, 3 tiles per slice. That represents $27! \approx 10^{28}$ possible arrangements of netlist components onto the DPUs of a slice.

Gradient Methods

Combinatorial problems can be contrasted with continuous-parameter problems such as finding the inductances and capacitances of a matching network to give the best broad band matching. The continuity of the solution space allows for algorithms to use the gradient of the cost function at some solution point to decide on the direction in which to search next. Attempts to use gradient-based search methods for combinatorial problems usually give poor results. This is because the continuous-parameter solution must be discretized to the nearest discrete solution, which cause a large change in the cost function. In the case of an ordering problem, the result can be even worse because the nearby solution points may have cost values that are completely unrelated, thus rendering the notion of local gradient irrelevant.

For example, assume a cost function for Figure 5.1 by penalizing over-length wires according to some scheme. In this case, there is not even a continuous-parameter counterpart i.e. there is no fractional DPU#. One might attempt to find a discrete “gradient” by perturbing the solution e.g. by replacing DPU#3 with DPU#4 in position#2. Since the ID number for the DPUs are arbitrary, DPU#3 and DPU#4 likely have no similarity in their interconnections, and the disparity in wire-length violations and cost values may be drastic. For this reason, the gradient does not give any information about how to perturb the solution point to obtain a better solution.

Min-Cut

For the DPU placement problem, one heuristic method that was considered was the min-cut bipartitioning algorithm [KL70]. Min-cut attempts to find a good way to split a graph into two equal halves such that the cost of connections between them is small. Because it works by refining an initial bipartition, it is an iterative method, though it lacks the problem-independent framework of the search space methods described above. The idea was to partition the linear DPU array into two halves, then recursively split each half into smaller halves.

Despite its simplicity and elegance, min-cut was discarded as a candidate solution for several reasons. First, it assumes a graph where each edge is point-to-point. In contrast, actual netlists consist of nets that connect many components. Thus, netlists actually correspond to *hypergraphs*, which consist of nodes (components) and *hyperedges* (nets). A graph is not even a good approximation of a hypergraph, since most of the nets were hyperedges. This is a recognized limitation in min-cut [MR99]. It was not, however, the deciding factor, as algorithms exist for hypergraph partitioning (e.g. [KAKS97, KK99, CW03] and references in their introduction).

The deciding factor to seek out more flexible algorithms was because the goal of partitioning does not exactly match the actual constraint imposed by the placement problem. Partitioning minimizes the number wires passing through any cut of the linear DPU array. The actual constraint is that any wire should not exceed the distance of 8 rows. For this reason, we sought a search space method using the more generic frameworks. This allows more freedom in tailoring the constraints of the search algorithm to match the problem.

5.2 Distinguishing Features of Placement for Chameleon-Like Architecture

Three major features that distinguish the Chameleon placement problem from most physical design problems is (1) simultaneous placement and routing, (2) the hard limit on the maximum acceptable wire length (8 rows), and (3) the heterogeneous placement constraints e.g. due to the proximity of certain memory banks, or the differing capabilities of the DPUs. This has several implications for placement and routing.

5.3 Choosing GAs for the Iterative Search

Implications Against Deterministic Heuristic Search

As mentioned, the hard limit on wire length is essentially the same as the simultaneous placement and routing. If the placement is such that no nets exceed the limit, the routing is automatic. This pass/fail characteristic has implications for how the placement is sought. Nonstochastic heuristics that search a *deterministic* subset of the search space will return a best-effort solution. Placement problems that don't have a hard boundary on acceptability can tolerate such solutions even if they don't happen to be very good. For problems with hard limits on acceptability, a bad solution is simply unacceptable, and the deterministic nature of the search does not allow another solution to be found. In this case, stochastic searches may be preferred for their ability to potentially visit any point in the search space.

GAs vs. Simulated Annealing

Of the 3 search space methods discussed in Section 2.2, this leaves simulated annealing and evolutionary algorithms as preferable approaches. The motivation for choosing GAs as the main

approach was their theoretical implicit parallelism (Section 2.5.3), as well as the potential for parallel processing. As a promising alternative, efforts were also devoted to simulated annealing for comparison purposes (Section 8.4.3).

5.4 Implications for Cost Structuring Against Clustering DPUs

The implications against heuristic search has to do with what happens when pass/fail thresholds are *not* met. However, such thresholds also have implications for how a search should be conducted in regions where the thresholds *are* met. For example, many conventional placement algorithms cluster logic blocks together by imposing a monotonic penalty based on wire length (or estimate thereof, since actual routing is not typically performed during placement). This is not justified if the platform architecture is such that all wires that comply with the limit are equally good, regardless of exact length.

One might first assume that it doesn't hurt to cluster interconnected DPUs closer together, and that the monotonic penalty for legal wire lengths is beneficial, or at least harmless. In fact, manual mapping shows that it can be problematic for the DPUs of a subcircuit to be placed close together. DPUs on either side of the cluster have to reach further to connect to the other side. In these cases, it can be better to intersperse DPUs from subcircuits that don't connect to each other very much. This lets the DPUs from one subcircuit connect to each other by leap-frogging over the DPUs from the other subcircuit, and vice-versa.

The case against clustering interconnected DPUs is unintuitive at first. It is these unpredictable interactions that *rationaly* premised heuristics are vulnerable to. This is why GAs are attractive, to stochastically find the right building blocks and proliferate them.

5.5 Comparison With Placement for Surveyed Platforms

Physical design for the surveyed platforms is discussed in Sections 3.2 and 3.3, and summarized under "Resource Binding" on page 53. Chameleon is compared with them next. None of

these methods seem to address the 3 features above that distinguish the Chameleon placement problem from others.

There seem to be similarities between Chameleon and Garp; both stack operators vertically, have local wires with unit stride, and global wires. One might be tempted to conclude that their placement algorithms can be similar. However, their placement problems are quite different. The Garp placement problem does not seem to have the hard limits on wire length that Chameleon has. The heuristic placement gives a best-effort result, as described above, and the user has to take it or leave it. In contrast to the constraint driven problem in Chameleon placement, the Garp placement heuristic attempts to solve a true optimization problem, since delay is one of the parameters considered in the cost minimization.

Due to similarities in architecture and placement between Garp and Chimaera architectures, it is not surprising that Chimaera placement is also unsuitable for Chameleon. Like Garp, Chimaera's placement heuristic does not seem to consider hard limits in the separation between operators. Chimaera's placement has the freedom to evict operators to the next row if the required routing cannot be realized during row packing. It seems that the solution found is best-effort, with potentially long delays. Note also that Chimaera is exploiting fine granularity by packing operators into rows, which Chameleon cannot do.

Both RaPiD and Xputer use SA for placement. SA may also be suitable for Chameleon placement. This is beyond the scope of this thesis (though it is being investigated outside of the thesis; see Section 8.4.3 for preliminary results). Unlike Chameleon, RaPiD's routing is done after placement, using a deterministic heuristic.

PipeRench's placement seems to have very little in common with Chameleon placement. Chameleon's linear ordering of DPUs bears no resemblance to PipeRench's packing of operators into rows of DPUs, with full cross-bar between rows.

5.6 Comparison with GAs in Other Placement Problems

GAs are used in many placement problems. Most of these problems deal with placement for ASICs. Most do not describe any heterogeneous placement constraints such as those that can be expected from a coarse grain prebuilt platform. For most, routing happens after placement, and

the placement phase merely tries to minimize a cost based on an empirical formulae that estimate routability, area, or some other metric. Estimates based on empirical formulae are appropriate when the platform allows for fine adjustments in placement, routing, and porosity (how densely to pack the logic). For coarse grain arrays, such estimates can be expected to be inaccurate due to the lesser freedom with which logic can be positioned, spread out, or routed (analogous to quantization noise).

Here are some example of GAs used for placement. Differences with the Chameleon placement problem are pointed out.

A GA is used in [BMSK03] for FPGA placement. It seems to be an ordering problem only, with no positional constraints. No limits on wire length are imposed, and routing is not done. Fitness is based on empirical formulae for estimating area and routability.

In [YTFY03], a GA is used for ASIC placement. Each cell to be placed has gross and fine positional specifications. Fitness is based partially on formulaic estimated delay. Unlike Chameleon, logic is not necessarily registered. This complicates delay calculation. This may be the reason why all path delays in a solution are only calculated every T generations. In between, the delay for only the worst few paths (as identified every T generations) are recalculated every generation. The GA also uses a line cutting method to quantify congestion. Recall that min-cut works similarly, and was abandoned in Section 5.1.4. Congestion was not the best quantity to control or monitor in Chameleon placement.

In [MM02], a GA are used the placement of ASIC standard cells and macro-cells. Macro-cells are flexibly-shaped groups of standard cells. Placement is made tractable by partitioning standard cells into macro-cells, placing the macro-cells, then placing the standard cells within the macro-cells. Area and wirelength estimates contribute to fitness. There are no heterogeneous constraints. Most of the genetic complexity seems to be devoted to ASIC-specific details, such as hierarchically representing the macro-cell, number of standard cell rows in each macro-cell, orientation, etc..

A GA is also used in [SV97] for macro-cell placement. The problem addressed is the fact that the macro-cell aspect ratios are not known until after the macro-cells are placed, and the standard cells inside the macro-cells are placed. The genotype contains information for partitioning a layout area into macro-cell compartments, and various candidate sizings for the compartments. A clustering process ensures that tightly connected macro-cells are close together, thus accounting

for routing. These features do not seem to fit the Chameleon placement problem. Again, no mention is made of heterogeneous location constraints.

5.7 Summary and Conclusions

- ◆ Due to the strongly defined interconnect infrastructure, gross and fine nonuniformities in the distribution of resources, and predefined clock, the place and route problem is one of constraint satisfaction rather than strict optimization.
- ◆ Iterative algorithms are favoured because they adapt easily to platform specific features and are robust against vagaries in problem encoding.
- ◆ GAs were selected for their implicit parallelism and amenability to parallel processing.
- ◆ Chameleon placement differs from other platforms and other uses of GAs for physical design in the following ways:
 - (1) Simultaneous place and route
 - (2) Binary pass/fail for both the entire genome and individual constraint compliance
 - (3) Emphasizes interconnect distance rather than congestion
 - (4) Heterogeneous position constraints
- ◆ The binary pass/fail nature of solutions favours a stochastic search.
- ◆ Min-cut (and partitioning algorithms in general) are not suitable for placement because of the pass/fail threshold for wirelength, in contrast to a graded acceptability based on congestion.

6. Developing the GA for Placement

Section 5 discusses how the Chameleon placement problem differs from the physical design problems addressed in the surveyed reconfigurable platforms. It also points out fundamental differences with the general placement problems for which GAs are typically used.

In this chapter, we discuss problem-specific details and options for applying GAs to the placement problem described in Section 5.1 (please refer to chapter 2 for general GA background). The development of the GA for the placement problem represents the major contribution of this thesis. We note that the platform constraints are representative of limitations that can be expected of real life platforms in general (as opposed hypothetical architectures).

The placement problem is illustrated in Figure 4.22 on page 91. The GA is constructed in two steps. First, in Section 6.1, a GA is developed for the *simplified DPU ordering problem*. Then, in Section 6.2, the heterogeneous constraints from the platform architecture are incorporated, thus developing the GA for the *full placement problem*.

In developing the GA for the simplified placement problem (Section 6.1), problem-specific design issues need solving, in addition to the basic GA components described in Section 2.5. The GA framework of Figure 2.5 is presented in Figure 6.1 in greater detail. These issues are discussed next, and include the following.

1. Determining a suitable encoding of candidate solutions into chromosome form (genotype).
In the DPU placement problem, the genotype must encode a permutation of DPUs. The encoding should minimize the parts of the search space containing grossly infeasible solutions.
2. Related to the encoding scheme is a suitable crossover method (used in Figure 6.1, step 4).
In the placement problem, it must preserve the permutation nature of the solution.
3. An approach to handling violations of architectural constraints by candidate solutions (genomes) e.g. violations are minimized by the encoding, repaired, and/or penalized during the evaluation. This is used in Figure 6.1, step 6. For the simplified placement, violations

1. Initialize random population of solutions (genomes).
2. Loop
 3. Select parents according to fitness.
 4. Perform crossover on parents to generate offspring.
 5. Evaluate costs of offspring.
 6. Repair violations if possible.
 7. Penalize outstanding violations.
 8. Generate mutants by perturbing some solutions.
Evaluate their costs (same as steps 5, 6, and 7).
 9. Replace current population members with offspring and mutants using suitable scheme.
10. Stop when a valid solution is found or if the maximum search time is exceeded.

Figure 6.1. GA framework of Figure 2.5 in greater detail.

consist of local wire length violations and, if the encoding permits, solutions that don't represent permutations of the DPUs in the netlist.

4. A cost scheme for the violations (used in Figure 6.1, step 7).
5. A scheme for converting overall cost to fitness. This conversion can be done for each genome immediately upon assessing its cost, or it can be assessed across the entire population, one generation at a time, as in ranked fitness (Section 6.1.4.3).
6. A scheme with which to replace population members with kids (Figure 6.1, step 9). This includes the mutant to offspring ratio, whether kids unconditionally enter the population regardless of cost, and which genomes they replace.

Figure 6.2 provides a flowchart for many of these components, for the case of *random keys encoding* (Section 6.1.1.2). In addition to these problem-specific features, Section 6.1 also presents a method for gauging genetic diversity in the population.

For the full placement problem (Section 6.2), the additional complexity in constraint handling necessitated a move from Matlab to C++. The violations handled in issues #3 and #4 above are expanded to include tile bound DPUs, polarity-bound DPUs, and the use of global wires to reduce

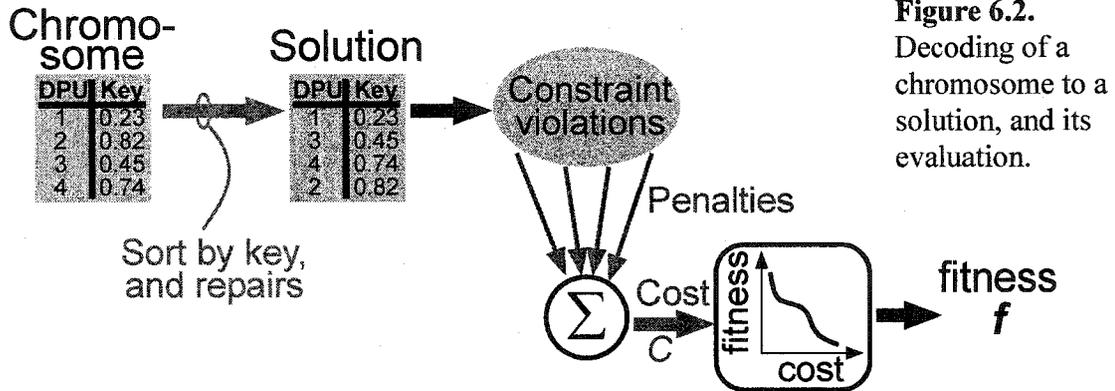


Figure 6.2. Decoding of a chromosome to a solution, and its evaluation.

violations of local wire length. Figure 6.31 provides a flow chart of how these violations are handled, while Figure 6.32 provide a bird's eye view of the overall GA.

6.1 Simplified GA for the Basic Ordering Problem

For the *basic, simplified ordering problem* of Figure 5.1, the simplest way to encode an arrangement of DPUs is to treat the DPU slot positions as the genes (Figure 6.3). Each gene is able to take on the value of a DPU ID number which occupies the associated slot. A chromosome consists of an assignment of all DPUs to all the DPU positions i.e. a permutation of the DPU ID numbers. A *population* consists of a set of such assignments, initially generated randomly.

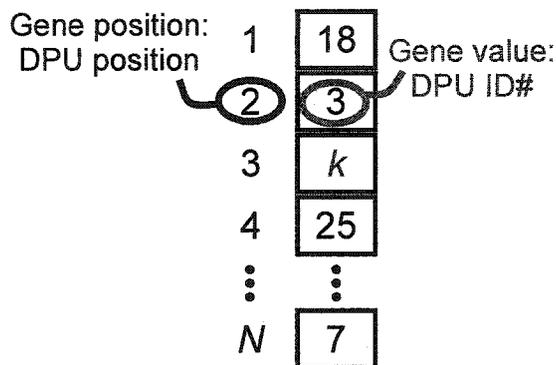


Figure 6.3. Chromosome for the basic ordering problem.

Note that this is not the only way to encode the gene. An alternative *random key encoding* is used in Section 6.1.1.2, which avoids some of the problems with this basic gene encoding, and simplifies mechanisms for enforcing heterogeneous constraints.

6.1.1 Gene Encoding and Crossover

For iterative optimizers, ordering problems form a special subculture. The distinguishing feature is that parameters are not independent. Each DPU index number must occur once in the chromosome, and only once. This means that the simple crossovers will not work, since they can easily yield an offspring that is not a permutation of all the DPU ID numbers. We will refer to the repeat occurrence of a DPU as a *collision* (Figure 6.4). If a collision occurs, it is clear that not all DPU ID numbers are present in the chromosome.

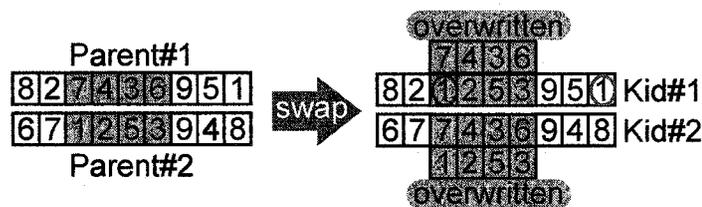


Figure 6.4. Two-point crossover applied to an ordering problem. Numerous collisions are created e.g. kid#1 has a collision in DPU#1.

The various crossover schemes for ordering problems have typically been developed using an archetypal ordering problem known as the *travelling salesman problem* (TSP). In this problem, a salesman must visit all the cities given in a list. The cost of a trip between two cities depends on the two specific cities. The goal is to visit all the cities in an order that minimizes the total cost. When applying GAs to the TSP, the genes represent cities rather than DPUs.

[Gol89,GC97,Mic96] describe a number of ways to combine genetic information from two parents while avoiding collisions. The various crossovers preserve different kinds of information, including the absolute position of DPUs, the adjacency of DPUs, and the nonadjacent relative order of DPUs. Many of the crossover schemes target the TSP problem; they take advantage of rotational invariance i.e. {1,2,3} is taken to be the same as {2,3,1}. Such schemes are not suitable for the placement problem, since the topmost DPU cannot connect to the bottom most DPU with-

out using valuable global interconnect. The availability of global interconnect will be factored into the problem later in a more directly controllable manner. From the manual placement effort in Chapter 4, it was clear that favourable arrangements of subsets of DPUs were embodied in both relative and absolute DPU positions. Another reason for maintaining absolute position information is that, in addition to connecting with other DPUs, the final solution must deal with connections to fixed resources like local memory. A few basic crossovers suitable for the linear placement problems are introduced next.

6.1.1.1 Partially Matched Crossover

Partially matched crossover (PMX) is a crossover scheme for ordering problems that preserves absolute position and adjacency information *within* the 2-point crossover region. This information is partially preserved to varying degrees outside of the crossover region. After performing a regular 2-point crossover, PMX applies a repair algorithm to change the genes outside the crossover region in order to remove collisions with genes inside the crossover region. One drawback of PMX is that the outside region is left scrambled to varying degrees, like a gross mutation, thus limiting the transmission of genetic information to the offspring.

PMX's repair algorithm works as follows. Figure 6.4 shows a collision between two instances of DPU#1 in kid#1, which will now be resolved. The instance inside the crossover region is to be kept. Since it overwrote DPU#7, the offspring is potentially missing DPU#7 (unless it occurs elsewhere in the crossover region). The duplicate DPU#1 in the exterior region is changed to DPU#7 in hopes of replacing a missing DPU. The interior region is then scanned for DPU#7 to detect a 2nd collision. If it is not detected, then the collision in DPU#1 has been resolved. Otherwise, the repair procedure must be repeated to resolve the new collision in DPU#7. The repair procedure is repeated until no collision is detected. This repetition of the repair only resolved 1 of the collisions that resulted from crossover. It must be performed for each gene inside, or outside, the crossover region.

Note that the ring representation of a chromosome is very important for PMX because the interior and exterior of the crossover region are not treated symmetrically. Only the interior is guaranteed to remain intact. The ring representation allows this valuable interior region to straddle the ends of the chromosome. Thus, any building blocks at the chromosome extremities have an equal opportunity of being preserved.

6.1.1.2 Random Keys Representation

Random keys representation (RKR) is not simply a crossover method. It is a different way to encode the problem to simplify the standard 2-point crossover for ordering problems. This is the example initial presented in Figure 6.2. In most chromosome definitions for ordering problems, the gene position represents the order, and the gene value identifies the object (task, city, DPU) to take place at a specific time or position. In random keys, each gene position is associated with a *DPU*, and the gene value represents the *position* of the DPU *in a vague or sloppy way*. The sloppy specification of position is important for avoiding collisions. If each DPU is simply assigned an integer that represents its position number, then the mechanical details would be identical those in Figures 6.3 and 6.4; each solution would have to be a permutation of the position numbers, and collisions due to crossover must be repaired.

The random keys representation assigns a random real number to each DPU, arbitrarily in the range $[0,1]$ (Figure 6.5). The chromosome is converted into a DPU ordering by sorting the genes according to the random key. Thus, a low key value (for example) means that a DPU is more likely to have a low-numbered DPU position. Regardless of whether the key values are arrived at through random generation or crossover, there is negligible likelihood that two DPUs have the same key value. Collisions are thus avoided. The trade-off is that a sort is required to obtain the ordering for evaluation.

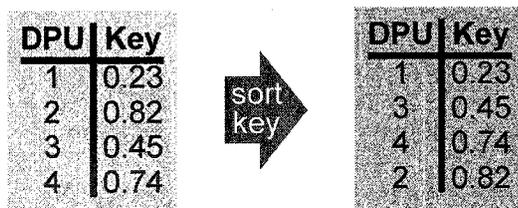


Figure 6.5. Obtaining an ordering from a chromosome under RKR.

One consequence of RKR's beneficial sloppiness is a DPU's key does not by itself indicate the DPU's position. It is the key value, in the context of all the other DPUs' key values, that determines the DPU position. Likewise, a change in the key value of one DPU can potentially change the position of other DPUs. Thus, the position information for any and all DPUs is distributed over the entire chromosome.

The distributed position information complicates some of the metrics used in controlling the GA. For example, some chromosomes can be completely different in their key values, yet gener-

ate the same solution when the keys are sorted. Thus, it is not straightforward to recognize repetition in the kids. It is not a good solution to compare the DPU orderings either. For example, a DPU with key value 0.7 may translate into DPU slot#20 in a parent chromosome, but not necessarily in a child chromosome. Thus, two chromosomes c_1 and c_2 that give the same DPU ordering do not necessarily generate the same children when each is mated with some other chromosome c_3 . Therefore, c_1 and c_2 cannot be treated as identical.

Even if one ignores the degeneracy that arises when decoding the chromosome by sorting, RKR is still a continuous parameter encoding. It inherits some of the issues that accompany GAs when applied to continuous parameter problems. It is difficult to recognize uniqueness or devise a measure of population diversity when “unique” solutions can differ in some gene values by small fractional amounts. The issue of measuring diversity is discussed in Sections .

A benefit of RKR is that the gene value, as a sloppy indicator of position, actually has physical significance. This contrasts with the encoding used in PMX, where a gene value corresponds to a DPU ID number; the exact value has no physical meaning. We make use of this physical meaning in RKR when modifying the GA to respect tile boundaries, and other heterogeneous features of the target platform.

An implementation drawback of the RKR is that the code base is very different from that for a regular GA implementation. One of the benefits of GAs is that their various parts can be treated like black boxes, in a plug-and-play manner. These parts include selection, crossover, evaluation, statistic monitoring, etc. However, RKR differs in the underlying problem encoding, and it was not found to be cleanly plug-and-play.

Difference Between Random Keys and Continuous Problem

The use of continuous valued problem parameters for a combinatorial problem raises the question of whether we can use search-space methods for continuous search-spaces e.g. gradient based methods. This is not possible because the ordering of the DPUs does not change continuously with a continuous variation of the random keys. Instead, the DPU order changes only at discontinuities in the search space e.g. when one random keys changes enough to exceed the random key of an adjacent DPU, the DPUs change positions. Hence, there is no gradient in the optimization function.

For similar reasons, the random keys formulation is also not a candidate for solution by linear programming [HL90]. Linear programming requires an optimization function that is continuous

and linear in the variables in order to move from corner point to corner point in search of an optimum.

6.1.1.3 Uniform Crossover

Uniform crossover is useful for when building block genes do not occur close together. In uniform crossover, the gene positions at which parents will exchange alleles are randomly determined (Figure 6.6). This avoids the bias in preserving compact building blocks exhibited by 1- and 2-point crossovers.

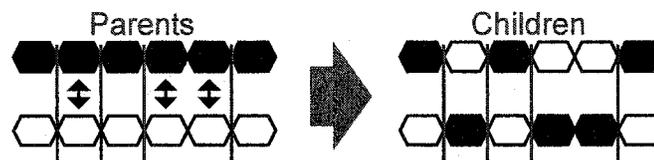


Figure 6.6. Uniform crossover.

The genes to be swapped can be determined in many different ways. For example, if a coin is tossed for each gene position, the number of genes to be swapped will always be close to half of the length of the chromosome. If the number of swapped genes is significantly more or less than half the chromosome length, any building block (distributed or compact) will be more likely to be preserved in the crossover.

One way to induce greater variability in the number of genes swapped is to randomly determine the number of genes to swap before randomly determining their positions in the chromosome. It is straightforward to control the statistics when randomly determining to number of genes to swap.

6.1.2 Infeasibilities

In a regular optimization, a lower cost is preferable, if the solution is *feasible*. Infeasible points in the design space may be specified by *constraints*. In the simplest form, this may specify a range of illegal values for parameter. Sections 6.1.3 and 6.2.2 discuss constraints due to wire length limits and heterogeneous distribution of hardware resources. Some constraints, or combinations thereof, can lead to infeasible regions in the design space that are scattered or oddly shaped e.g. DPUs that write to memory can only occupy odd numbered rows. The simple DPU ordering problem is one of constraint satisfaction i.e. no interconnected DPU may be more than 8 rows apart. In

this thesis, constraints merely refer to the formal recognition and specification of infeasibilities. Infeasibilities may be unspecified, but may also arise in an obvious way from nature of the problem being solved.

If RKR is not used for the ordering problem e.g. in the above encoding for PMX, many combinations of gene values are infeasible simply because they are not permutations of all the DPU indices (this is an implied constraint for ordering problems). For RKR, *all* of the solution space is feasible. Thus, the “ambiguity” in RKR arises from mapping the few feasible points in the space of DPU position numbers to the entire space of random keys.

Infeasibilities can be dealt with in two ways. The first is to assign a penalizing cost to designs that reside in infeasible regions (Figure 6.7(a)), which diminishes their fitnesses. The goal of cost minimization (fitness maximization) then biases the search away from the infeasible regions. The second solution is to redefine the problem parameters so that they cannot represent infeasible solutions. For example, in the scheduling problem of high-level synthesis [Tor99], gene values represent scheduling times for operations relative to their earliest possible times. This precludes scheduling an operation before it can legally occur.

The second solution can be generalized as resolving infeasibilities in the translation between the chromosome and the problem parameters. This can be taken to any level of sophistication. For example, a solution to the simple DPU ordering problem will violate some of the heterogeneous restrictions in the target platform. This infeasibilities can possibly be repaired by trying several heuristic-driven rearrangements of the ordering, which amounts to a deterministic local search in the neighbourhood of the infeasible solution. Thus, Figure 6.7 can also represent the repair procedure, viewed as an additional transformation that maps infeasible points to nearby feasible points. In general, a repair procedure can be regarded as part of the decoding of the chromosome if it doesn't code changes back into the chromosome by changing gene values.

Because of the selective nature of repair (it only changes infeasible genomes), it is sometimes regarded as separate 3rd way to resolve infeasible genomes e.g. in [Par94]. That is, it is not necessarily thought of as a modification in the genotype-phenotype mapping. However, local search can be applied to more than just repair of violations; local *optimization* through local search can be applied to all generated kids, both feasible and infeasible, to find a nearby solution with lower cost. Any nonpopulation-based search space method can be used to accomplish this e.g. simulated annealing.

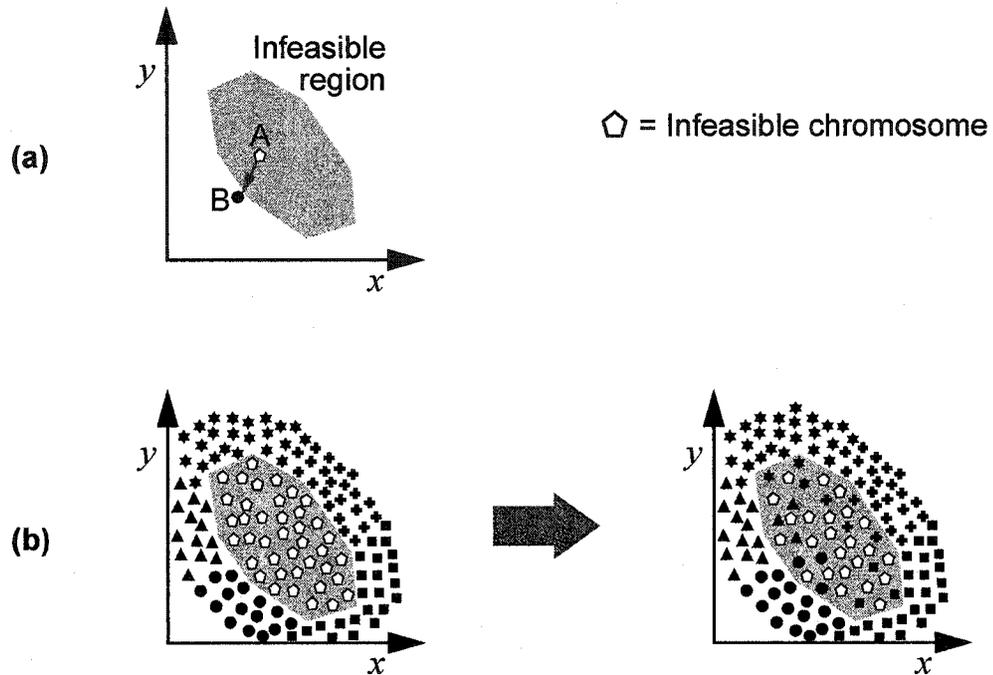


Figure 6.7. Handling infeasibilities.

(a) The cost structure can progressively penalize solutions deeper in the infeasible region more severely. The cost minimization goal then biases the search toward feasible regions. (b) Non-Lamarckian local search changes the genotype space by replacing infeasible points with feasible points, thus making feasible solutions easier to find. All points with the same symbol map to the same (feasible) phenotype.

6.1.2.1 Local Search and Lamarckian Evolution

It is useful to distinguish between the repairs that tamper with the chromosome, such as in PMX repair, and those that don't, such as the local search in the above decoding. Changing genes in the decoding process implies that a temporary copy of a chromosome is modified for the purposes of decoding only; the source chromosome retains its unmodified position within the search space. Lamarckian evolution refers to changing source chromosome based on things learned (such as PMX repair or local search for feasible solutions) rather than through mating or mutation. It has been thoroughly disproved in the natural world, but has been successfully exploited in GAs,

despite controversy over its effects [Whi94]. This local optimization of the chromosome may be applied to existing population members, or kids before possible injection into the population.

There is a case to be made for avoiding Lamarckian back-encoding of the gene values. For example, if the feasible solution B in Figure 6.7(a) was found by local search and coded back to the underlying chromosome, the chromosome A will be “pushed back to shore”. This represents a greedy search, and may prevent the sample points from wandering into infeasible intermediate points that lead to good solutions. Studies have shown that Lamarckian evolution can prevent discovery of a feasible solution that could otherwise be found ([Mic96] p. 321). We believe that it is better to restrict local optimization of the chromosome to a transitory effect in the decoding process e.g. to map an infeasible point to a nearby feasible solution. For points that are deep into an infeasible region, a local search procedure is less likely to map it to a nearby feasible solution; the normal penalization should prevent the GA from wasting too many sampling points there.

For repair procedures embedded into the decoding of the chromosome, the approach taken in this thesis was to perform only small scale changes in the DPU order. The large scale changes were left to mating and mutation of the underlying chromosome. This was to maintain locality of the mapping due to repair. We wanted to avoid the possibility of deceiving the GA search, if the mapped feasible solution is only remotely related to the underlying chromosome.

The natural question arises as to how the evolutionary search can make use of the information learned during local searches if it is not back encoded into the chromosome. The answer is that it does not. The exact role played by the local search can be seen by considering how the infeasible regions are affected by the local search. Figure 6.7(b) shows how previously infeasible points map to feasible phenotype by incorporating local search into the chromosome decoding. Therefore, by changing the genotype-phenotype mapping, the entire genotype space is altered. Infeasible regions become smaller, while feasible solutions have more representation, thus making them easier to find. It is not necessary to modify the position of the chromosome in the genotype space, since the entire genotype space is modified.

6.1.3 Cost Structure

Since the simple DPU ordering problem contains only wirelength constraints, chromosomes were assigned a cost (i.e. penalized) based on their wirelengths. Figure 6.8 shows 3 wire cost schemes that were tested using PMX crossover and generational replacement. In the first scheme, overlength wires were penalized quadratically. Legal wires were lightly priced linearly according

to length so as to encourage clustering of interconnected DPUs. The second scheme was meant to allow more freedom for arranging DPUs within the maximum wirelength by removing the cost for legal wires. All overlength wires were all penalized equally with the rationale that illegal is illegal, regardless of by how much; similarly, legal is legal, and wirelength is regarded as irrelevant. The third scheme attempted to give freedom of arrangement of DPU connected by legal wires, but quadratically penalized overlength wires to push solutions into the feasible region.

The convergence was good for cost schemes in which overlength wires were quadratically penalized, regardless of whether the *legal* wires were penalized to encourage clustering of interconnected DPUs. Convergence was poor if the cost scheme ignored the length of wires in penalizing overlength connections (Figure 6.8(b)). This confirms that infeasibilities have to be quantified and penalized commensurately in order to create a cost gradient that pressures the search into feasible regions.

We elected to use the cost scheme of Figure 6.8(c) instead of Figure 6.8(a), based on findings during the manual placement of the FFT kernel (Section 4.7). As described in Section 5.4, it was found that many connections had to be at their maximum length before a valid placement could be found. Therefore, it didn't make sense to apply pressure to push connected DPUs closer together.

Since connected DPUs had to be spread out, there is a fundamental difference between the building blocks of the TSP and the placement problem. Information about good cities to travel between can be encoded by the adjacency between alleles. However, a natural building block for DPU placement is a cluster of DPUs that share a net. DPUs that are physically adjacent may have nothing to do with each other, while connected DPUs were found to be better scattered apart. This irrelevance of adjacency can also be said of RKR. Therefore, it is worthwhile to try crossovers such as uniform crossover, which are not biased in preserving compact building blocks.

In the adopted cost scheme, cost arises solely from penalizing infeasibilities, as is expected for a problem driven solely by constraint satisfaction. Unlike regular optimization problems, no solution is feasible unless the cost is zero. Conversely, all solutions that do not violate the constraints are equally good.

6.1.4 Fitness Scaling

Since selective pressure governs the convergence of the GA, the relationship between cost and fitness is very important. The population may consist of diverse individuals, but the degree to which good chromosomes are favoured will determine how much of the diversity is used. It is not

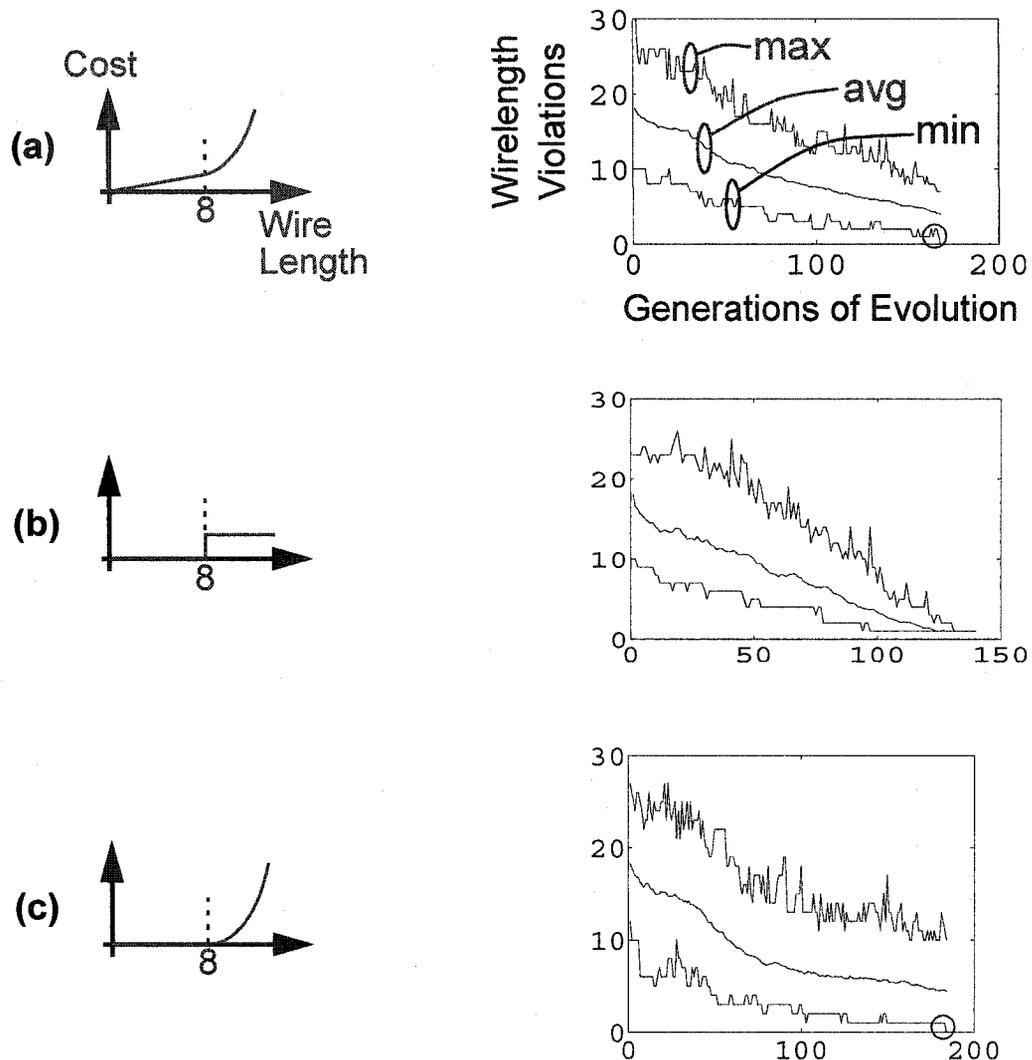


Figure 6.8. Typical convergence for different interconnect cost schemes. Wires were for each source-destination pair. Maximum legal length of wires is 8 rows. **(a)** Linear ramped for legal wires, quadratic penalty for overlength wires. **(b)** No cost for legal wires, unit cost for overlength wires. **(c)** No cost for legal wires, quadratic cost for overlength wires. In **(b)** and **(c)**, cost is due solely to penalization of infeasibilities.

acceptable to simply take the reciprocal of cost to obtain a fitness figure. The asymptotic behaviour of $1/\text{cost}$ ensures that good fitnesses explode toward infinity as the cost approaches zero. This

leads to premature convergence, since the leading favourable gene will hog all the mating opportunities and proliferate its genes throughout the population. In contrast to a more general optimization problem, this is especially harmful in a problem driven only by constraint satisfaction because the cost *must* be zero before a feasible solution can be obtained. A good method of translating cost to fitness cannot have the severe nonlinearity of $1/\text{cost}$.

Three methods were examined for fitness scaling: simple linear scaling, sigma truncation, and rank based fitness.

6.1.4.1 Simple Linear Fitness Scaling

As its name implies, linear scaling relates the cost and fitness by a straight line (Figure 6.9). The slope of the line doesn't matter, since any chromosome's fitness is normalized by the total fitness of the population in determining its chances for mating. Any gain induced by linear fitness scaling will be cancelled in the normalization.

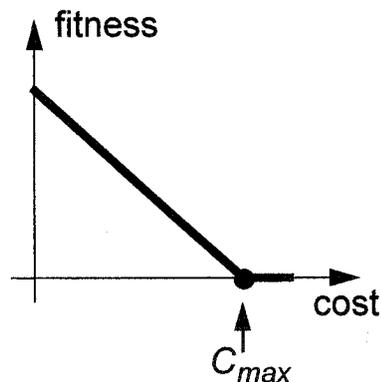


Figure 6.9. Linear fitness scaling. The cost in a general problem can be arbitrarily large. Therefore, a maximum cost must be specified, above which a solution is regarded as insane and given zero fitness.

The first implementation of fitness scaling was rather naive. It was decided that the need to specify C_{max} was bad because it excluded some population members from mating, effectively shrinking the population. This was addressed by dictating that C_{max} will take on the highest cost in the population for the current generation. Thus, zero fitness point changes from generation to generation, and only the worst chromosome is excluded from mating.

This dynamic dependence of C_{max} on the worse chromosome was found to cause problems. For large populations, the larger outliers yield a large C_{max} , even though the cost distribution has the same mean and variance (Figure 6.10). For example, suppose that the cost distribution is such that

there is 90% chance that any one chromosome's cost is below some value C . For a population of N , the chances of having a cost greater than C is $(1-0.9^N)$, which increases with N . Therefore, the highest cost is typically higher for a large population. Defining C_{max} as the largest cost in each generation results in a *mean fitness* which is larger for a large population, even though the variance is the same as for a small population. In other words, a single high-cost outlier will compress the *relative* spread of the fitness for the bulk of the large population, resulting in less selective pressure. This can be seen in the drawn-out convergence of Figure 6.11.

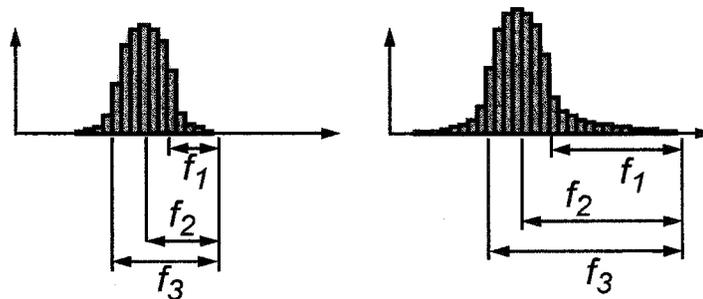


Figure 6.10. Cost distributions for small and large populations. Large populations have greater chance of greater outliers with larger costs.

6.1.4.2 Sigma Truncation

Since the simple linear fitness scaling modulates selective pressure undesirably, scaling with *sigma truncation* was seen as a potentially better alternative. Sigma truncation fixes C_{max} at some number of standard deviations away from the mean cost (Figure 6.12). There are several reasons why this scheme was passed over in favour in rank based fitness. The first was that a plot of cost distribution often showed extremely nongaussian and/or asymmetric distributions. The second reason was that we encounter the problem which motivated simple linear scaling; that is, chromosomes with greater cost than C_{max} are excluded from mating, thus effectively reducing the size of the population. Finally, the mean and standard deviation must be calculated every generation.

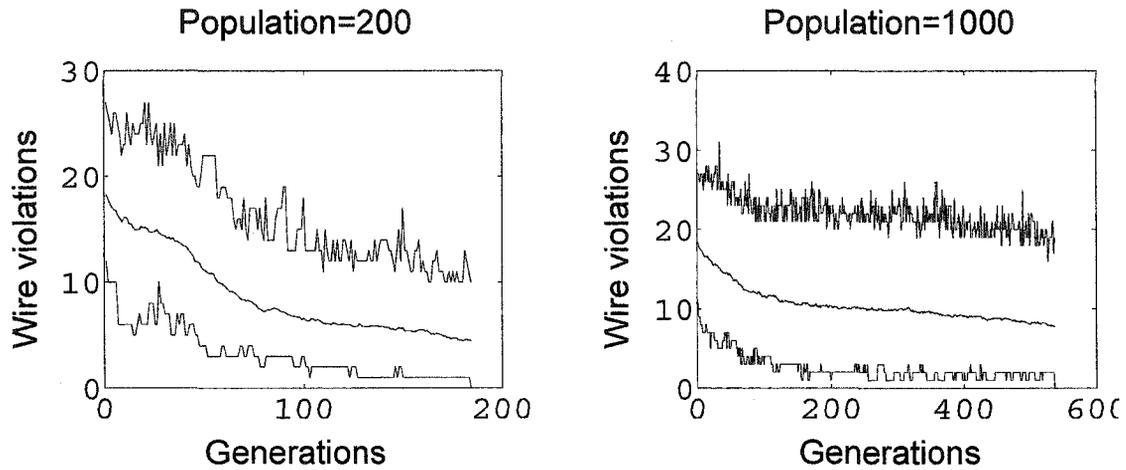


Figure 6.11. Linear fitness scaling with C_{\max} dynamically taking the value of the greatest cost in each generation.

This reduces the selective pressure for large populations and delays convergence. The population of 200 found a solution (zero violations) in approximately 36,000 matings, while the population of 1000 found a solution in approximately 570,000 matings.

6.1.4.3 Rank Based Fitness

In rank based fitness, the chromosomes are ranked in order of descending cost. Their ranks are then linearly mapped to fitness. In the literature, the usual implication is that fitness (and thus probability of selection) is proportional to rank (Figure 6.13a). In principle, though, the ranking can be mapped in an arbitrary manner to fitness e.g. to control the greediness of the search. One such modification that was made in this work was to boost the fitness ratio of worst to best chromosome by adding a constant fitness offset f_0 to all the ranks to come up with the fitnesses (Figure 6.13b).

As an example of typical values for f_0 , consider a population of $N=200$. From Figure 6.13a, the unadjusted probability of selecting the worst chromosome is $1/200$ of that for selecting the best chromosome i.e. very small. Many of the “bottom feeders” may be relatively new population members, since evolution makes the population very fit, which makes it harder for kids to come in with a high rank. These bottom feeders are important for contributing new genetic material to the

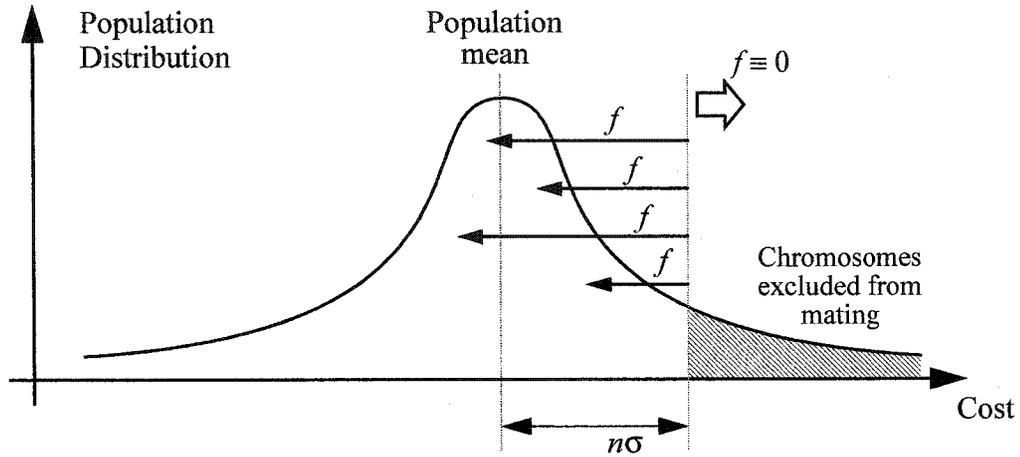


Figure 6.12. Scaling by sigma truncation.

Fitness is defined as zero at a cost that is some number of standard deviations above the mean. The population size is effectively reduced because chromosomes with higher costs have no chance of mating.

search and avoiding the attraction of local minima. Assume that we would like to boost the probability of selecting the worst chromosome to 1/10 of selecting the best chromosome. Thus,

$$\frac{f_0 + N}{f_0 + 1} = P_{b2w}$$

where $P_{b2w}=10$ is the ratio of probabilities of selecting the *best* and *worst* chromosomes. This yields the offset

$$f_0 = \frac{N - P_{b2w}}{P_{b2w} - 1} \cong \frac{N}{P_{b2w}}$$

For our example, $f_0 = 190/9 \cong 21$.

There are many other advantages to making fitness proportional to rank. These benefits are related to artificially redistributing the cost differences between adjacently ranked chromosomes so that they are uniform. So long as uniformity in adjacent difference in fitness is maintained, the advantages still apply even in the case of modifications such as the above addition of a constant to the rank.

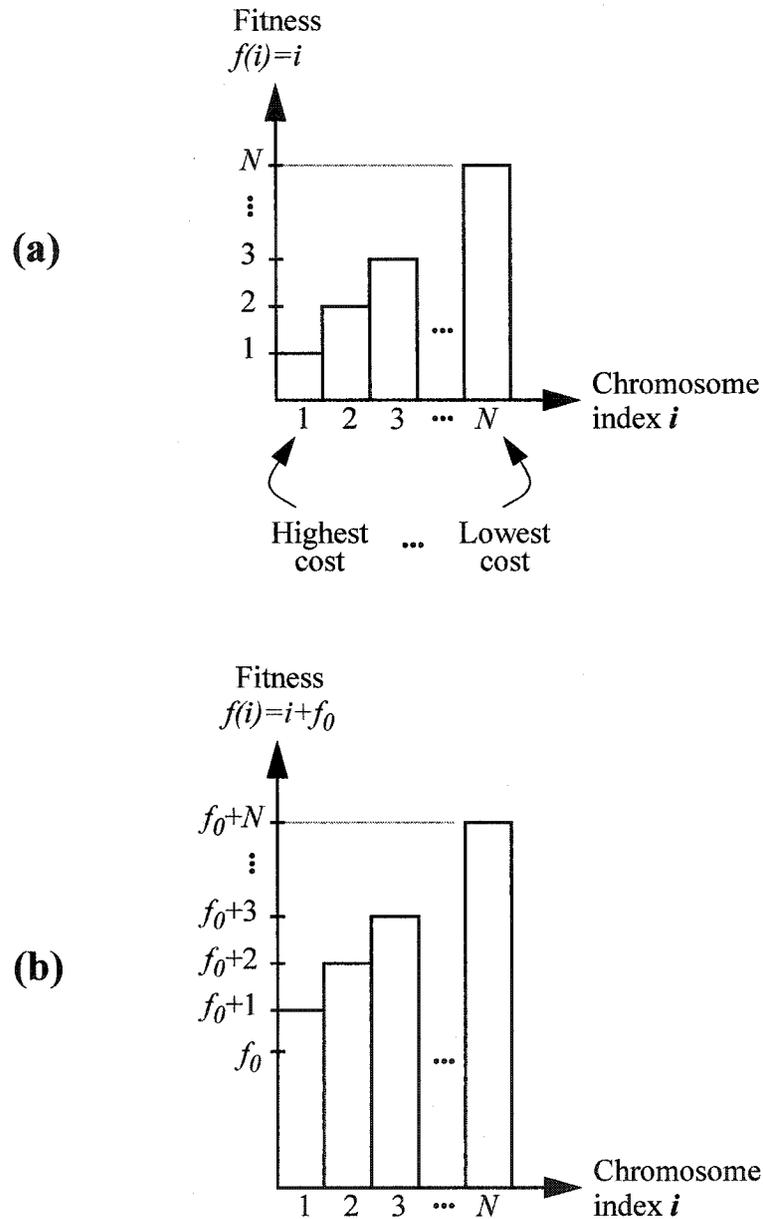


Figure 6.13. Ranked based fitness. Chromosome index assumes that chromosomes are sorted according to descending cost.
 (a) Default mapping of chromosome index to fitness. (b) Adding an offset to all fitnesses causes the relative fitnesses to compress in a controlled manner, thus decreasing selective pressure evenly throughout entire population, and increasing diversity.

The first benefit is that no outliers are excluded from selection; the entire population available for mating. Furthermore, it doesn't matter how bad the worst chromosomes are; they will always be given the same *reasonably* small and *controllable* opportunity to mate, thus ensuring a certain level of diversity in the exploration. This avoids the problem where a bad chromosome is technically allowed to mate, but its fitness is near zero, thus rendering it effectively nonexistent. Remember that the whole purpose of having a population of a certain size is diversity.

Secondly, if there is an outlier with exceptionally low cost, that chromosome cannot monopolize the mating opportunities and cause premature convergence. This is because the next best chromosome has a rank that is almost as good. In fact, the process of ranking behaves like a monotonic nonlinearity that provides locally adapting gain. It blunts the disparity in the cost of an outlier, yet provides discrimination between chromosomes that are bunched together in cost.

By the same token, a single bad outlier cannot compress the relative fitnesses of the bulk of the population. Such degraded differentiation delays convergence in an uncontrolled and possibly unbeneficial way, as in the case of large populations using the simple linear scaling.

Another benefit of the locally adapting gain is if there are several exceptional outliers, inbreeding is prevented by the moderation in cost differences between the elite and mediocre chromosomes.

The implementation of roulette wheel was found to be greatly simplified for ranked fitness. As in the general roulette wheel, selection for the usual ranked fitness (Figure 6.13a) is performed by unrolling the roulette wheel onto the real number line (Figure 6.14), then generating a random number along the real number line. In regular implementation of the roulette wheel, each randomly generated number must be converted to a chromosome index via a lookup table. This involves either a binary search for a 1-arm roulette wheel, or a scan of the table for a multi-arm roulette wheel.

In contrast, the lookup procedure is extremely simple for ranked fitness. We know that the chromosome fitnesses are merely the chromosome indices $\{1, 2, \dots, N\}$. To select parents by generating a random number, the points on the real number line must be segmented into lengths that are proportional to these fitnesses. Since these segments are abutting, the delineation points are just the cumulative sum of the linearly ramped fitnesses i.e. the sum of an arithmetic series. Thus, there is a simple algebraic mapping from chromosome index to the delineation points on the real number line, as well as in reverse. Conversely, a random number can be translated to a chromo-

some index using simple math. This is true even with the above modification of adding a constant to all ranks to obtain fitness; only the arithmetic series changes.

For the usual roulette wheel of Figure 6.13a, the cumulative fitness on the real number line is $F(i)=(i+1)/2$ for the chromosome ranked with index i (Figure 6.14). These real numbers delineate the chromosomes to be selected, and can be mapped back to i with the reverse mapping

$$i = \frac{\sqrt{8F(i) + 1} - 1}{2}$$

As in general roulette wheel selection, any random number x landing within line segment $f(i)$ should map back to chromosome i . Therefore, x should map upward to $F(i)$ before mapping back to i (Figure 6.15). Equivalently, it is simpler to directly apply the mapping $F(i) \rightarrow i$ to x and round the result upward:

$$i = \left\lceil \frac{\sqrt{8x + 1} - 1}{2} \right\rceil$$

For the diversity-enhanced fitness of Figure 6.13b, the mapping becomes:

$$i = \left\lceil \frac{\sqrt{8x + (2f_0 + 1)^2} - 1 - 2f_0}{2} \right\rceil$$

Ranked Selection and Tournament Selection

Because an $O(N \log N)$ sort is required in order to rank the population of genomes, a linear-complexity alternative has become well known as having some of the same advantages. *Tournament selection* has similar first order statistics as a ranked fitness scheme that uses a *linear* mapping between rank and fitness. However, tournament selection is “noisier” [Whi94,GD91,Bli00]; in other words, the expected change in fitness distribution between a population and the set of parents selected from it is the same as for ranked fitness, but the variance in actual change may be greater than in ranked fitness. This can be addressed by the use of a multi-arm roulette wheel [Bli00]. Furthermore, tournament selection is amenable to parallel processing of the GA.

In tournament selection, each parent is selected through a tournament among n genomes, which in turn are selected from the population with uniform probability. n is often just two. The winner of a tournament can be either the fittest of the n competitors, or probabilistically selected based on their relative fitnesses.

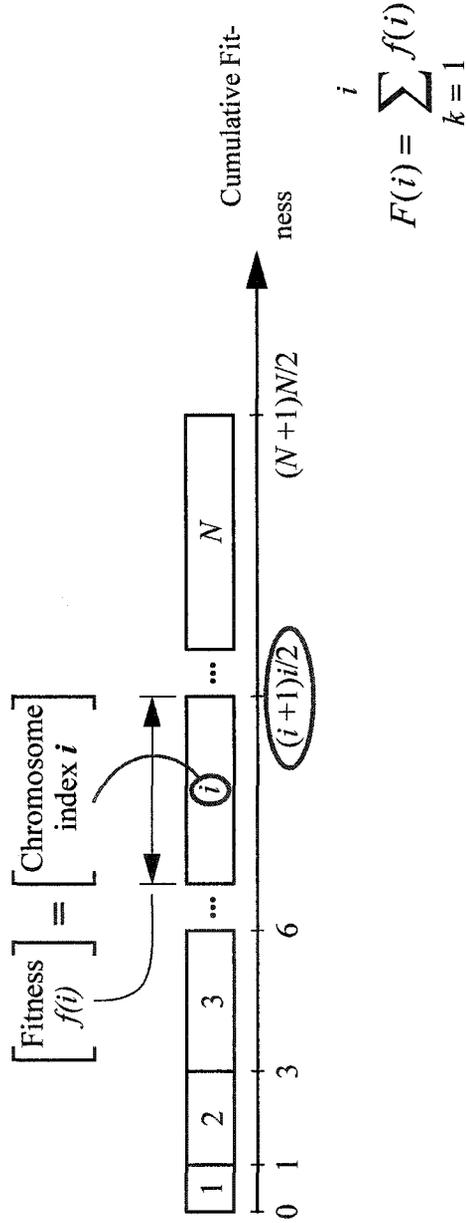


Figure 6.14. Selection of parents by unrolling the roulette wheel of ranked based fitnesses.

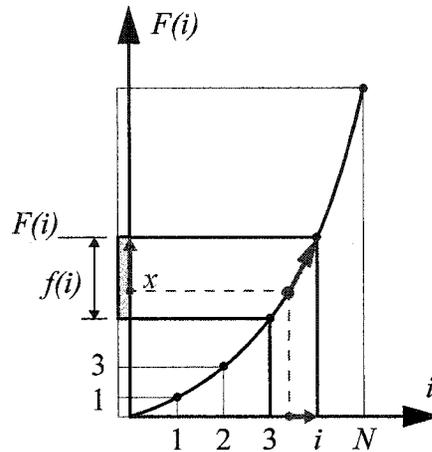


Figure 6.15. Spinning the roulette wheel arm.

For probability of selecting chromosome i to be proportional to fitness $f(i)$, a random number x falling within the fitness segment $f(i)$ must be mapped up to the $F(i)$ before mapping back to the chromosome index. More simply, the $F(i) \rightarrow i$ may be applied directly to x , and the result rounded up.

Use of tournament selection was considered and relegated as a possible future exploration. Aside from time constraints, one of the major reasons for this is because the genomes have to be sorted anyway to root out clones and twins. Note also that the population is already sorted, so only the kids have to be sorted in each generation before merging with the population in linear time. This does not change the order of computational complexity, but it mitigates the *worst-case* super-linear complexity of the sort.

6.1.5 Measuring Diversity

Section 2.5.2 described the importance of balancing diversity with selective pressure. In this section, we develop an intuitively-based method of measuring diversity. It was motivated by the need to understand some cases of slow convergence (degenerate genetic material was suspected). This approach is tested and fine tuned in Section 7.2. For the purpose of feedback to modify diver-

sification mechanisms, the incorporation of this metric into a GA with time varying parameters is future work, (Section 8.4.1).

6.1.5.1 Challenges to Quantifying Diversity

For now, let us ignore the fact that Euclidean distance along any axis (or gene value) may not even be meaningful; the intuitive basis for a diversity metric is still elusive. Let us first consider the reduced problem of finding the diversity in the values for a single gene. Consider the first gene in all solution points. If the valid range was $[0.0, M]$ and gene values were equally distributed among the two values $\{0.0, M\}$, the variance would be very large even though there is hardly any diversity. This is identical to the case of a purely binary signal, which can have significant AC power (which corresponds to variance) even though it only takes on two possible voltage levels. Therefore, there are serious caveats with using variance to characterizing diversity, even though it was the metric that was initially used in this study.

To evaluate the scatter of entire chromosomes within the design space, the literature contains suggestions of using pairwise Euclidean distance between all possible chromosome pairs (Figure 6.16(a)). For problems in which Euclidean distance has no meaning, one theoretical alternative might be to find the pairwise Hamming distance between all possible chromosome pairs. Until recently, this was very expensive to calculate, since there are in the order of N^2 pairs that could be formed out of a population of N . Furthermore, we encounter the same problem as with variance. For example, if half of the chromosomes were around one corner of the solution space and the other half were around the diametrically opposite corner, the average pairwise distance could be large, even though they effectively only sample two small regions in the solution space.

The all-pairs Euclidean distances can be combined in different ways to arrive at a diversity metric e.g. linear sum, L_2 -norm, harmonic average, etc. It was recently shown that the L_2 -norm of Euclidean distances between all possible genomes is simply the L_2 -norm of the standard deviations of genes within each locus [WO03], which can be calculated in linear time. Because of their relationship, it is not surprising that variance and all-pairs Euclidean distances share some caveats. However, the equivalence circumvents the computational barrier to one form of the all-pairs Euclidean distance metric. In the case where the gene values are discrete and not representative of a physical quantity, [WO03] also provides a linear time method to linearly sum all-pairs Hamming distances.

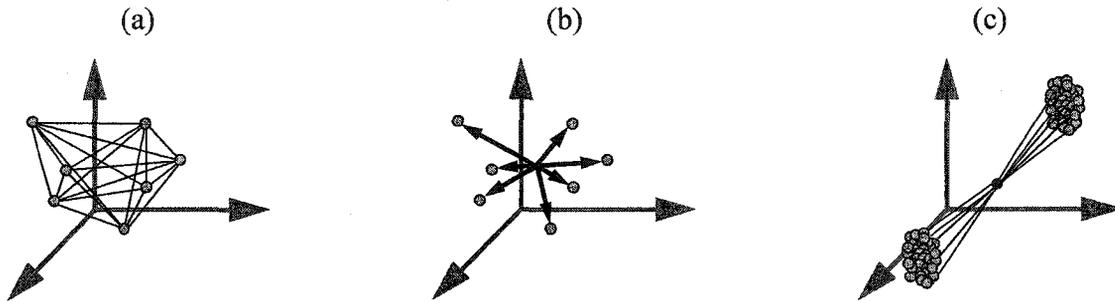


Figure 6.16. Some methods of measuring diversity of solutions in a search space. (a) Based on pairwise distances. (b) Based on distances from a reference point e.g. the centroid. (c) Poor diversity can be hidden from a metric based on (b).

Examples from the Literature

The following are recent works in which diversity measurement plays a prominent role.

- ◆ Reference [OW99] sums the Hamming distances between all possible pairs of genomes.
- ◆ Reference [BM00] sums pairwise distances. Euclidean and Hamming distances are used.
- ◆ In [Urs02], the population's center of mass in the search space is determined by averaging each gene across the population. The distances of all solution points from the center of mass are then averaged over the population (Figure 6.16(b)). To avoid scaling issues (possibly due to the dimensionality of the search space), the average distance is normalized by the diagonal distance from one corner of the search space to the opposite corner. To avoid scaling issues *between* genes, the gene values are scaled to have identical value ranges. Note that this method contains the same problem as with variance in a single gene, where reasonable values from metrics based on distances don't always correspond to good diversity (Figure 6.16(c)).
- ◆ Reference [TVFG03] quantifies the variability in some scalar metric across the population. The measure of variability used here is entropy. In the domain of the problem parameters, the entropy in *fitness* is evaluated over the population. Note that this measures variability in fitness only, not variability in genetic material. In the domain of the search space, [TVFG03] takes the entropy in *distance to a reference point*; as with fitnesses, similar distances can be obtained from widely separate points in space. Therefore, this method can underestimate diversity, in contrast to the overestimation in the variance and pairwise distance measures.

- ◆ Reference [EN02] uses the pairwise distance between chromosomes. This work deals with genetic programming, so the distance is determined by an *edit distance*; this is the smallest number of operations needed to convert one tree-based chromosome to the other.

6.1.5.2 New Intuitive Basis for Quantifying Scatter

The decision was made to quantify diversity across the population for a single gene. This measure may then be scaled to give comparable value ranges across all genes in the chromosome, then averaged across the chromosome.

By looking at scatter within individual genes, it appears that any metric that may be created doesn't capture true Euclidean-like scatter in N -space. As mentioned above, however, the all-pairs Euclidean can also deceptively indicate reasonable diversity. In contrast, scatter within a single gene is easily recognizable intuitively, regardless of whether the genes are merely indices or correspond to physical quantities. We develop a method to quantify this scalar scatter.

Consider how the scatter over $[0.0, 1.0]$ might be measured without regard to computational complexity at first. We can then try to find a quick approximation. We initially assume that the gene value has physical meaning, rather than just being an identifying index. To simplify the discussion, we assume a continuous value, though the development is applicable for discrete valued genes too.

Gene Value Distribution and Histogram "Signal" Energy

One way to assess the scatter in a gene's values is to histogram the data for the population (Figure 6.17). In order for a histogram to be useful, there should be significantly more data points than histogram bins; conversely, the bin width should be significantly wider than the average spacing between data points. Under this circumstance, a bin's height approximates the density of data points in the vicinity of the bin. Poorly scattered data will show up in the distribution as a few peaky bars. For uniformly distributed or well scattered data, all the bars would be of roughly equal height (Figure 6.17(a)).

One indication of nonuniform density of gene values is variability in histogram bar heights. Now think of the histogram as a signal in time (rather than gene value). Since the number of data points being histogrammed is fixed, any variability in the "signal" will not change the mean bar height (the DC component), but does add an AC component to the signal power. The AC signal power is merely the variance of the bar heights. This serves as a metric for the nonuniformity in

the gene values. In the following discussion, it is helpful to remember that for a sequence of N data points, signal energy and signal power differ merely by a normalization factor N^2 .

It is computationally simpler to simply sum the square of the bar heights, which equates to finding the total energy of the histogram signal. Since the DC portion is constant and independent of the variability in the bar heights, this merely adds an offset to the AC power. This is merely the stochastic signal view of variance:

$$\begin{array}{c} \textit{Total} \\ \textit{signal} \\ \textit{energy} \end{array} = E(X^2) = \underbrace{\textit{Variance}}_{\substack{\textit{AC} \\ \textit{power}}} + \underbrace{[E(X)]^2}_{\substack{\textit{DC} \\ \textit{power}}}$$

While it is conceptually elegant to find the signal energy of the histogram, a more direct and less costly method is needed for implementation. Furthermore, the method should not be sensitive to the number of bins, or their widths. As well, it should address the fact that poor scatter is more than just randomness of bar heights; it is *expanses of low or zero bar heights, punctuated by few much higher bar heights*.

That variability in the histogram is not the same as poor scatter can be seen in parallels between the aforementioned histogram and the energy spectrum of a (finite) random data sequence with uniform PDF. In theory, the energy spectrum should uniform, but a simple FFT would show a highly varying spectrum. The spectral uniformity only shows if the frequency bin widths are made wider than the frequency sampling i.e. if the local average spectral density is considered, or if we average many such FFT's from many such random signals. Obviously, the problem is that any specific random sequence is only one signal from an infinite ensemble of signals for a random process, and the flat spectrum only applies to the random process (hence the need to apply the FFT to the autocorrelation function rather than the signal itself). However, this does show that unless proper consideration is given to the bin widths, it is possible to see great dips and peak at specific points in the histogram, even if data points are well scattered at a macro level. The only time this risk is minimized for perfectly equidistant values, which is not very random at all.

For the following development, we again assume a set of N data points distributed over $[0.0,1.0]$.

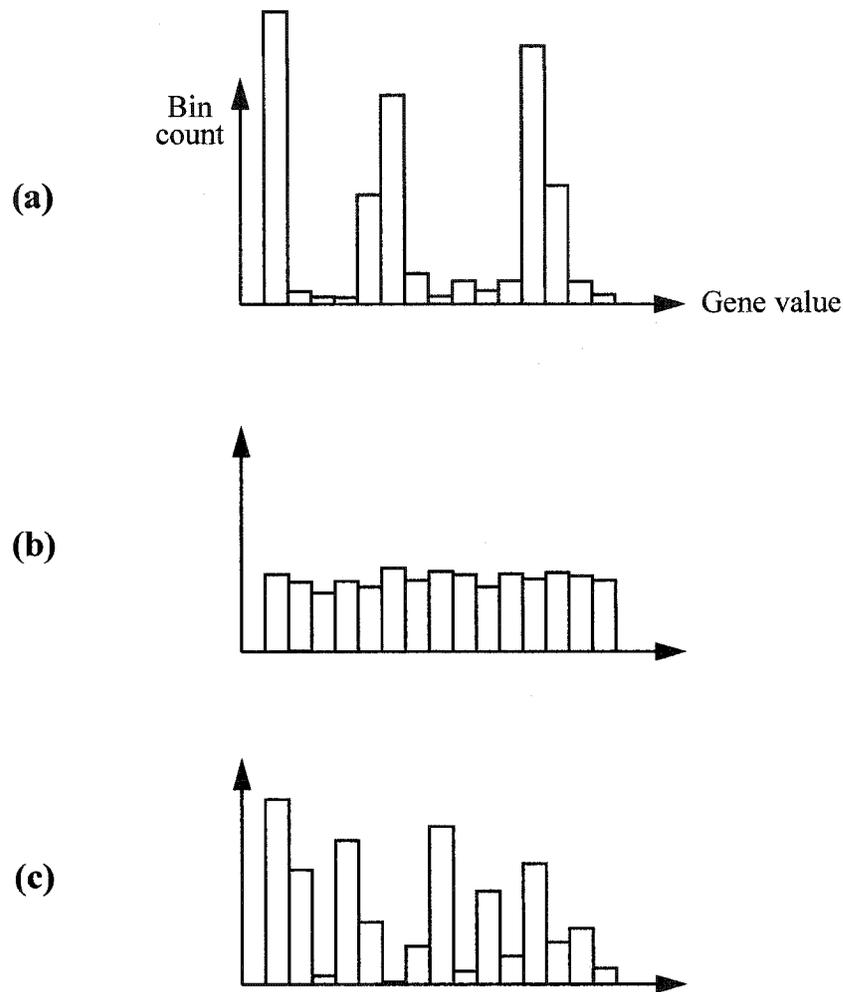


Figure 6.17. Intuitive recognition of scatter in a gene by histogramming the gene's values for the entire population.

(a) When the gene's values collect in a few regions, there are expanses of near-zero-height bars, punctuated by few tall bars.

(b) Even histogram bars result when the gene values are uniformly distributed or randomly scattered values.

(c) Variability of histogram bars indicates nonuniform distribution of gene values, but is an imperfect indicator of a lack of diversity.

Summed Energy of Adjacent Differences in Gene Value

Consider a set of perfectly scattered data points that are equally spaced by $1/N$ (Figure 6.18(a)). If some datapoints are pushed together, some of the *neighbouring* distances will become very small, but a few distances will become large. Because of the conservation of distance, the above properties for histogram signals apply here as well. That is, the mean *difference between adjacent gene values* is not changed by the variability of adjacent differences. As in the histogram signal, the disparity in adjacent differences shows up in either the total energy, or the variance, of the difference signal; they differ only by a constant offset.

This metric can be easily modified for gene values which have no physical meaning e.g. DPU indices, but normalized such that the gene value resides in $[0.0, 1.0]$. However, we will see that the modification also improves the metric even for cases where distance between gene values has meaning. We assume that the diversity can still be intuitively assessed by looking for random scatter in the data points. Thus, we are still interested in variability in the spacing between data points.

Position Independence of Scattered Gene Values

Consider the case where all the data points are concentrated around one end of $[0.0, 1.0]$ e.g. 0.1 (Figure 6.18(c)). The adjacent differences between data points will be small; this *deceptively* indicates good diversity. The problem is that there is a wide expanse of the value range which contains no data points; the wide expanses at the extremities of the value range are not accounted for in the difference energies because the last (or first) data point is missing a neighbour with which to difference.

The wide expanses at the ends can be accounted for by curling the real number line segment $[0.0, 1.0]$ into a circle (Figure 6.18(d)). All points have neighbours in a circle, even though the neighbour may be reachable by the long way around the circle. Taking the *circular* adjacent difference makes intuitive sense because it doesn't matter where the data points concentrate around; the metric gives the same result because the adjacent differences are position independent.

Taking the circular difference was motivated by the fact that the gene value has no physical meaning, so the 0.0 can be justifiably synonymous with 1.0. However, it should be obvious that summing the circular difference energies is a good way to assess scatter over $[0.0, 1.0]$ in general; it is irrelevant whether the numbers have physical meaning or not.

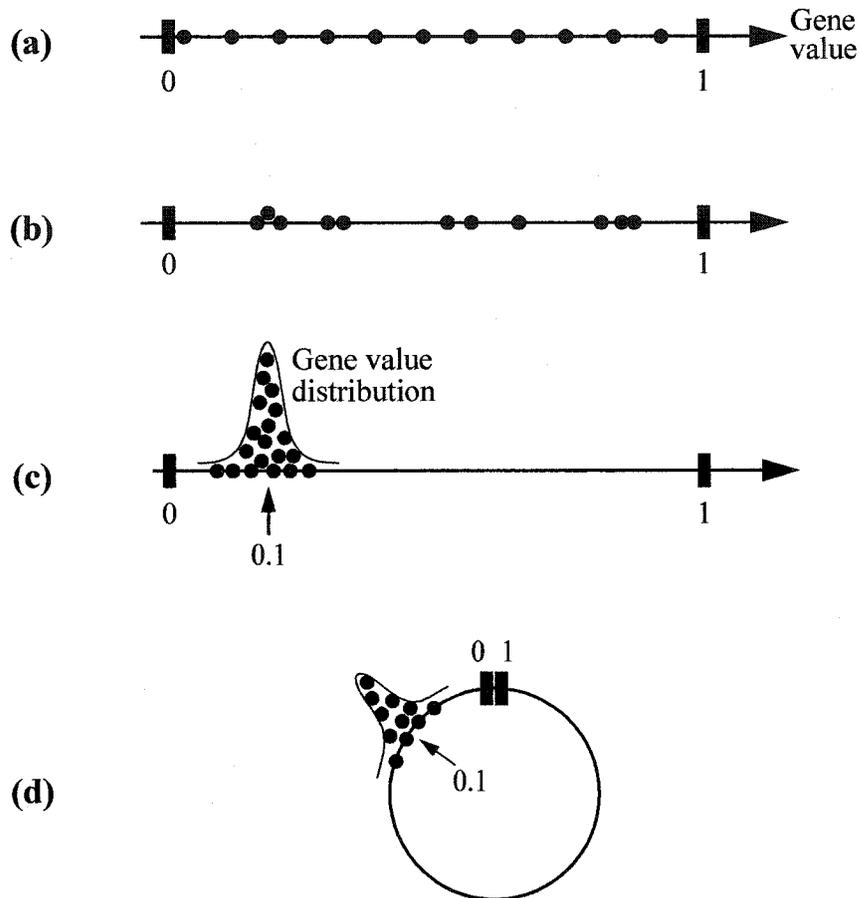


Figure 6.18. Distribution of values for a single gene over the population. The gene's value range is $[0.0,1.0]$.

(a) Uniformly distributed values. (b) Randomly scattered values. (c) Values bunched up around a single value. The adjacent differences are small because the highest and lowest values have no neighbours, thus excluding the wide expanses of the $[0.0,1.0]$ range from the differencing. (d) Circular differencing includes *all* regions of the $[0.0,1.0]$ range, and makes the differencing insensitive to absolute position.

6.1.5.3 Intuitive Normalization for Metric

Finally, we look for a good normalization figure for this metric (energy for the circular difference signal). So far, we have used the case of equidistant data points as the most uniform distribution of gene values. It gives the lowest possible total difference energy. However, truly random data points will have a variance in their differences, thus giving rise to a higher total difference energy. For randomly scattered data, the variance in differences can be quantified by considering the occurrences of data points along the circular $[0.0, 1.0]$ line as a Poisson process with occurrence rate of $1/N$. The variance of inter-event spacing is $1/N^2$. An initially random distribution of gene values yields a metric of approximately 1 when this normalization is used. The metric increases as the GA search proceeds and diversity decreases.

We present a sanity checking demonstration of the metric with a simple use of this normalization, then use the $1/N^2$ variance to formally develop normalizations to improve the diversity metric.

6.1.5.4 Generating Trial Data

In order to demonstrate the metric on various degrees of diversity, we need to randomly generate gene values that are then subject to varying degrees of coalescence. Thus, random data generated with a uniform *PDF* undergoes an $R \rightarrow R$ mapping so that some portions of the range $[0.0, 1.0]$ compressed, thus increasing the *PDF*, while other portions are stretched, thus decreasing the *PDF*. Therefore, the exact mapping is determined by the *PDF* being sought. Intuitively, the mapping must be monotonic.

The mapping can be determined as follows. Let:

x = Domain of the random gene values with uniform $PDF_x(x) = 1$

y = Domain of the mapped gene values with the desired $PDF_y(y)$

$y = G_{yx}(x)$ is the mapping from x to y being sought

Let x_0 be any specific value for x , and $y_0 = G_{yx}(x_0)$. Then, from the conservation of probability,

$$\int_{x=0}^{x_0} PDF_x(x) dx = \int_{y=0}^{y|_{x=x_0}} PDF_y(y) dy = \int_{y=0}^{y_0} PDF_y(y) dy$$

$$CDF_x(x_0) = CDF_y(y_0)$$

This makes intuitive sense, due to the monotonicity of the mapping; all sample values $x \leq x_0$ map to values $y \leq y_0$.

Since we started with the simple uniform PDF in the source domain x , $CDF_x(x_0) = x_0$:

$$x_0 = CDF_y(y_0)$$

This gives the desired mapping G_{yx} :

$$y_0 = CDF_y^{-1}(x_0)$$

Since the inverse is taken of $CDF_y(y)$, it must be a “one-to-one” function (or “bijective”). Thus, for this morphing of PDFs to work, any perfectly flat or vertical portions of $CDF_y(y)$ must be of infinitesimal length i.e. the desired $PDF_y(y)$ cannot be zero over any finite interval within $[0.0, 1.0]$. If it does have stretches of zero value, these regions must be artificially boosted up to a negligible amount (e.g. 10^{-3}), and the entire PDF rescaled to have unit area.

$CDF_y(y)$ is obtained from $PDF_y(y)$ by numerical integration. Since $CDF_y(y)$ is surjective, the inverse $CDF_y^{-1}(x)$ can be realized by interpolating between lookup table entries for $CDF_y(y)$. An example of this PDF shaping is shown in Figure 6.19. Starting from a desired PDF in plot (a), we generate sample random gene values to empirically test our diversity measure.

Figure 6.20 shows sample values of the diversity metric for different distributions of gene values. As expected, the metric takes on the value of unity for uniformly scattered data points, and increases with nonuniformity of the data point distribution.

6.1.5.5 Reference Points for Normalization

Consider a population of N genomes, and let us focus on a single locus k in the genotype. We seek first seek a locus-specific diversity D_k evaluated over the population, then average this across

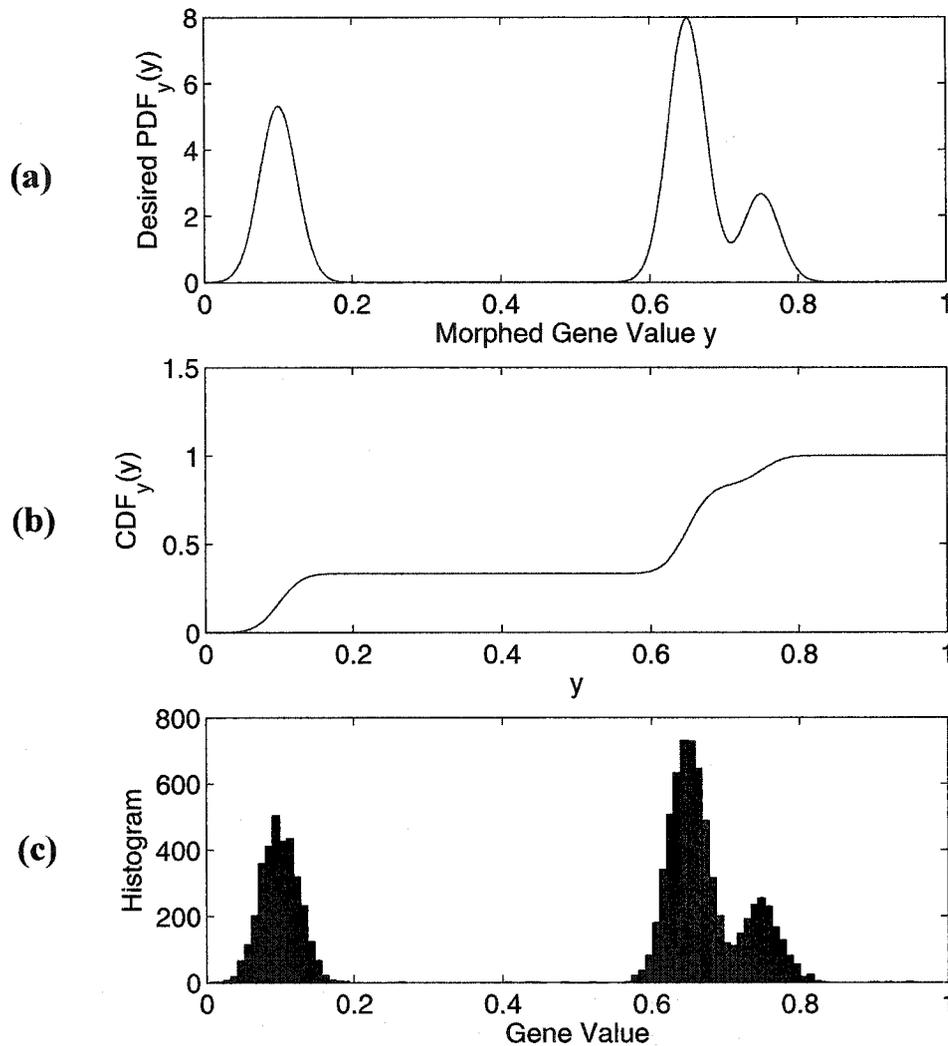


Figure 6.19. Shaping the PDF of randomly generated gene values. (a) Desired distribution. (b) The inverse of the mapping to be applied to gene values generated with a uniform PDF. (c) The distribution of the morphed gene values from which our diversity metric will be calculated.

the loci in the genotype. The N alleles in locus k have values in the range $[0.0, 1.0]$. In the discussion below, let d_i be the circular differences between adjacent values i.e. $\sum_{i=1}^N d_i = 1$. In addition

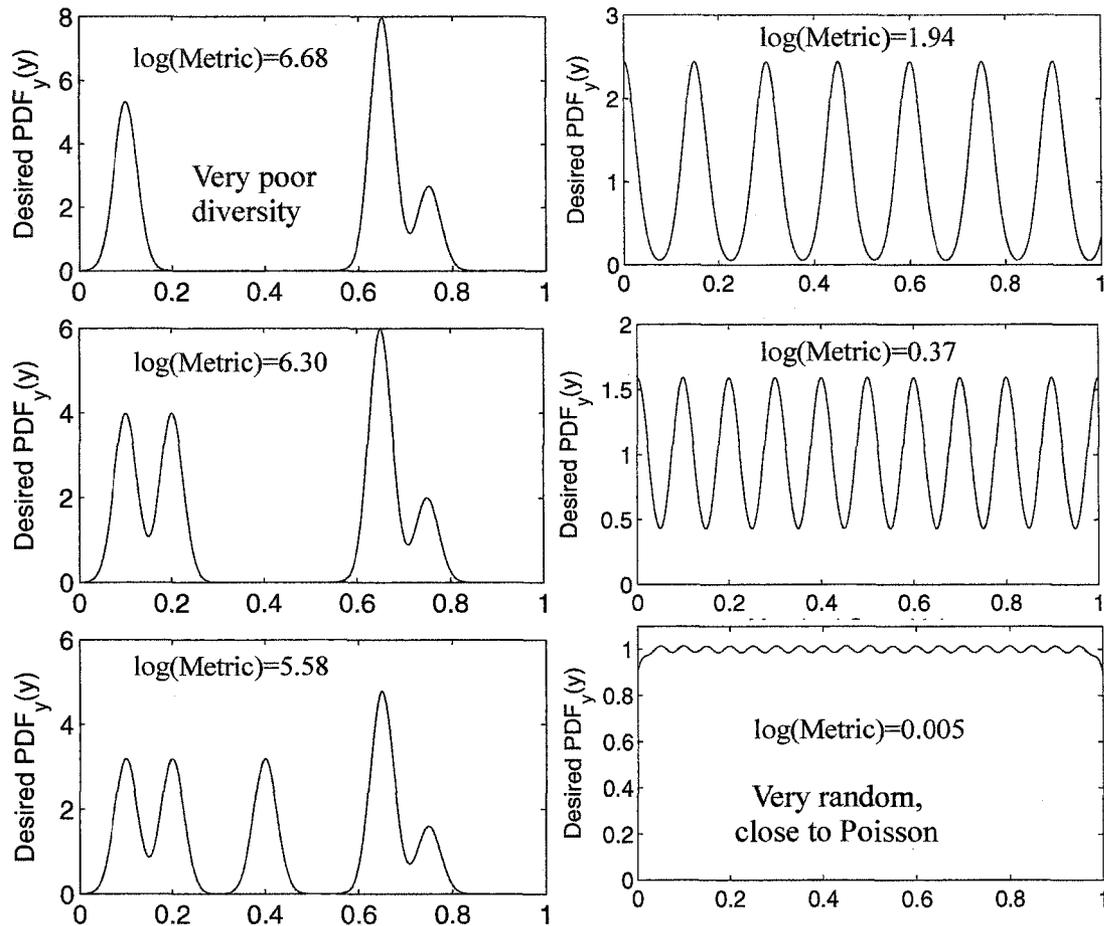


Figure 6.20. The diversity metric for gene value distributions that have coalesced into varying degrees of nonuniformity. Each trial consisted of 10,000 data points. Each metric is the average of 10 trials.

to monotonicity, any sensible scaling scheme should be sane at the following three diversity scenarios at the very least, which we use as reference points.

- ◆ Pathologically *high diversity* [designated “(++)”]: $d_i=1/N$. This corresponds to equidistant allele values.
- ◆ *No diversity* [designated “(0)”]: $d_i=\delta_{ij}$ for a single integer value j . δ_{ij} is unity when $i=j$, and zero otherwise. Since the gene value range has been curled into a circle, it doesn’t matter what j is, but there is only one such value.

- ◆ Completely *random scattering* [designated “(+)”]. This results from N samples of a random variable with a uniform PDF. d_i has the statistics of the event separations of a Poisson process. This is scenario at the start of a GA simulation. Even so, it is considered “intermediate” diversity, since it lies between the two conditions above.

We seek a D_k that behaves sanely in the regime bounded by the cases (+) and (0). Since (++) is a case that is not reasonable to expect in practical situations, we do not care how D_k scales between (+) and (++) so long as it is in the right direction. For the sake of intuition, we will use the following relatively standard notions in handling the variance of the circular difference signal $\{d_i\}$.

- ◆ $\mu=1/N$ is the mean of $\{d_i\}$, or it’s “DC component”
- ◆ $P_{DC}=\mu^2=1/N^2$ is the DC power of $\{d_i\}$ and depends only on N .
- ◆ $P_T = \frac{1}{N} \sum_{i=1}^N d_i^2$ is the total signal power.
- ◆ $P_{AC} = \frac{1}{N} \sum_{i=1}^N (d_i - \mu)^2$ is the sample variance or AC power of $\{d_i\}$.

It is also a common notion in signal theory and statistics that $P_{AC} = P_T - P_{DC}$. To speed up the computation a bit, it is straightforward to confirm that this holds for the circular difference signal. It is prudent to do this because the formal definition of sample variance uses a slightly smaller normalization $N-1$ in estimating the true variance. From the above definition of P_{AC} ,

$$P_{AC} = \frac{1}{N} \sum_{i=1}^N \left(d_i - \frac{1}{N} \right)^2 = \frac{1}{N} \sum_{i=1}^N d_i^2 - \frac{2}{N^2} \sum_{i=1}^N d_i + \frac{1}{N^2} = \frac{1}{N} \sum_{i=1}^N d_i^2 - \frac{1}{N^2} = P_T - P_{DC}$$

We mentioned that either P_{AC} or P_{DC} can be used to measure diversity within a gene. Here we examine the use of P_{AC} .

Three Normalizations , Three Diversity Measures

For the 3 reference points above, from most to least diverse, P_{AC} turns out to be:

$$P_{AC}^{(++)} = 0 \quad P_{AC}^{(+)} = 1/N^2 \quad P_{AC}^{(0)} = (N-1)/N^2$$

This suggests the following possible definition for the locus diversity D_k :

$$D_k^{Simple} \equiv \frac{1}{N^2 P_{AC}}$$

This merely inverts the Poisson-normalized variance discussed earlier, so that the metric goes down rather than up with decreasingly uniform scattering. It gives a fairly well-behaved D_k , at first glance. When the population is initialized, the metric under the *random scattering case* gives

$$D_k = \frac{1}{N^2 P_{AC}^{(+)}} = 1 \text{ and decreases thereafter. If the population should converge to the point of}$$

no diversity, the metric gives $D_k = \frac{1}{N^2 P_{AC}^{(0)}} = \frac{1}{N-1}$, which is close to zero for reasonable pop-

ulation size. If P_{AC} approaches $P_{AC}^{(++)} = 0$ due to some fortuitous set of circumstances, D_k^{Simple} shoots up toward infinity as a reflection of it's extreme unlikeliness.

This definition D_k^{Simple} suggests that the natural quantity to be transforming is the Poisson - normalized variance $N^2 P_{AC}$ in order to get a D_k that behaves correctly regardless of N . D_k^{Simple} is conceptually plotted against $N^2 P_{AC}$ in Figure 6.21. D_k^{Simple} can be made more "correct" by a vertical down-shift so that $D_k=0$ at $N^2 P_{AC}^{(0)} = N-1$. It must also be vertically stretched back up to maintain $D_k=1$ at $N^2 P_{AC}^{(+)} = 1$. This yields a "good normalization" modification to D_k^{Simple} :

$$D_k^{GoodNorm} \equiv \left(\frac{N-1}{N^2 P_{AC}} - 1 \right) / (N-2)$$

$D_k^{GoodNorm}$ is created strictly to clean up the "rough edges" of D_k^{Simple} and make it more intuitively satisfying. As in an indicator of diversity, it is not expected to differ significantly from D_k^{Simple} . (Note: The author acknowledges that $N^2 P_{AC} \rightarrow D_k^{GoodNorm}$ is not technically a normalization; but then, neither does the common term "fitness scaling" refer to a simple scaling).

Though D_k^{Simple} and $D_k^{GoodNorm}$ meet the algebraic sanity checks, there are some serious problems that can be anticipated from Figure 6.21. Any practical population size N is much greater than unity, so $N^2 P_{AC}^{(0)} = N-1 \gg 1 = N^2 P_{AC}^{(+)}$. That means $N^2 P_{AC}^{(0)}$ is on a very flat

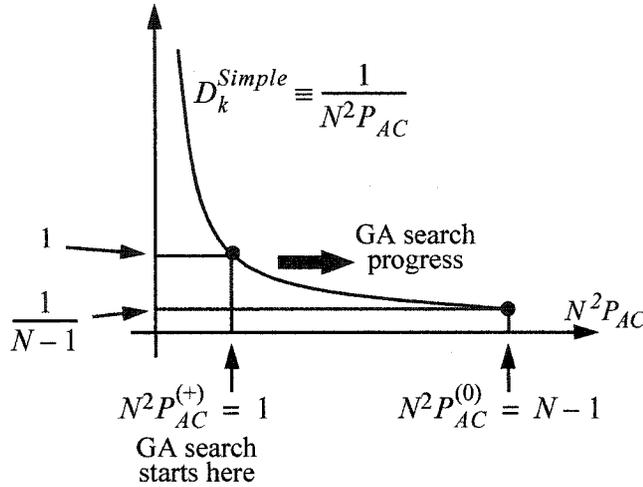


Figure 6.21. The simple gene diversity metric.

$$D_k^{Simple} \equiv \frac{1}{N^2 P_{AC}}$$

part of the curve; a significant portion of the dynamic range of D_k^{Simple} and $D_k^{GoodNorm}$ is consumed by the initial drop as $N^2 P_{AC}$ increases from 1, while much of the valid stretch of $N^2 P_{AC}$ beyond this regime will have little effect D_k . Hence, it was worthwhile to also try a scaling that wasn't motivated by algebraic simplicity. Instead, geometric simplicity suggests a linear $N^2 P_{AC} \rightarrow D_k$ mapping to avoid compression of D_k over much of the tail of $N^2 P_{AC}$, again maintaining $D_k = \{1.0, 0.0\}$ at $N^2 P_{AC} = \{N^2 P_{AC}^{(+)}, N^2 P_{AC}^{(0)}\}$. This linear interpolation mapping is also quite simple, algebraically:

$$D_k^{Linterp} \equiv \frac{N^2 P_{AC} - N + 1}{2 - N}$$

The metrics D_k^{Simple} , $D_k^{GoodNorm}$, and $D_k^{Linterp}$ measure the diversity of a single locus k . For a genotype with loci $k \in \{1, 2, \dots, l\}$, the diversity D of the population is mean of D_k over k :

$$D \equiv \text{mean}_k D_k$$

Depending on the D_k definition used, this generates the global metrics D^{Simple} , $D^{GoodNorm}$, and $D^{Linterp}$.

6.1.5.6 The Reference Measure for Comparison

The 3 diversity measures D^{Simple} , $D^{GoodNorm}$, and $D^{Linterp}$ will be compared with the diversity metric Div in [WO03], which is based on the all-pairs Euclidean distance:

$$Div \equiv \frac{1}{l} \sqrt{\sum_{k=1}^l \sigma_k^2}$$

where σ_k^2 is the variance in gene values at locus k . For the comparison to be meaningful, the normalization should be similar to the various definitions of D_k . Therefore, consider the values of Div and σ_k^2 at the start of the GA search, where values at locus k are *uniformly* distributed over $[0.0,1.0]$. This basic PDF has variance $1/12$. Under this *random scattering* condition, we can expect Div to take on the value $\sqrt{1/12l}$. Using this as a normalization, Div becomes

$$Div^{Norm} \equiv \sqrt{\frac{12}{l} \sum_{k=1}^l \sigma_k^2}$$

Like the 3 definitions for D_k , Div^{Norm} has the expected value of 1 under conditions of *random scattering* at the start of the GA search, and decreases with diversity. As with the D_k 's under conditions of *no diversity*, $Div^{Norm} = 0$ (since $\sigma_k^2 = 0$).

D^{Simple} , $D^{GoodNorm}$, $D^{Linterp}$, and Div^{Norm} are compared on the final GA in Section 7.2. They are then calibrated based on that comparison.

6.1.6 Implementation Environment

Matlab was initially chosen to implement the GA for the simplified placement problem. This reasons were because it allowed quick and powerful coding, as well as interactive monitoring and changing of the execution environment. It also attempts to rival the speed of 3rd generation languages (3GLs) like C by optimizing for some vector operations and automatically converting some loops to vectorized operation (process called *acceleration*).

Unfortunately, these mechanisms are still under development. At the time of this investigation, they are broken by many combinations of operations and control flow structures. As the underlying optimization problem was made more complex, more intricate operations were needed, requiring extensive experiments with round-about coding styles to exploit vectorization and maintain

acceleration. This detracted from the high-level advantages of the Matlab, and hurt the code readability and evolvability.

One of the most debilitating limitations was that function calls break acceleration. It was therefore impossible to modularize the code for reusability or to hide/manage complexity. This breaking of acceleration applied even to accelerated code modules written in C or C++, thus defeating the purpose of coding bottlenecks in 3GL unless the entire program was written in 3GL. Therefore, the Matlab implementation took the form of enormous loops of unmanageable inline code. This very form of coding exacerbated the penalty of breaking acceleration, since a break anywhere within the loop prevents acceleration throughout the entire loop iteration.

Examples of procedures that are highly nonvectorizable include the evaluation of a chromosome's cost, as well as PMX's repair. Within these routines, there is a lot of dependence of program flow control on the data (mostly table lookup results or operations thereon). In the case of cost evaluation, the same procedure would take place across different GA schemes to be investigated, so a meaningful assessment is still possible. However, the PMX's nonvectorizable repair was specific to PMX; in contrast, the sort required for RKR is highly optimized in Matlab, making comparison difficult.

Despite the nonvectorizable portions, the simplicity of the basic ordering problem made much of the implementation vectorizable or acceleratable, with some effort and for a price, as described above. As the heterogeneous constraints of the real platforms are incorporated into the ordering problem, however, much of the chromosome decoding would become nonvectorizable. These include testing for positional conformity to tile bindings, and heuristics to rearrange the DPU ordering so that required operations are matched with properly equipped DPUs.

Due to the mismatch of Matlab's strengths with the computation needs above, it was very clear that the full placement problem had to be migrated to a 3GL. For this thesis, C++ and its recently maturing *Standard Template Library* (STL) was selected. The STL contains functions to apply standard algorithms like sort, merge, uniquify, and differencing to standard data structures like vectors, lists, and maps. It differs from conventional libraries in that it is based on *generic programming*; this refers to the practice of coding in a way that the data underlying the data structures can be of any type and complexity. This made it appealing for the aggregate data types of arbitrary complexity, such as chromosomes.

Because the STL makes extensive use of C++ features to support generic programming, it resembles a meta language for which there are extensive protocols. However, the library has also

been recently subjected to global efforts in standardization, thus guaranteeing a consistent body of knowledge in its usage. The specifications generally favour fast execution at the expense of graceful behaviour in the event of violations of the protocol.

6.1.6.1 Tool Flow Transition

Before the GA can be applied to the sample kernel of Chapter 4, the kernel has to be rendered into a flat netlist of components on the model platform. The kernel design was hierarchically specified in Verilog. Flattening this would be manually intensive and error prone. As part of the tool flow, what is needed is a CAD environment that can read in the hierarchical structure, then traverse and manipulate it. In addition to flattening the netlist, this includes removal of all components besides DPUs, and nets that do not specifically connect DPUs to DPUs. For example, all local memories, connections to logic high/low, control logic, etc. Furthermore, the 32-bit buses must be reduced to “single-wire” abstract connections.

The most suitable environment for this is Synopsys Design Compiler (SDC), a tool normally used for logic synthesis in an ASIC design flow. The use of SDC for manipulating the sample kernel needed to be investigated at length because it is unconventional. Initially, the tool flow transition can be handled partly by SDC scripts for hierarchical traversal and pruning, and partly by a series of text filtering steps applied to the text file output. The filtering can be initially performed manually using the advanced regular expression capabilities in [Oua01], and eventually automated using the Perl text manipulation package [Per].

6.1.7 Example: Simple DPU Ordering for FFT

For the following examples, a successful search is one that finds a solution where no wires exceed the local wire length. In the Chameleon platform, this spans 8 DPU positions (Figure 5.1 is a simplified example; see Figure 4.22). Wires that exceed this are referred to as *wire (length) violations*.

6.1.7.1 PMX

The basic ordering problem of Figure 6.3 was implemented with steady-state replacement, and ranked fitness. Being steady-state, there was one mating per “generation”. The population was $N=200$. PMX crossover was used.

Mutation rate was specified as the probability that *both* offspring will undergo mutation. Mutants were pooled with unmutated offspring to form a group of kids. Note if that we just randomly mutate one gene, the resulting chromosome would no longer be a permutation of all the DPU indices. To avoid this, mutation was applied by swapping a pair of randomly determined genes.

In the replacement scheme, twins and clones were first eliminated. The remaining kids were then merged with the general population, and the best N were retained. This contributes greediness to the search compared to the less greedy policy of replacing source chromosomes. Replacement policies that impart intermediate greediness to the search include unconditional entry into the population by displacing the source chromosomes, random members, or worst chromosomes.

The results are shown in Figure 6.22. Greater mutation leads to faster convergence. There also is a sudden drop in successful simulations when there are zero mutations [plot (e)]. This underscores the criticality of mutation for nonbinary genes. We also see that despite its benefit for convergence time, the amount of mutation doesn't really matter much to the probability of achieving successful convergence. (By successful convergence, we mean that a feasible solution is found.)

In obtaining the data in Figure 6.22, there is some arbitrariness in defining nonconvergence. In this test case, we decided that the search had not found a solution in a practical amount of time if one of two conditions are encountered. The first is if the number of matings exceeded 200×10^3 , whatever the reason. The second is if none of the kids were good enough to make it into the population in 3000 consecutive matings; in this case, we assume premature convergence and lack of genetic material to escape local minima. This mechanism typically kicked in well into the simulation i.e. above 100×10^3 matings. Regardless of which nonconvergence criteria was met, nonconvergent cases are excluded from the run times and mating counts for finding a solution, as well as the scatter graphs of execution times.

From the tapering off of the scatter graph (c) near the top, it is clear that the definition of nonconvergence was adequately liberal to capture most of the data points and properly characterize the GA parameters. If the termination conditions were more stringent, we may get much greater runtimes and negligibly better success rates. If the termination conditions were relaxed, we may get apparently shorter run times, but fewer successful searches.

The scatter graph also reveals the reason for the lack of dependence of the success rate on mutation rate (beyond 0% mutation). The reason is because the vertical scatter in runtimes is

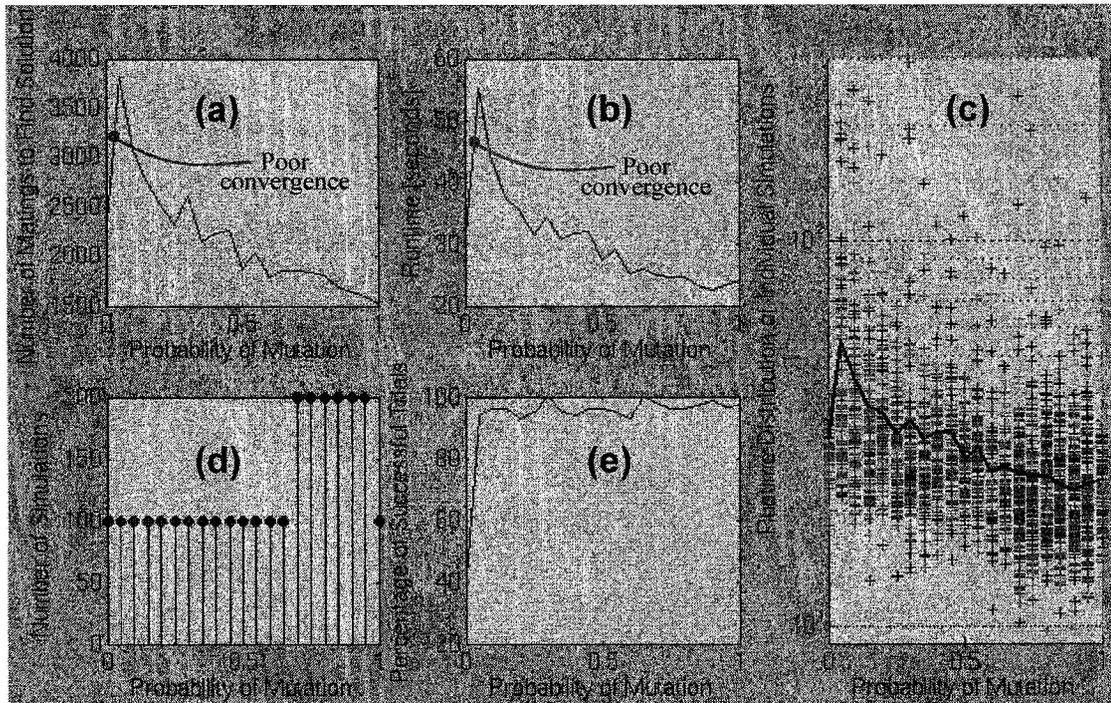


Figure 6.22. The basic ordering problem in Matlab using PMX and steady-state evolution.

(a) and (b) show the average number of generations and runtime for searches that *successfully* found a solution. Note that the sudden drop off for 0% mutation does not reflect the large runtimes of the many unsuccessful searches (see plot (e)). Scatter graph (c) shows the distribution of individual run times for successful searches. The number of searches tried (plot(d)) is doubled for some runs because of extra data from a previous set of runs that was interrupted.

much greater than the average improvement of runtime with mutation rate. The limits on simulation time are high enough that there are very few data points, and very little difference in the number of simulations truncated by termination. As might be obvious from the remarks on termination conditions, lowering the ceiling on simulation times toward the crowded portion of the scatter graph would yield a more pronounced dependence of success rate on mutation rate; in fact, greatest variation on success rate will be seen if the limit on computation time cuts right across the densest part of the scatter graph.

The extreme bottom-heavy distribution Figure 6.22(c) is good because it means most searches will converge on a solution quickly. The average is distorted by the fact that the outliers are quite high, even with log scaling. This means that we can terminate searches after 100 seconds and still have most runs find a solution. Therefore, bottom-heaviness implies that trying multiple short runs is better than allowing a single long run. For example, say the search time is limited to a generations count such that there is 95% chance of finding a solution. In two such searches, there is a 99.75% chance of finding a solution.

For the purpose of tuning the mutation rate, it appears that more mutation is better. This corresponds to local search in the vicinity of offspring. The improvement must be interpreted with caution. When mutation rate is 100%, there are twice as many kids as when mutation rate is near zero. The extra processing may not be reflected in Figure 6.22 because much of the execution time is taken by intergenerational overhead e.g. to monitor population statistics. This hiding of chromosome processing time is can be amplified by the use of Matlab, since intergenerational overhead can be high; the generation loop is at a fairly high level, with M-file invocations that break acceleration. Due to the penalty of function calls, there is greater efficiency from processing 4 kids per generation rather than 2. Conceptually, this can be viewed in simple algebraic terms by considering the dominant, nonacceleratable contributions to computation time. Let

$P_{Mut} \in [0.0, 1.0]$ = Probability of mutation

$M(P_{Mut})$ = Number of Matings (or generations)

t_{Eval} = Execution time to Evaluate one chromosome

t_{PMX} = Average execution time of PMX repair,
incurred by each offspring but not mutants

t_{IG} = Execution time for InterGenerational overhead in one generation, including
performance penalties from function calls

Since there 2 offspring and an average of $2P_{Mut}$ mutants per generation, the execution time can then be expressed as

$$T = M(P_{Mut}) [t_{IG} + 2(t_{Eval} + t_{PMX}) + 2 P_{Mut} t_{Eval}] . \quad (EQ 1)$$

The decrease of $M(P_{Mut})$ with P_{Mut} (Figure 6.22) will have a large effect on T if $t_{Fixed} = t_{IG} + 2(t_{Eval} + t_{PMX})$ is significant compared to $2P_{Mut}t_{Eval}$. If t_{Fixed} can be minimized, then the increase of $2P_{Mut}t_{Eval}$ with P_{Mut} counters the decrease of $M(P_{Mut})$. This can mitigate the

decrease in T with P_{Mut} , or even cause T to increase with P_{Mut} (though T will still be minimized if $2P_{\text{Mut}}t_{\text{Eval}}$ is). In Matlab, the high penalty for function calls boosts t_{IG} and t_{Fixed} , and can hide some of the mitigating effect of $2P_{\text{Mut}}t_{\text{Eval}}$ on T 's decrease with P_{Mut} . t_{Fixed} is less severe in a 3GL implementation because there is not the same high penalty from function calls.

6.1.7.2 RKR

Initially, there were two main reasons to try RKR. One was to avoid the cost of PMX repair. The second reason was the hope that RKR would have better convergence because it avoids the large-scale disruptions in repairing the genes outside of the crossover region. It later became apparent the random keys could be used as sloppy position indicators in constraining the placement of DPUs (Section 6.2.2.1).

Appendix D presents the examination of the search dynamics in making the GA work with RKR. We present the final results in Figure 6.23. The key change was to mutate general population members rather than offspring. The achievable and average run time in Figure 6.23 is less than with PMX in Figure 6.22, and the success rate is 100%. Furthermore, the convergence failures in Figure 6.22 represent a significant amount of time not reflected in the run time curves.

The GA for Figure 6.23 applies mutation to general population members rather than offspring. Recall that for this kind of mutation, mutation rate is taken to be the portion of the population that will be subjected to mutation in each generation. Thus, the number of mutants per generation can be significantly more than the number of offspring, and can approach the size of the population. Hence, a lower matings count does not automatically imply a more efficient search. For a population size of N_{Pop} , (EQ 1) must be rewritten as

$$T = M(P_{\text{Mut}}) (t_{\text{IG}} + 2 t_{\text{Eval}} + t_{\text{Eval}} P_{\text{Mut}} N_{\text{Pop}}) . \quad (\text{EQ 2})$$

The run times in Figure 6.23 increase with mutation probability, which differs from 6.22. There are two contributions to this that we can identify. The first relates to minimizing t_{Fixed} . This reduction results from not having to perform PMX repair, which is highly nonvectorizable. Thus, there is no longer a t_{PMX} contribution to t_{Fixed} in <2>, and P_{Mut} has a larger effect because of multiplication with N_{Pop} . Even though $M(P_{\text{Mut}})$ decreases with P_{Mut} for most of P_{Mut} 's value range, T increases because

$$t_{\text{Eval}} P_{\text{Mut}} N_{\text{Pop}} > t_{\text{Fixed}} \equiv t_{\text{IG}} + 2t_{\text{Eval}}.$$

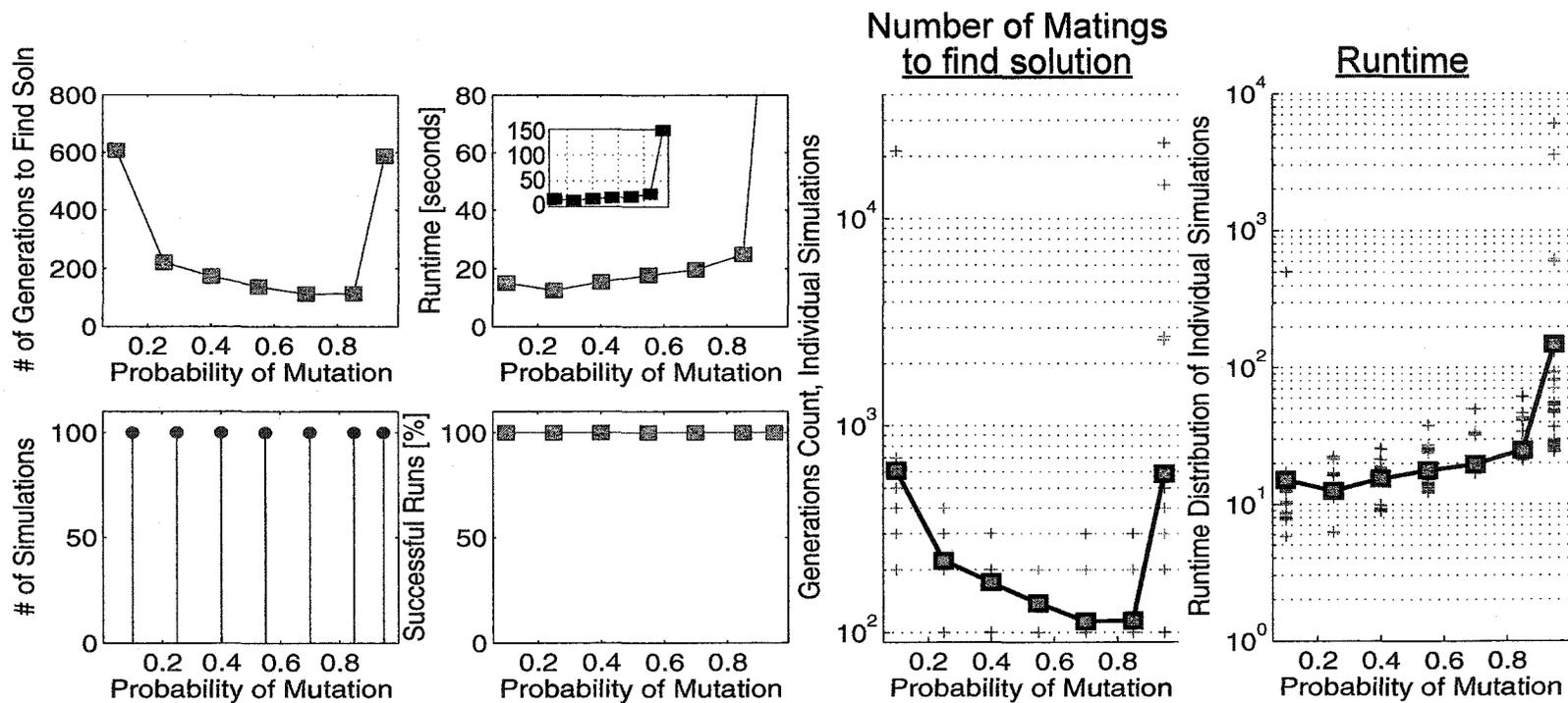


Figure 6.23. RKR, steady-state evolution, 2-point crossover, mutation of general population members.
NOTE: Runtimes are shown only to indicate trends and give an idea of the order of magnitude. They are affected by the exact computing platform (Sun Blade 100, solaris 8) and other tasks that may be running at the same time.

As $P \rightarrow 1$, the second contribution to increased run time takes effect: $M(P)$ increases with P . This is because the kids are unconditionally injected into the population; as mutation approaches 100%, the entire population is being replaced at once by a generation of mutants. This resembles the lack of directedness in generational replacement; aside from the information carried by the kids themselves, we throw out all the information in the current generation, embodied by the distribution of sampling points among the different hyperplanes. According to the Schema Theorem, this information should be contained in the replacement population; however, that is based on a replacement population consisting of offspring rather than dominated by mutants. It also presumes binary genes.

Figure 6.23 suggests that the greater directedness of a lower mutation rate is better. Intuitively, the greater directedness from less mutation can be viewed giving more opportunity to newcomers to interact with existing sampling points; this leverages the existing information about promising building blocks and good regions in the search space.

The run time scattering appears much more even in Figure 6.22(c) than in Figure 6.23. This is not necessarily significant. Many of the run times in Figure 6.23 are almost identical, so the data points are almost coincident with each other. We attribute the variability in the run times of Figure 6.22(c) partly to the nonvectorizable PMX repair, which may require a variable number of passes to resolve all collisions in each offspring.

Effects of Selective Pressure

As described in Section 6.1.4.3, an offset was added to ranked fitness such that the probabilities of selecting the best and worst genome was of the ratio $P_{b2w}=10$ (Figure 6.24). The selective pressure is lower than for regular ranked fitness, where P_{b2w} would be the population size (200 in this case). Due to varying workstation load, the disparity in run times are not necessarily significant. The run time curves merely give an idea of the range for run times. The generations counts show the convergence to be apparently similar for both cases. For significant mutation rates, however, the likelihood of a successful search is better when selective pressure is lower. A possible explanation is that mutation confers greater genetic diversity to the population, but it is the lesser selective pressure that allows this diversity to be used for a more thorough search.

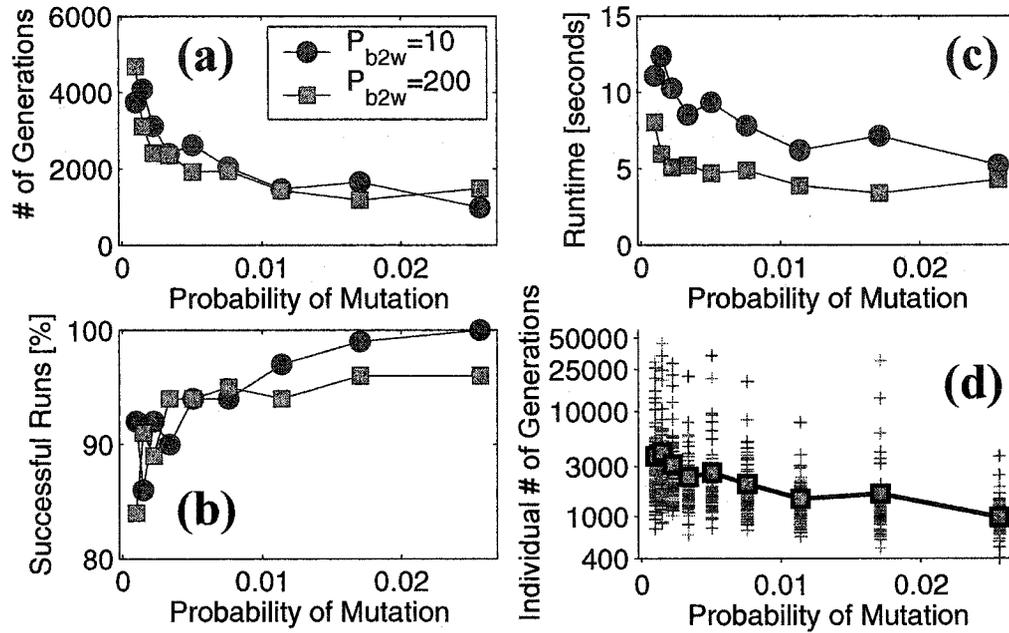


Figure 6.24. Controlling selective pressure in ranked fitness.

6.1.7.3 The Importance of Mutation

Because of the marked effect on RKR of mutating the general population, we investigated the use of mutation alone, without parental selection and mating, for both RKR and PMX. The comparisons are shown in Figure 6.25, which uses the following notation:

- ◆ **X+MGPM:** 2-point crossover + mutation of general population members
- ◆ **O_MGPM:** Only mutation of general population members
- ◆ **X+MOO:** 2-point crossover + mutation of offspring only

Surprisingly, MGPM is nearly the same as with and without crossover. Of course, when mutation approaches 0%, there is no evolution and hence, no convergence. This was confirmed in a more detailed scan of mutation probabilities. For significant mutation rates in linear placement of a netlist with local wire length limits, evolution due to mutation alone appears to be approximately as good as conventional GA with mating.

For the GAs Figure 6.25 that use only mutation without crossover (O_MGPM), the reason for the increase in $M(P_{Mut})$ as $P_{Mut} \rightarrow 1$ is not apparently explained by the rationale given for

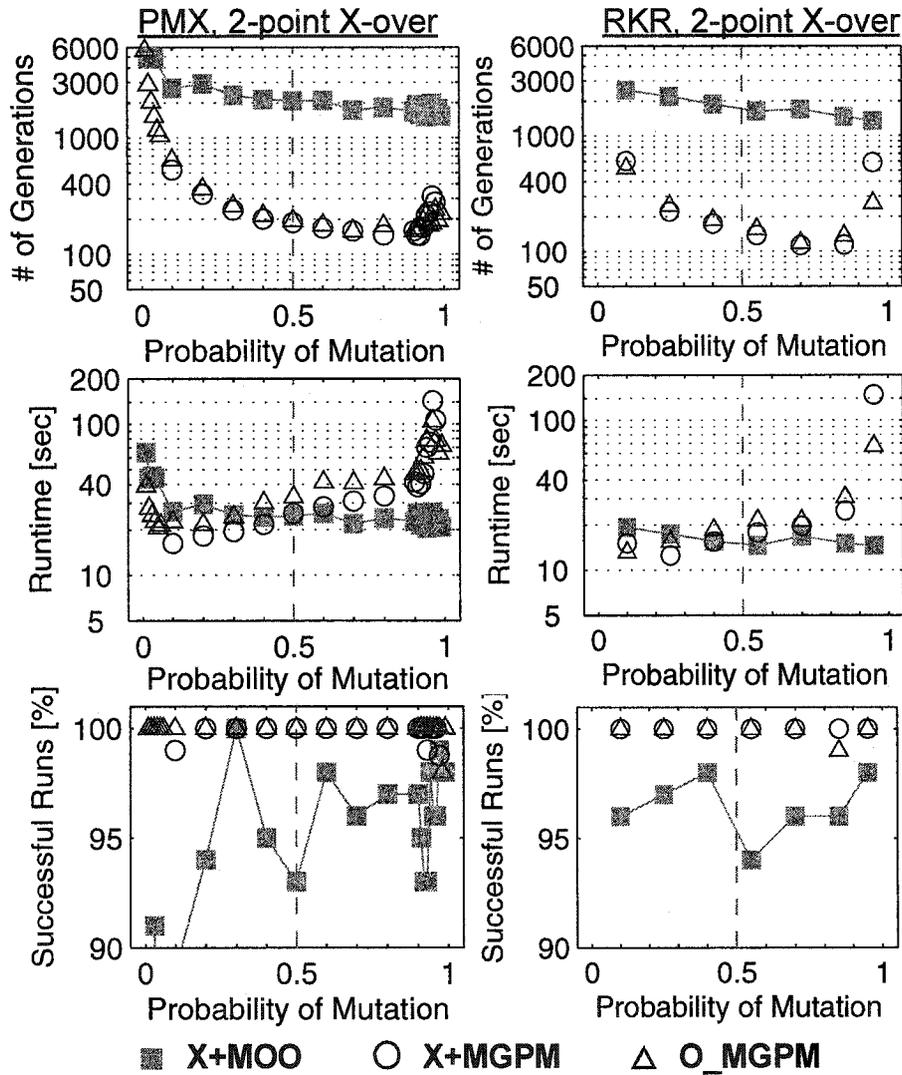


Figure 6.25. Simple DPU ordering problem: Comparison of different mating/mutation schemes for PMX and RKR.

Figure 6.23. In Figure 6.23, $M(P_{Mut})$ decreases as P_{Mut} does because the search becomes more directed. Intuitively, smaller changes in the population allow more potential interaction between kids and existing sampling points, rather than throwing out all information between generations. In Figure 6.25, there is no mating, and therefore, no interaction between chromosomes. There is not even the interaction of competing fitnesses in being selected for mutation, since population members are selected with uniform probability for mutation. Neither is there any *direct* fitness-

based competition during the injection of the kids into the population, since acceptance was unconditional.

Because of the apparent lack of selective pressure, the increase in $M(P_{Mut}) P_{Mut} \rightarrow 1$ seems counterintuitive at first. However, there *is* some kind of competition happening in the replacement scheme; it is in choosing the worst population members to displace. If this is done a few chromosomes at a time, the good chromosomes linger and contribute to further mutations. If the entire population is changed at once, however, there is no selective pressure from the preferential lingering of better chromosomes; the search relies completely on the chances that a random perturbation on every sampling point will improve the population. $M(P_{Mut})$ increases because this “brownian motion” is no better in searching than exhaustive enumeration. On the contrary, it is much worse because the genotype space is real rather than combinatorial. Furthermore, there is nothing to prevent a point in the search space from being visited more than once.

The search schemes in this thesis mostly accept unique kids into the next generation unconditionally. Appendix D.1.1.2 explores characterizations in Figure 6.25 using *competitive* acceptance of kids in updating the population.

6.2 Full Placement Problem

The random keys implementation was redone in C++/STL to deal with the nonvectorizable heuristics that would be needed to handle the heterogeneous constraints of the actual reconfigurable platform.

6.2.1 Insights from C++/STL Implementation

6.2.1.1 Artifacts Due to Ranking and Cost Compression

One of the sensitivities of ranking that needed attention was the loss of differentiation as cost decreased and homogenizes throughout the population. In this case, many chromosomes would have the same cost. This would not be reflected in the rank, which artificially creates a fitness difference of 1 between adjacently ranked chromosomes. Which chromosomes received this artificial advantage is dependent on the details of the sort routine used, but there was a strong possibility that it would induce a systematic bias in the selection for mating.

To avoid this, it was necessary to randomly shuffle the ranking between chromosomes with the same cost. This was done in every generation. Since steady-state evolution was used, this ensured that the bias was randomized between matings. This was not done in the Matlab implementation because the emphasis was on doing everything possible to cut down computation time, especially if vectorizability was not apparent. The fragility of acceleration turned any new undertaking into a significant effort in code optimization.

Since all chromosomes with the same cost are stored adjacently, it was computationally simpler in C++/STL to realize the effect of shuffling by selecting a chromosome with the roulette wheel. This selection step only identifies the swath of chromosomes with the same cost, from which the final parent is selected with uniform probability. This also had the benefits of shuffling between the selection of *each* parent in a single mating.

6.2.1.2 Migration of Basic Ordering Problem

The C++/STL implementation of RKR used *uniform* crossover. During its code development, this was found to get less infeasible solutions in the face of the more demanding constraints, to be discussed next (Section 6.2.2). The simple DPU ordering problem was first studied in greater detail for the 3 cases X+MOO, X+MGPM, and O_MGPM. The faster runtimes with C++/STL made it possible to explore extreme values for the probability of mutation, which exhibit very poor convergence. This allowed for the identification of regimes for the onset of good and bad convergence. The results (Figure 6.26) confirm that mutation of offspring (X+MOO) is not as good as mutation of general population members, both in terms of successful searches and how quickly a solution is found. As well, the performance of the latter is not much affected by whether crossover is performed.

Despite the fact that X+MOO requires more than an order of magnitude more generations to find a solution, the actual search time is only several times more than X+MGPM and O_MGPM. This is because there are far fewer mutants and kids per generation in X+MOO. At 100% mutation rate, X+MOO generates 2 mutants per generation (1 per offspring), whereas the number of mutants in X+MGPM and O_MGPM is equal to the size of the population.

6.2.2 Heterogeneous Constraints and Features

The guiding idea in synthesizing a means to handle constraints is to avoid the introduction of unnecessary infeasibility into the design space. This may involve modifying the raw solution to

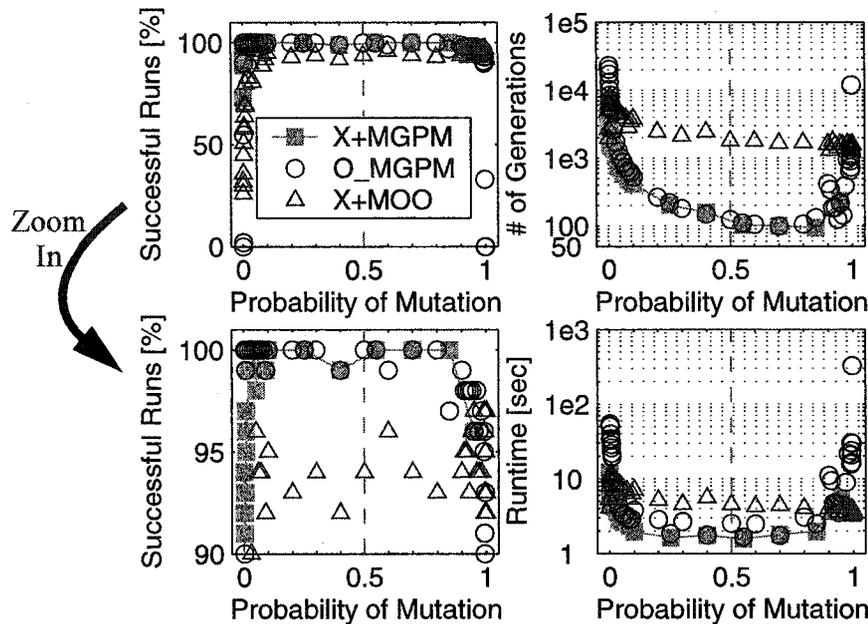


Figure 6.26. Comparison of differing mating/mutation schemes using C++/STL implementation.

X+MGPM: Uniform crossover + mutation of general population members

O_MGPM: Only mutation of general population members

X+MOO: Uniform crossover + mutation of offspring only

comply with constraints. Such “repairs” map an infeasible solution to feasible one, and should not cause them to be vastly different i.e. the perturbation of the chromosome in the search space due to repairs should be small. We want to rule out the possibility of deceiving the search if the repairs are too numerous compounded, or too radical, so that the final solution is only remotely related to the source chromosome.

6.2.2.1 Binding Datapath Units to Tiles

Recall that specific DPUs may be bound to certain tiles because of proximity to resources e.g. banks of memory to which the DPUs connect. For the purposes of illustration in this section, we will assume a reduced architecture of $N_{tiles}=3$ tiles/slice, and 4 DPUs/tile.

Straight Penalization

The simplest way to restrict certain DPUs to certain tiles is to simply penalize solutions in which DPUs do not conform to their tile bindings. However, this would render vast portions of the search space infeasible.

For RKR, straight penalization would be harmful in yet another way. Since the random keys can be seen as sloppy specifications of position, there will also be selective pressure for bound DPUs to be positioned away from the edges of their home tiles. However, it was found from the manual placement effort that DPUs bound to upper and lower tiles still competed to be close to the central tile simply because *everything* “wanted” to be close to everything else. This can be viewed as an attraction of DPUs toward the center of the array in order to avoid violating the local wire lengths. In fact, for problems that push resource limits, solutions often have DPUs very close to infeasible regions.

Fixing Equidistant Key Values for the Tile Boundaries

The better solution seems to be to define a mapping from the raw solution to a feasible solution. Thus, a temporary feasible solution is created in the process of evaluating cost. There are many possible ways to do this. For example, the random key value for a bound DPU might be mapped to a smaller value range corresponding to a tile (Figure 6.27). The mapping allows the raw key to have the same value range as an unbound DPU. This simplifies crossover and mutation.

However, a DPU’s random key, taken in isolation from other DPUs in the chromosome, is only a very rough indication of the DPU’s position. Because of this, it is not possible to determine the exact subrange for the mapping. In a fictitious world, the keys values would be evenly distributed in range $[0,1]$, and the tile boundaries would correspond to key values $1/3$ and $2/3$ for a 3-tile platform. A DPU can be bound to a tile by simply attenuating its key value to the range $[0,1/3]$, then offsetting it to coincide with the desired tile. A real chromosome’s key values, however, are arbitrarily distributed. Fixing the key values for the tile edges would result in a random number of DPUs within each tile. This is regardless of whether any DPUs were even bound to tiles.

Interpolating Key Values for Tile Boundaries

An alternative to fixed tile boundaries is to sort the unbound DPUs, then combine them with the bound DPUs in each tile in such a way that the tile capacities are not exceeded. A simple approach is to interpolate between sorted unbound DPUs to obtain the key value of a tile edge

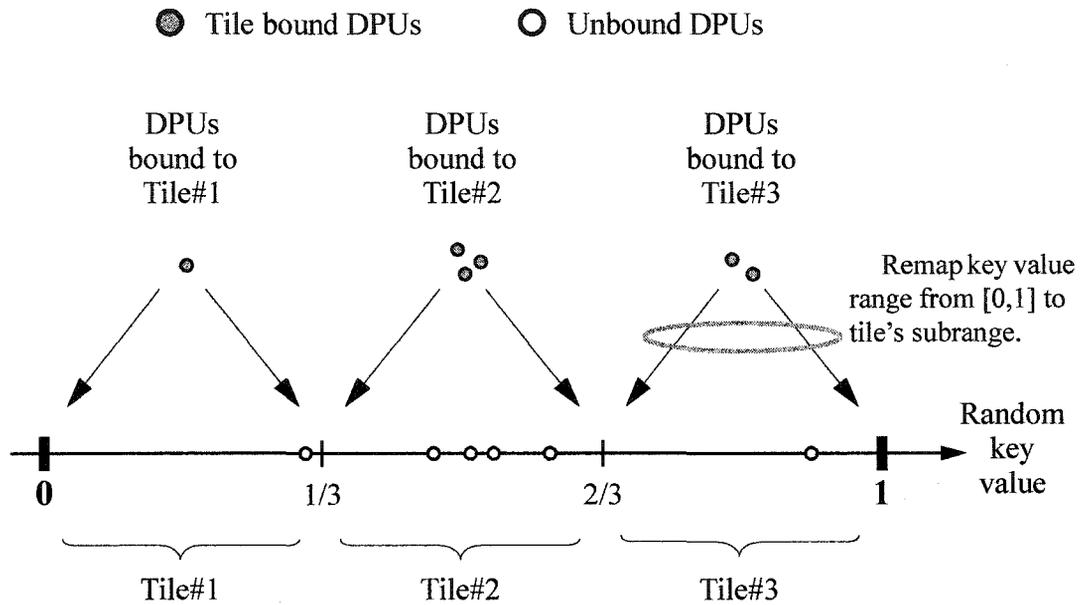


Figure 6.27. Initial approach to binding DPUs to tiles, assuming 4 DPUs/tile and $N_{\text{tiles}}=3$ tiles.

Each bound DPU's key value is linearly mapped from the original range $[0,1]$ to the home tile's subrange. Tile subranges are rigidly taken to be delineated by multiples of $1/N_{\text{tiles}}$. This may overfill some tiles.

(Figure 6.28(a)). Once the tile edge values are known, the keys for bound DPUs can be linearly mapped from $[0,1]$ to the value range for a tile. The bound and unbound DPUs within that tile can then be sorted again for intra-tile ordering.

Figure 6.28(b) shows how this can lead to pathological positioning of tile bound DPUs. Assume 3 tiles of 4 DPUs each, totalling 12 DPU positions. Say a kernel has 1 bound DPU in each of tiles 1 and 2. There are 10 unbound DPUs sorted by key value. Since tile#1 has 1 bound DPU already, the first 3 unbound DPUs will go into there. The other tiles are filled with unbound DPUs in similar manner. The remaining problem is where to place the bound DPU(s) for a tile among the sorted unbound DPUs in that tile. One way is to get the key value at (say) the 1st tile edge by interpolating between the last unbound DPU of the first tile and the first unbound DPU of the 2nd tile. Once all the tile edges are determined in a similar fashion, then the keys for the bound tiles can be mapped into their tiles' subranges.

The problem lies in assuming that the tile edge lies half-way between the pair of keys for the straddling DPUs. This is a very arbitrary and unjustified restriction in the search. In Figure 6.28(b), the keys in tile#2 are evenly spread out, whereas the unbound keys in tile#1 are concentrated near 0.1. Taking the tile boundary as the midpoint between the last DPU in tile#1 and the first DPU in tile#2 leaves much more room for tile#1's bound DPU to fall outside of the unbound DPUs rather than among them. In particular, the bound tile is far more likely to fall on the high key value side of the unbound DPUs. This problem is exacerbated by the fact that a design doesn't even have to have a full 12 DPUs, and may have significantly fewer DPUs. In that case, which unbound DPUs go into which tiles, even for the purposes of determining tile boundaries?

Aside from the problem with having fewer than 12 DPUs, it may be argued that it is the role of the GA is to cultivate the above bias in tile edges in such a way as to favour good solutions. However, this bias is not only arbitrary, but also not randomized to compensate for the arbitrariness. For this reason, it may be viewed as a *systematic and poorly justified restriction* on the search. Furthermore, it *cannot* be randomized because it is part of the mapping that decodes the solution. That mapping must be completely deterministic. Otherwise, the whole idea of cost is meaningless, since it changes every time the chromosome is evaluated.

Floating Positions for Tile Boundaries

The final solution was to revert back to fictitious world depicted in Figure 6.27; for 3 tiles, the tile edges would be *assumed* to have key values of $n/3$, where n is the tile number (Figure 6.29). The key value for bound tiles can take the full range $[0,1]$, but for the purposes of evaluation, they are mapped to the subrange for their home tiles. When the keys for *all* DPUs are sorted together, there may be more or fewer DPUs within the tile's key subrange than can physically fit in a tile. We then choose to look at this in another way; the tile boundaries, as defined by the assumed key values above, may not actually be located exactly 4 DPUs apart. If we regard the *physical* tile edges as imaginary markers that occur between DPUs and spaced 4 DPUs apart, then the *key-based* tile edges and *physical* tile edges fall between different DPU pairs. We penalize the distance by which they are offset in the same manner that overlength wires are penalized i.e. by first quantifying the distance in terms of integral DPU slots. Selective pressure then gently guides the search into regions which respect the assumed tile boundaries.

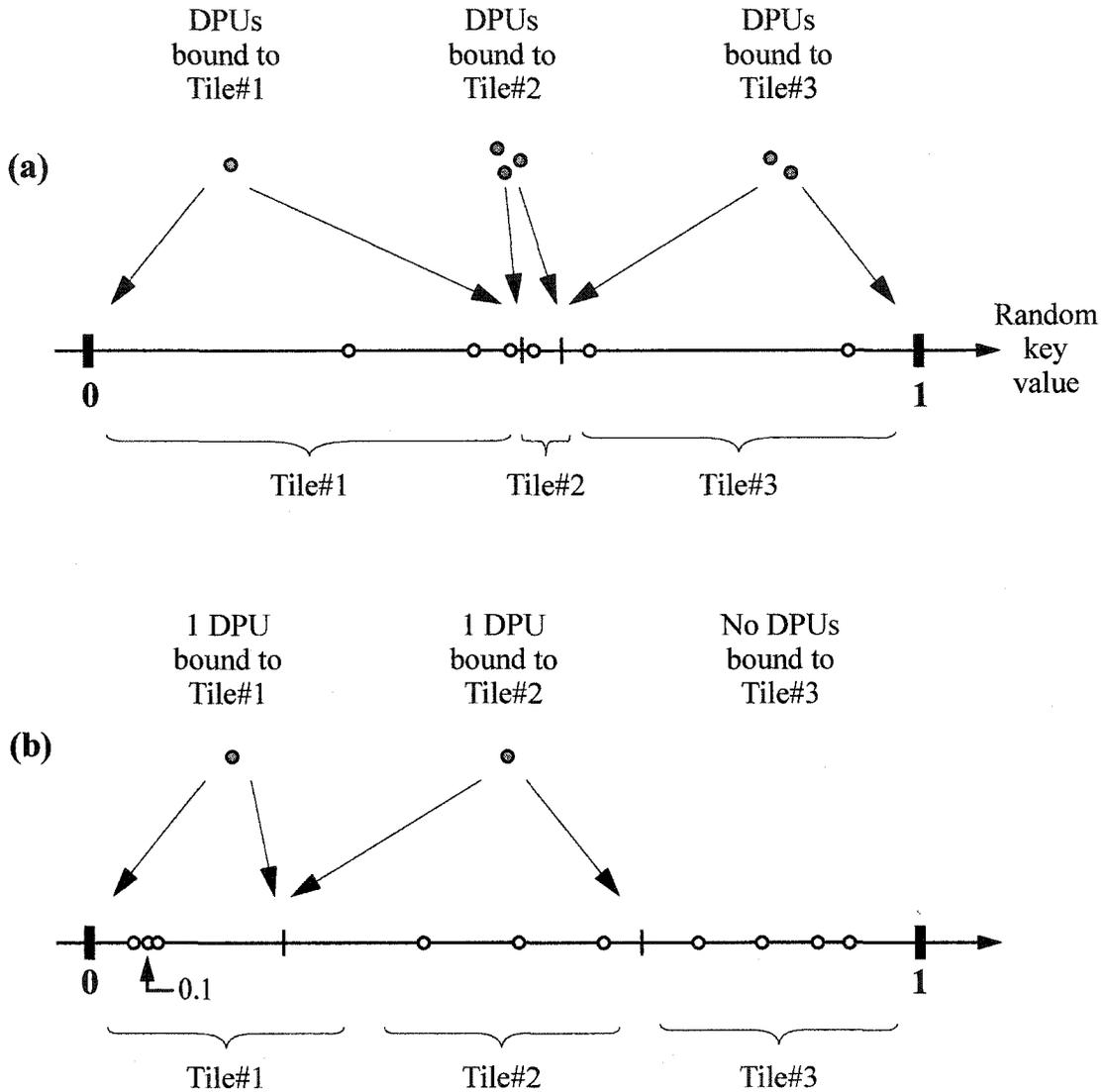


Figure 6.28. Tile subranges delineated by interpolating between unbound DPU. Architecture assumes 3 tiles of 4 DPUs each. The number of unbound DPUs in each tile is 4 minus the number of bound DPUs for that tile. (a) Typical delineation. (b) Pathological delineation.

It can be seen that our approach favours solutions with key values $n/3$ occurring 4 DPUs apart. This only make sense if there are a full 12 DPUs in the design. To ensure this, we add dummy DPUs to the designs with fewer DPUs.

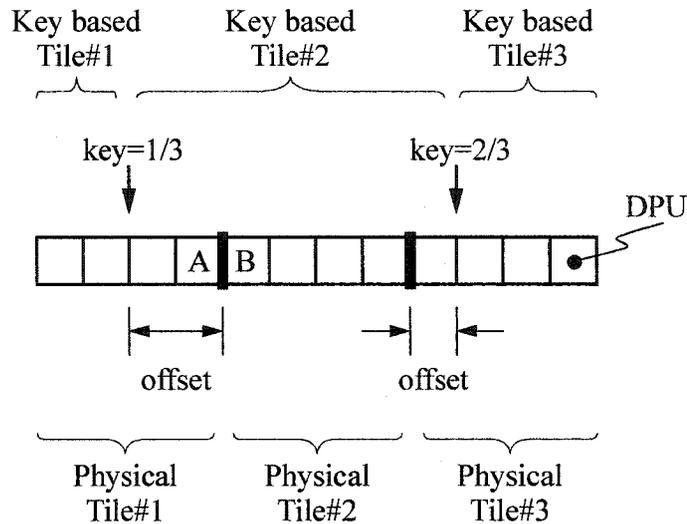


Figure 6.29. Floating key-based tile boundaries and physical tile boundaries for the case of Figure 6.27(a), *after* bound DPUs have their keys remapped to the *fixed* subranges of their home tiles, and all DPUs are combined and sorted.

The positions for key values of $1/3$ and $2/3$ are found by a binary search of the DPU keys. Hence, each of these key based tile edges represents the position where a fictitious DPU would be inserted if it had such a key value. These key values are not necessarily midway between the keys for the straddling DPUs.

Because we penalize the drift of $n/3$ values away from their ideal positions, this approach is a hybrid between mapping of gene values to valid ranges and defining infeasibility regions to penalize. The hope is that the shores of the infeasibility regions are adequately fuzzy as to permit a lot of search to take place there. To see this, first denote the key values of DPUs that straddle a physical tile edge as *straddling keys* e.g. in Figure 6.29, the keys for DPUs *A* and *B* form a pair of straddling keys. The fuzziness results from the fact that the physical tile edge locations are determined by DPU ordering, which in turn is dependent on their key values in a sloppy way. Furthermore, once the ordering is established, there is a range of values that straddling keys can take, and yet still be straddling the key-based tile edge (thus avoiding penalization). As mentioned above, man-

ual experience showed that pushing the boundaries of resource restrictions translates into flirting with infeasibility; this is facilitated by fuzzifying the criteria for correctly positioned tile edges.

In this hybrid method, there are biases, but they can be justified. The bias is such that DPUs that are destined for a particular tile will likely have key values distributed in the corresponding subrange. This ensures a reasonable degree of scatter of DPU key values throughout the entire range $[0,1]$, and therefore a stronger relationship between the key values and their actual positions. A key's sloppy position specification is thus made less sloppy, since the tiles in which the tile in which the DPU will be placed is predictable. For example, there is little chance of a pathological chromosome with good fitness, but having genes values within 0.001 of each other.

Departure from Purely Constraint Satisfaction to General Optimization

It is important to note that positional disparity between key-based and physical tile boundaries do not necessarily indicate an infeasible solution. Tile bindings are upheld as long as tile bound DPUs are not positioned in the wrong physical tile. The key-based tile edges are fictitious. We only want to align them with the physical tile edges in the course of the search because it removes the *possibility* of having a tile bound DPU positioned outside of its home tile when its key is remapped to the tile's subrange. By penalizing the tile-edge offsets, we are penalizing the *precondition* to tile binding violations rather than the violations themselves. However, this means that a feasible solution can now have nonzero cost. Thus, the problem changes from purely constraint satisfaction to general optimization. The algorithm must be modified to recognize that a solution has been found based on zero violations of constraints rather than zero cost. However, if a zero cost is obtained, it is clear that there are also no violations.

Alternative Penalization Scheme

The decision to penalize disparities in the tile edges was only partly arbitrary. There was also the option to count the number of DPU keys within each key-based tile, then impose a penalty if there were not exactly 8 DPUs. We denote the number of DPUs within a tile as the tile's *size*. Various scenarios can be drawn to show the pros and cons of each, but the scenario which motivated the selection of the current penalization scheme was that *locally* misplaced DPU results in a double penalization based on tile size. For example, a single position mismatch between key-based and physical tile edges would make the tile on one side too big and the tile on the other side too small. This seemed like excessive penalization for what can be seen as a single violation.

Scenarios that view the penalization of tile size in similar favour to penalization tile edges involve nonlocal DPU displacements. For example, consider a solution with perfectly balanced tile sizes and zero drift in key-based tile edges. If a DPU was moved from the top tile to the bottom tile, it would cause a single position offset in two key-based tiles edges with respect to their corresponding physical tile edges. Thus, it can be argued that there is a double penalization regardless of the penalization scheme. However, there is no scenario in which there is a single violation in tile size, and a double penalization of tile edge offset.

Since it was deemed important to make the penalization meaningful for selecting between locally different solutions, we chose to penalize disparity between key-based and physical tile edges. Furthermore, penalization of tile size did not seem to be advantageous for *nonlocally* differing chromosomes.

Remapping Random Keys Versus Initializing Population with Constrained Key Values

For a tile-bound DPU, remapping the random key from [0.0,1.0] to the tile's subrange has the same effect as initializing the population in such a way all tile-bound DPUs have random keys falling within their proper subranges. Having the remapping as an explicit decoding step allows the genes to have uniform representation, which simplifies diversity measurement and mutation. An explicit remapping also simplifies the enabling/disabling of this remapping.

6.2.2.2 Repair of Tile Binding Violations

The above method of aligning tile boundaries does not eliminate all tile binding violations. Remaining tile bindings violations are also subject to non-Lamarckian repair i.e. for the purpose of evaluating the chromosome.

Where Do the Violations Come From?

Tile binding violations may occur because of disparity between key-based and physical tile edges (Figure 6.29). This is because the keys for tile bound DPUs are mapped to the value ranges for their home tiles. These ranges are defined by the key-based tile edges, not the physical edges; the physical edges do not have specific key values associated with them. Since tile bound DPUs have their keys remapped to value ranges defined by the key-based tile edges, the disparity between key-based and physical edges create the possibility of these DPUs ending up in physical tiles that are not their home tiles. Therefore, the solutions are not feasible.

In practice, the tile edge disparities typically reduce to an offset of a single DPU position very quickly in the search, but the repair algorithm must be prepared for any amount of disparity. This is especially true at the beginning of the search when the gene values are totally random. In the worst case, the algorithm must be prepared for the possibility that the key-based tile edges have crossed into tiles beyond just the neighbouring tiles.

Beyond the initial stages of the search, it is more likely that no tile bound DPUs will be outside of their home tiles. This is because key-based and physical tile edges will tend not be misaligned, due to penalization. Therefore, the tile bound DPUs must have keys at the extremes of the value range in order to be positioned between key-based and physical tile edges, thus violating its tile binding. Note that this can only happen if the misalignment is such that the key-based tile edge lies *outside* of the home tile, as defined by the physical tile edge. For example, in Figure 6.29, a DPU bound to the left tile cannot violate its tile binding because its remapped key value is in the range $[0, 1/3]$; this cannot put it outside of the left most tile, as defined by the physical tile edges.

Why Repair Tile Binding Violations

Repair of tile binding violations is particularly beneficial because it raises certain points at the edges of infeasible regions to feasibility. This avoids the effect of driving solutions away from the edge of feasibility, which is a popular place when resources are tight. We expect that this deleterious repulsion effect is exacerbated by the sloppiness of position specification by the keys. The sloppiness means that the edge of feasibility is not well defined in terms of key values, thus allowing spill over effects from the penalization of infeasibility into the regions of feasibility.

Repair Heuristic

The heuristic to repair tile binding violations can be summarized as follows. The tile binding violator is the swapper in this case, and we seek a swapee within the swapper's home tile. We scan for such a swapee starting from the nearest DPU in the home tile (Figure 6.30(a,b,c)). The following conditions are to be met by a swapee.

- ◆ If the swapee is tile bound, it should not be brought out of its home tile by the swap.
- ◆ If the swapee is already out of its home tile, the swap should not bring it further away. Such a solution would be more grossly infeasible, since the swap brings it a full tile further away.

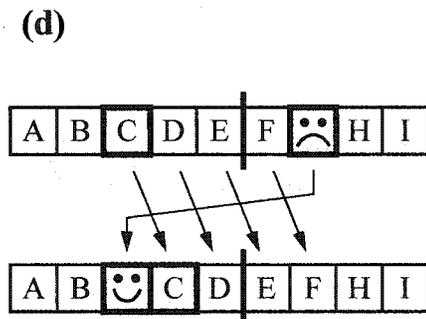
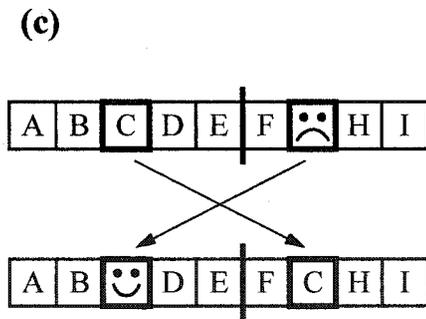
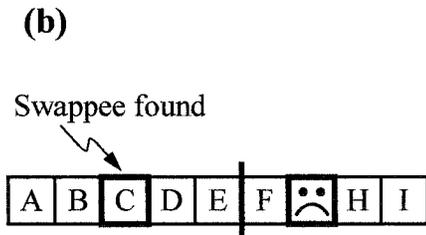
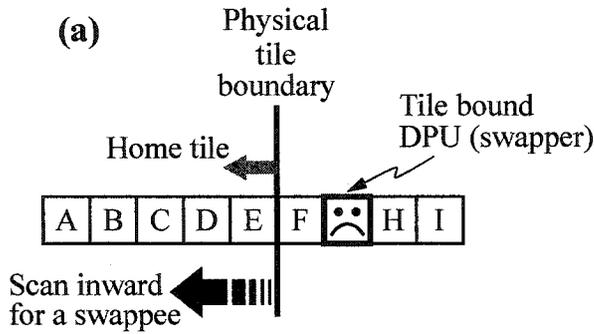


Figure 6.30. Repair heuristic for tile binding violation. (a) The “offending” DPU (swapper) is located outside of its home time. A swappee is sought by scanning inward in the home tile, starting from the closest edge (b). (c) Swapper and swappee can be swapped. (d) The swapper can also be shoved into the position of the swappee by pushing all intermediate DPUs toward the original position of the swapper.

Swapping Versus Budding and Shoving

Swapping was not the first fix that was considered. An appealing first alternative was to shove the swapper into the position of the swappee, “budding in line” so to speak (Figure 6.30(d)). The swappee, and all DPUs between it and the empty slot left behind by the swapper, get shifted toward the empty slot. The intuitive appeal behind this approach is that the ordering is not changed as much, in the spirit of minimizing the perturbation due to repair. In particular, the swappee undergoes a 1-slot shift rather than a disruptive swap. This is a potentially bigger swap than the adjacent swap for polarity repair, even though any tile-binding violator will most likely be within one slot of its home tile. This is because it may be necessary to scan a few DPUs into the home tile before finding a suitable swappee.

There are a few reasons for preferring the swap rather than having the swapper shove its way into the home tile. A minor reason is that it is computationally cheaper to swap chromosomes rather than shift a series of chromosomes. A better reason is that it avoids changing any other DPUs aside from the swappee. If shoving was used instead, the nearest DPU in the home tile would get pushed outside of the home tile. This potentially causes tile-binding violations; it may also cause polarity violations in any of the shifted DPUs. Swapping avoids this by testing the for violations induced only in a candidate swappee by the swap i.e. there are potentially a lot fewer tests.

Not Penalizing Tile Binding Violations

The final questions in resolving tile binding violations was whether or not to penalize violations that needed repair, and if so, whether or not to let the penalty stand even if it was repairable. The reason for penalizing repaired violations was to discourage such chromosomes that need repair, since repair can deceive the search. However, this directly contradicts the goal of not discouraging searches in the valuable transition region between feasibility and infeasibility. Therefore, penalties were not imposed on tile binding violators that exist prior to repair.

It was also decided not to penalize unrepairable genes because that would amount to a double penalty, thus discouraging the search from the transition region. The double penalization arises from the fact that tile binding violators can only happen if key-based tile edges drifted from physical tile edges. This disparity was already being penalized.

6.2.2.3 “Polarity”: Even/Odd Row Constraints

Certain DPUs must be bound to even or odd rows because the capabilities of hardware are not fully duplicated in all rows. In the model platform, this refers to the ability to read/write the local memory. We will refer to evenness/oddness as *polarity*. The ordering that results from the algorithm discussed thus far does nothing to ensure or encourage conformance to polarity constraints. That is, incorrect polarity is neither penalized to guide the evolution, nor is it repaired for evaluation.

The most obvious fix for polarity violations is a local variation on the ordering to attempt to get a solution that is free of such violations. This is much better done in the repair for evaluation, rather than hoping that the GA will find such a perturbation through proper penalization. Neither crossover nor mutation is particularly good for local search. The mutation can be made local by deliberately limiting the allowed perturbation, but that has restrictive implications for the GA search in general.

Fragility of Polarity Correctness

Another reason why polarity violations are best handled as a repair is because polarity correctness of polarity-constrained DPUs is fragile, and the local search must be conducted in a deterministic fashion to preserve it. For example, if many of the DPUs had a polarity constraint and a chromosome happened to conform to all of them, this compliance can be shattered by inserting a single DPU into a low key value position, thus shifting all DPUs by 1 slot. (Similarly, DPUs that are noncompliant with polarity constraints can easily be made compliant by fortuitous random events). The dependence of compliance on more than 1 DPU can be understood graphically by considering a 2-DPU with DPU D_1 and D_2 having random keys K_1 and K_2 . The search space is simply a 2D plane. The boundary at which D_1 and D_2 change positions is $K_1=K_2$, which is not aligned to any axis. Hence, the boundary can be crossed by a change in either one of K_1 or K_2 . A similar picture applies to N -space; the finely interleaved regions of compliance and noncompliance have boundaries that are at an angle to the axes, making it not only risky to change the random key for the polarity-bound DPU, but also for other DPUs.

Because of the fragility of polarity compliance, it is prudent to *expect* that many chromosomes will not be compliant, particularly if more than one DPU has a polarity constraint. Therefore, a repair mapping is required as a routine part of evaluation i.e. in the same way that a mobility gene is automatically mapped to correct operator times in HLS. (The fragility of polarity can be viewed

as interdigitated regions of feasibility and infeasibility in the design space, which explains why a stochastic search faces great risks when trying to find a violation-free local point).

Repair Procedure

The simplest repair, and most local perturbation, is to swap the violating DPU with an adjacent neighbour. We will refer to the violating DPU as the *swapper*, and the DPU with which it swaps positions as the *swappee*. There are a number of details that should be kept in mind when choosing a swappee. Most are motivated by the hope of not creating worse violations.

- ◆ In the absence of other considerations, choose one of the two adjacent neighbours with the closest key value to minimize the perturbation.
- ◆ Both the key and the position must be swapped to maintain monotonicity between key values and DPU positions. Recall that the DPU keys determine the position of the key-based tile edges.
- ◆ The swappee should not be made to violate any polarity constraints of its own by the swap. In the best case, the swappee is also in violation of its own polarity constraint, in which case the swap will fix violations in both swapper and swappee.
- ◆ The swap should not cause the swapper or the swappee to violate any tile bindings that they don't already violate. If the swap facilitates compliance to any tile bindings, so much the better.
- ◆ Physical tile edges will be used to determine compliance with tile bindings, since key-based tile edges are fictitious.
- ◆ If the swapper/swappee is outside of its home tile, we must decide whether it is important for the swapper/swappee to not be positioned even farther by the swap.

Tracking and Penalizing Polarity Violations

If neither neighbours are able to meet all the conditions of a swappee, the polarity violation is left unfixed and penalized a token cost of 1. One reason for this cost is because DPU is only one position removed from the correct polarity. Another reason is that the GA can't make good use of cost for local search anyway, so it initially acted more like a flag of an infeasibility rather than evolutionary pressure.

Since zero cost was no longer synonymous with zero violations, it was necessary to track the polarity violations explicitly. In that case, the only remaining role of the unit cost for a polarity violation was its questionable role as evolutionary pressure. It was left in because it was deemed more likely to help than to hurt. Furthermore, if genetic diversity degenerates as the population converges, the mutations and matings may very well yield kids that are slightly different rather than drastically different from the source chromosomes. Such changes are more likely to cause isolated changes in polarity, thus giving hope that evolution can fix isolated polarity violations without creating more polarity violations in other DPUs.

6.2.2.4 Global Vertical Wires

We address the *simplified* problem of incorporating the vertical global wires (Section) into the placement of DPUs. We do not address the potential retiming problems at this point. The availability of vertical global wires can be accounted for by simply making allowances for some number of wire violations. This is a minor modification of the cost calculation.

To use the vertical wires to maximum benefit, the longest wire length violators are chosen to be “pardoned” from overlength penalization. This assumes that the shorter overlength wires are more likely to resolve themselves as the search progresses.

The model platform allowed 3 global wires to be driven from within each tile. However, 3 made the automated placement too easy without having first incorporated all the constraints. The nominal number of global wires was set at 2 to challenge the search algorithm and test the efficiency of how the constraints were incorporated into the algorithm.

6.2.2.5 Placement of Multipliers

The multipliers are configurable elements that are in separate class from DPUs. That is, DPUs cannot perform multiplication, and multipliers do not have the same capabilities of DPUs. Two multipliers are located at the bottom row of each tile. This positional constraint is easily accommodated in the RKR scheme by sorting the multipliers separately from the DPUs. The simplicity of this approach was not available for enforcing DPU tile bindings because the tile bound DPUs must still be interspersed with DPUs that are not tile bound.

This approach is applicable to multiple classes of DPUs. DPUs are sorted by key for each class k separately. All DPUs of all classes must show up in one chromosome, however, to allow building blocks to form across classes.

In the case of PMX applied to vanilla encoding, the genes for each class should be a permutation of integers 1 to N_{class} , where N_{class} is the number of array elements in the class. This may require PMX repair, which must be separately applied within each class.

For both vanilla encoding and RKR, dummy DPUs may be needed to ensure a full set of DPUs within each class. Dummy DPUs were used in our test case to ensure 27 DPUs for the example platform (including two multipliers at the bottom of each tile).

Due to its simplicity, it may or may not be incorporated into the proof of concept.

6.2.2.6 Scheduling Repairs/Penalization of Violations

The repair and penalization procedures discussed can be broken into 3 categories of decoding and evaluation: wire length, tile binding, and polarity (the use of vertical global wires can be viewed as a magic repair procedure for the wire length category). They can be applied in any order, with varying degrees of difficulty. For example, if the wire violations were assessed first, we can make use of this information in deciding whether to swap DPUs in repairing other violations e.g. decide on possible fixes by choosing one with least wire cost, or smallest wire violation. The wiring cost would then have to be corrected for each fix. This interdependence between the 3 categories can lead to enormous complexity.

It should be kept in mind that repair is intended to be a local search, a minor and possibly ad-hoc fix to a point in the design space in hopes of bettering it. Otherwise, the heuristic repair is doing the search that the GA should be doing, thus defeating the reasons for using GAs. Two of the main reasons are (1) natural simplicity in encoding problem parameters, and (2) global search. The repairs were restricted in scope so as minimize obscuring the relationship between genes and final solution parameters e.g. via too many compounded and complex contingencies during the repairs. The repair were also restricted to variations in the underlying chromosome so as to complement GA's global search with simple local searches.

We limit the complexity of the overall decoding process by treating each of the 3 categories of decoding as largely decoupled and sequential phases. That is, the considerations in determining if and how to repair a violation in one stage will not try to anticipate how the penalization in the next stage is affected.

The order of the 3 phases is driven by simplicity (Figure 6.31). Since wire length assessment does not induce any changes in ordering due to repairs, it should be the last phase. Of the remaining two phases, it makes sense to put polarity repairs after tile binding repairs because changes in

DPU position for polarity repairs are local changes, or *fine-tuning* changes. By scheduling it after the repairs for tile binding violations, we can justify ignoring the creation of polarity violations in resolving tile binding violations. As described in Section 6.2.2.3, however, the polarity repairs do respect the tile binding constraints. This is the limit of the coupling between phases. Therefore, a swap that resolves a tile-binding violation may cause great penalties in wire length evaluation phase, but we rely on feedback from the GA search to minimize the existence of such chromosomes.

Within the repair phases for tile-binding and polarity, several passes of the phase are possible. This is because each repair changes the DPU ordering and opens up the possibility that previously unrepairable violations become repairable. The repetition of each phase can be stopped if no repairs were made in the last pass.

There is also the possibility of going back to the tile binding phase after the repetitions in the polarity phase have ended. This outer loop can also be stopped when no repairs were possible in the last iteration. This outer loop was not implemented. If it was, the implementation should keep a copy of the best solution of all outer iterations. This is because there is no guarantee that successive iterations of the outer loop will generate better results, since tile binding repairs may create polarity violations that did not exist in previous iterations.

6.3 Overall GA

The overall GA is shown in Figure 6.32.

6.4 Summary/Conclusions

The foundation for the GA model was first established in Matlab based on the reduce ordering problem, which only considered local wires. Sophistication was limited by the need for vectorizability and/or acceleratability. C++/STL was more suitable for the full placement problem. Its greater speed allows for a more complete investigation of the regions of good/bad convergence. An automatable tool flow transition was described to transfer the design information from Verilog to a simplified netlist for the GA placement program.

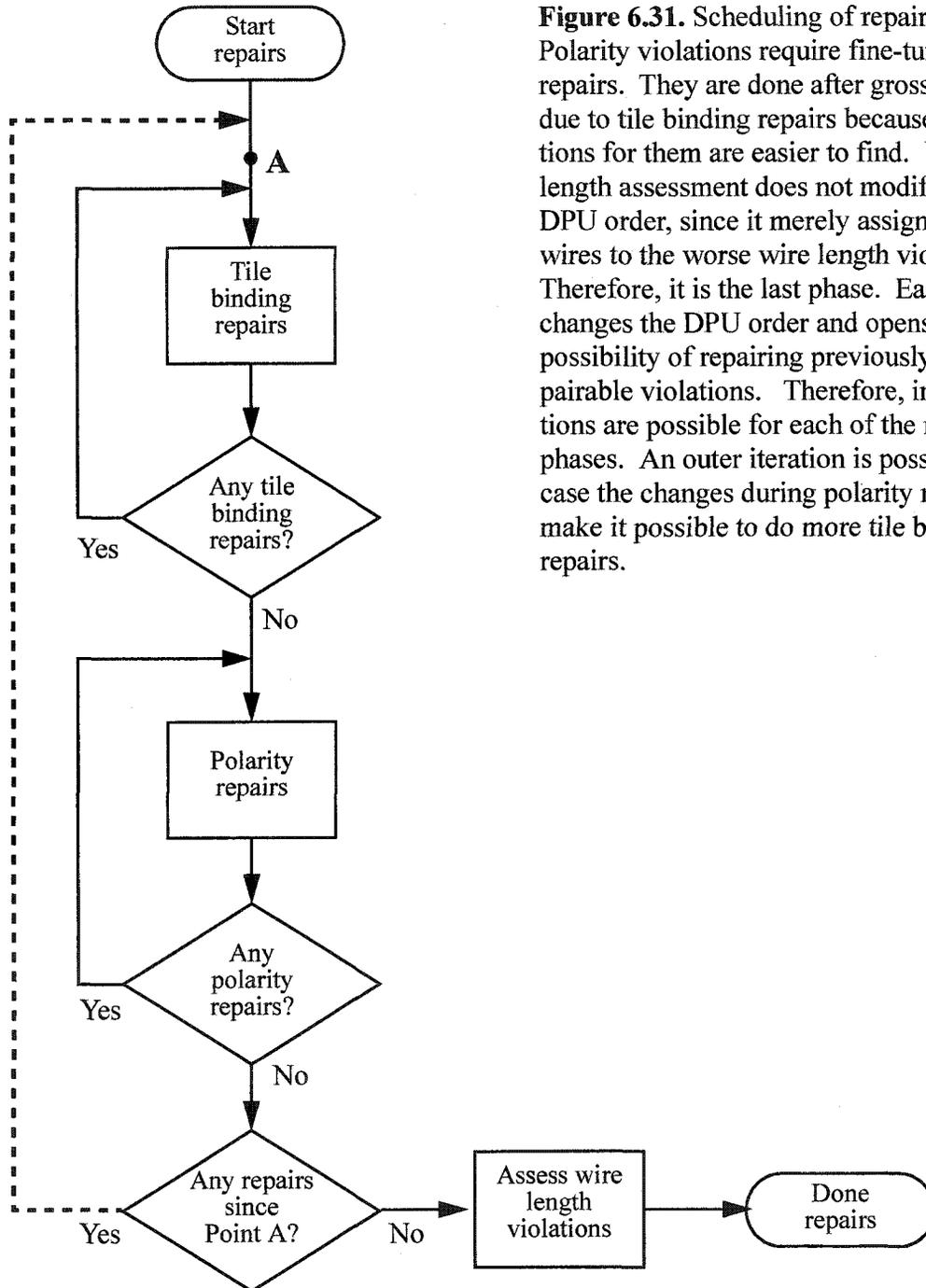


Figure 6.31. Scheduling of repairs.

Polarity violations require fine-tuning repairs. They are done after gross changes due to tile binding repairs because solutions for them are easier to find. Wire length assessment does not modify the DPU order, since it merely assigns global wires to the worse wire length violators. Therefore, it is the last phase. Each repair changes the DPU order and opens up the possibility of repairing previously unreparable violations. Therefore, inner iterations are possible for each of the repair phases. An outer iteration is possible, in case the changes during polarity repairs make it possible to do more tile binding repairs.

1. Initialize random population and rank genomes by cost.
2. Loop
3. Generate offspring (select parents and mate them) and evaluate them using the procedure of Figure 6.31.
4. Generate mutants and evaluate (Figure 6.31).
5. Merge kids (offspring and mutants) and sort by cost.
6. Remove twins among kids.
7. Remove clones between kids and current population.
8. Do a sorted merge between kids and population, then throw away the worst to maintain population size.
9. Stop when a valid solution is found or if the maximum search time is

Figure 6.32. Final genetic algorithm, as an elaboration of Figure 2.5. Whenever a genome is generated (e.g. offspring, mutant), the cost is automatically assessed using the constraint handling and cost determination of Figure 6.31.

Gene Encoding and Crossover Method

PMX crossover was tried and passed over in favour of RKR because of the disruption of building blocks outside the crossover region during PMX repair. RKR avoided the gross migration of an offspring chromosome from an infeasible region (where an offspring is not a permutation of the DPUs) to a feasible region. Within RKR, uniform crossover was favoured over 2-point crossover. This was because the building blocks for a DPU netlist are distributed rather than compact.

Penalization and Non-Lamarckian Repair of Constraint Violations

Violations of constraints such as local wire length and tile/polarity bindings were dealt with using a combination of penalization and heuristic local search. Local wire length violations were best dealt with using a penalty that was commensurate with the degree of violation. There was no advantage to penalizing underlength wires to promote clustering of connected DPUs.

For tile/polarity violations, the result of a local search for a compliant perturbation was applied as a non-Lamarckian repair during chromosome decoding. The feasible points in the search space

are thus replicated and replace some of the infeasible points (the repairable ones). Non-Lamarckian search is less likely to prevent the finding of a feasible solution.

Ranked Fitness

A linear cost-fitness conversion was initially sought in order to avoid the asymptotic explosion of fitness as cost approaches zero. Such nonlinearity makes the search greedy and more likely to prematurely converge on a very suboptimal local minimum (in cost). Sigma truncation was passed over to avoid exclusion of population members from selection, which effectively shrinks the population size and diversity. Instead, the “reference” cost for zero fitness was taken as the population’s highest cost. However, this was found to detrimentally reduce selective pressure in large populations.

Ranked fitness was the most suitable way to maintain controlled selective pressure, both over the entire population, regardless of cost distribution, as well as across generations. Selective pressure can be controlled through an arbitrary mapping between cost ranking and fitness. The selective pressure is more predictable than a direct cost-fitness mapping. Ranked fitness also simplified roulette wheel selection.

Cost Degeneracy

Artifacts from cost degeneracy and cost ranking were discussed. The ranking of kids were skewed upward by this. Systematic bias in selection was eliminated by shuffling chromosomes with the same cost prior to selection.

RKR’s Improvement over PMX, and the Necessity of Mutation

The initial implementation of the reduced ordering problem used PMX with mutation of offspring. The bottom heavy distribution of convergence times suggests a strategy of multiple, short time-limited searches. Caveats to this conclusion were pointed out based on efficiency limitations in the Matlab implementation.

Better performance was obtained from RKR with 2-point and uniform crossovers, and mutation of general population members. The key enabling element was mutation, since no new gene values are created by the disruption of PMX repair. Moreover, nonbinary quantities were not assembled from multiple binary genes, so new nonbinary values could not be created by the disruption of crossover.

Overriding Influence of Mutation

Mutation of general population members yielded much better performance than mutating offspring. It was found to be such a determinant of successful search that crossover was unnecessary. In that case, there was no parental selection. Instead, selective pressure resulted from the lingering of fit chromosomes as less fit chromosomes are unconditionally displaced by mutants.

The overwhelming advantage of mutating general population members versus crossover is not as surprising as it first seems. Early incarnations of *evolution strategies* used only mutation; its practice today still relies primarily on mutation, with multiple “parents” used only to find a centroid in the genotype space from which take a random step. One may be tempted to conclude that building blocks play a limited role if crossover is of such limited usefulness. However, it can be argued that mutations preserve building blocks much more than crossover. This is because mutation changes only one gene. It is recognized in [FG96] that, in the presence of mutation, insensitivity of the search to crossover is not an uncommon observation. The role of crossover is still being debated e.g. [Spe93].

Genotype Complexity Enabled by Mutation Without Crossover

A distinct advantage of mutation without crossover is that complex genotype representations can be devised without worrying about how to mate dissimilar structures. This allows the genotype to much more closely match the phenotype e.g. in the case of a graph representation for a netlist of DPUs. An entire branch of evolutionary computation known *evolutionary programming* [FF96, Whi02] has developed based on similarity between genotypes and complex phenotypes, made possible by circumventing the issue of crossover.

Optimality of Non-Background Levels of Mutation

Based on the reduced ordering problem, greater mutation rates appeared to be better when offspring rather than general population members were being mutated. When general population members were mutated, lower mutation rates were better. This is because there are many more mutants for the same mutation rate, compared to mutation of offspring. Good performance was obtained when the mutants numbered in the order of half the population (20%-80%).

Minimization of Tile Violations

Tile bindings of DPUs were enforced by mapping their random keys to fixed, equal-sized sub-ranges for their home tiles. Imbalances in the number of DPUs per tile were penalized in the same

manner as overlength wires. Local heuristic search was conducted to fix tile violations arising from unbalanced tile sizes.

Polarity Repair as a Fragile Fine Tuning

Polarity bindings were viewed as positional fine tuning. Compliance to polarity constraints are fragile because of the finely interleaved regions of compliance and noncompliance, the boundaries of which are not aligned with the search space axes. For this reason, we relied completely on deterministic local search to find polarity compliant variations of a chromosome without breaking any existing polarity compliances. This polarity resolution phase was scheduled after tile binding resolution to simplify the latter.

Incorporation of Vertical Global Wires

Vertical global wires were utilized by assigning them to the worst violators of local wire lengths. This was the last phase in constraint resolution because it changes neither the phenotype nor any penalties from the tile/polarity constraints.

Iterative Repairs of Violations

In each of the phases for resolving tile and polarity bindings, all violations were rescanned for further repairable violations after a repair is successfully made. An outer loop is also possible so that the tile binding phase is revisited after the phases for polarity resolution and global vertical wires. However, tile violation repairs do not respect polarity bindings, so tile violations might be further reduced at the expense of more polarity violations. Similarly, wire length violations can change unpredictably due to the additional repairs in each iteration of the outer loop. Getting the best solution requires an approach similar to that used in min-cut. In min-cut, a number of perturbations are incrementally applied to a solution, and only leading subset of perturbations that minimize cost are retained. Similarly, for DPU placement, we can keep only the sequence of outer loop iterations that gives the lowest cost.

Encouraging Search Near Feasibility Limits for Tough Problems

The resolution of constraint violations was designed to encourage search close to the regions of infeasibility. This is because problems that push resource limits are expected to have solutions that are close to infeasible regions. Search along the border between feasible and infeasible regions is encouraged by avoiding straight penalization in enforcing tile bindings. The border is also made

fuzzy by the use of random keys, as well as a soft definition of tile size imbalances to be penalized. Finally, the use of non-Lamarckian fixes to tile/polarity violations ensures that a chromosome is not pushed out of regions of infeasibility.

Intuitive Diversity Metric and a General Diversity Scaling

Diversity measurement is subjective, and can defy intuition in some cases. A gene diversity was devised based on the variance of differences between adjacent gene values. Three reference points are introduced to scale the metric. Based on these, three intuitively motivated normalizations gave rise to three measurements for genome diversity. For the purpose of comparison, this scaling was also applied to a traditional variance metric based on all-pairs differences, for which there is a linear-time calculation method.

7. Experimental Results and Discussion

This chapter presents a thorough characterization of the GA for the full placement problem and the diversity metric developed in Chapter 6.

7.1 Algorithm Performance

The metrics to characterize the GA of Chapter 6 were first developed on the FFT, then applied to a suite of kernels.

7.1.1 GA Characterization Metrics Developed on the FFT

Why Seek A Better Metric than Generations Count

In Section 6.2.1.2, it was observed that the generations counts for crossover with mutation of offspring (X+MOO) were more than an order of magnitude higher than schemes that mutate general population members (X+MGPM and O_MGPM). However, the run times for X+MOO were only several times higher than the MGPM schemes. This is because the mutation rate for X+MOO was the probability of each parent spawning a mutant. Hence, no more than two mutants can be generated per steady-state generation. This limits the amount of searching that is possible in a generation, hence the large generations counts. That this is a likely cause for the large generations count is evident in Figure 6.26; as mutation rate for X+MOO increases from zero, the generations count hardly drops at all, and seem stuck near the value at which X+MOO meets the MGPM schemes at low mutation rates. Hence, all of the X+MOO performance correspond to the MGPM schemes at a point where the latter have low generational kid count.

Total Kids Versus Run Time

A metric that could possibly reveal trends missed in “generations count” curves is the total number of kids processed before finding a solution. Even this is just a rough indication of computational effort, since it doesn't reflect the fact that more generations incur more intergenerational

overhead, even if the total kid count is the same. Recall that intergenerational overhead is made up in part by ranking, elimination of duplicates, diversity measurement, etc. The greater intergenerational overhead accounts for some of the several-fold difference in run times for X+MOO over the MGPM schemes. Run time is the only practical metric, regardless of how sensitive it is to workstation loading and coding style. In this chapter, all these metrics will be used.

Deceptively Ineffective Crossover Under Steady-State Evolution

The single-mating generations of steady-state evolution may also be why crossover seems to make little difference among the MGPM schemes. When general population members are mutated, the mutation rate represents the portion of the population that will spawn mutants. Hence, the generational mutant count can approach the population size. In contrast, the generational offspring count is 2, which is miniscule compared to the population size. Since most of the search is done by generating mutants, it is not surprising that the presence or absence of offspring has little effect.

Two Widely Encompassing Parameters to Sweep

Compared to simply varying the mutation rate under steady-state evolution, a better way to explore the benefits of offspring and mutants is to vary the generational kid count *and* the fraction of kids to be made up of offspring rather than mutants. We refer to the ratio of generational kid count to population size as *population fraction kids* (PFK) and the fraction of kids to be made up of offspring as *kids fraction offspring* (KFO). These two parameters are not all-encompassing, but they are widely encompassing. When PFK and KFO approach 1 (i.e. 100%), the scheme approaches “generational evolution” for GAs. When $PFK \rightarrow 0^+$ (the minimum number of nonzero offspring i.e. 2 offspring), the scheme approaches “steady-state evolution”. When $KFO \rightarrow 1$, the scheme resembles “evolutionary strategies” in the sense that evolution is dominated by mutation (though the exact manner in which mutants are generated is different in the case of classical evolutionary strategies).

7.1.1.1 GA Performance for the FFT With All Constraint Enforcement Mechanisms Enabled

Figure 7.1 shows the characterization of the GA for the FFT kernel with all placement constraints, penalties, encoding transformations, and penalties discussed in Section 6.2.2. The run

times are somewhat insensitive over much of the parameter ranges, despite increased generation counts for low PFK. The KFOs with fewest peaks in run time are $KFO=0.3\sim 0.5$. Of the KFOs with peaky run times, $PFK=0.5$ seems to generate the least dramatic peaks. High KFOs are counter-indicated in Figure 7.1(c) due to degraded success rate, particularly for low PFKs. From Figure 7.1(d), $KFO=0.3$ also seems to generate the fewest total kids over the entire search, which roughly indicates computation effort. It should be kept in mind, however, that intergenerational overhead exacts a greater run time penalty for low PFKs.

Sweet Spots for GA Search

For the FFT, $KFO=0.3$ and $PFK=0.5$ seems to be a regime of good convergence, with performance being robust over a significant range around that. A lower PFK risks degrading the probability of successful search. A higher PFK causes negligible improvement in success rate, but potentially increases generation counts. Even for KFO's for which generations counts have plateaued (the $PFK>0.5$) regime, run time increases with PFK simply because of the greater number of kids per generation (and thus overall).

Greedy Search is Bad for Tough Problems

The degradation of success rate for high KFO combined with low PFK can be intuitively understood. This combination yields a greedy search. High KFO means more exploitation by generating offspring rather than diversification by generating mutants. Low PFK leans toward a steady-state replacement strategy, which again is greedy because most of the population remains undisplaced between generations. For tough problems such as the FFT, an overly greedy search can impede escape from local optima. The FFT turns out to be the toughest kernel that is examined, using all but 1 DPU in the entire slice.

In the characterizations of Figure 7.1, mutates are selected from the general population with uniform probability. Appendix D.1.2.1 explores the selection of mutates based on fitness. This did not seem to affect the search.

7.1.1.2 Testing the Individual Constraint Mechanisms

The effectiveness of the various mechanisms to enforce constraints was investigated. Aside from local and global wire length limits, these mechanisms are devoted either to tile constraints or polarity constraints. Each mechanism can be independently enabled or disabled via a software switch.

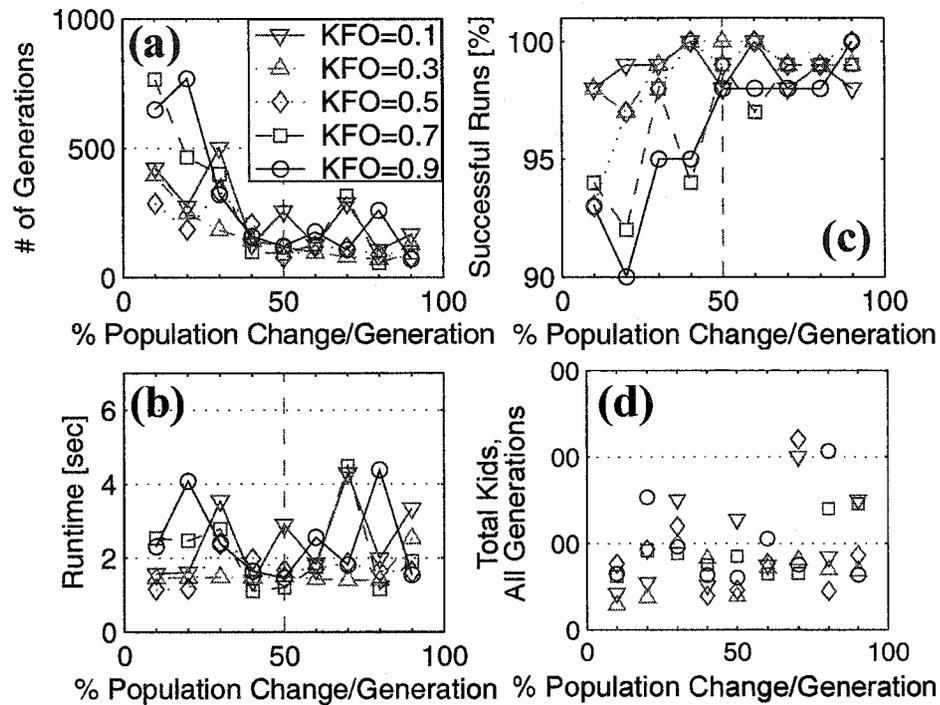


Figure 7.1. GA performance for various PFK (shown here as “Population Change/Generation”) and KFO.

The total kid count (d) has normalized to the population size $N_{Pop}=200$.

There are 3 tile constraint mechanisms and 2 polarity constraint mechanisms. The tile mechanisms are:

- ◆ Remapping the random key from the range [0.0,1.0] to a tile’s subrange
- ◆ Penalizing remaining tile boundary offsets
- ◆ Repairing remaining tile violations by locally searching for a modification of the DPU arrangement

The polarity mechanisms are:

- ◆ Repair polarity violations by locally searching for a modification of the DPU arrangement
- ◆ Marginally penalize any unrepairable polarity violations

The testing of these separate mechanisms consists of turning each one off while all others remained on. The alternative of turning each one on while all others are disabled would take too long to investigate, due to strong likelihood of poor convergence. In addition to disabling each mechanism in isolation, all tile binding mechanisms were disabled while polarity binding mechanisms were enabled, and vice-versa.

Due to the large volume of data, the results for these selective disabling of constraint mechanisms are contained in Appendix D.1.2.2 on page 283 rather than attached immediate here.

Tile Binding Mechanisms

Figures D.12 and D.13 show a dramatic degradation in success rate when all tile binding mechanisms are disabled. Surprisingly, for the successful runs, the generations counts do not differ much from enabling of all constraint mechanisms (Figure D.11). It seems that a simple GA with no tile binding mechanisms has little hope of finding a solution if a “low hanging fruit” is not found immediately. The failed searches do not contribute to the generations count data, so the data for generations count does not show any degradation.

Figure D.14 shows that the repair of tile binding violations is critical to the performance of the GA. In its absence, both success rates and generations counts are significantly degraded compared to the case where all constraint mechanisms are enabled (Figure D.11). This is in line with the findings in the GA community showing strong advantages in combining local search with GAs. Compared to the case of no tile binding mechanisms (Figures D.12 and D.13), however, the higher generations counts and success rates show that the “high hanging fruit” are being found by the remaining tile mechanisms (key remapping and penalization of tile edge misalignments) after a more protracted search.

Despite their contribution, Figures D.15, D.16, and D.17 show that key remapping and penalization of tile edge misalignments can be disabled individually or simultaneously without significantly degrading the search. In the latter case, compliance with tile bindings is attributable entirely to local search and repair of tile violations. This implies that any benefits from key remapping and penalization of tile edge misalignments can be obtained from local search and repair alone.

Polarity Binding Mechanisms

Figure D.18 shows that disabling all polarity binding mechanisms is about as harmful to the search as disabling all tile binding mechanisms (Figure D.14). Figure D.19 shows that this degradation remains largely unchanged when penalization is enabled, but local search and repair is kept

disabled. Therefore, penalization of polarity violations alone is not very useful; a local search and repair is required. As mentioned in Section 6.2.2.3, this is to be expected due to the fragility of compliance with polarity bindings. Since the GA cannot use such fine spatial information, it is conceivable that the marginal penalization of polarity violations can do more harm than good by the introduction of spatial noise in the cost function. In fact, Figure D.19 seems to show a slightly worse success rate than Figure D.18. The slightly lower generations count hints that this is due to a lesser ability to find fruit beyond those that are “low-hanging”.

7.1.2 Other Kernels

7.1.2.1 The Case Against Random Kernels

Because of the novelty of the Chameleon platform, benchmark netlists are not readily available. The generation of random netlists was considered. By itself, it is conceptually trivial. However, the generation of satisfactory and realistic circuits for exercising EDA algorithms is quite involved and somewhat contentious e.g. [Alp98,HGRC96,HRC97,IH94,IHKS97,VSC02,SVC00,PLM00]. The algorithms for doing this target large netlists of fine grain elements e.g. gates or FPGA LUTs, and are nontrivial. Based on intuitively defined global metrics for large netlists, the algorithm parameters are calibrated to generate artificial circuits with the same characteristics as standard benchmarks. This requires a large enough netlist for these global parameters to be meaningful in an average sense.

Coarse grain circuits are already at a high level of hierarchical abstraction, however, and the netlists are relatively small, with a high degree of nonrandom structure. The coarse grain elements also have many more terminals than elementary circuit elements. Whether or not the existing algorithms can be adapted to generate random netlists of coarse grain elements is a topic that is beyond the scope of thesis. If it is possible, however, the algorithm parameters would still need to be calibrated by studying actual netlists, the absence of which is the problem. Heterogeneous constraints would also need to be generated. For these reasons, real kernels are needed.

7.1.2.2 Datapath Design of Real Kernels

The decision was made to generate a suite of real kernels. There are four groups of kernels: (1) variations of the FFT; (2) variations of a FIR filter; (3) variations of an IIR filter, and (4) variations of the IIR filter that interleaves 4 data streams. The FIR and IIR filters have standard configura-

tions that can be found in most DSP texts [JGP96]. We briefly summarize the distinguishing features below.

Since a full design of a kernel can be quite involved, only the datapath was designed for the FIR and IIR filters. The control path was assumed to generate the required signals at the right times. Some high-level conceptual information for internal kernel organization was obtained from Chameleon documentation.

FFT Kernels

One variation of the FFT was to make the kernel harder by adding extra tile and polarity binding constraints. Another was to remove all tile and polarity constraints. Including the unmodified FFT kernel, this represents 3 kernels, referred to as “FFT”, “FFT Easy”, and “FFT Hard”.

FIR Filters

Two realizations of a 9-tap symmetric finite impulse response (FIR) filter were mapped to the array. The first is the folded direct-form structure, referred to as “FIR DF-I”. This form has a multi-input summation, which was implemented as an adder tree. An adder tree can contribute complexity to the placement (and implied routing).

The second realization is known as the transposed structure, referred to as “FIR Xpose”. It consists of a cascade of adders separated by algorithmic delays, thus forming a natural pipeline. Not only is it easy to lay out, but it fits the coarse grain array very well, since the algorithmic delays can align with the DPU output registers at each adder’s output. However, the input signal must be broadcast to all adders.

Simple IIR Filters

Two realizations of the single-data-stream infinite impulse response (IIR) filters were mapped. The two mapped filters are different realizations of a circuit commonly known as a “biquad”. These circuits do not have large fan-out or fan-in nodes like the FIR filters above (the multi-input adder can be regarded as a large fan-in node; its realization as an adder tree merely changes the form of the topological complexity). The first biquad realization is the direct-form II structure, referred to as “IIR DF-II”. The second realization used a direct-form II transposed structure, referred to as “IIR Xpose”.

IIR Filters for Interleaved Data Streams

The IIR filters have feedback loops with latency. The physical delays around a loop are often greater than the algorithmic delays, so it is necessary to slow transform the original problem so that the algorithmic delays match the physical delays. This is known as n -slow timing [LS91], and requires that $n-1$ zero-valued samples be inserted between every valid data sample to make up for the extra $n-1$ algorithmic delays appended to each original algorithmic delay. Rather than waste $n-1$ out of n computations, those empty data slots can be populated with data from $n-1$ independent data streams, effectively interleaving these data streams in such a way that the same filtering is being performed on all streams.

The IIR filter required 4-slow timing. This amortizes the scarce multipliers over 4 data streams, but exacts a cost in terms of kernel complexity. The kernel netlist is bigger because extra DPUs are needed for the additional delays, when such delays cannot be absorbed into already recruited DPUs with available registers. Furthermore, the local memories must be organized into 4 data banks, each with its own DPUs for reading and writing. Not only does this push the DPU count up, it boosts the number of DPUs with gross and fine positional constraints. The gross constraints come from the read/write DPUs having to be adjacent to the memories for their data banks, while the fine constraints come from the interleaving of DPUs capable of reading and writing memory. As well, the gross positional constraints are much tighter, since each read/write DPU is bound to the quarter of the slice occupied by its data bank. Finally, the interleaved streams must enter and exit the biquad at a common point, leading to high fan-in and fan-out nodes DPUs. Such DPUs must be close to 4 different read/write DPUs simultaneously, making it harder to find a satisfactory placement.

Each of the two simple IIR filters “IIR DF-II” and “IIR Xpose” were implemented as a 4-slow version, referred to as “IIR DF-II x4” and “IIR Xpose x4”. To have the GA recognize the 4 data banks as 4 placement regions of the slice, we defined a tile as one quarter of the slice. Strictly speaking, a tile is determined by the PLA (Figure 3.10). However, Section 6.2.2.1 uses the concept of tiles to constrain DPU placement. This simplification has no consequence, since the control path design phase can use whatever definition it wants. Our GA only uses “tiles” to constrain placement of the memory read/write DPUs.

Kernel Sizes and Complexities

The resources required for each kernel are shown in Table 7.1.

Table 7.1: Kernels tested.*MULs:* Multipliers. *DF-I:* Direct form I.*DF-II:* Direct form II. *Xpose:* Transposed DF-II.*x4:* 4 interleaved streams. The number of memory read/write DPUs is indicated in brackets.

Easy



Hard

Kernel	DPUs	MULs	DPUs+ MULs	Nets	Driven Ports	Tiles
IIR Xpose	7(2)	5	12	11	15	1
IIR DF-II	8(2)	5	13	12	15	1
IIR DF-II x4	14(8)	5	19	15	21	4
FIR Xpose	10(2)	5	15	14	22	1
IIR Xpose x4	13(8)	5	18	14	27	4
FIR DF-I	19(2)	5	24	23	31	1
FFT Easy	20(2)	6	26	24	50	3
FFT	20(2)	6	26	24	50	3
FFT Hard	20(2)	6	26	24	50	3

7.1.2.3 GA Performance Across Kernels

As in the FFT, the performance of the search was found to be fairly robust across most of the KFO and PFK range. Success rates for the FIR/IIR kernels were all 100%. This means that a solution was found to meet all constraints before the termination condition i.e. when there have been no improvement in the minimum number of violations for 35,000 generations. Thereafter, the search was assumed to be stuck. This termination condition was empirically determined to allow the capture of most of the data points in the convergence time distributions. Its adequacy is evident from the distribution tails in Figure 7.2.

Since there is far too much data to present for all the test cases, and due to the robustness of convergence with respect to PFK and KFO, these distributions were obtained by aggregating the results from an even spread of KFO and PFK values: {0.1, 0.3, ..., 0.9} and {0.1, 0.2, ..., 0.9} respectively. 100 searches were performed for each KFO/PFK combination, thus yielding 4,500 searches per curve. For each search, the population consisted of 200 genomes. Because of the ubiquitous 100% success for the nonFFT kernels, the caveat against high-KFO/low-PFK does not apply. On the contrary, the searches tend to be slightly faster in this regime. It seems that for less difficult problems, the search space is more densely populated with feasible solutions, and a greedy search is beneficial.

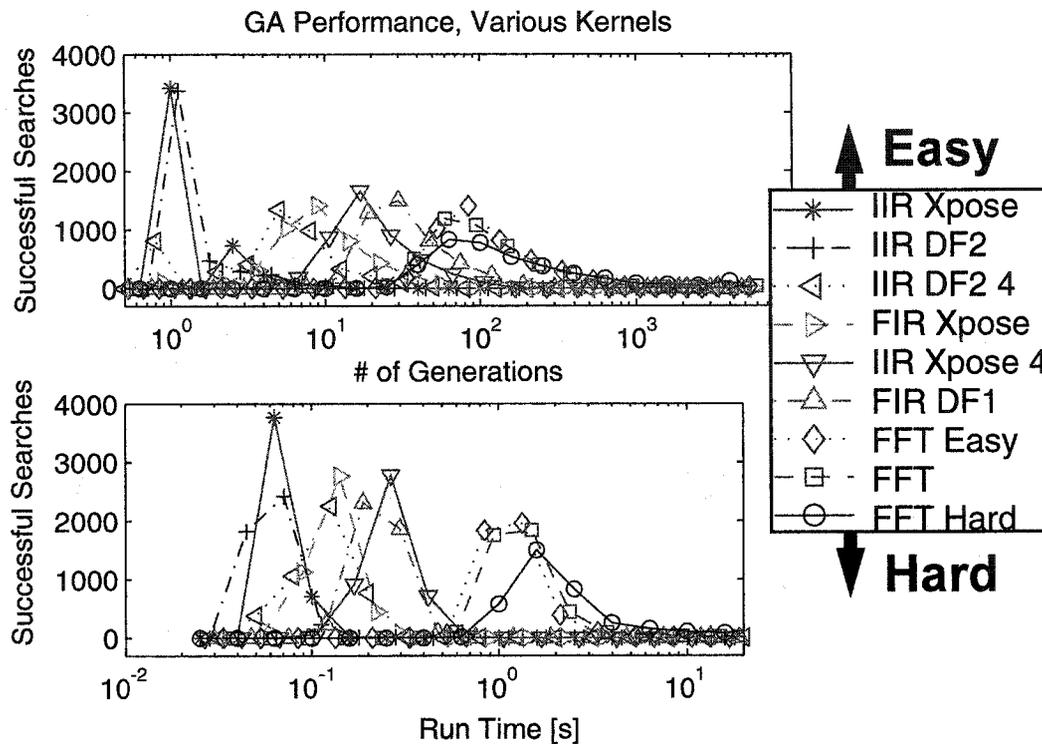


Figure 7.2. Distribution of convergence times for placement for different kernels.

Histogram data was generated in the log- x domain.

Host: Sun Blade 100 running Solaris 8.

Figure 7.2 shows that the search times are mostly well below the 1-second regime, with the hardest searches typically taking 1~2 seconds. Rarely does a search take 10 seconds or more. For more modern computing platforms, search times will be much shorter. This is well within the realm for an interactive CAD environment. It compares favourably with manual placement, which can take from hours to days. Recall that a feasible solution is rarely found by the proprietary vendor tools for the example platform adopted by this study; hence, most cases reduce to manual placement.

Search Times Correlate with Circuit Complexity

Both in Figure 7.2 and Table 7.1 kernels are listed in order of difficulty of placement, as determined by their convergence time distributions. Note the strong correlation between placement difficulty and the intuitive measures of circuit complexity in Table 7.1. For all the FFT variants, the

convergence distribution doesn't seem to change shape, but shrinks with difficulty due to failed searches.

Why Some Kernels are Hard to Place

The difficulty rankings in Table 7.1 make intuitive sense. In rows 1-2, the easiest kernels are the simple IIR kernels, which have no high fan-in or fan-out nodes, and use very few resources. FIR Xpose is harder to place than the simple IIRs because it represents a folded 9-tap filter rather than a 5-tap biquad. More algorithmic delays and adders yield a more complex circuit. The “x4” IIRs in row 4-6 are more complicated than their noninterleaved counterparts because there are 4 times as many memory read/write DPUs. Furthermore, these DPUs are constrained to the tiles for their associated data sets. FIR DF-I is the most complicated of the non-FFT kernels (and in particular, FIR Xpose) because this structure uses an adder tree, which is known to be hard to organize physically. The adder tree also requires its own DPUs; in contrast, the adders in a DF-II structure are in-line with the delay line, and can share DPUs with algorithmic delays. The FFT and its variants are the most complex.

7.2 Calibration of Diversity Metric

7.2.1 Comparison of Metrics

The diversity metrics developed in Section 6.1.5 were applied to the FFT kernel with $PFK=0.5=KFO$. Figure 7.3(a) compares D^{Simple} , $D^{GoodNorm}$, $D^{Linterp}$, and Div^{Norm} , as well as showing other population metrics to show the correlation with these metrics. They all indicate a convergence at approximately 100 generations. As expected, D^{Simple} and $D^{GoodNorm}$ decrease quickly and flatten out quickly. As indicators of diversity, they lose much of their resolution in the initial stages of the search. Notice that not even log scale adequately compensates for this nonlinearity. (Appendix D.2 also discusses reasons to avoid logarithmic scaling when measuring diversity.) Div^{Norm} progresses down steadily and flattens out abruptly. $D^{Linterp}$ has the opposite problem of D^{Simple} and $D^{GoodNorm}$; that is, very little of $D^{Linterp}$'s headroom is used by the diversity's range of interest, which indicates that the linear $N^2 P_{AC} \rightarrow D_k$ mapping was an overcompensation for the nonlinearity in D^{Simple} and $D^{GoodNorm}$.

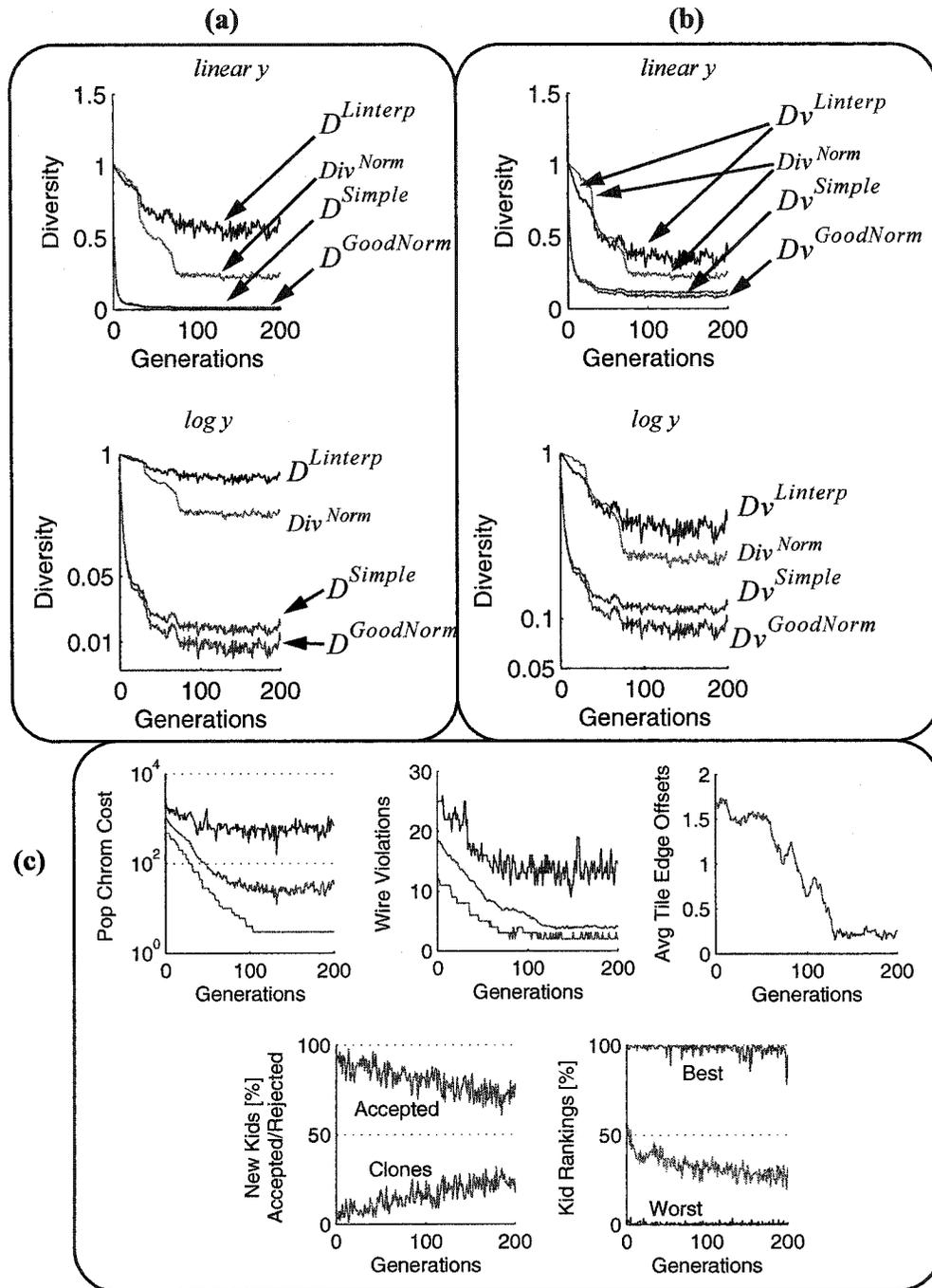


Figure 7.3. Sample FFT placement to demonstrate diversity measures, $KFO=0.5=PFK$.

(a) Comparison of diversity measures using D_k versus Div^{Norm} . (b) Comparison of diversity measures using Dv_k versus Div^{Norm} . The diversity measures are calculated *after* each generation. It was confirmed that all diversity measures were approximately 1 *prior* to the first generation. The change in diversity can be compared with the change in other population metrics (c) to gauge its meaningfulness as an indicator of convergence.

7.2.2 Adjustment of Normalization

In order to make the metric more usable, the various D_k must undergo a nonlinear transformation so that there is steady progressively decreasing D through most of its value range. This mapping should be continuous and monotonic, and should not change the diversity value at the reference points i.e. it must maintain $D_k = \{1.0, 0.0\}$ at $N^2 P_{AC} = \{N^2 P_{AC}^{(+)}, N^2 P_{AC}^{(0)}\}$. The power functions $y = x^P$ are ideal candidates, since they contribute a controllable amount of nonlinearity within $[0.0, 1.0]$. The choice of P allows either compression at high values of x and “gain” at low values, or vice-versa. Figures 6.21 and 7.3 indicate D^{Simple} and $D^{GoodNorm}$ need to be compressed at high values, while $D^{Linterp}$ needs to be expanded at high values. Figure 7.3(b) shows the result of the following power function transformations.

$$\begin{aligned} Dv_k^{Simple} &= (D_k^{Simple})^{1/2} \\ Dv_k^{GoodNorm} &= (D_k^{GoodNorm})^{1/2} \\ Dv_k^{Linterp} &= (D_k^{Linterp})^2 \end{aligned}$$

As with D , the global metrics are $Dv \equiv \text{mean}_k Dv_k$. With this power transform, $Dv^{Linterp}$ can be made to emulate Div^{Norm} much more closely (assuming that is desired). Dv^{Simple} and $Dv^{GoodNorm}$ are still far from ideal for monitoring diversity, though log scale seems to improve them (ignoring for the moment that use of log scale itself is not desirable). Due to the extensive correction required, we conclude that D_k^{Simple} and $D_k^{GoodNorm}$ are not particularly good scalings of P_{AC} for the purpose of measuring overall diversity. This might have been anticipated by the need for log scaling in Figure 6.20.

7.3 Summary and Conclusions

Favourable Performance of Search

The GA reduces placement time to seconds (and less), compared to the hours and days of a manual effort. Success rates were 100% for all but the hardest kernels, and nearly 100% for the latter. The stochastic nature of the search allows failed searches to be easily circumvented by launching another search to get different solution.

Sweet Spots for GA Search

Two parameters that allow the GA to assume a wide variety of evolutionary styles are fractional population change per generation (PFK) and the fraction of the kids that made up of offspring rather than mutants (KFO). The GA was demonstrated on a suite of kernels over a range of PFK, from steady-state to generational, and over a range of KFO. As measured by generations counts, the difficulty in finding a solution correlates strongly with various intuitive indicators of circuit complexity. The effects of KFO and PFK were studied for the most complex kernel, the FFT. Normally, mutation is considered a low-level background operator (high KFO), and population is either steady-state or generational (extreme values for PFK). We found that KFO=0.3~0.5 and PFK~0.5 is a robust combination, with significant leeway in both parameters around this regime. The GA was not significantly affected by selection scheme for mutates.

Effectiveness of Constraint Enforcing Mechanisms

We examined the different mechanisms for enforcing tile and polarity constraints. With no tile mechanisms, the search has a dramatic drop in success rate. Key remapping and penalization of tile edge misalignments do improve the success rate, though searches took 2~3 times as many generations as with all constraint mechanisms enabled. Of the 3 tile mechanisms, the most critical one is repair of constraint violations i.e. local search. It's use yields search behaviour that is similar to the enabling of all 3 mechanisms i.e. when violations are repaired, the presence or absence of key remapping and penalizing tiled edge misalignments did not matter much.

Of the 2 polarity mechanisms, penalizing violations did not make much difference. This is attributable to the fragility of polarity compliance, and the GA's inherent focus on global search rather than local search, as well as its stochastic nature. As expected, compliance relied primarily on deterministic local search. Its absence causes as much degradation as disabling all tile mechanisms.

Local Search Complements GA's Global Search

From the dramatic effect of repairing tile and polarity violations, it appears that local search complements the GA's global search very well.

Promising Behaviour from Characterized Diversity Metric

The 3 scalings for our diversity metric were calibrated based on comparison with a conventional variance metric. The diversity measure that varies linearly with the variance of gene value

differences (dubbed $D_V^{Linterp}$) was able to provide similar information as the traditional diversity metric, with a certain degree of nonlinear adjustment. Both metrics correlate with other indicators of population convergence. Our metric appears to be reflect the smoothness of other indicators, as well as their later plateauing, compared to the conventional variance-based metric. It remains to use this metric in a feedback loop for adaptive PFK and KFO.

8. Conclusions

The placement problem described in Section 1.6 was generalized to be easily adapted to other architectures of this type. Many architectural features of the example architecture are encompassed by the generalized problem. An easily extensible hybrid of GA and local search heuristics was devised to deal with the heterogeneous constraints that arise from both the array architecture and the kernel under design. In contrast to the low success rate of the vendor tools for the example platform, this algorithm found feasible solutions for kernels of varying complexities. It did so in trivial time, compared to the duration that could be expected of a manual effort.

In this chapter, we present summarizing points from this research, followed by conclusions, contributions, and future work.

8.1 Discussion

Where Coarse Grain Arrays Have a Role

Coarse grain reconfigurable arrays fill a need that is not optimally met by FPGAs, processors, or ASICs. This need is for high-speed word-oriented signal processing of stream data. The characteristic of this vague application domain is repetitive and regular computations on word oriented data, with little control flow that depends on data. The aim of coarse grain arrays is to sacrifice generality in operation and tailor themselves to such needs, thus allowing optimizations in speed, chip area, and power. This requires that the array resources be more predefined prior to fabrication. Conversely, to ensure alignment with computational needs, such arrays need to target more specific applications domains, where the hardware requirements have been characterized over a suite of applications.

Sensitivity of Mapping Methodologies to Architecture

Mature design paradigms and EDA tools for coarse grain arrays are severely lacking. Compilation and physical design algorithms are highly dependent on the target platform architecture and interconnect network. The computational model of some platforms assumes a regular looping kernel, with operations in the loop body that can be time-space mapped to the processing elements of

a pipelined array. Other platforms are meant for a more FPGA-like design flow, where the array elements can be used in a more arbitrary manner.

Generalizable Features of the Chameleon Platform

This thesis focuses on the physical design phase, targeting a Chameleon-like architecture. The Chameleon architecture contains many features that can be generalized for coarse grain arrays. For example, it separates resources for datapath and control path, caches DPU personalities, caches an entire plane of kernel configuration bits, spatially partitions the array to limit complexity of interconnect and control PLA, uses distributed PLAs for control (as opposed to centralized control), contains mostly local wires and limited global wires, primarily orientates its interconnect along one array axes, and contains distributed local memories to alleviate external memory bandwidth. As well, greater functionality is packed into the array by heterogeneously distributing some operations among the array elements. This imposes additional complexity on the physical design phase to find a proper resource binding for a DPU netlist.

Tractability of Simultaneous Place-and-Route for Coarse Grain Arrays

Place-and-route for ASICs and FPGAs is normally a 2-step process, with routability being estimated in the placement phase using empirical formulae. Coarse grain arrays, however, present components at a higher level of hierarchical abstraction. Not only are many bit-slices aggregated into words, but the functionality of the array elements and interconnect network are more completely defined. This reduction in the search space allows for determination of routability during placement, thus combining the two steps.

8.2 Conclusions

Advantages of GAs for Heterogeneous Constraints of Coarse Grain Arrays

The caveat for the more predefined functionality of coarse-grain arrays is that the heterogeneous resources lead to more heterogeneous constraints on component binding e.g. multipliers versus DPUs, predefined memory segments at fixed locations, and interleaved DPUs of differing capabilities. The GA approach is preferred over optimum-finding approaches because the problem is one of constraint satisfaction rather than optimization i.e. the search space is sprinkled with valid solutions, each of which is as good as the other. Furthermore, search space methods allow

for natural problem encoding through the genotype design; this is flexible enough for constraint enforcement to be incorporated into the problem encoding, via penalization, and by augmented local search. More importantly, the flexibility in problem encoding allows quick adaption of a physical binding algorithm to the changing constraints as the platform architecture evolves. Finally, the stochastic nature of the GA provides robustness in the face of the pass/fail characteristic of solutions.

GA Design for Placement (and Routing) for Coarse Grain Array

The following GA design was found to be suitable for the placement (and routing) for a coarse grain array. The developed GA used:

- ◆ ranked fitness to avoid scaling issues,
- ◆ random keys as sloppy position indicators to avoid a host of problems with permutation chromosomes, and to develop enforcement mechanisms for gross positional constraints; and
- ◆ uniform crossover to avoid favouring compact building blocks.

Furthermore, hybridization of the GA with local search/repair contributed greatly to the performance of the GA.

8.3 Contributions

Formulation of the placement problem into a formal combinatorial problem

Based on investigation of a novel design flow, the placement problem was formalized into a form that was suitable for solving using combinatorial methods.

Constructed Multi-Faceted GA Approach to Handle Heterogeneous Constraints

For the physical resource binding phase, a genetic algorithm was developed to handle the heterogeneous constraints in the physical design phase.

- ◆ In the problem encoding, random key remapping was used
- ◆ Penalization was used not only in the phenotype space, as is typical, but also in the genotype space by penalizing tile edge misalignments to head off situations that give rise to tile binding violations.

- ◆ Deterministic local search heuristics were used to repair violations of gross and fine positional constraints. These local searches were complementary to the GA's stochastic global search, and were found to be the most effective of mechanisms to enforce constraints.

This multifaceted approach to gross and fine positional constraints illustrates that the flexibility of a GA can tackle the seemingly arbitrary heterogeneous constraints in coarse grain arrays. This is particularly significant in the face of changing array architectures.

In the GA development, an efficient implementation of ranked fitness selection was devised. Artifacts arising from cost compression were discovered, and workarounds were devised.

Compared to the hours and days for manual placement, the GA took from fractions of seconds to seconds, making it suitable for a CAD environment. Failed searches were minimal; due to the stochastic nature of GAs, they are not of much consequence in the context of multiple searches.

Characterization of the Algorithm for Course Grained Placement

The algorithm above was extensively tested over variations of the genetic algorithm. Certain characteristics were noted:

- ◆ The performance was fairly robust over significant variations between generational and steady state population change, but was best with partial generational changes.
- ◆ The performance was fairly robust for mixes of mutation and crossover, but was best where two are equal. This departs from the conventional notion of mutation as a background operator.

These evaluations were done on a suit of kernels ranging from very difficult to quite easy (Table 7.1). The subjective estimate of the problem difficulty was in good agreement with the run time of the program.

Developed Intuitive Diversity Metric and Calibration Procedure

An intuitively based diversity metric was developed using *adjacent* differences in gene values. The new metric could be scaled to give information similar to the classical metric. It also gave a smoother decrease.

To provide a common framework for diversity metrics in general, three intuitive diversity scenarios were identified as calibration points. Any useful measure of gene diversity should be normalizable to have the prescribed behaviour at these points.

For the "random scatter" scenario, normalizations are introduced for the classical variance-based diversity metric and the new metric. The classical diversity metric simply used the variance of a uniform PDF; for the metric based on adjacent differences, the data was viewed as a Poisson process, and its normalization used the variance in interevent spacing.

Proof of Viability of New Architecture

An early contribution resulted from the use of an unproven design flow targeting a new coarse grain reconfigurable platform to design the nontrivial FFT kernel. The knowledge gained was that of proving the real-world viability of an untried computational model and architecture. For new platforms in general, the mapping of the FFT is often regarded as a serious proof of concept, due to its complexity and the diversity of control and datapath logic required.

Established Efficient Design Practices for Coarse Grain Arrays

The design case gave rise to several further contributions:

- ◆ The establishment of efficient datapath design practices, such as DPU packing strategies.
- ◆ Examples of opportunistically piggybacking of control path functionality onto already recruited DPUs.
- ◆ The application of retiming practices are demonstrated in the context of array elements, where even the block memories can be regarded as graph vertices (with lookup table functionality).

Without established design practices, there is no starting point for development of EDA algorithms.

Efficient Control Path State Machines

For the control path, PLA-efficient state machines were devised that are suitable for overlapping the filling of a pipeline with its emptying.

Control Path Placement Procedure

A heuristic method was developed for reassigning state bits among PLAs, and was manually proven. This heuristic treats the vendor tool's boolean optimizer as a black box, in much the same way that conventional placement treats routing as a black box and merely estimates routability.

Performed Preliminary Analysis of Requirements for Evolutionary Synthesis

The automation of place-and-route was a focused thrust in a larger EDA effort, which included synthesis of kernel netlists from specifications of desired functionality. Applicable GA approaches were investigated for generating time-varying DPU netlists, specifically for stream data (Section 8.4.2 and Appendix E). An approach was established for future development of a suitable GA.

Publications

The work in this thesis has led to two conference publications [MKP02,MKP04] and one journal publication (accepted) [MKPss].

8.4 Future Work

8.4.1 Diversity Feedback

The diversity measurement developed in this thesis is merely a diagnostic tool unless it is incorporated as feedback into the GA. The population's diversity can be calculated every T generations, T being user-defined. Mutation or selective pressure can be modified to enhance diversity if the metric shows a low value. For the latter, we discussed how a constant offset added to the chromosome rankings results in a smaller relative difference in fitness throughout the population. As an example of incorporating feedback, the final GA of Figure 6.32 might be modified as in Figure 8.1.

8.4.2 A GA Approach for Synthesis

The larger context for the research in this thesis was to apply GAs to the entire design flow, including kernel synthesis from functional specifications. Appendix E contains an in-depth analysis of the requirements for an evolutionary approach to synthesis, including critical reviews of genotypes from which ideas may be borrowed for evolving a time-varying DPU netlist. The use of GAs for synthesis in coarse grain arrays is a quite a new research area with much room for the exploration of seminal ideas.

Of the evolutionary approaches presented, NEAT and EGG seem to offer the most promising starting points for a GA that evolves DPU netlists, with fully embedded configuration information.

1. Initialize random population and rank genomes by cost.
2. For $i_{Generation} = 1$ to ∞
3. For $j_{Generation} = 1$ to T
4. Generate offspring and mutants.
5. Remove twins and clones
6. Incorporate kids into population.
7. Measure diversity and push it into a FIFO of some user-defined length N
8. If the average diversities in the FIFO falls below some schedule $DiversityLowerBound(i_{Generation})$, increase mutation rate and/or decrease selective pressure.
9. If the average diversities in the FIFO rises above some schedule $DiversityUpperBound(i_{Generation})$, decrease mutation rate and/or increase selective pressure.
10. Stop when a valid solution is found or if the maximum search time is exceeded.

Figure 8.1. A possible way to feed diversity back to the GA. $DiversityUpperBound(i_{Generation})$ and $DiversityLowerBound(i_{Generation})$ are time-dependent sanity limits on the diversity of the search. How these curves vary with time can be driven by what may be considered sane limits on search time. They play a similar role to temperature in simulated annealing.

One way to avoid the thorny issues of crossover in EGG is to simply use mutation only (evolutionary programming) as a first step. The priority encoding concepts from NetKeys can be used to vary the number of DPUs in the netlist. A simple way to deal with the time varying interconnect of reconfiguration (a much more complex problem than the static graphs being evolved so far) is to have the DPUs fully interconnected (Figure 8.2) and let the time-varying DPU personalities determine whether the input muxes make use of each wire, cycle by cycle. Each DPU gene is an extremely complex gene, specifying up to 8 personalities, including personalities for steady-state operation, as well as for filling and emptying the pipeline. Since the number of DPUs is variable, one complication that will have to be resolved is how to interpret the case where an input mux is directed by its host DPU personality to select a net that is sourced by a missing DPU.

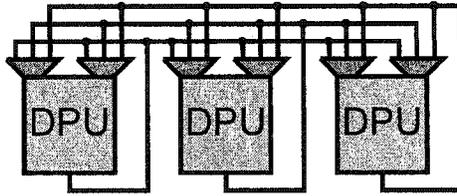


Figure 8.2. Fully interconnected DPUs. The embedded time-varying personalities determine which wires are used by which muxes in each cycle.

8.4.3 Other Approaches to Placement

Simulated annealing (SA) is currently being investigated for the placement problem. Preliminary investigations show similar efficacy to the GA approach when the same constraint mechanisms are built into the decoding and evaluation of the solution point (Figure 8.3). Such a comparison can be explored much more comprehensively e.g. by allowing KFO/PFK to vary according to a schedule, in the same manner that SA's temperature does.

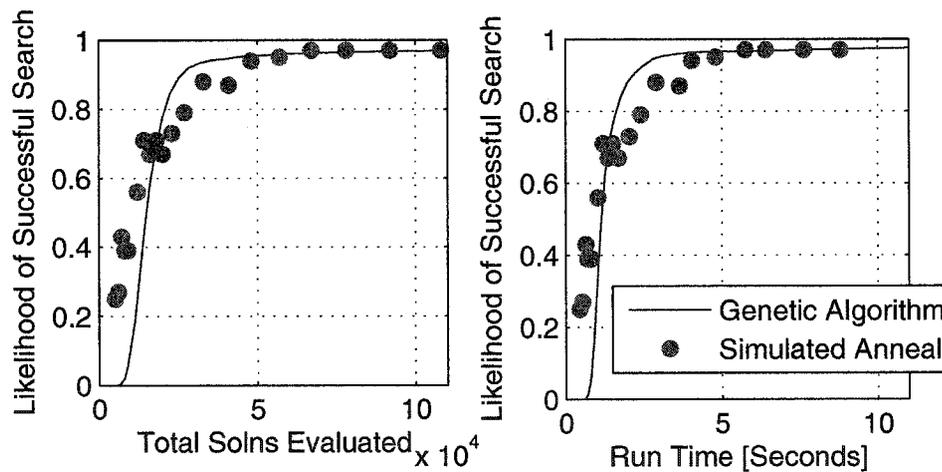


Figure 8.3. GA and simulated annealing perform similarly.

Casting the Chameleon placement problem as a boolean satisfiability problem (SAT) seems like a potentially good fit at first glance, since placement deals largely with constraint satisfaction. However, the independent variables are not boolean; only the compliance with the constraints are. The possibilities for applying SAT approaches bear investigation.

Within the evolutionary algorithm community, there are approaches for constraint problems that depart from the optimization model by coevolving sets of constraints along side the solutions. Examples of such “predator-prey” evolutionary schemes include [LKH02,Par94]. This coevolu-

tionary approach holds much potential for the place and route problem under study, since it is primarily constraints driven.

Finally, as mentioned in Section 2.1, integer linear programming (ILP) can be adapted to constraint problems. Systematic methods for doing this may eliminate the difficulty of casting arbitrary platform characteristics into an ILP problem. Despite ILP's origins as a true optimizer, it would therefore be worthwhile to investigate such methods, and the empirical tractability of the resulting optimization problem.

Appendices

Appendix A: Generic Design Flows

A.1 ASIC Flow

An ASIC flow consists of the following steps.

- ◆ Hardware design specification through hardware/software partitioning
- ◆ Design, HDL coding, and functional verification through simulation
- ◆ Area/speed/power-driven synthesis to target cell library, followed by simulated verification
- ◆ Floorplanning, cell placement, routing, including pad frame planning and routing/buffering to distribute power and clock. A device package must also be chosen.
- ◆ Delay extraction and annotation back to synthesized design, followed by delay-accurate verification through simulation. Depending on the difficulties in meeting timing, the results could require additional iterations by looping back to any of the prior steps, though reworking immediately preceding steps are preferable to reworking design flow from higher up.
- ◆ A series of layout related fine tuning and verification, including slotting metal lines, geometric design rule verification, and verification of consistency between layout and netlist.
- ◆ Submission for mask fabrication, device fabrication and packaging, and manufacturing tests (IDDQ and stuck-at faults, timing verification).
- ◆ Functional verification with packaged parts.
- ◆ Testing chip in place.

A.2 Coarse Grain Reconfigurable Array Flow

Some design flows targeting reconfigurable logic can overlap with ASIC design flows. This is particularly the case for targeting popular commercial FPGAs like Xilinx, especially if run-time

reconfigurability is not sought. In this case, a high level view of an **FPGA flow** is similar to the ASIC flow, minus everything related to layout, packaging, and manufacturing testing. The cycle of synthesis, place/route, and delay-accurate verification is quicker because of the limited possibilities for mapping, as well as the better characterized delay. Compared to the library cells of ASIC design, there may be more reliance on bigger library macros because of custom optimizations required by the predefined array, logic elements, and interconnect.

Design flows that target less generic platforms will have greater differences from the ASIC flow. If run-time reconfigurability is a goal, more front-end work is needed to determine how to break down a large application into kernels as opposed to software, based on the strengths of a reconfigurable platform's architecture, and how often to run each one. The following are typical steps that can be expected in a **design flow for reconfigurable logic**.

- ◆ Look for inherent structure in an algorithm to cast it in terms of a hardware/software partitioning that uses the characteristics of the reconfigurable platform to maximum advantage. The application is broken into subtasks, which serve as candidates for implementation as kernels. This phase can also take to account how often to run each kernel before changing kernels, based on I/O bandwidth, configuration loading bandwidth, depth of configuration caching, and local versus extra-array memory. It can decide whether to lean toward data-centric or kernel-centric scheduling, and apply loop transformations for such a decision. A major goal is to find a decomposition and a schedule that meets real-time requirements.
- ◆ An application is needed to capture the hardware design, if this isn't automatically generated by automated kernel identification e.g. as a high-level block diagram, schematic, or high-level software language, or high-level hardware description language (HDL).
- ◆ Synthesizing the captured design into resources on the reconfigurable platform. The kernel might first be translated into a generic form before technology-specific mapping e.g. template matching a dataflow graph to a library of macros for the array. Technology and library specific mapping also takes place in ASIC synthesis, but it can play a more prominent role for targeting prebuilt arrays because a macro needs to be optimized around the particular array architecture. It is expected to be less prominent in coarse grain arrays because the array elements are already aggregated into coarse blocks. Aside from the ready mapping of vectorizable loops to SIMD (Single Instruction, Multiple Data) arrays, synthesis for coarse grain arrays is an open research topic.

- ◆ Verification of hardware functionality through simulation.
- ◆ Functional verification through hardware/software co-simulation with the behavioural models of the components that make up the hardware kernels.
- ◆ Logic resource binding/packing and routing. This stage binds the resources required by the kernel to site-specific resources on the chip, including interconnection (wires, interconnecting switches, and possibly pipelining registers). This stage affects, and is affected by, the kernel design and synthesis in that component proximity may change the pipelining requirements, which changes the pipeline initiation and control logic for the pipeline, which in turn affects the design's compactness, component proximity, and efficiency of resource utilization. This is expected to be especially true in a coarse grain platform.
- ◆ Convert the placed/routed design, including control logic, into configuration bits. This should be fairly automatic.
- ◆ Software/hardware co-simulation with the detailed hardware model of the reconfigurable platform, including data and configuration download. This accounts for software details related to hardware/software integration e.g. scheduling activities such as issuing requests for configuration loading and execution, requests for DMAing data in and out, and associated system response times. For well-characterized devices with limited configurability, dedicated cycle-accurate simulation is far more efficient than general-purpose event-driven simulation such as that used for Verilog and VHDL. Therefore, unless there are asynchronous components on chip, hardware can be simulated using cycle-accurate simulation, assuming that synthesis and place-and-route avoids timing violations.
- ◆ Software/hardware verification with the actual hardware.

Two main factors determine how an algorithm or task is decomposed into kernels for execution on a reconfigurable platform. The top-down view of identifying the self-relating subtasks that make up the main task; that is, subtasks within which activities are so related that it would be messy to partition. Elementary tasks are clustered into subtasks in a way that minimizes communication between subtasks. These subtasks are candidates for implementation as kernels. The bottom-up view consists of familiarity with the platform architecture so as to know how the resources can be used to perform the subtasks. There are many factors to consider besides DPU availability and speed: how DPUs are arranged, datapath interconnect and delays, on-chip memory (its organization and interface with DPUs), control path logic and its interface with the datapath, I/O, and

reconfiguration caching. Using library macros for kernel building blocks can alleviate the difficulty of manually optimizing designs for a specific array architecture. Even for coarse grain arrays, macros can be developed for (say) address generators that scan data memory in specific patterns, circular buffers, FIRs, interleaved data streams to keep logic busy in the case of loops with inter-iteration dependencies (such as for IIRs), etc.

Together with the algorithm characteristics and real-time requirements, the top-down and bottom-up views determine whether subtasks must be further broken up to fit as kernels on a device, whether multiple subtasks are small enough to combine into a single kernel, or whether it is better to replicate smaller kernels to run in parallel. This scheduling activity varies widely depending on whether the task and subtasks are functionally specified in a serial fashion, such as with pseudocode, or parallel fashion, such as with dataflow diagrams. With pseudocode, the challenge is to identify dependencies between loop iterations and determine how early each iteration can begin; this is an exercise in “exposing” parallelism. With dataflow diagrams, the goal can be similar in principle, but opposite in practice. For example, the goal may be one of serialization in the sense of clustering nodal operations into kernels, to be executed in succession. As noted above, an opposite strategy may be to replicate kernels for parallel execution. Which is better depends on the on-chip memory and off-chip bandwidth.

Due to the greater maturity of fine grain FPGAs, and the prevalence of Xilinx in particular, there is ready information in the literature on which a design flow can be conceptualized for run-time reconfiguration (Figure C.1 on page 249). Such schemes have not much penetrated industry practice. Some of reasons are related to hardware. Xilinx devices are not designed specifically for run-time reconfigurability. This results from numerous factors: (1) the fine granularity implies a large number of configuration bits; (2) serial configuration loading compounds the lengthy configuration time; (3) the lack of vendor-endorsed standards and publicly released information on the configuration scheme (especially interconnect); and (4) logistical problems of ensuring proper functionality of hard macros as they are relocated across the device, in the face of highly nonuniform distribution of resources. The lack of vendor-endorsed and disclosed information and standardization also discourages third party development of commercial EDA tools for run-time reconfiguration, as well as commercial take-up of such tools. Some of the reluctance to standardize and divulge such information may stem from the desire to protect customer IP.

Appendix B: Details of Surveyed Platforms

B.1 Caching Kernels and Configuration Bits

Note that caching of configuration bits differs in principle from use of the reconfigurable fabric to cache hardware kernels, but they have similar effect in a typical kernel caching system where the host processor only fires one kernel at a time. In this case, a large device had seven different kernels cached would have similar capabilities and performance to a smaller device that fit only two kernels, but stored the configuration bits for the remaining five kernels. This presumes a low delay to switch between configuration bits, which is probably not a very stringent requirement considering that kernels are meant to accelerate repetitive functions over a long stream of data; thus, the delay in switching cached configurations is easily amortized over long periods of execution. The larger device, however, still has the advantage that it is capable of more parallelism. In fact, Garp just caches one kernel per device configuration, but caches a number of device configurations.

B.2 Interconnect and Scalability

Interconnect infrastructure (physical routing and programmable switch boxes) is a key feature for reconfigurable logic. The best choice of configurable hardware leggo blocks will have limited usefulness if the interconnect infrastructure overly constrains the manner in which they can be connected. But interconnect flexibility costs area as well as speed, the latter due to the capacitance of extensive routing and switches, and the distributed resistance of the switches. The capacitive loading also increases the energy per clock edge. Buffering or registering the interconnects convert bidirectional wires into unidirectional wires, requiring more wires to support the same level of communication. The design of the interconnect network cannot be relegated to a second priority without severely impacting the achievable utilization of device resources.

The challenge of interconnect goes beyond judiciously designing a system that minimizes the amount of overhead, yet maximizes the flexibility with which logic can be interconnected. For both ASIC and FPGA, the ratio of required interconnect resources to usable logic increases with the design size, which is reflected by the greater number of metal layers in recent fabrication technologies and by sophistication of tool assisted floorplanning and delineation of hierarchical blocks to avoid routing congestion.

The increasing proportion of interconnect to logic may be understood by considering a hypothetical hierarchical design. Assuming that a desired system level design is hierarchically decomposed for circuit design, B.1 illustrates the intuitive idea that as the system increases in size to encompass more basic library cells, more layers of aggregation used to integrate micro-functionality into larger physical blocks of greater hierarchical abstraction. Obviously, this takes more chip resources than just butting the cells up against each other; a layout that butts the cells together contains only intracell interconnect, whereas each block of aggregation requires additional interconnect to integrate the constituent blocks. Furthermore, the more complicated blocks at higher levels can be expected to have more input/output signals, possibly of wider width. Therefore, it may be more accurate to replace the pyramid structures of B.1 with rectangles of increasing width and height to reflect the fact that I/O ports and interconnect requirements do not thin out at the upper levels of hierarchy, towards the system level design. The end result is that one can expect interconnect requirements to scale superlinearly with design size (e.g. gate count), possibly scaling as $O(n^2)$, depending on the nature of the problem, the solution, and the circuit implementation chosen.

This superlinear scaling of interconnect with design size presents a challenge to the mapping of large systems onto a chip. Resolving it would reduce component count by fitting more onto a chip, as well as easing the design effort by avoiding much the off-chip bandwidth limitation in distributing a design over multiple devices. Since prefabricated reconfigurable logic platforms do not have the freedom to repartition device resources between interconnect and logic as required by the prevailing design, a first crack at circumventing the problem may be to identify classes of kernels that have significant overlap in interconnect architecture so that the platform can be properly aligned to those needs. However, this narrows the usability of the platform and it doesn't really solve the superlinear requirement of interconnect with design size. Moreover, the kernels required by an algorithm may not have similar interconnect requirements.

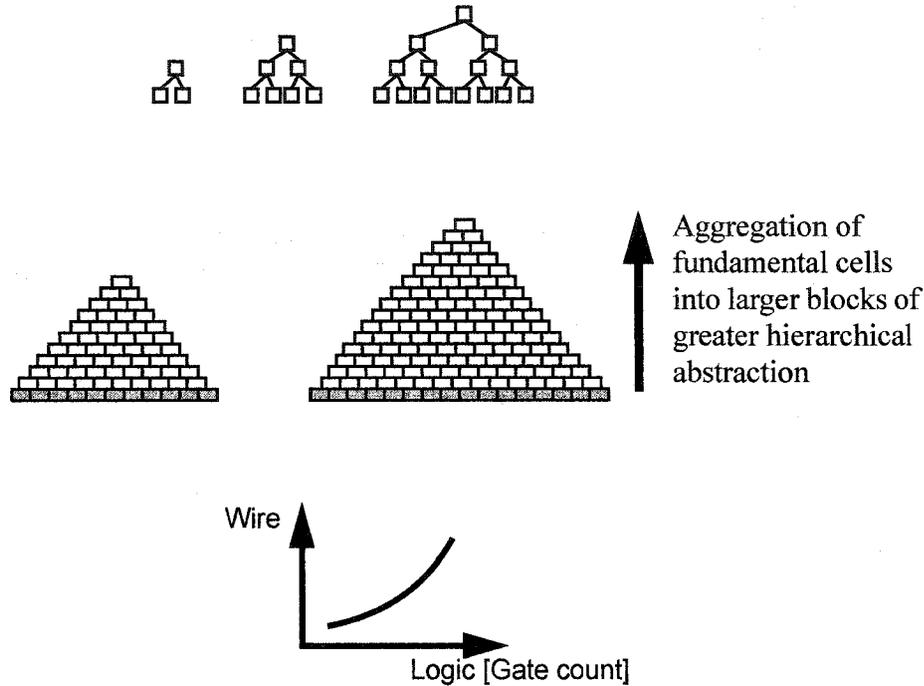


Figure B.1. Conceptual design hierarchy as design increases in size. Larger designs have more layers of aggregation and integration, thereby increasing ratio of interconnect to logic.

Since the required interconnect is mutually dependent on the algorithm and the implementation, one evolutionary path that could alleviate the superlinear need for interconnect is the development of new paradigms for algorithms and implementations that rely more on local communications, though not so stringent as to support only systolic-like (nearest-neighbour) messaging. This would simplify the task of designing the interconnect network for a reconfigurable platform. Furthermore, distributed local communications is generally associated with greater parallelism than a centrally shared interconnect scheme. [Concurrent point-to-point interconnections between arbitrarily located source and destination is not considered a solution because the $O(n^2)$ need for switches limits its feasibility to small domains; this quadratic dependence probably translates into a much more severe demand for the routing itself, both of which slow down signal propagation]. For a desired functionality, one can expect that algorithms and implementations using primarily local

communication would be favoured for their ease of mapping, even at the sacrifice of efficiency and performance, especially if the supporting platforms are readily available.

B.3 Dynamic Instruction Set Computer

Figure B.2a shows Brigham Young University's DISC project [WH95]. It consisted of an 8-bit CPU that was built from the logic cells of a commercial FPGA, National Semiconductor's fine grain, partially (re)configurable logic array (CLAy). Specialized opcodes were realized by loading kernels into the FPGA as required. The top 10 rows of the 56x56-cell FPGA hosted a controller (Figure B.2b) to sequence through an instruction cycle, while the rest of the array served as an opcode cache for the most recent kernels. Under the management of an external configuration controller, new kernels overwrote the oldest kernels as the controller executed its machine code (Figure B.3a). Complex kernels can take thousands of clock cycles and retrieve large amounts of data from memory. Thus the processor instruction set could grow as more kernels were developed. Kernels for common opcodes for flow control stayed resident.

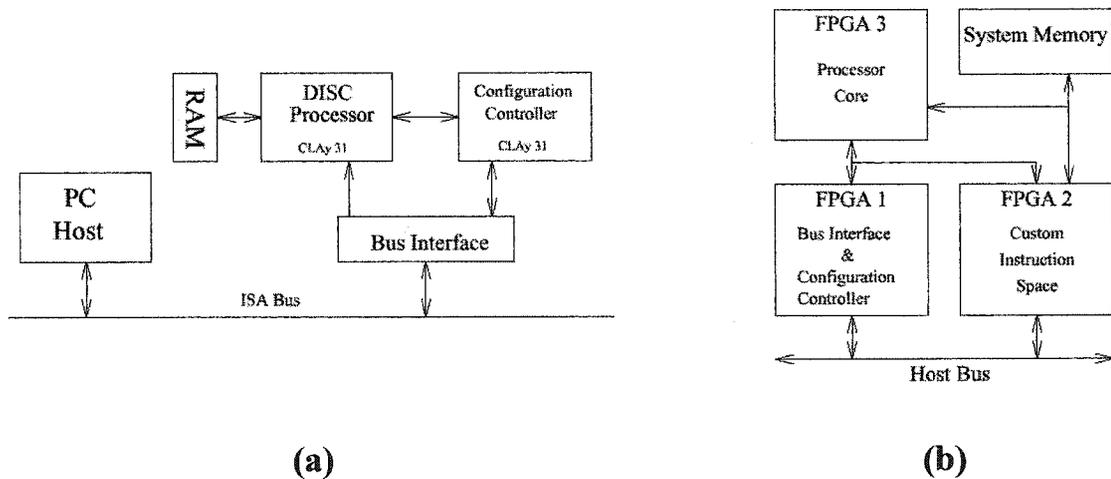


Figure B.3. DISC implementations.

(a) PC-ISA custom board with static bus interface for 8-bit DISC [WH95]. (b) National Semiconductor board for 16-bit DISC-II [WH96].

Since each loaded kernel displaces one or more kernels in the array, kernels designs had to be position independent, as well as easily interchangeable. This was simplified by imposing a coarse

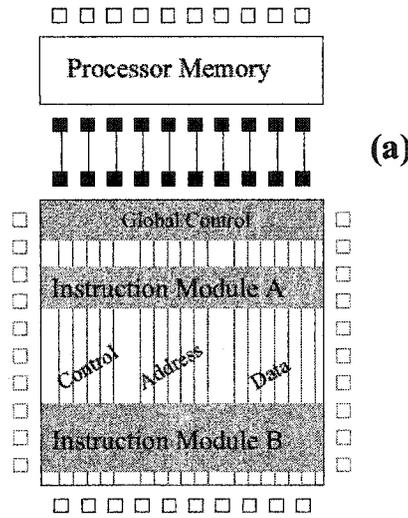
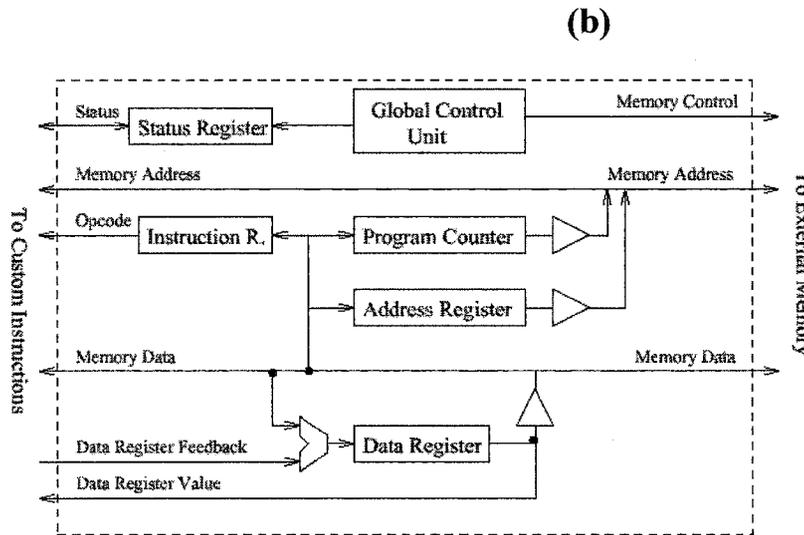


Figure B.2. DISC caching of hardware kernels corresponding to opcodes of an FPGA-based processor.

(a) Row-based kernels (one instruction module is one opcode) [WH95].

(b) Global controller/processor [WH95].



granularization; kernels took up entire contiguous rows in the array, and the array was partitioned into placement sites. Kernel I/O took place over predefined global signals lines that run vertically across the kernels (Figure B.2a).

DISC Performance Evaluation

The first DISC design used one FPGA for kernel caching and another to manage configuration loading (Figure B.3a). The test application was the blurring of a 128x64 gray scale image by averaging each pixel with its neighbours in a 3x3 block. The unaccelerated test case used only single-cycle opcodes, such as those found in a simple microprocessor; many cycles were spent calculating pixel addresses. The accelerated test case configured the array into 3 shift registers to

mimic the horizontal scanning of the image by a 3x3 block, thus minimizing address calculation and data loads. This took 4x the area, but ran 80x faster. If configuration overhead was considered, the speedup is 24x. Both implementations ran at 7.5MHz. Admittedly, though, a fair comparison would consider a non-FPGA based processor (or processors) with as many resources as could occupy the area of both FPGAs. However, it did demonstrate the customizability of opcodes.

A second DISC-II (Figure B.3b) [WH96] architecture was based on a 16-bit datapath. One FPGA hosted the processor, another controlled configuration and bus interfacing, and a third FPGA served as a kernel cache. Image processing kernels were developed for inversion, thresholding, histogram generation, edge detection, and high-pass filtering. The test algorithm recursively thinned solid foreground objects into lines (Figure B.4), and required twice the number of kernels that would fit in the cache. This ensured that kernel swapping was in fact exercised. Execution on an 8MHz DISC-II took an order of magnitude less time than a 64MHz 486 processor, even though 25% of the time was configuration and “instruction moving” (not explained). As a system integration effort, a retargetable C compiler [FH95] was modified to allow source level recruitment of kernels without resorting to assembler to access the opcodes.

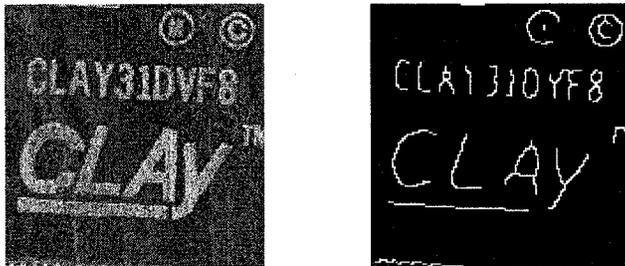


Figure B.4. DISC-II's recursive object thinning algorithm

B.4 Garp Details

B.4.1 Replaced Ultrasparc in Garp

- ◆ Bit-width: 64-bits
- ◆ Clock rate: 167MHz

- ◆ Speed of memory system: 133MHz
- ◆ 16KB instruction and data caches
- ◆ Process: 0.5 μ m 4-metal process
- ◆ Area: 17.5x17.8mm²

B.4.2 Garp Array Details

The leftmost column of the Garp array is reserved for tasks such as interrupting the main processor, array-initiated memory accesses, and register transfers with the host processor.

Processor Synchronization

- ◆ As an alternative to stalling the processor until kernel execution finishes, the processor can set the number of clock cycles for which a kernel executes.
- ◆ The clock for the array does not have to be the same as for the processor

2-bit Logic Block

Figure B.5 shows the Garp logic block architecture. Every 4-input 2-output array cell has a datapath width of 2 bits for each input and output. Thus, there are two sets of four inputs, and the cell's LUT is independently applied to each set of four inputs. Muxes allow the logic block to select its inputs from the wires or from internal feedback. A fast carry chain is propagated row-wise. Fast multiplication is facilitated by a special triple-add mode.

Configuration Loading

The configured part of the array must consist of entire adjacent rows.

Despite the 192 vertical wires for communicating outside the array (4 bit-pairs for each of 24 columns), the bus width for configuration loading is described as 128 bits wide. Each CLB has 64 configuration bits (this includes interconnect switches, since they are realized through the CLBs), so an array of 24x32 requires 384 clock cycles, or 3 μ s at 133MHz. If loading from main memory instead of cache, it will be about an order of magnitude more. Typically, however, only two thirds of the array is used, requiring 2 μ s configuration time from memory cache. Furthermore, configuration cache resources are distributed throughout array for recently loaded configurations, thereby avoiding configuration loading. The cache depth and width is selected at the time that the chip is

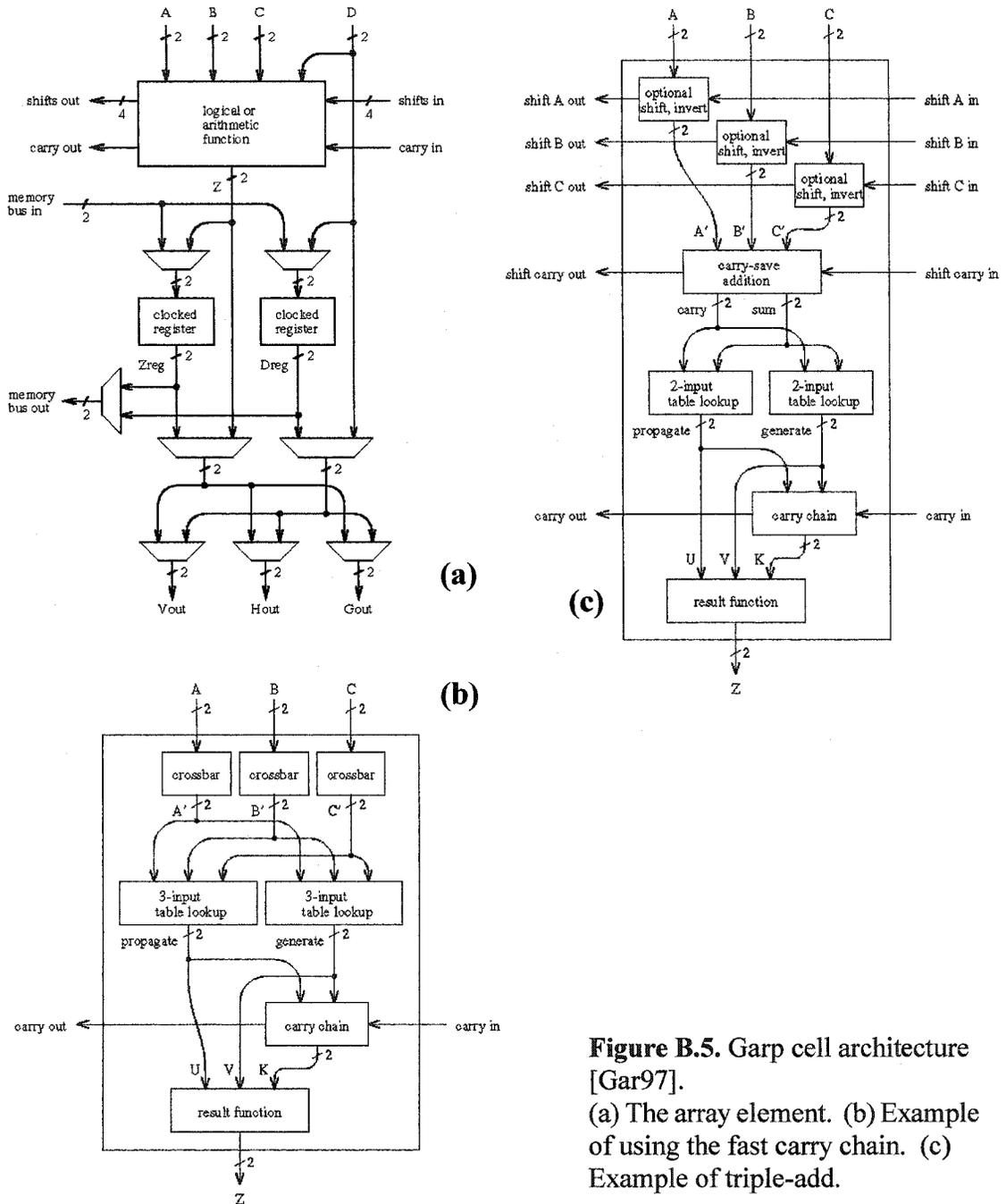


Figure B.5. Garp cell architecture [Gar97].

(a) The array element. (b) Example of using the fast carry chain. (c) Example of triple-add.

designed; it can be four configurations deep for the entire 32-row array or n configurations deep for an m -row array, where $n \cdot m = 128$.

B.4.3 Garp Performance Evaluation

In lieu of a designed chip, a simulator was developed to evaluate the performance on a number of applications, including DES encryption of 1MB file, dither of a 64x480 image, and a sort of 1 million records. Figure B.6 shows hand-coded kernels with 2~43 times less execution time than all-software execution on the UltraSPARC from which Garp started, as well as sample breakdown of array execution for a tool-compiled wavelet image compression program. The latter figures are not explained in detail. The Garp execution times for the hand-coded kernels assume that the configuration cache already holds the configurations, though there is no explicit mention of the configuration cache depth and width, or whether all configurations for a specific function fit within one width/depth setup.

It was concluded that, as with pipelining and parallel processors in general, the ability to keep all hardware constantly functioning depends on regular memory accesses. This permits the use of queues for high throughput data access and avoids contention for the single address bus. Not shown is an example for which this is not true; gzip mapping was slower than the Ultrasparc execution because of irregular data accesses and short executions times per configuration load.

B.4.4 Garp Low Level Mapping

At the hardware level, logic and interconnect within the array are well characterized with maximum delays so that the compiler can pipeline a cascade of logic. Initial efforts to systemize mapping include identifying a few common sequences of interconnect types and logic functions that take comparable amount of time, thus minimizing the time spent by some fast combinational logic waiting for slower logic to settle. The comparable propagation time determined the array clock period. Constraining the combinations of cascaded elements to these few predetermined templates also reduces the complexity of mapping datapath operations to the array and pipelining them. As well, it supposedly simplifies migration of designs to finer and faster fabrication technologies, since propagation delay of the identified sequences speed up roughly proportionally with each other.

B.4.5 Garp Compiler

At the C-code level of abstraction, behavioural to RTL decomposition is described in [CW98]. This work identifies the mapping problem as being similar to that found in pipelined processors

Garp's speedups over Ultraspac for hand-coded functions.

Function	Data size	Speedup	Limiting factor
Image median filter	640 × 480 pixels	43	Compute throughput
DES (ECB mode)	1 Mbyte	18.7	Compute throughput
Image dithering	640 × 480 pixels	17.0	Compute throughput
strlen	1,024 chars	14.2	Memory bandwidth
strlen	16 chars	1.84	Overhead
Sort	2 Mbytes	2.2	Scattered memory accesses

Kernels from a wavelet image compression program.

Kernel	Percentage of original software execution time	Iteration Interval	No. of queues used	ILP (average operations per nonstall cycle)	No. of executions	Average no. of compute cycles per execution (including stalls)	Average no. of overhead cycles per execution	Net speedup over MIPS only
forward_wavelet_696	18.2	2	2	10.0	448	1,176	114	2.1
forward_wavelet_647	13.8	2	2	10.0	448	310	91	5.1
init_image_354	12.8	1	2	8.0	1	65,852	564	12.7
forward_wavelet_711	10.1	2	2	7.0	448	241	59	4.9
entropy_encode_544	10.0	1	1	5.0	1	65,538	989	9.9
forward_wavelet_674	9.3	1	3	13.0	448	128	76	6.6
block_quantize_411	5.5	2	0	5.5	320	353	56	2.8
entropy_encode_557	3.9	6	0	2.8	3,262	31	24	1.4
RLE_encode_509	3.8	1	1	11.0	774	22	48	4.6

Figure B.6. Garp evaluation.

Top: Evaluation of hand mapped kernels on Garp [CHW00].

Not shown is tool-compiled “gzip” compression, 14% slower than Ultraspac, and almost the same as MIPS core alone without configurable array.

Bottom: Overall speedup factor of tool-compiled wavelet image compression was 1.68. Breakdown of wavelet image compression is good indicator of array utilization, but comparison with MIPS core alone is of questionable meaningfulness.

and instruction scheduling for parallel processors, namely how to deal with conditional exceptions to single-branch flow of operations. An approach is borrowed from VLIW compilers to identify parallelizable portions of a software loop called “hyperblocks” to be realized on configurable array. The general idea can be illustrated by considering a software loop to be accelerated by having the array run through the repetitive operations. The dataflow graph (DFG) of the code within the loop has divergences and convergences due to conditional statements (Figure B.7). A hyperblock is formed from the DFG by excluding flow branches to less common paths, or those realizable only in software. These hyperblock *exits* cause control to return to main processor for

handling of the functionality for that branch in software. A single point of entry to the hyperblock is maintained by duplicating portions of dataflow to be handled during return to processor, thereby preventing reentry to hyperblock upon an exit except from top on next iteration.

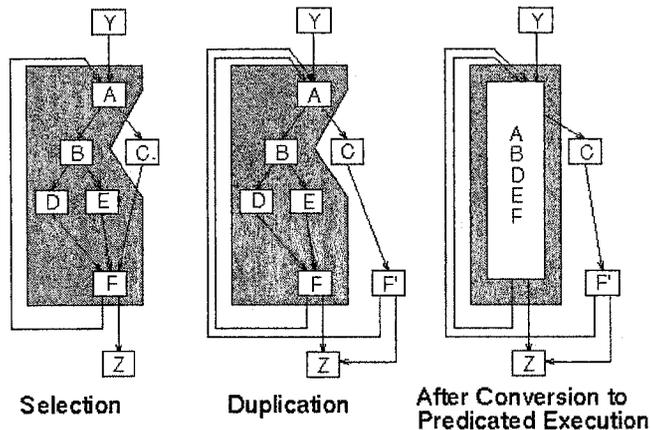


Figure B.7. Hyperblock formation for VLIW compilation [CW98].

Which branches within the hyperblock get executed at a divergence point is determined by the simplest method, grandly known as “partially predicated execution”. It just means that *all* contingent branches included in the hyperblock are parallelized onto the array and “speculatively” executed at the same time. The right result is selected by a mux controlled by boolean logic that matches controlling logic expression in the conditional statement. Since operations for all included paths are simultaneously performed, this scheme uses most parallelism to get the least delay penalty due to wrongly guessed instruction branches, as might be the case in a pipelined processor. However, it gives the greatest “fanout” in datapath, resulting in an explosion of required array resources. The branch fanout is reduced by selecting the branches to include in the hyperblock according to relative execution frequency, resource limitation, impact on critical path (how much remaining hyperblock speeds up without a particular branch), and whether a branch leads only to excluded branches. It is not clear whether the relative execution frequency of the branches are determined by the designer or by software profiling.

A DFG branch that contains a loop within it is also excluded from the hyperblock, since it is a nested loop and cannot be handled by the platform. This allows the remainder of the outer loop to still form a hyperblock for acceleration. However, the nested loop can form its own hyperblock to be loaded into the array when the original hyperblock exits due to the excluded path.

As opposed to running loops entirely in software, it is only worthwhile to run hyperblocks on the hardware if the loops iterate enough to recover time lost to configuration and context saving/loading (particularly if the configuration is not loaded from the configuration cache) and transfer of control. To avoid unnecessary transfer of control to the host processor, a modification is made to the conventional delineation of hyperblocks in VLIW compilation; datapaths that exit the hyperblock only to loop back and reenter at the top without any intermediate operations required of the host processor are considered internal to hyperblock, and control remains within array. This is likely to be a “normal” finish to the current loop iteration (i.e. the path that is expected to followed for most of the iterations in software), followed by the start of the next iteration, but “nonnormal” (i.e. “exceptional”) hyperblock exits may also do this.

There is some ambiguity about how the datapath operators are sequenced. The Garp manual [Gar97] describes a simple sequencer that counts array cycles in each loop iteration and activates the right portions of array at the right time, whereas the more recent article [CHW00] describes the hyperblock as a pipeline, which would imply that all DFG nodes are operating every array clock cycle. This author believes it is a pipeline where some stages take more than one cycle, as there is some vague discussion in the latter about varying degrees to which operations in successive loop iterations may be overlapped, depending on data dependence in the diverging branches.

The Garp’s C compiler front end was implemented on a platform known as Stanford University Intermediate Format (SUIF) compiler [HAA+96,AAW+96], a compiler development framework targeting multiprocessors (SIMD/vector processors). It uses a standardized and easily accessible format to represent the program being compiled as an object-oriented hierarchy, and contains tool-kits for identifying loop-level parallelizable operations from high level sequential code. It apparently uses the same retargetable C compiler as DISC [FH95] for its own front end [FH91]. However, DISC software code merely compiles to assembler for the FPGA-based processor rather than dealing with kernel mapping to the array, though supervisory opcodes were in place to load and invoke these kernels. Presumably, the DISC kernels were predefined and manually mapped. In contrast, the Garp compiler uses the more encompassing SUIF to identify parallelizable pieces of code with which to form hyperblocks to map to the array.

The resulting hyperblocks, which are just DFGs with excluded branches trimmed off, are fed to the a “Gama” synthesizer, described next.

The Garp compiler has piqued the interest of Synopsys to develop it further in [Yan] (a somewhat deep and theoretically involved compiler article).

B.4.6 Gama: Garp's Technology Mapping and Placement

The acyclic graphs from the compiler are fed to a custom mapping tool “Gama” [CCDW98] for technology mapping and physical design. Gama is a research program in the BRASS project for fast “low effort” synthesis targeting FPGAs (namely, Garp and Xilinx XC4000 series). It uses dynamic programming to explore a subset of the design space in linear time. Synthesis and placement are performed together, without flattening modules to gates, with the intent of exploiting high-level (datapath level) knowledge to improve an otherwise sloppy synthesis. Here, a “module” is a cluster of DFG operators corresponding optimized hard library macros, specific to an array architecture. The fact that placement information is considered during synthesis also reduces delays due to long routing.

Gama searches for a good partitioning of DFG into *trees*; these are single-output subcircuits in which each vertex, or gate, feeds exactly one gate (no fanout). In the case of FPGAs, gates are logic blocks. For each tree, dynamic programming is used to find a good tree cover from the library of hard macros i.e. the whole tree becomes an interconnection of macros. The modules are word-width operators occupying the entire height of the array. Note that, although Garp macros are designed row-wise, we describe Gama in the context of its publications; operators occupy the entire *height* of the array and have fixed or varying widths, while data flows from left to right. Some of the word-level optimizations that can be exploited in module design include operations that use the fast carry chain. Since operators occupy entire vertical slices of logic blocks, the placement reduces to a simple linear (1D) arrangement, or ordering, of modules.

To find a good partitioning, or clustering, of a tree into macros, matches are sought between subgraphs within the tree and modules (single-output subgraph *patterns*) from the library. In the context of Gama, the match is regarded as between the macro and the *output* vertex of the matching subgraph. For each vertex V in the tree, an attempt is made to match every library macro. V is then tagged with the lowest cost match; here, the cost of the match is the cost of the macro itself, *and* the costs associated with all vertices that feed the inputs of the matching subgraph S in the tree. Hence, the cost of S includes the cost of all logic that feed S , directly or indirectly, all the way back to the tree's inputs i.e. the *input cone* for the matching vertex. Such cumulative costs are calculated starting from the tree inputs and working toward the output i.e. a vertex is visited after any vertices that directly feed its inputs (this is known as “topologically sorted” order). The lowest

cost match at the tree's output vertex then represents the cost of covering the entire tree. Working backwards from the inputs to this last match, one can recursively determine the predecessor matches until the entire tree is covered by nonoverlapping matches to library modules.

As mentioned, this finds a good tree cover, though not necessarily optimum. The optimum tree covering problem is NP-complete. Since the above algorithm matches each vertex to each library module, it has the linear complexity $O(N_{\text{vertices}} N_{\text{modules}})$.

This tree covering algorithm also simultaneously explores linear placement of the module instances in the tree cover. To do this, an n -input module M has $n!$ patterns in the library, one per permutation of the inputs. They all have the same functionality, but each imply a different linear ordering for the logic that feeds M 's inputs. These chunks of predecessor logic are abutted side by side, with the matching module itself abutted on the right side. Therefore, each pattern match at V also implies an ordering of all modules that feed the matching subgraph, directly or indirectly. Note that *all* the modules for each input cone that feeds into V are treated as one big chunk of internally placed logic. This internal positioning was fixed when the lowest cost match was made at the predecessor vertex. Since each match at V implies a placement of all nonoverlapping modules in V 's input cone, the wiring delays are easily estimated and incorporated into the cost of the match.

B.5 Chimaera Details

- ◆ The array is designed such that no self-destructive states are possible e.g. due to conflicting drivers. It is not left to the mapping software to ensure this.
- ◆ Logic blocks have multiple LUTs and multiple inputs/outputs, allowing data forwarding at the same times as computation.

Chimaera Compiling, Placing, and Routing

A compiler was developed to map blocks of straight-line C code to array logic blocks[YSB00]. A heuristic place and route algorithm has been developed [HFHK04]. To minimize the number of rows used, the placement phase packs as many logic blocks as possible into each row, starting with the cluster of logic blocks with the longest carry chain (widest datapath). A logic block is only considered for packing if the logic blocks driving its inputs have been placed. When the packing procedure has accumulated a row's worth of logic blocks, a horizontal arrangement of those logic

blocks is sought to minimize the total distance of horizontal signals. (Note that the goal is not to minimize the longest wire length). The routing phase then tries to realize the required interconnections using a quick, greedy heuristic that is specific to the interconnect network; if a signal is not already available via any of the horizontal wires (e.g. because it was routed for some other logic block), then concoct some other (unspecified) means of routing. Any logic block whose input cannot be obtained through this routing attempt is evicted from the current row. This makes it a candidate for placement in the next row, which opens up routing options, but incurs greater latency.

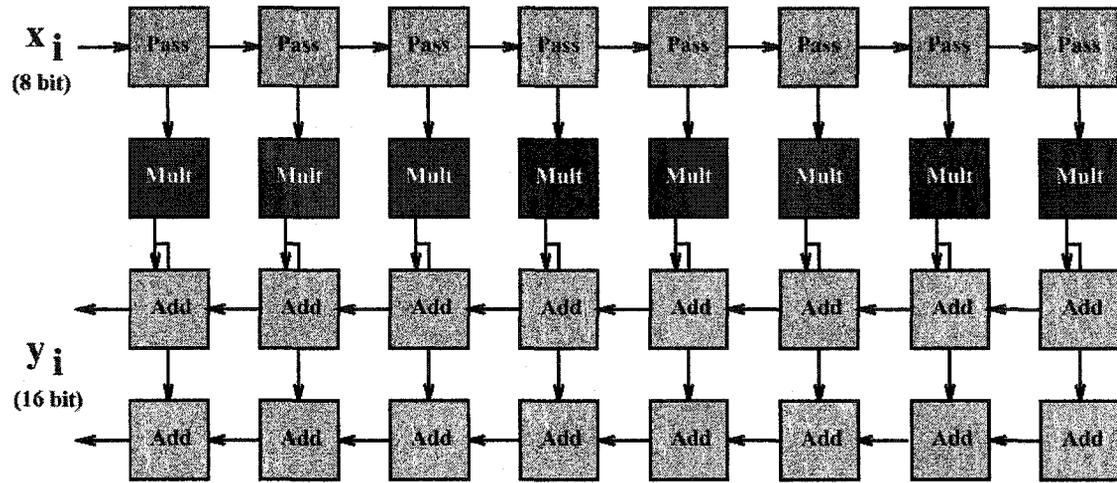
A plethora of work has also been done on relocatable kernels, techniques to manage configuration caching, general mapping issues, and multichip FPGA design.

B.6 MATRIX Details

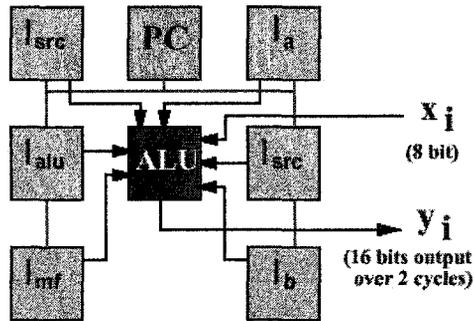
The MATRIX array was used to implement a FIR filter as a systolic array, a microcoded processor, a VLIW processor, and a VLIW/multiple-SIMD processor (Figure B.8).

B.7 Pathfinder Routing

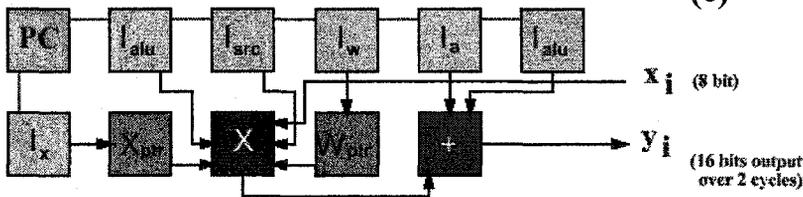
Pathfinder [ME95] departs from older routing methods that require separate global and detailed routing phases. The interconnect network is modelled as a graph, with the vertices representing wires and edges representing switches that connect wires. The fact that switches are typically clustered into switch boxes is not represented, nor is it relevant to the method. Each net is routed using the Lee algorithm [Vai01] (also known as maze routing, and analogous to Dijkstra's shortest path algorithm [Fou84]) to do a breadth-first search for the lowest cost path. For each iteration thereafter, each net is ripped up and rerouted in the same order. After each net is routed, wire costs are factored up according to their overuse. Thus, signals are encouraged to use less crowded paths in successive iterations. To reflect the priority of critical signals, the actual cost of a wire is a combination the overuse and the time delay of a signal. If there is little time slack, more weight is placed on the time delay.



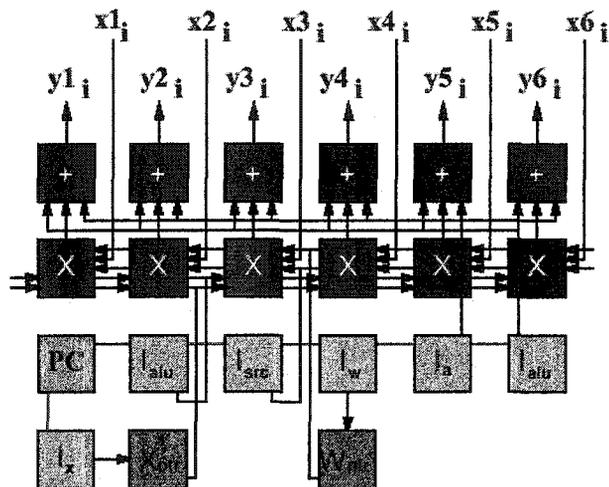
(a)



(b)



(c)



(d)

Figure B.8. MATRIX configured to perform convolution in four different ways.

- (a) Systolic array.
- (b) Microcoded processor.
- (c) VLIW processor.
- (d) Hybrid VLIW/multiple-SIMD processor.

B.8 Pleiades Details

B.8.1 Pleiades Interconnect

Interconnect schemes were explored in [ZWGR99]. A 6-metal process was targeted. Several global interconnect schemes were dismissed as being inefficient for area or energy-per-operation: full crossbar, multistage interconnection networks to achieve full crossbar connectivity, and multiple parallel global buses. These were also deemed unnecessary for intermediate specificity, since the required configurability is limited and predictable.

In contrast to the uniformity of CLBs in FPGAs, the satellites were at a higher level of abstraction and very heterogeneous; a simple binary tree network was not considered suitable.

To partition the satellites for local and global interconnect, the satellites were grouped into clusters based on the communications needs of kernels drawn from speech coding applications. For local connections, they took the Xilinx 4000 series FPGAs as an example and generalized the homogeneous mesh to route between satellites. The clusters have ports that connect to a global mesh, which for some reason was also routed between satellites. The clustering process greatly affected the congestion at the ports.

This two-level hierarchical mesh resulted from trying to minimize the power consumed by communications. In comparing different schemes, a 10ns limit on propagation time was imposed. It was shown that the optimality of an interconnect topology was very dependent on the target applications, as well as the number of parallel lines made available. In [ZWGR99], 2 lines per channel was chosen to be able to support the target applications.

B.8.2 Pleiades Power Reduction

Power reduction measures considered for Pleiades included adjustable VDD. For times when throughput is not a priority, lowering VDD costs less energy per operation. It is not clear what provisions were made in the prototype to operate in this manner, or if VDD can differ between satellites. It seemed clear that the DC-DC converters were not built into the chip. Energy and area efficient DC-DC converters in CMOS are an active area of research.

To be able to use a lower VDD, asynchronous clocking was considered so that each satellite is clocked only as fast as necessary. A clock can be suspended when a satellite is not used. The communications network must then be designed to support synchronicity. It was not clear what fea-

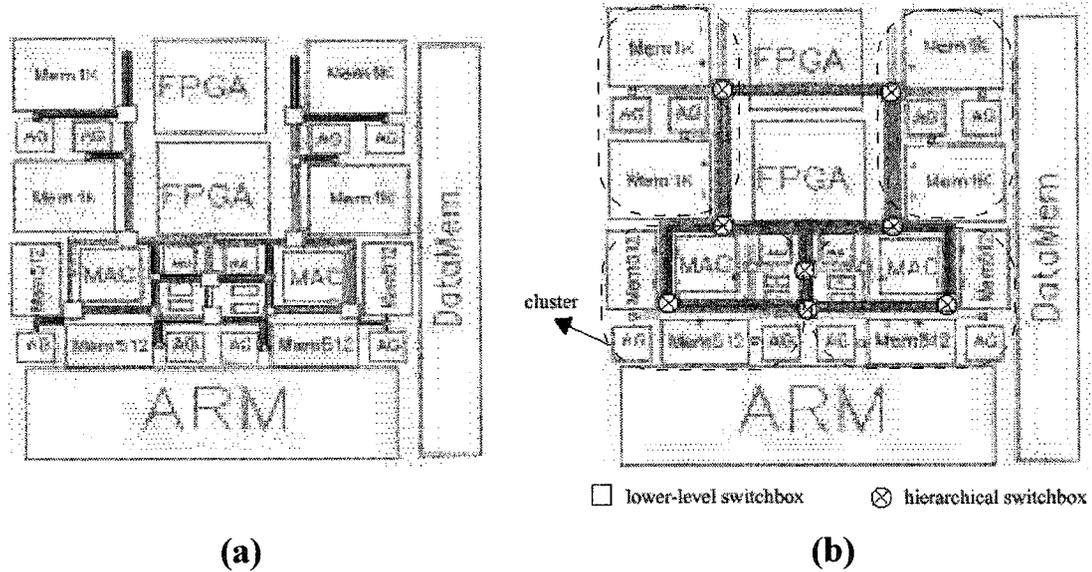


Figure B.9. Hierarchical 2-level mesh for Pleiades [ZWGR99].
 (a) Lower mesh. (b) Upper mesh.

tures were incorporated into the test chip to support asynchronous clocks; it seemed clear that different voltage-controlled oscillators (VCOs) were not built into each satellite.

B.8.3 Pleiades Prototype Part

The first prototype, “Maia”, was a 1.2 million transistor device with the following characteristics.

- ◆ ARM8
- ◆ 4x8 FPGA of 5-input, 3-output CLBs
- ◆ 2 MAC units, 2 ALUs
- ◆ 4 RAMs of 512x16 bits, 4 RAMS of 1Kx16 bits
- ◆ Local and global interconnect meshes
- ◆ 1V VDD
- ◆ A differential signalling optimized for low power

The $5.2 \times 6.7 \text{mm}^2$ device was built with SGS-Thomson's 0.25 μm 6-metal CMOS

A number of functions were mapped to the platform and analyzed for power: CDMA correlator [Rab97], DCT, motion compensation, VSELP speech encoding, and an adaptive LMS filter for channel estimation in a multi-user detection [WZBR99]. Most functions could be decomposed into satellite operations: dot product, FIR filter, IIR filter, vector sum with scalar multiply, and computation of covariance matrix.

It is not clear whether performance measurements were made in the lab. [ZPG+00] mentions voltage rails of 1.1V, and the separate testing of kernels and ARM programs to estimate energy dissipation. This should be interpreted conservatively, since simulation and analysis is referred to as "experimental" in [ZWGR99]. More clearly, similar efforts by Chandrakasan and Brodersen, also with Berkeley Wireless Research Center, demonstrated a chip set for multimedia terminal. It was optimized for low power and operated on 1.3V, with performance predictions for 1.1V. It does not appear to be part of the Pleiades project, though InfoPad and IPGraphics were mentioned as the test vehicle.

B.8.4 CAD Development

[WZG+01], [LWR99], and [WILR98] describe efforts in developing CAD for Pleiades. The aim was to synthesize from algorithmic specification, starting with identifying portions of software code that can be accelerated as kernels. Kernels were then mapped, first based on speed requirements, then on energy efficiency. Also described are ways to capture front-end data/control flow information, details of binding with satellites, and analysis of efficiency. Because of custom terminology, the details are not always clear. The general idea is that early design exploration is facilitated by different mapping methods at different levels of abstraction. These include rudimentary mappings to establish power/speed bounds, mappings that consider simple performance/resource metrics, libraries of commonly used kernels, and manual mapping coupled with performance/resource analysis tool.

B.9 PipeRench Details

B.9.1 Technology Migration

Aside from its effect on throughput, the length of the virtual pipeline is independent of the number of stages on the physical device. For successive generations of devices that fit more stages onto a die and run with a faster clock, the exact same compiled configurations will run with greater throughput.

B.9.2 PipeRench Computational Model

Virtualizing a pipeline imposes constraints on configuration speed and communication between pipeline stages over more than one stage. The configuration of any one stage must be completed in one clock cycle. Furthermore, the current stage can only depend on the contents/state of the current and preceding stage. These limitations actually help reduce the design space of the device and suggest an architecture for interconnect (Figure B.10). Any processing element (PE) can read the registers of the PE immediately above, and any data in any PE in the same stripe as itself. Vertical global buses are also required because the pipeline stages may not follow their physical ordering on the device. A clever feature is that if the PE's ALU does not write one of its registers, that register gets its value from the same register in the PE immediately above. This behaviour has some resemblance to a microprocessor, where an unwritten register carries its value into the next clock cycle. Here, values not overwritten are carried into the next stage for use by the PE/ALU.

B.9.3 PipeRench Compiling

To map behavioural code to the device, a compiler was written to accept a C-like description of datapath operations, with optional bit-width specification. By default, the bit-width is inferred to avoid overflow. As in Verilog, straight-line code is generated by in-lining module invocations and unrolling loops. Logic for operators are then optimized e.g. constant multiplicands generate shifts/adds. Operations exceeding one clock cycle are broken up.

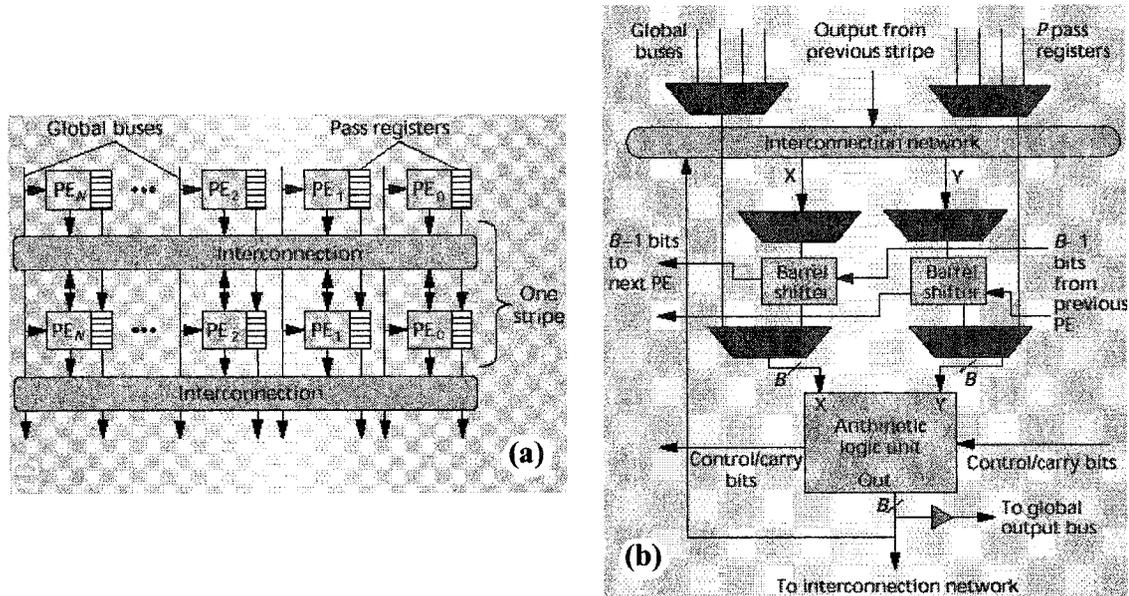


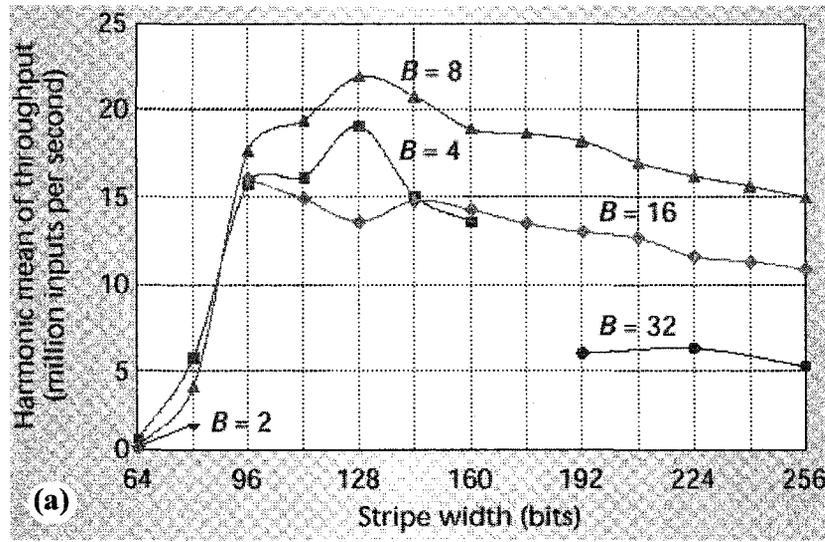
Figure B.10. PipeRench architecture [GSB+00].

(a) Fabric of processing elements (PE) forming the pipeline of a PipeRench device. One stripe is one pipeline stage, and the PE register file acts as pipeline registers. (b) A PE element. Each ALU has LUTs and logic for carry chains and zero detection, among other things.

B.9.4 Exploring PipeRench Architectural Parameters and Implementation

The performance of the device was investigated using different stripe widths, PE widths, and register file lengths based on mapping of a broad cross-section of algorithms, including automatic target recognition, 1D and 2D DCT, FIR filtering, and data encryption. It is not clear if this design was ever submitted for fabrication, though a PE was synthesized with CAD tools and the interconnect layout was custom designed to extract delay information to assist in evaluation. The area was evenly split between the array of PEs versus configuration storage, control, and I/O (50mm² each). The role of the register file as interstripe interconnect was emphasized as an important use of PE space because it alleviated routing congestion and improves stripe utilization. The optimal choice of parameters was 128-bit stripes with 8-bit PEs and 8 registers per PE, 100MHz operation (Figure B.11). Throughput was compared across the different algorithms, which accounts for

functional density, interconnect congestion, stripe utilization, delay due to logic and achievable interconnect, and the number of PE's fittable in one stripe, and the number of stripes fitting in the allotted 50mm². PipeRench was found to have an order of magnitude higher throughput than conventional processors.



(b) Comparison of IDEA implementations with other designs.

Processor	Clock speed (MHz)	Clocks per block	Throughput (Mbytes/s)
PipeRench	100	6.3	126.6
Pentium II using MMX ⁵	450	358.0	10.0
Pentium ⁶	450 (scaled)	590.0	6.1
IDEACrypt Kernel	100	3.0	90.0

Figure B.11. Optimizing PipeRench[GSB+00].

(a) Optimization of word width and strip width over a suite of kernels.

(b) Superiority of PipeRench over other platforms.

B.9.5 PipeRench Interface to Host Processor

PipeRench has been further developed to be application friendly and to interface with a PCI bus in PCI-PipeRench [LTS99], with plans to circumvent even the bus as an interface to the host processor in later efforts. Verilog models have successfully cosimulated with client applications, and VLSI design was in progress.

B.10 Wormhole Details

B.10.1 Wormhole's Datapath-Heavy Logic Resources

In describing how the design is based on wormhole concepts, no distinction is made between dynamic control versus datapath configuration. While there are control structures in the interconnect switches and IFUs to support the distributed loading of configurations, there are actually few resources for users to design control structures, such as narrow bit-width state machines, perhaps in keeping with the stream-based paradigm. This places the onus on the surrounding system to provide data at exactly the correct times for continuous operation of the logic, as later realized in [Bit97]. Control within the device is realized by the three flags, carry, shift, and conditional, that pass in and out of each IFU, with interconnect support paralleling the datapath.

B.10.2 Wormhole Configuration Loading

Colt loads configurations using the high bandwidth datapath by prepending the data stream with configuration data. For each IFU along the intended datapath, a header packet is prepended to the stream to carry the configuration information. As the head of the stream passes each IFU in the datapath, the associated header packet is read and removed from the stream, thereby forwarding only the minimum necessary header packets. The packet also configures the interconnect to establish connections to the next IFU for the following packets, and the actual data stream thereafter. Thus, the stream's head "worms" its way through the device, with the data body following immediately behind. This scheme supports partial configurability in the widest sense, as the process of establishing one stream's pipeline need not interfere with the operation of an existing stream.

Header packets can have more than one IFU as the "next" IFU, allowing streams to diverge as successive packets are sent down both paths (or all paths). The target IFU for each header packet is still uniquely identified, so different branches of diverged streams can be programmed differently. Different streams or forks can converge, exchange data, and affect each others IFU functions.

Embedding configuration data in the data stream is similar to "in-band" signalling over switch networks in that the configuration data uses the same physical transfer network as the data. It is much more efficient than having a separate physical configuration distribution network if a single configuration is applied over a large set of data. Conforming with the tenet of distributed coordi-

nation, prepended configuration data makes the local IFUs and switches more complex to simplify the global infrastructure i.e. no configuration load network and no global controls to use the datapath for the same purpose. More circuitry is needed within the IFUs and switching points to implement the extra intelligence for monitoring the stream, removing their packets, and configuring themselves.

B.10.3 Wormhole's Allocation of Device Resources

Allocation of the IFU and interconnect is performed off-chip by stream controllers. Relocatable configuration code is tailored by resource arbitrators after they allocate a sequence of IFUs through the device. The relative positions of the IFUs in a stream are not predefined, allowing flexibility in fitting them into freed resource e.g. like hard pieces of a soft macro. It seems that configuration of the IFUs and interconnect for a stream is static for the duration of the stream.

B.10.4 Wormhole Implementations

B.10.4.1 Wormhole's First Prototype

Colt is a coarse grain device based on a 16-bit data width, and consists of a 4x4 grid of IFUs (Figure 3.9). It was submitted for fabrication on a 3-metal 0.8um CMOS process. The die is 7x7mm² with 132 pins. Despite the localized interconnect philosophy, predicted clock frequency is only 50MHz, with a major limitation cited as the carry chain within each IFU. No mention is made of the crossbar as a speed limitation. No reports of results based on real parts could be found, though [Bit97] contains hand mappings of dot product, floating point multiplication, factorial, and despreading correlator based on matched filtering. Based on the despreading application, [Bit97] also performs numerous comparisons of metrics and rationalizations showing the superiority of Colt, and its successor, Stallion, over a hypothetical RISC and Xilinx's XC4013 and XC6216.

B.10.4.2 Stallion: The Second Wormhole Design

The successor to Colt is a bigger chip dubbed "Stallion" [He98], with two grids of 8x4 IFUs [SRA+00]. Some ambiguity exists about the crossbar, depicted in [He98] as a full crossbar, whereas [Bit97] describes that the full crossbar could not be maintained because of the $O(n^2)$

demand for switches. Multicast switching is supported. The device is expected to run at 100MHz. Global control wires used to enable stalling of configured pipelines. This addressed a problem with the Colt design, wherein null-tagged data samples were clocked in to avoid stalling the pipeline (and thus avoid global control wires). DSP circuits typically are not meant to handle streams of data interspersed with dummy samples at arbitrary locations.

B.11 Xputer Details

This Xputer architecture has been presented under different names, depending on the stage of development, including *rDPA* (reconfigurable datapath array) and *KressArray* (Kress being one of the researchers). Each array element is a reconfigurable ALU, referred to as a (reconfigurable) datapath unit ([r]DPU). His work is described here in 2 broad phases: (1) pre-mid-1990's and (2) mid-1990's and beyond.

B.11.1 Pre-mid-1990's [HHW90,HHR+91,HKR94]

The guiding operational model for the array is for highly repetitive computations, with extremely regular control flow. Computational tasks are programmed onto the array; data then flows through these operations in the array downward and/or rightward. This suggests that the model was motivated by systolic or wave-type calculations, though the array can apparently handle nonsystolic computations as well [HHW90]. Pipeline registers in the connections to nearest neighbours can apparently be used for storage of numeric constants. Global buses allow the import/export of data between DPUs and the outside world (there are vague insinuations of using global buses for communications between nonadjacent DPUs as well). For scalability, the DPUs at the edge of the array also serialize data going off-chip, and deserialize data coming on-chip. This circumvents the pin-count bottleneck to inter-chip communication, thus allowing arrays to be tiled to make bigger arrays.

Data flows asynchronously, thus avoiding the need for a clock. Strangely, [HHR+91] also emphasizes that the operational model's difference from data flow machines is that many more compilation decisions are done at compile time rather than run time, which is suitable for highly regular control flow. Apparently, this differs from a data-driven machine's nondeterministically triggered operations "by firing".

The control flow is assumed to be of such regularity that there is no provision to receive any instruction stream at all; even the program counter is dispensed with. In its place is a data sequencers, or memory address generator, to retrieve data from memory in various patterns. As the data flows through the array, the array is viewed as window that scans through a 2D array of data in memory. This is referred to as scan caching, with the array itself acting as the scan cache. The need for explicit control flow is minimized, and any residual control flow needed is done by tagging data samples entering the array as containing control information.

There are repeated references to coarse granularity, flexible bit-width, and Xilinx FPGAs as an implementation target. It seems that at a concept level, the array elements are coarse grain, but the realization is done with fine grain FPGAs i.e. a coarse grain reconfigurable array built out of a fine grain reconfigurable array. A compiler was described as generating both the DPU array and the configurations for it, and as capable of targeting both FPGAs and gate arrays.

Use of the array was illustrated with a simple software loop containing no inter-iteration dependencies. A 16-point FFT was vaguely described. Brief mention was made of the platform's potential for accelerating EDA algorithms, such as Lee's routing algorithm and ERC/DRC. In [HKR94], several kernels are mapped to FPGA and standard cell implementations of rDPAs in very general terms. Kernels include 1024-point FFT, 1D & 2D FIR, and bubble sort. Performance for the standard cell implementation was estimated, while the FPGA implementation was underway.

B.11.2 Mid-1990's and Beyond **[HK95,HHHN99,HHHN00a,HHHN00b]**

In [HK95], it was realized that building a coarse grain array using fine grain resources does not allow the low-level customizations of the coarse grain circuitry. Emphasis shifted to architectures for a true coarse grain array. This effort included optimizations in compiling a C-like language, motivated by a 1-level looping example, and high-level synthesis (HLS) optimizations. Bandwidth limits on the global bus is vaguely described as being addressed by scheduling of I/O. Operations are HLS-scheduled across clock cycles (this seems inconsistent with previously described data-driven operation). Design of a second implementation was mentioned, using standard cells (it is not clear whether the first one was an earlier ASIC or the above FPGA realization).

The compiler for the array performs high-level technology-independent manipulations on a kernel design prior to mapping to the KressArray. The mapper takes the kernel, along with parameters that describe the particular realization of the KressArray, and uses simulated annealing (SA) to perform place-and-route, followed by I/O scheduling. The KressArray Xplorer is a wrapper around the mapping utilities that generates performance logistics and statistics based on the mapping; it then generates suggestions for modifying the array architecture. For example, it might suggest modifying the amount of interconnect or the operations supported by the DPUs. It also recognizes that global interconnect is slow, and only uses it when “other” communications resources are inadequate (from this, it seems that nearest neighbour interconnect doesn’t just connect adjacent DPUs). The finally fine-tuned array can be rendered into Verilog for simulation or physical design. The estimated area for a 10x16 array of 18-bit DPUs was 10mm².

[ARNL+03] recently illustrated how to repeatedly reconfigure an rDPA’s interconnect to perform an 8-point FFT.

B.12 MorphoSys Kernel Scheduler

[MKF+01] is a comprehensive high-level analysis of kernel scheduling, again using MorphoSys as a model architecture. The kernels are taken as predefined. A loop body is formed by a single thread of sequential kernels, and inter-iteration dependencies are not considered. Effectively, loop fission is applied to the kernels to various degrees in order to explore the trade-offs between data-centric and kernel-centric operation. The optimum schedule takes into account time savings from overlapping data/kernel loading with kernel execution, as well as the limited memory in buffering data for kernel-centric operation. Consideration is also given to reusing configuration information. The platform can be reconfigured by the *configuration context word* (either 256 or 8 bits, depending on the paper [MKF+01,SLL+00]), and an algorithm selects the words to re-use.

B.13 Chameleon Details

- ◆ On-chip ARC processor [Tur00] for scheduling kernel loads/execution and data I/O
- ◆ Programmable I/Os
- ◆ PCI interface

- ◆ DMA engines.

B.14 Other Platforms

This appendix presents light details on REACT [GB98], Sonic [HSC+00], PAM (Programmable Active Memories) [VBR+96] and PAMETTE [MS97], Splash [GHK+91] and Splash-2 [BAK96] for historical perspective, Paddi, Remarc, and Chess. Most of the information comes from [TB01].

PAMs are FPGAs that interfaced with a host processor on a backplane bus via memory read/write transactions, originally designed to explore FPGA-based reconfigurable coprocessors. Perle-0 and DECPeRLe-1 are PAM-based reconfigurable computing engines developed by DEC. Pamette reconfigurable computing platform, based on 4000 series FPGAs, is a scaled down version of PAM emphasizing high bandwidth I/O for pipelined processing. An excellent history of PAM and PAMETTE can be found at <http://www.research.compaq.com/SRC/pamette/overview/>.

The original Splash was one of the first reconfigurable hardware platforms, used for DNA pattern matching. It had strictly systolic (nearest neighbour) communication. Splash-2 targeted image processing using block-systolic computation, and had sixteen Xilinx XC4000 processors in a systolic arrangement, each with its own SRAM. Among other things, it was used for target recognition [RH97] and for a genetic algorithm applied to the travelling salesman problem [GN95]. Communication constraints were relaxed with inter-FPGA crossbars for multi-hop and broadcast communication. VHDL circuits had to be designed for each processor, while scheduling software synchronized the communications. These machines were not reconfigured at run time.

On the side of 2D arrays of coarse grain processors, *Paddi* consisted of 8 ALUs and localized memory. Global instruction addressing of locally stored configurations gave high instruction bandwidth and I/O bandwidth. The interconnect switch between processors changed cycle-by-cycle. *Remarc* is a 2D array of 16-bit processors with global SIMD-like instructions. Communication took the form of nearest neighbour interconnect and horizontal and vertical global buses. *Chess* consisted of 4-bit ALUs with pipelined interconnects. Each tile has memory for data or instructions.

B.15 Interconnect Networks

We examine interconnect schemes are meant to scale well with system size. HSRA is a binary trees with retiming registers. A newer H-tree is also presented, with sample mappings. aSOC is a manhattan mesh with configurable crossbars.

B.15.1 H-Tree Networks

High-Speed, Hierarchical Synchronous Reconfigurable Array (HSRA) [TMJ+99] is, from the BRASS group at Berkeley, addresses the problem of clock frequencies on FPGA designs being an order of magnitude below CISCs and RISCs in the same technology, despite fast LUT propagation times and flip-flop clock-to-output delays in the order of 1.5ns. While a number of factors are provided, including the dominance of interconnect delays in large designs, reasons specific to FPGAs include lack of retiming support in FPGA tools and lack of available pipelining/retiming registers. The latter contribution is especially important when pipelining is needed to break up the propagation delay of long interconnects.

The HSRA architecture facilitates retiming by better quantifying interconnect delays using a binary tree wire network, similar to H-type tree structures for clock distribution [TMJ+99] (Figure B.12). Registered and nonregistered switches at the T-junctions allow for optional pipelining. Pipeline registers are placed along long wires as required by the target cycle time. *This means the clock speed is determined at design time.* Where space permits, this binary tree is augmented by cartesian “shortcut” wires for direct connections between adjacent corners of the hierarchical H’s.

HSRA’s basic logic block (BLB) is similar to an FPGA’s CLB in that it is LUT-based. Instead of registering the output, however, each input has a multitapped shift register of four flip-flops (Figure B.13). Each input selects from its tapped delay line as required by retiming. Independently choosing the retiming register depths at the different destinations is preferable to a single output register at the common source because the different logic/wire delays along different branches require different retimings. The alternative of providing *multiple* retimings at the source output using a multitapped delay line is that wire is wasted by routing several retimed versions of the signal up the wire tree to the points where the individual signals separate from each other. Though not mentioned in the paper, this problem is exacerbated by the hierarchical nature of the wire network. In the absence of a wire tree, the different delay line taps are not be forced to traverse a common path up a tree because more direct routing to the different destinations may

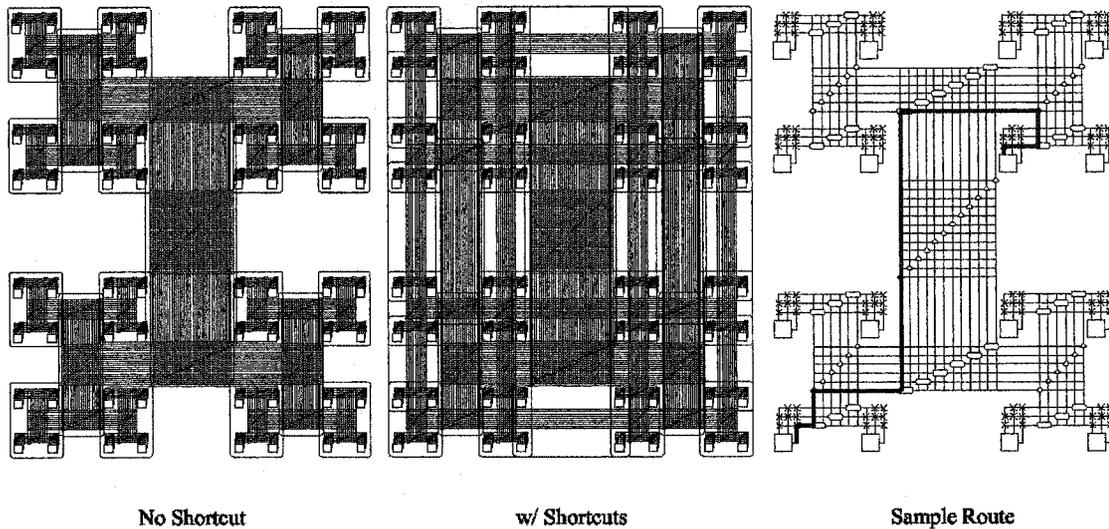


Figure B.12. HSRA's hierarchical interconnect network takes the form of a nested "H" shaped binary tree [TMJ+99].

exist. Using delay lines at the different destination inputs tips the scale of resources from combinational logic to registers to alleviate the interconnect bottleneck.

To configure the interconnect, a netlist is first mapped to HSRA without considering the interconnect registers. The input delay lines are then used to equalize the delay from LUT to LUT, including interconnect registers. Deep retiming requires the use of cascaded BLBs.

HSRA was implemented on a 0.4 μ m DRAM process (for reasons unexplained). It is not clear when or if the device will be fabbed/characterized. Evaluation of mappings to the chip concluded that a binary tree is not flat enough and creates too many hierarchical levels. A flatter tree with wider fan-out at each node is recommended.

A more recent attempt at an H-tree based FPGA was made by Myjak et al. [MADF04]. This is a medium grain array consisting of 4-bit CLBs that can also double as 64-byte RAMS. The H-tree provided the global mesh, and was intertwined with a local mesh-based wire infrastructure. Mapped macros include a 16x16 multiplier and a 254x32 memory with address generator. Macros were integrated into a 512-point FFT kernel. The FFT stages were executed with "ping-pong" processing. Unlike in-place processing, input data was read from one memory and output written to another; the roles of these two memories are then reversed for the processing of the next stage.

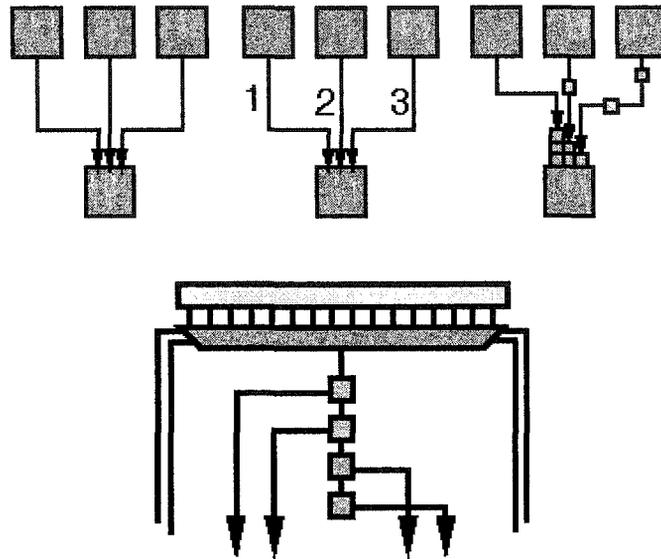


Figure B.13. HSRA deskewing
 [TMJ+99] Delay equalization using tapped delay lines at destination BLBs' inputs (top), and a tapped delay line at the source BLB output (bottom).

B.15.2 aSOC

B.15.2.1 The aSOC Architectural Concept

The Adaptive System-on-Chip (aSOC) [LST00] from University of Massachusetts is an effort in configurable interconnect rather than configurable logic. It permits the flexible integration of many cores, including reconfigurable computing engines. One of the primary motivations was to allow the interconnect network to scale well with communication bandwidth as the number of cores increases. Thus, wires and switches are not committed statically to specific signals. However, the scheduling of wire and switch usage is almost entirely performed at compile time, and is restricted to applications that lend themselves to this kind of deterministic data communication pattern. By trading off the flexibility of run-time determined routing, aSOC avoids the bandwidth bottleneck of arbitrated buses for large systems, as well as the circuitry to support the arbitration scheme. It also avoids the header processing delay and circuitry of a packet-based scheme.

The aSOC interconnect is based on a rectangular mesh with programmable crossbars at the intersections (Figure B.14). Each crossbar also connects to an IP core, and can be configured for

many-to-many simultaneous connections between the interconnect and the core. It is not clear if the signals are registered or buffered at the crossbar inputs. The switch normally uses a program counter to cycle through the locally stored configurations determined at compile time (Figure B.15). Each configuration also stores a jump control bit and a jump address to deterministically jump to a new configuration in local memory. The jump bit can also be activated by the IP core associated with the switch, allowing the IP core to determine how many words to transmit during run time. The connection with the IP core has input and output FIFOs, allowing the cores to have independent clocks, though the feasibility of this remains to be seen; integration of different clock generators on a common substrate is tricky because they tend to affect each other's behaviour via parasitic coupling (most notably through the substrate).

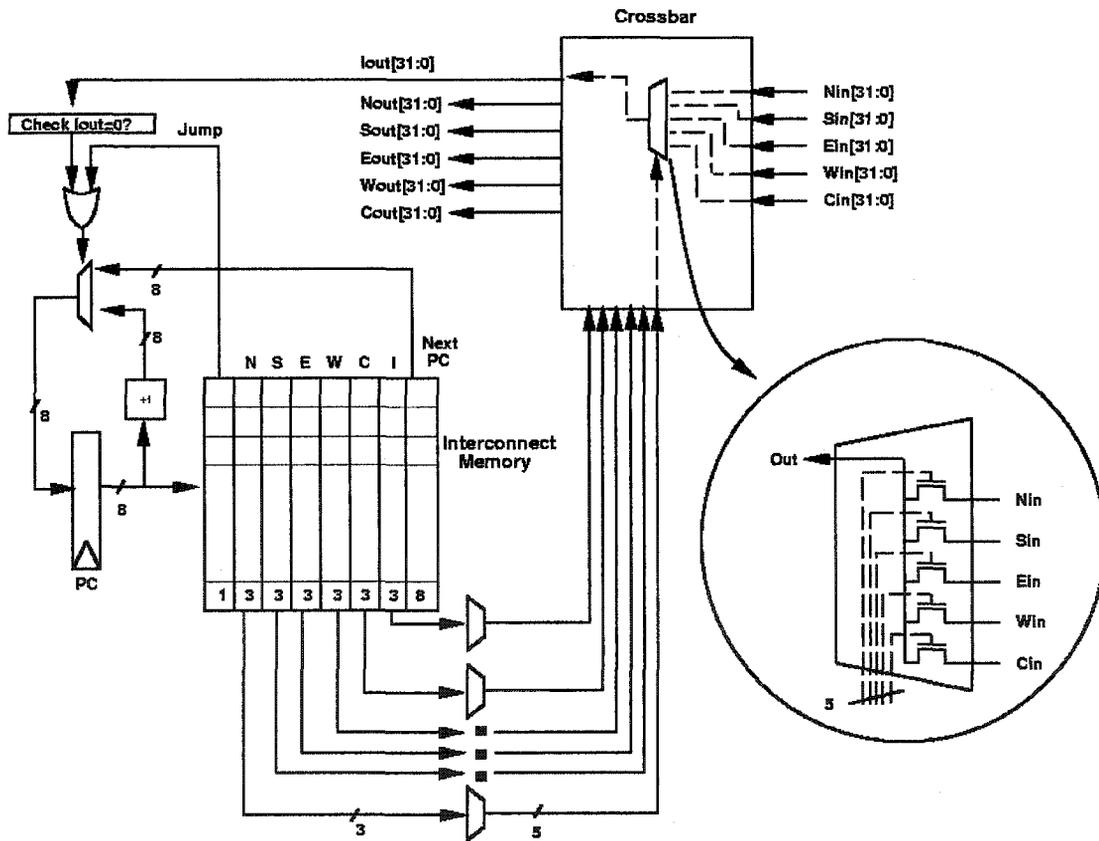


Figure B.15. aSOC crossbar switch and control circuitry [LST00]. The “C” signals connect to the core through input and output FIFOs.

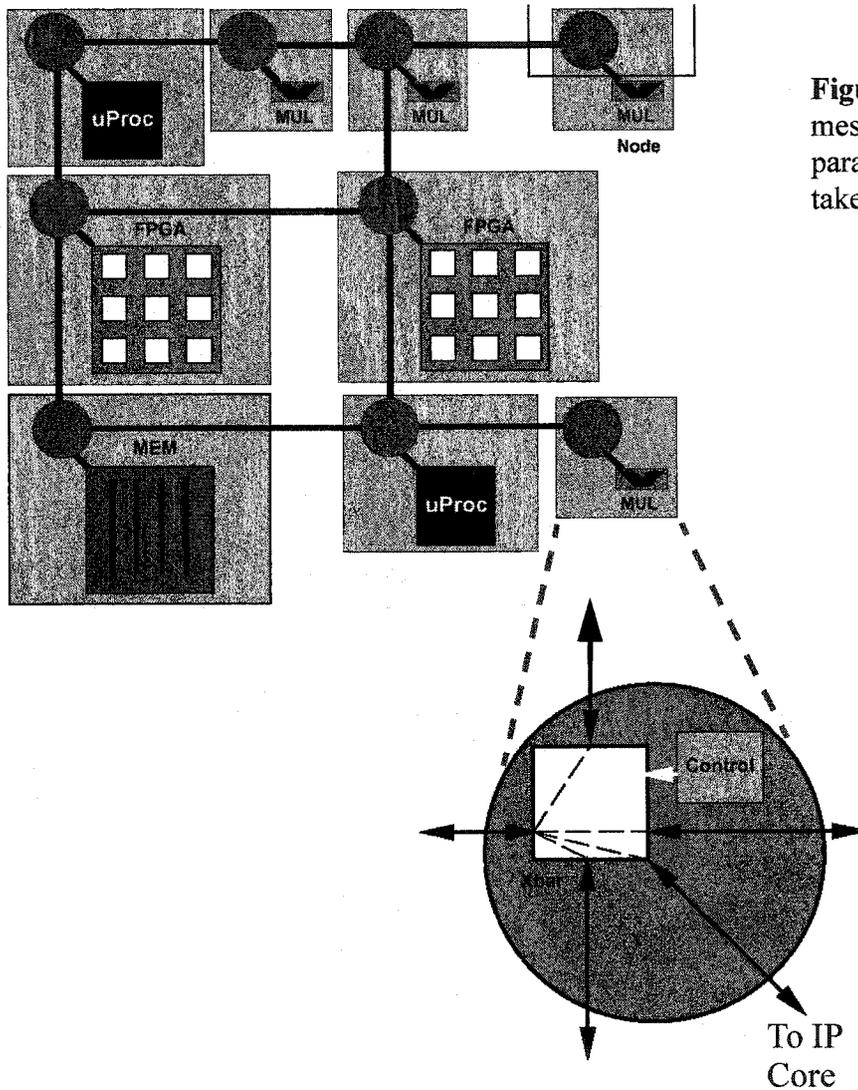
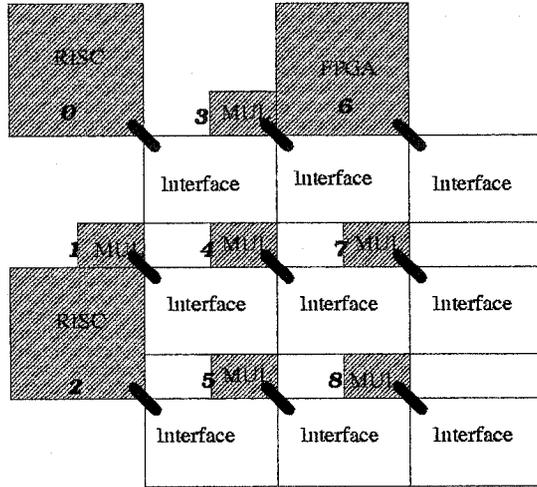


Figure B.14. aSOC's mesh to integrate disparate cores. Details taken from [LST00].

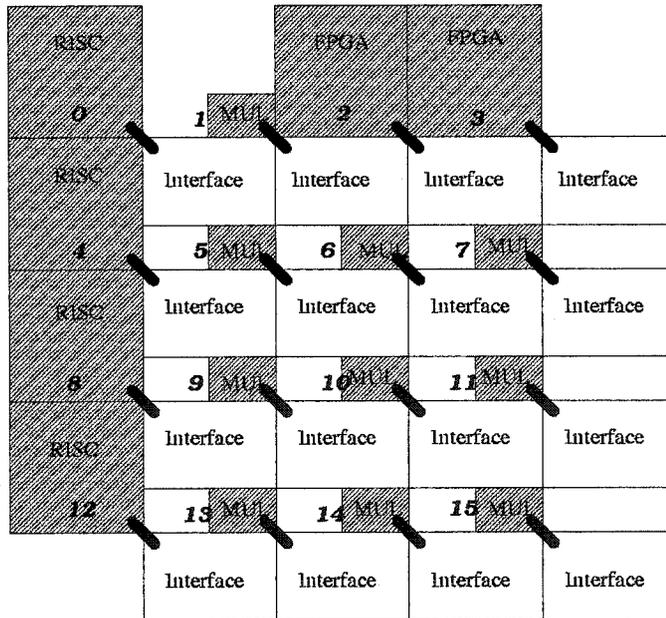
B.15.2.2 Evaluation of aSOC

Two designs were studied, with 9 and 16 cores (Figure B.16), consisting of MIPS R4000 RISCs, Altera FPGAs, and MAC engines. Each core type was simulated using its own native simulation tool and “integrated” using a high-level software package from MIT known as NSIM [Met97]. NSIM was developed specifically for modelling data communications over interconnect meshes in a heterogeneous assembly of cores. The hooks for simulating different cores are pre-

sented as function calls to NSIM, so the extent of integration of NSIM with the native simulators for the different cores is not clear.



9-core Model: 2 RISCs, 6 MULs, 1 FPGA



16-core Model: 4 RISCs, 10 MULs, 2 FPGAs

Figure B.16. aSOC arrays Each switching elements at the intersection of a cartesian interconnect grid also interfaces with its associated IP core [LST00]. Two assemblies of heterogeneous cores were considered.

The baseline to which the two architectures were compared was a single MIPS R4000 implemented in 0.3um technology, with a MAC core and no cache. This required a 5 cycle memory access, with the cycle rate at the fastest possible, 280MHz. The article did not specify whether this

applied to just the data or instructions as well. The R4000 specifications could not be located to confirm whether 280MHz referred to the data retrieval rate or the bus clock rate. There was also no rationale provided for the baseline; one would expect several processors and caches, or a more powerful processor, to be taken for the baseline in order to match the areas of the two multi-core designs.

To calibrate the simulations and comparison, layouts were created for the switch elements, FPGAs, and MAC in 1.2um technology. No explanation is provided for the difference in technology with the baseline R4000, though it is possible that it is determined by technology in which the FPGA core is available, the R4000 core (if it comes in HDL or as a hard core in a certain technology), as well as access to the tools and expertise for in-house design in different technologies. The simulated performance was scaled from 1.2um to 0.3um using constant voltage scaling, where delay is linearly proportional to minimum feature size. The resulting delay of the switch was in the order of 3ns for a switch with 256 configurations (thus the 8 address bits), though the fact that the switch circuitry ran at 333MHz suggests that this is the delay in the control circuitry rather than data passing through the switch. Since the cores are running with different clocks, a tag bit in the data serves as a flag for valid data, thereby allowing data driven operation rather centrally timed. For some reason, the R4000 used as an IP core was in 0.18um technology, but also ran at 280MHz.

The algorithms tested were image smoothing by pixel averaging, Viterbi decoding (with encoding and noise generation), and IIR filtering. Figure A.B.17 shows the results for the single R4000 baseline, aSOC designs, and the aSOC wire topology operating under packet-switched and arbitrated bus schemes. Not surprisingly, aSOC beats the single RISC by 6 to 10 times, depending on the algorithm and size of aSOC. The bus structure is 1 to 3 times slower, limited by interconnect bandwidth. Packet switched communication is slightly slower, but uses a greater proportion of the physical links, implying greater interconnect congestion. The actual packet processing is expected to slow it down further, as well as requiring significantly more logic for header processing and packet buffering/reassembly. It was acknowledged that packet processing was a questionable option for on-chip communications.

One discovered drawback was the deep switch configuration memory needed for reasonably complex traffic patterns, though the regularity of the data transfer in the testbench algorithms needed only 5 to 20 configurations. Future work includes a compiler to create configuration schedules from applications, devising a scheme to support run-time determined destinations for data, and pipelining of the switch control circuitry to approach 1GHz clock rates.

		Image Processing				IIR				Viterbi			
		Risc	Bus	Dyn.	aSOC	Risc	Bus	Dyn.	aSOC	Risc	Bus	Dyn.	aSOC
9 cores	Time (mS)	1072	332	151	133	114	25.0	22.4	21.6	29.7	6.4	6.0	5.5
	Link Util.	-	100%	11.1%	6.3%	-	100%	12.2%	5.5%	-	40%	2.4%	1.5%
	Used Links	-	1	9	9	-	1	7	8	-	1	7	6
16 cores	Time (mS)	1072	333	81.1	71.5	114	44.6	11.2	10.9	29.7	8.2	5.4	6.2
	Link Util.	-	100%	17.0%	10.6%	-	100%	24.4%	13.5%	-	84%	2.8%	1.2%
	Used Links	-	1	11	10	-	1	14	13	-	1	13	16

Figure B.17. aSOC results for testbench algorithms applied to the two cores compared with a single MIPS R4000, as well as the same cores with arbitrated bus and packet switch data communication [LST00].

Appendix C: Survey of CAD Efforts

This appendix reviews published efforts in mapping algorithmic specifications to reconfigurable hardware platforms. Compared to the mapping algorithms for coarse-grain platforms in Chapter 3, the approaches covered here target platforms that are more general-purpose or hypothetical. Also included are CAD efforts that target more commercially established platforms, such as FPGAs or board-level [re]configurable computers based on commercial FPGAs. Algorithms targeting more generic or hypothetical platforms naturally have greater freedom to define the mapping e.g. one algorithm presumes an system-on-chip (SOC) template (Section C.3.5 on page 262), and the design flow helps customize it for specific tasks.

Since coarse granularity brings with it more predetermined structure in the device, an impediment to its wider adoption is that the user must become familiar with the datapath and control structures in order to decide how best to map DSP algorithms; this must be alleviated by CAD. In addition to exploiting structure in the algorithms to optimize mapping, there is also the need for “blind” optimization, such as packing functionality into as few DPUs as possible. Being blind, this problem lends itself better to automation than the problem dependent structure mapping, which requires the creativeness of a live person.

The methodologies presented here are examples of functional specification and any presumed execution models, and can provide the starting points from which to conceptualize an EDA flow for coarse grain reconfigurable arrays. Since the back-end physical design phase for commercial FPGAs are somewhat mature, the innovation is at a higher level e.g. execution models to be realized, front end partitioning of large applications into kernels that time-share the array, kernel scheduling, languages that assist in hardware/software partitioning/verification, etc..

C.1 Input Specification and Typical Compilation Flows

The mapping problem is highly dependent on the method chosen to describe the algorithm to be implemented. Possible methods are:

- ◆ Sequential code e.g. C
- ◆ Sequential code with extensions for parallel sequences of operations
- ◆ Dataflow diagrams (DFGs)
- ◆ Control DFGs (CDFGs)
- ◆ Register transfer level description using HDLs
- ◆ Structural HDL

Figure C.1 shows typical flows associated with several input specifications and types of target platforms. Flow (a) targets a fine grain FPGA; this is very common in the literature. Specifications can be either sequential code, or one or more DFGs/CDFGs. In the case of sequential code, block extraction is required to translate code portions into [C]DFGs. These are then partitioned so that the pieces (hopefully) fit onto the FPGA when synthesized. Since synthesis can take a very long time, the search for a good partitioning is sped up by using estimates of resource utilization and latency/throughput to approximately evaluate candidate partitionings. These estimates are based not only on the computational/timing logic; they are also determined by the data buffering needed to forward data to the next temporal partition, the characteristics of the macros in the targeted library, and reconfiguration time overhead.

Some algorithms focus on the problem of scheduling kernels to maximize throughput based on delays for loading data, configuration, and kernel execution; these works assume that the kernels and their precedences are already determined. Commercial tools are usually used for synthesis in flow (a), followed by placement and routing (if appropriate) to see if the partitions can fit onto the platform, and if they run fast enough.

Flow (b) in Figure C.1 shows the conceptually reduced mapping problem when RTL code is mapped to target fine grain FPGAs. It consists of partitioning and scheduling. In cases where the operation model assumes pipeline-like execution, kernels run in a straight-line sequence in only one order; scheduling of the kernels is not an issue. Placement and routing of the kernels may still be required, depending on the level of abstraction being targeted by the algorithm.

Flow (c) shows the much less mature and defined process of mapping to coarse grain platforms. The input specification typically consists of a structural or behavioral HDL subset, or sequential code.

Input specifications at higher level of abstraction imply a greater mapping problem. Structural HDL is of the lowest abstraction, while all the other methods are behavioral. There can be a great

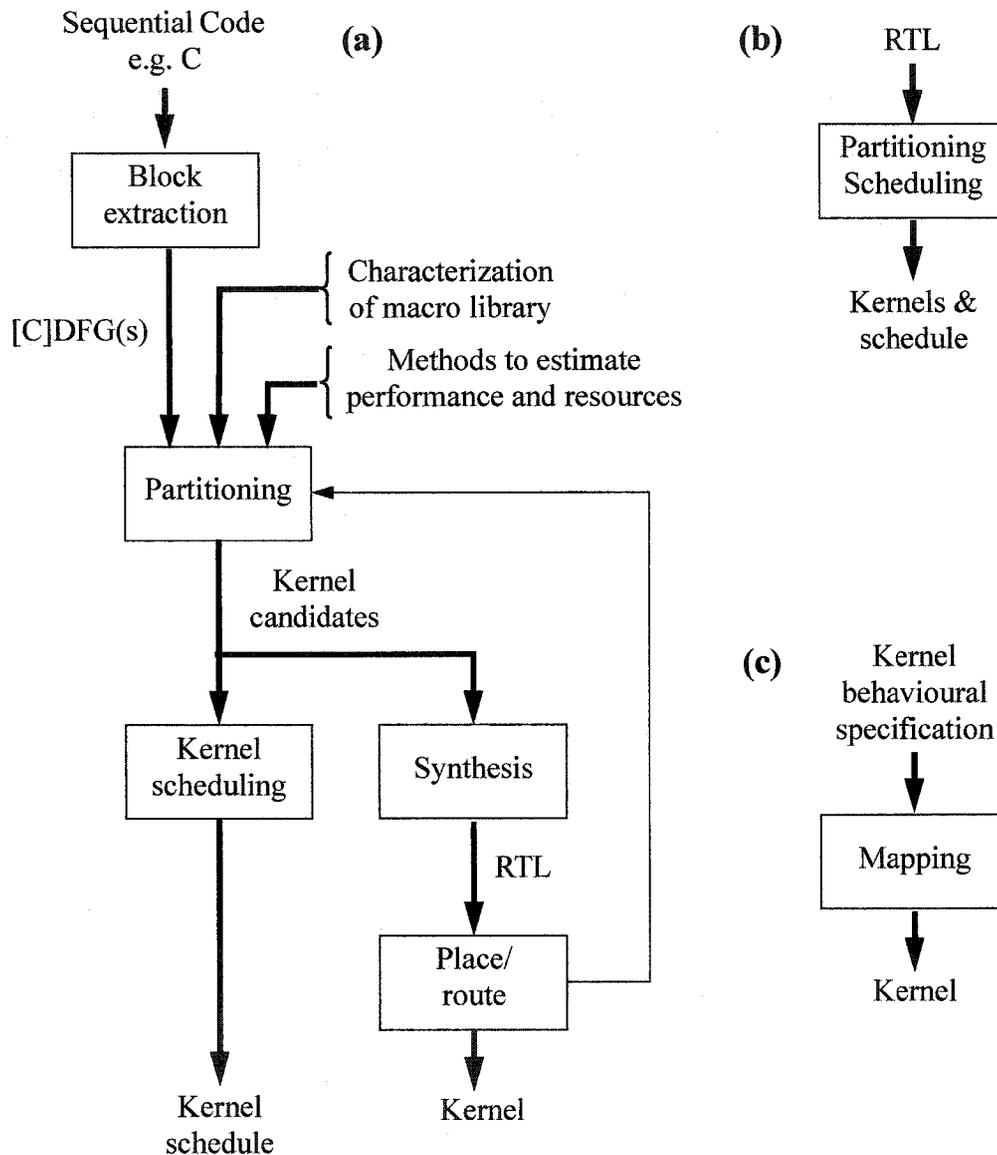


Figure C.1. Typical flows for mapping of functionality to kernels.

(a) Behavioral specifications at high level, with fine grain FPGA as target platform. (b) RTL level specifications, with fine grain FPGA as a platform. Behavioral specification of a kernel, with coarse grain reconfigurable platform as the target.

overlap in functionality between different specification styles e.g. CDFGs, procedural HDL, and C with or without parallel constructs.

Of the CAD methodologies that extend automation up to the software level, many search for structure in the source code after algorithm has been implemented in software. Candidates for hardware acceleration are tight loops of calculation.

DFGs are very intuitive for a pure datapath description but can be much clumsier than HDLs for specifying combined datapath/control information. This does not mean they are incapable of high levels of abstraction. We distinguish between two kinds of abstractions. If the designer is using large prebuilt macros/cells, s/he is working at a high level within the design hierarchy, even if blocks are being connected in a structural manner. This may be thought of as *hierarchical* abstraction. On the other hand, procedural HDL that describes functionality in a C-like syntax is abstract in the sense that the code does not reflect how the functionality is realized in hardware, even though the functionality described may be at a very low level in the design hierarchy. This kind of abstraction may be regarded as *semantic* abstraction. Since the design specification is removed from the implementation details, the designer can specify some kinds of functional behaviour more expediently, relying on tools to translate the design into hardware. In general, there is more than one way to fill in the “missing” implementation details; these details may be thought of as residing in the CAD tools, their assumptions, approaches, and algorithms, the cell/macro libraries, as well as any constraints provided by the designer.

The key implementation details that differ between C and structural or RTL-level HDL is the timing and resource binding. C does not maintain a notion of what hardware resources are available and which ones should be assigned to which operations. Aside from data dependencies inferred from the ordering of source code, C does not have constructs to specify exactly when functions should be performed in order to allow other functions to continue in a timely manner. In highly parallel processors and/or deeply pipelined processors, these detailed decisions in mapping to the processor resources are made by the compiler, and are inherently target-specific.

Many of the hardware platforms in Chapter 3.3 also deal with compiling C and HDL, including DISC, RaPiD, Pleiades, Garp, and Chimaera.

C.2 Verification

Beyond hardware/software partitioning by identifying kernel candidates, higher level support requires that hardware configuration and invocation be integrated into the software program that supervises kernel execution on the reconfigurable array. In order to functionally verify the partitioning and interaction of the hardware and software through co-simulation, the reconfigurable logic platform and its constituent components require behavioral models that operate in the same environment as the software. A kernel design that is represented in terms of these models assumes that timing is somehow met; therefore, the logic synthesis tools must be mature enough to enforce timing in the mappings to the reconfigurable logic.

Some of the approaches discussed next encompass the partitioning and co-design of software and hardware, sometimes erasing the distinction between the two e.g. Handel-C and JHDL discussed below. Most published efforts at compiling and mapping for reconfigurable logic are based on fine grain FPGA demonstrations, since they are more mature than coarse grain platforms.

C.3 Tools

C.3.1 DEFACTO

An academic effort known as DEFACTO [BDD+99] accepts high-level software code, performs analysis and optimization to partition functionality between “orchestration” software on a general-purpose host processor and configured hardware on a number of configurable computing units, and synthesizes the hardware (Figure C.2). These are goals at the moment (of their publication). DEFACTO is based on a the commonly used SUIF compiler exploration platform.

Details of DEFACTO matured in [DHP+03]. Annapolis Micro Systems’s Wildstar/PCI board was the target platform. Compiler optimizations include loop transformations and tiling. Methods are also presented to improve data access in memory and split the data into the different RAMS for the FPGA coprocessors.

C.3.2 Handel-C

Handel-C [Pag96] is an example of how mapping support is extended up to the software level by language/compiler modifications that assist in the automation of hardware/software partition-

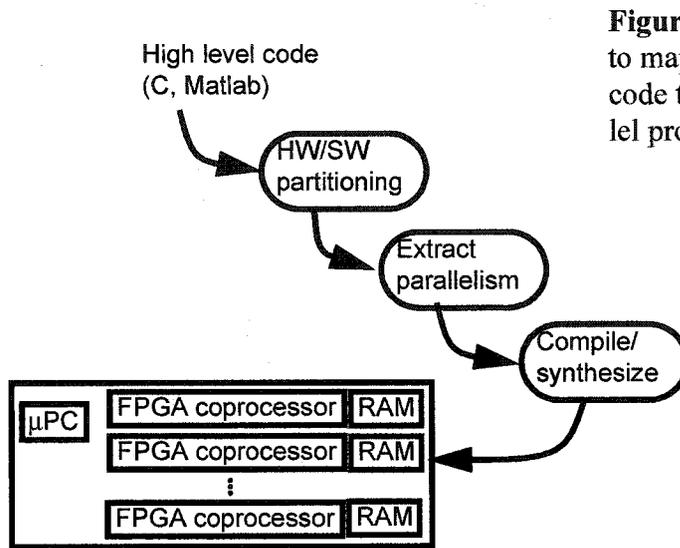


Figure C.2. Defacto flow to map high-level software code to configurable parallel processors.

ing. It is a common software/hardware language that originated from the former Oxford Hardware Compilation Research Group at Oxford University, based on the language “occam” for parallel processes. For the purposes of hardware/software integration, and in order to ease the burden of hardware knowledge for experts in algorithm and software, Handel-C is made to syntactically resemble C as much as possible. Occam extensions to C are provided to spawn parallel “instruction streams”, and to control duplication of logic for expression evaluation. To enable mappability from software, hardware support is restricted to high-level *procedural* description of *globally synchronous* logic; this makes most of the event-driven concepts in HDLs largely unnecessary, aside from the fact that all expression evaluation associated with a variable assignment occur in one clock cycle. This essentially reduces down to the execution of one sequential statement per clock (per parallelized instruction stream). Thus, Handel-C code reflects functional behaviour in a manner similar to that of a single-instruction-per-cycle microprocessor stepping through procedural code. The restriction to globally synchronous code is not overly restrictive for reconfigurable hardware because such platforms are physically designed to be globally synchronous anyway. However, the complexity of expressions to be evaluated in one clock cycle is limited by combinational logic delay if the clock is to run at a particular speed. The task of decomposing a complex expression for pipelining is moved up to the C source code level by introducing intermediate variables to break up a complex expression into a series of simpler ones that evaluate sequentially in time.

C.3.2.1 Hardware Inference

The Handel-C compiler, “MOAT” [Cla97], infers hardware from the Handel-C code and performs technology-independent optimization, such as elimination of common subexpressions. This is critical because all right-hand-side operators instantiate logic; the tools do not use muxes and registers to time share hardware for a binary *operator* that occurs in more than one expression/sub-expression. However, if the *operands* are the same in both occurrences of the operator, a common subexpression results, and the logic for it is shared rather than duplicated. Furthermore, if a *variable* is assigned two different expressions in the code at two different places, the two pieces of combinational logic for those expressions are muxed into the register for that variable, and control logic ensures that the variable/register receives the right input at the right time to realize the behaviour of the imaginary one-instruction-per-clock processor discussed above (see Figure C.3). This simple association between hardware and software allows the hardware behaviour to closely mimic the Handel-C code and makes it easy for the user to be aware of what the code synthesizes to. It is interesting to note that muxing different combinational logic blocks into one register is the reverse of the conventional time-sharing of resource-intense datapath operators by muxing different registers into the inputs of a shared operator block.

C.3.2.2 Timing Control

In order to realize the sequencing of the operations for a series of procedural statements, it is necessary to pulse the enable pins of the registers corresponding to the variables receiving the assignment in each statement. A cascaded set of control flip-flops, one per statement, ripples an enable pulse through the flip-flops in a one-hot fashion to ensure the data register for only one assignment statement is active in any clock cycle (Figure C.4). In this simplest case, the each control flip-flop associated with a data register represents a control circuit that gates the registered output of the assignment statement, and the enable pulse serves as a token to be passed from one statement’s control circuit to the next. In more complicated structures, such as “while” loops, a more complicated control circuit can trap the token to keep the circuit active until the loop exit condition is met, before passing it to the next circuit.

When the token output line pulses high (for one clock cycle only), it does more than just signal the next circuit to start; it also serves as a signal to the preceding circuit that the current circuit is able to accept the next piece(s) of data for further processing. Therefore, the token input line acts as a “start” signal for the circuit while the token output line serves as a “finish” signal. From this

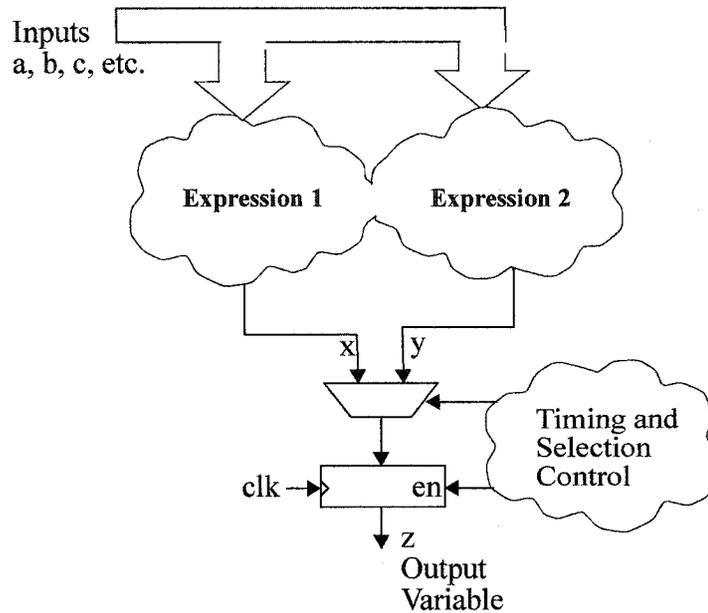


Figure C.3. Handel-C logic that results when a variable is assigned different expressions in different places in a linear sequence of assignments. Common subexpressions may share logic.

operational description, we conclude that in order to avoid overwriting data that has yet to be processed by the next circuit, the current circuit should not pass the token until the next circuit has pulsed its **finish** signal i.e. the next circuit should not be started again before finishing. Not only would doing so corrupt the unfinished operation of the next circuit, but the preceding circuit will think that the current circuit is ready to fire again, which would overwrite the current circuit's registered outputs before they are processed. The compiler ensures that all control circuits observe proper handshaking by never passing the token until the next control circuit signals a finish. Figure C.5 shows how control splits into a parallel set of tokens for parallelized processes.

If the operational description is accurate, we note some caveats. This scheme can potentially limit the throughput of any datapath to half if its maximum realizable rate. This is because the token output line must go high *before* the token input line can be driven high again by the preceding control circuit, requiring at least one clock cycle when the token input line is low. In the worse case, the token input line alternates between high and low, meaning that the circuit is inactive 50% of the time. Presumably, the negative impact can be minimized by having most of the high-speed

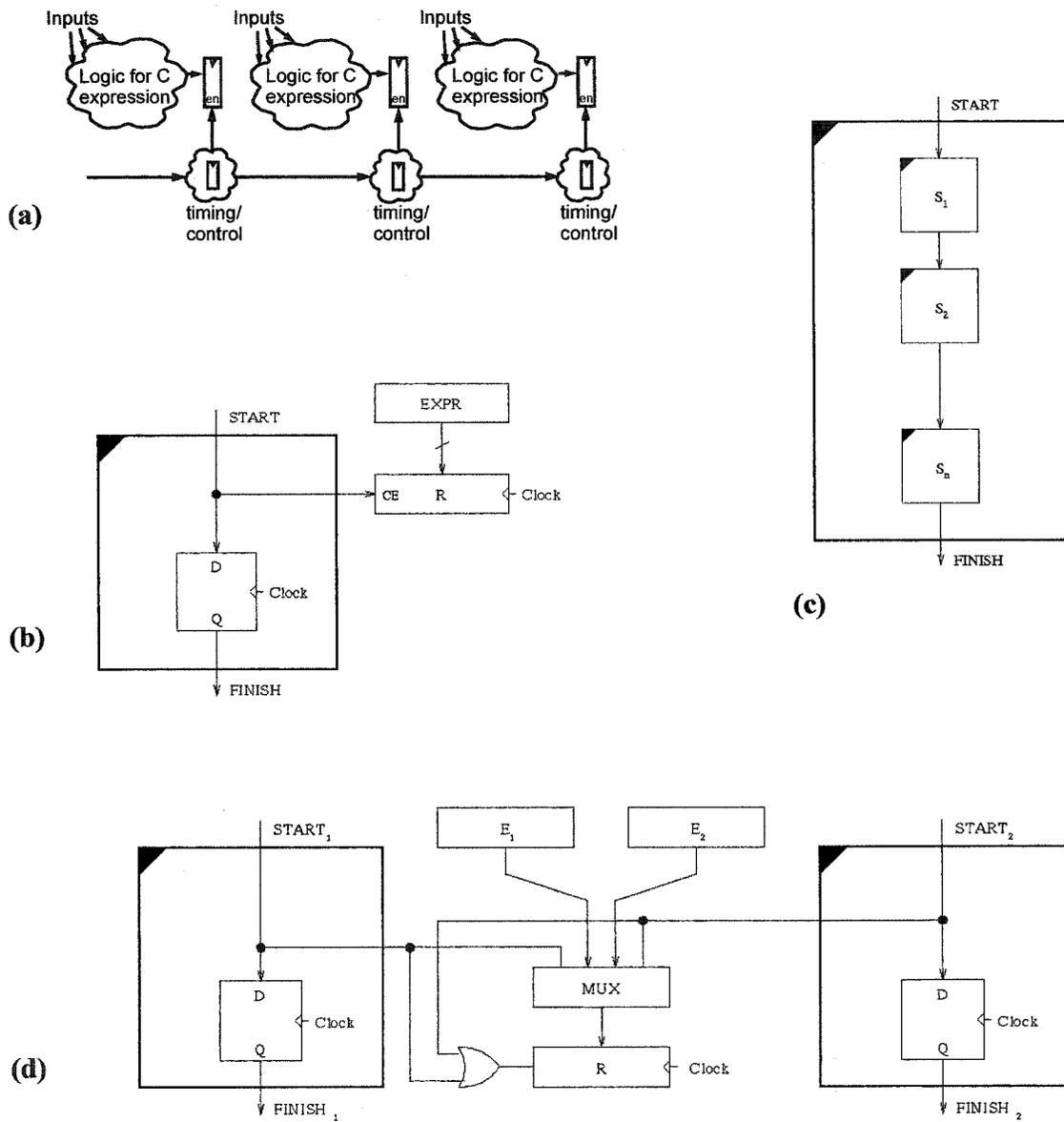


Figure C.4. Handel-C's token passing handshake for timing control.

(a) General idea.

(b) Handshaking for a simple assignment [Pag96].

(b) A sequence of assignments to happen in succession [Pag96].

(c) Two assignments to the same variable at different steps in a sequence [Pag96].

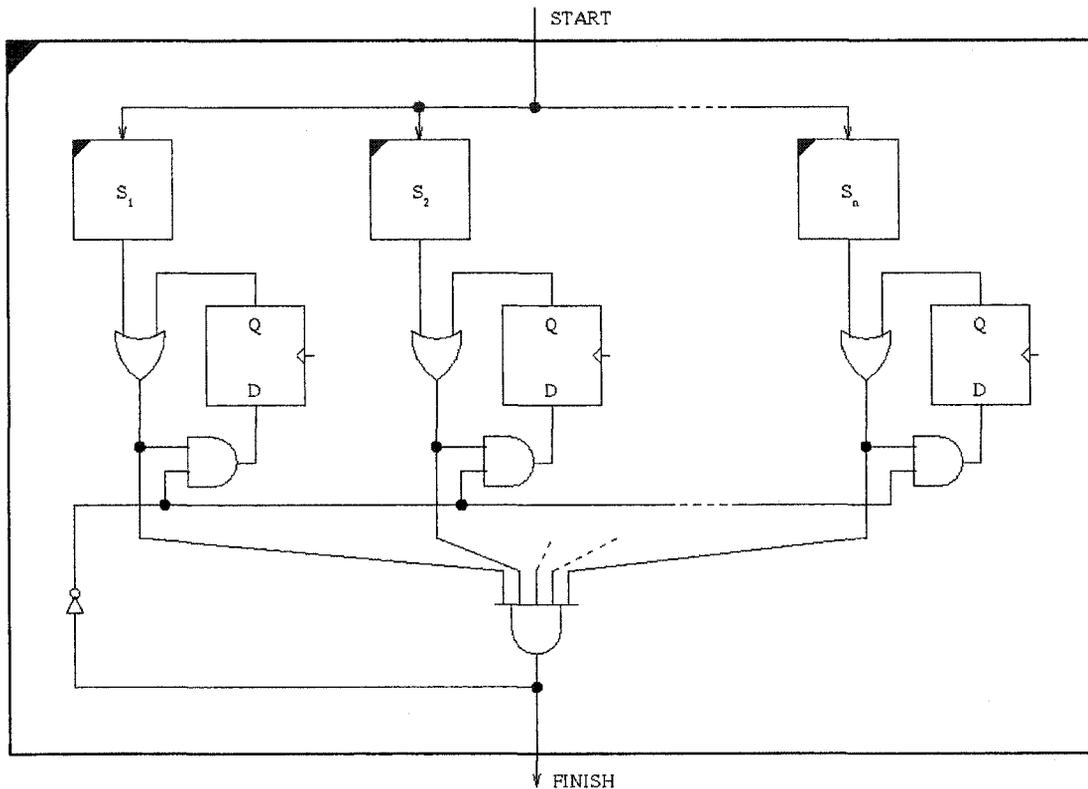


Figure C.5. Parallel sequences of Handel-C code require parallel token paths, and logic to wait for all sequences to complete [Pag96].

operation occur within a loop whose body executes in one cycle. Since the token is trapped within the loop's control circuit, the loop is continuously enabled and can fire at every clock cycle. This appears to be how a pipeline is realized, by having a series of parallelized one-cycle operations within a loop, where any one parallelized statement operates on the result of another parallel statement from the previous iteration. Any more than one assignment statement per parallelized instruction stream would require handshaking circuits operating in every loop cycle, thereby incurring the 50% maximum throughput penalty.

C.3.2.3 Non-Partial Configurability

The compiler creates a technology independent netlist of generic digital logic components, which can also be simulated by the compiler. The netlist is then massaged by back end tools in

preparation for synthesis to an ASIC or to different FPGA platforms via vendor-specific synthesizers. From this, we can see that Handel-C does not exploit run-time configurability, or even partial configurability. Rather, it is touted as an example of a functional specification language for hardware/software. In reality, the published examples show it being used as a high-level procedural and behavioural specification language that can map to either software or hardware; there is not much demonstration of the actual hardware/software partitioning and cooperation, automated or manually planned, aside from software initiated loading of the FPGA.

C.3.2.4 Handel-C Versus HDLs

It is claimed that the user need no knowledge of hardware to compose code. There is no elaboration on how such code can synthesize to hardware that is significantly more efficient than a conventional HDL synthesizer applied to a procedural HDL description of a synchronous digital circuit. Indeed, the claim is that hardware optimization is not the object; the objective is rapid acceleration of software-like code. The messiness of hardware detail is removed from the user by judiciously disallowing the freedoms that make hardware design complicated, such as latches and any asynchronous device, as well as by imposing/providing predetermined control and datapath structures. For example, the implementation of adders and multipliers appears to be out of the user's hands, and the token passing control structure is automatically implemented. Aside from the latter, it might be argued that HDL code used with a similarly restricted datapath library can be as simple as Handel-C, with a similar price paid in terms of area or speed. The contribution of Handel-C seems to be the fact that this trade-off point was chosen, the timing and control was defined and automated, and the development of the compiler.

C.3.2.5 Commercialization and Secrecy

Among other related publications, the Oxford group's publication website [Oxf] at one time contained a cleansed version of a master's thesis project attempt to map a Java virtual machine to FPGAs in order to avoid custom silicon implementation of the evolving Java standard. (All persons' names were removed, public access to documents has since been disabled, but the project report itself is still available via original author's website [Cla97]). Many further claims are made about hardware-speed video game demonstrations compiled from pure software with very little ramp up time to learn the system. Handel-C has recently been commercialized under the U.K. based Embedded Solutions Limited, which is why the group no longer exists at Oxford University.

C.3.3 JHDL

Brigham Young University's JHDL [BH98] targets a lower level of abstraction for providing a common environment for software and hardware specification and execution based on Java. (JHDL is known both as Java HDL and Just-another HDL). Not only does the standardization of Java ensure portability, it permits leveraging of the built-in GUI features. The object oriented nature allows an elaborate class to be developed for specifying hardware without formally departing from the all-Java all-software language i.e. a simplified HDL is built on top of Java. As in Handel-C, the synchronously clocked nature of the platform ensures very simple rules for determining the timing of events. However, JHDL differs from behavioural and procedural code of Handel-C in that the objects represent actual hardware, and the code represents structural HDL code at a low level i.e. like a netlist. Even though it uses pure Java software without extensions, it actually specifies hardware at a lower level and requires the user to understand hardware design. Mention is made of efforts to develop behavioural compilation.

The JHDL compiler further differs from Handel-C in that Handel-C requires all kernels for an application to be on the FPGA, whereas JHDL was designed to exploit runtime partial reconfigurability by overwriting finished kernels in an FPGA with new kernels. The kernel netlists are meant to be generated from the structural description in the JHDL code, though at this time, the JHDL primitives are only simulation models of gates. The gates of the circuit still have to be manually entered into a schematic capture tool before being synthesized into configuration bits (only the Xilinx 6200 has been mentioned so far). When the hardware is being simulated, a Java test-bench object takes care of clocking all instantiated hardware objects and updating signals. Kernel objects can be instantiated and destroyed as needed. When actual hardware is being used, the process that executes the Java code encounters object instantiation of software kernel models and triggers a download of the configuration bits for the kernel into the FPGA instead (Figure C.6). Conversely, the destructing of software kernel models in the code triggers the freeing up of reconfigurable resources occupied by those kernels. Partial configuration of kernels requires that the primitives be hand placed (for now) to avoid clobbering other kernels, which probably also means that the kernel loading order must be fully anticipated. Though JHDL is described as a hardware/software co-simulation and co-execution environment, only hardware invocation was illustrated in the paper. The exact details of success with test algorithms are vague.

JHDL differs from Handel-C in that it structurally describes an interconnection of gates at a low level and builds up more complicated circuits by structurally interconnecting them; the user com-

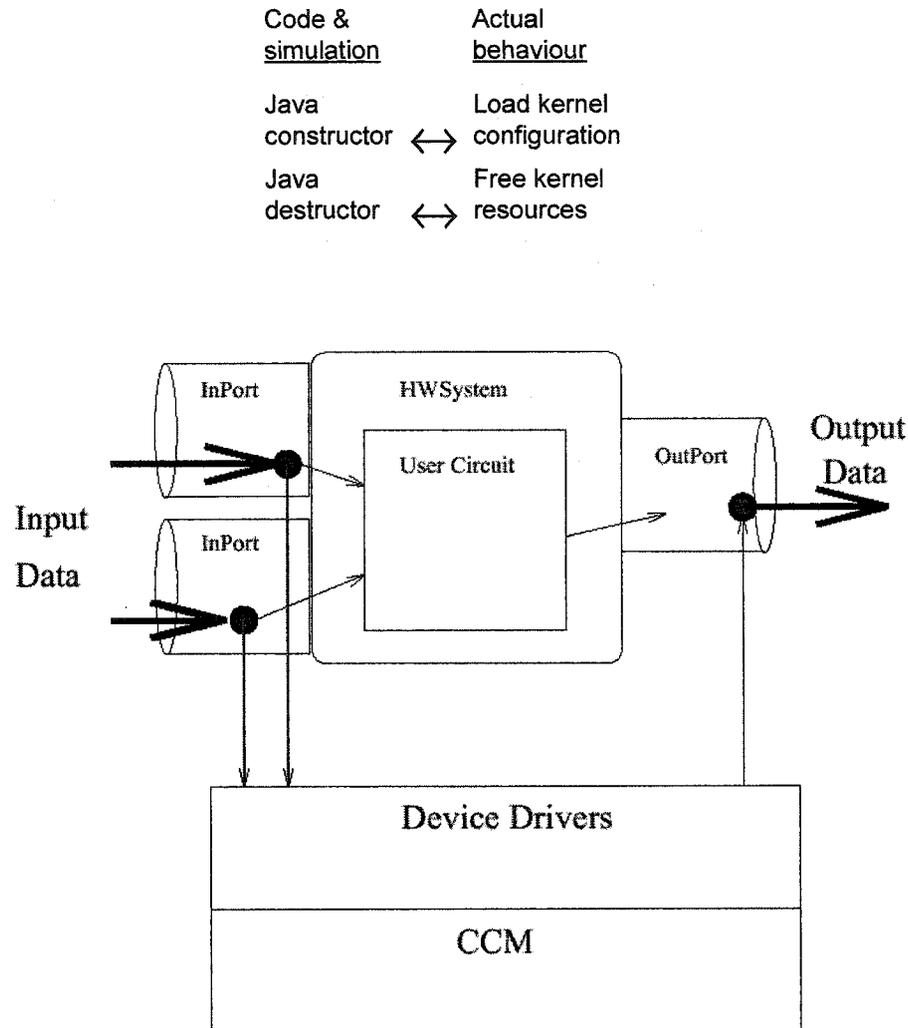


Figure C.6. JHDL's binding of Java constructors/destructors with the loading/freeing of kernels to/from the Xilinx FPGA cache on Virtual Computer Corporation's configurable computing machine (CCM). Schematic from [BH98].

poses a circuit whose design requires functional verification. (The tools could function flawlessly, but the user-designed hardware might be flawed.) Thus, JHDL has a hardware simulator that functions like a cycle-based HDL simulator. In Handel-C, the user only specifies high-level behavioural math and lets the compiler design the datapath operators, as well as timing and control hardware. Consequently, the design flow described in the literature and documentation hardly

mentions HDL simulation, though the capability apparently exists. Instead, much of the Handel-C effort seems to be directed at the higher level transformation of software code structures to a form that readily maps to hardware, and formally verifying this based on concepts of “communicating sequential processes” (CSP) [Hoa85].

C.3.4 Streams-C

Streams-C [GSAK00] is a C based language with constructs added to support FPGA-based parallel computation. The C level source code is processed by SUIF and all functionality beyond C takes the form of C callable functions. Like Handel-C, the parallel programming model follows the tenets of CSP. It claims to target an intermediate level of abstraction between JHDL and Handel-C. The syntax is built on an earlier C-extension known as Napa C, targeting a RISC platform with a single FPGA coprocessor. Streams-C targets a similar platform with multiple FPGA coprocessors. The programmer/designer parallelizes his/her algorithm onto the processors and arranges processor communications as in conventional parallel machines, but does not have to deal with cycle-by-cycle activity of gates. Pragmas permit the specification of which processor runs certain code, as well as the storage location and word width of variables. Functionality that is destined for the FPGAs is mapped to a library of modules written in heavily parameterized RTL-level VHDL (Figure).

The specific platform that has been used is Annapolis Micro Systems’s Wildforce configurable parallel processor board, though supposedly, they can also target Los Alamos National Laboratory’s RCA-2 boards and SLAAC. (RCA-2 is a board level reconfigurable computing array based on commercial FPGAs [RCA]. SLAAC is an open systems standard developed at the Information Sciences Institute at University of Southern California for high performance distributed computing using off-the-shelf PC’s and configurable computing accelerators [CSP98], with a project from Virginia Tech [Tow] given as a representative system.) The higher level code abstraction allows for functional testing using faster high-level multithreaded simulation at a software level compared to HDL simulation.

The sample application consisted of image contrast stretching. The Streams-C design took several days compared to a month by an experienced designer using conventional VHDL (an order of magnitude more time). However, the Streams-C design occupied 3x the area and ran at half the clock rate. Thus, there is a clear trade-off in area and speed for quick design due to automation, as

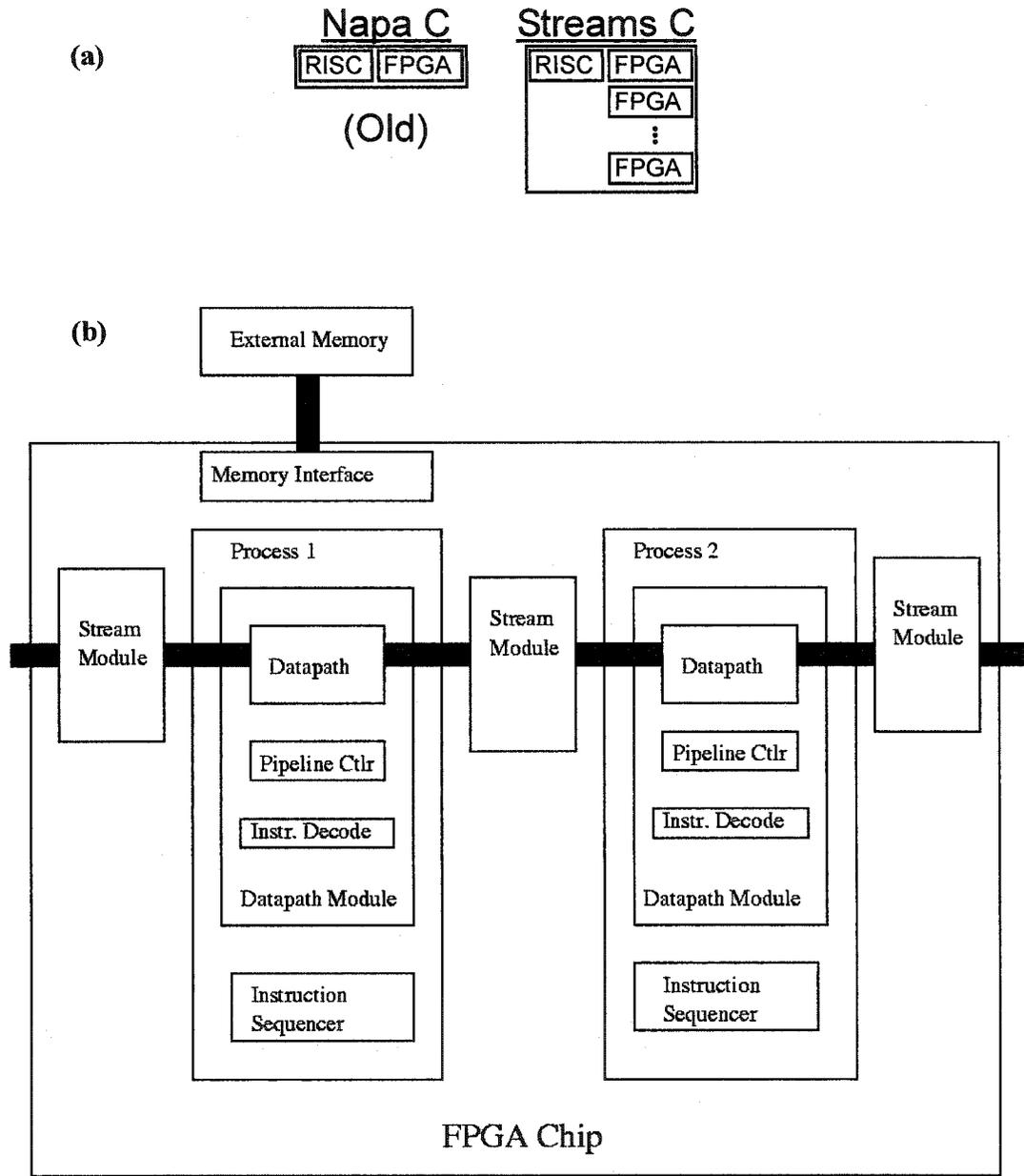


Figure C.7. Streams-C architecture.

(a) Streams-C built on Napa C. (b) Framework for the FPGA logic (bottom) [GSAK00].

is generally true for tool assisted digital design. No results such as images were shown, so it is not clear if working results were obtained from the Streams-C design.

C.3.5 C/Fortran Compilation to RTL for a Custom MIMD

[BRM+99] is an effort from the Laboratory for Computer Science at MIT demonstrating C/Fortran compilation for vector-based arithmetic to RTL, based on MIMD-like segmentation. Vectors are segmented into small memories distributed across “tiles” (Figure C.8), each having its own custom compiled data operators, interconnect muxes, FSM controller, and a program counter. Pipelined wires are used to time-multiplex signals for inter-tile communication to trade-off functional density for throughput. The granularity of the wire for time multiplexing is not at the pipeline segment level i.e. the entire pipelined series of wire segments must be time multiplexed, not each segment independently. The results is like a custom MIMD with local memory, very specific to the application for which the high-level code is written. As expected, the performance exceeds that of conventional processor arrays because each MIMD processor is custom compiled for the application. It is claimed that reconfigurable logic platforms can be targeted.

C.4 Algorithms and Methodologies

In the following, we survey a few algorithms in detail, and briefly review a number of others.

C.4.1 HySAM

In [BP98], a system is introduced to capture key kernel parameters and allow the kernels to be combined in various ways to form larger kernels. Based on this calculus, efficient ways were proposed to unroll loops, taking into account the overhead time for reconfiguration.

A more concrete demonstration of this approach was presented in [BP99]. In this work, dynamic reconfiguration was used for a technique referred to as *dynamic precision adjustment*. A hypothetical kernel represented by a loop in software was analyzed at an abstract level to determine how best to vary the precision of the calculations to minimize overall execution time. Three competing factors were (1) to minimize precision so as to allow faster operation, (2) to always have sufficient precision to accommodate bit width expansion in the recursive calculations, and (3) to minimize the number of reconfigurations required in changing precisions. The nomenclature of HySAM was used to set up a dynamic programming optimization of when to vary the precision, and by how much.

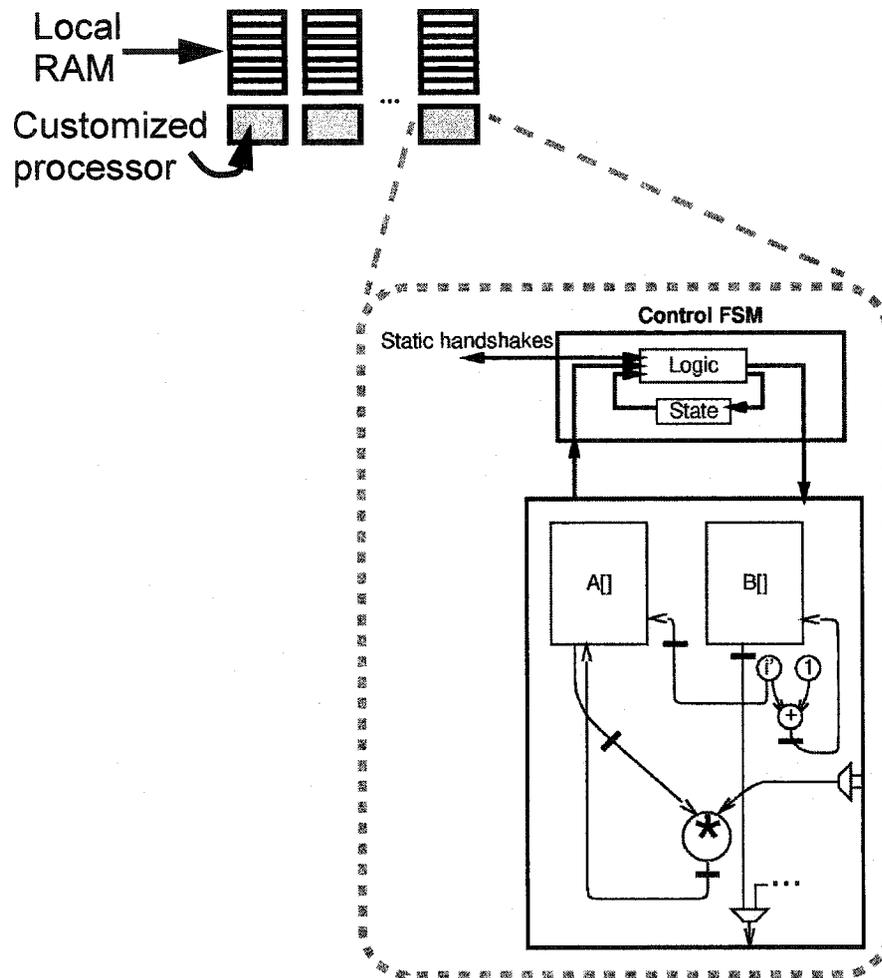


Figure C.8. Target for the compilation of high-level software to MIMD RTL for an ASIC flow. Processor detail from

Because the problem was modelled at an abstract level, some issues not explicitly addressed include whether to flush the pipeline between precision steps, as well as where to store intermediate data for consecutive kernels. We expect the problem of data forwarding to be complicated by the fact that consecutive kernels may have different number of pipeline stages, and different floor-plans.

C.4.2 Partitioning of netlist for FPGA

In [Ste98], a netlist of LUTs, potentially with registered outputs, is partitioned into a sequence of FPGA configurations to be executed in succession. The assumed circuit model was that of a Mealy machine. The target platform was a proposed multicontext FPGA architecture i.e. one which cached up to 8 configurations, thus allowing configurations to change instantaneously.

For the sake of clarity in this description, we will distinguish between the *act of partitioning*, which yields a set of time-sequenced pieces of the original netlist, versus one of the pieces of the set, which we call a *partitioning segment*. In [Ste98], *microcycles* refer to the time slots for the a partitioning segments. The entire sequence of configurations form a single Mealy machine clock cycle, referred to as a *user cycle*. LUTs were assigned to partition segments by an algorithm based on list scheduling.

State registers may reside in any segment. [Ste98] works out precedence-preserving rules for the partitioning to maintain functional equivalence with the underlying Mealy machine. In particular, registered LUT outputs must be handled differently from pure combinational outputs; it was established that all combinational logic driven by registered outputs were part of the next user cycle, and had to be placed in either the same partition segment as the register, or a preceding segment.

[LW98] takes a different approach to the same partitioning problem. Rather than treating the problem as one of HLS scheduling, the LUT netlist is treated as a directed hypergraph to be partitioned. The cost of the edges that are cut by the partitioning represents the cost of buffering any data that is to be forwarded to subsequent microcycles.

The graph is partitioned into a linear sequence of segments by recursive bipartitioning. Each bipartitioning is based on a min-cut/max-flow formulation to minimize the nets that are cut. The partitioning is then iteratively refined to balance the resources among partition segments. It is assumed that the resources needed for a partitioning segment is readily determined when the partitioning is performed or adjusted.

[LW98] introduces modifications of the graph partitioning problem to handle precedence. The precedence information is contained in the directed hyperedges in the graph. The partitioning uses unidirectional cuts to enforce this precedence. Where a cut intersects a directed edge, the cost of an directed edge is considered only if it crosses the cut in a specific direction. For simple nonhyperedges, dummy directed edges of infinite cost is added in the opposite direction to prevent the nodes from being put into partition segments that violate the precedence.

For unregistered combinational outputs, a cut of a directed hyperedge is forced to take place before its forking point in order to minimize the number of registers needed for data forwarding. This is accomplished by introducing a dummy node at the forking point; simple directed edges of infinite cost then connect the dummy node to each of the destinations of the original hyperedge. This prevents cuts from crossing the edges after the forking point. The above precedence rule for *registered* LUT outputs was also enforced by a similar use of auxiliary edges of infinite cost.

[MY03] refines some of the semantics of the graph transformations in [LW98], as well as introducing a method of duplicating graph nodes among partition segments to reduce the amount of data forwarded between segments.

C.4.3 Compilation of C Targeting Coarse Grain Array

In [HM02], kernel mapping targets a hypothetical VLIW-like array of fully connected function units (FUs). The array acts like an acceleration coprocessor to a simple main processor. The problem is simplified in that there is no presumption of a fixed number of each type of FU. The target architecture does not have locally distributed memories. Fast reconfiguration is assumed to be realized using configuration caching. Results of their simulated mapping take into account the overhead from reconfiguration and transfer of control between the main and reconfigurable coprocessor.

The input is taken to be C-coded loops. The DFG of the loop body is mapped to a pipelined datapath. Isomorphism is assumed between the operations in the code block and the FUs. If the code contains conditional branches, a graph and a circuit are created from each branch; a mux then selects the proper output according to the branch condition. Multiple graphs are overlaid to merge like FUs.

The critical path was identified in the control logic. It was observed that coarse grain platforms are, in general, capable of higher clock rates than fine grain platforms. We believe that this is inevitable, since the datapath logic in coarse grain units can be highly optimized compared to a macro built from fine grain resources. We also note that the clock rate limitation of fine grain arrays can manifest itself as a routability problem, since it is often possible to find a routing for an FPGA synthesis, even though it doesn't meet timing.

The test case presented was that of an MPEG 2 video coder, and a GSM coder. With the mapping of these kernels to the hypothetical device, execution was several times faster than for a sin-

gle issue processor with speculative processing. This improvement was found to be equivalent to that of a very large VLIW with advanced compilation techniques.

The speed advantage from using the reconfigurable array was found to recruit trivial amount of logic resources. This suggested that much more parallelization and speedup was possible by replicating logic. For this scenario, the memory bandwidth was expected to be a bottleneck. Note that, as mentioned, the triviality of resource usage seems to depend partly on the fact that the different types of resources were not fixed prior to mapping. We caution that, for actual device architectures, the limits for certain resource will be reached before others.

C.4.4 Other Mapping Algorithms

[Bon01] proposes a method of mapping loops with inter-iteration dependencies to the Chameleon coarse grain platform [Cha01a,TA00]. Normally, the loop dependencies prevent pipelining of the iterations, since each iteration must wait several clock cycles for certain results from the previous iteration. The solution, *data context switching*, was applied to an IIR kernel. In this basic form, the method amounts to interleaving the IIR filtering of several different data streams, thus keeping the pipeline filled.

[PV99] performs an end-to-end compile of behavioral specifications, targeting the Xilinx-based Wildforce reconfigurable computing board [Ann]; this platform uses XC4000 series FPGAs for processing elements and switch boxes. The reconfiguration time is taken to be much greater than the kernel latency. [PV99] indicates that the behavioural input can be either HDL or DFGs, but the focus seems to be on DFGs. A DFG is temporally partitioned into a linear sequence of kernels. Overall latency is minimized by minimizing both the number of kernels and the latency of each kernel. The partitioning is accomplished by a variation of force directed list scheduling [PK89] for resource constrained scheduling. An HLS estimator approximates the cost of a kernel based on characteristics of library macros. Design space exploration includes varying the combination of macros used and trying different proportions of logic versus interconnect.

[KVG099] partitions a DFG that represents the body of a loop. The constraints are the application's throughput requirements and the resources limits of the target platform. As in [PV99], it is demonstrated using the Wildforce platform, the reconfiguration is taken to be slow (milliseconds), and its presynthesis partitioning requires estimators of delay and resource usage based on macro library information. Commercial tools are used for the back-end synthesis. In contrast to [PV99], the partitioning problem is formulated as an integer linear programming (ILP) problem with tech-

niques to linearize nonlinear constraint equations. To reduce the frequency of slow reconfigurations, loop fission is used to run a portion of the loop body for many iterations before changing kernels i.e. kernel-centric operation. Memory limitations to this method are discussed.

[BP00] is an example of exploiting partial reconfiguration to minimize the configuration overhead of devices with slow configuration (a fine grain FPGA, in this case). The method applies to pipelined loop bodies, without inter-iteration dependencies. [BP00] shows the loop bodies specified as DFGs. In a manner similar to the kernel-centric loop fusion of [KVG099], the pipeline stages for loop body are segmented into a linear sequence of kernels, each to be applied to a large number of inputs before reconfiguring. Data forwarding between kernels is done via external memory. To further reduce configuration overhead, FPGA macros for operations that are used in consecutively loaded configurations are placed in the same positions across configurations to reduce the configuration bit stream. Though the details are only lightly alluded to, this is equated to permuting intrakernel pipeline stages to obtain maximum co-location of like macros across kernels. There appears to be an implicit assumption that the target platform segregates resources column-wise, so that macros occupy columns of resources of uniform width, regardless of the different resource requirements of the different macros. Column-wise partial reconfigurability must also be supported. The methodology was demonstrated using the Xilinx Virtex FPGA.

[SGV01] is another high level analysis of partitioning [C]DFGs, though it is spatial partitioning for physically separate FPGAs, rather than temporal partitioning. Resource estimation formulas incorporate area figures for library macros. While not directly applicable to time multiplexed kernels, many of the theoretical aspects of the partitioning problem are similar. There are no details at the level of macro selection, but there are allusions to scheduling at a low level.

[KH03] is a theoretical analysis of mapping nested loops to systolic arrays on LUT-clustered FPGAs.

[Car01] presents a heuristic temporal partitioning algorithm for DFGs to minimize the number of kernels and their latencies.

C.5 Conclusions/Summary

Algorithmic mapping to reconfigurable platforms involve exploiting problem-specific structure, which requires human cognition, as well as automated synthesis. Extending compilation up into software involves semantic abstraction rather than hierarchical abstraction, the latter being

obtainable from coarse grain platforms or by using macro libraries in FPGAs. Semantic abstraction requires a framework to fill in details about timing and hardware resource binding.

Approaches to this may build on compiler technologies for VLIW or multiprocessors. Hardware/software integration requires tools and models for functional co-simulation as well as timing verification at the hardware level.

The major portion of this appendix surveyed academic efforts representing compilers that target hardware platforms. DEFACTO is a system level plan to automate hardware/software partitioning and compilation. Handel-C provides an example of how timing and control can be handled in mapping C to hardware. JHDL is a kernel caching system that exemplifies how structural hardware specification, compilation, and interfacing/control can be realized in a Java environment to facilitate integration with software. Streams-C supports intermediate abstraction in compiling C to an FPGA-based parallel processor by using a library of parameterized kernels. A final example is shown which compiles C/Fortran into a customized MIMD, with prospects of targeting reconfigurable platforms.

Algorithms for time-multiplexing kernels onto reconfigurable platforms were reviewed. The bulk of the work targets fine grain FPGAs with slow reconfiguration. Most of these efforts dealt with temporal partitioning into kernels and scheduling of the kernels. Most also rely on estimations of speed and resources from macro libraries, as well as empirical formulae for interconnect overhead. Commercial tools are used where synthesis is actually performed. Kernel caching was proposed as architectural novelty to avoid the reconfiguration overhead.

Appendix D: GA Details

D.1 GA Convergence, FFT Kernel Design Case

D.1.1 Simple DPU Ordering Problem

D.1.1.1 RKR: Searching for Good Search Conditions

The initial attempt at RKR for the simple DPU ordering problem (Section 6.1.7.2) did not have the flawless convergence of Figure 6.23. Degeneracy in gene values was suspected. We describe the investigations that lead to Figure 6.23 because it provides insight into how the dynamics of the search can be affected by different aspects of the GA.

The first measure to address the suspected genetic degeneration was to adopt a less greedy replacement policy; all unique kids (minus clones and twins) would unconditionally replace the worst population members. Figure D.1 captures the search statistics for a representative nonconvergent case, in which the search was allowed to continue well after the population seemed to prematurely converge. The convergence behaviour of the search can be inferred from this data. Mutation was being relied upon to eventually break out of the local minimum.

The gene diversity (Figure D.1(h,m)) was calculated as follows. For each gene in the chromosome, the number of unique values in the population was determined (Figure D.2). This count was then averaged over all the genes in the chromosome.

The rejection of kids (Figure D.1(c,i)) was due to clones or twins rather than poor cost, since unique kids were accepted unconditionally. For RKR, the clones or twins are extremely unlikely to result from mutation, since the random real mutation of a gene is unlikely to duplicate an existing gene value. We attribute the rejection rate to duplicates occurring in the offspring. For example, if the parents are identical except for 1 gene, then the offspring will be clones of the parents. Cloning is also possible if the parents differ by more than 1 gene, but only if the differing genes are *all* either swapped or not swapped.

From Figure D.1, a number of insights can be drawn about of the search.

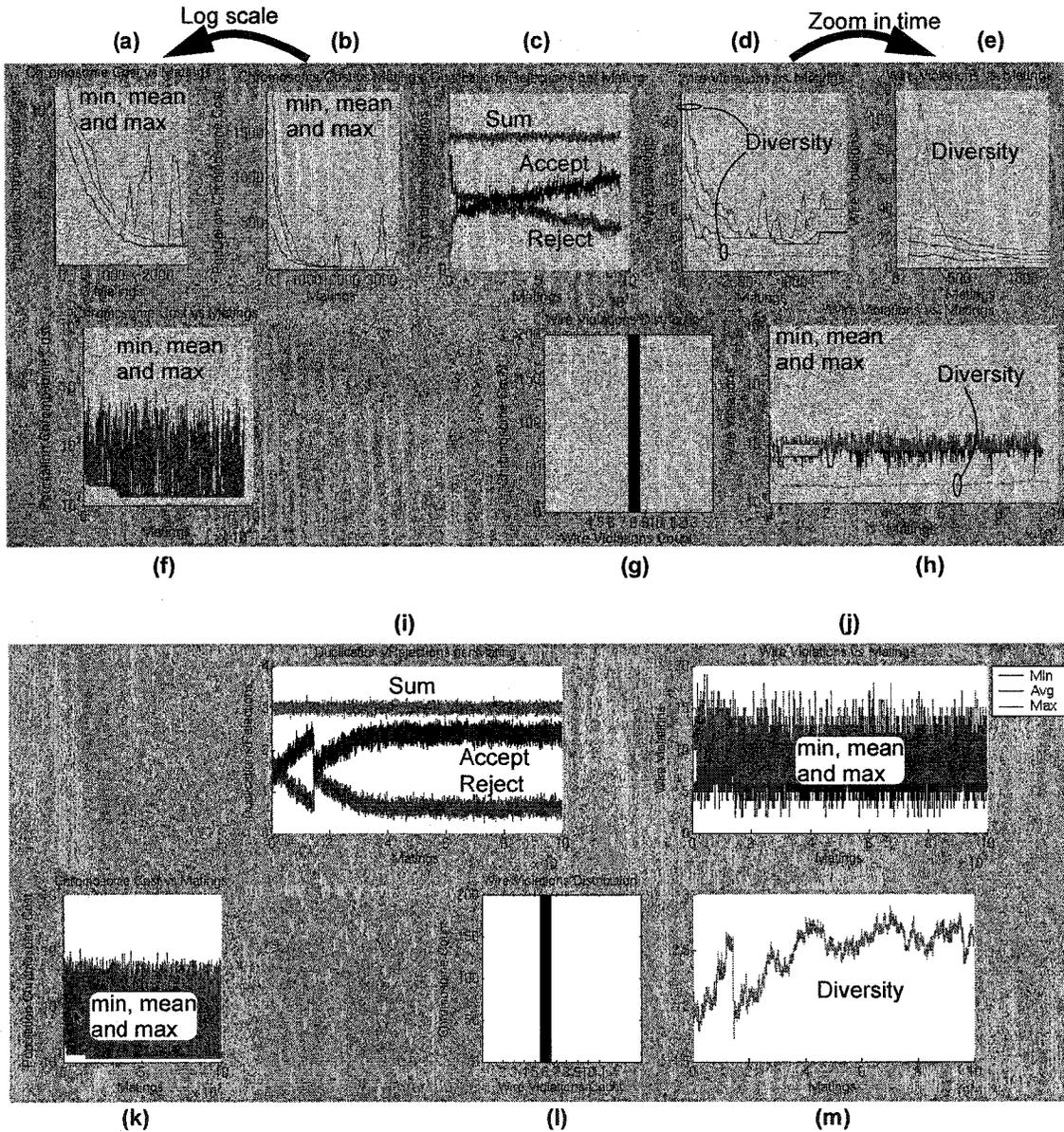


Figure D.1. Captured statistics from a nonconvergent case of initial RKR implementation, 50% mutation rate applied to *each* offspring i.e. 2 offspring and up to 2 mutants per generation.

The bottom panel captures the long term behaviour (10 times that of the top panel). Diversity is plotted with wire violations simply because they share the same scale. All data is sampled every 100 matings. The following data is *averaged* over the 100-mating subsample interval: Diversity (d,e,h,m) and acceptance of kids (c,i). All plots of the progress of wire violations and chromosome cost show population min, mean, and max (a,b,d,e,f,h,j,k); the mean is typically coincident with (and hidden by) min. The distribution of wire violations (g,l) are for the last generation.

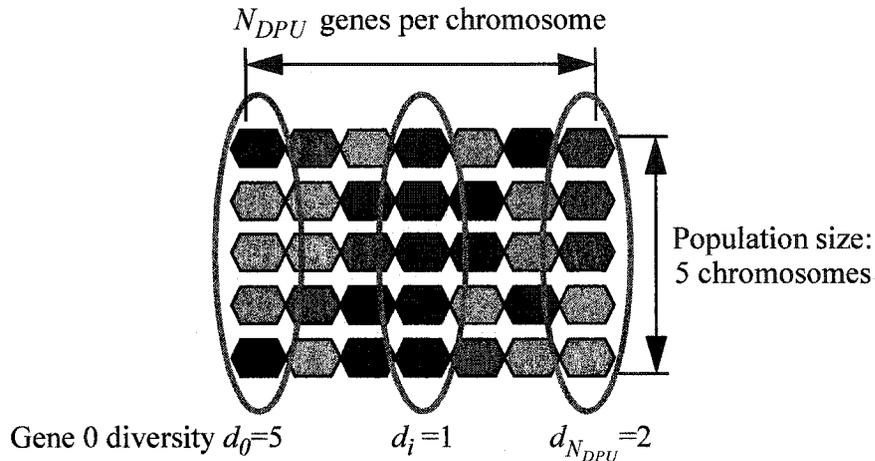


Figure D.2. Initial scheme for quantifying diversity.

For each gene in the chromosome definition, the *gene* diversity is taken to be the number of unique values throughout the population. The overall diversity is the average of gene diversity over all genes in the chromosome definition. This scheme was inherited from the PMX-based GA, where gene values correspond to DPU-identifying indices. For RKR, the diversity metric from Sections 6.1.5 and 7.2 is more appropriate because “distinct” gene values can actually be very close to each other.

- ◆ Frames (a,b,d,e) show the initial convergent behaviour of the cost, wire violation count, and rapidly dropping diversity.
- ◆ Unconditional acceptance of kids also resulted in a wildly fluctuating maximum cost (frames (f,k)) and wire violations (frames (h,j)), even though the bulk of the population had a cost that had converged to the minimum cost (mean and min virtually coincident, and correspond to the bottom boundaries of the curves in (f,k)). From a cost histogram for the population (not shown), this minimum cost is 14 for the top panel (short term), 12 for the bottom panel (long term). The fluctuation of maximum cost is a reflection of the more diverse search being performed by the kids.
- ◆ The wire count violations were mostly concentrated in a single value [frames (g) and (l)] after premature convergence.
- ◆ In frames (h,j), the minimum count of wire violations does not decrease monotonically because the cost being minimized is not the count of wire violations; such a cost would corre-

spond to Figure 6.8(c). This was shown to be a poor cost metric for guiding the search, albeit under different GA conditions.

- ◆ There are on average only 2 to 3 different values per gene once the population converges [frames (d,e,h,m)]. This can still give rise to a large number of chromosomes, since there are 24 genes. In fact, the unification of kids ensures that all 200 chromosomes are unique. Under RKR, however, it is possible for nonidentical gene values to differ by negligible amounts.
- ◆ Frame (c) shows the initially low rejection rate of clones and twins rapidly increasing as the population converges (diversity decreases). Thereafter, rejection slowly decreases, presumably due to increasing diversity from mutations.
- ◆ Frame (c) shows that the total number of kids is approximately 3 per generation. Two of these are offspring. With a 50% mutation rate, there is on average 1 mutant per generation.
- ◆ A marked discontinuity occurs at approximately 180K matings in frames (i,j,k,m). The cost and wire violations drop, indicating that a very good solution was created. The drop in diversity and acceptance of kids indicates that the good genes proliferate through the population via selection. Note that these changes are only sudden on the large time scale of the lower panel. After this proliferation, diversity slowly improves again due to mutations; acceptance of kids also increases again.

The gradual proliferation of a good combination of genes can be seen in Figure D.3. One of the valleys in the minimum wire violations corresponds to low cost and is retained as a favourable population member. There is some positive feedback occurring; the transition accelerates as the gene combination spreads to more chromosomes, thus increasing the prospects of the selected parents containing that gene combination. As in Figure D.1, this transition was accompanied by a drop in acceptance rate for kids.

Despite the less aggressive replacement policy of unconditionally accepting unique kids, convergence to a feasible solution was not occurring in all cases (Figure D.4). For the nonconvergent cases, it was first suspected that the kids were entering the bottom of the population and being constantly replaced i.e. serving as churn. To test this, rank of the kids was plotted (Figure D.5). Figure D.5(z) shows that most matings generated at least one unique kids i.e. kids that were accepted into the population. The kid rankings were initially distributed evenly over the population in rank, but immediately begin skewing upward by approximately 25%. This behaviour is an

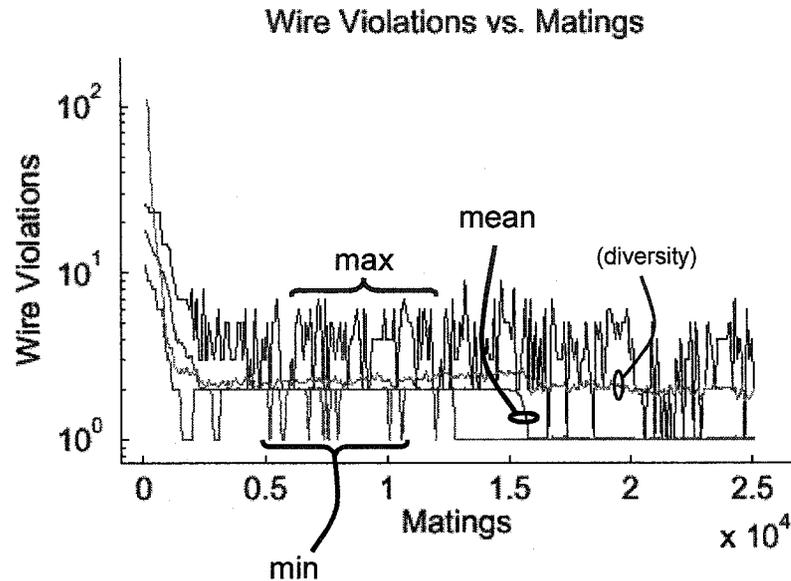


Figure D.3. A gradual transition of the mean violation count towards the new minimum as the genes proliferate through the population.

artifact of (1) the ranking routine, and (2) the degeneration in cost. It can be explained by several observations.

First, as mentioned above, a histogram of chromosome costs indicates that almost the entire population is at the population's minimum cost, which is flat with time. The few outliers are the last generation's high-cost kids, unconditionally accepted into the population. If a pair of parents are very similar, the offspring also be similar, and they will likely decode to the same DPU order. Therefore, there is a good possibility of the cost of the offspring will be the same as the bulk of the population. For now, consider the case where there are 0 mutants. Since kids displace the worst population members, the offspring are at the bottom of the population, thus replacing the high cost members that deviate from the single cost shared by most of the population. The population is then re-ranked by sorting according to ascending cost. The offspring remain at the bottom because cost is uniform (this was a confirmed behaviour of Matlab's sort function). However, the sorted population is assigned to its destination in *reverse* order because ranked fitness and its associated simple selection scheme (Figures 6.13 and 6.14) work with decreasing costs. This puts the off-

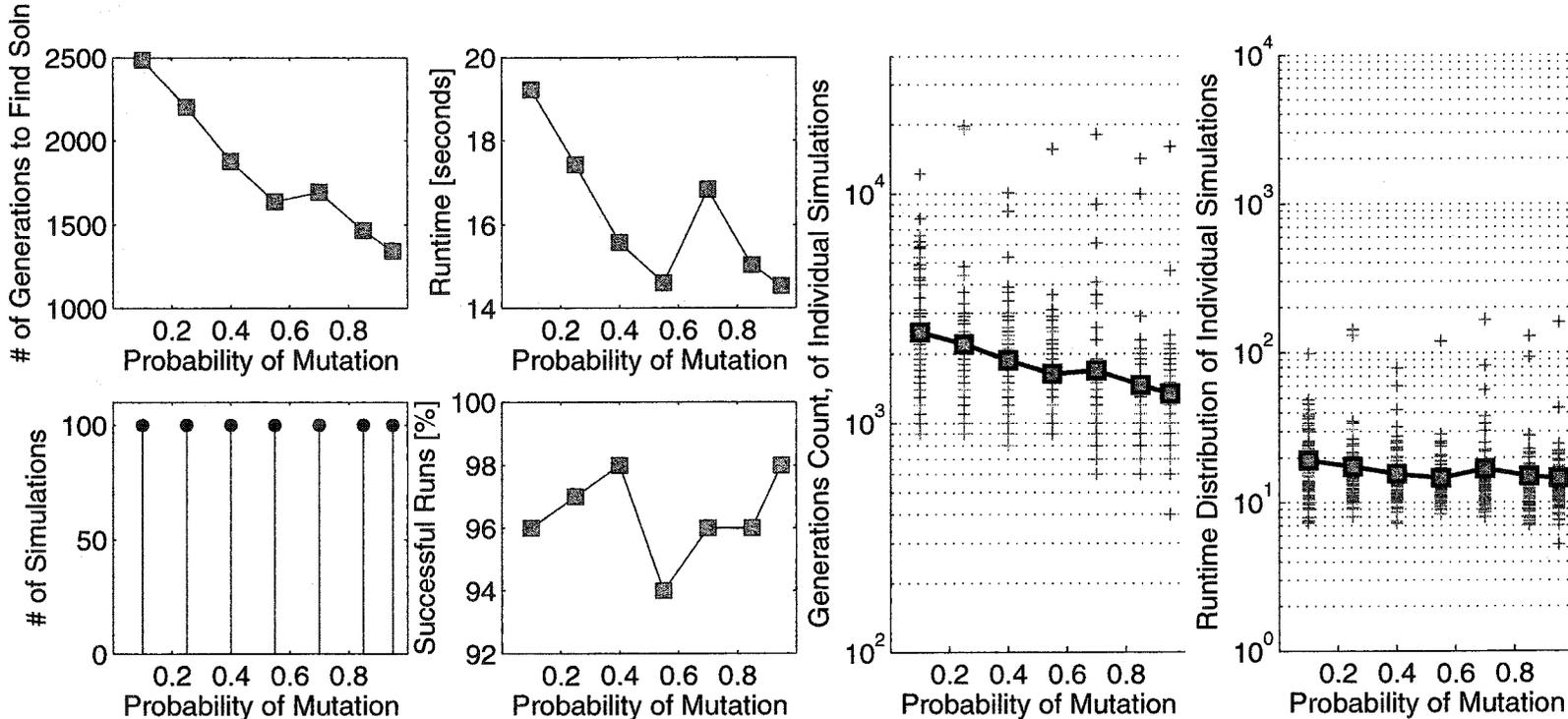


Figure D.4. Initial RKR implementation: 2-point crossover, mutation of offspring, unconditional acceptance of unique kids into population

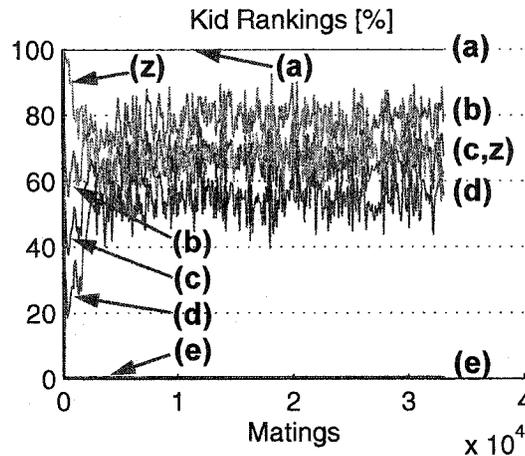


Figure D.5. Rank of kids relative to the population, for a search with 10% chance of mutating each offspring.

In the above ranking graph, 100% is a top ranked kid. Data points are taken over subsample periods of 100 matings. Only unique (i.e. accepted) kids are considered.

- (a) Highest population max in the subsample period.
- (b) Population max averaged over subsample period.
- (c) Population mean, averaged over subsample period.
- (d) Population min, averaged over subsample period.
- (e) Lowest population min in the subsample period.
- (z) Percentage of matings that resulted in at least one unique kid, including mutants.

spring at the *top* rank of population. In the absence of other factors, the kid rankings should always be 100%, once premature convergence is reached.

Other factors prevent the kid rankings from being pinned to 100%. Not all offspring have the same cost as the parents. Aside from impasse-breaking improvements, these offspring will have higher cost, or near zero ranking, since the rest of the population has the same cost. The same applies to mutants, except more so; that is, mutants are less likely to be like their “parents” in mapping to the same phenotype, and more likely to be an outlying high-cost kid, thus ranking at or near zero.

The upward skew of the kid rankings in Figure D.5 indicates that, on average, the kids are not bottom-feeding churn. As mentioned, however, the kids that are responsible for the wildly fluctu-

ating maximum costs are churn, since the poorest population members are replaced by the kids in the next generation. These kids are more likely to be mutants than offspring.

One of the conceptual problems with mutating offspring is that the genetic material is already biased by the selection of the parents. We tried to emulate nature by mutating general population members without bias. This managed to break the impasse in converging to a feasible solution almost all the time, thus yielding Figure 6.23.

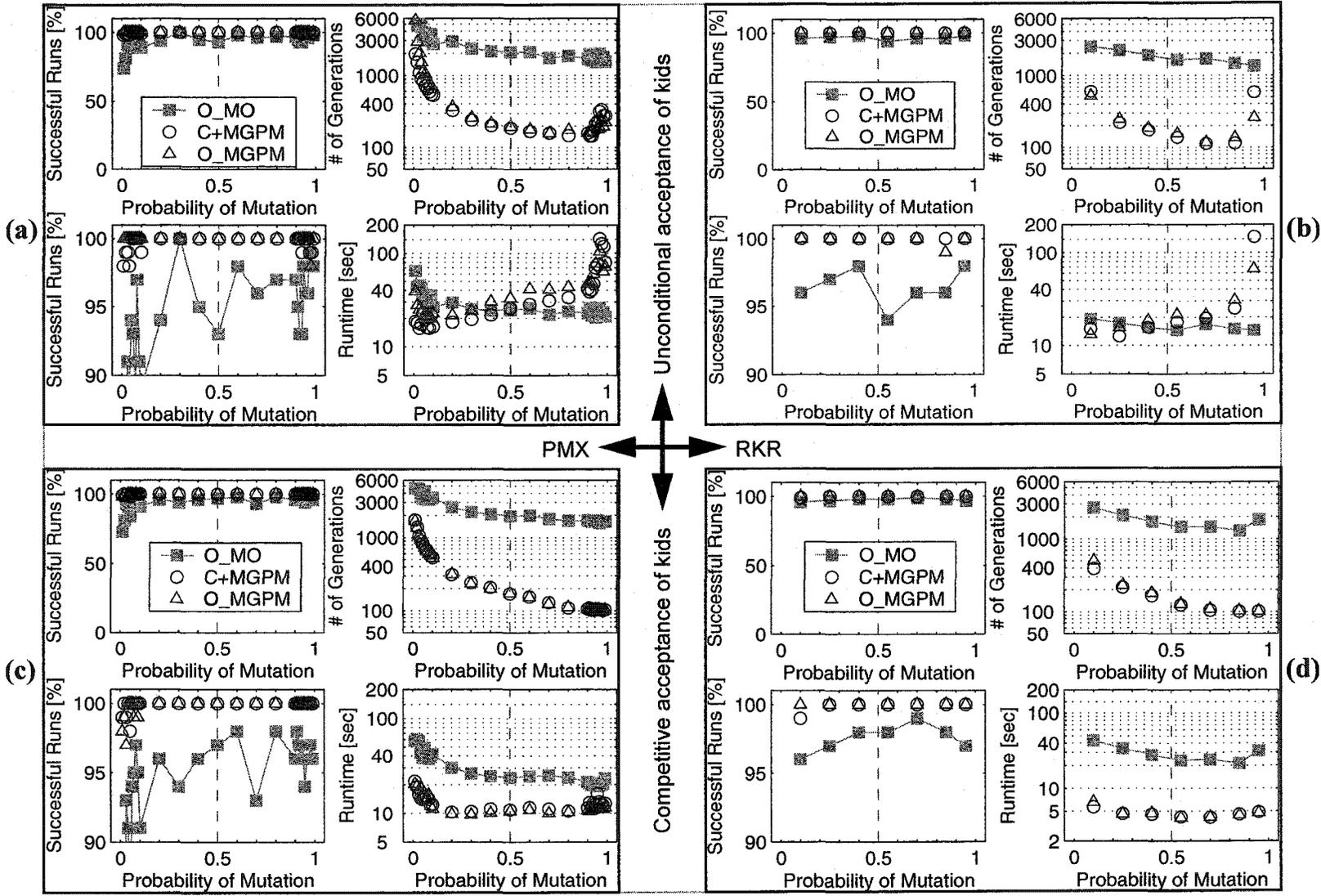
D.1.1.2 Population Replacement Scheme

For the GA developed in this thesis, the population replacement scheme was to *unconditionally* displace the population's worst genomes with all of the unique kids, regardless of their relative fitnesses. In the early course of making the GAs work, this was one of the many measures taken to improve diversity and avoid being trapped in local optima. For the basic DPU ordering problem, a much later effort investigated the *competitive* acceptance of kids into the population i.e. kids were combined with the population and the best N_{Pop} were retained as the next population. This was done for various mutation methods, under PMX and RKR. In Figure D.6, this is compared with a more detailed sweep of the corresponding test conditions in Figure 6.25 on page 157.

Figure D.6 shows that competitive acceptance of kids improves convergence slightly for RKR. For both PMX and RKR, it avoids or mitigates the poor convergence at high mutation rates. Since high mutation rates typically do not coincide with optimum search conditions, the question of whether this improvement applies to a fully constrained placement problem was relegated to a possible topic of future investigation. Intuition suggests that for easier problems like the simple DPU order problem, greedier searches such as competitive acceptance of kids is more likely to be favourable. This is because there are more feasible solution points in the search space, and the search has greater chance of encountering one before encountering a local optimum that is difficult to escape. For hard problems, in contrast, feasible solution points are sparse, so thoroughness of

Figure D.6. (Next page). Simple DPU order problem: Comparison of unconditional and competitive acceptance of kids for PMX and RKR.

As mutation rates approach 100%, unconditional acceptance of kids resembles generational evolution i.e. fewer of the best genomes linger, and information lost. For competitive acceptance of kids, only kids that are good make it into the population and the search is less likely to be set back by the generational scattering.



search becomes important. This requires less greedy search so as to be able to escape more easily from local optima.

D.1.2 Full Placement Constraints

D.1.2.1 Selection of Mutatees from Population

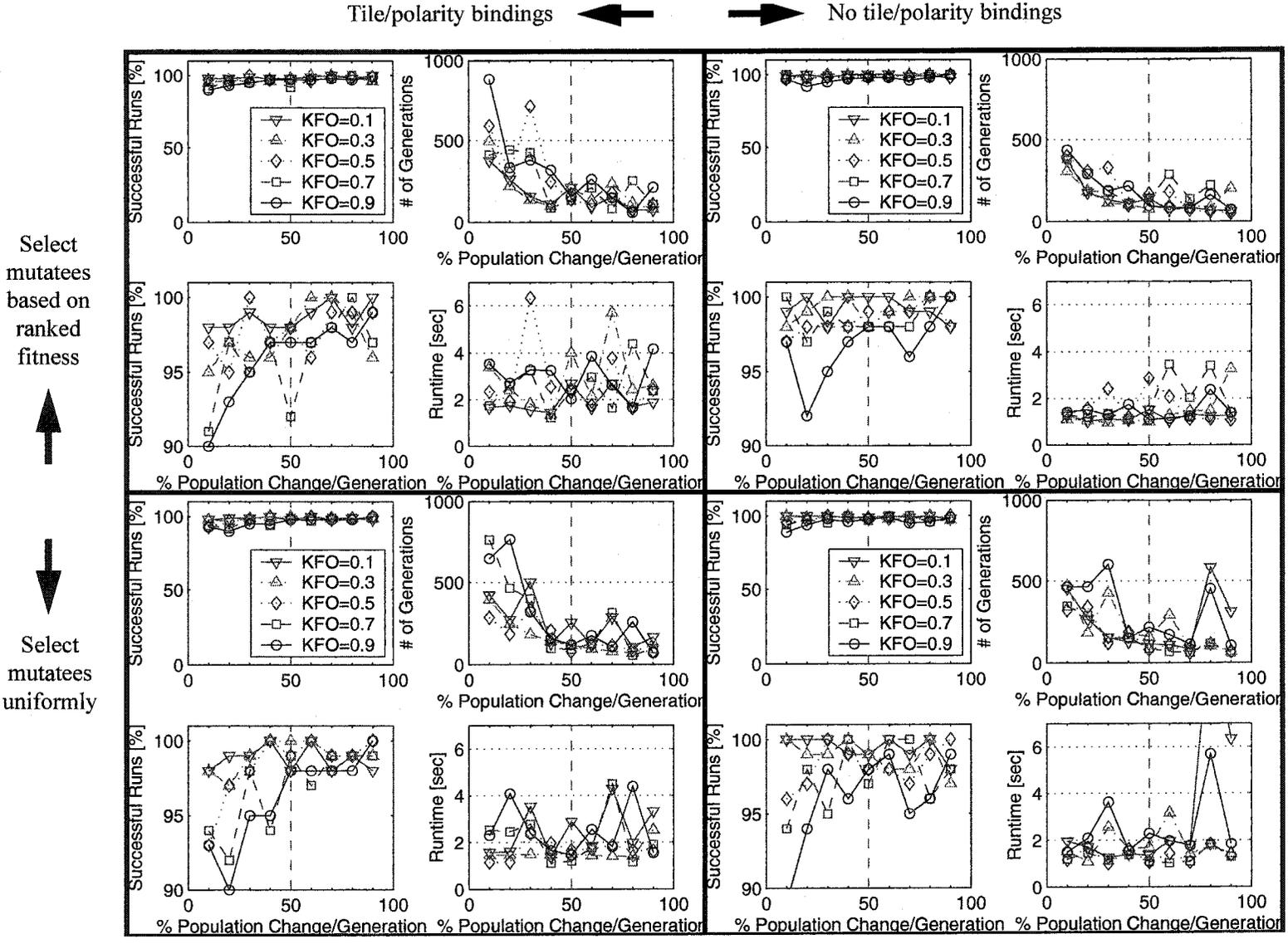
For mutation of general population members, the GAs in this thesis select mutatees from the population with uniform probability. This was to avoid excessive selective pressure and becoming trapped in local optima. Figure D.7 generalizes the characterization of the full placement problem across KFO and PFK (Section 7.1.1) by selecting mutatees by ranked fitness and comparing with corresponding behaviour in Figure 7.1 on page 185. This is also done for an the simple DPU ordering problem (only local wire length constraints) to see if the best method of selecting mutatees depends on problem difficulty.

The results do not conclusively favour one method of selecting mutatees over the other. For a simple DPU ordering problem (e.g. only wirelength constraints), ranked fitness seems slightly better, but for the full placement problem with all constraints, uniform selection seemed slightly faster. The latter is likely due to the less complex selection code itself, since there is no gross difference in generations count or success rate.

Since the simple DPU ordering problem represents a simpler placement problem than the full FFT kernel placement problem, an artificially harder problem was created by arbitrarily imposing additional tile and polarity constraints on the FFT kernel. Figure D.8 compares typical data from the full characterizations of Figures D.9 and D.10. No significant difference was observed between uniform and ranked selection of mutatees.

Due to the lack of significant difference between the two methods of selecting mutatees, a full investigation across various kernels was not pursued.

Figure D.7. (Next page): Comparing the selection of mutatees. Mutatees were selected either with uniform likeliness or according to ranked fitness. A “normal” case was considered, with all required placement constraints. An “easy” case was tried, with all constraints removed.



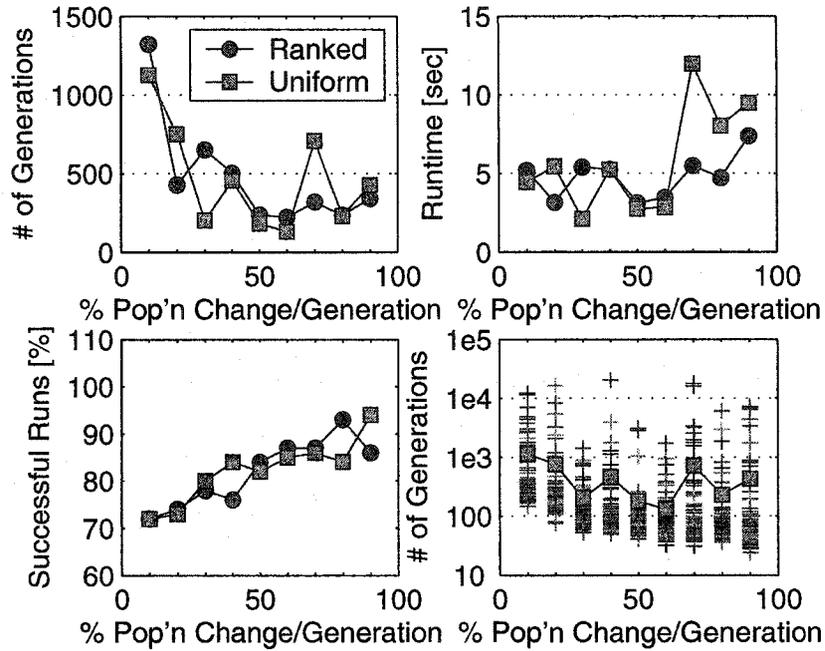


Figure D.8. Comparison of placement for the FFT netlist with extra tile/polarity bindings. The two variations correspond to selection of mutates uniformly and based on ranked fitness. Sample data taken from Figures D.9 and D.10.

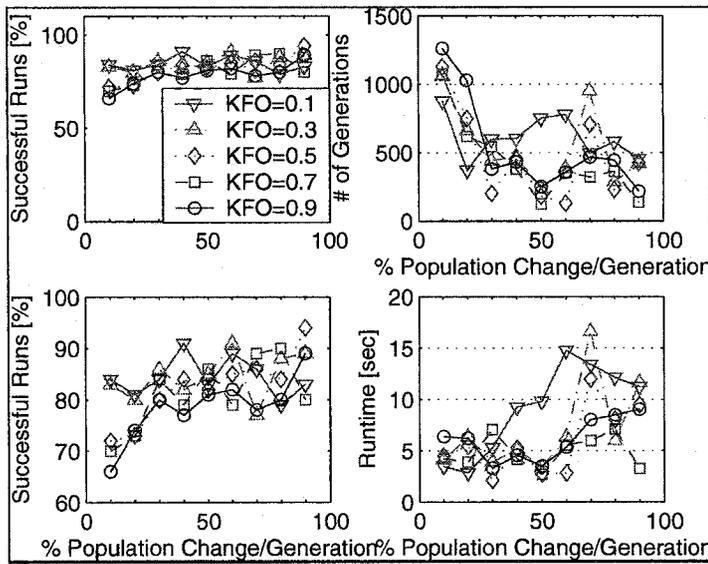
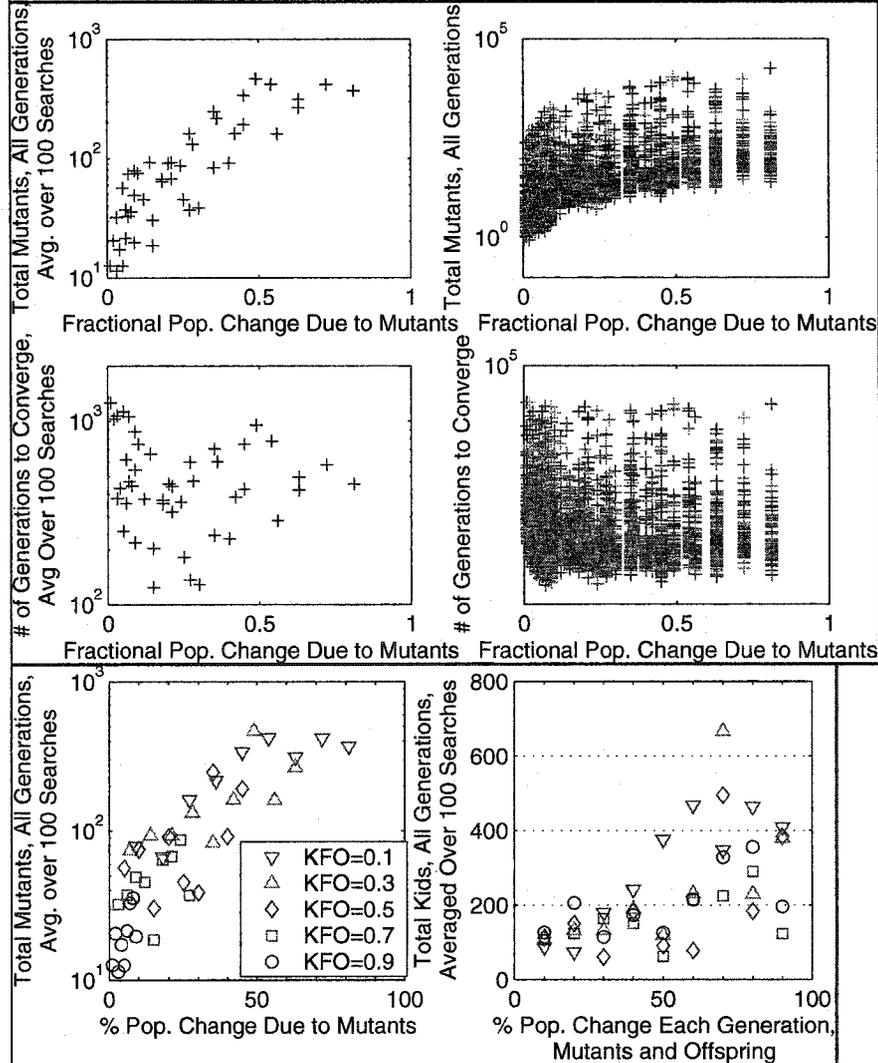


Figure D.9. Extra hard placement problem with *uniform* selection of mutates. Extra tile and polarity constraints were added to the FFT kernel. The kid counts and mutant counts have been normalized to $N_{Pop}=200$.



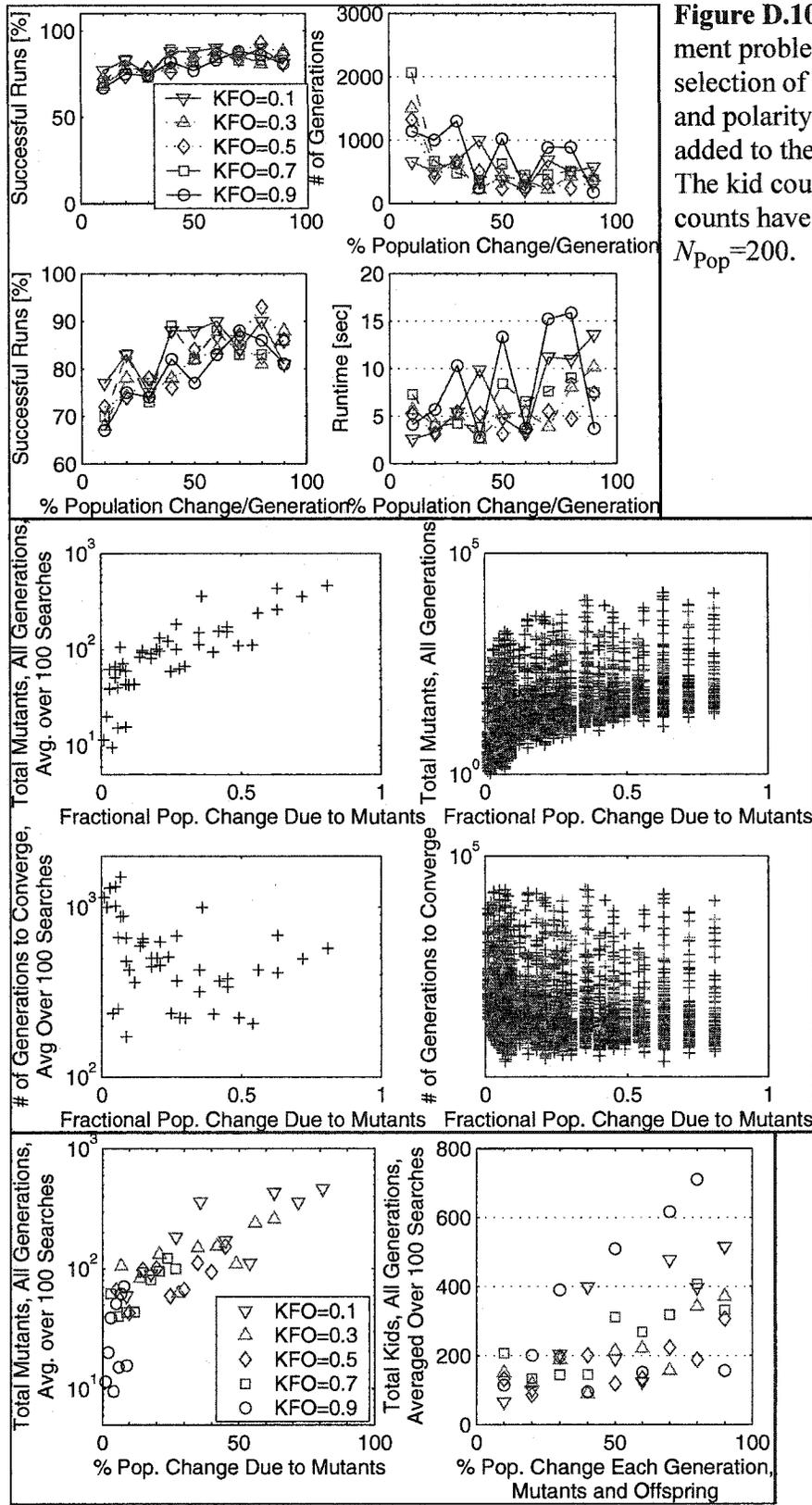


Figure D.10. Extra hard placement problem with *ranked* selection of mutates. Extra tile and polarity constraints were added to the FFT kernel. The kid counts and mutant counts have been normalized to $N_{Pop}=200$.

D.1.2.2 Data from Testing Constraint Mechanisms Separately

The following characterizations are for testing of the different constraint mechanisms separately (Section 7.1.1.2).

- ◆ The FFT kernel with all constraint mechanisms enabled is shown in Figure D.11.
- ◆ The disabling of tile binding mechanisms is shown in Figures D.12, D.13, D.14, D.15, D.16, and D.17.
- ◆ The disabling of polarity binding mechanisms is shown in Figures D.18 and D.19.

Each data set consists of 3 standardized panels of graphs to speculatively reveal trends which may not show up in generations count and success rate curves across PFK and KFO. For example, mutant count were shown in case mutants were largely responsible for successful searches. Scatter graphs are shown both in full density and sparsely (averaged over 100 searches), in linear and log scales, and both identified according to KFO and not. The dense scatter graphs serve as a reminder that the data is very asymmetrically distributed about the averages, even though the latter yields more readable curves. So far, only the expected trends are revealed e.g. total mutant count increases with increasing PFK, as well as with decreasing KFO. The most useful graphs are those in the top panels, as well as the total kids count in the bottom panel. The kid counts and mutant counts have been normalized to $N_{\text{Pop}}=200$.

D.2 The Case for a Simple Linear Diversity Measure

The reasons to avoid logarithmic scaling for diversity are utilitarian. The diversity metric is used for monitoring purposes so that appropriate action can be taken to avoid being trapped in local optima, and to balance off exploration and exploitation. To help make these decisions, the value range of the metric should be such that it is easy to infer or interpolate how good or bad a population is *relative to some well understood reference points*, such as the *no diversity* and *random scattering* cases defined above.

This last point cuts to the heart of the question of measuring diversity. Since diversity is an intuitive concept, any mathematical definition of a metric has large latitude for arbitrariness. The only common landmarks are the well understood reference points. For example, one can intu-

itively say that a certain measure is 75% away from the *random scattering* case and 25% away from the *zero diversity* case (Section 7.2). Logarithmic scaling causes the *no diversity* reference point to shoot off to $-\infty$, which renders it ineffective as a reference point. One is left to rely solely on the *random scattering* reference point. For example one population may be 50dB below the *random scattering* case, while another is 200dB below. It is not straightforward to decide whether that is bad or good. When diversity is regarded as a linear spectrum between *random scattering* and *no diversity*, the developer of the metric is forced to ensure that the metric behaves somewhat linearly as the GA search progresses. If there is too much nonlinearity (for example, D^{Simple} and $D^{GoodNorm}$ in Section 7.2), its problem as a metric becomes evident very quickly.

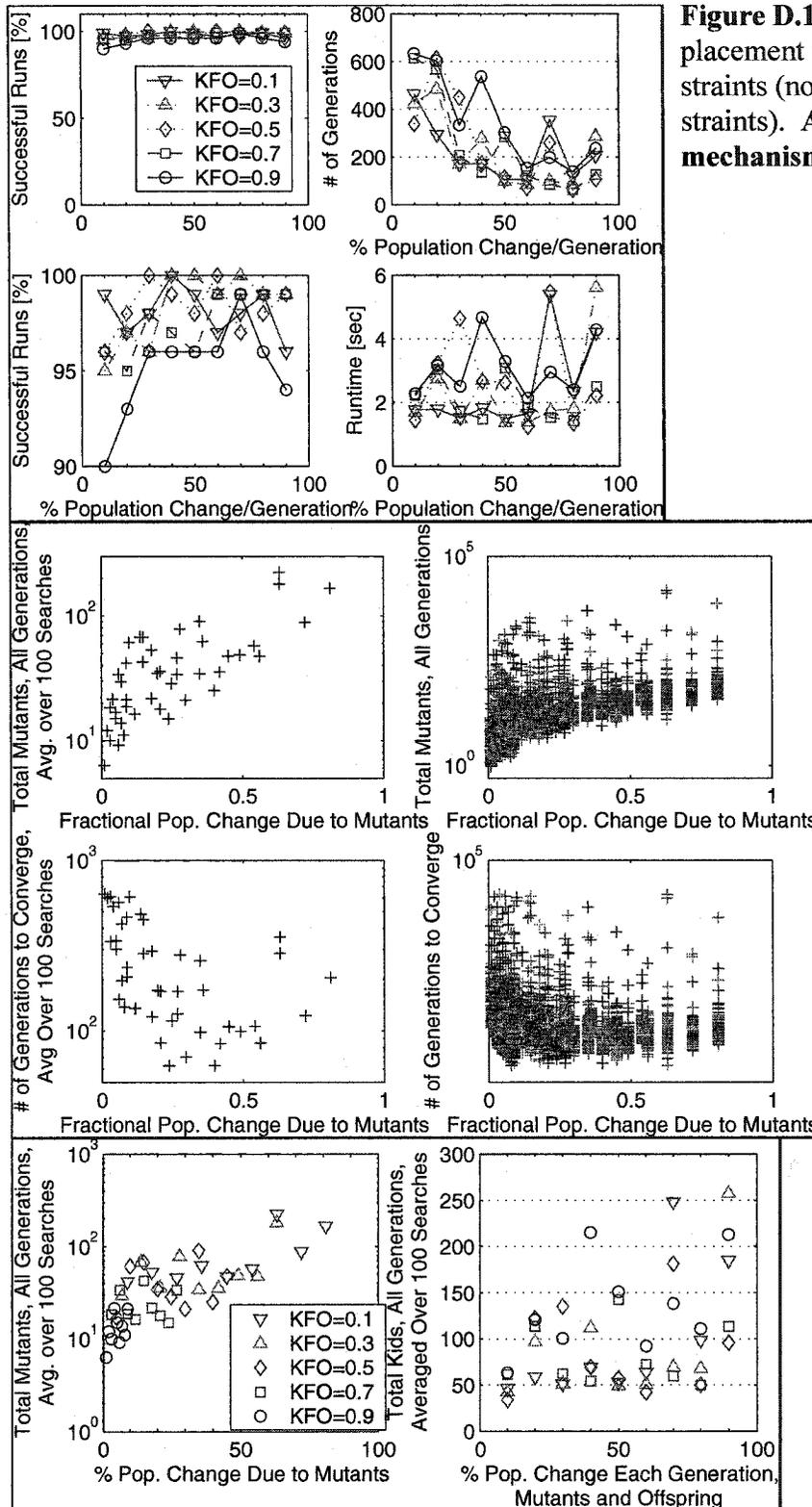


Figure D.11. The FFT kernel placement with only valid constraints (no arbitrary extra constraints). **All constraint mechanisms were enabled.**

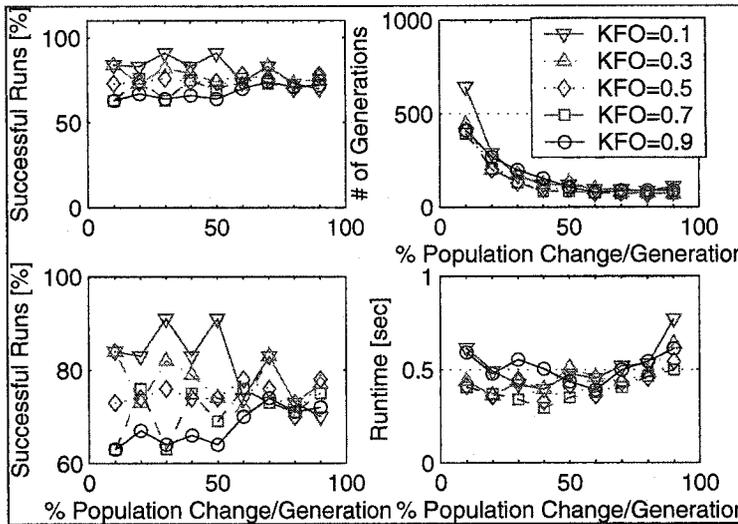
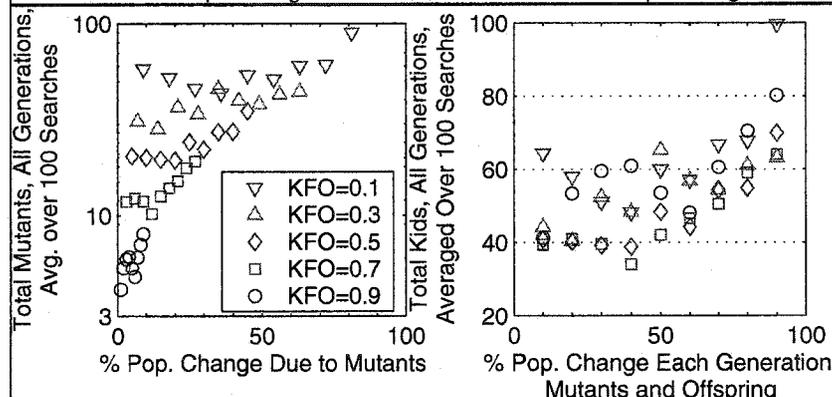
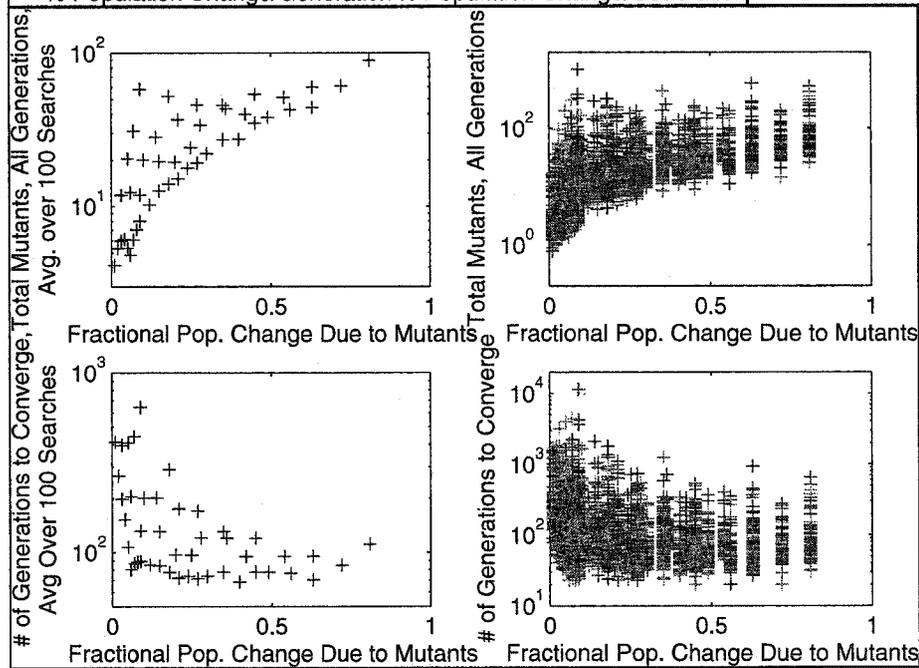


Figure D.12. FFT kernel placement. **Disable all tile binding mechanisms.** Only polarity binding mechanisms enabled. Compliance with tile bindings are completely random. Compared with enabling all constraint mechanisms, success rates are seriously degraded, though generations counts are not much impacted for the successful searches.



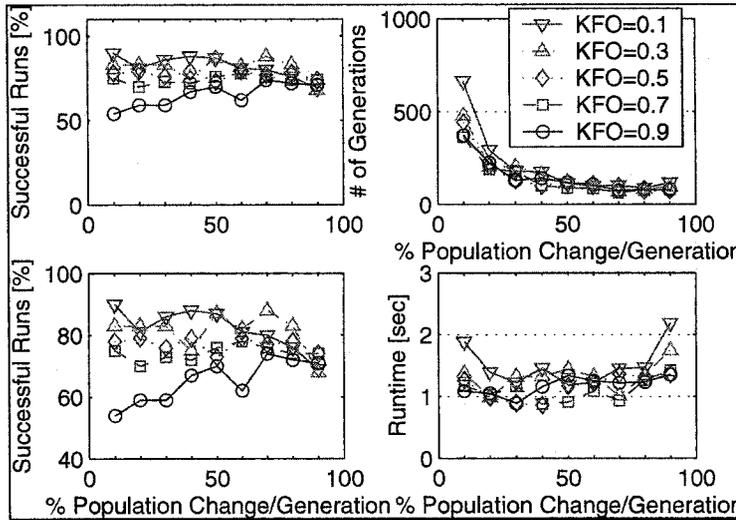
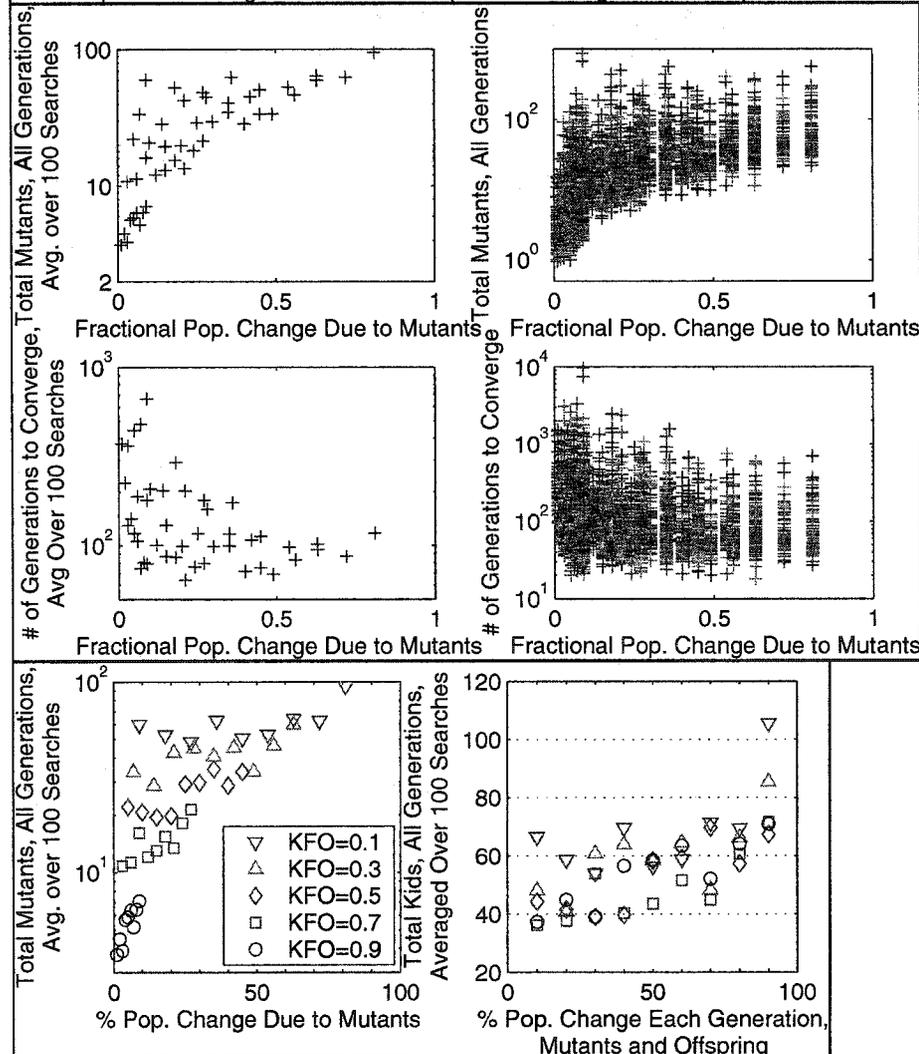


Figure D.13. A second data set for FFT kernel placement with only polarity binding mechanisms enabled. Compliance with *tile* bindings are completely random. Approximately the same distributions as Figure D.12.



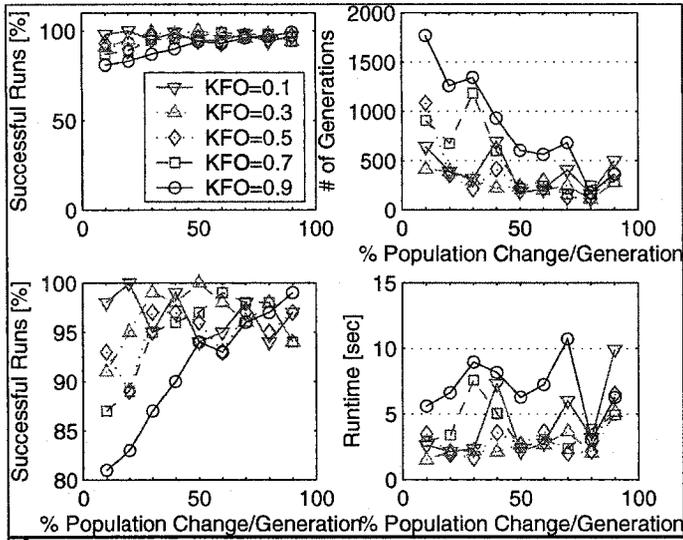
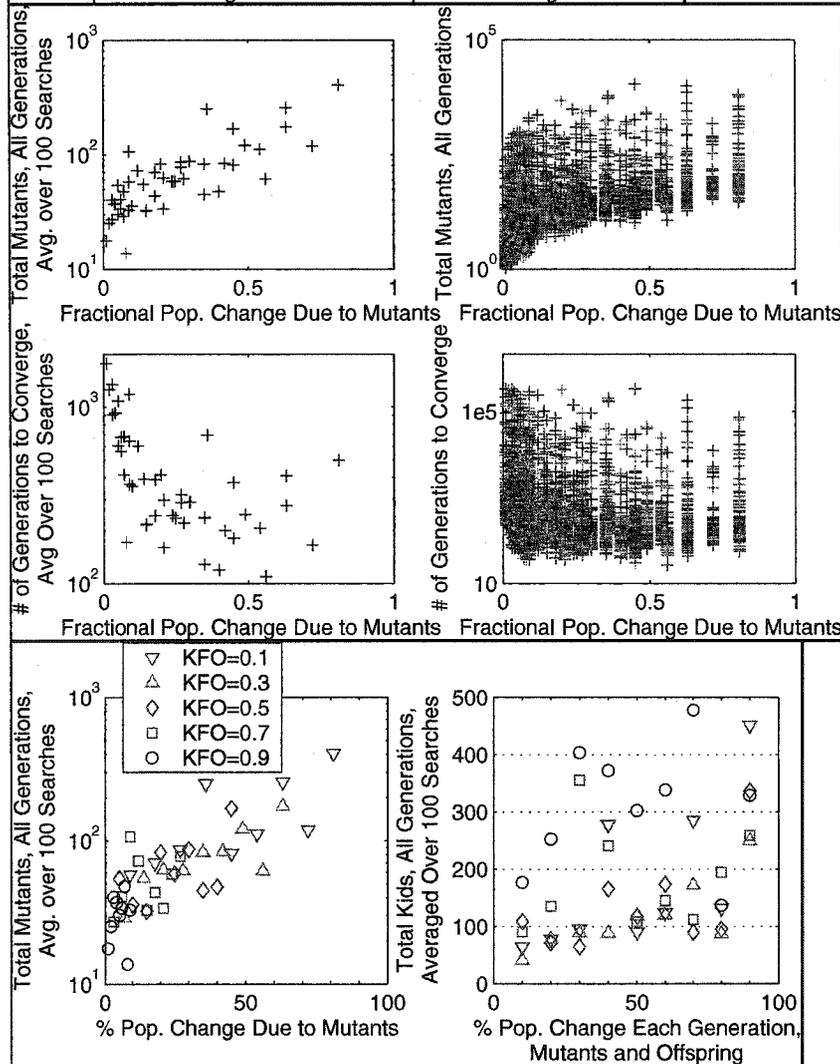


Figure D.14. FFT kernel placement. **Disable repair of tile violations.** Compliance with tile bindings rely entirely on key remapping and penalization of tile edge offsets. This significantly degrades the success rate and generations count compared to enabling all constraint mechanisms (Figure D.11).



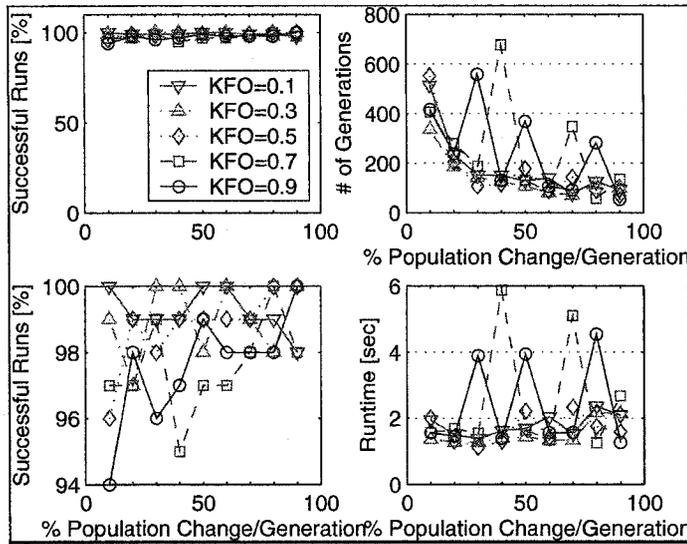
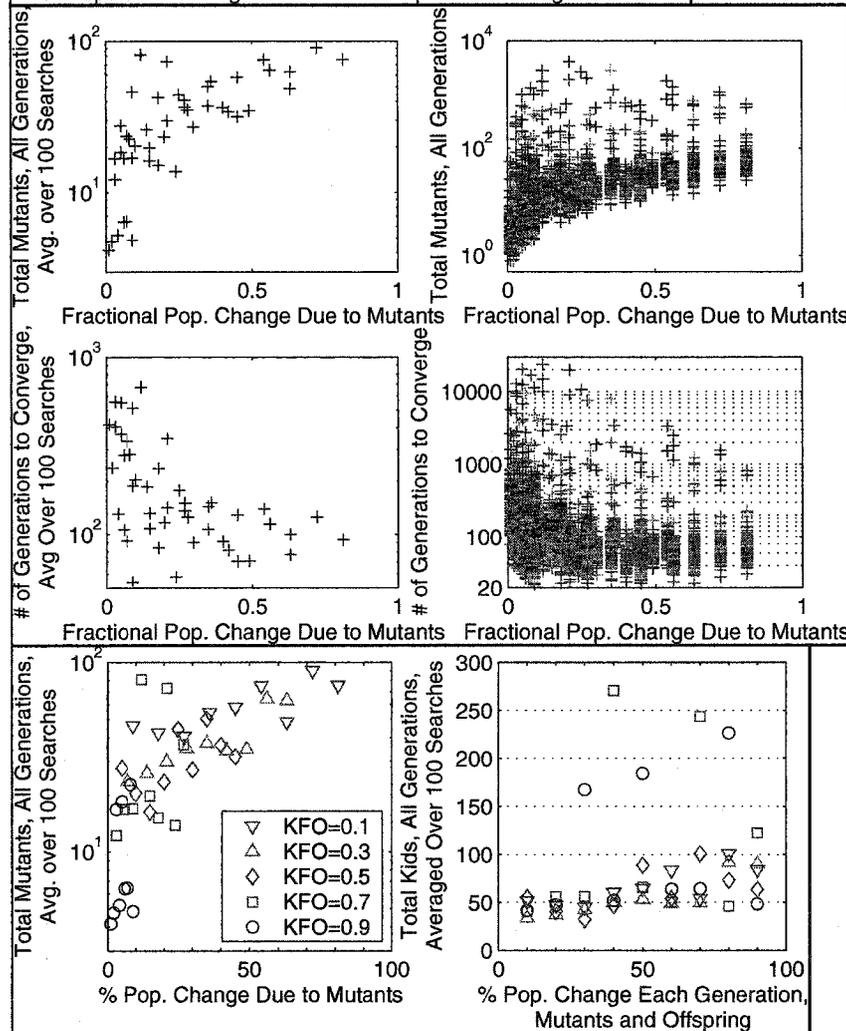


Figure D.15. FFT kernel placement. Disable penalization of *tile edge misalignments*. Compliance with tile bindings rely entirely on key remapping and repairs of violations. Doesn't seem significantly worse than with all mechanisms enabled (Figure D.11).



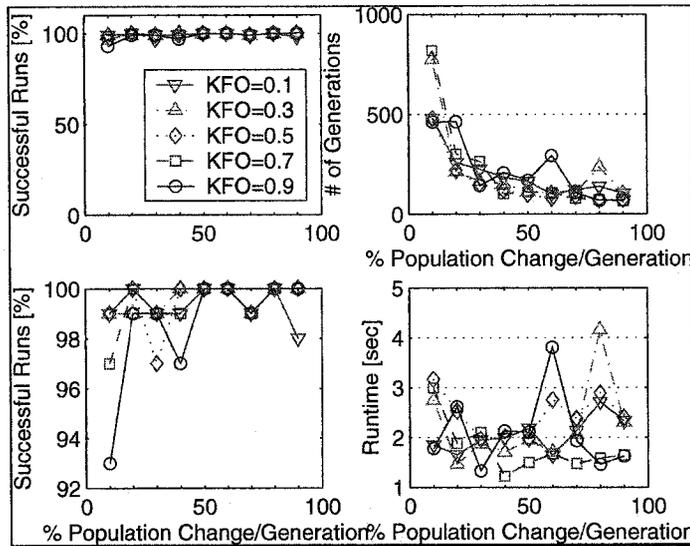
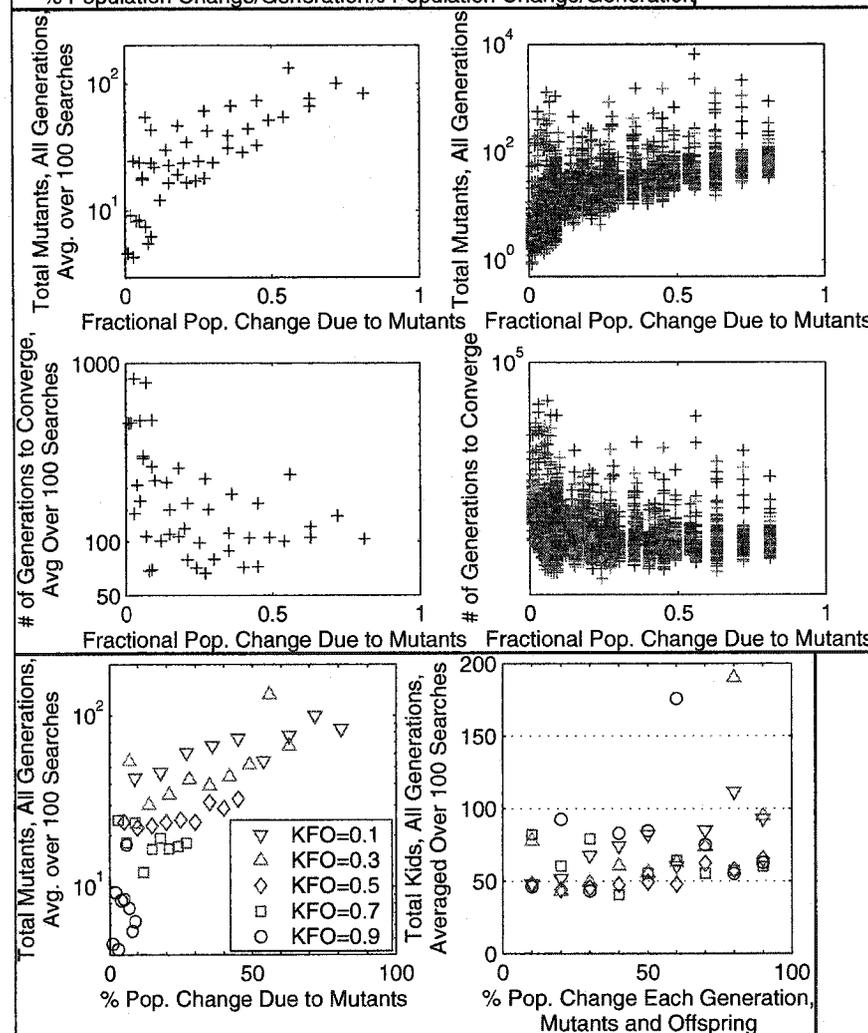


Figure D.16. FFT kernel placement. **Disable key remapping for tile-bound DPUs.** Gross infeasibilities are thus possible. Furthermore, minimization of tile edge offsets is rendered useless. Compliance with tile bindings rely entirely on repair of violations. Hence, it should be worse than D.15, but it isn't (not significantly). Therefore, neither is it significantly worse than enabling all constraint mechanisms (Figure D.11).



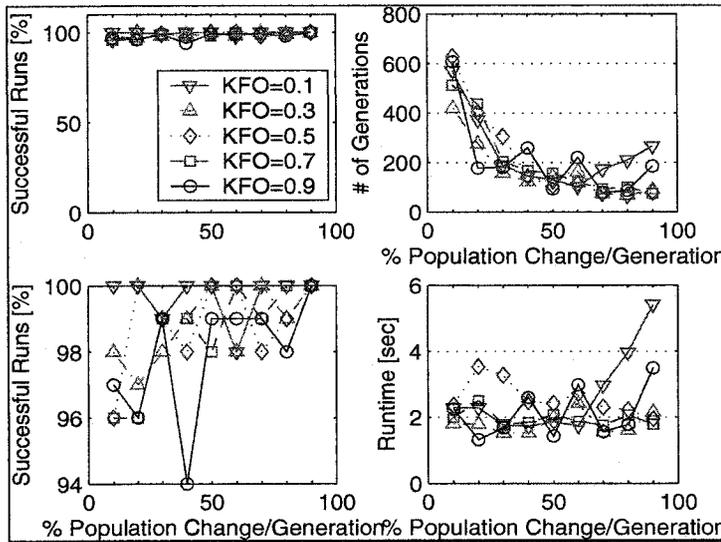
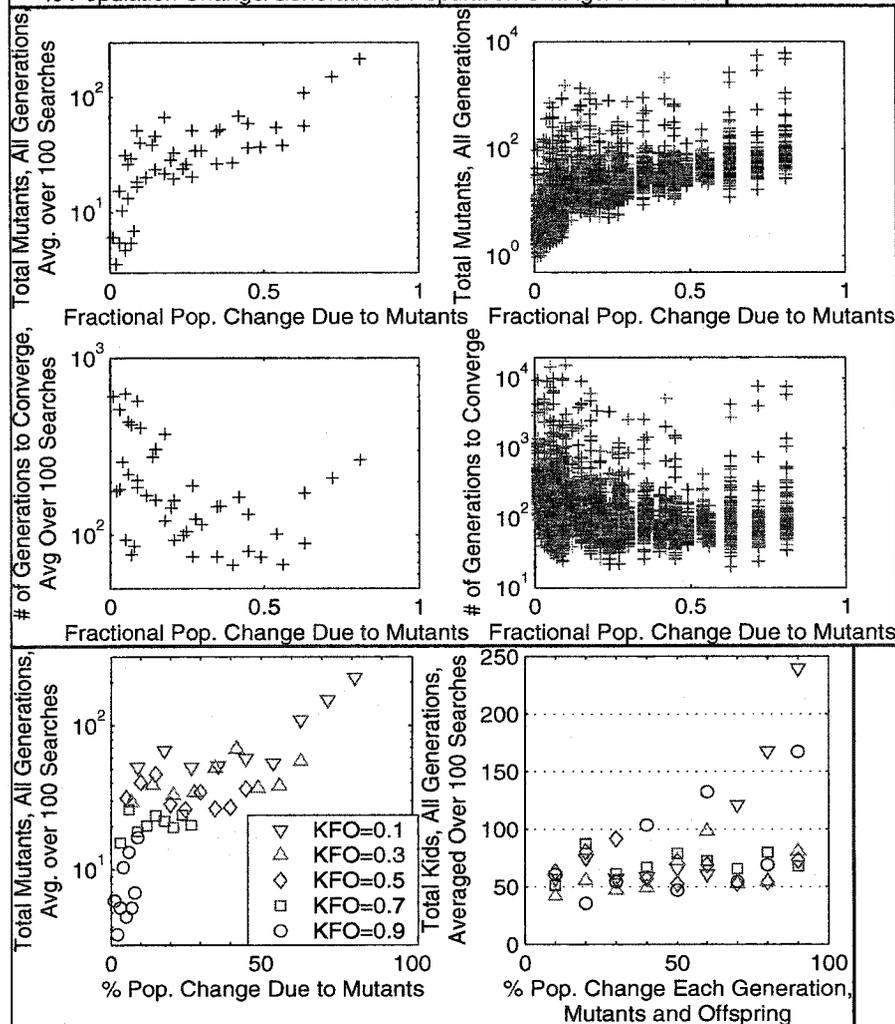


Figure D.17. FFT kernel placement. **Disable key remapping for *tile*-bound DPUs and penalization of *tile* edge misalignments.** Should be (and is) similar to Figure D.16, as well as the enabling of all constraint mechanisms (Figure D.11).



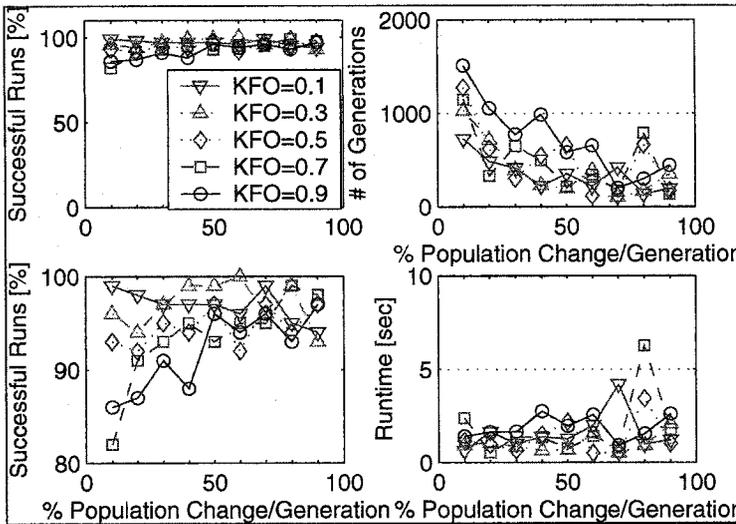
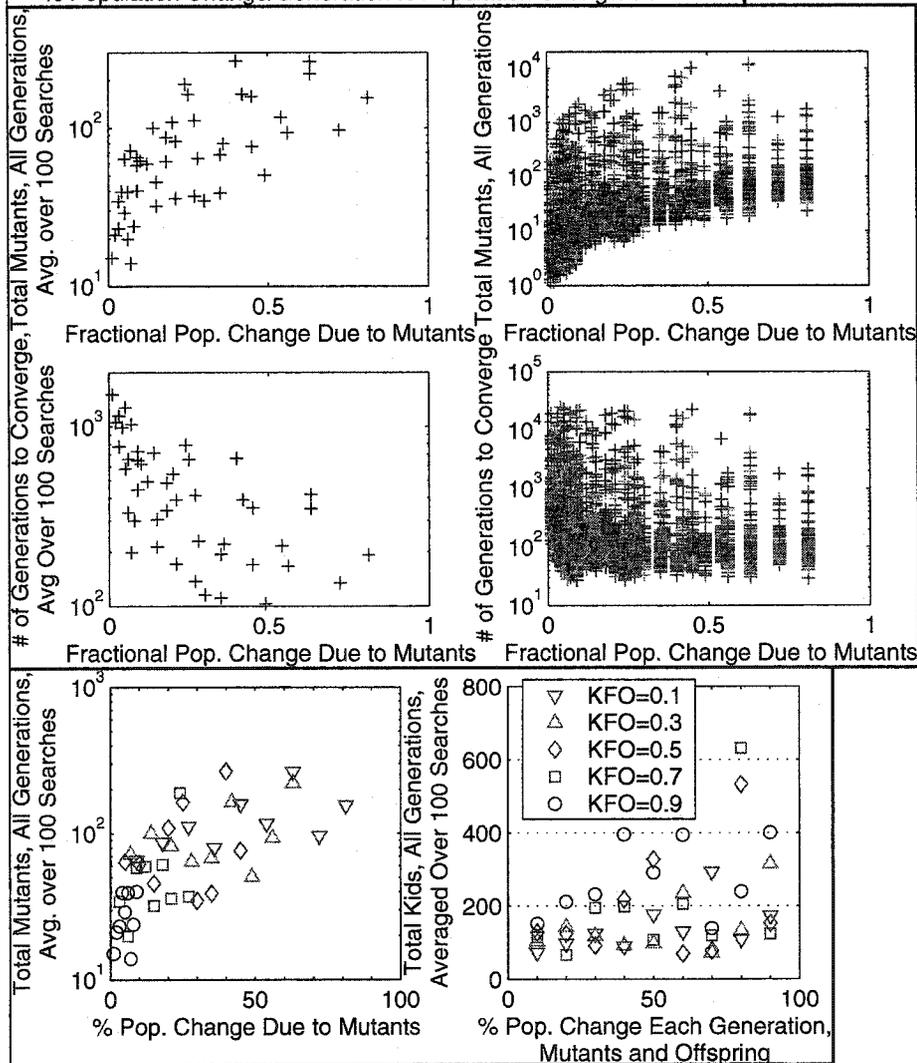


Figure D.18. FFT kernel placement. **Disable all polarity binding mechanisms.** Only tile binding mechanisms enabled. Compliance with polarity bindings are completely random.



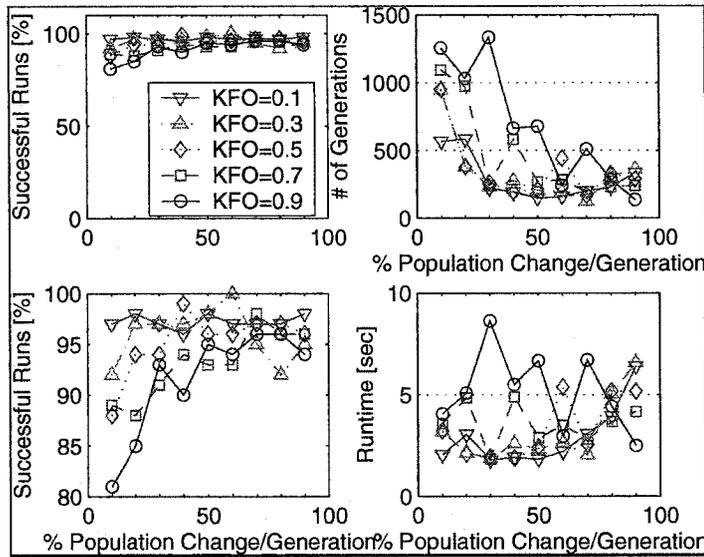
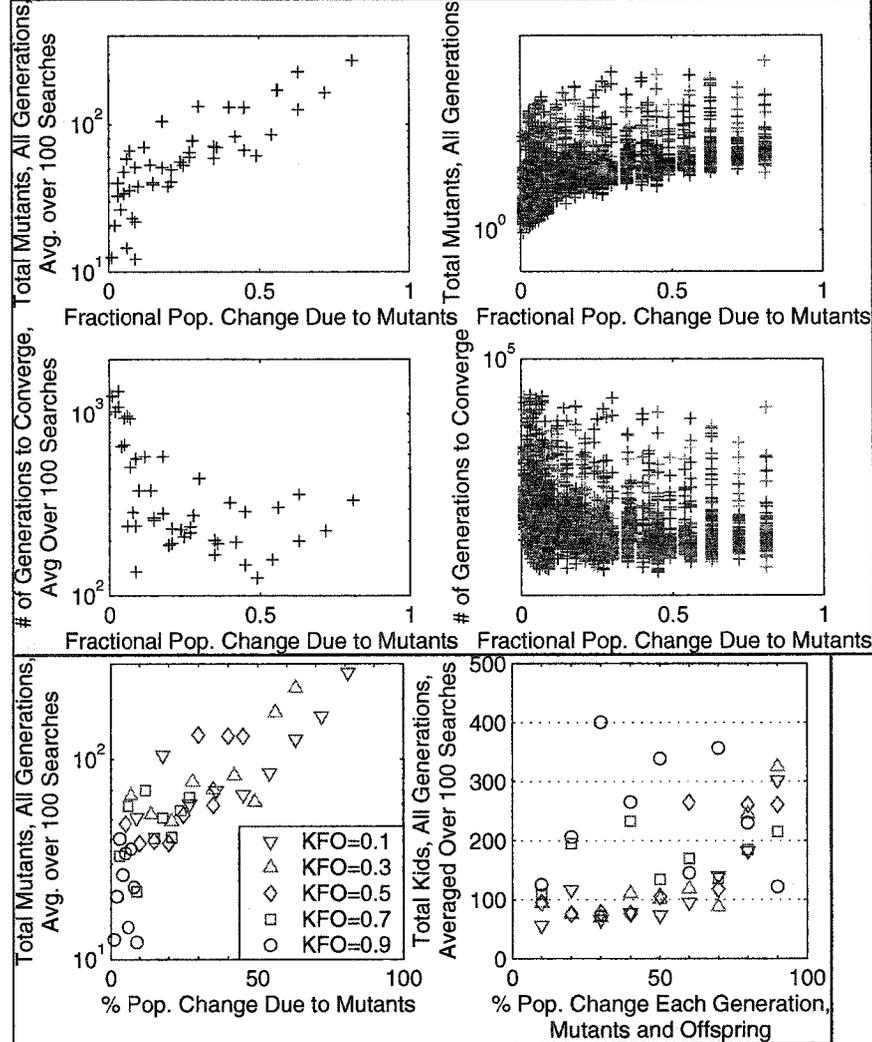


Figure D.19. FFT kernel placement. **Disable repair of polarity violations.** Compliance with polarity bindings rely entirely on penalization.



Appendix E: Genetic Algorithms for Synthesis

As mentioned in the outset of chapter 5, synthesis is a much more open-ended problem than place and route. Though it is tempting to draw a rough correspondence between a dataflow diagram and a netlist, it is possible for them to have very little correspondence at the component level. As in high-level synthesis for ASICs, this may arise due to the serialization of the operations onto standard cells.

For coarse grain reconfigurable platforms, another factor helps to obfuscate the correspondence between hardware and algorithm. The underlying hardware is already fabricated and cannot be changed at a low level, beyond the operational modes for which it can be configured. Therefore, the details of implementing the algorithm are tailored to the platform architecture. Examples from the FFT kernel include the splitting of the data set into 2 sets of concatenated local memories in order to retrieve the 2 data points required for the throughput of 1 butterfly per second. Other non-trivial implementation details that don't show up in the dataflow diagram include the twiddle factor storage; the counters, muxes, shifters, and masking that generate the addresses for the twiddle factors and butterfly data in a stage-dependent manner; and the DPUs to delay and interleave butterflies from the two data banks. In fact, none of the resources for timing or control show up in the dataflow diagram; a dataflow diagram at the algorithmic level shows the operations with all possible parallelism, before they have been scheduled.

As a reference point for a synthesis framework, we start with the Y-chart for ASIC design, as described in [GDWL92] (Figure E.1). The placement problem represents a transition from the structural to the physical domain. There will not be 100% correspondence between a design flow for reconfigurable logic and Figure E.1. From the CAD survey of Chapter C, we can break the compilation process into three rough activities:

- ◆ Partitioning algorithms into kernels
- ◆ Mapping each kernel to platform resources
- ◆ Scheduling kernel execution

Partitioning into kernels is similar in abstraction level to hardware/software partitioning/code-sign, since tasks may be performed in software or as a kernel. This phase takes place strictly in the

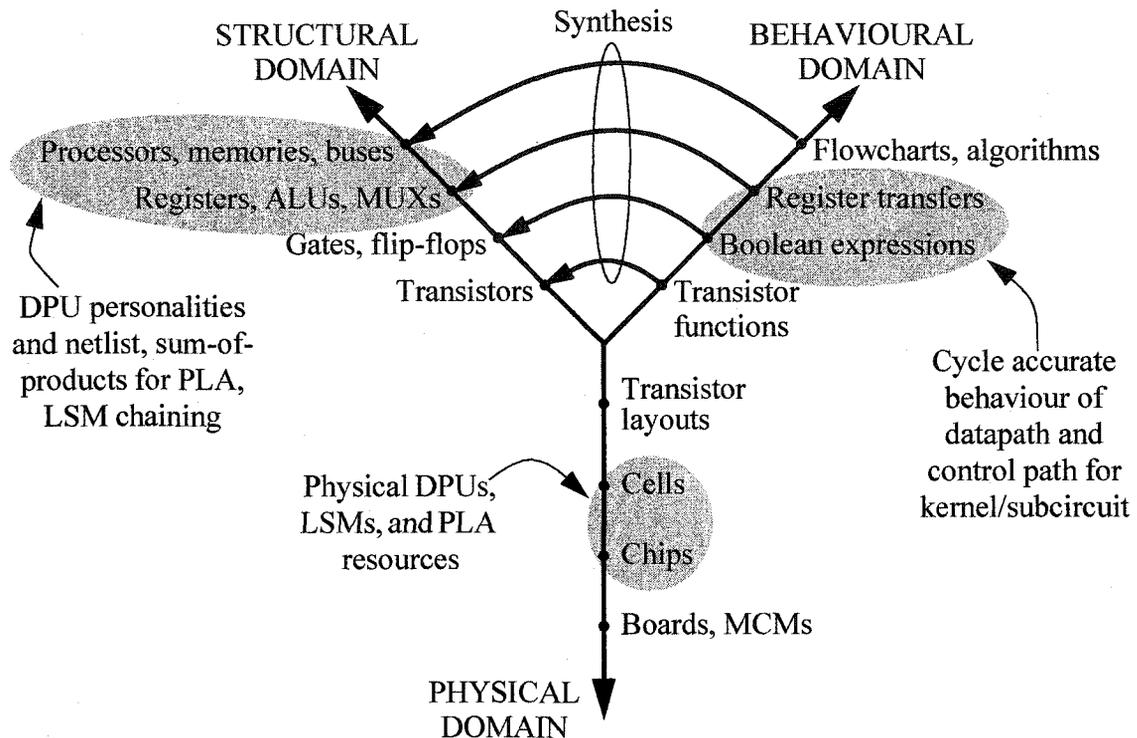


Figure E.1. Y-chart for ASIC design, as described in [GDWL92]. Corresponding elements for targeting coarse grain reconfigurable platforms are shown.

behavioural domain. The size of a kernel of functionality is comparable to the blocks in a typical signal processing chain of a modem e.g. FFT, QAM mapping, CRC, Viterbi decoding, etc. Complex blocks may be broken into several kernels.

The mapping of kernels deals with intrakernel scheduling, or timing of pipelines, as well as binding of intrakernel subcircuits, either to coarse grain DPUs or library macros for fine grain FPGAs. This corresponds to synthesis from the behavioural domain to the structural domain. This is the focus of this appendix.

The final step of kernel scheduling involves determining the order in which to load kernels, data, and configuration bits, and for how many iterations to run each kernel. It tries to maximize throughput and minimize latency at a high level by overlapping kernel execution with loading of data and configuration bits. It can also attempt to find a trade-off between data-centric and kernel-

centric schedules by analyzing number of loads of data/configuration bits based the amount of on-chip memory for each.

E.1 Problem at First Glance

In order to appreciate the synthesis problem in the context of Chameleon, it is necessary to briefly review the historical attempts at compiling for that platform.

E.1.1 Implications of Input Specification on Mapping

The initial attempt consisted of a compiler that accepted a subset of RTL Verilog, both behavioural and structural. A proprietary algorithm then synthesized the Verilog to components that make up a DPU, then tried to pack components into DPUs. Often, control path resources would be exceeded.

The second attempt had much more success. The combining of logic into DPUs and DPU personalities was left to the designer. This is the procedure described in Chapter 4. RTL control logic was still synthesized by the tools for the PLAs.

The contrast between the two approaches highlights a fundamental question in defining the synthesis problem. That is a need for an appropriate input specification. A corollary to that is that an intermediate form is needed for the design so that it can be manipulated and optimized during the mapping process. This may or may not be the same as the medium for input specification.

For example, in the second attempt at compiling for Chameleon, most of the synthesis work is done by the designer. Ideally, however, we would like to have an input specification that is not too close to the final form of the solution. Otherwise, the CAD mapping ends up playing a marginal synthesis role, and serves mainly as a format for input specification. The more abstract and free style the input specification, the greater the onus on the mapping algorithm to create a solution constructively. The closer the input specification is to the final form of the solution, the more iterative will be the nature of the mapping (as opposed to constructive). This can still, however, add significant value in the design process.

For the following discussion, we adopt the initial position that the input is in the form of a DFG or netlist, rather than sequential software. We do not address the issue of how the DFG is specified e.g. could be in the form of text netlist; neither do we deal with the extraction of the DFG from its initial text description.

We will not deal with gross serialization, such as conversion of the 5000 butterfly DFG of the 1K FFT to the single butterfly engine in the FFT kernel. The magnitude of this task is highlighted by the fact that most of the logic resources in the FFT kernel is devoted to feeding the butterfly with right data at the right time, as well as retrieving and storing the results. This is not apparent in the original DFG. Such serialization occurs at a high level of abstraction, and takes advantage of very problem specific structure. As such, human cognition is central to this kind of problem transformation. Therefore, any DFGs or netlists will be considered to be parallelized or serialized to the desired form, such as in Figure 4.9.

E.1.2 Challenges Beyond FPGA Mapping

In FPGA mapping, there is usually isomorphism between a DFG vertex and a LUT or macro. When libraries of macros are used, one can choose a macro with very nearly the desired operation.

DPUs, however, are complete ALUs, and much more complex than LUTs. In contrast to mapping to macros, a DPU is not tailored to the desired operation, so area cannot be saved by having a DPU do just the task of a single vertex. The mapping process should combine as much functionality into a DPU as possible to maximally exploit the area that it already occupies. Thus, there is not necessarily any isomorphism between the DFG vertices and DPUs.

The DPU may be compared to an FPGA macro that represents a multifunction unit. However, it would be a very complex one. Multifunction units typically perform functions that are similar. While they may be considered nonmemoryless due to pipelining, there is typically no recursive feedback. The DPU, however, cascades an elaborate shifter with an ALU, as well as optional registers for pipelining, storage, feedback, or retiming. Thus, a DPU has state, and can have different timing depending on the DPU personality that takes effect at any particular clock cycle i.e. whether registers are used for pipelining. In effect, the DPU can be a state machine that changes its combinational logic and the number of state bits and depending on the personality. Furthermore, feedback allows the starting state for each personality to depend on the results of the last personality.

E.2 Traditional or Apparent Approaches

E.2.1 CAD assisted microcoding

Because of the complexity of the DPU, and the need to pack functionality into it, one approach to CAD mapping is to expand on the current practice of assembling DPU personalities by providing CAD assisted microcoding. For example, software tools may popup menus to allow the designer to select the operational mode for an intrakernel component. However, the designer still performs much of the mapping by deciding on the functionality of the component, which modes for different components to integrate into a DPU personality, and which personalities take effect in each clock cycle. Therefore, this approach was not expected to add much value to the mapping process.

E.2.2 Vertex Clustering and DPU Packing

One technique that can be used for vertex clustering is template matching. In this scheme, the DFG is searched for operation/vertex/edge combinations that match a set of templates. Subgraphs that match are recognized as candidates from which to form DPUs.

Graph transformations can play a prominent role in refining the design. For example, delays were pushed along paths in the DFG in designing the FFT kernel. This is a simple retiming transformation, generalized in [LS91]. Here, it allows pipeline registers to be packed into DPUs that have already been allocated.

Other transformations may be borrowed from ASIC synthesis, though they are applied at a higher level of abstraction. For example, tree height reduction [GDWL92] reduces the resources required in the datapath through algebraic manipulation of arithmetic expressions. Similarly, binary decision tree manipulation simplifies boolean expression in the control path. Aside from reducing the number of vertices in the DFG, functionally identical rearrangements in the order of DFG vertices can be sought to permit greater packing of the DPUs.

Complications with Vertex Clustering

A major impediment to packing DPUs is that it ignores the effect on control path complexity. For example, consider two vertices that naturally belong to different clusters, each under the control of its own state machine. Let us refer to these state machines as the “natural” state machines

for the two vertices. If these two vertices are put into the same DPU, the control logic for this DPU can become extremely complex, depending on whether the natural state machines change states (and therefore DPU personalities) at the same time. Even if they do, there is still the possibility of a large combinatorial explosion in the required number of DPU personalities and states. This may have been a contributing factor in the difficulties in Chameleon's first approach of DPU packing.

Another complication that must be accounted for in vertex clustering is the issue of timing. A [C]DFG has no notion of timing. However, each DPU output must be registered in order for the array to run at its maximum clock frequency. Therefore, each DPU acts as at least one pipeline stage, and potentially more, depending on whether the registers in the operand arm are used to buffer the operands. Hence, exploring different ways of packing vertices into DPUs has the effect of scheduling vertices in different pipeline stages, as well as changing the number of stages in the pipeline. This effective retiming of operations must be tracked to maintain proper synchronicity when two data flows meet as operands to a vertex. Fortunately, for *basic blocks* (sequences of operations without conditional branches or loops), data flow is feed forward only, and this is the extent of the complication.

E.3 Using GAs to Determine DPU Personalities

Complication of the control path by packing DPUs can be avoided by respecting user-defined subcircuits. Examples of subcircuits from the FFT kernel are the butterfly counter, the twiddle address generator, the address generator for butterfly input data, and the butterfly itself. These subcircuits can be equated to "FSM with Datapath" (FSMD) blocks [GDWL92] in terms of hierarchical abstraction and degree of autonomy. That is, they have their own FSMs, though these do accept input signals from other FSMDs in determining their state changes.

With the subcircuit as the object to map, a GA can be applied to find the right DPU personalities, and their sequencing, to generate the desired result. In the next sections, we discuss issues that arise from this approach.

Note that the GA is not used to design the whole kernel at once. This is because evaluation of a kernel design by the GA requires that the kernel output be compared to ideal output. However, an

ideal output does not exist because it is a function of the input data. In contrast, many of the subcircuits in a kernel can have well defined output values over time, and are readily evaluated.

E.3.1 Combinational Explosion of DPU Personalities

Even though the control path is simplified by respecting subcircuit boundaries, the search space of DPU personalities is still very large. Each DPU has up to 8 configurations, selectable by a 3-bit selection signal. Therefore, for N DPUs, there are 8^N personality combinations possible in *each* clock cycle, selectable by $3N$ select bits. We expect up to approximately 4 DPUs per subcircuit.

The 8^N combinational explosion assumes that the DPU personalities are already selected, and that all 8 personalities are in fact used. Furthermore, it assumes that DPU personalities change independently of one another. It ignores the fact that the DPUs form a subcircuit, and the strong probability of a relationship between the times at which they change personalities. Instead of presuming this, we may search for correct personalities, and their sequencing, by considering each DPU configuration as a string of configuration bits. A DPU has approximately 50 configuration bits, so 4 DPUs will have approximately 200 configuration bits. Using this “superconfiguration” as a basis for a chromosome allows for building blocks that span DPU boundaries, thus exploiting their relationship as components of a common subcircuit.

There is a drawback to aggregating the configuration bits, as opposed to a chromosome encoding that separately encodes each DPU with up to 8 personalities. Even if we limit the number of superconfigurations to 8^N , the search space for the aggregated configuration bits is larger than not aggregating the configuration bits. As an extreme example, it is possible for all 8^N superconfigurations to differ only in one of the N DPUs. This means that one DPU has 8^N personalities while all the remaining DPUs have only 1 personality each. This violates the limit of 8 personalities storable in each DPU’s cache. These infeasible design points only exist because the distinction between DPUs has been removed in aggregating the configuration bits. The use of superconfigurations introduces a large number of such infeasible points into the search space.

E.3.2 Combinational Explosion due to Time Dimension

At this time, we anticipate two problems related to time varying personalities that must be addressed by the algorithm. The first is the staggered times for the DPUs to change personalities, due to pipeline delays. The second is the large number of clock cycles for which the kernel runs,

giving rise to a virtually infinite number of ways in which superconfigurations can be time sequenced (even if there are only 8^N superconfigurations).

The fact that the data has to ripple through the pipeline stages means that related DPU personality changes between the DPUs will be staggered in time by one clock cycle. Therefore, a subcircuit with a latency of M cycles may need M unique superconfigurations to fill the pipeline, and M unique superconfigurations to empty the pipeline. This detracts from the value of aggregating configuration bits, which benefits from DPUs changing personalities at the same time.

The problem may be amplified by overlapping the emptying of the pipeline with the filling of the pipeline by a new series of data samples. This is the way that the FFT kernel was designed. Therefore, to make the mapping amenable to automation, it may be better to allow the pipeline for one stage (or outer loop iteration) to empty before starting to fill the pipeline with data for the next outer loop iteration.

Long kernel execution time can be dealt with by exploiting the fact that the DPUs change personalities in a simple and regular way once the pipeline is filled. This is a consequence of the stream data nature of the kernel. For example, in the FFT kernel, most DPUs either had a fixed personality for an entire iteration of the outer loop, or alternated between 2 personalities. To take advantage of this in the general mapping problem, a chromosome should be able to specify a few superconfigurations over which to cycle for the entire steady-state operation of the pipeline.

E.3.3 Chromosome Evaluation

Each chromosome has to be evaluated by evaluating the behaviour of the subcircuit that it represents, through Verilog simulation. Hence, seamless and streamlined access to a standards-conforming Verilog simulator is needed by the GA program while it is running; it cannot be a pre- or post-processing step. For each generation, the GA will assemble the kid chromosomes into Verilog files and submit them for simulation along with the behavioural models for the reconfigurable array components.

At this time, attention is restricted to the datapath. This means that the personality bits should be loaded into the configuration cache for the DPU instance, and the testbench will merely signal to the DPU which personality to use in each clock cycle. In a real design, the control path FSM needs to be devised to generate the right personality selection signal at the right time. The real life FSM must be made to do this by relying on other timing signals from other FSMs.

Since the latency of a phenotype is not known before evaluation, the simulation output needs to be searched in time for the correct sequence of output values. However, it is not enough simply to examine the subcircuit output for correctness. The problem is that on a very special combination of DPU interconnections and time sequencing of DPU personalities can generate an output that looks even remotely correct. Therefore, there is not much guidance given to the GA search if the fitness relies only on a simple measure of correctness in the output signal. What is required is to pull apart the attributes of a correct output signal so that credit can be given to partially correct signals. For example, some possible attributes of the butterfly counter in Chapter 4 which can be rewarded are

- ◆ the fact that it changes at regular intervals
- ◆ The reward can be commensurate with how close the regular interval is to one cycle
- ◆ The more constant is the periodic change, the greater the reward
- ◆ A greater reward is given if the periodic change is closer to unity

The creditable attributes for the address generator for the data that feeds the butterfly will be more complicated. However, the butterfly address generator is based on the simplifying breakdown of the addressing pattern (Table 4.2 on page 75). This can be used to establish attributes to reward in the evolution of the circuit. For example, assume the simplified problem of designing the address generator for a single FFT stage i . All bits of the address aside from bit i will form a simple butterfly count, for which the attributes determined above are suitable. Therefore, bit i is assessed separately from the “non- i ” bits. As described in Chapter 4, bit i must alternate between 0 and 1 every cycle in order to alternately retrieve the butterfly’s upper and lower input data. Hence, bit i can be evaluated by correlated with an alternating signal. Note that the phase of this square wave is unknown, since the latency of the address generator is unknown. It is possible for a correctly alternating bit i to have zero correlation with a reference square wave if it is out of phase. Therefore, the fitness contribution of bit i should depend on how close the correlation is 0 or 100%.

A large problem at the implementation level is coupling the GA with a Verilog simulation environment. The simulation output data must also be rapidly retrievable by the GA program for evaluation. Since the DPU models are in Verilog, simple use of SystemC is not a solution. (Like JHDL, SystemC provides C++ classes with HDL functionality.) However, the NC-Verilog environment may be able to cosimulate SystemC with Verilog. Alternatively, the Verilog program-

ming language interface (PLI) can be investigated for this purpose. Finally, tools are available that convert Verilog to SystemC; their completeness and standards conformance need investigation.

E.4 Using GAs to Discover Good Subcircuits

E.4.1 Genome Representation for Netlists of Multiport Components

The chromosome representation for a netlist of DPUs determines what elements can be combined between parents during crossover, as well as the kind of mutation that can be applied. Since this dictates how information is mixed and leveraged during the search, it is central to the success of the GA. As mentioned in Chapter 6, the genotype can take the traditional form of a vector of genes, or a more complex form such as the netlist graph in Figure 5.1. Both schemes must be able to handle the fact that the final solution contains an unknown number of DPUs.

The information in a netlist graph can always be *encoded* as a vector, albeit of arbitrarily complex genes and possibly variable length. However, one must then address the question of how to do this so that 2-point or uniform crossover conveys meaningful information to the offspring. Aside from choosing vertices from two parents to form offspring, rules must also be devised so meaningful information is conveyed in constructing edges to reconnect the two subgraphs. The nature of “meaningful information” is very specific to the problem i.e. much more specific than simply focusing on directed hypergraphs; it matters whether the graphs represent, for example, neural nets or reconfigurable ALUs, since they are impacted in a different manner when changing the set of vertices, edges, or behavioural characteristics of the vertices.

The area of genotype representation in evolutionary computation is vast. This section covers some of the schemes that seem most promising for a netlist of configurable ALUs. None of these schemes address the nonmemoryless nature and cycle-by-cycle reconfigurability of our targeted archetypal platform.

It is worth noting that the evolutionary computing community has drawn a distinction between two strains of search problems: *optimization* versus *discovering the architecture or structure* of a (good) solution. In optimization, the form of the solution is already known e.g. the vertices and

edges in a graph, be it a neural net or an analog circuit. All that remains is the determination of the values of circuit components, or the weights of the edges. Thus, the solution is well represented by a fixed-length vector, usually of scalar parameters. Search space approaches for these problems are relatively mature.

For problems in which the structure of the solution is also being sought, the number of vertices and edges, and the interconnection pattern, for a good solution must be discovered. It is in this realm that complex genotypes are needed.

There are a number of characteristics specific to the DPU netlist which must be accommodated by any genotype scheme. The first is that the nets are directed hypergraphs. This complicates the edge specification, as well as reconnection heuristics that connect fragments of two parents to form a child.

The second characteristic is that the many different inputs to each DPU are not interchangeable; each of the two operand arms is fed by a wide input mux. The exact signal selected by each operand arm is determined by the DPU personality, and can vary from cycle to cycle. Despite the wide input muxes, the problem can be abstracted to a simpler form. This is because the wide mux is partially intended to deal with placement problems i.e. a different mux setting is required depending on the *physical position* of the driving DPU. Prior to placement, a subcircuit might consist of only 2 to 4 DPUs, and each DPU might have its input muxes driven by about the same number of sources (not all of which originate from within the subcircuit).

In the following discussion, it is important to note the distinction between a hypergraph node and a circuit node. A hypergraph node corresponds to a DPU, whereas a circuit node (or net or wire) corresponds to a hyperedge in the hypergraph. To avoid confusion, we avoid using the term *node* to refer to a vertex in a hypergraph. Similarly, we will avoid the use of *node* to refer to a circuit node; for this, the terms *net*, *wire*, *connection*, or *interconnect* are clearer.

E.4.2 Vector Chromosomes

E.4.2.1 Fixed Length Chromosomes

A simple way to deal with the fact that the final solution uses an unknown number of DPUs is to assume that all solutions consist of some maximum number of DPUs e.g. 4 DPUs. For each solution found, the mapping algorithm should then identify DPUs that are never used. This identification is necessary to recognize resource-infeasible solutions e.g. if the sum of DPUs from all

subcircuits for a kernel exceeds the available number of DPUs. Identification of unused DPUs is also needed to guide the search toward solutions that use fewer DPUs, since larger subcircuits can be penalized. To identifying unused DPUs, the chromosome evaluation must search for DPUs that are never connected to the subgraph which generates the output. We will refer to this subgraph as the *output subgraph*.

It is not straightforward to determine which DPUs are part of the output subgraph. This is because the interconnection pattern can change with each clock cycle, depending on how the input muxes are set by the DPU personalities. Therefore, it is necessary to more loosely define what is meant by connected DPUs, as well as what is meant by a subgraph. The criterion we now describe is a crude first attempt at a meaningfully defining these concepts in a dynamic environment. It is a recursive and constructive definition in that the vertices making up the output subgraph are determined by repeatedly examining the interconnections for all the clock cycles that the kernel runs. During this construction process, “nonsubgraph vertices” will refer to vertices that haven’t yet been incorporated into the output subgraph under construction.

Construction of the output subgraph is started with seed vertices, defined as those DPUs that drive the outputs of the FSM. Next, any DPUs that feed data to these seed vertices at any time during kernel execution are the incorporated into the output subgraph. In turn, any nonsubgraph vertices that feed these nonseed vertices at any time are also incorporated into the subgraph. This process is repeated until there is no further change in the subgraph.

It should be obvious that this is an overly generous guess at which DPUs belong in the output subgraph. Due to the time varying nature of the connections, it is possible that some DPUs within the output subgraph actually have no effect on the output of the subcircuit. Even if the connections were not time varying, it is still possible for some DPUs within the output subgraph to be irrelevant. One scenario in which this can happen is if the data supplied by these DPUs are not used at the destination DPUs. For example, the operation at a destination DPU might use only one of two operands, or it may use the feedback path, or it may use the registers to hold the input or output constant.

NetKeys

Network random keys encoding (NetKeys) [RGH02,SRP02] does not deal with variable number of vertices, but it does attempt to find a good set of edges between vertices to form a tree (as in graph theory, not the rooted tree of hierarchical data structures). For a graph of N vertices, there

are $N(N-1)/2$ possible edges, but a tree only has $N-1$ edges. NetKeys find the best $N-1$ edges by assigning a random key to each of the $N(N-1)/2$ candidate edges, thus forming a chromosome of fixed length $N(N-1)/2$. Each key represents the importance of the associated edge. A tree is formed from a chromosome as follows. The candidate edges are sorted by key, as in RKR, which ranks the candidate edges in order of importance. The ranked candidate edges are sequentially scanned, each edge being added to an initially edgeless graph. Any edges that form a ring are discarded as they are encountered, since that prevents a tree from forming. The tree is formed when $N-1$ undiscarded edges have been placed, which terminates the chromosome decoding process.

The evolutionary framework used in [RGH02,SRP02] was evolution strategies.

Adaptions for DPU Netlists

Using NetKeys as a starting point, there are various possibilities for adapting a fixed length chromosome for nontree graphs, as required for a netlist of DPUs. For example, edges can be retained even though they form a cycle. A variable number of edges can be used from the list of ranked candidate edges, with the number of edges determined from an additional gene e.g. $N_{\text{edge}} = s_{\text{edge}} N_{\text{edgeMax}}$, where the extra gene $s_{\text{edge}} \in [0.0, 1.0]$ is a scaling factor, and N_{edgeMax} is the maximum number of edges allowed for the solution. N_{edgeMax} might be allowed to rise or fall with the generations, depending on whether the population's average fitness improvement is stalling.

The idea of using a gene to specify a variable number of edges can be extended to genetically encode the variable number of vertices: $N_{\text{vertices}} = s_{\text{vertices}} N_{\text{verticesMax}}$, where $s_{\text{vertices}} \in [0.0, 1.0]$ is a scale factor gene, and $N_{\text{verticesMax}}$ is the maximum number of vertices allowed for a solution. Only the N_{vertices} most important vertices, as determined by their random keys, would be part of the solution. However, the information for all $N_{\text{verticesMax}}$ DPUs must be carried by all chromosomes, since N_{vertices} can change during crossover or mutation. Maintaining information for all N_{vertices} DPUs also keeps the chromosome length constant, thus simplifying crossover.

It should be remembered, however, that crossovers based on vector chromosomes may not preserve important building blocks, particularly when there are many interconnections between a few DPUs (the two inputs of each DPU have wide input muxes). This is because there is limited scope for complex building blocks to be compact when each gene only has two adjacent neighbours.

Keeping all N_{vertices} DPUs in the chromosome also simplifies the specification of edges. An edge is normally associated with source and destination vertices, or an ordered pair of DPU indices. If the number of DPU changes, so does their enumeration. This changes the meaning of each edge, effectively disrupting much of the interconnect information. It is possible to keep the DPU indices the same by assigning them with permanent tags. This idea is applied to edges (rather than vertices) in the NEAT system described in Section E.4.2.2.

With all N_{vertices} DPUs firmly attached to identity tags, it is necessary to discard edges that connect to DPUs that are not included in the solution. If these discarded edges are included in the count of N_{edge} edges to use, then it introduces variability into the number of actual edges used. In that case, it may not be necessary to actually accommodate a nonconstant N_{edge} .

E.4.2.2 Variable Length Vector Chromosomes

If the variable number of DPUs is addressed by using a variable length chromosome, then *evolutionary programming* can be used to avoid the complications of crossover, as mentioned in Chapter 6. Let us briefly examine the implications of using mutation only, without crossover.

In Chapter 6, mutation without crossover was loosely viewed as brownian motion of the population of chromosomes in N -space, where N is the chromosome length. For simplicity, assume that N is the number of DPUs, and that a DPU in the phenotype is encoded as a complex gene in the genome (Figure E.2). For now, we ignore the details of whether the various personalities and their timing are embedded in the DPU's gene or outside of it (as additional genes). Likewise, we ignore the question of how the interconnect information is to be included. Based on intuition or experimentation, one can choose a population size to be a reasonable number of chromosomes. If the population consists of chromosomes with different lengths, however, they don't all sample the same space. For example, some chromosomes may sample 3-space while others may sample 4-space. All of a sudden, it matters *how many* chromosomes sample 3-space versus 4-space or 5-space, etc. as evolution progresses. Furthermore, without crossover, there is no combining of information from the chromosomes in the different spaces. This can be alleviated somewhat by

allowing chromosomes to migrate between spaces e.g. via mutations that change the number of DPUs in the solution.



Figure E.2. An extremely simple vector genotype for a DPU netlist.

It can be argued that an equivalence can be drawn between the points in 3- or 4-space with points in 5-space e.g. all 5-DPU chromosomes having the same output subgraph correspond to the same point in n -space, where n is the number of DPUs in the output subgraph. As shown in Section E.4.2.1, however, the identification of an output subgraph can be subjective; thus, the equivalence between points in different spaces can be tenuous at best, and of questionable value at worst.

Because of the inconsistencies in the view of stochastic sampling in N -space, one is left instead with the more intuitive view of chromosomes as collections of building blocks. It is this view, however, that most strongly indicates the need for crossover. Mutation is not enough; the picture of brownian motion breaks down when chromosomes are strewn across disparate spaces, with possibly too few sampling points in some spaces for reliable coverage. To prevent this isolation between search spaces, information should be mixed between spaces via crossover and combining of building blocks. This also facilitates the migration of chromosomes between spaces by the way of variable length offspring. For variable length chromosomes, therefore, it is worthwhile to consider genotypes that allow crossover rather than just relying on evolutionary programming.

We have presented intuitive reasons for which crossover is important when using variable length chromosomes. One can take an opposing view based on parallels between sampling of different dimensionality spaces versus sampling of different hyperplanes in a single search space for the case of fixed length chromosomes. In the latter, mutation without mating suffices to thoroughly explore the solution space by pushing sampling points from one set of hyperplanes to the next. For variable length chromosomes, we can reasonably expect that potentially size-altering mutations will adequately push sampling points between different dimensionality spaces to thoroughly explore the search spaces. Therefore, it seems arbitrary and unnecessary to distinguish between search spaces based on dimensionality. However, this view presumes that size-changing mutations are common place. In contrast, there is a good reason to restrict most of the mutations

to perturbations that don't change the DPU count. Since DPUs are large complicated units, adding or removing DPUs are revolutionary rather than evolutionary changes.

Messy GAs (mGAs)

Messy GAs and their variations [GKD89,DG91,KG00,DG00,GDKH93] use variable length chromosomes, but are meant for problems with *fixed size phenotypes*. We present it here to point out its unsuitability, despite initially perceptions of its suitability.

In mGAs, there is an unspoken fixed length genotype, where the position (or locus) in the chromosome determines a gene's meaning. There is a bijective mapping between genes and problem parameters (each gene maps to a unique problem parameter, and vice-versa). This "golden" chromosome serves only as a dictionary, and does not undergo evolution; neither are any values assigned to its genes.

The actual population consists of abominations of the golden chromosome, or "nongolden" chromosomes. Each nongolden chromosome consists of a varying number of genes, where each gene consists of a locus tag and a gene value. The tag indicates which gene in the golden chromosome will be given the value. A nongolden chromosome may have more than one gene with the same tag (a conflict, or *overspecification*); it may also have zero genes with a particular tag value (incomplete chromosome, or *underspecification*).

In decoding a nongolden chromosome, various conflict resolution schemes are presented for tags that occur more than once e.g. first occurrence of a tag supersedes all others. Values for missing tags are obtained from a *template* chromosome, which is just a locally optimum set of gene values i.e. a complete but not overspecified gene found from local search.

Crossover is performed by probabilistically cutting and splicing chromosome segments from two parents. The probabilities that govern these encourage creation of longer strings from shorter strings, and dissuade the creation of even longer strings from already long strings.

Aside from the requirement for a fixed length golden chromosome, one of the difficulties with mGAs was its presumption that the fitness was the linear combination of partial fitnesses, each corresponding to a *nonoverlapping* building block. This limits the kind of epistasis that can be formed between genes.

SAGA

In SAGA [Har92b,Har92a], variable length vector chromosomes were used more as an illustration of how complex life forms tend to arise rather than to solve an engineering problem. Artificial

partial fitnesses were specified for predefined building blocks. Total fitness was devised in such a way as to favour longer chromosomes. It was shown that evolution caused chromosomes to grow only if growth was constrained to be incremental rather than large steps. This illustrated that mating between *similar* organisms makes more sense than blind application of crossover, since good building blocks are expected to have similar meanings in similar contexts.

NEAT

NEAT [SM02b,SM02a] evolves neural networks, both in terms of topology and edge weights. The chromosome consists of an array of vertices and an array of edges. Edges are specified not only by source and destination vertex indices, but also by an enable bit, which determines whether the edge is used, as well as an *innovation number* (IN). In a mimicry of gene alignment prior to crossover in nature, the IN ensures that only corresponding edges can be swapped between parents i.e. only edges with the same IN can be swapped.

The IN uniquely identifies an edge in terms of the vertices it connects i.e. if an edge with IN i connects vertices j and k in one chromosome, then it does so in all chromosomes in which it appears, in all generations. Thus, the IN takes the vertex enumeration as reference points to enforce swapping between similar edges. This makes sense for neural nets, where there is not much difference between most vertices i.e. there is not much reason to uniquely distinguish between vertices with IN numbers.

Since the vertex enumeration acts as reference points for alignment of edges, it is important that IN numbers be maintained for nodes as well as edges. This is much more critical for DPU netlists than for neural nets. Many of the vertices in a neural net are identical i.e. neural net's behaviour is determined largely by the interconnection pattern and weights. The only caveat is that the vertices must be marked as either input, output, or hidden. Within each category, the vertices do not need to be distinguished, and they can be exchanged without changing the neural net. DPUs cannot be treated alike, however, since each DPU can have a different set of personalities. Each DPU (vertex) carries much more information than a 3-way categorization. Care must be taken to maintain their distinction during crossover in order to maximally preserve building blocks. Since DPUs differ greatly, it is important to maintain IN numbers for DPUs. (In personal communications, the originator of NEAT clarified that node IN numbers were in fact used to distinguish between nodes).

After corresponding genes are aligned between parents, genes for which there are counterparts in both parents (*matching* genes) are “mixed” in that each gene value for the child is taken from a randomly selected parent i.e. a coin is tossed for each matching gene. Each parent will also have a set of nonmatching genes. The child inherits the nonmatching genes from the fitter parent. Unlike more conventional crossovers, it is not necessary to generate two complementary children.

Mutation is performed by splitting an edge with a new vertex. Again, this makes sense for neural nets, since it perturbs the topology, which is the main determinant of neural net behaviour. In contrast, DPUs are high level constructs, and introducing a new DPU into a netlist of few DPUs is a revolutionary change. This kind of mutation should be kept to a minimum. Perturbing a DPU’s personalities should be the main form of mutation.

One of the discoveries in developing NEAT was the necessity to restrict crossover between dissimilarly structured chromosomes. This is in line with SAGA’s conclusions. In NEAT, the rationale was that new topologies initially cause a drop in fitness. Any improvement in fitness came about after evolution was given some time to find the best way to use the new genetic material by combining it with other topologies. In order to give the new structures the opportunity to do this, chromosomes were restricted in competing for representation within niches. Similar chromosomes were grouped into species if their difference (linear combination of Hamming and Euclidean/manhattan distances) fell below some threshold.

To protect the size of each species, a simplified version of *shared fitness* was adapted from [Gol89]. Shared fitness was developed for multimodal fitness landscapes to ensure that the high-est optimum did not eventually attract all the chromosomes, as would be the case in conventional GAs. In shared fitness, a chromosome’s fitness is attenuated by the amount of crowding in its vicinity of genotype space, as measured by various intuitively motivated metrics. This reduction in fitness imposes a self-limiting effect on the attraction of chromosomes to high fitness regions, and prevents the migration of all chromosomes to the highest peak. The result is that each peak, or niche/species, is populated with a density of chromosomes commensurate with the species’s average fitness. The implementation in NEAT used much more concretely defined boundaries between species than in [Gol89], but was computationally more expedient.

Most of the mating was between members of the same species, with a low probability of inter-species mating. Within each species, the top 40% of the chromosomes were repeatedly mated to generate enough offspring to replace the entire species. In keeping with the self limiting effect due

to crowding, the size of each species was expected to grow or shrink in proportion to the species average shared fitness relative to the entire population's average shared fitness.

The ability of GAs to discover resource efficient solutions is often called *parsimony*. NEAT avoids penalizing the chromosome size in achieving parsimony. The reason is that the penalty is tricky to calibrate, and can compromise correct functionality. NEAT avoids excessively large graphs by starting with minimal graphs and letting fitness decide whether size-increasing mutations survive. This strategy must be viewed with caution in the case of DPU netlists. The reason is that DPU netlists do not have the same degree of gradual progression in fitness as graph complexity increases to the right level.

E.4.3 Nonvector Chromosomes

E.4.3.1 Genetic Programming

Genetic programming (GP) [KIAK99] was developed to evolve functional language programs initially in the form of a parse tree. The genotypic tree represents instructions on how to construct the phenotypic program. This tree based recipe was adapted to construct analogue circuits, thus allowing circuits to be designed via genetic search. The circuit construction process starts off with a minimal circuit consisting of a single wire component. Each step in the tree based recipe replaces or adds to a component in the circuit until all instructions have been executed. In this way, the phenotype is developed from a single wire into a complete circuit. The nets in the circuit are not directed. Heuristics are devised to connect complex multiport devices into the circuit in place of a simple 2-port component during the development process.

Crossover between parent parse trees is performed by swapping subtrees. There is complete compatibility because each subtree returns a single value to the parent vertex in the parse tree.

There are three approaches that can be taken in adapting GP to DPU netlist evolution. One is to interpret the parse tree as an acyclic graph, and modify the procedure to accommodate nonacyclic directed hypergraphs. This is not a trivial step, and we discuss some difficulties to rule out this option. The second possibility is to interpret the parse tree as a hierarchical design tree, or a *cluster tree*, for the DPU netlist, in the same manner as the coarsening algorithms of [KAKS97, KK99, CW03] in Section 5.1.4. Unfortunately, this does nothing to address the complexities of compatible subgraphs from two parent graphs, to be stitched together into an offspring graph. The final approach is to encode the development of the phenotypic circuit as a parse tree of

instructions, similar to the way it is done for analogue circuits above. This approach is not without its problems, too.

The first approach treats the parse tree as an acyclic graph. However, a tree is simply too restrictive a representation for general graphs. There is simply no way to draw an equivalence between the acyclic graph and a nonacyclic graph. In reality, the leaf vertices of the tree are not leaf vertices in the DPU netlist, but simply regular vertices with directed edges that feed into other vertices. It is possible that an acyclic graph can be generated by cutting the regular graph at appropriate edges, but this cut is arbitrary and has no intuitive bases. There is also the problem of how to convert the tree back into a nonacyclic graph after crossover. Even assuming the absence of cycles in the netlist, each vertex of a tree always has a single incoming edge and potentially multiple outgoing edges; no vertex has multiple incoming edges. There is no guarantee that this property is satisfied by a graph, even if cycles were somehow removed.

The second approach treats the parse tree as a clustering tree. That is, all DPUs correspond to leaf vertices. Each nonleaf vertex does not correspond to a DPU; instead, it represents the aggregation of all the DPUs in the leaf vertices of its subtree. Basically, the nonleaf vertex represents the aggregation of all its children vertices, some of which may be leaf vertices while others may be aggregation of leaf vertices and nonleaf vertices. The edges between a nonleaf vertex and its children vertices represent a subsumption relationship rather than physical interconnect. In fact, the physical interconnect does not appear in the clustering tree at all. Because of the coarsening algorithms that create the cluster tree, the DPUs in the subtree of each nonleaf vertex form a subgraph with a somewhat minimized cut set. Therefore, the clustering tree may help delineate a good subgraph of DPUs for crossover, but it does nothing to ensure the cutest is compatible between the two parents. Neither does it help determine which edges to connect between the subgraphs from two parents in forming the offspring.

The following are some less-than-elegant remedial measures that can be taken in the case of incompatible cut sets between subgraphs to be connected together to form a child graph. Since the actual DPU netlist is a hypergraph, some ad hoc heuristics can be formed to resolve incompatible cut sets e.g. by connecting one signal driver to several destinations, in the case where there are more edge destinations on one subgraph than there are edge sources in the other subgraph. If there are more edge sources on one subgraph than there are destinations in the other subgraph, there is the possibility of feeding some sourced signals back to the same subgraph, if there are destinations available. Note also that the child graph does not become infeasible if some of the extra edge

sources are not connected to any destinations. Alternatively, incompatible cut sets can be avoided by trying to find nodes with matching cut sets in the 2 parents.

This elaboration on reconciling incompatible cut sets does highlight the converse problem in evolving a netlist of configurable DPUs. That is, what happens to the DPU inputs that are not explicitly driven. This does not necessarily arise from cut set incompatibilities during crossover. Some DPU personalities can cause these inputs to be read. As mentioned previously, however, the synthesis algorithm will deal with an abstraction of the DPU which has fewer inputs. The abstraction can be designed such that it is illegal to have mux inputs that are not driven. The actual physical platform likely will never encounter this problem, since all inputs of its very wide mux are most likely driven by something (if even an unused DPU output, ground, or VDD).

The third and final approach of using a GP parse tree is probably the least speculative and most promising. A DPU netlist can be built up in the same manner in which an analog circuit is developed using the parse tree instructions. The complication here, however, is that the circuit building procedure seems to be meant for analogue circuits i.e. the interconnection of many components that don't have many ports. The many rules for incorporating even 3, 4, and 5-port devices become very complex, and still do not take into account the fact that the DPU netlist uses directed hyperedges i.e. two sources cannot be connected to the same net, and all components are "unilateral" in the sense that stimuli at the outputs do not propagate to the inputs. The complex rules and contingencies arises from

- ◆ the different types of devices with different port counts that can replace or connect to a 2-port device
- ◆ the different ways to split up a net before a 2-port device can be replaced by an N -port device during construction of the circuit
- ◆ the many different ways to bind each port of an N -port device to the nets in the circuit

It may be unfair to view this complexity as a drawback, since the genotypes discussed so far for handling graphs do not distinguish between different input terminals to a DPU. All incoming edges are merely "incident" to the vertex. These genotypes may also become unwieldy when appropriately modified to maintain separate identities for the different input terminals.

E.4.3.2 EGG

Evolutionary generation of graphs (EGG) [AHH99,HAH02,CAH+02,HAMH03] works directly with a graph netlist rather than a vector or tree chromosome. Subgraphs with compatible cut sets are exchanged between parents. There is no need for the parent graphs to have the same size, and there are no requirement for corresponding vertices or edges between parents to be aligned. However, in order for EGG to leverage the concept of compatible cut sets to identify crossover subgraphs, the method is restricted to directed graphs rather than directed hypergraphs. That means each net is point-to-point.

To speed up the crossover, each graph chromosome carries with it information about a collection of pre-identified subgraphs and the cut set properties. These subgraphs are randomly delineated. Parents are mated only if their chromosomes contain subgraphs with compatible cut sets. It is not clear if the subgraphs for each chromosome are nonoverlapping subgraphs. If they overlap, some subgraphs may be fragmented by crossover, thus requiring that post-crossover step to ensure the integrity of all subgraphs in an offspring e.g. by re-delineating a new collection of subgraphs, or by checking all subgraphs and modifying the fragmented ones.

The integration of subgraphs from two parents requires that connections be made along the compatible cut sets. Within this cut set, there is some arbitrariness in which sources of one subgraph connect to which destinations in the other subgraph. The required connections are obvious from examining the connection matrix for the child graph, prior to creating connections between its subgraphs. A greedy algorithm is used to assign connections in the matrix in a way that balances the outstanding source/destination requirements. Algorithm details are not provided.

The level of abstraction for the components represented by graph vertices is bit-level boolean logic. Circuits designed by the evolutionary search include acyclic arithmetic circuits. The extent of nonmemorylessness was that of a pipeline i.e. due to latency rather than recursive feedback, as would be the case in (say) an IIR filter. Designs were evaluated by generating and simulating the corresponding Verilog.

E.5 Summary

The mapping between algorithm and kernel can be thought of in the 3 phases: partitioning into kernels, detailed design of a kernel based on the platform resources, and kernel scheduling. This

chapter discussed mapping of kernels to platform resources, which will be the focus in the remaining work for this thesis.

The form of the input specifications for the kernel design is intimately related to how much of the design task will be performed by an automated synthesis tool. A microcode-like specification leaves much of the work to the designer, since it includes details of function binding to DPU components, as well as clustering of functions into DPUs. Automated clustering in the datapath can lead to an overly complex control path, since each DPU personality is defined for the entire DPU rather than for components internal to the DPU.

The approach here will be functional specification at the subcircuit level. The initial kernel functionality will be assumed to be in the form of a DFG, properly serialized or parallelized for the target platform and desired throughput. The designer then partitions the DFG into a higher-level DFG of subcircuits, completely specifying the behaviour at the boundaries between subcircuits.

Unlike evolution of static circuits, the evolution of a kernel's subcircuit is greatly complicated by the fact that each DPU will alternate between 8 personalities. A chromosome must include information about when each personality takes effect in each of potentially many clock cycles. This complexity will be dealt with by restricting the number of personalities to be repeated over a 1 to 3 cycle period e.g. 1 to 3 personalities in repeated order for the bulk of the kernel execution.

One possibility in addressing the complexity is to aggregate the personalities for multiple DPUs into superpersonalities. This allows building blocks to form across DPUs with synchronized personality changes. However, it increases the search space with infeasible chromosomes. Furthermore, the synchronicity of personality changes across multiple DPUs might not be achievable due to the latency of the DPUs.

To evaluate the fitness of subcircuits, they may be written out in Verilog form for simulation. An efficient means of cooperation between the GA and the simulator must be established. Alternatively, to simulate within the GA program, a means of translating the Verilog DPU models to SystemC can be sought. The models are cycle accurate behavioural models, and the simulation need only be cycle accurate.

The fitness function needs to provide hints to guide the search. The designer must determine attributes of a subcircuit's output signals to reward.

To deal with a netlist of DPUs, any genome representation must uniquely identify the I/O terminals of a DPU. Ideally, the GA can search among solutions with varying sizes, or DPU counts.

Furthermore, a nonvector genome would allow greater flexibility for graphs, since building block formation is not limited by adjacency/proximity along a single dimension of gene arrangement.

Fixed length vectors genomes can still represent variable sized subcircuits by fixing the length to accommodate the largest solution allowed or expected. However, it is necessary to find the subcircuit components that do not affect the subcircuit. This is not straightforward to do by analyzing the interconnect pattern of the graph while it is being simulated. To determine whether a DPU actually affects a subcircuit, it may be necessary to redo simulations with and without the DPU being tested. This must be done for each DPU in the subcircuit. Furthermore, once a DPU is found to be removable without affecting the subcircuit's output behaviour, all tested DPUs must be retested. Since this is computationally costly, this test should be restricted only to feasible solutions that fully achieve the desired output behaviour. Parsimonious solutions can then be rewarded with higher fitness.

It is worthwhile to note that maintaining a long, fixed-length chromosome may frustrate the search for a solution of shorter length. For example, if a 3-DPU solution exists, it may be hard to find by searching for 4-DPU solutions. The extra DPU must be carefully programmed in order not to interfere with the correct operation of the 3-DPU solution. This is just a concrete way of seeing that the search space is broadened with unsuitable solutions.

NetKeys provides some ideas than can be used fixed length chromosome encoding of graphs. The random keys for the DPUs and edges represent their importance. Extra genes may be needed to determine how many of the most important vertices/edges to use. Its fixed length means that all unused DPUs/interconnect are retained in the chromosome, but it simplifies crossover. This is because DPU enumeration doesn't change, which minimizes disruptions in the interconnect.

For variable length vector genomes, a gene's significance is no longer associated with its locus in the vector. To ensure that a gene in a "father" genome is meaningfully swapped with a "corresponding" gene in the "mother" genome, genes need to be tagged. Intuitively, long arms may be useful in a world with tall coconut trees, but it does no good for the extra length information to impart a long nose to the offspring.

For similar reasons, crossover is more effective between similar chromosomes, whether the similarity is in length, as in SAGA, or by the smallness of some difference metric. For variable sized solutions, of course, the difference metric must span multiple dimensionality spaces, as it does in NEAT.

The complications in crossover between different size genomes have traditionally been avoided by using mutation alone. However, the case for crossover is that of avoiding the isolation of information in different dimensionality spaces.

Some variable length genotypes, such as mGAs, are not suitable for variable size netlists because the phenotype is fixed length. In contrast, NEAT allows variable length phenotypes in representing neural nets. Matching edges between parents are randomly combined to form the offspring. Nonmatching edges are taken from the fitter parent. Nodes are taken from the parents as they are require inheriting edges. Parsimony is achieved by starting with minimal neural nets, incrementally expanding some genomes by mutation, and letting fitness determine whether they survive.

NEAT used “speciation” to promote mating between similar genomes and to protect newly generated structures until they have a chance to optimize through evolution. Speciation was implemented by sharing the fitness between the members of a species, thus discouraging crowding in any one niche. Speciation is potentially more important for DPU netlists because of the large difference in the subcircuit with the removal or addition of a single DPU. This is directly related to their coarse grain nature. For this reason, we believe that most of the evolution should come from changing the configuration of DPUs rather than the topology of the netlist.

Genetic programming is another approach for variable size circuits. Its rooted tree structure represents the program to build a circuit. Crossover is simplified because the tree does not represent the netlist graph. In fact, there are many difficulties to trying to take advantage of this simplicity by drawing parallels between the rooted tree and the netlist graph. However, there are also issues to resolve in applying the GP circuit build procedure to a DPU netlist. There are many ad-hoc rules for fitting multi-port devices in place of 2-port devices or wires. Furthermore, the modifications are needed for the build procedure to respect the unilateral nature of digital circuits.

EKG was the final approach reviewed. Its encoding scheme has the advantage of distinguishing between the input and output ports of circuit components (as does genetic programming). However, it needs to be modified to handle directed hypergraphs. Moreover, it was intended to handle larger graphs of fine grain vertices (bit level logic). This allows more subgraphs to be delineated within each parent, and a greater chance for matching cut sets when mating.

As can be seen, a genotype for DPU netlists require features from various schemes developed for other applications. The most outstanding are variable size and topography, the need to handle directed hypergraphs in stitching the subgraphs from the parents to from the offspring, and the

need to maintain separate identities for the various DPU ports. Topology can be handled by the DPU personalities of a fully connected netlist, while variable DPU count can be handled by Net-Keys's priority encoding scheme, along with an additional priority threshold gene. The genome must maintain information about all the DPU personalities, as well as how they will sequence in time. An reduced abstraction of the DPU will be used to constrain the possibilities and manage the complexities. Evaluation can be done by integration with a commercial simulator, or by System C, with suitable conversion of the DPU models.

The initial scheme should handle mutation only, with crossover as a possible expansion. The test circuit would be the address generator for the butterfly data, the most complex piece from the FFT kernel. The foundation of this design methodology would be laid by obtaining correct functionality for 1 FFT stage, with expansion to all 10 stages as a next step.

References

- [AAW+96] S.P. Amarasinghe, J.M. Anderson, C.S. Wilson, Shih-Wei Liao, B.R. Murphy, R.S. French, M.S. Lam, and M.W. Hall. Multiprocessors from a software perspective. *IEEE Micro*, 16:52–61, June 1996.
- [AHH99] T. Aoki, N. Homma, and T. Higuchi. Evolutionary design of arithmetic circuits. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A:798, 1999.
- [Alp98] C. J. Alpert. The ISPD98 circuit benchmark suite. In *Proceedings of 1998 International Symposium on Physical Design (ISPD-98)*, pages 80–85, 1998.
- [Ann] Annapolis Micro Systems, Inc. *Wildforce Reference Manual*.
- [Ant89] H. J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In J. Schaffer, editor, *proceedings of the third international conference on genetic algorithms*, pages 86–91, San Mateo, CA, 1989. Morgan Kaufmann.
- [ARNL+03] M. Ayala-Rincon, R.B. Nogueira, C.H. Llanos, R.P. Jacobi, and R.W. Hartenstein. Modeling a reconfigurable system for computing the fft in place via rewriting-logic. In *Proceedings 16th Symposium on Integrated Circuits and Systems Design. SBCCI 2003*, page 205, 2003.
- [BA97] R. Bittner and P. Athanas. Wormhole run-time reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 79–85, Monterey, CA, 1997.
- [BAK96] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. Wiley-IEEE Press, May 1996.
- [BAM96] R. A. Bittner, Jr., P. M. Athanas, and M. D. Musgrove. Colt: an experiment in wormhole runtime reconfiguration. In John Schewel, editor, *High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic, Proc. SPIE 2914*, pages 187–195, Bellingham, WA, 1996. SPIE – The International Society for Optical Engineering.
- [BDD+99] Kiran Bondalapati, Pedro Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: a design environment for adaptive computing technology. In *6th Reconfigurable Architectures Workshop (RAW 99)*, page 570, San Juan, Puerto Rico, April 1999. Part of Parallel and Distributed Processing. 11th IPSP/SPDP'99 Workshops. Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. Proceedings.
- [BG99] M. Budiu and S. C. Goldstein. Fast compilation for pipelined reconfigurable

- fabrics. In S. Kaptanoglu and S. Trimberger, editors, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 195–205, Monterey, CA, February 1999. ACM Press.
- [BH98] Peter Bellows and Brad Hutchings. JHDL—an HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, page 175, April 1998.
- [Bit97] Ray Bittner. *Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System*. PhD thesis, Virginia Polytechnic Institute and State University, 1997.
- [Bli00] Tobias Blickle. Tournament selection. In David Fogel, Thomas Back, and Zbigniew Michalewicz, editors, *Evolutionary Computation 1*. Institute of Physics Publishing, 2000.
- [BM00] Allen L. Barker and Worthy N. Martin. Dynamics of a distance-based population diversity measure. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 1002–1009, La Jolla Marriott Hotel La Jolla, California, USA, July 2000. IEEE Press.
- [BMSK03] S.N.R. Borra, A. Muthukaruppan, S. Suresh, and V. Kamakoti. A parallel genetic approach to the placement problem for field programmable gate arrays. In *International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.
- [Bon01] Kiran Bondalapati. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In *Design Automation Conference*, pages 273–276, 2001.
- [BP98] Kiran Bondalapati and Viktor K. Prasanna. Mapping loops onto reconfigurable architectures. In Reiner W. Hartenstein and Andres Kevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 268–277. Springer-Verlag, Berlin, / 1998.
- [BP99] Kiran Bondalapati and Viktor K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 249–258, Los Alamitos, CA, 1999. IEEE Computer Society Press.
- [BP00] Kiran Bondalapati and Viktor K. Prasanna. Loop pipelining and optimization for run time reconfiguration. In *Reconfigurable Architecture Workshop (RAW 2000)*, Cancun, Mexico, May 1 2000.
- [BRM+99] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, page 70, Napa Valley, CA, 1999.

- [Bur01] Alister Burr. *Modulation and Coding for Wireless Communications*. Prentice-Hall, 2001.
- [CAH+02] D. Chen, T. Aoki, N. Homma, T. Terasaki, and T. Higuchi. Graph-based evolutionary design of arithmetic circuits. *IEEE Transactions on Evolutionary Computation*, 6:86, 2002.
- [Car01] J.M.P. Cardoso. Novel algorithm combining temporal partitioning and sharing. In *Proceedings of the Ninth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, California, April 29 – May 2 2001.
- [CCDW98] T. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, Monterey, CA, 1998.
- [CFBE98] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and compiling applications for RaPiD. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 116–125, Los Alamitos, CA, 1998. IEEE Computer Society Press.
- [CFF+96] Darren C. Cronquist, Paul Franklin, Chris Fisher, Miguel Figueroa, and Carl Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1996.
- [CG99] Srihari Cadambi and Seth Copen Goldstein. CPR: A configuration profiling tool. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 104–113, Los Alamitos, CA, 1999. IEEE Computer Society Press.
- [Cha] Chameleon Systems, Inc. *CS2000 Reconfigurable Communications Processor Family Product Brief*.
- [Cha00] Chameleon Systems, Inc. *IFFT1K-TP CS2000: 1K-point DIT In-Place FFT Data Sheet*, September 20 2000.
- [Cha01a] Chameleon Systems, Inc. *CS2112 Reconfigurable Communications Processor Data Book Version 1.3*, July 2001.
- [Cha01b] Chameleon Systems, Inc. *CS2112 Reconfigurable Communications Processor User Manual Version 1.3*, July 2001.
- [CHW00] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The garp architecture and C compiler. *IEEE Computer*, 33(4):62, April 2000.
- [Cla97] Chris Cladingboel. Real java virtual machines; hardware compilation and the java virtual machine. M.Sc. project report Candidate J1925, Oxford University Computing Lab, September 5 1997. Available at <http://www.wadham.ox.ac.uk/~chris/project> and <http://www.hummingbird-it.com/>

- chris/project. Not registered with university's Libraries, possibly not a thesis document.
- [CSP98] S. Crago, B. Schott, and R. Parker. SLAAC: A distributed architecture for adaptive computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 286–287, Napa Valley, CA, April 15-17 1998.
- [CW98] Timothy J. Callahan and John Wawrzynek. Instruction-level parallelism for reconfigurable computing. In Reiner W. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm*, LNCS 1482, pages 248–257, Tallinn, Estonia, September 1998. Springer-Verlag, Berlin.
- [CW03] Yongseok Cheon and M.D.F. Wong. Design hierarchy-guided multilevel circuit partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:420, 2003.
- [CWG+98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas. Managing pipeline-reconfigurable FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55–64, Monterey, CA, 1998.
- [DeH94] Andr'e DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Los Alamitos, CA, 1994. IEEE Computer Society Press.
- [DeH95] Andr'e DeHon. Notes on coupling processors with reconfigurable logic. Technical Report Transit Note#118, MIT Transit Project, March 1995.
- [DeH96] Andr'e DeHon. Reconfigurable architectures for general-purpose computing. Technical Report AITR-1586, MIT Artificial Intelligence Laboratory, 1996.
- [DG91] Kalyanmoy Deb and David E. Goldberg. mGA in C: A messy genetic algorithm in C. Technical Report 91008, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, September 1991.
- [DG00] Kalyanmoy Deb and David E. Goldberg. OMEGA – ordering messy GA: Solving permutation problems with the fast messy genetic algorithm and random keys. Technical Report 2000004, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, January 2000.
- [DHP+03] P. Diniz, M. Hall, Joonseok Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the defacto system. In *Languages and Compilers for Parallel Computing. 14th International Workshop, LCPC 2001. Revised Papers*, page 52, 2003.
- [ECF97] C. Ebeling, D.C. Cronquist, and P. Franklin. Configurable computing: the

- catalyst for high-performance architectures. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 364–72, 1997.
- [ECFF96] Carl Ebeling, Darren C. Cronquist, Paul Fraklin, and Chris Fisher. RaPiD – reconfigurable pipelined datapath. In *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996.
- [ECF+97] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 106–115, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [EN02] A. Ekart and S.Z. Nemeth. Maintaining the diversity of genetic programs. In *Genetic Programming. 5th European Conference, EuroGP 2002*, pages 162–171, 2002.
- [FF96] D.B. Fogel and L.J. Fogel. An introduction to evolutionary programming. In *Artificial Evolution European Conference, AE'95*, pages 21–33, 1996.
- [FG96] David B. Fogel and Adam Ghozeil. Using fitness distributions to design more efficient evolutionary computations. In *International Conference on Evolutionary Computation*, pages 11–19, 1996.
- [FH91] C.W. Fraser and D.R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29, 1991.
- [FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing, 1995.
- [Fou84] L. R. Foulds. *Combinatorial Optimization for Undergraduates*. Springer-Verlag, New York, 1984.
- [Gar97] *The Garp Architecture*, October 1997. This is the Garp manual, available at <http://brass.cs.berkeley.edu/documents/GarpArchitecture.html>.
- [GB98] K. M. GajjalaPurna and D. Bhatia. Temporal partitioning and scheduling for reconfigurable computing. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, page 329, 1998.
- [GC97] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms and Engineering Design*. John Wiley & Sons, Inc., New York, NY, 1997.
- [GD91] D. E. Goldberg and Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, CA, 1991.
- [GDKH93] David E. Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges Harik. Rapid accurate optimization of difficult problems using fast messy genetic algorithms. In Stephanie Forrest, editor, *Proc. of the Fifth Int. Conf. on*

-
- Genetic Algorithms*, pages 56–64, San Mateo, CA, 1993. Morgan Kaufmann.
- [GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer, 1992.
- [GHK+91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24:81–89, January 1991.
- [GKD89] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.
- [GN95] P. Graham and B. Nelson. A hardware genetic algorithm for the traveling salesman problem on splash 2. In W. Moore and W. Luk, editors, *Fifth International Workshop on Field Programmable Logic and Applications*, pages 352–361, Oxford, England, August 1995. Springer.
- [Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [GSAK00] Maya Gokhale, Janice Stone, Jeff Arnold, and Mirek Kakinowaki. Stream-oriented FPGA computing in the Streams-C high level language. In *8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, pages 49–56, Marriott at Napa Valley, Napa, California, April 17-19 2000. Formerly available at <http://rcc.lanl.gov/sw/Streamc/Welcome.html>.
- [GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. Pipherench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, April 2000.
- [HAA+96] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.
- [HAH02] N. Homma, T. Aoki, and T. Higuchi. Graph-based individual representation for evolutionary synthesis of arithmetic circuits. In *Proceedings of 2002 World Congress on Computational Intelligence - WCCI'02*, page 1492, 2002.
- [HAMH03] N. Homma, T. Aoki, M. Motegi, and T. Higuchi. A framework of evolutionary graph generation system and its application to circuit synthesis. In *ISCAS 2003. International Symposium on Circuits and Systems*, 2003.
- [Har92a] I. Harvey. Species adaptation genetic algorithms: a basis for a continuing SAGA. In F. J. Varela and P. Bourguine, editors, *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems*, pages 346–354, Paris, France, 1992. MIT Press, Cambridge, MA.
- [Har92b] Inman Harvey. The SAGA cross: The mechanics of recombination for

- species with variable-length genotypes. In Reinhard Männer and Bernard Manderick, editors, *Parallel problem solving from nature 2*, pages 269–278, Amsterdam, 1992. North-Holland.
- [HB91] Frank Hoffmeister and Thomas Bäck. Genetic algorithms and evolution strategies: Similarities and differences. In *Parallel Problem Solving from Nature (First Workshop)*, page 455, 1991.
- [He98] Yingchun He. VLSI implementation of run-time configurable computing integrated circuit – the stallion chip. Master’s thesis, Virginia Polytechnic Institute and State University, 1998.
- [HFHK97] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffery P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffery Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’97)*, pages 87–96. IEEE Computer Society Press, 1997.
- [HFHK04] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12:206, 2004.
- [HGRC96] Michael D. Hutton, J. P. Grossman, Jonathan Rose, and Derek G. Corneil. Characterization and parameterized random generation of digital circuits. In *Design Automation Conference*, pages 94–99, 1996.
- [HHHN99] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping applications onto reconfigurable KressArrays. In *9th International Workshop on Field Programmable Logic and Applications (FPL’99)*, Glasgow, UK, Aug. 30 – Sept. 2 1999.
- [HHHN00a] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Kressarray explorer: a new CAD environment to optimize reconfigurable datapath array architectures. In *Proceedings of ASP-DAC2000: Asia and South Pacific Design Automation Conference 2000*, page 163, 2000.
- [HHHN00b] R. Hartenstein, M. Herz, Th. Hoffmann, and U. Nageldinger. Generation of design suggestions for coarse-grain reconfigurable architectures. In *Proceedings of FPL 2000. 10th International Conference on Field Programmable Logic and Applications*, page 389, 2000.
- [HHR+91] Reiner W. Hartenstein, Alexander G. Hirschbiel, Michael Riedmüller, Karin Schmidt, and Michael Weber. A novel ASIC design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7):975–989, July 1991.
- [HHW90] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber. Xputers: very high throughput by innovative computing principles. In *Proceedings of the 5th Jerusalem Conference on Information Technology (JCIT). Next Decade in Information Technology (Cat. No.90TH0326-9)*, page 43, 1990.

- [HK95] R.W. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Asia and South Pacific Design Automation Conference (ASP-DAC) 95*, page 479, 1995.
- [HKR94] R.W. Hartenstein, R. Kress, and H. Reinig. A reconfigurable data-driven ALU for xputers. In *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*, page 139, 1994.
- [HL90] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Mathematical Programming*. McGraw-Hill Publishing Company, 1990.
- [HLP+02] F. Herrera, M. Lozano, E. Perez, A.M. Sanchez, and P. Villar. Multiple crossover per couple with selection of the two best offspring: an experimental study with the blx- alpha crossover operator for real-coded genetic algorithms. *Advances in Artificial Intelligence - IBERAMIA 2002. 8th Ibero-American Conference on AI*, page 392, 2002.
- [HLV97] F. Herrera, M. Lozano, and J.L. Verdegay. Fuzzy connectives based crossover operators to model genetic algorithms population diversity. *Fuzzy Sets and Systems*, 92:21, 1997.
- [HM02] Z. Huang and S. Malik. Exploiting operation level parallelism through dynamically reconfigurable datapaths. In *Design Automation Conference*, June 2002.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Number ISBN 0-13-153271-5 in Computer Science. Prentice Hall International, 1985. Highly theoretical.
- [Hol75] J. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [HRB+99] Jeffrey Hammes, Bob Rinker, A. P. Wim Bohm, Walid A. Najjar, Bruce A. Draper, and J. Ross Beveridge. Cameron: High level language compilation for reconfigurable systems. In *IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 236–244, Newport Beach, CA, 1999.
- [HRC97] Michael D. Hutton, Jonathan Rose, and Derek G. Corneil. Generation of synthetic sequential benchmark circuits. In *FPGA*, pages 149–155, 1997.
- [HSC+00] Simon D. Haynes, John Stone, Peter Y.K. Cheung, and Wayne Luk. Video image processing with the sonic architecture. *Computer*, 33(4):50, April 2000.
- [HW97] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 12–21, Los Alamitos, CA, April 16-18 1997. IEEE Computer Society Press.

- [IC99] R. Ivimey-Cook. Legacy of the transputer. In *Architectures, Languages and Techniques for Concurrent Systems. WoTUG-22. Proceedings of the 22nd World Occam and Transputer User Group Technical Meeting*, page 197, 1999.
- [IH94] K. Iwama and K. Hino. Random generation of test instances for logic optimizers. In *Proceedings of 31st ACM/IEE Design Automation Conference*, page 430, 1994.
- [IHKS97] Kazuo Iwama, Kensuke Hino, Hiroyuki Kurokawa, and Sunao Sawada. Random benchmark circuits with controlled attributes. In *Proceedings European Design and Test Conference. ED & TC 97*, page 90, 1997.
- [JGP96] Dimitris G. Manolakis John G. Proakis. *Digital Signal Processing*. Prentice-Hall, Upper Saddle River, NJ, third edition, 1996.
- [KAKS97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. In *34th ACM/IEEE Design Automation Conference*, pages 526–529, 1997.
- [KG00] D. Knjazew and D.E. Goldberg. Large-scale permutation optimization with the ordering messy genetic algorithm. In *Proceedings of 6th International Conference on Parallel Problem Solving from Nature*, page 631, 2000.
- [KH03] S. Kittitornkun and Yu Hen Hu. Mapping deep nested do-loop DSP algorithms to large scale FPGA array structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11:208, 2003.
- [KIAK99] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Kean. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.
- [KK99] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *ACM/IEEE Design Automation Conference*, pages 343–348, 1999.
- [KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.
- [KT03] S. Krishnamoorthy and R. Tessier. Technology mapping algorithms for hybrid FPGAs containing lookup tables and PLAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:545, 2003.
- [KVG099] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan, and Iyad Ouass. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *Design Automation Conference*, pages 616–622, 1999.
- [LCD02] Jong-eun Lee, Kiyoun Choi, and Nikil Dutt. Mapping loops on coarse-grain reconfigurable architectures using memory operation sharing. In *First Workshop on Application Specific Processors (WASP-1)*, Istanbul, Turkey, Nov. 19 2002.

- [LKH02] J.D. Lohn, W.F. Kraus, and G.L. Haith. Comparing a coevolutionary genetic algorithm for multiobjective optimization. In *Proceedings of 2002 World Congress on Computational Intelligence - WCCI'02*, page 1157, 2002.
- [LS91] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.
- [LST00] Jian Liang, Sriram Swaminathan, and Russell Tessier. aSOC: A scalable, single-chip communications architecture. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 37–46, October 2000.
- [LTS99] Ronald Laufer, R. Reed Taylor, and Herman Schmit. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 200–208, Los Alamitos, CA, 1999. IEEE Computer Society Press.
- [LW98] Huiqun Liu and D. F. Wong. Network flow based circuit partitioning for time-multiplexed FPGAs. In *ICCAD*, pages 497–504, 1998.
- [LWR99] Suet-Fei Li, M. Wan, and J. Rabaey. Configuration code generation and optimizations for heterogeneous reconfigurable DSPs. In *1999 IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation*, page 169, Tapei, Taiwan, October 1999.
- [MADF04] Mitchell J. Myjak, Fredrick L. Anderson, and Jos'e G. Delgado-Frias. H-tree interconnection structure for reconfigurable DSP hardware. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, pages 170–176, Las Vegas, NV, June 21–24 2004. CSREA Press.
- [MD96] Ethan Mirsky and Andr'e DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Los Alamitos, CA, April 17-19 1996. IEEE Computer Society Press.
- [ME95] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of Association for Computing Machinery International Symposium on Field Programmable Gate Arrays (FPGA '95)*, page 111, 1995.
- [Met97] Chris Metcalf. The newmesh simulator, nsim v3. Systems Memo 24, Laboratory of Computer Science, MIT Computer Architecture Group, August 26 1997. Available via <ftp://cag.lcs.mit.edu/pub/numesh/memos/nsim.ps.Z>.
- [Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin Heidelberg, third, revised and extended

- edition, 1996.
- [MKF+01] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernandez, Roman Hermida, Nader Bagherzadeh, and Hartej Singh. A framework for reconfigurable computing: task scheduling and context management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, December 2001.
- [MKP02] Fred Ma, John Knight, and Calvin Plett. Reconfigurable logic design case. In *Fourth SPIE Conference on Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications (part of ITCOM 2002)*, volume 4867, pages 113–126, July 30 2002.
- [MKP04] F. Ma, J. P. Knight, and C. Plett. Physical resource binding for a coarse grain reconfigurable array. In *Engineering of Reconfigurable Systems and Algorithms*, June 21-24 2004.
- [MKPss] Fred Ma, John P. Knight, and Calvin Plett. Physical resource binding for a coarse grain reconfigurable array using evolutionary algorithms. *IEEE Transactions on Very Large Scale Integrated Systems*, (accepted).
- [MM02] T.W. Manikas and M.H. Mickle. A genetic algorithm for mixed macro and standard cell placement. In *Midwest Symposium on Circuits and Systems*, page II, 2002.
- [MR99] Pinaki Mazumder and Elizabeth M. Rudnick. *Genetic Algorithms for VLSI Design, Layout & Test Automation*. Prentice Hall, Upper Saddle River, NJ, 1999.
- [MS97] Laurent Moll and Mark Shand. Systems performance measurement on PCI pamette. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 125–133, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [MY03] Wai-Kei Mak and E.F.Y. Young. Temporal logic replication for dynamically reconfigurable FPGA partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:952, July 2003.
- [OS89] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [Oua01] Steve Oualline. *Vi Improved–Vim*. New Riders Publishing, Indianapolis, IN, April 2001.
- [OW99] F. Oppacher and M. Wineberg. The shifting balance genetic algorithm: improving the GA in a dynamic environment. In *Proceedings GECCO-99. Genetic and Evolutionary Computation Conference. Eighth International Conference on Genetic Algorithms (ICGA-99) and the Fourth Annual Genetic Programming Conference (GP-99)*, page 504, 1999.
- [Oxf] Formerly available at <http://oldwww.comlab.ox.ac.uk/oucl/groups/hwcweb/general/papers.html>.

- [Pag96] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12:87, 1996. INSPEC says the above. My survey says vol. 4, no. 10, pp. 1339-1354.
- [Par94] J. Paredis. Co-evolutionary constraint satisfaction. In *Parallel Problem Solving from Nature - PPSN III International Conference on Evolutionary Computation The Third Conference on Parallel Problem Solving from Nature*, page 46, 1994.
- [Per] <http://www.perldoc.com>.
- [PK89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.
- [PLM00] J. Pistorius, E. Legai, and M. Minoux. Partgen: a generator of very large circuits to benchmark the partitioning of fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1314, 2000.
- [PV99] Awartika Pandey and Randga Vemuri. Combined temporal partitioning and scheduling for reconfigurable architectures. In John Schewel, Peter M. Athanas, Steven A. Guccione, Stefan Ludwig, and John T. McHenry, editors, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 93–103, Bellingham, WA, 1999. SPIE – The International Society for Optical Engineering.
- [Rab97] J.M. Rabaey. Reconfigurable processing: the solution to low-power programmable DSP. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, page 275, Munich, April 1997.
- [RCA] No publications can be found on RCA-2, but technical papers are available at <http://www.lanl.gov:80/rcc/hw/index.html>.
- [RGH02] F. Rothlauf, D.E. Goldberg, and A. Heinzl. Network random keys - a tree representation scheme for genetic and evolutionary algorithms. In *Evolutionary Computation*, volume 10, page 75, 2002.
- [RH97] Michael Rencher and Brad L. Hutchings. Automated target recognition on SPLASH 2. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 192–200, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [SC01] Bill Salefski and Levent Caglar. Re-configurable computing in wireless. In *Design Automation Conference*, pages 178–183, 2001.
- [Sch97] Herman Schmit. Incremental reconfiguration for pipelined applications. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [SGV01] V. Srinivasan, S. Govindarajan, and R. Vemuri. Fine-grained and coarse-

- grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9:140, 2001.
- [Sha01] Akshay Sharma. Development of a place and route tool for the RaPiD architecture. Master's thesis, University of Washington, 2001.
- [Sha03] Akshay Sharma. Piperoute: A pipelining-aware router for FPGAs. In *International Symposium on FPGAs (FPGA'03)*, Monterey, CA, 2003.
- [SLL+00] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.
- [SM02a] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, San Francisco, CA, 2002. Morgan Kaufmann.
- [SM02b] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
- [Spe93] William M. Spears. Crossover or mutation? In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 221–237. Morgan Kaufmann, San Mateo, CA, 1993.
- [SR95] Hans-Paul Schwefel and Gunter Rudolph. Contemporary evolution strategies. In *European Conference on Artificial Life*, pages 893–907, 1995.
- [SRA+00] Srikathyayani Srikanteswara, Jeffrey H. Reed, Peter Athanas, and Robert Boyle. A soft radio architecture for reconfigurable platforms. *IEEE Communications Magazine*, 38(2):140–147, February 2000.
- [SRP02] B. Schindler, F. Rothlauf, and H.-J. Pesch. Evolution strategies, network random keys, and the one-max tree problem. In *Applications of Evolutionary Computing. EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIME/EvoPLAN. Proceedings*, page 143, 2002.
- [Ste98] Steve Trimberger. Scheduling designs into a time-multiplexed FPGA. In *Proceedings of FPGA 98. 1998 International Symposium on Field Programmable Gate Arrays*, page 153, 1998.
- [SV97] V. Schneck and O. Vornberger. Hybrid genetic algorithms for constrained placement problems. *IEEE Transactions on Evolutionary Computation*, 1:266, 1997.
- [SVC00] Dirk Stroobandt, Peter Verplaetse, and Jan Van Campenhout. Generating synthetic benchmark circuits for evaluating CAD tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1011–

- 1022, September 2000.
- [TA00] X. Tang and M. Aalsma. A compiler directed approach to hiding configuration loading latency in chameleon reconfigurable chips. In *10th International Conference on Field Programmable Logic and Applications (FPL'2000)*, Villach, Austria, 2000.
- [TB01] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1):7–27, June 2001.
- [TMJ+99] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, George Varghese, John Wawrzynek, and Andre DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. In *FPGA*, pages 125–134, 1999.
- [Tor99] Elie Torbey. *Control/Data Flow Graph Synthesis Using Evolutionary Computation and Behavioural Estimation*. PhD thesis, Carleton University, 1999.
- [Tow] Tower of Power supercomputer architecture based on commercial Annapolis Micro Systems WildForce reconfigurable computing boards. Information formerly available at <http://www.ccm.ece.vt.edu/slaac/>.
- [Tur00] Jim Turley. Flexible cores optimize architecture. *EE Times*, (1126):94, August 14 2000.
- [TVFG03] Marco Tomassini, Leonardo Vanneschi, Francisco Fernández, and Germán Galeano. Diversity in multipopulation genetic programming. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1812–1813, Chicago, 12-16 July 2003. Springer-Verlag. GECCO-2003. A joint meeting of the twelfth International Conference on Genetic Algorithms (ICGA-2003) and the eighth Annual Genetic Programming Conference (GP-2003).
- [Urs02] Rasmus K. Ursem. Diversity-guided evolutionary algorithms. In *Proceedings of Parallel Problem Solving from Nature VII (PPSN-2002)*, pages 462–471. Springer Verlag, 2002.
- [Vai01] M. Michael Vai. *VLSI Design*. CRC Press, 2001.
- [VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.
- [VNK+03] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm, and J. Hammes. Automatic compilation to a coarse-grained reconfigurable

- system-on-chip. In *ACM Trans. on Embedded Computing Systems (to be published)*. ACM, November 2003.
- [VSC02] P. Verplaetse, D. Stroobandt, and J. Van Campenhout. Synthetic benchmark circuits for timing-driven physical design applications. In *International Conference on VLSI (VLSI'02)*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 24 - 27 2002.
- [WH95] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In John Schewel, editor, *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, pages 92–103, Bellingham, WA, 1995. SPIE – The International Society for Optical Engineering.
- [WH96] M. J. Wirthlin and B. L. Hutchings. Sequencing run-time reconfigured hardware with software. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 122–128, Monterey, CA, 1996.
- [Whi89] Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufman.
- [Whi94] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.
- [Whi02] Darrell Whitley. Genetic algorithms and evolutionary computing. In *Van Nostrand's Scientific Encyclopedia*. Wiley, 2002.
- [WILR98] M. Wan, Y. Ichikawa, D. Lidsky, and J. Rabaey. An energy conscious methodology for early design exploration of heterogeneous DSPs. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, page 111, Santa Clara, CA, May 1998.
- [WO03] Mark Wineberg and Franz Oppacher. The underlying similarity of diversity measures used in evolutionary computation. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1493–1504, Chicago, 12-16 July 2003. Springer-Verlag.
- [WZBR99] M. Wan, Hui Zhang, M. Benes, and J. Rabaey. A low-power reconfigurable data-flow driven DSP system. In *IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation*, Taipei, Taiwan, October 1999.
- [WZG+01] M. Wan, Hui Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey. Design methodology of a low-energy reconfigurable single-chip DSP system. *Journal of VLSI Signal Processing Systems for Signal, Image,*

-
- and Video Technology*, 28(1):47–61, May 2001. Available via, http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures/papers.htm l, <http://www.kluweronline.com/issn/0922-5773/contents>.
- [Yan] Reconfigurable Architectures Yanbing. Hardware-software co-design of embedded.
- [YSB00] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *FPGA*, pages 95–100, 2000.
- [YTFFY03] M. Yoshikawa, H. Terai, T. Fujita, and H. Yamauchi. A novel timing-driven placement using genetic algorithm. In *2003 Southwest Symposium on Mixed-Signal Design*, page 237, 2003.
- [ZB00] Ning Zhang and R.W. Brodersen. Architectural evaluation of flexible digital signal processing for wireless receivers. In *Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, page 78, Pacific Grove, CA, October 2000.
- [ZPG+00] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J.M. Rabaey. A 1-v heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing. *IEEE Journal of Solid-State Circuits*, 35(4):1697–1704, November 2000.
- [ZWGR99] Hui Zhang, Marlene Wan, V. George, and J. Rabaey. Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs. In *IEEE Computer Society Workshop on VLSI*, page 2, Orlando, FL, 1999.