Control/Data Flow Graph Synthesis using Evolutionary Computation and Behavioral Estimation

by
Elie Torbey

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

Doctor of Philosophy

Department of Electronics
Faculty of Engineering
Carleton University
January 1999

© copyright
1999, Elie Torbey

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-37080-1

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

Control/Data Flow Graph Synthesis using Evolutionary Computation and Behavioral Estimation

submitted by Elie Torbey
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

Chair, Department of Electronics

Thesis Supervisor

External Examiner

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Faculty of Engineering
Carleton University
January 1999

Acknowledgment

I would like to give special thanks to my supervisor Professor John Knight for his patience, insight and encouragement throughout the work on this thesis. I thank him for accommodating my schedule, tolerating my ideas and putting me back on track every once in a while. None of this work would have been possible without his help.

I also would like to thank my friends and family for their help, patience and understanding.

-

To my mother
and my father

Table of Contents

Table of Contents	vi
List of Figures	xv
List of Tables	xix
List of Symbols and Acronyms	xxi
Chapter 1. Introduction	1
1.1. Design automation and high-level synthesis	1
1.1.1. Thesis Statement	2
1.2. Evolutionary Computation and Global Optimization	2
1.3. Complexity estimation and controller area	3
1.4. Contributions	4
1.5. Thesis outline	5
Chapter 2. High-Level Synthesis Background	7
2.1. Introduction	7
2.2. High-Level Synthesis Overview	8
2.2.1. Definition	8
2.2.2. High-Level Synthesis in the IC design cycle	9
2.2.3. Complexity	9
2.3. High-level synthesis steps	11
2.3.1. Allocation and Binding	12
2.3.2. Scheduling	13
2.3.3. Interdependence of the steps	13
2.4. Design descriptions and constraints	14
2.4.1. Design behavioral specification	14

2.4.2. Architectural descriptions	15
2.4.3. Constraints	16
2.5. Scheduling	17
2.5.1. ASAP/ALAP scheduling	18
2.5.2. List scheduling	19
2.5.3. Force-directed scheduling	21
2.5.4. Other Heuristic Schedulers	23
2.5.5. Neural Networks scheduling	23
2.5.6. Simulated annealing scheduling	23
2.5.7. Integer Linear Programming scheduling.	24
2.5.8. Control flow schedulers	25
2.6. Scheduling and synthesis issues	26
2.6.1. Pipelining	26
2.6.2. Chaining	26
2.6.3. Multi-cycling	27
2.6.4. Loop folding/unwinding	28
2.6.5. Relative scheduling	29
2.6.6. Controller synthesis	30
2.7. Control-scheduling issues	30
2.7.1. Mutual exclusion	30
2.7.2. Out-of-order execution	30
2.7.3. CDFG synthesis tools	31
2.8. Coloring Techniques	32
2.8.1. Node Coloring Algorithm	32
2.8.2. Condition Vector method	33
2.8.3. Control Tag Method	35
2.8.4. Comparison of the methods	37
2.9. Architectural issues in high-level synthesis	38
2.9.1. Multi-rate systems	38
2.9.2. Clock rate synthesis	38
2.9.3. Varying bit-widths	41

2.9.4. Design style and other issues	41
2.10. Conclusion	42
-	
Chapter 3. Evolutionary Computation Background	44
3.1. Introduction	44
3.2. Evolutionary Algorithms	45
3.2.1. Optimization	45
3.2.2. Algorithms	46
3.3. Genetic Algorithm definitions and basics	48
3.3.1. Genetic algorithms	48
3.3.2. Chromosome	49
3.3.3. Symbol	49
3.3.4. Population	50
3.3.5. Crossover	50
3.3.6. Mutation	52
3.3.7. Reproduction	52
3.3.8. Replacement schemes	54
3.3.9. Initial population generation	54
3.3.10. Fitness	55
3.3.11. Scaling	55
3.3.12. Dominance, Diploidy and Abeyance	56
3.3.13. Inversion	56
3.3.14. Infeasible solutions	56
3.3.15. Isomorphism	57
3.4. Optimization using GAs compared to other common methods	58
3.4.1. Schemata	58
3.4.2. Schema Theorem	59
3.5. Genetic algorithm synthesis survey	60
3.5.1. San Martin's work	60
3.5.2. Heijlingers' work	61
3.5.3. Problem Space GAs	62

3.5.4. Simulated Evolution approach	62
3.6. Conclusion	63
Chapter 4. Spectral Techniques and Complexity Estimation	65
4.1. Introduction	65
4.2. Control Unit Synthesis	65
4.2.1. Microcode control	66
4.2.2. Hard-Wired Control Synthesis	66
4.3. Spectral techniques	67
4.3.1. Definition	68
4.3.2. Complexity measures	68
4.3.3. Auto-correlation method	69
4.3.4. The entropy method	72
4.4. Handling incompletely specified functions	74
4.5. High-level Synthesis and a New complexity measure	74
4.6. Conclusion	75
Chapter 5. Genetic Algorithm Synthesis	76
5.1. Introduction	76
5.2. Basic genetic algorithm synthesis	77
5.2.1. Problem formulation	77
5.2.2. CDFG description and input	77
5.2.3. Hardware availability and cost description	82
5.2.4. Genetic algorithm setup	82
5.2.4.1. Encoding	83
5.2.4.2. Relative clock cycles	85
5.2.4.3. Evaluation of the encoding	88
5.2.4.4. Chromosome decoding	91
5.2.4.5. GA operators	91
5.2.4.6. GA inputs/outputs	92
5.3. Functional/Storage Units	93

5.3.1. Example F/SUs	94
5.4. Objective function calculation	95
5.4.1. From bit strings to architectures	95
5.4.2. Chromosome Decoding	97
5.4.3. Area cost	97
5.4.3.1. FU area	97
5.4.3.2. SU area	99
5.4.4. Power cost	100
5.4.5. Latency cost	101
5.4.6. Cost Function or fitness function	102
5.5. Control Synthesis	103
5.5.1. Different Control Synthesis Options	103
5.5.2. CDFG coloring	104
5.5.3. Color updating	107
5.5.4. Coloring Pitfall	108
5.5.4.1. Color-coding method	108
5.5.4.2. Path-Tracing method	110
5.5.5. Non Series-Parallel Graphs	112
5.5.6. Data-dependent Loops	113
5.6. Conclusion	114
Chapter 6. Architectural Optimizations and Heuristics	116
6.1. Introduction	116
6.2. Pipelining, chaining and multi-cycling	117
6.2.1. Pipelining	117
6.2.2. Multi-cycling	118
6.2.3. Chaining	118
6.3. Template matching and varying bit-widths	119
6.3.1. Template matching	119
6.3.2. Varying bit-widths	120
6.4. Clock period selection and synthesis	120

6.4.1. GA Clock selection	120
6.4.2. Multi-rate systems	122
6.5. Multi-path clocking	123
6.5.1. Path-dependent selection example	123
6.5.2. Path-dependent selection proof	124
6.5.3. Implementation of path-dependent selection	127
6.6. Mobility calculation	128
6.6.1. ASAP to ALAP	129
6.6.2. Fixed or fixed increment	130
6.6.3. Unbounded Relative clock cycle decoding	130
6.6.4. ASAP to ARAP	130
6.7. Storage and Interconnect Optimization	131
6.7.1. CDFG modification	132
6.7.2. Interconnect and Routing Optimization	134
6.8. Hybridization of GA synthesis	135
6.8.1. The design-space reduction method	135
6.8.2. Proof of superiority of the reduction method	136
6.8.3. Force-directed scheduling using the reduction method	140
6.8.4. Suggested evolutionary approach using the reduction method	141
6.9. Conclusion	142
Chapter 7. Synthesis Benchmarks and Results	143
7.1. Introduction	143
7.2. Quality and Speed of Synthesis	144
7.2.1. The Elliptic Wave Filter	144
7.2.1.1. Benchmark Description	144
7.2.1.2. EWF results	144
7.2.1.3. Design space exploration	147
7.3. Quality of Conditional Synthesis	148
7.3.1. Kim's CDFG example	149
7.3.2. The Rotor example	151

7.4. Large Circuit Synthesis	153
7.4.1. S2R	154
7.4.2. Discrete Cosine Transform	154
7.4.2.1. DCT Description	154
7.4.2.2. Modified Arai's algorithm	156
7.4.2.3. Kovac's Implementation	158
7.4.2.4. Time-Shared Implementation	159
7.4.2.5. Architectural enhancements	160
7.4.2.6. 2-D DCT computation	162
7.4.2.7. High-level synthesis exploration	163
7.4.2.8. Performance comparison	165
7.5. Clock Selection and Synthesis	166
7.5.1. EWF results	166
7.5.2. Path-dependent clock selection	168
7.6. Synthesis with Multiple Units	170
7.7. Conclusion	172
Chapter 8. GA Evaluations	174
8.1. Introduction	174
8.2. Selection schemes	176
8.2.1. Steady-state versus Generational	176
8.2.2. Two different steady-state methods	176
8.3. The effect of GA operators	177
8.3.1. The effect of the mutation rate	177
8.3.2. The effect of the crossover rate	178
8.3.3. The effect of crossover	179
8.4. Scaling effect	181
8.5. Crossover position	185
8.6. Simulated Annealing versus GAs	185
8.7. Multi-objective functions	189
8.7.1. Multi-objective function with Haploid GAs	190

8.7.2. Multi-objective Functions with Diploid GAs	194
8.7.3. Separate populations (multi-culturalism)	198
8.8. Conclusion	200
Chapter 9. The Gradient Method for Complexity Estimation	202
9.1. Introduction	202
9.2. Gradient technique, an intuitive view	203
9.2.1. Gradient method	203
9.2.2. Why it works	204
9.2.3. State machine decomposition	206
9.3. The Gradient technique and other representations	207
9.3.1. Definition	207
9.3.2. Implementation	208
9.3.3. Gradient with sorted rows/columns	209
9.3.4. Example	209
9.4. Benchmark Results	213
9.4.1. Gradient versus Autocorrelation	215
9.4.2. Gradient versus literals	215
9.4.3. FSM size estimation	219
9.5. Comparison with Entropy	221
9.6. Permutations	224
9.6.1. Permutation of input and output variables	224
9.6.2. Permutation of variables and patterns	227
9.7. Conclusion	228
Chapter 10. Conclusion and Future Work	229
10.1. Conclusion	229
10.2. Contributions	229
10.3. Future Work	231
Bibliography	233

APPENDIX A: Algorithms and Derivations	242
A.1. List scheduling algorithm	242
A.2. Simulated Annealing algorithm	242
A.3. GA Synthesis algorithms	243
A.4. Second order derivative	245
 APPENDIX B: DCT Details	 248
B.1. Distributed arithmetic implementation	248
B.2. Data transfers	248
B.3. Scheduled 8-point DCT DFG	251
B.4. Resource scheduling	253
B.5. 2-D DCT	254

List of Figures

Figure 2.1: Typical design flow using high-level synthesis.	10
Figure 2.2: Benefits of simultaneous storage allocation and scheduling	14
Figure 2.3: Example algorithm and corresponding CDFG.	15
Figure 2.4: Example synthesized architecture	17
Figure 2.5: Example ASAP schedule	18
Figure 2.6: Example ALAP schedule	19
Figure 2.7: Example urgency	20
Figure 2.8: Mobility and distribution graph example	21
Figure 2.9: Pipelining Example.	26
Figure 2.10: Chaining example.	27
Figure 2.11: Multi-cycling example.	27
Figure 2.12: Loop unwinding example.	28
Figure 2.13: Loop folding/unwinding example.	29
Figure 2.14: Example is out-of-order execution of conditional operations	31
Figure 2.15: Example coloring using Park's method	33
Figure 2.16: Example coloring using Condition Vectors	35
Figure 2.17: Example coloring using the control tag method	36
Figure 2.18: Comparison of the three coloring methods	37
Figure 2.19: Effect of clock period on synthesis results	39
Figure 3.1: Evolutionary Algorithm	46
Figure 3.2: Sample Genetic Algorithm	48
Figure 3.3: Chromosome encoding for the GA example.	50
Figure 3.4: Single-point crossover example.	51
Figure 3.5: Multi-point crossover	52
Figure 3.6: Roulette wheel selection	53
Figure 3.7: Baker's multi-arm roulette wheel selection	53
Figure 3.8: Inversion operator example	57
Figure 5.1: Example Control/Data Flow Graph.	78

Figure 5.2: Chromosome Encoding for multi-rate systems	84
Figure 5.3: Example chromosome encoding.	85
Figure 5.4: General Functional/Storage Unit description	94
Figure 5.5: Example decoded chromosomes	96
Figure 5.6: Chromosome decoding and performance evaluation.	98
Figure 5.7: Chromosome splitting example	104
Figure 5.8: Out-of order representation using chromosome splitting	105
Figure 5.9: Coloring example: pre-schedule	106
Figure 5.10: Coloring example: post-schedule	108
Figure 5.11: Color-coded method of coloring	110
Figure 5.12: Path-Trace method of coloring	111
Figure 5.13: Non series-parallel graphs	112
Figure 5.14: Data-dependent loop representation	113
Figure 6.1: Chromosome encoding with clock selection	122
Figure 6.2: Typical multi-rate chromosome encoding.	123
Figure 6.3: Example Path-dependent Clock Selection	124
Figure 6.4: Path-based objective functions	128
Figure 6.5: Mobility example for ASAP, ALAP and ARAP.	129
Figure 6.6: Example CDFG modification: storage operation insertion.	133
Figure 6.7: Different types of SUs.	133
Figure 6.8: Example register assignment and resulting topology.	134
Figure 6.9: Example mobility based encoding	141
Figure 7.1: Elliptic Wave Filter Data Flow Graph	145
Figure 7.2: Population fitness of a typical EWF run. (Steady State)	148
Figure 7.3: Elliptic Wave Filter Data Flow Graph	149
Figure 7.4: Kim's CDFG example	150
Figure 7.5: ROTOR algorithm	152
Figure 7.6: ROTOR CDFG	153
Figure 7.7: S2R CDFG	155
Figure 7.8: Modified Fast DCT showing operations	157
Figure 7.9: Arai's 1-D DCT implementation	158

Figure 7.10: 1-D DCT Time-shared architecture	159
Figure 7.11: Resource utilization table	160
Figure 7.12: Table for 2 consecutive initiations of 8 points	161
Figure 7.13: Resource utilization for 4x8 points	162
Figure 7.14: HLS convergence for the DCT example	164
Figure 7.15: CDFG example	168
Figure 7.16: Example area/delay trade-off curves for the GA runs	172
Figure 8.1: Steady-state vs. generational	177
Figure 8.2: Replacing below average or “average & below”	178
Figure 8.3: Mutation rate effect	179
Figure 8.4: Crossover rate effect	180
Figure 8.5: Zero crossover rate experiment	181
Figure 8.6: Zero crossover rate experiment with larger design space	182
Figure 8.7: Zero crossover rate experiment with larger design space	183
Figure 8.8: Linear scaling	184
Figure 8.9: Scaling versus no scaling.	184
Figure 8.10: Crossover location	186
Figure 8.11: Crossover location II	187
Figure 8.12: Simulated annealing vs. GA results I	188
Figure 8.13: Simulated annealing vs. GA results II	189
Figure 8.14: Simulated annealing vs. GA results III	190
Figure 8.15: Simulated annealing vs. GA results IV	191
Figure 8.16: Multi-modal objective function $1/\text{area} + 4/\text{delay}$	192
Figure 8.17: Environmental change with $f_1 = 1/\text{totalarea}$ and $f_2 = 4/\text{totaldelay}$	194
Figure 8.18: Environmental change experiment with the combined fitness f_{ec}	195
Figure 8.19: Diploid with no change of dominance nor environmental change	195
Figure 8.20: Diploid with a dominance change at 50 but no environmental change. ..	196
Figure 8.21: Diploid with a dominance change 50 and environmental change at 50 ..	197
Figure 8.22: Diploid with a dominance change at 18 but no environmental change. ..	198
Figure 8.23: Diploid with a dominance change 18 and environmental change at 18 ..	199
Figure 8.24: Comparison of (a) unified and (b) separate populations	200

Figure 9.1: Cartesian representation of finite differences	206
Figure 9.2: FSM example	210
Figure 9.3: Example encoding	210
Figure 9.4: Gradient and Autocorrelation estimate vs. the reduced SOP	216
Figure 9.5: Gradient and literals with respect to the reduced SOP	217
Figure 9.6: Gradient/pattern vs. lits with respect to SOP	217
Figure 9.7: Gradient/pattern vs. lits with respect to SOP for small circuits	218
Figure 9.8: Gradient/pattern vs. lits/pattern for small circuits	219
Figure 9.9: Gradient and Literals vs. SOP for multiple FSM encodings	220
Figure 9.10: Entropy versus literals for random and Mustang encoded FSMs	222
Figure 9.11: Gradient versus literals for an arbitrary encoding	223
Figure 9.12: Gradient versus literals for NOVA encoding	223
Figure 9.13: Permutation of the input and output variable ordering	224
Figure 9.14: Normalized results for the permutation	225
Figure 9.15: Permutation of the input and output variable ordering	225
Figure 9.16: Normalized results for the input and output permutations	226
Figure 9.17: Normalized results for the input, output and pattern permutations	226
Figure 9.18: Normalized results for the input, output and pattern permutations	227
Figure A.1: List Scheduling Algorithm	242
Figure A.2: Simulated Annealing	243
Figure A.3: basic GA Synthesis algorithm	243
Figure A.4: Objective function calculation	244
Figure A.5: Find latest precedence routine	244
Figure A.6: Vertical List processing	245
Figure B.1: Distributed Arithmetic Sum of Products architecture	248
Figure B.2: Example time-shared schedule for a 8-point 1D DCT	252

List of Tables

Table 2.1: Number of feasible solutions for an instance of the EWF	10
Table 2.2: Number of feasible solutions under resource constraints	11
Table 2.3: Number of feasible solutions under register constraints.	11
Table 2.4: Resulting latency and hardware usage from different clock periods	39
Table 4.1: Autocorrelation calculations	70
Table 5.1: Summary of Sets and Lists	78
Table 6.1: Example FU latencies.	121
Table 6.2: Slack for clocks from optimal set.	121
Table 7.1: EWF benchmark results from GA Synthesis	146
Table 7.2: EWF benchmark results from OASIC	146
Table 7.3: EWF benchmark results from PSGA	147
Table 7.4: Kim's CDFG	151
Table 7.5: Radicojevic's Rotor examples	152
Table 7.6: Radicojevic's S2R Comparison	154
Table 7.7: Completion times of 8 8-point DCT	163
Table 7.8: Sample HLS results	164
Table 7.9: Architecture Performance comparison	165
Table 7.10: Operation latencies and resulting optimal clock set from [10]	166
Table 7.11: EWF with clock selection	167
Table 7.12: Clock Selection/Synthesis Comparison between [10] and GA Synthesis	167
Table 7.13: Synthesis results for 1*, 2 +	169
Table 7.14: Individual Paths Synthesis Results	169
Table 7.15: Multi-clock selection results	170
Table 7.16: 3.3V Adder and Multiplier characteristics	171
Table 7.17: 5V Adder and Multiplier characteristics	171
Table 7.18: Hardware Library used for the 3 EWF runs	172
Table 9.1: Toggle FSM example results	211
Table 9.2: Gradient coefficients for the first encoding	212

Table 9.3: Gradient coefficients for the fifth encoding	213
Table 9.4: PLA benchmark results	214
Table 9.5: Gradient comparison with Entropy	221
Table B.1: ALU1 connections	249
Table B.2: ALU2 Connections	249
Table B.3: ALU3 Connections	250
Table B.4: ALU4 connections	250
Table B.5: MUL1 connections	251
Table B.6: Resource scheduling for 1-D DCT	253
Table B.7: Completion times of 2-D 8 point DCT	254

List of Symbols and Acronyms

ALAP	As Late As Possible
ARAP	As Resource Constrained As Possible
ALU	Arithmetic/Logic Unit
ASAP	As soon as possible
CB	Conditional Branch
CDFG	Control/Data Flow Graph
CV	Condition Vector
CW	Computational Work
DCT	Discrete Cosine Transform
DFG	Data Flow Graph
DG	Distribution Graph
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EWf	Elliptic Wave Filter
FAC	FACtored sum of products form
FIR	Finite impulse response filter
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
FU	Functional Unit
GA	Genetic Algorithms
HLS	High-level Synthesis
IIR	Infinite Impulse Response filter
ILP	Integer Linear Programming
IU	Interconnection Unit
JPEG	Joint Picture Expert Group
MAC	Multiply-ACcumulate
MPEG	Moving Picture Expert Group
NP	Non-Polynomial
PLA	Programmable Logic Array
PSGA	Problem Space Genetic Algorithm
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register-Transfer Level
SOP	Sum Of Products
SPICE	Simulation Program Integrated Circuits Especially
S2R	Spherical to Rectangular
SU	Storage Unit
TU	Topology Unit
VHDL	Very High Speed Hardware Description Language
VLSI	Very Large Scale Integration

$*$	don't care position in a schema within a chromosome
A	alphabet used in GA optimization
$alap$	ALAP value in clock cycles of an operation
$asap$	ASAP value in clock cycles of an operation
$area$	area of a F/SU
$AvgLatency$	average of the different latencies of circuit
$AvgPower$	average power of a circuit
$B^{(f,f)}$	autocorrelation of a boolean function
b_i	i th digit in a conditional chain
$bits$	number of bits of F/SUs (width)
$\bar{c}$	cost matrix in the ILP optimization
$\bar{c}$	upper limit on clock period
$\underline{c}$	lower limit on clock period
C	logical cover of a boolean function.
C	set of conditional precedences in a graph
χ	GA chromosome
X	set of all potentially optimal clocks
cc	clock cycle (step) or relative clock cycle
CC	total latency of the graph in clock cycles
clk	clock period
clr	color of an operation prior to scheduling
$\bar{clr}$	reduced color of an operation prior to scheduling
C_{μ}	confidence of a heuristic making the correct assignment
$cost$	is the cost of a hardware unit
$Cost$	hardware cost list
c_p	optimum clock cycle position
CP	set of all conditional operations
c_{rel}	operation's clock cycle relative to latest precedent
$CTLarea$	control unit area
$CTLpower$	power consumed by the control circuitry
d	is the fraction of don't cares in a boolean function
δ	defining length of a GA schema
$delay$	delay or latency of a F/SU or operation
DG	distribution graph value
$disp$	change in operation's probability (displacement)
$distance$	minimum precedence required between two operations
e	number of precedence edges in a graph
ε	is a limit on the optimization distance
E	set of data and control precedence edges in a graph
E_{sa}	Simulated annealing energy
f	is a function.
$\bar{f}(\eta)$	average fitness of strings with schemata η
$\bar{f}$	is the average fitness of the entire population
$\vec{F}$	force in force-directed scheduling
$\vec{F}$	is the feasible solution set.
ϕ	angle of rotation.

M	mobility list of a graph
$\overline{M}$	set of clock steps in which an operation can be scheduled
μ	mobility of an operation
mem	number of internal storage elements of a F/SU.
n	number of inputs of a boolean function
N	number of optimal scheduling positions for an operation
$nbits$	number of bits dedicated for a field in the chromosome
n_d	number of don't cares in a truth table
$NFSU$	number of all F/SU types used
NFU	number of all FU types used
NFU_k	number of functional units of type k
n_O	number of outputs vectors producing the output O
NSU	number of all SU types used
num_{op}	number of operations of type op
$numstepstoend$	number of steps between the current and last cycles
o	order of a schema or order of complexity
O	set of outputs of a FSM
$\mathcal{O}$	set of operations in a graph describing their types
ω	state transition function of a FSM
op	number of output ports
P	probability of making the correct assignment
P_{ent}	probability that output $O_i = f$, used in the entropy method
P	list of data precedences in a graph
Π	number of patterns in a cover
P_c	crossover probability
$PeakPower$	instantaneous or peak power
P_m	probability of mutation
P_{sa}	simulated annealing probability
$Path$	list of operations in the path
$PATHS$	set of all conditional paths
$\ paths\ $	number of all considered paths
Pop	Population in evolutionary computation
$power$	power of the F/SU
Q	FSM quintuple description
r	random seed for GA operations
R	radius in the spherical coordinates
$\Re$	the real space
ρ	optimization distance measure
s	slack between an operation's latency and the clock period
S	set of states of a FSM
σ	standard deviation
σ_p	path slack (sum of all slacks in a path)
$sclr$	scheduled color of an operation
$slack$	number of cycles between the current and last cycles
$slope$	slope value in linear scaling
$SUarea$	area of a SU

SU_{mut}	number of SUs that operate on mutually exclusive data
SU_{num}	number of SUs
SU_{share}	time-sharing variable for SUs
$SU_{storage}$	number of storage elements in the SU
t	schedule time matrix in the ILP optimization
t	time step in evolutionary computation
t_v	execution time of operation v
T_{sa}	simulated annealing temperature
T_{sa0}	initial simulated annealing temperature
τ	latency of an operation
θ	angle of rotation
$TotalArea$	total hardware area
$TotalLatency$	total latency of the algorithm
$TotalDelay$	total latency of the algorithm
$TRIG$	trigonometric table lookup
TU	topology unit area
$type$	type of F/SU
$urgency$	urgency with which an operation ought to be scheduled
$util$	utilization list for pipelined F/SUs
v	number of operations in a graph
V	set of vertices (operations) in a graph
$WeightedLatency$	weighted average of latencies of circuit
x_{il}	implies operation i is initiated in clock cycle l
$\hat{x}$	optimization solution vector
$\hat{x}^*$	optimal solution vector
X	mutual exclusion list of a graph
X, Y, Z	cartesian coordinates
ξ	complexity criterion based on the autocorrelation
ξ'	complementary version of ξ
ψ	autocorrelation for a hamming distance of 1
z	GA population size
ζ	complexity criterion

Chapter 1: Introduction

1.1. Design automation and high-level synthesis

Improvements in chip design are dependent on two related phenomena. First is the underlying processing technology that keeps allowing faster, denser and more efficient systems. Second is the presence of design aids and tools to exploit the speed, density and efficiency of these systems.

As technologies get denser and designs get larger and more complex, the design process gets increasingly more complicated. In most cases, it is not practical nor cost-efficient to handcraft modern designs in order to attain performance and area constraints. There is a need therefore for automated methodologies that can help pack vast amounts of functionality into small areas. Design tools today are common and improving constantly.

Most of these design tools are analysis tools. Some, however, are dedicated to the synthesis process. The majority of synthesis tools in use today concentrate on lower levels of abstraction (from register-level to layout optimization). It is clear, however, that enormous gains can be achieved by moving the automation to a higher level. Systems for high-level synthesis have existed in academia for over a decade and are now slowly being introduced in industry. While many of these systems have achieved concentrated success, they are still plagued by many issues such as their efficiency and applicability. The bulk of synthesis contributions has targeted straight-line implementations of pipelined, compute-intensive DSP (Digital Signal Processing) algorithms. They have employed crude models of area and latency. They have divided the synthesis process into multiple steps to be optimized separately. Newer tools are attempting to attack the general synthesis problem which can include systems with complex control and storage requirements. They are trying to merge the different synthesis steps (scheduling, allocation and binding) as they are interdependent. They are also improving the area, delay and power models used. All these improvements naturally complicate the optimization process, and new heuristics as well as exact techniques are continuously sought.

1.1.1. Thesis Statement

-

This thesis proposes an approach to High Level Synthesis that is flexible, accommodates real Functional and Storage Units, allows multiple simultaneous objectives (area, timing and power) and performs multiple optimizations simultaneously (i.e. simultaneous scheduling, allocation and binding). It is adaptable and easy to modify for new technological changes and new objectives. It is comprehensive, handling general graphs including control-intensive graphs and considering multiple architectural optimizations (e.g. clocking). It is also powerful, using Genetic Algorithms as the optimization engine and producing *good* results on *practical* problems in *reasonable* times.

1.2. Evolutionary Computation and Global Optimization

The high-level synthesis problem contains a subset of problems that are known to be NP-complete [23] making synthesis at least NP-hard [14]. Even with the advances in synthesis techniques, there is still a demand for efficient optimization strategies that can ease the implementation of synthesis systems.

Current synthesis systems mostly employ heuristics or Integer Linear Programming (ILP) solutions. Good heuristics are usually very fast, but limited in scope and flexibility. New VLSI technology constraints often make the heuristics obsolete. Revising heuristics for new constraints is usually a long labor-intensive process. They also optimize in steps (the problem is made easier by dividing it into steps and developing heuristics for each step) and therefore miss optimization opportunities. ILP formulations, which guarantee optimality, are more easily adapted to new constraints, but suffer from a basic exponential time-complexity. Excessively long compute times often result if many constraints are combined in one optimization.

Evolutionary computation presents a solid global optimization methodology that is easily adaptable to the synthesis problem. Some work applying evolutionary techniques

such as genetic algorithms and simulated evolution to scheduling and general synthesis has been performed with promising results [30] [67] [18] [50].

Evolutionary Computation algorithms mimic the process of natural evolution. They tend to work on populations of individuals by modifying and combining the basic characteristics of these individuals. This process results in progressively better individuals, with higher average fitness, over the course of the evolution. The most common approach in Evolutionary Computation algorithms is the use of Genetic Algorithms.

In general, the drawbacks of other synthesis algorithms is the over-simplification of the problem. This is done, since it is assumed that the problem is unsolvable otherwise. This, for example is what led to the separation of the allocation, scheduling and binding steps and the abstraction of functional units.

This thesis employs a Genetic Algorithm approach for synthesis. This approach allows for better models to be incorporated. It is flexible and easily adaptable. Further, it is combined with exact techniques that allow control-dependent algorithms to be implemented.

1.3. Complexity estimation and controller area

One of the areas in high-level synthesis that has not been fully developed is the estimation of the controller area. Many models have been proposed for microcode type implementations. The estimation of the size of controllers implemented as finite state machines has been relatively neglected on the assumption that it does not play a major role in the final result. The FSM is only optimized after the synthesis is complete. This is again an example of performing high-level synthesis in steps. We believe there is a need to estimate the controller size and further implement trade-off decisions based on the controller size versus the size and behavior of the datapath. For this, we implement a new technique we refer to as the *gradient* technique and typify it as a spatial technique, as opposed to the spectral techniques common for complexity estimation.

Spectral methods have been used in the design of digital circuits with applications including verification, analysis and testing. Most of these methods, however, are of a prohibitive complexity [38] [74]. Many of these methods rely on the use of complexity criteria that can be easily defined in the context of spectral techniques.

The advantage of the method proposed in this thesis is its linear complexity in the size of the cover. Its disadvantage is that it is dependent on the ordering of the bits whereas spectral techniques are order-independent.

1.4. Contributions

The ideas introduced in this thesis advance the state of the art of high-level synthesis in multiple ways. Following is a list of the main ideas that are new in concept or implementation.

- **Using a GA formulation for the whole synthesis process not just scheduling:** Previous approaches either addressed the scheduling and allocation portions of straight line code only or they used extensive heuristics in the implementation. This method performs other synthesis tasks like control-dependent and control synthesis, register and mux allocation, and clock selection in a single GA framework.
- **Merging an exact coloring method with the GA synthesis formulation to implement CDFGs:** Previous methods either added constraints for conditional operations, transformed the graphs into non-conditional ones or used some coloring techniques to guide the heuristic schedulers. This method uses the coloring scheme to evaluate the solutions and relies on the GA for the optimization.
- **Providing a novel heuristic for the estimation of the area cost of controllers:** Previous methods used the area of a microcode ROM as the estimate. Most methods did not use the controller area during the optimization at all. This method integrates the area calculation with the optimization resulting in synthesized architectures that have better controllers. The heuristic is an improvement over complexity estimates in many

using the GA synthesis technique proposed including clock-rate selection. It also presents a novel heuristic technique for scheduling.

Chapter 7 presents synthesis benchmarks and optimization results demonstrating the quality of the solutions obtained using this method.

Chapter 8 presents different optimization strategies using genetic algorithms techniques. Some of the methods used are novel and result in promising improvements.

Chapter 9 introduces a novel spatial technique for the complexity estimation of digital circuits. It also presents results of complexity estimation on benchmark PLA and FSM circuits.

Chapter 10 concludes the thesis and recommends future work directions.

Chapter 2: High-Level Synthesis Background

2.1. Introduction

This chapter is a review of High-Level Synthesis (HLS). It defines the problem, analyzes its complexity and presents the established methods of solving it. It further introduces several concepts involved in the synthesis process including control-dependent synthesis, clock selection and controller area estimation.

Section 2.2 is an overview of High-Level Synthesis. It defines the problem in the context of Integrated Circuit design and addresses its complexity. The design space for a small benchmark is shown to be quite large.

Section 2.3 presents the three main steps in High-Level Synthesis. Allocation, binding and scheduling are defined. We also briefly show that these steps are inter-dependent and an ideal synthesis system should optimize them simultaneously.

Section 2.4 discusses the design descriptions and constraints. This shows the input to a synthesis system as a control/data flow graph, the resulting architecture and possible optimization constraints such as area, delay and power.

Section 2.5 presents the major scheduling approaches like list scheduling, force-directed scheduling and Integer-Linear programming approaches.

Section 2.6 describes some issues in High-Level Synthesis such as pipelining, chaining and multi-cycling. We also describe loop folding and loop unwinding.

Section 2.7 discusses the different control/data flow graph synthesis techniques. We present the concepts of mutual exclusion and out-of-order scheduling and briefly review existing synthesis tools that can handle branching.

Section 2.8 presents the three main coloring methods that are used to detect

mutually exclusive operation. We show that two of these methods can lead to erroneous mutual exclusion decisions. The third method was developed to avoid this problem.

Section 2.9 presents some architectural issues in synthesis. The most important of these is clock rate selection. We present the effect of clock selection on the results of synthesis and review the state of the art in clock selection. Other issues such as varying bit widths and the effect of the design style are briefly mentioned.

Section 2.10 concludes the chapter. It also argues the need for a synthesis tool that perform simultaneous scheduling, allocation and binding as well as perform clock rate synthesis. The tool should be flexible enough to adapt to new technological advances.

2.2. High-Level Synthesis Overview

2.2.1. Definition

High-Level Synthesis or *Architectural Synthesis* is the automated design process that converts design data specified at a behavioral or algorithmic level into an appropriate digital implementation at the architectural (register-transfer) level. The input representation is either a high-level description language or a control/data flow graph (CDFG). The output representation is an architectural description that typically consists of:

- operational hardware resources or functional units (FUs) that implement arithmetic and logic functions
- registers and RAMs that store intermediate data or storage units (SUs)
- a set of interconnections for transmitting data between the storage and functional elements (which includes appropriate circuitry for bus control, selection and multiplexing) that we will refer to as topology units (TUs)
- a controller for the whole system

This synthesis process is usually performed under physical constraints and/or

physical objectives such as area, latency, cycle-time and power dissipation. The constraints can be hard (e.g. less than 10mm^2) or soft (e.g. as fast as possible).

2.2.2. High-Level Synthesis in the IC design cycle

Figure 2.1 shows where the high-level synthesis process fits within the design cycle of today's integrated circuits. The main high-level synthesis tasks (Allocation, Scheduling and Binding) are shown in the rectangle. These are discussed in Section 2.3. The *specification* can be an algorithm or a CDFG. An example of each is given in Figure 2.3. The synthesis process relies on the availability of a library of units (normally FUs) that can be used to implement the given algorithm. If the required constraints are not met a re-design is required which could mean a re-synthesis or a modification to the design specification if necessary. Once a satisfactory RTL description is available, the traditional design processes of logic synthesis, simulation and verification as well as physical design are employed to arrive at the final integrated circuit design. High-level synthesis is used for architectural explorations and in the hope of reducing the total design cycle time and enable easy modifications of the design.

2.2.3. Complexity

The problem of high-level synthesis includes several sub-problems that have large computational complexity. In particular the scheduling problem in high-level synthesis, which involves a number of processors with different execution times and possibly multiple mappings between operations and operators, is proven to be NP-hard. The register allocation problem in the case of cyclic graphs is also NP-hard. The binding problem is also NP-hard [14][52].

Considering the number of variables and the complexity of the circuitry in modern designs, a vast variety of implementations are possible resulting in a large design space necessitating efficient search and optimization techniques. The large design space and the inherent complexity of the synthesis algorithms make synthesis hard to tackle.

Figure 2.1: Typical design flow using high-level synthesis.
It ends with the traditional logic synthesis/simulation/validation functions.

As an example of the size of the design space, consider the Elliptic Wave Filter benchmark (discussed in details in Section 7.2.1). The exhaustive synthesis technique employed in [15] results in the number of feasible solutions to the problem given in Table 2.1. Note that this is for a given allocation but we use it here simply as a guide to the size of the design space for benchmarks. As the number of allowed cycles increases the number of feasible schedules increases exponentially.

Clock Steps	# possible schedules
16	9,900
17	4,669,095
18	580,432,280

Table 2.1: Number of feasible solutions for an instance of the EWF

If we restrict the resources (i.e. specify the number of adders and multipliers available), we will get the number of solutions given in Table 2.2. The number of feasible solutions is a subset of the non-resource constrained schedules.

Clock Steps	Adders	Multipliers	# possible solutions	% of feasible solutions
16	3	1	77	0.78%
17	2	1	98	0.002%
18	2	1	131,983	0.023%
19	2	1	33,948,842	--

Table 2.2: Number of feasible solutions under resource constraints

Now, if we include register constraints we get the results in Table 2.3. The number of feasible solutions is even smaller. Note that these results show only the number of *feasible* solutions. The design space itself is much larger as will be discussed in Section 7.2.1.

Clock Steps	Adders	Multipliers	Registers	# possible solutions	% of feasible solutions
16	3	1	7	21	0.21%
17	2	1	7	73	0.0016%
18	2	1	7	40,723	0.0070%
19	2	1	7	3,056,246	--

Table 2.3: Number of feasible solutions under register constraints.

2.3. High-level synthesis steps

High-level synthesis is composed of three inter-related problems: *allocation*, *scheduling* and *binding*. Whether they are performed consecutively or simultaneously, the decisions made in any step have an effect on the ease and performance of the other steps. It is unclear which should be performed first in a non-simultaneous solution [14].

The *iterative* approach uses techniques like branch-and-bound to modify existing assignments. Simulated annealing is a form of randomized iterative technique.

2.3.2. Scheduling

Scheduling is the process of assigning time steps (control steps) to each operation in the given behavioral description. Each control step usually corresponds to a clock cycle in the resulting architecture. This assignment will have to ensure precedence relations are not violated. The two scheduling extremes are the completely parallel and completely serial approaches. Scheduling to meet certain constraints will result in a schedule between those two extremes. The full-parallel case corresponds to a design where no hardware restrictions are present. The full-serial approach uses the minimum amount of hardware, i.e. one of each operation type.

Different algorithms to obtain these schedules as well as other intermediate ones are presented in Section 2.5. It is important to note that the scheduling problem in synthesis is complicated by the fact that multiple different resources with different execution time exist. Section 2.5 discusses different scheduling techniques.

2.3.3. Interdependence of the steps

The three synthesis tasks described above are closely interdependent. The hardware chosen in the allocation stage results in different execution times and therefore affects the schedule, which in turn affects the binding of operations to the hardware and the resulting area (register and interconnect area). [52]

Figure 2.2 shows an example where the schedules in (a) and (b) are both optimal in time requiring 3 clock cycles, two adders and one multiplier. The number of registers required for each, however, is different. Figure 2.2 (a) requires 3 registers whereas Figure 2.2 (b) requires only two. Such considerations are easily missed if the synthesis steps are separated.

Figure 2.2: Benefits of simultaneous storage allocation and scheduling

2.4. Design descriptions and constraints

2.4.1. Design behavioral specification

The behavioral design description can be represented in many ways. The most popular and easiest approach is the use of high-level description languages such as VHDL [48]. Any such description can be modeled in a common intermediate form using Control/ Data Flow Graphs (CDFGs). A CDFG describes the algorithm in a graph form where the nodes represent the operations of the algorithm and the arcs represent the data precedences.

Figure 2.3 shows an example algorithm. The circles and lozenges denote the operations being performed. The circles represent all operations except conditional operations which cause a fork in the graph. The solid lines denote the data precedences. For instance, the condition step $c > 0$ cannot be started until c is available from the preceding addition $c = a + b$. The dashed lines are conditional precedences and are tagged by the conditional value that activates them. The control portion of the graph is sometimes separated from the data flow [52]. CDFGs do not imply execution times or hardware allocations. They are simply a representation of the algorithm to be synthesized.

Figure 2.3: Example algorithm and corresponding CDFG. Solid arcs show data precedences. Dotted arcs show control precedences.

2.4.2. Architectural descriptions

The output of the synthesis tool is an architectural description of the circuit. This is normally represented in terms of a data-path that implements the desired functionality and a controller that controls the behavior of the data-path (i.e. produces the necessary control bits).

The data-path is composed of:

- Hardware units that implement logic or arithmetic functions such as additions, multiplications or comparisons. These will be referred to as functional units (FUs).
- Storage units such as registers and RAMs used for intermediate storage of the variables. These will be referred to as storage units (SUs).

- Interconnections (buses) and multiplexers which transmit and control the paths of the variables. These will be referred to as topology units (TUs).
- Input and output logic to communicate with other devices and systems which will be referred to as interface units (IUs).

The controller is usually implemented as a microcoded ROM or a Finite State Machine (FSM). Either description may require circuitry that includes SUs and FUs. The FSM controller can further be either centralized or distributed.

Figure 2.4 shows an example synthesized architecture of the CDFG of Figure 2.3. The data-path consists of 3 FUs, an adder, a multiplier and an ALU (The ALU is used for the comparisons). Three SUs are used and they are all single storage registers. As far as TUs are concerned, the architecture employs three 2-1 multiplexers and eight buses are used. The controller is shown as divided in two parts. One to generate the control bits and another to control the states of the circuit. If no control operations are present the state logic is a simple counter. The controller in the figure controls all the registers, multiplexers and FUs.

This architecture is a random topology architecture. Other architectures have been used in high-level synthesis that restrict the topology in order to result in a more ordered architecture or to simplify the synthesis process. An example is bus based architectures where all the storage is grouped in register files and global buses are connected to the required SUs and FUs [29].

2.4.3. Constraints

The main optimization constraints used in high-level synthesis are area and delay. The area has traditionally been included as the number of FUs and registers used to implement a certain algorithm. Delay has consisted of the number of clock cycles needed to implement the algorithm. More recently, some synthesis tools have implemented area constraints in die area (mm^2) and algorithm latency in time units (ns). Other constraints

Figure 2.4: Example synthesized architecture showing the data-path and the control

have been used such as throughput for pipelined systems and power (average or peak) [66]. Some synthesis systems optimize for ease of testability [70] [37].

The synthesis can have hard constraints such as *less than 10 mm²* or *less than 10ms*, or it can have objectives like *as small as possible* or *as fast as possible*.

2.5. Scheduling

Scheduling algorithms can be classified as *transformational* or *constructive*. Constructive schedulers step through the CDFG and build a schedule one step at a time. Constructive scheduling includes ASAP, ALAP, List scheduling and Force-directed scheduling. These algorithms differ mainly in the selection method they employ for choosing the next operation to be scheduled.

Transformational algorithms, on the other hand, start from a certain schedule (e.g. full parallel) and apply transformations to improve it. Examples of these algorithms are ILP, simulated annealing and neural networks.

2.5.1. ASAP/ALAP scheduling

The simplest types of constructive scheduling are the ASAP or ALAP schedules. These are schedules that are based on the full-parallel implementation. ASAP refers to a schedule that assigns operations to be executed as soon as all the data necessary for their execution is available. The selection of the next operation is done topologically. The control step in which an operation is assigned is the earliest possible. An example ASAP schedule is shown in Figure 2.5.

Figure 2.5: Example ASAP schedule

ALAP refers to assigning operations to be executed right before the values they produce are needed. The selection of the next operation is done in reverse topological order. The control step in which an operation is assigned is the latest possible. The same example shown for ASAP is now shown for ALAP in Figure 2.6.

Both these schedules can be solved in polynomial time ($O(v+e)$ where v is the number of operations and e the number of precedences). These schedules, while providing the fastest total execution time of an algorithm (CDFG), do not take into account any resource limitations. They are however useful since they indicate the critical path of the algorithm (minimum achievable total latency) and the flexibility of the operations to move around without affecting the total latency (mobility). The critical path of a CDFG is defined as the path(s) in which all operations are scheduled in the same control step in

Figure 2.6: Example ALAP schedule

both the ASAP and ALAP schedules. The mobility of any operations is defined as the difference between its ASAP and ALAP assignments:

$$\mu_n = cc_{n_{ALAP}} - cc_{n_{ASAP}}$$

n is the operation number

cc is the clock step in which operation n is scheduled.

If the total number of steps causes some operation to have zero mobility, we can re-define the critical path as the path(s) in which all operations have no mobility, i.e. $\mu_n = 0, \forall n \in Path$ where $Path$ is the list of all operations in the path.

2.5.2. List scheduling

Heuristics are often employed to take into account resource information that is not used in the ASAP/ALAP schedules. A variety of heuristics are used, the most popular of which are list scheduling [26] and force-directed scheduling [60].

List scheduling is an approach that orders the operations in a list using a heuristic priority function and then schedules them to control steps, subject to constraints including resource constraints.

The computational complexity of most list scheduling algorithm is $O(n \log(n))$. These can easily result in a schedule that is not optimal, even when the priority function is a global criterion.

Figure 2.7: Example urgency

A typical priority function is the mobility (e.g. least mobile first). The mobility of the ASAP and ALAP example is shown in Figure 2.8 (a). Another is urgency [26]. *Urgency* is the urgency at which operations ought to be scheduled or it may be hard to schedule them cheaply later.

$$urgency_n = \frac{numstepstoend_n - (slack_n - 1)}{numstepstoend_n}$$

numstepstoend is the number of steps between the current clock cycle and the latest clock cycle for the graph.

slack is the number of cycles between the current clock cycle and the latest possible cycle in which the operation can be scheduled.

An example of urgency is shown in Figure 2.7. Consider operation 6. If we are attempting to schedule it at clock cycle 1, we get

$$urgency_6 = \frac{3-1}{3} = \frac{2}{3}$$

For clock cycle 2, this becomes

Figure 2.8: Mobility and distribution graph example based on the ASAP shown in Figure 2.5 and the ALAP shown in Figure 2.6.

$$urgency_6 = \frac{2-0}{2} = 1$$

The urgency of scheduling operation 7 to clock cycle 1 is, on the other hand

$$urgency_7 = \frac{3-2}{3} = \frac{1}{3}$$

For clock cycle 2, this becomes

$$urgency_7 = \frac{2-1}{2} = \frac{1}{2}$$

This implies that scheduling operation 6 is more *urgent* than operation 7.

2.5.3. Force-directed scheduling

Force directed scheduling [60] uses a different criterion to select the next operation. It creates a distribution graph based on the operations' mobilities to decide on an operation's assignment. The distribution graph represents the probability of operations being assigned to certain control steps. *Forces* are then calculated that are proportional to the need of scheduling an operation into a specific control step. The assignment with the most negative force is performed and the distribution graph is recalculated after each

assignment. An example of a distribution graph is shown in Figure 2.8 (b). It is based on the mobilities of Figure 2.8 (a) and assumes that all operations are of the same type. One can view the distribution graph as a series of springs that exert forces on all operations. The force for a given operation n in clock cycle i is

$$F_n(i) = DG(i) \cdot disp_n(i)$$

$DG(i)$ is the distribution graph value at clock cycle i (i.e. the spring's constant).

$disp_n(i)$ is the change in the operation's probability (i.e. the spring's displacement).

We can define the force for scheduling a node n to a clock cycle i as

$$F(n \rightarrow i) = DG(i) - DG_{avg} = DG(i) - \frac{1}{\mu} \sum_{j \in \bar{M}} DG(j)$$

with μ being the mobility and $\bar{M}$ the set of all clock steps where n can be scheduled. i.e. $\mu = |\bar{M}|$.

In general, the force can be calculated as follows:

$$F_{hw}(n \rightarrow CC_l) = \left[\sum_{\substack{j=1 \\ j \neq l}}^{\mu} DG_{HW}(cc_j) \cdot \left(\frac{1}{\mu} \right) \right] + DG_{HW}(cc_l) \cdot \left(\frac{\mu - 1}{\mu} \right)$$

with DG being the distributed graph value calculated as:

$$DG_{HW}(cc) = cost(HW) \times num_{op}$$

num_{op} is the number of operations of type HW with mobilities overlapping cycle cc .

The operation to be scheduled is the one with the most negative force. It will be scheduled in the control step that results in the most negative force.

2.5.4. Other Heuristic Schedulers

Many other heuristic techniques have been proposed for scheduling [78]. For example, trace scheduling [21], applies a series of transformations that displace operations in and out of control blocks to improve performance. Percolation scheduling [61], on the other hand, starts with a schedule that is iteratively improved by moving operations to different time slots. Rules are used to define valid transformations.

2.5.5. Neural Networks scheduling

Neural networks have been used in the scheduling problem [32]. The implementation uses a self-organizing algorithm. All operations compete for control steps within their execution interval and the neighborhood information is gradually reduced until one operation at a time is considered. The disadvantage of this approach is the difficulty and low-speed of training as well as the choice of training sets.

2.5.6. Simulated annealing scheduling

Simulated annealing randomly generates modifications in the schedule and accepts or rejects them according to a random rule and a temperature parameter that gradually decreases with time. It is a hill-climbing algorithm that has the ability to get out of local optima.

Simulated Annealing is a stochastic relaxation technique based on the analogy of the physical process of annealing a metal. At high temperatures the atoms are randomly distributed. With decreasing temperatures they tend to arrange themselves in a crystalline state which minimizes their energy. The algorithm, then generates randomly new solutions. If the solution is an improvement, the algorithm will move to it. If it is not, the decision of moving to the new solution is governed by the following probability:

$$P_{sa} = e^{\frac{-E_{sa}}{k_{sa}T_{sa}}}$$

E_{sa} is related to the error decrease (or fitness) between the current solution and the next solution. T_{sa} is a temperature value that starts high and is gradually decreased and k_{sa} is a constant.

Simulated Annealing has been used for scheduling [17][56]. In this thesis, the synthesis problem is cast as a simulated annealing problem and compared to genetic algorithms in Section 8.6.

2.5.7. Integer Linear Programming scheduling.

Integer Linear Programming (ILP) is an exact optimization technique. ILP scheduling is formulated as an optimization problem using certain constraints. It suffers from increasing time-complexity however, since it is exponential in the number of variables and inequalities. This puts larger, more realistic problems out of its scope. The major advantage is the optimality of the solution. Synthesis approaches that use ILP can be found in [24] [35] [46].

The scheduling problem can be formulated as an ILP problem as follows [14]:

minimize $c^T t$ where c is a hardware cost matrix and t is an operation scheduling time matrix, such that

$$\text{Uniqueness: } \sum_l x_{il} = 1, i = 0, 1, \dots, v \quad (2.1)$$

$$\text{Precedence: } \sum_l l \cdot x_{il} - \sum_l l \cdot x_{jl} - \text{delay}_j \geq 0, \quad i, j = 0, 1, \dots, v, \quad (v_j, v_i) \in E \quad (2.2)$$

$$\text{Resource Constraints: } \sum_i \sum_l x_{il} \leq NFU_k, \quad k=1,2,\dots,|K|, \quad l = 1, 2, \dots, \bar{\lambda} + 1 \quad (2.3)$$

$$\text{Non-negativity: } x_{il} \in \{0, 1\}, \quad i = 0, 1, \dots, n, \quad l = 1, 2, \dots, \bar{\lambda} + 1 \quad (2.4)$$

v is the number of operations

x_{il} implies operation i is initiated in clock cycle l

$delay_j$ is the delay of operation j

NFU_k is the number of functional units of type k

$|K|$ is the number of different available resources.

$\bar{\lambda}$ is an upper bound on latency.

c is the cost matrix.

t is the schedule time matrix.

Equation (2.1) restricts operations to unique start times, i.e. the operation must be scheduled only once.

Equation (2.2) ensures that an operation is scheduled only after its predecessors are finished, guaranteeing operation dependency.

Equation (2.3) defines the resource constraint where all operations scheduled in one cycle are less than or equal to the maximum available hardware resources. (This formulation assumes that each operation has latency = 1)

Equation (2.4) ensures that scheduling in the non-negative domain.

This formulation is not unique and can be (has been) improved and expanded to handle more constraints.

2.5.8. Control flow schedulers

While most of the above schedulers can be enhanced to support graphs that include conditional execution of operations, most of them were intended and work better for straight line code. Some schedulers were developed specifically to deal with conditional and loop constructs. Some of these are discussed in Section 2.8.

Many other synthesis systems exist. For a good survey, refer to [78]. Some of the more recent synthesis systems are mentioned in [47].

2.6. Scheduling and synthesis issues

This section presents some synthesis issues and features that affect the performance of the synthesis tools and the quality of synthesized architectures.

2.6.1. Pipelining

Pipelining in high-level synthesis refers to the synthesis using pipelined units. A synthesis tool can have the ability to use pipelined modules in its library and assign operations to them. Most synthesis tools use simple pipelined units that perform one type of operation with a predefined throughput and latency. Figure 2.9 shows an example schedule including a pipelined unit (a) and an implementation of a pipelined adder (b). Note that the pipelined register would be internal to the adder and cannot be shared for other storage. This is not a hard restriction and circuitry can be made available to time-share this register. The assumption, however, is usually that the hardware unit is custom-designed and optimized then included in a library without options to customize it. In a pipelined unit, the pipeline registers can be implemented in dynamic logic.

Figure 2.9: Pipelining Example.

(a) shows the pipelined operation and (b) the pipelined implementation

2.6.2. Chaining

Chaining is the process of performing more than one operation in one clock cycle.

Most systems have demonstrated chaining of two operations. Extensions to multiple operations are deemed trivial. Figure 2.10 shows an example of operation chaining where operations 1 and 2 are executed in the same cycle as operation 3 which is dependent on their results.

Figure 2.10: Chaining example.
Operations 1,2 and 3 are chained in one control step.

2.6.3. Multi-cycling

Multi-cycling of operations implies that an operation can be scheduled over two or more time steps when the latency of the hardware used exceeds the clock period of the system. While synthesis systems use that for small, slow hardware units, it can also be useful for multi-phase, multi-clock designs. Figure 2.11 shows an example where operation 4 is executed over 2 cycles.

Figure 2.11: Multi-cycling example.
Operation 4 is executed over 2 control steps

2.6.4. Loop folding/unwinding

Loop unwinding and loop folding are CDFG modification techniques that can impact the performance of the synthesis system. Loop unwinding unrolls (or flattens) a loop whereas loop folding increasing parallelism by taking flattened portions and folding the operations that normally belong in the loop such that they can be performed in parallel. Different optimizations can be performed for each. Operations in the folded loop may need to be time-shared with ones outside the loop. If this is not supported by the synthesis tool, the CDFG can be unwound. Figure 2.12 shows an example of a CDFG loop (a) and its unwound version (b). The unwound loop requires one adder and one multiplier for a nine cycle implementation.

Figure 2.12: Loop unwinding example.
The loop in (a) is unwound into the scheduled CDFG in (b).

By folding this CDFG segment as shown in Figure 2.13, we can decrease the execution time of the algorithm. The first unfolded version (a) reduces the number of cycles by two while using the same number of FUs (You have to note here that the

additions are done in the same cycle as one of the multiplications). The second version (b) reduces the number of cycles by another two by using one additional FU.

These types of optimizations have to be performed by a synthesis tool for pipelined CDFGs.

Figure 2.13: Loop folding/unwinding example.

2.6.5. Relative scheduling

Relative scheduling is employed when there are operations in the algorithm that are *unbounded*. These operations do not have a predefined execution time but rather issue a completion signal when they are finished. Since we cannot determine the latency of unbounded operations before run time, the scheduling has to be performed in a relative manner. The operations with unspecified operation delays will be scheduled relative to those which have specified operation delays. A relative scheduling method was proposed in [43].

2.6.6. Controller synthesis

- An important problem in high-level synthesis research today is controller synthesis. Most synthesis tools optimize the data-path and then build a controller, after the fact, that consists of either a finite state machine or a microcode ROM. While this is acceptable for datapath-intensive applications such as DSP algorithms, it may not be appropriate for more control-intensive applications such as communications systems. A good synthesis tool will have to include the controller restrictions and effects in the optimization of the resulting architectures [7].

2.7. Control-scheduling issues

This section describes the issues involved in the synthesis of CDFGs as opposed to straight-line code segments. By control scheduling, we refer to the scheduling of conditional operations as well as data-dependent loops.

2.7.1. Mutual exclusion

Operations are mutually exclusive if they correspond to different branches of a conditional. As these operations will never need to be executed at the same time, they can share the same hardware without incurring functional unit area overhead. Section 2.8 shows different techniques used for coloring. Coloring here is simply a **labeling** of the nodes to distinguish mutually exclusive branches.

2.7.2. Out-of-order execution

Out-of-order execution is the execution of a conditional operation prior to the execution of the condition on which it is dependent. This is shown in Figure 2.14. In the ordered schedule, the comparison is performed first at clock cycle 1, only one of the addition or subtraction is then performed in clock cycle 2 since the condition has been evaluated. In out-of-order scheduling, both the addition and subtraction are executed in cycle 1 concurrently with the evaluation of the condition. the addition and subtraction

executed simultaneously with the comparison are also out-of-order. This technique can speed up the execution time of an algorithm when the condition evaluation is in the critical path. This will have no FU cost impact if idle FUs are available.

Figure 2.14: Example is out-of-order execution of conditional operations

2.7.3. CDFG synthesis tools

In [40], a method was proposed that transforms a CDFG into a data flow graph without conditional branches. The graph is then scheduled and the resulting schedule is mapped back to the conditional graph. This method does not support out-of-order execution.

In [8], a path-based method was proposed. It is an *As Fast As Possible* scheduler which extracts all possible execution paths from a CDFG, schedules them separately and combines them while resolving conflicts. Other methods [58] [64] [77] use mutual exclusion tests during scheduling to determine whether operations are mutually exclusive or not. Section 2.8 describes these methods referred to as coloring.

2.8. Coloring Techniques

- This section describes three coloring techniques employed in scheduling CDFGs that include conditional branch constructs.

The main reason for coloring CDFGs is to recognize mutually exclusive operations. These are useful in determining which hardware can be time-shared among different operations. Several coloring algorithms have been developed, three of which are presented here.

2.8.1. Node Coloring Algorithm

A node coloring algorithm put forward by Park and Parker for testing of mutual exclusion in DFGs is presented in [57]. It assigns a color code consisting of a sequence of one or more integers to each node.

Coloring this graph is based on an algorithm that adheres to the following rules:

- All unconditional operations are each colored with the same integer (0).
- Children of a conditional node are *colored* with a sequence of numbers derived from the sequence number of their parents followed by an additional integer that is unique to each child.
- Join nodes have the same color sequence as their associated condition (fork).
- A node whose parent is not a fork and is not itself a join is given the same color sequence as its parent.

This results in a graph where each node is given a sequence of integers whose length represents how many levels of distribute-join pairs the node is nested in. An example coloring is shown in Figure 2.15.

In the graph, mutual exclusion results from the output of a conditional node selecting a particular path while falsifying the other and vice versa. The node coloring allows detection of mutually exclusive nodes based on the following rules:

Figure 2.15: Example coloring using Park's method

- Nodes with colour 0 are not mutually exclusive.
- Two nodes with the same length but different colours are mutually exclusive.
- Nodes whose color numbers differ in length are mutually exclusive if when reduced (truncated from the right) to the same length they result in different colours.

Examples of mutual exclusion testing for nodes in Figure 2.15 follow:

- n_2 (11) is mutually exclusive from n_3 (12). Same length and different colours.
- n_5 (21) is mutually exclusive from C_2 (1). When reduced, (2) and (1) have same length and are different.
- n_6 (22) is not mutually exclusive from C_3 (2). When reduced, (2) and (2) have same length and are equal.

2.8.2. Condition Vector method

The condition vector (CV) method was proposed in [77] and was later extended into a more global scheduling algorithm. It can accomplish conditional resource sharing

thoroughly and can optimize all possible execution paths.

- CVs show the execution conditions of an operation in vector form. A CV is a bit-wise one-hot encoding of distinct leaf branches. CVs for non-leaf nodes are the OR-ed result of all leaf branches reachable from these operations. Basic condition vectors are an encoding of the execution conditions of operations.

Wakabayashi further extended the concept of CVs to include Extended CVs which encode the necessity of operations to be executed. This means that if an operation with a CV of $[1,1,1]$ produces data that is used only in the first and last branch, then its extended CV is $[1,0,1]$ since it is essentially mutually exclusive with all data in the middle branch. Actual CVs were then introduced to enable scheduling operations before or after their conditional node has been evaluated. Actual CVs describe the mutual exclusion states of operations without assuming that all the branching operations are resolved. After all the operations are scheduled, the Actual CVs are equal to the Extended CVs. Actual CVs, then allow for dynamic mutual exclusiveness.

Figure 2.16 shows the coloring of an example graph. The four leaf nodes (n_2 , n_3 , n_5 and n_6) require four bits for representation with each having a single bit set. The color of each parent is the *bit-wise or* of its children, therefore C_2 is colored with $[1000] + [0100] = [1100]$.

If CVs have no common asserted bits, their corresponding operations are mutually exclusive and can share the same hardware resources. CVs can further express dynamic execution conditions. If a particular conditional node, has been executed (scheduled before its children) then it results in one CV (done). If it has not been executed, it results in a different CV (not done). These CVs allow for dynamic mutual exclusion testing. Example mutual exclusion testing of the example in Figure 2.16 follows:

- n_2 (only bit 1 set) is mutually exclusive with n_3 (only bit 2 set).
- n_5 (only bit 3 set) is mutually exclusive with C_2 (bits 1 and 2 set)
- n_6 (only bit 4 set) is not mutually exclusive with n_4 (bits 3 and 4 set).

Figure 2.16: Example coloring using Condition Vectors

2.8.3. Control Tag Method

In this algorithm [64], a unique integer is assigned to each branch reachable from a Conditional Branch (CB) node. Unconditional nodes are empty “()”. CEs (Conditional Ends) have the same tag as their corresponding CBs. Conditional nodes are tagged with a sequence of *condition* and *selection* fields. The condition field is determined by the boolean value of the CB node, and the selection field is determined by the attributes of the conditional precedence edge (T or F). This representation captures both the path leading to a node and the condition of each branch on that path. A unique integer is used for each Boolean value incident to a CB node.

An example of Control Tag node coloring is shown in Figure 2.17.

Each node is colored by pairs of values specifying the condition and selection fields. A color specifies not only the path it followed from the top ancestor but also the condition evaluation. For instance $(1T\ 2F)$ of node n_3 specifies not only the fact that its predecessors are C_1 and C_2 but also that these have to be True and False respectively to

traverse the path to n_2 . This method is identical to Park's method [58] if one only considers two mutually exclusive branches for each branching condition.

If two nodes are activated with the same condition but different selections (e.g. 1T and 1F), they are mutually exclusive. Then for every pair of nodes, if the same condition is present in both but with different selections then they are mutually exclusive; otherwise they are not. This is shown in Figure 2.17.

Example mutual exclusion tests from Figure 2.17 follow:

- n_2 (1T 2T) is mutually exclusive from n_3 (1T 2F). Condition 2 is common and the corresponding selections are different.
- n_2 (1T 2T) is mutually exclusive from C_3 (1F). Condition 1 is common and the corresponding selections are different.
- n_3 (1T 2F) is not mutually exclusive from C_2 (1T). Condition 1 is common and the corresponding selections are identical.

Figure 2.17: Example coloring using the control tag method

2.8.4. Comparison of the methods

The three mutual exclusion algorithms presented above are the best known solutions to the coloring problem. Figure 2.18 shows the coloring of all three methods. The Control Tag method is more rigid since it is based on the boolean variables selecting the branches. The other methods can incorrectly identify non-mutually exclusive nodes as mutually exclusive. Note that these methods can be easily corrected by correctly identifying the two *T* branches as parallel.

Figure 2.18 is depicted to show the inherent weakness of the node coloring algorithm and the condition vector method which will incorrectly identify nodes *n2* and *n5* to be mutually exclusive. Rim's algorithm avoids this pitfall in a simple manner.

Figure 2.18: Comparison of the three coloring methods

Following is the mutual exclusion test for n_1 and n_3 in all three methods:

Node coloring method:

- n_2 (111) and n_5 (121) are mutually exclusive since they have the same lengths and are different.

Condition vector method:

- • n_2 (10000) and n_5 (01000) are mutually exclusive since they don't have a common set bit.

Control tag method:

- n_2 (1T 2T) and n_5 (1T 3T) are not mutually exclusive since 1 is common but the corresponding selection fields are identical.

2.9. Architectural issues in high-level synthesis

This section presents some additional architectural issues in high level synthesis. These can all be supported with the synthesis methodology presented in this thesis.

2.9.1. Multi-rate systems

Practical systems have more than one clock domain. A general synthesis tool should have the ability to synthesize and optimize multi-rate systems. An example of such a system is a parallel to serial interface. While the synthesis tool may have the freedom of selecting the actual clock frequency, it still has to guarantee the relationship between the serial and parallel clocks. Most synthesis tools today do not consider multi-rate systems.

2.9.2. Clock rate synthesis

The choice of period can affect the execution time and resource usage of CDFG. Consider the three different cases from [9] shown in Figure 2.19. Considering for example that adders have a latency of 80ns and multipliers a latency of 150ns, (a) is the schedule for a clock period of 380ns, (b) corresponds to a clock period of 150ns and (c) is the schedule for a clock period of 80ns. The effect of the clock selection on the total algorithm latency and hardware usage is shown in Table 2.4.

It is clear that different clock periods result in dramatically different results. The

Figure 2.19: Effect of clock period on synthesis results

Clock Period	380	150	80
Total Latency	380	600	400
Multipliers	2	1	1
Adders	4	1	1

Table 2.4: Resulting latency and hardware usage from different clock periods

method of choosing the clock period is therefore important.

Traditionally HLS tools assumed that the designer will select the clock period. The delays of the units are then expressed in clock cycles. Some tools however have addressed the clock selection problem. In [59], the clock period chosen is equal to the largest operator delay in the critical path. The critical path is determined first. In [58], a lower bound on the clock period is equal to the longest stage delay. This results in a clock period that is greater or equal to the longest operator delay. In [36], the DFG is divided into a number of time steps. The clock period is then the greater of the largest operator delay or

the critical path delay per time step. None of these methods considered multi-cycling.

- In [9] a heuristic is proposed based on the slack minimization criteria. Slack is defined as

$$s_v(clk) = \left(\left\lceil \frac{\tau(v)}{clk} \right\rceil \times clk \right) - \tau(v)$$

where $s_v(clk)$ is the slack for operation-type v when using a clock period clk .

$\tau(v)$ is the latency of operation v .

The latency of the operation was based on a regular topology architecture that allowed the following simplification:

$$\tau(v) = \tau_r + \tau_{FU}(v) + \tau_i$$

where τ_r and τ_i are the delays introduced by the register read/write latency and the interconnect delay respectively. $\tau_{FU}(v)$ is the delay of the hardware unit that can execute operation v . There is an assumption of one FU per operation type.

It was demonstrated that the optimum clock periods (i.e. ones that minimize the total slack in the algorithm's latency) corresponded to ones that integrally divide one or more $\tau(v)$. All potentially optimal clocks can then be calculated for the set of FUs in the library.

The selection heuristic then assumes that minimizing the average slack for all operations in the DFG amongst the set of potentially optimal clocks will result in a good clock period estimate. This estimate corresponds to minimizing

$$\sum_i \left\{ k_i \times \left(\left\lceil \frac{\tau_i}{clk} \right\rceil \times clk - \tau_i \right) \right\} \quad \forall clk \in X$$

with

$$k_i = \frac{v(k)}{\|V\|}, i \in V,$$

and X is the set of all potentially optimal clocks. V is the set of all operations and $v(k)$ is the number of operations of type k .

The method resolves the problems with the previous methods and easily allows chaining and multi-cycling. It still suffers from two problems. The first is that the estimate is still performed prior to the synthesis and therefore cannot be optimal. It is beneficial to perform this during synthesis. The second problem is that including multiple types of hardware units for the same operation types is not easily incorporated.

The first problem is resolved in [10] where the same optimal clock set selection is done. Rather than using a heuristic, however, a complete ILP is run for each clock in the set. The clock is selected corresponding to the ILP producing the best results. While this method proves optimality, it can only be applied to a small set of clocks. For larger libraries containing many units, this method will result in running a large number of ILPs each of which can be very lengthy.

2.9.3. Varying bit-widths

Real life algorithms do not always use data with the same widths. This is particularly important for control-intensive applications where the width of the control signals varies widely. Some DSP algorithms also require different data-path widths for different portions. Recognizing different bit widths and synthesizing the hardware accordingly are an important part of synthesis.

2.9.4. Design style and other issues

There are many different architectural and design styles ranging from full-custom design to cell-based and gate array to programmable structures. Designers often know prior to engaging in a design which style it is suited for, based on prior experience. There are cases however where an evaluation has to be made. Synthesis tools that rely on hard

heuristics based on the design style may be difficult to modify (e.g. HLS tools optimized for FPGAs). It is beneficial to have soft encodings of design style criteria such that one tool can be used to evaluate multiple design styles or even a mixture of design styles.

While the bulk of the work in high-level synthesis has concentrated on single chip multi-processor architectures, the drive towards higher levels of abstraction and more automation has produced solutions for system synthesis. Many issues involved in system synthesis such as hardware/software partitioning are beyond the scope of this thesis. Multi-chip solutions, however, should be a simple extension of any synthesis tool.

Partitioning is essential for large algorithms that have to be implemented on multiple chips [78]. It is also beneficial when interconnect delays are significant on a chip. This implies that different portions of the device have to be clustered according to their interconnections. This clustering can impact delay and area as well as power.

2.10. Conclusion

This chapter presented a review of High-Level Synthesis. It defined the synthesis problem and explored its complexity. The problem being NP-hard implies the need for efficient optimization methods. The Genetic Synthesis approach used in this thesis seems to be suited for such problems.

This chapter also presented the three main steps in High-Level Synthesis: allocation, binding and scheduling. We explained how the simultaneous optimization of these tasks is necessary. The Genetic Synthesis encoding allows for such simultaneous optimizations.

We also discussed the design descriptions and constraints. Some of these constraints work against each other, so a good optimization system must allow for multiple constraints. Genetic algorithms can easily accommodate multiple constraints.

This chapter presented the major approaches to scheduling. The heuristic methods that were presented are fast but hard to modify and could easily get caught in local optima.

Allowing for more global heuristics increases their complexity. Other entrenched methods such as Integer Linear programming provide optimal schedules but suffer from exponential complexity in the worst case. Genetic Algorithms seem to provide the appropriate level of complexity and quality required for this problem.

We also presented several techniques that help in the synthesis of data-flow graphs such as pipelining, chaining, multi-cycling, loop folding and loop unwinding. All of these features can be implemented with the Genetic Synthesis approach used in this thesis.

We presented the ideas of mutual exclusion and out-of-order scheduling for control/data flow graphs. We also focused on coloring methods for mutual exclusion detection. This thesis incorporates an exact coloring scheme in the Genetic encoding and therefore can handle these graphs.

The importance of clock rate selection was discussed. In this thesis we incorporate clock selection with synthesis. The flexibility of the Genetic Synthesis approach allows us to optimize these simultaneously. We can also perform other optimization that have not been previously attempted such as multi-rate clock selection.

This thesis proposes a comprehensive method that utilizes many of the techniques presented in this chapter and solves many of the problems and shortcomings of existing synthesis tools.

Chapter 3: Evolutionary Computation Background

3.1. Introduction

This chapter presents background material on Evolutionary Computation. It also presents the features of Genetic Algorithms as one of the major Evolutionary Computation techniques. It then surveys the use of Evolutionary Techniques in High-Level Synthesis.

Section 3.2 introduces Evolutionary Algorithms. Here we present a general algorithm for Evolutionary Computation and then present a Genetic Algorithm. The main differences between Genetic Algorithms and other evolutionary techniques are presented. For example, optimizing in the genotype versus phenotype space are presented.

Section 3.3 discusses the basics of genetic algorithms and the different enhancements that can be incorporated in their use. We present an example encoding and define the basic terms and concepts such as population, crossover, mutation, reproduction, fitness, scaling dominance and isomorphism.

Section 3.4 presents the schema theorem underlying the success of Genetic Algorithms in optimization. The theorem proves that successive generations will get exponentially increasing representations of chromosome building blocks that lead to fitter individuals.

Section 3.5 is a survey of High-Level Synthesis methods that employ Evolutionary techniques. We show that these methods either were not general enough (e.g. concentrated only on straight-line code or performed scheduling only) or incorporated heuristics that may reduce their flexibility.

Section 3.6 concludes the chapter and presents arguments for the Genetic Algorithm Synthesis tool presented in this thesis that is more comprehensive than other evolutionary methods and does not incorporate application-specific heuristics that limit its flexibility.

3.2. Evolutionary Algorithms

Evolutionary algorithms (EAs) present a powerful optimization technique. They are particularly effective in complex search spaces. They have been successfully used in a variety of applications. They are attractive due to their simplicity and power. They are also inherently parallel and very flexible [5]. This section presents the basics of Evolutionary Algorithms and discusses their various features.

3.2.1. Optimization

Optimization implies finding a vector $\mathbf{x} \in M$ such that an objective function $f: M \rightarrow \mathfrak{R}$ is maximized (or minimized) $f(\mathbf{x}) \rightarrow \max$. An optimum solution vector $\mathbf{x}^*$ is such that $\forall \mathbf{x} \in M: f(\mathbf{x}) \leq f(\mathbf{x}^*)$ ¹.

Often, there are multiple local optima $\mathbf{x}'$ (local with respect to a certain distance measure ρ). This is known as multimodality:

$$\left. \begin{array}{l} \forall \mathbf{x} \in M, \exists \varepsilon > 0 \\ \exists \rho(\mathbf{x}, \mathbf{x}') < \varepsilon \end{array} \right\} \Rightarrow f(\mathbf{x}) \leq f(\mathbf{x}')$$

where ρ is a distance measure and ε is a limit on the distance.

Optimization is typically performed under certain constraints that have to be satisfied: $g_j: M \rightarrow \mathfrak{R}$ such that the set of feasible solutions $\hat{F}$ is a subset of the domain variable M :

$$\hat{F} = \{\mathbf{x} \in M | g_j(\mathbf{x}) \geq 0, \forall j\}$$

Typically g_i are expressed as inequalities.

1. $\forall$ means “for all”, $\exists$ means “there exists” and $\{0, 1\}^l$ defines a binary string of length l .

3.2.2. Algorithms

During the last thirty years many problem solving methods based on the principle of evolution have been developed. We refer to these methods as *Evolutionary Algorithms*. Evolutionary algorithms are stochastic algorithms that use genetic principles as their underlying search technique. These include Evolution Strategies [68], Evolutionary Programming [22], Genetic Algorithms [34] and Genetic Programming [42].

In EAs, a single individual is affected by the other individuals, by exchanging building blocks with them, as well as by the environment that determines its fitness and survival. A sample algorithm is shown in Figure 3.1. We start by initializing a population $P(t)$ of one or more individuals. The fitness (determined by the application-specific objective function) of the individuals in the population is determined. A new population is selected from the current one, then modified and evaluated. The process of selection, modification and evaluation is repeated until some termination condition based on the fitness or run time.

```
EA()  
     $t = 0$   
    Initialize  $Pop(t)$ , a population of 1 or more  $\mathbf{x}$  vectors.  
    Evaluate the fitness of each  $\mathbf{x}$  in  $Pop(t)$   
    While (not termination condition) do  
         $t = t + 1$   
        Select  $Pop(t)$  from  $Pop(t-1)$   
        Modify  $Pop(t)$   
        Evaluate the fitness of each  $\mathbf{x}$  in  $Pop(t)$   
    end
```

Figure 3.1: Evolutionary Algorithm

Genetic Algorithms, which are the most commonly used type of EAs represent

individuals in a population by binary strings. The objective function maps $f:\{0,1\}^l \rightarrow \mathbb{R}$. Such GAs require an encoding step $h:M \rightarrow \{0,1\}^l$ and a decoding step $h':\{0,1\}^l \rightarrow M$. They operate on the bit strings or the *genotype* space as opposed to the original problem space *phenotype*. The encoding function might introduce an additional multimodality making the problem more complex but the genotype method benefits from the existence of operations proven to improve the population's fitness.

Evolutionary Strategies and evolutionary programming operate on the phenotype space. $f:(M \subseteq \mathbb{R}^n) \rightarrow \mathbb{R}$.

“The actual development of the field is characterized by a progressing integration of the different approaches, such that the utilization of the common labels *Genetic Algorithms*, *Evolutionary Strategies* and *Evolutionary Programming* might be sometimes even misleading” [5]. We therefore will refer to these methods as Evolutionary Algorithms and the underlying techniques as Evolutionary Computation.

The above arguments notwithstanding, we will still describe GAs in terms of Evolutionary Computation by limiting GAs to the manner in which they have been traditionally used. The GA algorithm follows the one presented for Evolutionary Algorithms in general. The population is initialized either randomly or seeded using a problem-specific heuristic. The population is described in terms of chromosomes made up of symbols, (normally $\{0,1\}^l$) as is explained later. An encoding h (and decoding h') procedure is needed from the phenotype to the genotype space. The evaluation is application-specific. The termination condition is typically either CPU-time, number of generations (or iterations) or closeness to desired results or objectives if known. The selection is performed according to a fitness-based criterion. The modification employs crossover and mutation operators (described later). Figure 3.2 shows a sample GA algorithm.

```

GA()
     $t = 0$ 
    Encode Pop( $t$ )
    Initialize Pop( $t$ ), a population of many  $\mathbf{x}$  vectors
    Decode Pop( $t$ )
    Evaluate the fitness of each  $\mathbf{x}$  in Pop( $t$ )
    While (not termination condition) do
         $t = t + 1$ 
        Select Pop( $t$ ) from Pop( $t-1$ )
        if ( $P_c$ ) crossover Pop( $t$ )
        if ( $P_m$ ) mutate Pop( $t$ )
        Decode Pop( $t$ )
        Evaluate the fitness of each  $\mathbf{x}$  in Pop( $t$ )
    end

```

Figure 3.2: Sample Genetic Algorithm

3.3. Genetic Algorithm definitions and basics

3.3.1. Genetic algorithms

GAs [27] [54] are a class of algorithms modeled after natural evolution. They can be used to optimize functions by operating on populations of individuals represented by chromosomes in the form of bit strings. The operations ensure that the best-fit solutions (individuals) are selected and reproduced by combining them together. This process is known as *crossover* and is guided by the application-specific fitness. The other basic GA operation is the *mutation* which is a random modification of the chromosome. This introduces variety in the population and keeps the solutions from saturating quickly.

3.3.2. Chromosome

A chromosome is a sequence of symbols typically represented as bits. The analogy is between the sequence of symbols in the representation and chromosomes in biological systems. Each problem solution of a GA is a chromosome. GAs as originally described by Holland [34] used bit-string encoding. This means that every bit string in the chromosome corresponds to a symbol. Different symbol encodings are possible and have been used. Examples of these encodings include integers, floating point numbers and graph structures. If the function

$$f(x) = ax^2 + bx + c$$

is to be minimized subject to some constraints:

$$0 < x < 256$$

$$-16 < a \leq 16$$

$$0 < b < 28$$

$$-64 \leq c < 64$$

then a possible encoding is given in Figure 3.3 where 4 symbols are used. $\{x, a, b, c\}$ is the representation of the problem in a list of symbols. In an integer encoding, these symbols would take on integer values as shown in the integer example with $\{10, -5, 7, 2\}$. Similarly a binary encoding can be easily devised. Note that, in the binary encoding the constraint $0 < b < 28$ still requires 5 bits. The values 28 to 31 are normally mapped to another 4 integers.

Chromosome usually denotes the encoding $\{x, a, b, c\}$. *Individual* or *bit-string* usually denotes the chromosome with values annotated to the symbols, i.e. a particular solution of the algorithm like $\{10, -5, 7, 2\}$.

3.3.3. Symbol

Symbol refers to the building blocks of chromosomes as described above. If a positive integer encoding is used then the symbols are natural numbers. The

Figure 3.3: Chromosome encoding for the GA example.
Each oval represents a symbol.

implementation of these symbols can use a bit-string, binary representation or any other representation. Symbols are also known as genes.

3.3.4. Population

A GA population is a collection of individuals. Since each individual represents an encoded solution to the problem, a population is then just a collection of solutions at any point in time. The GA will operate on the population of chromosomes rather than the chromosomes themselves, hence the inherent parallelism. GAs usually start by generating a random population. GA operators are then used to generate subsequent populations that tend towards better solutions. This is the reason populations are often referred to as generations.

3.3.5. Crossover

Crossover is the process of combining chromosomes in a certain generation to produce offsprings (chromosomes for the next generation). There are many different kinds of crossovers. Figure 3.4 shows the 1-point crossover case. The shown chromosomes χ_1 and χ_2 exchange symbols beyond the crossover point k .

Two offsprings can result from this operation with:

Figure 3.4: Single-point crossover example.
Parent chromosomes have different shading. Offsprings are vertically separated.

$$\chi'_1(i) = \begin{cases} \chi_1(i) & \text{if } (i \in [0, k-1]) \\ \chi_2(i) & \text{if } (i \in [k, l-1]) \end{cases}$$

$$\chi'_2(i) = \begin{cases} \chi_2(i) & \text{if } (i \in [0, k-1]) \\ \chi_1(i) & \text{if } (i \in [k, l-1]) \end{cases}$$

k is the crossover point

l is the chromosome length

χ_1 and χ_2 are the parent chromosomes

χ'_1 and χ'_2 are the offsprings.

It is possible for this crossover to happen in the middle of the *symbol* represented by a string of bits. Usually, however, the crossover points are selected such as not to disrupt the exchanged symbols.

Figure 3.5 shows a case where 3-point crossover is implemented. Each oval represents a symbol in the chromosome. The parent chromosomes have different shading.

Different types of crossovers are also possible. For instance, uniform crossover: uses a mask vector to determine whether a symbol is to be exchanged or not, rather than exchanging at a randomly generated crossover point.

Figure 3.5: Multi-point crossover

3.3.6. Mutation

Mutation is a popular GA operation that randomly modifies the value of one bit in the chromosome. This operator is usually executed with very low probability. Its main use is to ensure that populations do not saturate. The mutation will ensure that new genetic material is being injected in the population possibly uncovering some important characteristics or getting out of local optima. The bit is then transformed as $\chi'(i) = \overline{\chi(i)}$ with a certain probability P_m .

3.3.7. Reproduction

Reproduction is the mechanism used to replace a population with a new generation of individuals. This operates by selection and replacement. Selection is the act of choosing which individuals from this generation will remain in the next generation. Replacement is the process of selecting which of the individuals in the current generation to replace with the selected individuals. Essentially, this is where we select who shall live and who shall die.

Individuals selected for reproduction are normally those with high fitness (as calculated by the user-defined objective function). One way of implementing this is by rank, where the highest ranking individuals are picked for crossover). Another method which is more popular, is the roulette wheel approach where individuals have selection probabilities proportional to their fitness. Figure 3.6 shows an example Roulette Wheel

Figure 3.6: Roulette wheel selection

selection for six individuals with fitnesses $[2, 120, 255, 180, 120, 230]$. Each individual gets a representation on the Roulette Wheel proportional to its fitness. The wheel is spun six times to generate the new individuals. A variant of this selection technique is Baker's multi-arm spinner method where there are as many arms as individuals [6]. The arms are equidistant as shown in Figure 3.7. This is a better method since it reduces the possibility of not selecting low fitness individuals while avoiding early saturation possible by repeatedly selecting the best fit individuals. The Roulette Wheel Selection is also referred to as stochastic sampling with replacement.

Figure 3.7: Baker's multi-arm roulette wheel selection

A third approach is the tournament selection which works by drawing successive pairs using the roulette wheel and picking only the one with the higher fitness. Many other techniques are used [27].

Individuals replaced are either the parents, the individuals with least fitness or individuals that are closest or some similar scheme. There is a main division between two types of replacement. These are discussed next.

3.3.8. Replacement schemes

Once the contributors to the next generation are selected, a replacement scheme chooses to replace some of the old individuals with the new offspring. There are two basic reproductive replacement techniques, generational (simple) reproduction and steady-state reproduction. In generational reproduction, the entire generation is replaced by the new offsprings. In steady-state reproduction, several replacement techniques exist, including replacing the worst current solutions, the best solutions, the parents, the most similar ones or even random replacement. In general, the best steady-state replacements replace the worst or most-similar individuals. Steady-state reproductions tend to converge faster than the generational approach. This can be, however, application-dependent [66].

3.3.9. Initial population generation

Traditional GAs randomly produce the first generation. Some newer applications have allowed for some seeding of the initial population where known good solutions are introduced in the hope that they will achieve better convergence time and produce other good (or better) solutions that are similar to the ones introduced. This seeding is often application-dependent. It is also not guaranteed to produce better solutions and may cause fast convergence to local minima. A new seeding approach will be introduced in this thesis (Section 5.2.4.5).

3.3.10. Fitness

- The fitness of an individual is a relative measure providing more weight to individuals that are closer to the problem solution as calculated by a user-defined objective function. This is application-specific and sometimes may be very difficult to calculate. For complex problems, such as the one in this thesis, the fitness calculation is the major factor in the cpu time required to find a solution.

3.3.11. Scaling

There are different ways of turning the objective function value into a fitness. It can be used as calculated or in a normalized form or it can be scaled. Scaling is often used to maintain diversity (otherwise, extremely fit individuals will dominate the population) or to differentiate between very similar individuals.

Scaling is needed due to the fact that an individual's selection for reproduction is relative to the average population fitness. This does not account for the variance or spread between the least and most fit individuals. If scaling is not performed as the generations get more fit, the probability of the best individuals being reproduced decreases since they are likely to be closer to the mean. Scaling normalizes the population and makes the selection of highly fit individuals much more likely than average individuals.

The scaling can be a simple ranking of all solutions in the generation. It can also be linear, based on the minimum and maximum values in a generation. Another method is sigma truncations where poor individuals are assigned a zero fitness. Yet another technique derates the fitness of individuals that are similar to other individuals in a population. A power scaling was also proposed [27].

Linear scaling is implemented as: $f' = af + b$ where a and b are constants.

Sigma truncation is implemented as: $f' = f - (\bar{f} - c \cdot \sigma)$ where $\bar{f}$ is the average fitness, σ is the standard deviation and c is a constant usually between 1 and 3. Negative results are set to zero.

The power scaling is implemented as: $f^* = f^k$ with k being usually 1.005.

- The scaling methods and the values of the constants are application-dependent.

Scaling experiments for the synthesis problem are performed in Section 8.4.

3.3.12. Dominance, Diploidy and Abeyance

The chromosome as described in Section 3.3.2 is a haploid or single-stranded chromosome. The more complex life-forms in nature however, use the diploid or double-stranded chromosome. Both strands contain the same symbols and the same encoding. The values for specific individuals are different in each strand. The characteristics of the individual are defined by the dominance of each of those values. The usefulness of this seemingly wasteful duplication is abeyance which is the process of shielding the recessive characteristic for potential future use, when the environmental conditions have changed. The concepts of dominance, diploidy and abeyance can and have been used successfully.

Some experiments with diploids are shown in Section 8.7.2.

3.3.13. Inversion

In Section 3.4, schemata will be discussed and it will be clear that specific encodings have better chance of success than others after being subjected to crossover and mutation. In most practical problems, it is not clear and sometimes impossible to provide an encoding within such structures. The inversion operation remedies this problem by providing a method to re-order the chromosome bits. The operator works by randomly selecting two positions in a bit string and flipping their positions as shown in Figure 3.8. One can think of this operation as re-coding of the chromosome on the fly.

3.3.14. Infeasible solutions

One of the main problems encountered in constrained optimization is the presence of infeasible solutions. In GAs, there are several methods to avoid or overcome these.

Figure 3.8: Inversion operator example

These include modifying the problem encoding such that no infeasible solutions are possible, modifying the genetic operators as to not produce any infeasible solutions or using penalty function for infeasible solutions. Removing infeasible solution is also an option when the infeasibility can be determined by a quick check. Methods have also been proposed to repair infeasible individuals.

It is very important not to generate infeasible solutions in synthesis. Section 5.2.4.2 describes the encoding used in this thesis to eliminate infeasible solutions due to data precedence violations.

3.3.15. Isomorphism

Isomorphism is the representational redundancy that can result from problem encodings in GAs. This redundancy means that more than one chromosome (genome) maps to the same solution (phenome). This is quite common in scheduling problems. It is not immediately clear whether this isomorphism provides better or worse solutions. On the one hand, allowing isomorphic chromosomes guarantees more diversity in the population and may allow more building blocks to be joined together to form better individuals. On the other hand, the redundancy implies that the search space of the GA is larger than what it needs to be making it more difficult to converge to the good solutions.

Some problems present isomorphisms that are easy to remove. For example, when extracting cycles in a graph two chromosomes might be $x_1=[1\ 4\ 2\ 5\ 3]$ and $x_2=[2\ 5\ 3\ 1\ 4]$. These represent the same cycle and are easily detected by shifting the chromosome. Many other problems present isomorphisms that are difficult to eliminate. For these, one can use normalization or heuristic techniques to eliminate isomorphism.

3.4. Optimization using GAs compared to other common methods

GAs present a good solution for multi-dimensional optimization problems. As an optimization technique, GAs are usually better than gradient search methods if the search space has many optima. They also have the advantage of inherent parallelism as they simultaneously operate on a large number of solutions.

They appear to be superior to simulated annealing approaches in their inherent parallelism and their history to make use of many previously found solutions. Simulated Annealing methods, however are simpler and easier to modify for new constraints. Experiments were performed in Section 8.6 that compare simulated annealing with genetic algorithms for the synthesis problem.

GAs can outperform integer linear programming (ILP) for large problems where the use of ILP is prohibitive. The ease with which they can be customized makes them a better approach for some problems than the more rigid ILP and heuristic techniques. For problems where the objective functions are not clearly defined, ILP methods are difficult to set up. On the other hand, ILP techniques find optimum solutions. GAs have no optimality guarantee and no optimality checking. The advantage of heuristics is the speed.

In the following sections, we describe the schema theorem that shows how GAs optimize the fitness.

3.4.1. Schemata

A schema η is a string of length l using the alphabet $A \cup *$ where A is the

chromosome's alphabet and * represents a don't care. For a binary alphabet, here are example schema: 0****11 and **0100*

The order of a schema $o(\eta)$ is the number of all positions in η that are not equal to *. For example, $o(0****11) = 3$ and $o(**0100*) = 4$.

The defining length of a schema $\delta(\eta)$ is the difference between the first and last fixed position in η . For example, $\delta(0****11) = 5$ and $\delta(**0100*) = 3$.

3.4.2. Schema Theorem

GAs operate by interchanging symbols amongst the individuals in a population. Each symbol or collection of contiguous symbols (in the chromosome string) is known as a schema. The schema theorem [27] provides the only theoretical explanation of the success of GAs. It shows that short, low order, above average schema receive increasing representation in subsequent generations. This theorem does not guarantee the performance of GAs. It is also unclear how to encode GA problems as to achieve better performance. Most implementations rely only on empirical data.

Theorem 3.1: Short, low order, above-average-fitness schema receive exponentially increasing representation in subsequent generations i.e.

$$m(\eta, t+1) \geq m(\eta, t) \cdot \frac{\bar{f}(\eta)}{\bar{f}} \left[1 - P_c \frac{\delta(\eta)}{l-1} - o(\eta)P_m \right] \quad (3.1)$$

$m(\eta, t)$ is the number of chromosomes containing schemata η at time t .

$\bar{f}(\eta)$ is the average fitness of the strings representing schemata η at time t

$\bar{f}$ is the average fitness of the entire population at time t

P_c is the probability of crossover

$\delta(\eta)$ is the defining length of the schemata

l is the length of the chromosome

$o(\eta)$ is the order of the schemata

P_m is the probability of mutation

We can see from the above that the representation of the schemata in the subsequent generations increases with the fitness of the strings containing the schemata $\tilde{f}(\eta)$. Above average schema will survive, below average schema will die out. Also, the lower the order $o(\eta)$ and the defining length $\delta(\eta)$, the higher the representation in subsequent generations.

Without crossover and mutation, the schema theorem (Equation 3.1) becomes

$$m(\eta, t+1) = m(\eta, t) \cdot \frac{\tilde{f}(\eta)}{\bar{f}}$$

So the growth of a schema is proportional to the ratio of the average fitness of the schema to the average fitness of the population. This is the effect of reproduction alone. Note that this is done for all schemata in parallel.

3.5. Genetic algorithm synthesis survey

GAs are effective multi-dimensional optimization methods since they are not restricted by any application-specific information about the search space. Their use has been proven in many applications. They were used in scheduling problems to optimize DFGs with promising results. Those implementations targeted the scheduling problem only and did not address many practical aspects of the synthesis problems such as register/storage optimization and control-dependent graphs. This thesis presents a similar approach that includes CDFG descriptions (branching), optimizes register allocation and uses practical optimization criteria such as real delays, clock rate and bit widths.

3.5.1. San Martin's work

A synthesis tool based on Genetic Algorithms was developed by San Martin [66]. The tool optimized circuits based on area, latency and power constraints. It used DFGs as its inputs. Optimal results for known problems were reported. The tool was also demonstrated to work on large DFGs. Multiply-unfolded EWF (Elliptic Wave Filter, see Section 7.2.1) DFGs, consisting of up to 170 operations, were successfully synthesized.

The synthesis tool however, had the same limitations as San Martin's in that it did not consider CDFGs (no Branching). Register, mux, interconnect and controller synthesis were also not performed. It was also more restrictive in the allocation process. It essentially performed scheduling using one type of FU per operation type.

3.5.3. Problem Space GAs

The Problem Space GAs method described in [18] presents a different approach to synthesis that uses genetic algorithms. This approach uses GAs to produce problem encodings not solutions. Then, it uses heuristics to solve the synthesis problem. This type of encoding is a method of getting rid of infeasible solutions. What it also does is combine GAs with problem specific heuristics.

In our view, the heuristic scheduling performed after the GA permutations of the problem may work against the GA fitness in the sense that these heuristic methods can easily get stuck in local minima. Using a *regular* GA encoding with an objective function based on real constraints, on the other hand is the closest way to use GAs to solve the real problem.

PSGA was also used to perform simultaneous allocation and scheduling [1]. It was extended to allow the number of allocated resources to be included in the chromosome. This allowed the use of different types of FUs for the same operation. The method then assigned the first available version of each of the units in a greedy fashion. The method proposed in this thesis incorporates the allocation with the optimization. Rather than simply allowing the GA to decide on the number of each unit used, each operation is assigned to some unit and the number of units is totalled. This provides more flexibility and avoids the *greediness* of the other approach.

3.5.4. Simulated Evolution approach

In [50], a Simulated Evolution (SE) approach was used for the synthesis problem. It is a transformational approach that repeatedly rips out and recombines parts of the

design using a probabilistic method. These parts are then reconstructed using application specific heuristics. The formulation performs two distinct steps: *Select* and *Generate*. In the *Select* phase, the costly elements in the current solution are probabilistically discarded. In the *Generate* phase these elements are heuristically replaced. The sequence of element generation affects the results, therefore a heuristic *Priority* is used which is a function of the operation's mobility. An incremental heuristic is applied to a node to determine its assignment.

The advantage of the SE is that it can probabilistically get out of local optima. On the other hand, the *Generate* heuristics tend to drive it quickly into one. The use of Genetic Algorithms on the other hand does not have to be limited by the application-specific heuristics. The presence of the different heuristics also makes the system less flexible than the approach presented in this thesis.

3.6. Conclusion

This chapter presented the basic concepts behind Evolutionary Computation. A general algorithm for Evolutionary Computation was presented. Genetic Algorithms were then introduced and contrasted with other evolutionary techniques. The main difference between Genetic Algorithms and other evolutionary techniques is optimization in the genotype versus phenotype space.

Genetic Algorithms and the different enhancements that can be incorporated in their use were presented. The basic terms and concepts such as population, crossover, mutation, reproduction, fitness, scaling dominance and isomorphism were explained. The schema theorem proves that successive generations will get exponentially increasing representations of chromosome building blocks leading to fitter individuals.

A survey of High-Level Synthesis methods that employ Evolutionary techniques was provided. Two conclusions should be made. First, Genetic Algorithms provide an effective multi-modal optimization technique that is simple and flexible. Flexibility in the context of hardware design is important, due to the ever changing optimization

requirements imposed by emerging technologies. Second, the existing Evolutionary High-Level Synthesis systems either rely heavily on heuristics sacrificing the flexibility and resulting in the same challenges facing purely heuristic techniques, or they simply solve the scheduling problem and do not consider important issues such as control synthesis.

Chapter 4: Spectral Techniques and Complexity Estimation

-

4.1. Introduction

This chapter discusses background information on control unit synthesis and spectral estimation techniques. Those two subjects are grouped together since this thesis presents a novel spatial technique that is used to estimate the complexity of control units in high-level synthesis. The spectral techniques discussed also attempt to estimate the complexity of circuits.

Section 4.2 discusses the techniques of control unit synthesis. The two main approaches (microcode control and hard-wired control) are presented. Hard-wired control is chosen in this thesis since it usually results in faster, smaller and more power efficient controllers.

Section 4.3 discusses the main spectral techniques previously used for complexity estimation. These are the auto-correlation and the entropy methods.

Section 4.4 discusses the ways that incompletely specified functions are handled.

Section 4.5 introduces the idea of using complexity estimation techniques in High-Level Synthesis. This is the gradient method detailed in Chapter 9.

Section 4.6 concludes the chapter and presents arguments for the necessity of estimating the controller size in high-level synthesis.

4.2. Control Unit Synthesis

The control circuit in the synthesized architectures contributes to the design's area and delay. Control units are typically implemented as either ROM-based implementations

or as Finite State Machines (FSM). Optimizing these circuits is difficult since the ROM implementations usually benefit from techniques such as word-length reduction and FSMs benefit from state minimization and encoding. This complicates the estimation process required in HLS to quickly gauge the effect of a controller on the circuit's quality. In this section we present the ROM and FSM implementations [14].

4.2.1. Microcode control

A microcoded implementation of the control logic in straight-line code synthesis requires a number of memory words equal to the latency CC of the circuit in clock cycles. A simple $\lceil \log_2 CC \rceil$ bit counter can address the ROM. Controlling the circuit can be done in one of two ways. In the *horizontal* method, each field in the ROM controls one of the hardware resources such as FUs and SUs. The total ROM size is $CC * (\text{number of control bits})$. In the *vertical* approach, the resource fields are encoded to reduce the width of the ROM. This reduces the overall size of the memory at the expense of additional decode logic. *Microcode compaction* attempts to optimize the circuit somewhere between the two extremes.

Branching operations have to be handled by modifying the simple counter to an address generation unit activated by resource signals. It is easy to see that for more general circuits with data-dependent branches, loops and unbounded latency operations, the complexity of the controller increases, and packing the ROM becomes a more difficult problem. Further, the estimation process gets progressively more difficult.

4.2.2. Hard-Wired Control Synthesis

The hard-wired approach for straight line code is simple. The FSM state transitions are unconditional and can be implemented by a counter. Controlling the circuit is done by decoding the state to activate the appropriate output signals for control. This is the ultimate limit of vertical microcode.

Extending the hard-wired controller to control circuits with data-dependent branches, loops and unbounded operation delays is relatively simple since the specification of the circuit can still follow the execution path of the scheduled resources. Optimizing the circuit is a more difficult problem. It can be much smaller and faster than the microcode. Estimating the contribution of the FSM to the circuit is a difficult problem in either case.

In this thesis we use the FSM model to generate the controller since this results in faster, smaller and more power efficient controllers. We also restrict our investigations of the effect of controllers to the area they require. While their impact on delay is quite important, it is not addressed here. To that end, we investigate the different methods of estimating the complexity of Boolean circuits. The HLS tool that will be presented in Chapter 5 can generate the FSM specifications for each iteration of the tool. A quick estimator will then be used to determine the impact of the FSM on area. That impact is included as part of the overall objective function of the circuit.

4.3. Spectral techniques

This section describes logic-design estimation methods that could probably be used in estimating the controller area in High-Level Synthesis, but have not so far been applied to this problem.

Spectral techniques have been used for digital design (logic design) with success in areas such as compression of test signature and linear decomposition for logic synthesis. These uses were limited to small problems since these techniques have high complexity. Research is under way on providing simpler techniques or powerful calculation methods to make them feasible for larger circuits [74]. No work has been done to apply these techniques to high level synthesis.

4.3.1. Definition

Spectral techniques use transform methods analogous to time-to-frequency transforms such as the Fourier transform in order to represent boolean data as spectral data in the hope of simplifying problems such as logic synthesis.

Many transform domains have been used. They all attempt to represent spectral information of a digital circuit to aid in extracting patterns that simplify problems. Some of these transforms are listed below. All these domains require some form of global computation. Most of these techniques use complexity measures such as the *autocorrelation* method.

4.3.2. Complexity measures

A complexity function should be monotonic. If area increases, the complexity should increase.

If a n -input, m -output boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$ is expressed as:

$$f = \sum_{i=0}^{m-1} f_i 2^i$$

where f_i denotes each of the m outputs and the inputs are defined as:

$$x = \sum_{i=0}^{n-1} x_i 2^i$$

where x_i denotes each of the n inputs, then many complexity criteria $\zeta(f)$ can be defined, on this representation. A good complexity function should have a linear relationship with the circuit area. Without performing circuit layout, the area is usually

assumed to be proportional to the number of literals $L(F)$ where F is the simplified Boolean expression implementing f .

A simple complexity measure is $\zeta(f) = \frac{2^n}{n}$ [38]. To be of practical use, a complexity function should provide a better estimate than $\frac{2^n}{n}$.

4.3.3. Auto-correlation method

The *autocorrelation* is the most often used complexity criterion in spectral synthesis methods [38] [74]. The *autocorrelation* of a boolean function f is defined as:

$$B^{(f,f)}(\tau) = \sum_x f(x)f(x \oplus \tau), \quad 0 \leq \tau \leq 2^{n-1}$$

on the boolean representation above. $(x \oplus \tau)$ is the bit by bit XOR of x and τ expressed in binary. Thus, in the example in Table 4.1, $x=0011=3$, $f(x \oplus 1)=f(3 \oplus 1)=f(2)=0$.

For the minterms at a Hamming distance one away, one needs to calculate $B^{(f,f)}(\tau)$ for $\tau=1,2,4,8\dots 2^{t-1}$.

The function $\psi(f)$, defined below, is then simply a sum of the correlation within the function at a unit Hamming distance. It is basically the minterms that can be combined to form an implicant.

Functions that have most of their minterms at distances of one, have a lower realization complexity. Larger distances are not part of $\psi(f)$. This implies that the autocorrelation function can be used to predict the complexity of a circuit. Such a complexity criterion is [74]:

$$\xi(f) = n2^n - \frac{\psi(f)}{k}$$

$$\text{with } \psi(f) = \sum_{\tau} B^{(f, \bar{f})}(\tau), \tau \in 1, 2, 4, \dots, 2^{n-1}$$

To better estimate functions with easier complementary implementations, we use:

$$\xi' = \xi - \frac{\psi'}{k} \text{ with } \psi' = \psi(\bar{f})$$

where $\bar{f}$ is the complement of f .

As an example, consider the 4 input, 1 output function defined by the following vector: $f=[0,0,0,1,1,1,1,1,1,1,1,1,1,-,1]$, where - is a don't care.

If we assign the don't care to a value of 1, the implementation can be minimized to $f=x_3+x_2+x_1x_0$. This is a literal count of 4.

x		$f(x)$	$f(x).f(x\oplus 1)$	$f(x).f(x\oplus 2)$	$f(x).f(x\oplus 4)$	$f(x).f(x\oplus 8)$
0	0000	0	0	0	0	0
1	0001	0	0	0	0	0
2	0010	0	0	0	0	0
3	0011	1	0	0	1	1
4	0100	1	1	1	0	1
5	0101	1	1	1	0	1
6	0110	1	1	1	0	1
7	0111	1	1	1	1	1
8	1000	1	1	1	1	0
9	1001	1	1	1	1	0
10	1010	1	1	1	1	0
11	1011	1	1	1	1	1
12	1100	1	1	1	1	1
13	1101	1	1	1	1	1
14	1110	1,0	1,0	1,0	1,0	1,0
15	1111	1	1	1	1	1
B(τ)	-	-	12,11	12,11	10,9	10,9

Table 4.1: Autocorrelation calculations

The autocorrelation functions $B(\tau)$ are calculated as shown in Table 4.1 resulting in:

$$B(1)=12, B(2)=12, B(4)=10, B(8)=10.$$

To see how the individual calculations are done consider $f(5).f(5\oplus 1) = f(5).f(4) = 1.1 = 1$. We can think of $5\oplus 1$ as $0101\oplus 0001=0100=4$.

This results in

$$\psi(f) = \sum_{\tau} B^{f \cdot f}(\tau), \tau \in 1, 2, 4, \dots, 2^{n-1} = 12 + 12 + 10 + 10 = 44.$$

and

$$\xi(f) = n2^n - \frac{\psi(f)}{k} = 4 \times 2^4 - \frac{44}{1} = 20$$

If we assign the don't-care to 0, the function can be minimized to $f=x_2\bar{x}_1+x_3\bar{x}_2+\bar{x}_3x_2+x_1x_0$. The literal count is 8. The complexity calculation results in $\xi(f) = 24$. So we can see that the autocorrelation complexity estimate increases for increasing literal complexity.

The *autocorrelation* function can be used to estimate the implementation complexity of a boolean function in a technology independent manner without ordering the inputs and considering complementary implementations. The complexity of this transform, however, limits its usefulness. The *autocorrelation* is not a good complexity indicator for functions that have XOR characteristics. An XOR function results in $\psi(f) = 0$.

This method originally required the function to be fully specified (i.e. with a truth table). In [73], a method was developed to calculate an approximation of this transform

method from a given cover of the boolean function, thus reducing the computational complexity, making it more amenable to automation. The heuristics employed in this approximation were developed for the problem of linear decomposition.

4.3.4. The entropy method

Entropy has also been proposed as a complexity measure [31][12]. It estimates the complexity of a boolean function in terms of computational work. The generalization to multi-output functions was done in [11].

For a function f with n inputs and m outputs, there are 2^n possible input vectors and 2^m possible output vectors. The entropy H of the function f is a function of $P_{ent} = \{P_1, P_2, \dots, P_{2^m}\}$ and is defined as:

$$H(P_{ent}) = \sum_{i=1}^{2^m} P_i \cdot \log_2 \frac{1}{P_i}$$

Where P_i is the probability that output vector $O_i = f$.

$$P_{ent}(f = O_i) = P_i = \frac{n_{O_i}}{2^n}$$

n_{O_i} is the number input vectors producing output O_i .

i is $1 \dots 2^m$

The computational work in a n -input combinational function is defined as [39]:

$$CW = 2^n \cdot H(P_{ent})$$

Assuming then that the circuit area is proportional to the computational work, for multi-output functions, the number of estimated literals (assumed to be proportional to circuit size) is:

$$L_{entropy} = (1 - d) \cdot k \cdot 2^n \cdot H(P_{ent})$$

d is the fraction of don't cares. k is a constant to estimate in the appropriate units (literals, gates, etc.)

Consider the same example function $f=[0,0,0,1,1,1,1,1,1,1,1,1,1,-,1]$.

First we calculate the entropy of $f=[0,0,0,1,1,1,1,1,1,1,1,1,1,1]$

$$P_1 = 13/16 \Rightarrow H(13/16) = 0.6962$$

Now we calculate the entropy of $f=[0,0,0,1,1,1,1,1,1,1,1,1,0,1]$

$$P_1 = 12/16=0.75 \Rightarrow H(3/4)= 0.8113$$

Recall that the first results in 4 literals and the second 8. The entropy increases correspondingly.

The entropy of the original function containing the don't care can also be determined by adjusting the probability of 1's as follows

$$P_1 = \frac{n_{0_1}}{2^n - n_d}$$

For this function $P_1 = 12/(16-1)=12/15$. This results in an entropy $H(12/15)=0.7219$ which is somewhere between the two values previously calculated.

The entropy method is a statistical method and works for randomly generated functions. Its disadvantage is that it does not take into account any neighborhood information. Many functions produce the same percentages of *Is*, *Os* and *don't cares* and vary dramatically in implementation. The entropy method cannot track this variation. This will be illustrated by the quality of the results shown in Section 9.5.

4.4. Handling incompletely specified functions

The autocorrelation literature does not clearly indicate how incompletely specified functions are implemented.

For the entropy measure, the number of don't cares in a function is included in the probability as follows (This was used in the example in Section 4.3.4):

$$Pr(f = O_i) = P_i = \frac{n_{O_i}}{2^n - n_d}$$

where n_d is the number of don't cares in the truth table.

The entropy estimates in [11] were performed for different percentages of don't cares and tracked well.

4.5. High-level Synthesis and a New complexity measure

After considerable search, we have been unable to find a High-level Synthesis system has used formal spectral transforms. Their complexity is a disadvantage for the size of problems high-level synthesis involves. Their potential use as structure complexity evaluation can be useful, however.

In Chapter 9, we will propose a new method similar to spectral complexity

techniques we refer to as the *Gradient*. The *Gradient* is a spatial (rather than spectral technique). It is of linear complexity in the size of the cover and therefore easier to calculate than the auto-correlation.

We will show results that strongly indicate that the gradient method is a better estimate than the autocorrelation or entropy methods.

4.6. Conclusion

This chapter discussed background information on control unit synthesis and spectral estimation techniques.

The two main approaches to control unit design are microcode control and hard-wired control. Hard-wired control was chosen in this thesis since it usually results in faster, smaller and more power efficient controllers.

The two main spectral techniques used for complexity estimation were presented: the auto-correlation and the entropy methods. The autocorrelation has a large computational complexity since it is order independent and based on the proximity of the inputs vectors. The entropy method is statistical and does not incorporate any neighborhood information.

The gradient method introduced in this thesis (Chapter 9) helps overcome these problems and experimentally provides better estimates than the ones found using auto-correlation and entropy. The intent is to use this method to estimate the control unit area in high-level synthesis.

Chapter 5: Genetic Algorithm Synthesis

5.1. Introduction

The Genetic Algorithm synthesis method described in this thesis is a flexible method capable of efficiently synthesizing Control/Data Flow Graphs allowing for code motion and optimization of different paths. The use of Evolutionary Computation allows for a more natural evaluation of the resulting architectures without the need to simplify the problem. It also allows the use of multiple architectural techniques that are not normally utilized such as multi-rate synthesis, simultaneous clock selection and simultaneous storage optimization. Performing the allocation, scheduling and binding steps simultaneously allows the investigations of more architectural trade-offs than possible when restricting the execution of these steps. The use of this method is an improvement over more classical synthesis tools as it is easy to modify and adapt to new constraints and is efficiently scalable. Flexibility in the optimization criteria is essential as the technological assumptions and issues of concern are constantly changing. Everything presented in this section represents the original implementation used in this thesis.

Section 5.2 describes the GA synthesis implementation. We formulate the problem, describe the CDFG input, discuss the hardware availability and cost, and present the GA setups and techniques. We also present the encoding method. A relative clock cycle encoding is used to minimize infeasible solutions. We further evaluate the encoding by examining certain properties believed to help in the application of GAs to optimization problems. We find that the encoding used is in line with the state-of-the-art recommendations.

Section 5.3 describes the functional and storage units used in the synthesis hardware library. We present a unified approach to the hardware unit description.

Section 5.4 details the objective function calculation used. We show how

chromosomes are decoded into architectural descriptions and how the different hardware costs such as area, delay and power are calculated.

Section 5.5 describes the implementation of synthesis for control-intensive circuits. We investigate different methods of representing branching information in the GA synthesis method then present a coloring scheme used for mutual exclusion testing. We further show how data-dependent loops can be synthesized using this methodology.

Section 5.6 concludes the chapter and summarizes its major contributions.

5.2. Basic genetic algorithm synthesis

This section describes the synthesis methodology used in the thesis.

5.2.1. Problem formulation

The objective of the GA high-level synthesis tool is to achieve a near optimal hardware implementation of an algorithm presented as a CDFG (Control/Data Flow Graph), given a library of available Functional Units (*FUs*), Storage Units (*SUs*), Topology Units (*TUs*) and Input/Output units (*IUs*). The optimum is defined in terms of performance criteria such as algorithm latency, delay, area and power. The basic synthesis encompasses scheduling of the operations, allocation of hardware units and binding.

5.2.2. CDFG description and input

The input to the synthesis tool is a CDFG graph such as the one shown in Figure 5.1. This graph is parsed and run through setup algorithms to obtain a description which can be denoted as $G(V, E, O, P, C, X, M)$ whose components are defined in Table 5.1 and the text that follows. Table 5.1 shows a summary of the sets and lists described above for reference.

Figure 5.1: Example Control/Data Flow Graph.
 Dashed arrows represent control precedences. C_x represents conditional operations, n_x represents non-conditional operations. e_x denotes all edges.

Set	Explanation
V	Set of nodes (operations)
E	Set of edges (Precedences and control precedences)
K	Set of HW unit types (kind)
O	Set of operations to be scheduled
P	Set of hard precedences
C	Set of conditional (soft) precedences
X	Set of operation colors (Mutual exclusion list)
M	Set of mobilities (e.g. ASAP->ARAP)

Table 5.1: Summary of Sets and Lists

- **V is the set of vertices representing the operations in the graph (including ϕ).**

- For the example, $V = \{\phi, c_1, n_1, n_4, c_2, n_2, n_3, n_5\}$

- **E is the set of edges representing data and control precedences.**

For the example $E = \{e_1, e_2, e_3, e_4, \dots, e_{15}\}$.

The remaining sets consist of:

- **O is an operation list:**

$O(v, k)$ where $v \in V, k \in K$

This list describes the types of operations in the graph, where:

- **K the set of all possible operation types (kinds) in the CDFG.**

The operations can include additions, multiplications, comparisons, shifts, multiply-accumulates, or any other operation that can be executed on the available functional/storage unit, regardless of complexity.

The ordering of the operations in a CDFG is arbitrary and does not affect the synthesis. The ordering choice used is described later. The position of the bit sequences in a chromosome does affect the performance of the GA algorithm to a certain degree since the resulting schemata vary widely, but this is highly application-dependent and cannot be easily predicted. It also varies with optimization constraints as well as the algorithm described.

The synthesis methodology requires a database of hardware units from which the synthesizer can choose. A type is defined as a FU that can perform a certain function on

data of a certain width at a clock frequency. Each of those unit is described using a cost list that is used in determining the fitness of solutions. This is described in Section 5.3.

Operation types $k(f,b,r)$ contain all the information needed about the operation:

- f is the function, b the number of bits and
- r the clock rate for multi-rate systems.

For the example in Figure 5.1:

$$O = \{(c_1, k_1), (n_1, k_2), (n_4, k_3), (c_2, k_1), (n_2, k_3), (n_3, k_2), (n_5, k_2)\}$$

with

$$k_1 = (>0, 1 \text{ bit}, 100\text{MHz})$$

$$k_2 = (+, 8 \text{ bit}, 100\text{MHz})$$

$$k_3 = (\times, 8 \text{ bit}, 100\text{MHz})$$

$$k_4 = (+, 8 \text{ bit}, 100\text{MHz})$$

- P is a precedence list:

$$P(v,w) \text{ where } v,w \in V, e_{vw} \in E$$

This list describes the data precedences in the graph, where V is the set of operations (nodes), E the set of precedences (edges).

Let v and w be operations and (v, w) be a data precedence relation meaning that the input data for w is not ready until v calculates it. This separates it from conditional precedence. The problem is choosing hardware h_v, h_w to execute v, w and scheduling them so that $(t_w > t_v + d_{hv} + d_r + d_l) \forall (v,w) \in P$ and a cost function is minimized.

- ***M* is a mobility list:**

$$- M(v, [t_{vmin}, t_{vmax}]), t_{vmin} < t_{vmax}$$

where t_{vmin} and t_{vmax} are the timing bounds on operation v such that all feasible solutions will satisfy $t_{vmin} \leq t_v \leq t_{vmax}$. t_{vmin} is the earliest possible time the operation is allowed to be scheduled, which usually corresponds to the earliest time the operation can be scheduled due to precedence constraints. t_{vmax} is the latest possible time the operation is allowed to be scheduled which usually corresponds to the latest time the operation can be scheduled to achieve a certain constraint (minimum latency, minimum hardware used etc...). One way to select this boundary is by using the “as soon as possible” (ASAP) and “as late as possible” (ALAP) schedules where $t_{vmin} \in T_{ASAP}$ and $t_{vmax} \in T_{ALAP}$. Note that this ALAP schedule is the most constrained (in time). This implies that the execution time for ASAP is the same as ALAP. To relax the constraint we extend the schedule to $T_{ALAP} + \theta$. This is a good choice for delay optimization. For area optimization or resource-constrained scheduling, a larger interval may be required as limited mobilities restrict time sharing. To perform this we can allow solutions to extend beyond T_{ALAP} to use $T_{ALAP} + \theta$, or to use $[T_{ASAP}, T_{ARAP}]$ where T_{ARAP} represents the minimum resource schedule based on ASAP. Note that ALAP can be stretched to the required arrival time in timing-constrained scheduling. For the example shown, a possible M might be

$$M = \{(c_1, [1,1]), (n_1, [1,2]), (n_4, [1,2]), (c_2, [2,3]), (n_2, [2,4]), (n_3, [2,4]), (n_5, [2,3])\}$$

5.2.3. Hardware availability and cost description

5.2.4. Genetic algorithm setup

Given the above description, the solutions are chromosome that respect the data precedences. These are selected according to their fitness. The GA is used to generate

solutions that will have higher fitness with successive generations. The evaluation of those solutions is an application-dependent problem and is solved by multiple algorithms and heuristics. This section describes the GA encoding and the various algorithms employed.

5.2.4.1. Encoding

The chromosome encoding is made up of all the information that needs to be optimized for each operation, namely, the time the operation is executed, the hardware unit on which it is executed and, for multi-rate systems, the clock rate for the operation. When performing simultaneous binding, a HW id is also used corresponding to the actual instantiation of the hardware units.

From these values all required optimization criteria γ can be extracted:
 $\gamma(hw, cc, rate)$

where *hw* is a field designating the *id* of the *FU* used

cc is the relative clock cycle for the operation initiation based on the mobility

rate is the clock *rate*.

All other values needed for optimization but that do not need to be optimized themselves are maintained in dynamic lists.

A relative clock cycle is used to reduce the size of the chromosome and more importantly eliminate all the infeasible solutions due to violations of precedences. The chromosome encoding is shown in Figure 5.2 where the chromosome section corresponding to each operation is shown along with the derivation of the size of each component. The HW and Template lists will be discussed in Section 5.3.

The number of bits used to encode a given CDFG is:

$$\# \text{ bits/operation} = \text{hw} + \text{cc} + \text{period} + \text{id}$$

Figure 5.2: Chromosome Encoding for multi-rate systems

$$num_{bits} = \sum_{v=1}^V (\log_2 K_v + \log_2 \mu_v + \log_2 rate_v + \log_2 id_v)$$

where K is the type of possible hardware that can execute the operation and V is the number of all operations.

If, for the example in Figure 5.1, there were 2 types of adders/subtractors, a single type of multiplier, and a single type of comparator, with a fixed 100 MHz rate (which implies that the rate field is not used), the chromosome encoding would be as shown in Figure 5.3. Note that unused variables are not represented in the chromosome bits. Since only 1 multiplier and 1 comparator are available, there are no hw fields for n_4 , C_2 and n_2 .

Figure 5.3: Example chromosome encoding.

Operations, parameters and the number of bits per parameter for example in Figure 5.1.

The number of bits in the *cc* fields is determined by the mobility of the operations as per the example mobilities presented in Section 5.2.1, i.e.:

$$M = \{(c_1, [1,1]), (n_1, [1,2]), (n_4, [1,2]), (c_2, [2,3]), (n_2, [2,4]), (n_3, [2,4]), (n_5, [2,3])\}$$

$$set\ of\ num_{bits} = \{(c_1, 0), (n_1, 1), (n_4, 1), (c_2, 1), (n_2, 2), (n_3, 2), (n_5, 1)\}$$

c_1 does not appear in the chromosomes since its mobility is 0 (assuming no out-of-order and tight stop time). Example scheduled graphs are shown in the decoding section (Section 5.4).

5.2.4.2. Relative clock cycles

The encoding described in the previous section used a clock-cycle definition based on the mobility not a fixed clock cycle for each operation. This is known as relative clock cycle encoding. Rather than assigning an absolute clock cycle to each scheduled operation, a relative clock cycle (relative with respect to the completion time of the operation's latest precedent) can be used. Note that we ensure the operations are decoded in successive order so we identify and track the latest precedent. (We also account for register-transfer delays)

The advantages of this encoding method are:

- Minimize infeasible solutions: If absolute timing intervals are used, the GA will produce many chromosomes which violate data precedences. For instance, if a random solution assigned n_5 to clock cycle 2 and n_4 to clock cycle 3, then the precedence defined by (n_4, n_5) is violated. These solutions would be declared infeasible and not used. Alternatively, the relative encoding scheme can be used, and all clock cycles are calculated relative to their precedences and all solutions can be guaranteed to be time-feasible by construction. This appears simpler than the other methods used to resolve this problem. These include using special crossover operators, using problem-space GAs, permutation encoding or penalty functions.
- Reduce the number of bits per chromosome: Using a relative encoding minimizes the number of bits used to encode the clock cycle symbol. In large algorithms, some operations may have to be scheduled at absolute cycles in the hundreds, this would require a large number of bits. This number impacts the convergence time of the algorithm. The larger the number of bits in a symbol, the larger the design space and the longer the GA will need to run to obtain good solutions.

The GA tool employs a variety of the relative encoding schemes explained below.

Unbounded relative encoding

In the first method, *unbounded relative encoding*, the clock cycle assignment is allowed to exceed the *alap* (or *arap*) value.

$$t_v = \underset{u}{MAX}(t_u) + cc_v$$

where $MAX(t_u)$ is the latest scheduled precedent for operation v and cc_v is the relative clock cycle value as decoded from the chromosome with $cc_v \leq alap_v - asap_v$. This method results in the later operations having more probability of exceeding their *alap* value. The reason this can extend beyond the limit is the possibility that

$$MAX_u(t_u) > asap_v .$$

-

This is a good method for graphs that get progressively less mobile under the CC_{ALAP} limits.

Bounded relative encoding

If one has a good bound for $arap$ or the $alap$ limit is satisfactory (or required), then we have to limit each operation to this value. We can use a *bounded relative encoding*:

$$t_v = MAX\left(alap_v, MAX_u(t_u) + cc_v\right)$$

This ensures we are within the limit but ends up with a non-proportional assignment of codes in the gene. Populations may saturate quickly. This is due to the fact that more genes will result in values greater than the $alap$ value and all succeeding operations will be forced to their $alap$ values as well. A better encoding, is the circular relative encoding.

Circular relative encoding

$$\begin{aligned} t_v &= MAX_u(t_u) + cc_v & \text{if } \left(MAX_u(t_u) + cc_v\right) &\leq alap_v \\ t_v &= MAX_u(t_u) + \left\lceil \frac{cc_v}{alap_v - MAX_u(t_u)} \right\rceil & \text{if } \left(MAX_u(t_u) + cc_v\right) &> alap_v \end{aligned}$$

This guarantees the precedences and the limits and provides proportional encoding in the gene. This means that we are less likely to see genes revert to the $alap$ value than the previous method.

Note that in all of the above, the relative value cc_v has to be limited to the mobility for non-power of 2 mobilities in a similar manner to the circular encoding. i.e.

$$cc_v = \left\lceil \frac{2^{cc_{gene_v}}}{alap_v - asap_v} \right\rceil \quad \text{if} \quad 2^{cc_{gene_v}} > alap_v - asap_v$$

Note also that in this circular method, no updates of the mobilities are necessary as opposed to the implementation in [30], where the relative encoding was implemented as follows:

$$t_v = asap_v + \lceil \zeta_v \cdot (alap_v - asap_v - \delta_v) \rceil \quad \zeta_v \in [0, 1]$$

Then, the $asap$ and $alap$ values of all other operations are updated as follows:

$$asap_v = MAX(asap_v, t_u + distance_{(u,v)})$$

$$alap_v = MIN(alap_v, t_u + delay_u - distance_{(v,u)})$$

$delay$ is the operation's delay and $distance$ is the minimum precedence required between the operations.

Another advantage of the relative clock cycle encoding method is that it is by nature amenable to be used for *relative scheduling* described in Section 2.6.5 where unbounded operations can be scheduled only in a relative manner.

5.2.4.3. Evaluation of the encoding

To validate the encoding used we evaluate it against the properties discussed by Simon Ronald in [65]. These properties are collected from many researchers in the field and encompass both theoretical and practical aspects of GA encoding. They represent the best known, current recommendations. The properties are listed here followed by

comments regarding the encoding used in this thesis.

- The encoding:

- *embodies the fundamental building-blocks that are important for the problem type.*

We believe that the encoding embodies the building-blocks that are important. All the variables encoded in the chromosome are ones that have to be optimised. No extraneous data is included.

- *is amenable to a set of genetic operators that can propagate these building blocks from parent genotypes through to the children genotypes during child generation*

The binary encoding and the use of the crossover operator guarantee this. The use of crossover at the symbol boundary guarantees that the building blocks can be passed without alteration to the children (affected only by the mutation).

- *is not subject to epistasis where the effect of one gene suppresses the action of one or more other genes.*

The genes are inter-dependent but no gene suppresses the action of others.

- *allows a tractable mapping to the phenotype, allowing fitness information to be calculated in the minimum number of steps.*

The decoding process is linear and simple.

- *exploits an appropriate genotype-phenotype mapping if a simple mapping to the phenotype is not possible.*

Not applicable.

- *embodies feasible solutions if possible - penalty systems or repair strategies are options if illegal solutions are produced.*

The use of relative clock cycle encoding ensures that most of the potentially infeasible solutions are avoided. The only remaining infeasible solutions are related to hard constraints (in time or resource). These are unavoidable and the use of penalty functions is effective in overcoming them.

- *suppresses isomorphic forms, i.e. many genotypes that map to the one problem point.*

This is very difficult to accomplish in the current encoding. Permutation encoding would have partially resolved this problem as we could have easily eliminated isomorphisms caused by the scheduling. It is not clear, however that removing isomorphic forms would be beneficial for this problem.

- *uses gene values taken from an alphabet of the smallest possible cardinality, with binary encodings considered the best if suited to the problem.*

Binary encodings are used. With the exception of having to handle non-power of two genes, the encoding seems natural.

- *represents the problem at the correct level of abstraction ranging from a completely specified point in the problem space or a set of aggregate qualities that describe a family of possible solutions.*

We believe the level of abstraction is appropriate for this problem. The chromosome is a description of a scheduled and bound graph. This is a good description of the resulting architecture.

For the above checklist that the chosen encoding appears well in-line with the state of the art recommendations for evolutionary computation encoding.

5.2.4.4. Chromosome decoding

- The encoding of the chromosome is such that an operation's hard precedences occupy preceding locations in the chromosome. Thus the decoding takes the operations in order and evaluates them. In other words, the greedy approach is guaranteed to produce optimal decodings of each chromosome due to the encoding scheme. Various other heuristics are used to evaluate each chromosome. These are described in detail in Section 5.4.

5.2.4.5. GA operators

The selection technique of choice for this thesis is the roulette wheel approach (Section 3.3.7). The replacement scheme is a steady-state one. The crossover points are chosen at the symbol boundaries, not at individual bit boundaries. The symbol boundaries are maintained after the initial encoding. The mutation operator is the only other operator used. The stopping criteria is the number of generations. The effect of choosing other methods is discussed in Chapter 8.

As far as seeding is concerned, we use a novel [0,1]-seeding. It is known that solutions like "000...00" or "111...11" are difficult to obtain due to the large defining length of the schema (Chapter 3.4.2). While generating the initial population, a certain percentage of the population can be artificially cloned from all-0 and all-1 chromosomes. This will help obtain those good solutions that have long schemata. Note that an all-0 solution corresponds to the ASAP on the first available HW units. The all-1 solution depends on the mobilities and the relative encoding scheme.

In Chapter 8 we investigate different methods of obtaining a good population before applying the appropriate fitness function in the optimization.

5.2.4.6. GA inputs/outputs

- The GA requires input $GA(z, g, P_c, P_m, r)$, where:

- **z is the population size.**

This has a direct impact on CPU time. A larger population size increases CPU time but provides better average solutions. Typical values for high-level synthesis problems range from a hundred to several thousands depending on the size of the circuit and the optimization criteria.

- **g is the maximum number of generations.**

This has a direct impact on CPU time. Typical problems of this magnitude are sometimes run up to thousands of generations. Often, however, a good solution is found within a hundred. Most of the experiments run during this thesis work were limited to hundreds.

- **P_c is the crossover probability.**

This sets the probability with which a member of the population will be crossed with another member. Typical values range from 20 to 60%. The crossover is normally done at the symbol boundary.

- **P_m is the probability of mutation.**

This is the probability that a bit of the chromosome will change its value. The higher the probability the more diverse and new each population will be. Typical values are 0.1 to 1%.

- **r is a random seed.**

The GA outputs are $GA(i,f)$ where:

- i denotes an individual in a generation.
- f its corresponding fitness.

The best solution is the one with the highest fitness value f .

5.3. Functional/Storage Units

Functional units are hardware units that implement a certain logical or arithmetic function. Storage units (SUs) are hardware units that store values until they are no longer required (e.g. Registers, RAMs). The use of SUs is explained in Section 6.7. Both FU and SU costs are described by a list $Cost(type,id,area,delay,power,ip,op,bits,mem,util,freq)$ where:

- $type$ is the type of F/SU. e.g. add, multiply, shift, compare, table-lookup, register, RAM.
- id is the identification or name of the F/SU often denoting a specific architecture. (Booth, Wallace...)
- $area$ is the area of the F/SU. (in true or relative *nand* equivalent gates or in mm^2 of die area).
- $delay$ is the delay or latency of the F/SU. (maximum delay in ns)
- $power$ is the power of the F/SU. (Average worst case power per cycle of F/SU, or a function of frequency power)
- ip is the number of input ports.
- op is the number of output ports.
- $bits$ is the number of bits (or width) of the F/SU.
- mem is the number of internal storage elements.
- $util$ is a pipelining, local resource utilization list identifying the pipeline stages and potential hazards.

- $freq$ is the clock frequency range over which the F/SU can operate.

This description can be used for all F/SUs regardless of their complexity, e. g. a DSP, an adder, a register or a multi-port RAM. It can also be used to describe non-functional units like storage elements such as registers and RAMs. Note that we are not describing the behavior of the unit, rather only what we need for an operation's execution. For instance we can easily say that implementing an addition on a DSP can be completed in 10ns consuming 1mW of power with the whole DSP area being used. We do not need to know the internal behavior of the DSP.

Figure 5.4: General Functional/Storage Unit description

5.3.1. Example F/SUs

Figure 5.4 (a) shows a graphical representation of the generic Functional/Storage unit. The unit has a variable number of inputs and outputs with variable widths. The unit has a number S of storage or pipeline locations and functional stages. The description of a

F/SU is general and can be applied to any hardware unit.

- For instance, the adder shown in Figure 5.4 (b) can be described as:

*Cost(+,Carry Look-ahead-2,461 gates, 9.98ns, 230*F mW/MHz, 2 inputs, 1 output, 32 bits, 0 mem, non-pipelined, 10-100MHz).*

The multi-port register file shown in Figure 5.4 (c) can be described as:

*Cost(R,Register-File-5,2120 gates, 5.3ns, 565*F mW/Mhz, 2 inputs, 2 outputs, 32bits, 10 mem, non-pipelined, 10-100MHz).*

The only difference between storage and functional units is in their lifetime. An operation executed on a FU will be completed d_{FU} time after initiation (fixed pre-determined latency), but the lifetime of the operation assigned to a SU is dependent on the time the stored variable will be required. This is not known until the scheduling is done.

5.4. Objective function calculation

This section describes the techniques used to evaluate each chromosome's fitness. The different optimization criteria are described first. Then, the whole objective function is presented.

5.4.1. From bit strings to architectures

The GA produces many bit-strings during the optimization process, each of which represents an architecture that has to be evaluated. Here we show examples of bit strings and their meaning.

Figure 5.5 shows two examples. We see the chromosome as described before and

two example strings (the symbol values here are shown as integer values. In the implementation, they are binary). We can also see how these strings map to a scheduled graph.

Consider operation *C2* in Figure 5.5 (a). The relative clock cycle value for it is 2. Knowing that *n1* is the latest scheduled precedent of *C2*, we assign *C2* to a time slot two clock cycles after *n1*. *C2* does not have any hardware bits since there is only one type of comparator.

Figure 5.5: Example decoded chromosomes

These tables simply concern scheduling and allocation. To complete the binding, one has to assign the FU types to unique instances, select different inputs and add multiplexers.

5.4.2. Chromosome Decoding

Figure 5.6 shows some of the decoding procedures such as Area and Delay derivations. The first step involves decoding the chromosome's information for each operation (hw , cc , $rate$). The hw data and the $rate$ data are then interpreted through the operation and hardware lists to obtain the resource costs for the operation. The cc data provides the relative clock cycle information. This, in conjunction with the information on the previously scheduled precedences and the operation's mobility combines to produce the absolute clock cycle as described in Section 5.2.4.2. The area, power and latency calculations are shown in the following section.

5.4.3. Area cost

The size of the design is calculated in gate count as:

$$TotalArea = FUarea + SUarea + TUarea + IUarea + CTLarea$$

where FU, SU, TU, IU and CTL represent functional units, storage units, topology units (multiplexers and buses), I/O units and the controller respectively.

5.4.3.1. FU area

The total functional unit area is:

$$FUarea = \sum_{i=1}^{CC} \sum_{j=1}^{NFU} (FUnum_{ij} - FUm_{ij}) \cdot FUarea_j \cdot FUshare_{ij}$$

where

CC is the total latency of the graph in clock cycles.

Figure 5.6: Chromosome decoding and performance evaluation.

NFU is the number of all FU types used.

$FU_{num_{ij}}$ is the number of FUs of type j used in clock cycle i .

$FU_{mut_{ij}}$ is the number of FUs of type j used in clock cycle i that operate on mutually exclusive data.

$FUarea_j$ is the area of FU of type j in gates, transistors or mm^2 .

$FUshare_{ij}$ is a time-sharing variable. $FUshare_{ij} = 0$ implies the operation will use one of the FUs that have already been allocated in previous cycles. $FUshare_{ij} = 1$ implies that we need to allocate a new FU for this operation.

This is of $O(n)$ complexity. n being the number of operations.

5.4.3.2. SU area

The SU area is also calculated in the same manner, however, it is of $O(n^2)$ complexity in the number of operations. Namely, the SU area is:

$$SUarea = \sum_{i=1}^{CC} \sum_{j=1}^{NSU} \left(\frac{1}{SUstorage_j} \right) \cdot (SUnum_{ij} - SUMut_{ij}) \cdot SUarea \cdot SUshare_{ij}$$

where

NSU is the number of all SU types used.

$SUstorage_j$ is the number of storage elements in the SU of type j . (e.g number of registers in a register file)

$SUnum_{ij}$ is the number of SUs of type j used in clock cycle i .

$SUMut_{ij}$ is the number of SUs of type j used in clock cycle i that operate on mutually exclusive data.

$SUarea_j$ is the area of SU of type j in gates, transistors or mm^2 .

$FUshare_{ij}$ is a time-sharing variable. $FUshare_{ij} = 0$ implies the operation will use one of the FUs that have already been allocated in previous cycles. $FUshare_{ij} = 1$ implies that we need to allocate a new FU for this operation.

The decoding of the chromosome is first performed linearly and the SU information is stored. Once all operations have been assigned to clock steps and bound to SUs, a traversal of the resulting schedule is performed to calculate the SU area.

Note that if the lifetime is 0, then the SU degenerates into a wire and $SUarea = 0$.

- Note that the selection of the actual storage unit is implied. When loops are considered, care has to be taken in the register assignments.

$TUarea$ and $IUarea$ are calculated in the same manner since TU and IU units are described and used in the same manner as any functional/storage operations. These units are simply assigned based on the F/SU allocation.

$CTLarea$ is estimated using the gradient technique introduced in Chapter 9.

5.4.4. Power cost

The system calculates both the maximum instantaneous power (per clock cycle) and the average power dissipation over the whole algorithm execution. When the FU/SU power is given as a function of the frequency F , the instantaneous and average powers are:

$$PeakPower = \max_{i=1}^{CC} \left(\sum_{j=1}^{NFSU} (FSU_{num_{ij}} - FSU_{mut_{ij}}) \cdot FSU_{power_j} \cdot FSU_{freq_j} \right) + CTL_{power}$$

and

$$AvgPower = \frac{\sum_{i=1}^{CC} \sum_{j=1}^{NFSU} (FSU_{num_{ij}} - FSU_{mut_{ij}}) \cdot FSU_{power_j} \cdot FSU_{freq_j} + CTL_{power_i}}{CC}$$

$NFSU$ is the number of all F/SU types used.

$FSU_{num_{ij}}$ is the number of F/SUs of type j used in clock cycle i .

$FSU_{mut_{ij}}$ is the number of F/SUs of type j used in clock cycle i that operate on mutually exclusive data.

$FSUpower_j$ is the power consumption of F/SU of type j in W/MHz .

$FSUfreq_j$ is the frequency of the F/SU.

- $CTLpower_i$ is the power consumed by the control circuitry in cycle i .

5.4.5. Latency cost

The last control step in the CDFG determines the latency of the CDFG in clock cycles. This is multiplied by the clock period to obtain the circuit execution latency in ns . The clock period is selected as explained in Section 6.4.

For control-intensive schedules, one may be interested in the latencies in different paths in the graph corresponding to different conditions. The corresponding ending operations have to be identified, then a cost measure describing the required minimizations has to be included. This allows minimizing all possible latencies in the graph or averaging out latencies or even ranking some as more critical than others.

It is possible to optimize for the longest possible path in the CDFG, for the average path length or for specific paths. This is essential for control applications that may have a large variation between the fastest and slowest conditional paths.

Optimizing for different control paths requires calculating the latency of each of those paths. When optimizing for the average delay for example we consider:

$$AvgLatency = \frac{\sum_{paths \in PATHS} \sum_{op} ccrel_{op} \times clk_{op}}{\|paths\|} \quad paths \subseteq PATHS$$

$PATHS$ is the set of all conditional paths and $\|paths\|$ is the number of all considered paths (Cardinal number of $paths$). $ccrel$ is the clock cycle of the operation relative to its latest precedent and clk is the clock period.

5.5. Control Synthesis

- This section describes the graph coloring needed to accommodate control (conditional) operations and perform speculative scheduling and out-of-order execution.

5.5.1. Different Control Synthesis Options

Many methods were investigated for possible implementation in the CDFG synthesis. The coloring method used in this thesis, and explained in Section 5.5.2, colors the nodes of the graph for easy mutual exclusion testing and encodes the chromosome as described earlier. Another alternative that was considered is the idea of splitting chromosomes. In this method, the chromosomes are split according to the different paths in the graph as shown in Figure 5.7. This results in three distinct sections. The first contains the common operations, the second, the mutually exclusive operation and the third consists of the recombinant portions with are joint but different than the common ones since they depend on the synthesis of the mutually exclusive portions. One can schedule the common portions first, then the mutually exclusive portions and later the combined portions.

Handling a general chromosome of this type might be best addressed by Genetic Programming techniques where the structure is typically a tree. However, the rules for modification of the scheduling trees are limited. As the different strands are not totally independent (when implementing out-of order execution for instance), these structures have to be modifiable. Figure 5.8 shows an example modification of the original chromosome (a) when the gray operation is executed out of order (b) and the gray and black operations are executed out of order (c). Note that the actual order of the genes in this encoding does imply execution order. It is simply the location of a gene within a strand that determines its mutual exclusion with other genes.

The additional overhead of creating and using an efficient operator to perform

Figure 5.7: Chromosome splitting example

these modifications in conjunction with the other GA operators makes the coloring approach more attractive. Nevertheless, there might be some merit in this type of approach.

5.5.2. CDFG coloring

The following describes three original coloring variations proposed in this thesis.

Definition 5.1: Two operations are mutually exclusive if they belong to two branches of a conditional operation that are activated by different conditions.

To detect mutually exclusive operations, branches in a CDFG are colored according to their conditional paths in a manner to quickly identify mutual exclusion.

Figure 5.8: Out-of order representation using chromosome splitting

For each condition-dependent operation, a color c_{η} is selected to identify the path for the operation:

$$clr_v = b_0 \cdot b_1 \cdot \dots \cdot b_{t-1} \cdot b_t$$

where $[0-t]$ represents a chain of conditionals. b_i represents a digit of the resulting color corresponding to the i th conditional operation in the chain. Figure 5.9 (a) shows a CDFG with the pre-scheduling coloring. The mutual exclusion rule is:

- If the colors of two different nodes are truncated from the right until they are of the same length (reduced) they are mutually exclusive if they are not equal.

For any two operations with colors clr_x and clr_y , where $\iota_x = \|clr_x\|$ and $\iota_y = \|clr_y\|$, $\iota_{min} = MIN(\iota_x, \iota_y)$. The reduced colors are

$$c\bar{l}r_x = b_{x0} \cdot b_{x1} \cdot \dots \cdot b_{x_{t_{min}}}$$

and

$$c\bar{l}r_y = b_{y0} \cdot b_{y1} \cdot \dots \cdot b_{y_{t_{min}}}$$

The two operations are mutually exclusive if $c\bar{l}r_x \neq c\bar{l}r_y$.

So, in Figure 5.9, c_1 introduces the first branch in the graph and will have an arbitrary color of 0. Everything on its *true* branch will be tagged with 1 everything on the *false* branch with two. This process is continued for the whole graph.

The coloring scheme is not limited to the digits 0, 1 and 2, thus a node may have multiple conditional paths (e.g. a case statement).

Figure 5.9: Coloring example: pre-schedule

5.5.3. Color updating

-

To allow out-of-order execution, operations have to be scheduled prior to their conditional precedents. For such operations, (e.g. n_2 in Figure 5.10 scheduled prior to C_2) the mutual exclusion status has to be changed. Therefore, any time an operation is scheduled (which happens as we are decoding the chromosome) its color has to be updated.

If an operation is assigned to a clock cycle after all its conditional predecessors, then it retains its old color:

$$sclr_n = clr_n \quad \text{if } t_n > t_{ci} \quad \forall c_i \in CP_n$$

where c and sc are the pre and post-scheduling colors, respectively. ci is any conditional operation leading to n and CP is the set of all conditional operations.

If, on the other hand, some conditional precedents are scheduled at the same clock cycle or after n , then for every conditional precedent scheduled after n , the digit corresponding to the fork due to that conditional is modified to an X .

$$\forall (c_i \in CP(n)) \quad \text{and } t_{ci} \geq t_n \quad b_{ci} = X$$

where b is the digit in the operations color that corresponds to the conditional operation c_i .

As an example, let us consider operation n_2 which is dependent on both c_1 and c_2 . Its pre-scheduling color is “11”. After scheduling as shown in Figure 5.10, the color changes to: $c_{n_2} = 1X$ if $t_{c_1} < t_{n_2} \leq t_{c_2}$ (i.e. n_2 is scheduled before or at the same time as c_2). The same mutual exclusion check is performed where X is a *don't care* and therefore matches any digit. In the above example, n_2 is mutually exclusive with all operations in the *False* branch of C_1 but none of the operations in the *True* branch.

Figure 5.10: Coloring example: post-schedule

5.5.4. Coloring Pitfall

The above coloring method falls prey to the same pitfall discussed in Section 2.8.4. It can incorrectly tag operations as mutually exclusive when they are not. Here we consider two methods to get around this problem. Note that one can always recognize the multiple branches that are valid for the same condition as parallel and no corrections are required. Nevertheless, here we present methods to get around this.

5.5.4.1. Color-coding method

In this method, the symbols used for the coloring carry more information about the exclusivity. Only the 0/1 symbols are mutually exclusive. To implement this, we need to define a *distribute* operation. This operation acts as an AND as opposed to the normal control operation that acts as a XOR. It can contain a number of conditions that result in

the same value. We then color all mutually exclusive nodes with a 1 or 0. Nodes that belong to different AND branches are not mutually exclusive and are tagged with symbols $2, 3, \dots$.

Figure 5.11 shows this coloring for the example of Section 2.8.4. the AND operation is shown inserted prior to the two conditionals C_2 and C_3 . This operation and all its successors are mutually exclusive with C_4 since one has a 0 color and the other a 1.

The mutual exclusion testing rules are:

- Compare digit b_i starting on the left ($i = 0$) of each color clr_x and clr_y corresponding to nodes A and B.
- if $(clr_x \neq clr_y)$, $(clr_x = 2)$ and $(clr_y = 2)$ then clr_x and clr_y are not mutually exclusive.
- else if $(clr_x = \phi)$ or $(clr_y = \phi)$ then clr_x and clr_y are not mutually exclusive. (ϕ implies an empty bit place).
- else if $(clr_x, clr_y) = (1, 0)$ or $(clr_x, clr_y) = (0, 1)$, then clr_x and clr_y are mutually exclusive.
- else if $(clr_x = clr_y)$, then increment d and go to compare again.

If we compare n_2 and n_5 (the same ones compared in Section 2.8.4) we find that they are identified correctly as NOT mutually exclusive.

If we have more than two mutually exclusive operations, we can add multiple XOR nodes and find a correct coloring.

This method is simple and usually stops quite early in the comparison process. It

achieves all that can be done with the Control-Tag method [64] with less overhead.

The disadvantage is the addition of the coloring nodes *AND* and *XOR*.

Figure 5.11: Color-coded method of coloring

5.5.4.2. Path-Tracing method

In this method we use the same coloring described in Section 5.5.2,

$$c_v = b_0 \cdot b_1 \cdot \dots \cdot b_{i-1} \cdot b_i$$

and maintain the list of conditionals leading to these bits.

$$C_v = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_{i-1} \rightarrow C_i$$

The mutual exclusion check is then amended by the following:

- Symbols in the color code that are different result in a mutual exclusion confirmation only if the conditionals leading to them are different.

For example in Figure 5.12, the representation is $[color, conditional_path]$. If we examine n_2 and n_6 using only the colors we will deduce erroneously that they are exclusive $11 \neq 10$. With the new rule, we note that they differ in a digit in which the conditional tags differ 2 versus 3. Therefore we conclude that they are not exclusive. Note that the colors are not restricted to 1 and 0.

This method might seem like it is carrying more overhead than the color-coded method since it carries two lists. In the actual implementation, however we require the path information to extract path-based timing information and to implement out-of-order execution, so no additional overhead is incurred by this modification. Case statements can be easily handled using this method.

Figure 5.12: Path-Trace method of coloring

5.5.5. Non Series-Parallel Graphs

One of the situations that might arise in a CDFG is the case of non series-parallel graphs. These have not been addressed by other coloring methods. Even if the CDFG does not contain such constructs, it can undergo transformations that produce them.

An example CDFG with a non series-parallel portion is shown in Figure 5.13. the coloring is also shown as pairs of colors and path designators. In our coloring method, it is simple to obtain the operation's color matrix. The check has to be performed with all the operations colors, i.e. all the rows of the matrix. Schedulers might have the ability to duplicate the operation and split the colors. Of course, one can transform this graph into to eliminate the arc from c_1 to n_3 but duplicating n_3 . Another way to eliminate these constructs is to modify the conditionals to eliminate these conditions. This may result in additional operations. In the method used here we simply perform the check against two colors rather than just one.

Figure 5.13: Non series-parallel graphs

5.5.6. Data-dependent Loops

An important aspect of control-dependent synthesis is the problem of scheduling data dependent loops. Due to the flexibility of the approach we use in optimizing CDFGs, we can represent all data-dependent graphs as conditional graphs and use the methods previously described to optimize them.

Consider the loop L_I in Figure 5.14 (a) (which is possibly made of multiple operations). If the number of iterations of the loop are known prior to execution, we can unwind the loop and time-share the execution of the loop's operations with those outside the loop. If, on the other hand, we do not know the number of iterations (loop index) prior to execution, then we do not know how much to unwind the loop.

Figure 5.14: Data-dependent loop representation

There are two conditions we need to look at. In the first, the loop index of data-dependent loop is *bounded*. For example, if the loop index can assume only one of three

values (1, 2 or 3) then we can unwind the loop as shown in Figure 5.14 (b). Note that this is conditional unwinding (operation c_l). The three branches shown are mutually exclusive. We do not know the execution path until run-time. However, this representation allows us to schedule the graph and optimize all three paths simultaneously with the operations outside the loop.

In the other condition, the loop is *unbounded*. For example, the loop index can assume any integer value. Here we perform the same type of unwinding except we will always leave one branch that is not completely unwound. The number of branches (and by extension the number of unwound executions of the loop) should be limited to a number that allows maximum time-sharing. There is an upper limit on the loop's execution above which we cannot timeshare operations any more. This separates the scheduling of the unbounded loop as no time-sharing with external operations can be performed. An example is shown in Figure 5.14 (c). Assume that two executions of L_1 in addition to n_5 and n_6 is known to be always larger than the execution time of n_1 through n_4 . This implies that the maximum number of required unwound L_1 s in any branch is two. The last branch will then contain two unwound L_1 s followed by the unbounded loop.

We have to note here that unwinding may greatly complicate the controller as well as the register binding.

5.6. Conclusion

This chapter presents the basic encoding of the GA synthesis method. We presented a GA synthesis formulation that allows simultaneous scheduling, binding and allocation. We employed relative clock cycle encoding to minimize infeasible solutions. The general encoding used satisfies most of the recommendations for effective GA encoding.

It also presented the functional and storage units used and detailed the objective function calculation. This calculation is linear for functional units and square for register

optimization. It is very flexible and can easily accommodate new constraints.

- A major portion of this chapter was dedicated to the issues of scheduling control-intensive graphs. We presented multiple possible ways for implementing this and showed a coloring implementing that is exact, easily implemented and allows code motion. We further showed how data-dependent loops can be handled using this methodology.

In terms of contributions, this chapter presented a GA formulation for the complete synthesis process, not just scheduling, that is optimized by Genetic Algorithms alone. Previous methods either considered only synthesis subsets or used extensive heuristics. This chapter also presented a novel coloring scheme that is exact and is easily merged with the GA synthesis. The method allows code motion and path-based optimizations. We also presented a simple but new approach to performing data-dependent loop synthesis.

Chapter 6: Architectural Optimizations and Heuristics

6.1. Introduction

This chapter describes some architectural optimization techniques that are supported by the proposed synthesis methodology. These techniques improve the power of the synthesis tool and enhance the resulting architectures. The chapter also presents a general heuristic technique that improves scheduling methods. All of these techniques except for the scheduling heuristic and some mobility generation heuristics were implemented in the GA synthesis framework.

Section 6.2 discusses the implementation of pipelining, chaining and multi-cycling. We note that these are very easily implemented in the GA synthesis tool.

Section 6.3 discusses the implementation of template matching and multi-bit synthesis. Again, these are simple implementations.

Section 6.4 presents a clock selection strategy and shows how to encode clock cycle information in the GA formulation to simultaneously select the clock cycle and schedule the graph. The clock set selection is exact.

Section 6.5 discusses the novel idea of multi-rate clock selection. We present the method used to encode multiple clock rates in the GA. We also prove that multiple clocks are required to optimize separate paths in a CDFG. We show how this can be easily implemented in mutually exclusive paths.

Section 6.6 investigates the mobility that is part of the GA synthesis encoding. This mobility has a large effect on the design space and the convergence of the GA optimizations. Here we discuss different techniques to generate appropriate mobilities. Some of these were not implemented but are discussed here for completeness.

Section 6.7 discusses storage optimization. We show a simple storage optimization methodology that considers storage operations. This allows simultaneous register optimization and scheduling. There are many storage optimization techniques but they are mostly run on scheduled graphs. Here we take advantage of the flexibility and simplicity of GAs to implement register operations and schedule them on storage units. Interconnect optimization can be easily performed in this manner but this was not implemented.

Section 6.8 discusses a hybridization method and presents a novel heuristic approach. The heuristic is proven to be better than existing scheduling methods under the assumption that smaller mobilities lead to better assignments. We show here how this method can be incorporated with Genetic Algorithms to perform scheduling in a novel manner.

Section 6.9 concludes the chapter and presents the main contributions of this chapter.

6.2. Pipelining, chaining and multi-cycling

6.2.1. Pipelining

The synthesis tool can handle pipelined FUs. The effect of pipelining on the chromosome decoding and the objective function calculation is limited. When determining the latest scheduled precedent, the latency is used. When calculating the area, only the use of the same stage results in incrementing the hardware usage and growing the area. The area equation presented in Section 5.4.3 should be changed as follows:

$$FUarea = \sum_{i=1}^{CC} \sum_{j=1}^{NFU_i} (FUnum_{sij} - FUmut_{sij} - FUstage_{sij}) \cdot FUarea_{sj} \cdot FUshare_{sij}$$

where the added s subscript denotes the FU stage that is being used in clock step i

and $FUstage_{sij}$ denotes all the operations executing on the same functional unit but on different stages.

Note that pipelining can be defined on specific functional units. The synthesis tool then has the capability of assigning an operation to either a pipelined or a non-pipelined unit if then are both available. This is a useful trade-off to make.

6.2.2. Multi-cycling

Multi-cycling can be simply implemented by specifying the latency of the operation. The clock cycle assignments take into account the total latency of the operation. The area calculation does not change because the same FUs are occupied in multiple cycles. The storage assignment does change but is still dependent on the storage unit assignment. Recall that register operations are only inserted on the data arcs. They are not automatically put in at every clock step.

6.2.3. Chaining

Chaining can also be implemented where successive operations can be linked in one clock cycle. Again, the method used to handle this is quite simple.

Two operations n_1 and n_2 are considered chained if all the following are true:

- (i) $(n_1, n_2) \in P$ i.e. There is a hard precedence edge between n_1 and n_2 .
- (ii) n_1 and n_2 are assigned to two different FUs.
- (iii) n_1 and n_2 are scheduled in the same clock step.

For simple chaining, an additional condition exists:

- (iv) $latency_{combined} = (d_{FU_{n_1}} + d_{FU_{n_2}} + it_{n_1} \leq clk)$
- where it_{n_1} is the initiation time of n_1 within the cycle which is usually 0.

For a combination of chaining and multi-cycling the combined latency can exceed one clock cycle but still needs to be determined.

When chaining is allowed (this can be defined per types of FUs or in general), we will remove the condition that we normally impose that $cc_{n2} > cc_{n1}$, unless the same FU is used for both (and this to avoid asynchronous feedback loops).

The description above explained chaining that is only two levels deep (only two operations are chained). The same applies for additional chaining levels.

Chaining also affects the register assignment. If two functional operations are chained, the storage operation in between them is discarded. There are additional implications on multiplexer and interconnect cost.

6.3. Template matching and varying bit-widths

6.3.1. Template matching

Some FUs can implement several operations (e.g. ALUs can perform additions and subtractions) and some operations can be implemented on multiple FUs (e.g. Additions can be implemented on ALUs and dedicated adders). A template list can be provided to the synthesis tool that allows this mapping. The chromosome encoding and decoding would be adjusted to recognize the different FU identifications. A generalization of this template matching can be used to perform transformations on the graph such as transforming serial additions into binary-tree additions or combining a multiplication followed by an addition into a multiply-accumulate operation. An example of a template list that accomplishes the ALU and MAC mappings is:

template{(+,ALU), (-,ALU), (>,ALU),(<,ALU),([*,+],MAC),([*,-],MAC)}

In the case of the MAC, this simply means that any operation of type $*$, $+$, $-$ or MAC can be implemented on this unit. Template matching does not allow you to combine a multiplication with an addition from the graph. This has to be performed ahead of time.

6.3.2. Varying bit-widths

An operation v which is $bits_v$ wide can be assigned to any compatible FU (from the template) that is equal or wider: $bits_{FU} \geq bits_v$. This allows for time-sharing of idle large FUs when needed. It also helps minimize bus area and congestion by using the appropriate width for the operations. Allowing variation in the bit-widths requires setting unused inputs and introducing muxes.

6.4. Clock period selection and synthesis

6.4.1. GA Clock selection

The GA synthesis method allows optimal clock selection during synthesis. The global clock is selected in a manner similar to that described in [9].

First, the user defines the technology-specific upper and lower bounds on the clock rate, $\underline{c}$ and $\bar{c}$. Conditions can also be put on the integrality or accuracy of the clocks (e.g. assume clocks with accuracy of 0.1ns). Then, the optimal clock rate, belongs to the set χ in the range $[\underline{c}, \bar{c}]$ that are integral divisors of all possible register transfer delays d_k (register-to-register speed of all FUs). This was demonstrated in [9] and proven in [10].

Second, the set χ is pruned for non-optimal clock rates. This was not performed in [9] or [10]. Non-optimal clock rates are those that integrally divide only the same register transfer delays and produce worse slack (larger) for all remaining delays.

if $\forall k \quad s_k(clk_1) \leq s_k(clk_2)$ then clk_2 can not be optimal

$s_k(clk_1)$ is the slack for FU k using clock 1 .

clk_2 will then be removed from the set X .

For example assume we have three FUs in Table 6.1. The optimal clocks were restricted to integer values between 5 and 50ns.

FU	*	+	-
Total Latency (ns)	48	22	23
Optimal clocks	48, 24, 12, 6	44, 22, 11	23, 12, 6

Table 6.1: Example FU latencies.

Table 6.2 shows the slacks resulting from all the clocks in the optimal clock set. Consider for example the clock periods 48 and 24. A clock period of 24 ns results consistently in slack that is less than or equal to the one resulting from the 48ns clock. Therefore clock period 48ns cannot be optimal and should be removed from the optimal clock set. Some of the clocks are also equivalent (24, 12 and 6). In this example, the clock set {48,44,24,22,12,11,6} should be reduced to {24,12,11,6}. This is a dramatic decrease in the design space.

Clock Period (ns)	* slack (ns)	+ slack (ns)	- slack (ns)
48	0	4	3
44	40	0	21
24	0	2	1
22	18	0	21
12	0	2	1
11	7	0	9
6	0	2	1

Table 6.2: Slack for clocks from optimal set.

Third, the reduced set X' is encoded in the chromosome and the GA optimizes the total latency in the circuit for the clocks selected. The best solution corresponds to the best clock selection. A chromosome encoding that includes the clock rate is shown in Figure 6.1. The number of bits required for the ck gene is $nbits_{ck} = \log_2 \|X'\|$.

The clock gene can assume any of the values in the set X . It gets decoded first and the clock cycle is determined. The remaining portions of the decoding work strictly on a clock cycle basis except during the evaluation of the chaining/multi-cycling times.

Figure 6.1: Chromosome encoding with clock selection

Note that counting clock cycles as explained in Chapter 5 can still be performed as the 9 genes are decoded first and the execution times are converted to clock cycles.

6.4.2. Multi-rate systems

For multi-rate systems (with multiple co-measurable clocks), each domain is assigned additional bits describing the clock rate. Note that, even for algorithms that are single-rate in nature, there might be advantages for having different FUs at different rates. The synthesis tool has the capability of turning single-rate system into multi-rate ones. This can be useful during power optimization when the clock frequency of a FU executing a non-critical operation is lowered. It is also useful for optimizing different paths in a CDFG.

Typical chromosome encodings for multi-rate systems are shown in Figure 6.2. Case 1 shows grouped clock domains where only the clock domains are encoded at the beginning of the chromosome. Of course, the position of the clock genes can be moved with the domains or in an arbitrary manner. Case 2 shows distributed clock domains where

each operation has its own clock-rate bits. Grouped clock domains reduce the length of the chromosomes while distributed clock domains possibly improve the schemata and the clock cycle variability.

Figure 6.2: Typical multi-rate chromosome encoding.
Case 1, grouped clock domains. Case 2, distributed clock domains.

6.5. Multi-path clocking

6.5.1. Path-dependent selection example

One of the advantages of this method of clock selection is the fact that it can accommodate multiple rates. Every operation in the graph can encode its clock rate as well as the hw and clock cycle information. This is particularly important when the circuit to be optimized can benefit from multiple rates in multiple branches or under multiple conditions.

This is true for many Control-dependent applications where different paths can

have dramatically different latencies and an optimal clock for one path is not necessarily optimal for another. Figure 6.3 shows an example where the two mutually exclusive paths p_1 and p_2 would benefit from having different clock s selected.

Figure 6.3: Example Path-dependent Clock Selection

6.5.2. Path-dependent selection proof

Definition 6.2: A path slack $\sigma_p(clk)$ is the sum of the slacks of all operations v in the critical path p for clock period clk .

$$\sigma_p(clk) = \sum_{v \in P} s_v(clk)$$

P is the set of all critical operations on path p .

Definition 6.3: An optimal path slack $\sigma_p(clk^*)$ is the smallest possible slack for a path p .

Theorem 6.3: For any two (or more) operations v_1 and v_2 , that are not integral dividers of each other and not equally distributed in all paths, there exist operation

distributions such that no single optimal clock exists for the paths s_{p1} and s_{p2} .

- *Proof.* Suppose α_{p1} and β_{p1} are the distributions of v_1 and v_2 in a specific path p_1 . Similarly α_{p2} and β_{p2} are the distributions of v_1 and v_2 in a specific path p_2 .

$$\begin{aligned}\sigma_{p_1}(clk^*_{p_1}) &= \alpha_{p_1 s_{v_1}}(clk_{p_1}) + \beta_{p_1 s_{v_2}}(clk_{p_1}) \Rightarrow \\ \sigma_{p_1}(clk^*_{p_1}) &= \alpha_{p_1 s_{v_1}}(clk^*_{p_1}) \quad \text{or} \quad \beta_{p_1 s_{v_2}}(clk^*_{p_1})\end{aligned}$$

based on the fact that the optimal clock should have zero slack on one of the operations. Similarly for path p_2 .

$$\sigma_{p_2}(clk^*_{p_2}) = \alpha_{p_2 s_{v_1}}(clk^*_{p_2}) \quad \text{or} \quad \beta_{p_2 s_{v_2}}(clk^*_{p_2})$$

if we use the clk^*_{p1} for both paths, then

$$\sigma_{p_2}(clk^*_{p_1}) = \alpha_{p_2 s_{v_1}}(clk^*_{p_1}) \quad \text{or} \quad \beta_{p_2 s_{v_2}}(clk^*_{p_1})$$

Proving the theorem implies finding the conditions where

$$\sigma_{p_2}(clk^*_{p_2}) < \sigma_{p_2}(clk^*_{p_1})$$

for

$$clk^*_{p_2} \neq clk^*_{p_1}$$

which implies

$$\alpha_{p_2 s_{v_1}}(clk^*_{p_2}) < \alpha_{p_2 s_{v_1}}(clk^*_{p_1}) \quad \text{and} \quad \alpha_{p_2 s_{v_1}}(clk^*_{p_2}) < \beta_{p_2 s_{v_2}}(clk^*_{p_1}) \quad (6.2)$$

or

$$\beta_{p_2 s_{v_2}}(clk_{p_2}^*) < \alpha_{p_2 s_{v_1}}(clk_{p_1}^*) \quad \text{and} \quad \beta_{p_2 s_{v_2}}(clk_{p_2}^*) < \beta_{p_2 s_{v_2}}(clk_{p_1}^*) \quad (6.2)$$

but in (6.2)

$$\alpha_{p_2 s_{v_1}}(clk_{p_2}^*) < \alpha_{p_2 s_{v_1}}(clk_{p_1}^*)$$

applies when

$$s_{v_2}(clk_{p_1}^*) = 0 \quad \text{and} \quad s_{v_2}(clk_{p_2}^*) = 0$$

which means either $clk_{p_1}^* = clk_{p_2}^*$ or they are integral dividers of each other both of which are irrelevant conditions reducing (6.2) to

$$\alpha_{p_2 s_{v_1}}(clk_{p_2}^*) < \beta_{p_2 s_{v_2}}(clk_{p_1}^*)$$

Similarly (6.2) becomes

$$\beta_{p_2 s_{v_2}}(clk_{p_2}^*) < \alpha_{p_2 s_{v_1}}(clk_{p_1}^*)$$

Then $clk_{p_2}^* \neq clk_{p_1}^*$ if

$$s_{v_1}(clk_{p_2}^*) < s_{v_2}(clk_{p_1}^*) \quad \text{and} \quad \alpha_{p_2} < \beta_{p_2}$$

or

$$s_{v_2}(clk_{p_2}^*) < s_{v_1}(clk_{p_1}^*) \quad \text{and} \quad \beta_{p_2} < \alpha_{p_2}$$

The first inequalities define whether clk^*_{p1} is better for $v1$ or $v2$ and similarly for clk^*_{p2} . One of those inequalities is true and the corresponding distribution inequalities define the condition where it is beneficial to use a different clock for the second path.

Corollary 1: Any path can be optimized if it has its own clock.

Note that partially overlapping paths will require transfers between clock domains which will introduce additional slack.

6.5.3. Implementation of path-dependent selection

Introducing path-dependent clock selection can result in problems with many heuristic synthesis tools due to the additional complications related to resource allocation. In the GA synthesis framework, however, this is simple.

Since the different clock domains are mutually exclusive, then they are scheduled with the selected clocks and their timing and area requirements are calculated separately. Optimizing for different control paths requires calculating the latency of each of those paths.

The total FU area is calculated separately for the separate paths. Controller area, muxing and interconnect will be shared.

Figure 6.4 shows a CDFG with two different clocks for two different control paths. Note that $ck1$ and $ck2$ are not multiples of each other.

For reconvergence ($n6$ in Figure 6.4), the latest precedence is calculated based on latency not clock period.

The situation is more difficult when the two domains are not mutually exclusive or

Figure 6.4: Path-based objective functions

when operations cross control domain boundaries (out-of-order execution). Overlaps have to be accounted for. For instance clock cycle 3 of *ck2* overlaps with clock cycles 2 and 3 of *ck1*. This is simple to account for in this methodology since we are merely interpreting the GA strings which are scheduled graphs. We are not making the decisions on whether to assign operations to these clock cycles or not. The assignments are guided only by the objective function calculation.

This method gets more complicated with higher numbers of paths especially when there is much overlap between the different paths.

6.6. Mobility calculation

The type of mobility specification has an impact on the resulting schedules through the reduction of the design space. The tighter the mobilities are, the smaller the design space, the faster the solutions and the shorter the convergence time. Following are several methods that can be used for the mobility.

6.6.1. ASAP to ALAP

If the mobility is defined by the interval between the ASAP schedule and the ALAP schedule, the solution that minimizes the number of clock cycles (i.e. the most parallel) is relatively easy to achieve. In many cases an all-zero chromosome could be one of the solutions. The problem with this mobility choice is that it can restrict hardware-constrained systems. So for time-constrained scheduling, with a constraint equal to $t_{ALAP} + \tau$, we use the interval $[t_{ASAP_v} \rightarrow t_{ALAP_v} + \tau]$ and use a bounded relative encoding method as described in Section 5.2.4.2.

For resource-constrained scheduling or systems with soft area constraints, the interval between ASAP and ALAP (or $ASAP + \tau$) is not large enough to allow for the operations to be scheduled with a minimum or close to minimum hardware configuration. For example, Figure 6.5 shows the mobilities for 4 additions which are such that 4 additions need to be performed in 3 cycles. If only one adder is available (or the adder area is a liability), the ASAP to ALAP encoding will not allow the use of only one adder and thus the scheduling result is sub-optimal. To be totally inclusive of the whole architectural evaluation space, the mobility of the combination of all these operations should be such that it allows for the case of minimum hardware

	n1	n2	n3	n4				
					OP	ASAP	ALAP	ARAP
1				+	n1	1	3	4
2	+				n2	1	2	3
3			+		n3	2	3	4
4					n4	1	1	2

Figure 6.5: Mobility example for ASAP, ALAP and ARAP.

6.6.2. Fixed or fixed increment

-

The first method is to estimate the value of t in a heuristic manner or establish an upper bound. There are many techniques to provide an upper bound for the algorithm's latency. Another method of decreasing the probability of restrictive mobilities is to have fixed mobilities for the operation (regardless of the ASAP or ALAP schedules) or add fixed increments to the ASAP-ALAP mobility encoding. The fixed values could be the same across operations or some heuristics could be used to assign different values to different operations.

6.6.3. Unbounded Relative clock cycle decoding

Another method is the unbounded relative clock decoding as discussed in Section 5.2.4.2. This allows the GA to find solutions beyond the upper limit on the clock cycles set by the ALAP value. This method works very well for graphs that are mobility-restricted towards the end of the algorithm. It works for graphs that are mobility-restricted early on but not as well.

6.6.4. ASAP to ARAP

Another method that is used in this thesis is a mobility defined by the ASAP to ARAP (As Resource-constrained as possible) interval. ARAP is defined as a minimum hardware schedule. A time-optimal minimum hardware schedule is difficult to obtain (It is actually NP-complete for any FU number greater than 1) [51].

ARAP, as used in this methodology, is just a minimum hardware schedule that allows the flexibility that the ASAP to ALAP interval does not allow. It does not have to be latency optimal but near-optimality is a benefit. The important aspect, however, is to have an ASAP-ARAP interval that will contain all delay-area optimal solutions. This reduces the design space dramatically and enhances the performance of the GA synthesis system.

The essence of the ARAP schedule is that each $t_{ARAP_v} = t_{ALAP_v} + \tau_v$ uses a unique t_v rather than the global t obtained by the usual upper limits. This can dramatically reduce the design space.

Figure 6.5 shows an example ARAP that is obtained by incrementing all restrictive operations enough to use one adder only. Other ARAP schedules are possible. The interval ASAP-ARAP will then guarantee all possible useful solutions to the problem. Different ARAP schedules produce more or less optimal solution and have different generation complexity.

Finding an ARAP schedule is difficult but a simple approach can run heuristic algorithms to find an upper-bound for a number κ of CDFGs ranging from $\kappa=1$ to V where V is the total number of operations in the graph. While it may seem that the order of this estimate is increased from $o(V^Q)$ to $o(V^{Q+1})$, we have to keep in mind that the upper bound heuristic is dependent on the size of the graph and we are successively reducing the size. Of course, a better method would be to modify or create a new heuristic that takes the different operations into account. This method was not implemented.

6.7. Storage and Interconnect Optimization

The storage optimization method used in this thesis performs all storage optimizations simultaneously with the scheduling, allocation and binding of functional operations. It allows the use of multi-port register files and RAMs, on and off-chip. The system can also handle storage of data with varying bit-widths. Multiple clock frequencies can be used. The area, speed and power consumption of storage units as well as multiplexing circuits are taken into account

6.7.1. CDFG modification

The first step in storage optimization is modifying the input CDFG to include storage operations. A register operation is inserted following each functional operation in the graph. The resulting graph is input to the synthesis tool. Figure 6.6 shows how an example portion of a CDFG was modified by inserting storage operations (r1-r10). These operations are included in the chromosome and are optimized, scheduled and assigned to SUs. The differences between storage operations and functional operations include:

- Latency: functional operations have a pre-determined latency equal to the latency of the library functional units on which they are bound. Storage operations have the same type of access latency in addition to a *lifetime* which is dependent on the operations' successors. This implies that for each storage operation, all connected operations have to be scheduled before the storage cost can be determined.
- Precedents: Storage operations can have only one precedent each, which is the operation that generated the value to be stored.
- Simultaneous execution: the simultaneous execution of two storage operations on the same SU is allowed when both operations have the same precedent. This implies that only one register is needed to store a value that is destined to one or more FUs.
- Multiple accesses: SUs are allowed to have multiple access ports and their size depends on the number of storage elements and ports required.

Example SUs are shown in Figure 6.7. Examples of different units with different amount of storage and different numbers of input and output ports are shown. An example assignment of the example in Figure 6.6 that uses a register file is shown in Figure 6.8.

Note that the assignments are taken as per the chromosome values, meaning that the assignment of a register to a SU (R or rf) is performed and evaluated simultaneously

Figure 6.6: Example CDFG modification: storage operation insertion. The added operation (one storage operation per edge) are assigned to SUs.

Figure 6.7: Different types of SUs. Number of Ports, amount of storage and latency are shown.

with the scheduling and allocation and not as a post synthesis optimization. This allows the flexibility of using any of the SUs specified above.

Figure 6.8: Example register assignment and resulting topology.

6.7.2. Interconnect and Routing Optimization

The flexibility of the GA synthesis system presented in this thesis is such that many different optimization criteria can be used. One such criterion is interconnect. Interconnect minimization has traditionally been performed at a post-synthesis stage minimizing buses by time-sharing variables. In this methodology, bus area can be optimized simultaneously with the F/SU and multiplexer area. In most circuits, integrated optimization is necessary because minimizing the number of buses increases the number of multiplexers.

Another important issue that is partially solved by minimizing the number of buses is routability. This is particularly important in standard cell and gate array implementations. To help produce a design that is more routable (resulting in a smaller die array and easier layout flows), two parameters can be used: porosity and regularity. Porosity is a function of the design of a specific functional unit. The more porous a unit, the easier it is to pass metal tracks across it. The use of more porous units at the expense of

area can result in a more routable design. The library units can then have a porosity variable that is used in the objective function calculation. Regularity is a very important attribute especially in datapath designs. This is more difficult to ascertain but heuristics can be employed such as weights based on the variation in the source and destination of buses or the number of inputs of multiplexers.

Note that different types of SUs can be used simply by the GA picking them. In addition to FU criteria, the SU assignments consider the number of storage elements within a SU as well as the number of ports. Choosing memory locations is done arbitrarily.

In the GA synthesis tool, register optimization was implemented but multiplexer and interconnect optimization was not.

6.8. Hybridization of GA synthesis

We have shown how to cast the synthesis problem as a genetic algorithm. While this is ideal in terms of flexibility, one may still require faster convergence or a larger probability of obtaining good solutions. For this, one can revert back to general evolutionary concepts and combine them with the best available heuristic for the problem. What evolutionary techniques will do is provide a method for the application-specific heuristics to get out of local minima.

There are many ways to combine heuristics with evolutionary techniques. What we propose in here is a novel mobility reduction technique that is provably superior to most traditional techniques. It can further be easily integrated with evolutionary methods.

6.8.1. The design-space reduction method

Our experimentation with different encodings and different examples indicates that the most effect that any variable can have on the quality of the solutions is that of the

modification of the mobilities. In the following we show that the design-space reduction method is always superior (or in the worst case equal) to any other method for assigning an operation to a time-step. We note here that a limited form of mobility reduction was used to enhance the performance of Force-directed scheduling [76].

6.8.2. Proof of superiority of the reduction method

Mobility reduction is the process of iteratively removing the worst possible assignments from the operation's mobility. Here, we prove that the mobility reduction method leads to better solutions than the traditional heuristic methods. The heuristics in question are ones that are

(i) dependent on the size of the mobility i.e. the smaller the mobility the better the assignment.

(ii) Have the ability to order the different time steps of an operation's mobility in order of confidence in their optimality.

For these heuristics we show that iteratively removing the perceived worst possible assignment is better than immediately selecting the perceived best assignment.

Definition 6.4: The probability of selecting an optimum position to assign an operation to, in an iterative scheduling method is

$$P(v \rightarrow c_p) = C_\mu \frac{N}{\mu} \quad (6.3)$$

where N is the number of optimal positions in the mobility range μ .

C_μ is the confidence of making the correct assignment. If $C_\mu < 1$, then the

selection is probabilistic. If we use a heuristic, then C_μ is larger than 1. C_μ is by definition dependent on the mobility in such a way that $C_{\mu-1} \geq C_\mu$. This implies that the confidence in the heuristic selection increases when μ decreases. This seems to be the case for many heuristics including Force-directed scheduling [76].

Theorem 6.4: In a scheduling problem, iteratively reducing the operations' mobility by removing the worst possible assignment for each operation until one position remains results in either a superior or an equal schedule as compared with selecting the best possible position for an operation.

Proof:

First we prove by induction that reducing the mobility range one step at a time for N optimal time slots results in a probability of selecting an optimum position of

$$P_N(v \rightarrow c_p) = \left(\frac{n+1}{\mu}\right) \cdot \prod_{i=2}^{\mu} C_i.$$

Then we show that this is better than the original probability

$$P(v \rightarrow c_p) = C_\mu \frac{N}{\mu}$$

For $N = 1$, each time a selection is made there is a probability $\frac{\mu-1}{\mu}$ of making the correct selection (i.e. the removed clock step is not the optimal one). The mobility is successively reduced $\mu' = \mu - 1$. Since we are using a heuristic, each one of those selection probabilities should be multiplied by the confidence factor C_μ . This results in:

$$P_1(v \rightarrow c_p) = C_\mu \left(\frac{\mu-1}{\mu}\right) \cdot C_{\mu-1} \left(\frac{\mu-2}{\mu-1}\right) \dots C_2 \left(\frac{1}{2}\right) = \frac{1}{\mu} \cdot \prod_{i=2}^{\mu} C_i$$

$$\frac{1}{\mu} \cdot \prod_{i=2}^{\mu} C_i \geq C_{\mu} \cdot \frac{1}{\mu} \quad (6.4)$$

which implies that for $N=I$, the theorem holds.

Now, assume

$$P_n(v \rightarrow c_p) = \frac{n}{\mu} \cdot \prod_{i=2}^{\mu} C_i \quad \forall \mu$$

We need to prove that when $N = n + 1$,

$$P_N(v \rightarrow c_p) = \frac{n+1}{\mu} \cdot \prod_{i=2}^{\mu} C_i$$

The difference now is that one can make bad choices and still end up with an optimal solutions since there are more than one. So the probability of assigning the operation to an optimum clock step is then:

$$\begin{aligned} P_N(v \rightarrow c_p) = & P_{\mu}(good) \cdot (P_{\mu-1}(good) \cdot (P_{\mu-2}(good) \cdot (\dots) + P_{\mu-2}(bad) \cdot (\dots)) + P_{\mu-1}(bad) \cdot (\dots)) \\ & + P_{\mu}(bad) \cdot (P_{\mu-1}(good) \cdot (P_{\mu-2}(good) \cdot (\dots) + P_{\mu-2}(bad) \cdot (\dots)) + P_{\mu-1}(bad) \cdot (\dots)) \end{aligned}$$

where $P(good)$ corresponds to making a correct selection and $P(bad)$ corresponds to making an incorrect selection. So, using the assumption

$$P_n(v \rightarrow c_p) = \frac{n}{\mu} \cdot \prod_{i=2}^{\mu} C_i \quad \forall \mu$$

then

$$\begin{aligned}
P_N(v \rightarrow c_p) = & C_\mu \left(\frac{n+1}{\mu} \right) \cdot \left(\frac{n}{\mu-1} \right) \cdot \prod_{i=2}^{\mu-1} C_i + C_\mu \left(\frac{\mu-(n+1)}{\mu} \right) \left(C_{\mu-1} \left(\frac{n+1}{\mu-1} \right) \cdot \left(\frac{n}{\mu-2} \right) \cdot \prod_{i=2}^{\mu-2} C_i \right. \\
& + C_{\mu-1} \left(\frac{\mu-(n+1)-1}{\mu-1} \right) \left(C_{\mu-2} \left(\frac{n+1}{\mu-2} \right) \cdot \left(\frac{n}{\mu-3} \right) \cdot \prod_{i=2}^{\mu-3} C_i \right. \\
& + C_{\mu-2} \left(\frac{\mu-(n+1)-2}{\mu-2} \right) \dots \left(C_{n+2} \left(\frac{n+1}{n+2} \right) \cdot \left(\frac{n}{n+1} \right) \cdot \prod_{i=2}^{n+1} C_i + C_{n+2} \left(\frac{1}{n+2} \right) \left(C_{n+1} \cdot \prod_{i=2}^n C_i \right) \right) \left. \right) \left. \right) \left. \right) \left. \right) \quad (6.5)
\end{aligned}$$

which reduces to

$$P(v \rightarrow c_p) = \frac{n+1}{\mu} \cdot \prod_{i=2}^{\mu} C_i$$

The reduction proceeds as follows. Consider the last term of Equation 6.5,

$$\begin{aligned}
& C_{n+2} \left(\frac{n+1}{n+2} \right) \cdot \left(\frac{n}{n+1} \right) \cdot \prod_{i=2}^{n+1} C_i + C_{n+2} \left(\frac{1}{n+2} \right) \left(C_{n+1} \cdot \prod_{i=2}^n C_i \right) \\
& = \left(\frac{n+1}{n+2} \right) \cdot \left(\frac{n}{n+1} \right) \cdot \prod_{i=2}^{n+2} C_i + \left(\frac{1}{n+2} \right) \left(\prod_{i=2}^{n+2} C_i \right) \\
& = \left(\frac{n(n+1)}{(n+1)(n+2)} \right) \cdot \prod_{i=2}^{n+2} C_i + \left(\frac{1}{n+2} \right) \left(\prod_{i=2}^{n+2} C_i \right) \\
& = \frac{n+1}{(n+2)} \prod_{i=2}^{n+2} C_i
\end{aligned}$$

Proceed in the same manner until $\frac{n+1}{\mu}$ remains.

So, we have proven that the probability of selecting an optimum in the mobility reduction method is

$$P(v \rightarrow c_p) = \frac{N}{\mu} \cdot \prod_{i=2}^{\mu} C_i$$

Note that the probability of picking one of the N optimum cycles in one iteration is

$$\frac{N}{\mu} C_{\mu}$$

But

$$\frac{N}{\mu} C_{\mu} \leq \frac{N}{\mu} \cdot \prod_{i=2}^{\mu} C_i$$

because from Equation 6.4

$$\frac{1}{\mu} \cdot \prod_{i=2}^{\mu} C_i \geq C_{\mu} \cdot \frac{1}{\mu}$$

This is an important result since it indicates that any heuristic that places operations with their mobility range and can be applied in the reverse manner (i.e. find the worst not best location), can benefit from this technique as long as the heuristic's quality improves with reduced mobility.

As a matter of fact a partial implementation of this technique was employed in [76] to improve the global behavior of Force-Directed Scheduling. However, rather than selecting and removing the worst possible position from the mobility at a time, the removal in that technique only reduced the mobility from the top and bottom.

6.8.3. Force-directed scheduling using the reduction method

Modifying Force-Directed Scheduling to use a reduction technique was done in [76]. The reductions were limited, however, to occur from the top or bottom of the mobilities. This was referred to as gradual time-frame reduction and suspected to make the algorithm less greedy.

If we use the unrestricted reduction method, any part of the mobility can be reduced at any time. If the removed part is not the top or bottom of the mobility, only the distribution graph is affected. The successor operations are not affected. This implies that the additional complexity of implementing an unrestricted mobility reduction method is relatively small. The advantage of this method is that it is less restrictive.

Figure 6.9: Example mobility based encoding

6.8.4. Suggested evolutionary approach using the reduction method

Now that we have established the fact that the reduction approach is better than the straight assignment approach we show this problem can be effectively combined with evolutionary techniques.

Rather than encoding the *CC* in the GA as we have done in this thesis, we can encode the mobility range. Each bit corresponds to assigning an operation to a clock step within its mobility. A “1” indicates that this assignment is allowed, a “0” indicates it is not. Such an encoding is shown in Figure 6.9. If we use forces as in Force-directed scheduling or some function of the distribution graph as an objective function, a GA can modify the mobilities until we achieve the resource constraints. This effectively does what we proposed for force-directed scheduling but can stochastically eliminate multiple

assignments in each iteration rather than one at a time. It is possible that this technique can achieve good results at speeds comparable or better than Force-directed scheduling.

6.9. Conclusion

This chapter discussed the implementation of many architectural synthesis improvement techniques within the GA framework. These include pipelining, chaining, multi-cycling, template matching and multi-bit synthesis. These are shown to be very simple to implement in this methodology. We also present a clock selection strategy from an optimum clock set. Again, the use of GAs makes this process simple enough to accommodate multi-rate clock selection which has not been implemented in other synthesis systems. We prove here that multiple clocks are required in order to optimize separate paths in a CDFG.

This chapter also investigated the mobility that is part of the GA synthesis encoding. We showed multiple methods of determining this mobility to help reduce the design space and effectively the results of the synthesis.

This chapter also presented the storage optimization process performed in conjunction with scheduling. We showed a simple storage optimization methodology that considers storage operations. The method is flexible allowing the use of registers, register files and RAMs with single or multiple ports. It also allows the incorporation of the overhead associated with interconnect and multiplexers.

A novel heuristic approach to scheduling was also presented. This *reduction* approach was proven here to result in better schedules than classic heuristic methods. We showed how this technique can be applied to force-directed scheduling. We also propose a new GA hybrid method that uses this technique for scheduling. This method would have the advantage of a good scheduling heuristic combined with many of the advantages presented in this thesis for using GA techniques for synthesis.

Chapter 7: Synthesis Benchmarks and Results

7.1. Introduction

This chapter presents the results of the GA synthesis tool on several benchmarks and examples. The results were all selected to show certain aspects of the synthesis technique.

Section 7.2 shows the quality and speed of the GA synthesis. Results are compared with other systems for the traditional EWF benchmark. We show that the GA synthesis finds optimal solutions of this problem faster than the ILP used in OASIC [25]. We also show that better solutions can be found than the PSGA method [18] which combines a problem-space GA with scheduling heuristics.

Section 7.3 shows the quality of conditional synthesis (with CDFGs). Results on the example from [40] and the Rotor [62] are shown. The GA synthesis again finds all optimal solutions. We note here that the exact symbolic method in [62] seems to find the solutions faster. The advantage of the GA here is that it might find multiple solutions in a single run. Here we show optimizations for the worst path as well as the average paths.

Section 7.4 shows results for larger circuits like the S2R and a suite of DCT circuits up to a 2-D 8point DCT implementation. We find that for larger circuits, the GA method outperforms the symbolic method (in speed), as expected. The objective function calculation is linear unless register optimization is performed where it becomes $O(n^2)$. The DCT suite of architectures are included to show how the GA synthesis performs against two hand-crafted DCT architectures: a pipelined implementation from [41] and a time-shared implementation introduced here. The GA synthesis achieves good engineering results for this graph that has more than 1200 functional and storage operations.

Section 7.5 shows results for clock selection and synthesis. We show that we can

find the optimal results for scheduling and clock selection for the EWF benchmark in a short amount of CPU time. Here we compare with [10] where multiple ILP runs are required to find these solutions. We also show runs of multi-path multi-rate clock selection and synthesis that result in better solutions than can be achieved with single-clock synthesis.

Section 7.6 shows samples of library units that are used. The ability of performing allocation simultaneously with scheduling in this method allows for easy use of multiple units of the same type.

Section 7.7 concludes the chapter and summarizes the main results of the runs performed here.

7.2. Quality and Speed of Synthesis

7.2.1. The Elliptic Wave Filter

This section presents the EWF benchmark and compares the GA synthesis results with the literature. This benchmark is used for ease of comparison with the existing literature.

7.2.1.1. Benchmark Description

The Elliptic Wave Filter is the classic benchmark for high-level synthesis tools [20][44][53]. It is a low-pass IIR (Infinite Impulse Response) filter containing 34 operations as shown in Figure 7.1.

7.2.1.2. EWF results

The EWF circuit was synthesized using the system presented in this thesis and

Figure 7.1: Elliptic Wave Filter Data Flow Graph

produced the results shown in Table 7.1. In order to compare with published results, relative areas of 2 to 1 were given for the adder and multiplier units. The multiplier was assumed to be a 2-cycle multiplier (pipelined and non-pipelined) and the adder a 1-cycle adder.

The table shows results for different numbers of clock steps under different resource constraints. The CPU time is the number of seconds required to obtain the

	GA Synth ^a					
Clock steps	17	17	18	18	19	21
# Adders	3	3	2	3	2	2
# Multipliers	3		2			1
# Pipelined Mult.		2		1	1	
# Registers	10	10	10	10	9	9
CPU time (s)	3.6	0.2	11.0	27.35	31.4	66.8

Table 7.1: EWF benchmark results from GA Synthesis

a. CPU times are for SUN Ultra 1

solution given, which corresponds to the number of generations run until the solution is found. Note that in some cases, a single run of the algorithm produced more than one of these solutions. The different runs varied the mobility value and the objective function.

To evaluate the quality of these results, we compare with the OASIC [25] results shown in Table 7.2. These results are optimal since they use ILP. Note that the GA synthesis was able to obtain all these optimal results (This is not always guaranteed). Also, as expected the GA synthesis results were obtained more quickly. We expect this trend to be even more important as the design size increases.

	OASIC ^a [25]					
Clock steps	17	17	18	18	19	21
# Adders	3	3	2	3	2	2
# Multipliers	3		2			1
# Pipelined Mult.		2		1	1	
# Registers	10	10	10	10	9	9
CPU time (s)	30	30	180	180	348	1800

Table 7.2: EWF benchmark results from OASIC

a. The IP model was solved using GAMS/ZOOM on an IBM PS/2 model 80.

Now, we compare with the results from PSGA which is a hybrid system using GAs to provide ordering and heuristics to perform the synthesis based on this ordering.

While the speed of this system is comparable to the GA synthesis, it is not able to obtain some of the solutions. Note the resulting 10 registers for 21 cycles with 2 adders and 1 multiplier. The GA synthesis, we presume, does not get stuck in local minima as much as heuristic methods. However, the results are probably more anecdotal than statistically significant. Also, it takes much longer to achieve this result with the GA synthesis.

	PSGA [18]					
Clock steps	17	17	18	18	19	21
# Adders	3	3	3	3	2	2
# Multipliers	3		2			1
# Pipelined Mult.		2		1	1	
# Registers	10	10	10	10	9	10
CPU time (s)	10.0	10.0	10.2	10.3	10.3	10.3

Table 7.3: EWF benchmark results from PSGA^a

a. CPU times are for a Sun SPARCStation 2

Figure 7.2 shows the convergence of the EWF for a typical run. Shown are the best, worst and average solutions for each generation of chromosomes. It is clear that the algorithm improves the fitness of all individuals in successive generations.

The EWF with storage and I/O units is shown in Figure 7.3. Note that the black rectangles represent constants and the triangles represent circular registers and arcs.

7.2.1.3. Design space exploration

For the results presented on the EWF, the number of bits used in the chromosomes

Figure 7.2: Population fitness of a typical EWF run. (Steady State)

varied from 13^1 to 102 resulting in a design space varying from $2^{13}=8192$ to $2^{102}=5e^{30}$. Assuming the number of solutions is of the same order as the results described in Section 2.2.3, say varying from 70 to a few millions (The setup given in Section 2.2.3 is not identical), then the probability of finding an optimum solution is on the order of 0.008 to $2e^{-24}$, which is very small. These would be the probabilities of a random search finding the optimum solution. The GA finds the solutions much more efficiently.

This analysis considered only scheduling. With allocation and binding the design space gets even larger. Consider 2 types of adders and 2 types of multipliers, this alone increases the allocation/scheduling space by $34^2=1156$ times.

7.3. Quality of Conditional Synthesis

In order to demonstrate the results for CDFG synthesis (with branching), we use the example from Kim [40] and the Rotor presented in [62].

1. This is an encoding that performs scheduling only on 1 type of adders, 1 type of multipliers and mobilities between ASAP and ALAP.

Figure 7.3: Elliptic Wave Filter Data Flow Graph

7.3.1. Kim's CDFG example

Kim's example has two conditional blocks, with the depth of nesting being 2. It has 16 additions, 8 subtractions and two comparisons. The CDFG is shown in Figure 7.4.

			Heuristic [40] ^a	Symbolic [62] ^b	GA Synth ^c	
-	# of cycles	long	7	6	6	6
		avg	6.25	5.75	5.25	5.75
	Adders			2	2	2
	Subtractors			1	2	1
	Comparators			1	1	1
	CPU time (s)		-	4.1	6.2	11.9

Table 7.4: Kim's CDFG

a. C code on SunSPARCstation

b. C++ code on SunSPARCstation10

c. code on Sun Ultra 1

7.3.2. The Rotor example

The ROTOR, introduced in [62], performs a rotation of coordinate axes by an input angle θ . This transformation is used in many robotics and graphics. The algorithm is shown in Figure 7.5. We note that the trigonometric functions ($\sin(\theta)$ and $\cos(\theta)$) are pre-computed and stored in table lookups (read from using $T(x)$ in the algorithm). This algorithm is used in order to store only one quadrant of trigonometric values to keep the size of the lookup table small. The ROTOR has 3 branches with a maximum branch-chain depth of 2. The CDFG is shown in Figure 7.6.

In order to compare results, we use 1 single-port look-up table only (hard constraint) and assume that a *read* takes one cycle. $-T(\text{angle})$ is a *read* followed by a *negation*. Subtractions and comparisons can be performed simultaneously. Using pipelined control. e.g. if $(a=180-\text{Theta})$ is in step s , a is available at the beginning of $s+1$, but control flow is affected at the beginning of step $(s+2)$. ALUs can perform additions or subtractions in one cycle. Multiplications are performed by 2 two-cycle pipelined multipliers.

```

a=180-θ;
if (a>=0){
  b=90-θ
  if (b>=0){
    sinθ=TRIG(θ);
    cosθ=TRIG(θ);
  }
  else{
    sinθ=TRIG(a);
    cosθ=-TRIG(-b);
  }
}
else {
  c = 270-θ;
  if (c>=0) {
    sinθ=-TRIG(-a);
    cosθ=-TRIG(c);
  }
  else {
    sinθ=-TRIG(360-θ);
    cosθ=TRIG(-c);
  }
}
X=x*cosθ+y*sinθ;
Y=-x*sinθ+y*cosθ;

```

Figure 7.5: ROTOR algorithm

Table 7.5 shows the results including the longest latency interval as well as the average latency for different control paths. The CPU times listed are comparable to the ones obtained in [62].

		Symbolic [62]		GA Synthesis	
# of cycles	long	7	6	7	6
	avg	-	-	6.5	6
ALUs		2	4	2	4
Table Look.		1	1	1	1
CPU time (s)		12.3	1.3	17.3	2.1

Table 7.5: Radicojevic's Rotor examples

Figure 7.6: ROTOR CDFG
T represents the TRIG table lookup.

7.4. Large Circuit Synthesis

In this section we examine larger circuits than the ones synthesized in Section 7.2 and Section 7.3.

7.4.1. S2R

S2R was presented in [62]. While it is not particularly large, it is twice the size of the ROTOR example studied earlier. The S2R example translates spherical coordinates $[R, \theta, \phi]$ into cartesian (rectangular) coordinate values $[X, Y, Z]$. There are 42 operations and as many as 64 execution paths. The hierarchical CDFG is shown in Figure 7.7.

The results of the GA synthesis are compared with the symbolic approach [62], in Table 7.6. Note that the 10 clock step solution corresponds to one that does not allow out-of-order execution. We can see that the GA synthesis achieved the best solution (it is actually optimal) in a faster time than the symbolic approach.

	Symbolic [62]		GA Synth
Clock Steps	8	10	8
ALUs	3	3	3
Table Lookups	1	1	1
Multipliers	2	2	2
CPU time	108.8	177.9	74.7

Table 7.6: Radicojevic's S2R Comparison

7.4.2. Discrete Cosine Transform

7.4.2.1. DCT Description

The Discrete Cosine Transform is a Time to Frequency domain transform similar to the DFT. The DCT, first proposed in [2], has several properties that make it attractive for image compression.

When applied to an image block of size $L \times L$, the two-dimensional DCT is:

Figure 7.7: S2R CDFG

$$Y_{k,l} = \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} x_{i,j} C_{k,l} \cos \frac{(2i+1)k\pi}{2L} \cos \frac{(2j+1)k\pi}{2L}$$

for $k = 0, 1, \dots, L-1$; $l = 0, 1, \dots, L-1$; where

$$C_{k,l} = \begin{cases} \frac{1}{2}; & k=l=0 \\ \frac{1}{\sqrt{2}}; & k=0 \vee l=0 \\ 1; & \text{otherwise} \end{cases}$$

$x_{i,j}$ being the value of the pixel in the initial block; $Y_{k,l}$ the value of the coefficient in the transformed block.

It is possible to calculate a 2-D DCT by performing a 1-D DCT on the rows and then a 1-D DCT on the resulting columns which reduces the complexity (This is known as the separability property and requires a transpose function). In JPEG applications, the input image is divided into blocks of 8x8 pixels and input as signed integers to a 2-D DCT system. The DCT will then transform the block into a block of spatial frequencies.

DCTs can be implemented in a variety of methods, including direct implementations, Fast Cosine Transform (FCT) implementations similar to FFTs and using distributed arithmetic. Most implementations use the separability property. The implementations in this thesis will implement the DCT Signal Flow Graph and use the separability property.

7.4.2.2. Modified Arai's algorithm

Arai [4] developed a FCT algorithm that requires 5 multiplications and 29 additions/subtractions for an 8 point block. This algorithm was later modified by Kovac [41] to exchange additions and subtractions and reduce the number of subtractions from 16 to 12. This modified algorithm still requires 5 multiplications and 29 additions/

subtractions. The Signal Flow Graph is shown in Figure 7.8.

Figure 7.8: Modified Fast DCT showing operations

7.4.2.3. Kovac's Implementation

Kovac implemented this algorithm in the same form shown in the DFG of Figure 7.8 as a pipelined system. A total of 6 partitions were used, each containing a set of registers and an arithmetic unit. This is shown in Figure 7.9. Note how each of the layers labelled in Figure 7.8 is implemented on one hardware unit. In total five 1-cycle adders/subtractors are used and one 6-stage, 6-cycle Wallace Tree multiplier. The DCT is computed in 59 cycles after filling the first set of registers (8 cycles). There are 100 registers used. The throughput of this system is 1 pixel/cycle. The transpose memory is implemented as 64 registers with a latency of 64 clock cycles. In total, the latency of the 2-D DCT then is 326 cycles. The total HW used is: 2 multipliers, 10 adders and 264 registers. The control overhead is clearly minimal. This architecture, then allows limited timesharing of the functional units (i.e. timesharing is allowed only for each level separately).

Figure 7.9: Arai's 1-D DCT implementation

Each layer of the DFG in Figure 7.8 is implemented on 1 hardware unit. The data is input serially to the 1st set of registers (white) and then transferred in parallel to the second set (grey).

7.4.2.4. Time-Shared Implementation

If timesharing between operations is allowed, then the latency of Kovac's implementation can be dramatically reduced. The hardware requirements might also be decreased. Using the same type of units as in the previous implementation, one such architecture can be developed by time-sharing operations as shown in Figure 7.10.

The theoretical latency that can be achieved using four 1-cycle adders and one 6-cycle/6-stage pipelined multipliers is 19 cycles due to the operation precedences. This assumes 1 input port without pixel pre-ordering. (i.e. $a[0-7]$ are input serially). Re-ordering the inputs can improve latency. The first set of operations is then started at cycle 5. In each cycle, at most 4 adders/subtractors and 1 multiplier are used.

Figure 7.10: 1-D DCT Time-shared architecture

The chosen architecture requires 4 adders/subtractors (These can be further divided into 2 adders, 1 subtractor and 1 adder/subtractor) 1 pipelined six-stage multiplier and 9 registers. The latency of the system is then 20 cycles including the initial serial input of the operands. This architecture was chosen to reduce the hardware requirements while maintaining a latency close to the theoretical limit and allowing for future timesharing. The control unit is not shown but is relatively simple and can be easily implemented. Note that in the implementation of such an architecture, it may be useful to duplicate some registers in order to reduce the number and/or size of the multiplexers. Various improvements can also be made by grouping the registers into register files interfacing to the functional units through buses or allowing register-to-register transfers to minimize buses and multiplexers. It can be seen, that this architecture will require appropriate buffering on the buses shown. A trade-off between the number of registers and the interconnect as well as the buffering occurs, and some registers could be added to alleviate this problem.

Figure 7.11: Resource utilization table

7.4.2.5. Architectural enhancements

The latency improvement of the time-shared architecture over Kovac's implementation is 70%. The reduction in ALUs is 20%. The reduction of registers is 91%. However, the throughput of the system as implemented is worse than Kovac's fully pipelined architecture. The architecture can operate on 8 pixels every 15 cycles (with an initial 5 cycle overhead) resulting in a throughput of 0.53 pixels per cycle. This is a reduction of 47% in throughput.

To improve the throughput of the system using the same functional units without

Figure 7.12: Table for 2 consecutive initiations of 8 points

input re-ordering, we try to maximize the utilization of the FUs when operating on multiple 8-point DCTs simultaneously.

Figure 7.11 shows the resource utilization of the 4 ALUs and the multiplier for all 20 cycles. For the multiplier, only the initiations of the operations are shown (the latency is 6 cycles). First, we re-assign the operations in cycles 7, 11 and 13. This enables us to interleave the operations of two successive 8-point DCTs executing them in 27 cycles. Further the two operations from the new 2x8-point DCT in cycle 6 can be moved to cycle 7 without incurring any loss in latency or in hardware. Due to the restriction on the number of input ports (we will assume only one is available), and the restriction on input re-ordering, additional cycles are needed to input all the coefficients. This results in the utilization graph in Figure 7.12. The latency of each 2x8-point DCT is 27 resulting in a total throughput of 0.84 pixels/cycle.

By combining two 8-point transform, a 4x8 DCT can be executed in 46 cycles as shown in Figure 7.13. The throughput is again 0.84 pixels/cycle.

Figure 7.13: Resource utilization for 4x8 points

7.4.2.6. 2-D DCT computation

A 1-D 8-point DCT can be performed in 85 cycles at a throughput of 0.83 pixels/cycle. It should take less than 170 cycles to complete a 2-D 8-point DCT.

Table 7.7 shows the completion times of all 8 coefficients of an 8-point 1-D DCT. The arrival times of the different coefficients can be exploited in the 2-D calculation. The 2-D DCT including the transpose is then completed in 159 cycles with a throughput of 0.85 pixels/cycle (with cascaded 1-D DCTs).

To support a color image of 1024x1024 pixels at 30 frames/sec, the 2-D DCT for the whole image should be performed at a rate:

$$R = 1024^2 \text{ pixels} \times 30 \text{ frames/sec} \times 3 \text{ frames} = 94,371,840 \text{ pixels/sec}$$

With a throughput of 1 pixel/sec, Kovac's system was designed to operate at 100 MHz clearly surpassing this limit. For the time-shared system to obtain the same performance, it has to operate at a frequency of 111 MHz.

This corresponds to a critical path of 9 ns. This and the increased design complexity are essentially the trade-offs for the dramatic improvement in area and latency.

	0	1	2	3	4	5	6	7
0	15	20	34	39	53	58	72	78
1	24	28	42	47	61	66	80	85
2	22	27	41	46	60	65	79	84
3	23	28	42	47	61	66	80	85
4	16	21	35	40	54	59	73	78
5	23	28	42	47	61	66	80	85
6	22	27	41	46	60	65	79	84
7	23	28	42	47	61	66	80	85

Table 7.7: Completion times of 8 8-point DCT

7.4.2.7. High-level synthesis exploration

Different architectures were explored with the GA Synthesis tool. Table 7.8 shows a selection of results obtained from the 8-point, 1-D and 2-D DCT implementations. These architectures are still restricted to a single input port with no input re-ordering. Areas (Resource usages) similar to the time-shared implementation are desired. The same type of hardware units are used.

The results may not be optimal but are comparable to highly-optimized hand-designs. The size of this algorithm is larger than the usual HLS benchmarks. The 2D DCT with transpose consists of 1212 operations (including storage operations).

The 8-point DCT runs were on the order of seconds to minutes. The 1D DCT runs were on the order of an hour or two. The 2D DCT with transform were on the order of several hours (grater than 10 hours).

Figure 7.14 shows the convergence of a typical 2-D DCT architectural exploration run. The different lines represent the minimum, maximum and average fitness of each

	MUL	ALU	Registers	Clock Cycles
8-point DCT	1	3	9	20
	2	6	11	19
1D DCT	1	4	18	85
	1	5	15	79
	2	8	33	75
2D DCT	2	8	41	151
	3	12	94	131
	4	16	107	118

Table 7.8: Sample HLS results

population of architectures. This is a run for 100 architectures evolved in 500 generations. The optimization criteria is a combination of the area and latency of the architecture.

Figure 7.14: HLS convergence for the DCT example

7.4.2.8. Performance comparison

Table 7.9 shows a comparison of the pipelined and time-shared architecture in terms of latency, throughput, hardware requirements and cycle time. Throughput is in pixels/cycle. This is done for the 8-point, 1-D DCT and the 2-D DCT (with the transpose function.). The two Improv. columns show the percentage improvements of the time-shared and synthesized architectures, respectively, over the pipelined one. The 8-point and 1-D implementations use 1 multiplier. The 2-D implementations use 2 multipliers. It can be seen that a tremendous improvement can be achieved in latency and area with a minor decrease in throughput.

		Pipelined (Manual)	Time Shared (Manual)	GA Synthesis	Improv.(%) TS vs. Pipe.	Improv. (%) GA Synth vs. Pipe
8 point	Lat.ency	67	20	20	70	70
	# regs	100	9	9	91	91
	# ALU	5	4	3	20	40
	Throughput	1	0.53	0.53	-47	-47
1-D DCT (8x8)	Latency	131	85	85	35	35
	# regs	100	18	18	82	82
	# ALUs	5	4	4	20	20
	Throughput	1	0.83	0.83	-17	-17
2-D DCT	Latency	326	159	151	51	54
	# regs	264	74	41	72	84
	# ALUs	10	8	8	20	20
	Throughput	1	0.85	0.9	-15	-10
MPEG ^a	Required	100	111	105	-11	-5
	Freq. (MHz)					

Table 7.9: Architecture Performance comparison

a. 1024x1024 color block; 30 frames/sec

What the -11% and -5% in the MPEG row mean is that to be able to support MPEG rates, the FUs have to be run at 11% or 5% higher frequencies than the original pipelined implementation. This is 111 MHz and 105 MHz respectively.

7.5. Clock Selection and Synthesis

7.5.1. EWF results

To examine the quality of the clock selection and synthesis techniques, we use the hardware units from [10]. The operation latencies and the possible set of optimal clocks according are shown in Table 7.10.

Operation type	Latency/Clock Periods
addition	48 ns
multiplication	163 ns
Optimal Clock Set	{ 19,21,24,28,33,41,48,55,82,163 }

Table 7.10: Operation latencies and resulting optimal clock set from [10]

The EWF benchmark is then synthesized to obtain the results in Table 7.11, all of which are optimal. Note that a maximum of one synthesis run was performed for each listed result as opposed to the 10 ILP runs required in [10]. (Some of the results were found in the same run). Also, many of the near optimal results were also found in the same synthesis executions.

While the 3D synthesis required up to n ILP runs (between 4 and 40 ILP runs) to obtain the optimal results, the GA synthesis converged to it in a single run with little overhead. There is however no guarantee of optimality in the GA Synthesis.

In this case, where multiple FUs with different clocks can have a varying effect on

Allocation	Delay (ns)	Clock Period (ns)	GA Synthesis time
1*,2+	1584	24	28.8 s
2*,2+	1152	24	13.6 s
3*,3+	1104	24	6.4 s
4*,3+	1032	24	6.4 s

Table 7.11: EWF with clock selection

mobility, one of the methods used was to define the ASAP schedule on the slowest clock and the ALAP schedule on the fastest clock.

Clock Period	DFG total latency (ns)				CPU time (s)			
	ASAP	1*,2+	2*,2+	3*,3+	ASAP	1*,2+	2*2+	3*,3+
163	2282	2608	2608	2282	0.22		2.9	0.22
82	1394	1722	1476	1394	0.22			4.38
55	1100	1595	1210	1155	0.22		24	19.2
48	1104	1632	1248	1200	0.22			
41	1394	1599	1230	1189	0.22			
33	1221	1617	1221	1190	0.22			
28	1120	1596	1232	1176	0.22		11.6	10.4
24	1032	1584	1152	1104	0.22	28.8	13.6	6.4
21	1197	1596	1155	1113	0.22			
19	1140	1596	1159	1121	0.22			

Table 7.12: Clock Selection/Synthesis Comparison between [10] and GA Synthesis

Table 7.12 shows a full suite of results compared with [10]. The left portion specifies the latencies obtained in the ILP method for multiple resource allocations. The right portion shows the CPU time required to achieve the corresponding results using GA synthesis. Only two runs of the GA synthesis was performed. The optimal results are for a clock period of 24ns.

7.5.2. Path-dependent clock selection

Our method goes even further by allowing multiple clocks in the CDFG.

Figure 7.15: CDFG example

Here we consider multiple clock selection for multiple paths in a CDFG. Consider the example in Figure 7.4. $c1$ and $c2$ are branch operations. For simplicity, the synthesis assumes that $c1$ and $c2$ do not incur any latency. We consider three different conditional paths ($P1$, $P2$ and $P3$).

If we synthesize the CDFG using a single clock, we obtain the results shown in Table 7.13 for 1 multiplier and 2 adders. The results for different allocations vary slightly. The table shows the worst and average latencies of the CDFG. They both occur for a clock period of $24ns$ (Note that this is not always the case).

Note that paths P1 and P2 have an addition that is shared.

Clock Period	CDFG Latency (ns)	
	Worst	Avg
163	978	815
82	656	601.3
55	550	531.6
48	624	576
41	656	601.3
33	594	561
28	560	541.3
24	552	520
21	588	560
19	570	551

Table 7.13: Synthesis results for 1*, 2 +

If we allow 3 different clocks, the results for the individual paths are as shown in Table 7.14. The table shows the latency of all paths for each clock selection. Note that multiple runs were used to obtain these results, as the algorithm typically converges to the optimum clocks quickly and many suboptimal results are not be encountered. For *P3*, the optimum clock period is *55ns* resulting in a *2ns* improvement over the *24ns* period. This is a small improvement, but it shows the benefit of investigating alternate clocks.

Clock Period	P1 (ns)	P2 (ns)	P3 (ns)
163	815	978	652
82	574	656	574
55	495	550	550

Table 7.14: Individual Paths Synthesis Results

	Clock Period	P1 (ns)	P2 (ns)	P3 (ns)
-	48	528	576	624
	41	574	656	574
	33	528	594	561
	28	504	560	560
	24	480	528	552
	21	525	588	567
	19	513	570	570

Table 7.14: Individual Paths Synthesis Results

The multi-clock selection results are shown in Table 7.15. We see improvements in the worst CDFG latency and the average of the 3 paths.

	Clock Period (ns)	Latency (ns)
P1	24	480
P2	24	528
P3	55	550
Worst	24,55	550
Avg	24,55	519.3

Table 7.15: Multi-clock selection results

7.6. Synthesis with Multiple Units

Throughout this chapter, the results shown considered only one type of FU per operation. This was done for comparison purposes. The GA synthesis methodology, however, allows for the use of multiple FUs per operation type. Many heuristics cannot accommodate this. Further, rather than the abstracted clock cycle latencies and relative integer areas, transistor counts and delays in ns can be used.

As an example of the trade-offs that can be performed we show the hardware library that was implemented in [66]. The FU characteristics corresponding to 3.3V implementations are shown in Table 7.16. Those corresponding to 5V implementations are shown in Table 7.17. The library contains two types of adders and 4 types of multipliers. The GA synthesis can easily select the best FU for each operation. This allows a better examination of the design space.

	Adders		Multipliers			
	RCA	CLA	QBS	BOOTH	ARR 16	ARR 32
Stage delay (ns)	25	11.8	8.8	11.8	72	158.5
# of stages	1	1	64	16	4	1
# of transistors	1,330	2,326	1,944	3,808	11,386	32,896
Average Power (mW)	5.4	10.5	11	12.7	24.6	143.1
Energy (nJ)	162	210	7,040	4,064	8,118	22,896

Table 7.16: 3.3V Adder and Multiplier characteristics

	Adders		Multipliers			
	RCA	CLA	QBS	BOOTH	ARR 16	ARR 32
Stage delay (ns)	17.1	7.0	5.7	8	39.2	98.5
# of stages	1	1	64	16	4	1
# of transistors	1,330	2,326	1,944	3,808	11,386	32,896
Average Power (mW)	22.7	37.3	28.5	84.0	101.3	295.6
Energy (nJ)	454	373	18,240	13,440	17,221	29,560

Table 7.17: 5V Adder and Multiplier characteristics

As a sample of the Area/Delay trade-offs we shown here three GA synthesis runs of the EWF benchmark using the hardware characteristics in Table 7.18.

The results are shown in Figure 7.16. Each GA synthesis run is represented by one

	Adders		Multipliers		Registers/RFs		
- Stage delay (ns)	12.5	5.9	144	79.25	0.7	1.1	1.3
Stages/Storage	1/0	1/0	4/0	1/0	0/1	0/2	0/4
# of transistors	1,330	2,326	11,386	32,896	288	480	800

Table 7.18: Hardware Library used for the 3 EWF runs

line. Different setups with different design space sizes and weights for the constraints were used for each run. In all cases, the tool searches through the appropriate design points in the space.

Figure 7.16: Example area/delay trade-off curves for the GA runs

7.7. Conclusion

This chapter presented results of the GA synthesis tool on several benchmarks and examples. The results were all selected to show certain aspects of the synthesis technique.

We showed using the EWF benchmark that the GA synthesis found optimal solutions of this problem faster than the ILP OASIC method. We also showed that better solutions can be found that the PSGA method which combines a problem-space GA with scheduling heuristics.

We also showed results of conditional synthesis (with CDFGs).on Kim's example and the Rotor example. The GA synthesis again finds all optimal solutions. It also optimized for average as well as worst paths. It had the advantage of finding multiple solutions in a single run.

We also showed results for larger circuits like the S2R and a suite of DCT circuits up to a 2-D 8point DCT implementation. As expected, for larger circuits, the GA method was faster than the symbolic method.

A suite of DCT architectures were synthesized and compared with two hand-crafted DCT architectures: a pipelined implementation and a time-shared implementation that was introduced in this thesis. The GA synthesis achieves good engineering results for this large graph that has more than 1200 functional and storage operations. The results are better than the hand-crafted methods.

We also showed results for clock selection and synthesis. We showed optimal results for scheduling and clock selection for the EWF benchmark in a short amount of CPU time. This has to be compared with methods where multiple ILP runs are required to find these solutions. We also showed an example of multi-path multi-rate clock selection and synthesis that result in better solutions than can be achieved with single-clock synthesis.

We briefly touched on the ability of using multiple units of the same type and showed examples of area/delay trade-offs for runs using multiple FUs per operation type.

Chapter 8: GA Evaluations

8.1. Introduction

This chapter shows some of the different optimization techniques that can be used in the GA synthesis tool. It also evaluates some of the GA parameters involved. It is important to note that the experiments shown are not statistically significant but offer anecdotal evidence to backup the use of some techniques over others. A more significant analysis would require an encoding where the genotype to phenotype mapping is more precise, allowing it to be analytically analyzed. The results shown here apply to a specific problem, with a specific encoding. That said, no experiments contradicted any of the conclusions found here.

We note here that the algorithms in use are probabilistic and will exhibit variation in the results. More experiments than the ones conducted here are needed to determine the effect of these variations. We can get an idea, however, by examining the effect of some of the parameters. Changing the crossover parameters for example (as can be seen in Figure 8.4) could result in a variation of up to 5%.

Caution should be applied when looking at the fitness values in the different results. The objective functions are not entirely consistent between the different experiments. The numbers are relevant for comparisons within specific sets of experiments.

Section 8.2 compares steady-state and generational replacement schemes. The results clearly indicate that the steady state converges faster to solutions with higher fitness. This agrees with the experiments run in [66]. We further look at two different replacement schemes and show that replacing below average solutions is an effective method.

Section 8.3 shows the results of varying the rate of the GA operators. First we examine varying the mutation and crossover rates. We find that the GA optimization is relatively robust to the choice of parameter probability, except for extreme choices such as zero mutation or zero crossover. Then we concentrate on experiments with crossover rate of zero. We conclude that crossover is an essential part of the optimization.

Section 8.4 examines the effect of using scaling. We show that the results can be dramatically improved using scaling to distinguish between *close* candidates.

Section 8.5 discusses crossing over at any bit location versus at the symbol boundaries. We show that crossing at the symbol boundary is less destructive to the basic building blocks and results in better solutions.

Section 8.6 casts the problem in a simulated annealing framework and compares the solutions with the GA optimizations. We show that the GA optimization outperforms simulated annealing in all the experiments that were run. We conclude that the GA seems to be a better solution for this problem.

Section 8.7 discusses different techniques of handling multi-dimensional objective functions. Here we run experiments, some of which have never been performed before, that optimize different individuals using different objective functions and then merge the two populations together. We show that optimizing separate populations for separate features and combining them results in better solutions than optimizing them using multi-modal objectives. We perform this for haploids and diploids. For the haploids the change is performed using an environmental change. For the haploids, we also add dominance changes.

Section 8.8 concludes this chapter and summarizes the experimental findings.

8.2. Selection schemes

-

8.2.1. Steady-state versus Generational

To examine the effect of different replacement strategies, we compare a generational replacement strategy with a steady-state replacement. the generational replacement replaces all individuals in a population with their offspring. The steady-state scheme used here replaces all individuals that have less than average fitness with new offspring. The results are shown in Figure 8.1. It is clear that the steady-state method provides better solutions faster. In many cases, we have observed that the generational method yielded worse results for reasonable run times. For this reason the steady-state method was used for most of the analysis shown in Chapter 7.

In Figure 8.1, we should note that the generational scheme does not guarantee that the individual with maximum fitness survives (The peak in the maximum fitness after generation 90 disappears). This does not mean that all historical information in the GA optimization is lost for the schema process is still working. The steady state method guarantees that the best solutions are kept alive.

8.2.2. Two different steady-state methods

Now we examine two different types of steady-state methods. The first is the below-average replacement scheme, the second replaces solutions with average fitness and below. Replacing individuals that are equal to or below the average fitness can lead to premature saturation.

The reason this occurs is that replacing all the individuals that are average or below implies that all the genetic material left comes from only the above average solutions. Initially, this is good as the average is much lower than the best solutions. As the average improves, the number of above average solutions decreases. The genetic material available

Figure 8.1: Steady-state vs. generational

decreases and is heavily biased towards specific solutions. Eventually the populations saturate.

Figure 8.2 shows the same run for a scheme that replaces average and below versus one that replaces only the below average individuals. This dramatically demonstrates the effect of premature saturation. Note that until about generation 18, the behavior is similar. This is when the average and below scheme saturates and the below average scheme recovers and has a dramatic improvement in both the average and maximum fitness values.

8.3. The effect of GA operators

8.3.1. The effect of the mutation rate

This section shows the results of varying the rate of the GA operators (crossover and mutation). Figure 8.3 shows the effect of the mutation rate on the average fitness for the EWF problem for an arbitrary 0.5 crossover rate. The mutation rate is varied from 0 to

Figure 8.2: Replacing below average or “average & below”

0.001 at 0.01 intervals. The trend of improving average fitness is clear. It can also be seen that for the initial 20 generations, the mutation rate does not make much difference. The mutation rate has more impact where the rate of improvement is highest. We can note, however, that the optimization is relatively robust with respect to the mutation rate.

An important point to note here is the saturation at 0 mutation. This is expected since without mutation (and with crossover at the symbol boundaries as explained in Section 8.5) the genetic pool is made of what was initially generated randomly. There is no mechanism to generate new genes, we are only changing the make-ups of the different individuals. It is interesting to note that a very low probability of mutation (0.001) is enough to generate new material and dramatically increase the fitness.

8.3.2. The effect of the crossover rate

Figure 8.4 shows the effect of the crossover rate on the average fitness for an arbitrary 0.005 mutation rate. The crossover rate is varied from 0.1 to 1 at intervals of 0.1. The same trends that are seen when varying the mobility can be seen here as well. The

Figure 8.3: Mutation rate effect

difference seems to be a more random distribution of average fitness in the latter generations. Again, we see that the optimization is relatively robust with respect to the crossover rate. This experiment may suggest that the crossover is not an essential operation for this problem. In fact many evolutionary computation techniques do not employ the crossover operator. We suggest here that while its effect is not clear in this experiment, the operator is essential. In the following section we run experiments for the same problem with and without the use of crossover.

8.3.3. The effect of crossover

To examine the need for these operators, since, it is not clear from the above what conclusion to draw about their values, we performed two sets of experiments. One was run with the crossover probabilities set to zero. This means that reproduction alone is working. The second was cast as a simulated annealing problem. The SA is done in Section 8.6.

Here we ran experiments with zero crossover rate. Note that the example run has a larger design space than the one used for the previous experiments. Figure 8.5 shows a run

Figure 8.4: Crossover rate effect

with $P_c=0$ (a) and $P_c=0.07$ (b). No change can be seen in the maximum fitness with the crossover rate changing from 0 to 0.07.

While the crossover did not show any effect in this case, we can see an effect in the next experiment shown in Figure 8.6. The difference is that the design space was slightly increased. This is done by increasing the mobility for all operations.

The crossover has a more important role in problems with large design space. For the smaller examples, the mutation, selection and reproduction mechanisms appear to suffice.

A crossover of 0.07 seems to be quite small. Figure 8.7 shows the same kind of experiment with the addition of a run corresponding to $P_c=0.5$. In this figure the MIN fitness values are not shown for clarity. We note the very slight improvement between 0 and 0.07. The larger crossover rate had a much more dramatic effect, however.

Figure 8.5: Zero crossover rate experiment
(a) $P_c=0$ (b) $P_c=0.07$

The results shown here are hardly conclusive but they seem to indicate that, in accordance with GA theory, the use of crossover as a parameter is useful. Note that multiple examples were examined with different random seeds and all the results lead to the same conclusions.

8.4. Scaling effect

Scaling was described in Section 3.3.11 where its usefulness was discussed. Here we show the results of optimizations with and without scaling using the GA synthesis tool.

The scaling method used in this tool was a linear scaling with a modification to eliminate negative fitness as examined in [27]. First a pre-scaling step is performed to determine the slope of the line used for scaling. In the following, f refers to the fitness value. The slope is calculated as

Figure 8.6: Zero crossover rate experiment with larger design space
(a) $P_c=0$ (b) $P_c=0.07$

$$\text{slope} = \frac{f_{avg}}{f_{max} - f_{avg}} \quad \text{if } (f_{min} > 2f_{avg} - f_{max})$$

$$\text{slope} = \frac{f_{avg}}{f_{avg} - f_{min}} \quad \text{if } (f_{min} \leq 2f_{avg} - f_{max})$$

the intercept is then calculated as

$$\text{intercept} = \frac{f_{avg}}{f_{avg} \times \frac{f_{max} - 2f_{avg}}{f_{max} - f_{avg}}} \quad \text{if } (f_{min} > 2f_{avg} - f_{max})$$

$$\text{intercept} = -f_{min} \times \frac{f_{avg}}{f_{max} - f_{avg}} \quad \text{if } (f_{min} \leq 2f_{avg} - f_{max})$$

Figure 8.7: Zero crossover rate experiment with larger design space
(a) $P_c=0$ (b) $P_c=0.07$ (c) $P_c=0.5$

The scaling then proceeds as $f' = slope \times f + intercept$.

This scaling does not result in negative fitnesses as can occur in normal linear scaling shown in Figure 8.8.

Figure 8.9 shows the synthesis results with and without scaling. It is clear that scaling avoids early saturation.

Note that while the replacement strategy is such that the replaced individual are not affected by scaling (since it is based on the average), the replacement strategy is still affected in that the individuals selected for reproduction (i.e. the roulette wheel) are affected.

Figure 8.8: Linear scaling
Can produce negative fitnesses

Figure 8.9: Scaling versus no scaling.

8.5. Crossover position

-

When using a binary encoded GA, the crossover can be performed at any bit location within the chromosome or it can be restricted to symbol boundaries. Here we show that restricting the crossover to symbol boundaries results in better solutions for this problem. This is expected as this type of crossover is less disruptive to the gene pool. It also agrees with the findings in [66].

The crossover shown in Figure 8.10 (a) is a single-point crossover where the crossing point can be anywhere along the chromosome. The crossover shown in Figure 8.10 (b) is limited to crossover points at the symbol boundaries. This means that the CC values and HW values do not get modified when passed to offspring except when acted upon by the mutation operator. The solutions observed consistently produce better results for the restricted crossover case. The reason may be due to the disruptiveness of the first method. It seems that the symbols are important to preserve as building blocks for the next generation.

Figure 8.11 shows another experiment (with a different setup and objective function) where crossing over at the symbol boundary results in better solutions.

8.6. Simulated Annealing versus GAs

Simulated Annealing is a stochastic relaxation technique based on the analogy of the physical process of annealing a metal. At high temperatures the atoms are randomly distributed. With decreasing temperatures, they tend to arrange themselves in a crystalline state which minimizes their energy. The algorithm then randomly generates new solutions. The decision of moving to a new solution is governed by the following probability:

$$P_{sa} = e^{\frac{-E_{sa}}{k_{sa}T_{sa}}}$$

Figure 8.10: Crossover location

E is related to the error decrease (increasing fitness) between the current solution and the next solution. T is a the temperature that starts high and is gradually decreased.

For this problem.

$$P_{sa} = e^{\frac{f_{old} - f_{new}}{T_{sa}}}$$

The temperature is related to the number of generations $T_{sa} = T_{sa0} - \alpha G_{sa}$ where T_{sa0} is the initial temperature and G_{sa} is the generation number.

In order to emulate simulated annealing with the same encoding of the function,

Figure 8.11: Crossover location II

we set the crossover and mutation rate to zero, use 1 individual in the population and modify the selection process to behave according to the annealing process.

The following comparisons run the same number of function evaluations (10,000 for the simulated annealing and 100 generations x 100 individuals for the genetic algorithm). We did not experiment extensively with the parameters and some of the results can arguably be improved by modifying parameters such as the cooling function. However, we attempted not to bias one optimization strategy versus another and did not use the more advanced techniques. This is simply to compare the basic optimization capabilities.

Figure 8.12 shows the results of the annealing process for the EWF experiment compared with the GA results. The same number of solutions are generated (100 generations with 100 individuals each for the GA and 10000 solutions for the SA). The SA

Figure 8.12: Simulated annealing vs. GA results I

run is shown in the top graph with the number of iterations on the x-axis. The GA run is shown in the bottom graph. It is clear that the GA outperforms simulated annealing (Note the dashed line on the GA graph representing the maximum value obtained using SA). In this case the maximum solution for the GA is $(0.026-0.023)/0.023=13\%$ better than SA. What is more important in our view is that the GA shows a more consistent improvement.

These runs had a relatively small design space. We need to examine the difference between GA and SA as the designs space increases. Figure 8.13, Figure 8.14 and Figure 8.15 show runs for progressively larger design spaces. Again, we increase the design space by progressively increasing the mobility of all operations in the DFG. The GA results are consistently better ranging from 7% to 13%. Because of the long constant SA line in Figure 8.14, it can be argued that the temperature was decreased too quickly. A more

Figure 8.13: Simulated annealing vs. GA results II

adaptive temperature function might have produced better results. This is not true for other experiments such as the one shown in Figure 8.15. Also, the same argument can be presented for the GA where adaptive crossover and mutation rates might be beneficial.

The results shown suggest that GAs were the better choice for this problem.

8.7. Multi-objective functions

This section discusses different techniques of handling multi-objective functions. We run experiments with novel ideas of combining one-dimensional optimizations. This is

Figure 8.14: Simulated annealing vs. GA results III

done for haploids and diploids with changes in the environmental conditions (objective functions) and dominance.

8.7.1. Multi-objective function with Haploid GAs

Haploids are single-stranded chromosomes as explained in Section 3.3.12. These present the most common approach to GAs and have been used for all experiments presented so far.

Figure 8.15: Simulated annealing vs. GA results IV

The assumption for the implementation of the multi-modal objective function has been, so far, that it is a combination of all the requirements and constraints. For instance it could be

$$f = TotalDelay \times TotalArea.$$

Here, we investigate a different method of multi-modal optimization. For all objectives that may work against each other (e.g. area and delay), we optimize for one objective only first and then, the next objective. For instance we will use $f = TotalArea$

We now implement a change in the fitness function. We run for 18 generations using the objective function

$$f_1 = \frac{1}{TotalArea}$$

then switch to the fitness

$$f_2 = \frac{4}{TotalDelay}.$$

Figure 8.17 shows the results of the optimizations used in the environmental change experiment. Shown are f_1 and f_2 which are used in the optimization. We can see that the area optimization saturates before the delay optimization starts. The use of 18 generations was picked to achieve this saturation. Certainly, different criteria can be used to decide when the environmental change should occur. Note that in this experiment we are not optimizing for what we really need at any stage which is

$$f = \frac{1}{TotalArea} + \frac{4}{TotalDelay}$$

Figure 8.18 shows the combined fitness for the environmental change experiment. This is the same run as Figure 8.17 but shows the combined fitness:

$$f_{ec} = \frac{1}{TotalArea} + \frac{4}{TotalDelay}$$

which was not used in the optimization. The resulting maximum fitness 0.0173 is an improvement of 6%. This demonstrates that there might be merits in seeding the population with a set of good solutions based on one or more objective. There might also be merits in limiting the objective function to single dimensions. The environmental change method achieves both of these. You can see in the figure that the maximum fitness

Figure 8.17: Environmental change with $f_1 = 1/\text{totalarea}$ and $f_2 = 4/\text{totaldelay}$
 Same as Figure 8.16 with one dimensional fitnesses and an environmental change.

does not always increase. This is simply because we are not optimizing using this fitness.

8.7.2. Multi-objective Functions with Diploid GAs

We can implement the same type of environmental change but using a feature that nature seems to have created specifically for this: Diploidy. Diploidy implies the use of double-stranded chromosomes with recessive and dominant genes as explained in Section 3.3.12.

In order to implement diploidy, the chromosome encoding was modified to a $\{0,1,2\}$ alphabet rather than a $\{0,1\}$ alphabet. In the first part of the evolution, the “2” is considered the dominant gene, the “1” is recessive. This implies that their values can be mapped to the original “1” and “0” respectively. Simultaneously with the environmental change affecting the fitness we implement a dominance change and the “1” becomes

Figure 8.18: Environmental change experiment with the combined fitness f_{ec} . Same as Figure 8.17 showing the multi-dimensional function that was not used for optimization but is monitored during the environmental change experiment.

Figure 8.19: Diploid with no change of dominance nor environmental change

dominant while “2” is recessive. When initially seeding the population we ensure that the probability of getting a “0” is 50%, the probability of getting a “1” is 25%, that of getting a “2” is 25%. This results in the same kind of probabilities distribution of mapped “1”s and “0”s as in the haploid.

Figure 8.20: Diploid with a dominance change at 50 but no environmental change.
“2” dominant, “1” recessive switches to “1” dominant and “2” recessive

In the experiments, we use the area as the fitness criteria and modify to the delay in the same manner as the haploid experiment. This results in what are shown in Figure 8.19 through Figure 8.23.

Figure 8.19 shows the solution with no change in dominance and no environmental change. This is a normal run (but is different from the haploid run). It results in a maximum fitness of 0.0397.

Figure 8.20 shows the solution with a dominance change at generation 50 but no environmental change. We see that this change results in worse solutions. No benefit is extracted from it.

Figure 8.21: Diploid with a dominance change 50 and environmental change at 50 “2” dominant, “1” recessive switches to “1” dominant and “2” recessive. Shown is the combined fitness not the two fitnesses used for optimizations f_1 and f_2 .

Figure 8.21 shows the results for a dominance change at generation 50 that coincides with an environmental change. Shown is the combined fitness f_{ec} . The optimization fitnesses are $f_1 = 1/TotalArea$ then $f_2 = 1/TotalDelay$. The resulting maximum fitness is 0.0425 resulting in an improvement of 7.2%. So it appears that the change of environmental conditions (objective function) coinciding with a change of dominance improves the optimization potential for the combination of the functions used before and after the change. We have to note here that these modifications affect the complexity of the algorithm very little.

Now we perform the same experiment with the change happening at generation 18 rather than generation 50. Figure 8.22 shows the solution with a dominance change at generation 18 but no environmental change. In this case, the GA has time to improve producing a maximum solution of 0.0364 which is still less than the original run without dominance or environmental changes.

Figure 8.22: Diploid with a dominance change at 18 but no environmental change. “2” dominant, “1” recessive switches to “1” dominant and “2” recessive. Shown is the combined fitness not the two fitnesses used for optimizations f_1 and f_2

Figure 8.23 shows the results for a dominance change at generation 18 that coincides with an environmental change. Shown is the combined fitness f_{ec} . The optimization fitnesses are f_1 then f_2 . the resulting maximum fitness is 0.0458 resulting in an improvement of 15.4%.

8.7.3. Separate populations (multi-culturalism)

Another method of handling the multi-dimensionality in the GA synthesis is separating the populations and evolving them using different objective functions, then joining them under the final combined objective function. The first part will generate solutions that are very good in area and delay respectively for example and the second part will combine these solutions to obtain the best solutions for both timing and area.

Figure 8.23: Diploid with a dominance change 18 and environmental change at 18 “2” dominant, “1” recessive switches to “1” dominant and “2” recessive. Shown is the combined fitness, not the two fitnesses used for optimizations f_1 and f_2

This is implemented by splitting the population in two (unequal partitions are of course possible), and applying different fitnesses for them. After a certain number of generations, the original fitness is used on all individuals. Figure 8.24 shows a comparison of unified (a) and separate populations (b). The separate population run results in a slightly better solution (2%). Note that the fitness numbers are different than the earlier experiments since it is a different setup. The conclusion is that the process of evolving different populations favoring one of the multiple objectives and then combining them together can result in better overall solutions, although the improvement is small. Therefore, a good start if one needs to find the area/delay trade-offs is a population of solutions optimized for area and delay separately.

Figure 8.24: Comparison of (a) unified and (b) separate populations

8.8. Conclusion

In this chapter we ran several experiments to validate the choice of GAs in the synthesis problem, to examine the effect of the different GA parameters and settings and to study different techniques of improving the results obtained using the basic GA methods.

We presented results comparing steady-state and generational replacement schemes that clearly showed that the steady state converges faster to solutions with higher fitness. We examined two different replacement schemes and showed that replacing below average solutions is an effective method.

We presented the results of varying the rate of the GA operators and found that the GA optimization is relatively robust to the choice of parameter probability except for choices such as zero mutation or zero crossover. Experiments with zero crossover showed that the crossover operation is beneficial in the optimization.

We examined the effect of using scaling and showed that the results can be dramatically improved using a simple scaling method.

We compared performing crossover at any bit location or at the symbol boundaries and showed that crossing at the symbol boundary rather than at the bit boundary is less destructive to the basic building blocks and results in better solutions.

We compared the GA encoding to a Simulated Annealing encoding and showed results that indicate the GA produces better solutions than the SA validating its choice for this problem.

We discussed different techniques of handling multi-objective functions. Experiments like these do not appear in the literature. We showed that optimizing separate populations for separate features and combining them results in better solutions than optimizing them using multi-objective functions. This is true for haploid and diploid GA implementations.

In general we showed many ways that the results of the GAs can be improved without resorting to hybrid solutions or extensive local search techniques.

Chapter 9: The Gradient Method for Complexity Estimation

9.1. Introduction

For a complete CDFG synthesis approach, it is necessary to include the controller's contribution to the area and delay. This is not a trivial problem for CDFGs, especially when the controller is implemented as a finite state machine. This thesis introduces a technique that can help in the area estimation process. We refer to this technique as the *gradient* method. In this chapter, we present this method and show experiments comparing it to the established methods discussed in Chapter 4. We do not prove theoretically how the method works.

Section 9.2 introduces the gradient method. For a given state table (or cover) we can find the gradient as estimate of the closeness of any point in a 9-bit neighborhood to the other points.

Section 9.3 describes the gradient more formally. Here a different gradient estimate is derived using finite difference methods. We also show how the gradient is implemented. An example is shown comparing gradients to manual optimizations.

Section 9.4 shows some benchmark results comparing the gradient to the autocorrelation method and to the number of pre-simplification literals. The gradient outperforms the auto-correlation method and is clearly better than the pre-simplification literals.

Section 9.5 compares the gradient with the entropy method. Again, the gradient outperforms the entropy method for the benchmarks used.

Section 9.6 performs experiments on the permutation of the order and bits. These show that while the gradient is dependent on the bit and row ordering of the function's

cover, the inaccuracy introduced by this dependency is of the same order as the inaccuracy introduced by synthesis. This implies that the use of the gradient is a reasonable estimate.

Section 9.7 concludes the chapter and emphasizes the main results.

9.2. Gradient technique, an intuitive view

Spatial information that can aid in predicting the size of a digital circuit can be extracted using a heuristic inspired by gradient techniques used in pattern recognition [19]. Effectively, this gradient technique calculates the similarity between neighbors in a truth-table. Simply put, the more common the outputs of a circuit are for adjacent inputs, the smaller the design would be.

9.2.1. Gradient method

For a given state table, divide the table in 9 bit areas as follows:

a	b	c
d	e	f
g	h	i

Then define the gradient at bit e in the x direction as:

$$\nabla_{ex} = (a + 2b + c) - (g + 2h + i)$$

The gradient at bit e in the y direction then is:

$$\nabla_{ey} = (a + 2d + g) - (c + 2f + i)$$

The total gradient ∇_e is then defined as:

$$\nabla_e = \sqrt{\nabla_x^2 + \nabla_y^2}$$

An easier gradient for computational purposes can be defined as:

$$\nabla_e = |\nabla_x| + |\nabla_y|$$

This gradient provides a good representation of the 9-point neighborhood region. The bits in this 9 bit region can take on the value of 0 or 1. They can also include a “*don’t care*”. This *don’t care* can either be 0.5 or whichever of the 0 or 1 values minimizes the gradient. An alternative is a $[-1,0,1]$ encoding.

If added complexity is not detrimental a squared gradient approach can be used where ∇_e^2 is the gradient of the gradient table at point e .

Now, assuming the gradients of 9-bit neighborhoods in a table are proportional to the complexity of its digital hardware implementation, the problem of finite state machine area estimation reduces to dividing the state machine description into 9-point areas.

9.2.2. Why it works

The complexity estimate based on the *autocorrelation* and described in Section 4.3.3 depends on the number of similar outputs for inputs separated by a Hamming distance of one. The larger the similarities, the easier it is to implement a function. This mimics the process of logic synthesis which relies on this similarity of the outputs for closely related inputs to build the circuit.

The complexity estimate based on *entropy* and described in Section 4.3.4 is a statistical measure of the homogeneity of the circuit (percentage of 0, 1 and don’t cares)

for outputs producing a 1. This means that the less homogenous the function, the more complex its implementation will be.

The *gradient* is a similar approach to both of these methods. It is similar to the *autocorrelation* in the sense that it examines a neighborhood of points. We could, of course, make the space that the gradient examines the same one as the *autocorrelation* examines, but we would lose the computational simplicity of this approach. Therefore, while the *autocorrelation* examines points in a spectral neighborhood, the *gradient* examines them in the spatial neighborhood. This may appear as a disadvantage for the *gradient*. We will experimentally show results that indicate that the uncertainty introduced by the ordering of the function's variables (essentially the difference between the spatial and spectral spaces) is not larger than the difference in the synthesis results for the same ordering changes. This is discussed in Section 9.6.

The *gradient* is also similar to the *entropy* in the sense that it is a measure of non-homogeneity. We argue that it is a better measure since the *entropy* is purely statistical and does not incorporate any neighborhood information. Reordering the input variables for the *gradient* estimate will modify the result slightly but will keep essential information about the function. The *entropy* does not extract any of this information. In essence, the *gradient* incorporates some of the features of the *autocorrelation* measure and the *entropy* measure.

The *gradient* itself is nothing but a measure of the spatial difference in the given function. Again, this is in the spatial space, not the spectral space as in a Karnaugh map. This differentiation information is basically proportional to the complexity of building a circuit realization of the function. The analogy is with differentiating in image. The spatial differentiation of an image results in a simpler form that retains essential information about the image, namely the edges.

In the same vein, the gradient method is finding the “edges” in the boolean function and producing a simpler form that brings out these edges and suppresses the homogenous portions. If we simply sum up the values representing these edges, we have a

9.3. The Gradient technique and other representations

In this section we show how the gradient technique can be viewed as a finite difference gradient in cartesian space. We also show how one can derive different gradients. Some of the ones developed were attempted and the one already presented gave the best results. A better method can possibly be obtained using a different derivation of the derivative.

9.3.1. Definition

If we divide the circuit description (in a truth table or cover form) into the 9-point clusters as shown on the cartesian representation in Figure 9.1, then the gradient presented in Section 9.2 can be represented as

$$\begin{aligned} \nabla_{i,j} = & |(f_{i-1,j+1} + 2f_{i,j+1} + f_{i+1,j+1}) - (f_{i-1,j-1} + 2f_{i,j-1} + f_{i+1,j-1})| \\ & + |(f_{i-1,j+1} + 2f_{i-1,j} + f_{i-1,j-1}) - (f_{i+1,j+1} + 2f_{i+1,j} + f_{i+1,j-1})| \end{aligned}$$

We can also derive many other gradient representations based on approximations of the partial derivative

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i}$$

Consider the gradient form from [3]:

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i} = \frac{\partial^2}{\partial x^2} \left(\frac{\partial^2 f}{\partial y^2} \right)$$

which results in

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i} \approx \frac{1}{h^4} [f_{i-1, j-1} - 2f_{i, j-1} + f_{i+1, j-1} - 2f_{i-1, j} + 4f_{i, j} - 2f_{i+1, j} + f_{i-1, j+1} - 2f_{i, j+1} + f_{i+1, j+1}]$$

h is typically a small number usually around 0.1. For boolean systems h is 1. The derivation is shown in Appendix A.

9.3.2. Implementation

Let $f: \{0, 1, -\}^m \rightarrow \{0, 1, -\}^n$ where $-$ represents *don't cares*. This boolean function can be represented by a logical cover $C(f)$ consisting of product terms. Given this cover, we construct the set $C_t = [I \quad O]$ where I and O represent the encoded boolean input and output vectors. We can, then compute the gradient of this function based on this set. There is no need to obtain the full truth table of the function. The input and output portions of the cover will be estimated separately and the resulting complexities added.

Now the complexity estimation problem reduces to dividing the set into 9-point areas and summing the resulting gradients.

Then, in general, for a $n \times m$ block, the *gradient* is:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \nabla_{ij} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} |(b_{i-1, j+1} + 2b_{i, j+1} + b_{i+1, j+1}) - (b_{i-1, j-1} + 2b_{i, j-1} + b_{i+1, j-1})| \\ + |(b_{i-1, j+1} + 2b_{i-1, j} + b_{i-1, j-1}) - (b_{i+1, j+1} + 2b_{i+1, j} + b_{i+1, j-1})|$$

with a wrap-around boundary condition so that

$$\begin{cases} b_{-1, y} = b_{m, y} \\ b_{x, -1} = b_{x, n} \\ b_{-1, -1} = b_{m, n} \end{cases}$$

This describes an encoding of the bits $b_{i,j}$ in a cover implemented in a circular (or spherical) manner with the 9-bit area gradient summed over the whole region.

9.3.3. Gradient with sorted rows/columns

Factoring involves folding the truth tables to reduce the number of literals. To estimate this using the gradient technique, one can sort the columns and rows in order to maximize the number of ones in the top left corner and maximize the number of zeros in the lower right corner. This reduces the dependency on the ordering of the bits but does not completely eliminate it.

If we represent the cover as a $m \times n$ matrix S , then we would like to transform the matrix using row and column swapping operations resulting in a matrix S^* with the following characteristics:

$$\begin{aligned} \sum_{j=0}^{n-1} b_{i,j} &< \sum_{j=0}^{n-1} b_{i+1,j} & \forall i = 0 \dots m-1 \\ \sum_{i=0}^{m-1} b_{i,j} &< \sum_{i=0}^{m-1} b_{i,j+1} & \forall j = 0 \dots n-1 \end{aligned}$$

with $a_i, a_j \in \{-1, 0, 1\}$ being the matrix variables.

A better sorting technique would attempt to minimize the Hamming distance between the input vectors. This however would introduce more computational complexity into the method.

9.3.4. Example

As an example, consider a small FSM which toggles every time the serial input is “100”. Figure 9.2 shows the state transition diagram and the state table for this FSM. Note

The results for 5 different encodings are shown in Table 9.1 where estimates are compared to the results from the SIS [69] synthesis.

Encoding refers to one of the five state encodings used.

∇ is the gradient estimate

Lits (SOP) refers to the number of literals in SOP form prior to simplification.

Lits (FAC) refers to the number of literals in factored form prior to simplification.

SOP refers to the sum of products literal after synthesis

FAC refers to the factored number of literals after simplification

It is clear that increases in SOP literals are coincident with increases in the gradient estimates. The magnitude increase (The increase proportional to the magnitude) does not track as well. This is because this is a very small example where the different encodings result in minor differences. With the exception of encoding number 4, the gradient tracks the monotonicity. Also, the increase seems to be proportional. The number of literals, however, goes in an opposite direction for the last two encodings (SOP) and is clearly not a good estimate. We should note here that using NOVA [75] state encoding and minimizing using SIS resulted in 13 SOP literals and 11 FAC literals.

The cover C_l in this case is formed as follows: $C_l = [S_{n-1}I \quad S_nO]$ where S_{n-1} and S_n represent the current and next states respectively. I and O represent the input and output vectors.

Encoding	∇	Lits (SOP)	Lits (FAC)	SOP literals	FAC literals
1	572	92	42	20	18
2	572	92	43	20	18
3	576	100	45	21	17
4	576	96	51	22	19
5	580	80	48	24	22

Table 9.1: Toggle FSM example results

Let's examine the gradient values for the 1st and 5th encodings. Table 9.2 shows the function using the first encoding and the corresponding gradient coefficients (the values of the gradient at each point in the space).

One should note that the more homogenous areas result in gradients of 4 or 0 whereas the areas with many differences result in 8 or 12. Remember that the gradient calculation is wrapped around the edges.

Input	Output	∇ in	∇ out	∇
0000	0000	8 4 4 4	4 4 0 8	In 272
0001	0100	4 4 4 0	8 8 12 4	Out 300
0100	0110	4 4 4 0	8 4 8 4	Total 572
0101	0100	4 4 8 4	12 4 8 4	
0110	1110	4 8 8 8	8 0 8 0	
0111	0100	8 8 4 8	8 0 8 4	
1110	1111	8 4 4 4	8 4 4 8	
1111	1101	4 4 8 4	4 12 8 8	
1100	1001	4 8 8 8	4 8 4 8	
1101	1101	4 12 4 8	8 8 4 8	
1000	0001	4 12 4 8	8 8 8 4	
1001	1001	8 8 4 8	8 4 8 8	

Table 9.2: Gradient coefficients for the first encoding

Table 9.3 shows the results for the fifth encoding. We see that the landscape has changed dramatically. The homogenous areas have shifted around. In the input set, there are no more values with very high gradients such as 12, and the lower valued gradients (0) have disappeared as well. This was just enough to cause a total change of 8 in the total gradient value.

Again, it is difficult to have good proportionality between the gradient and the synthesis results with such a small example. The gradient being a spatial technique requires a significant space. We will see better correlation with larger examples.

Input	Output	∇ in	∇ out	∇
0000	0000	8 4 8 4	8 8 4 4	In 280
0001	0001	4 4 8 4	4 4 8 4	Out 300
0010	0010	4 8 8 8	4 4 4 4	Total 580
0011	0001	4 8 4 8	8 4 8 8	
0100	0011	4 4 4 4	8 4 8 4	
0101	0001	4 4 8 4	12 4 8 4	
0110	1011	4 8 8 8	8 8 4 4	
0111	1100	4 8 8 4	4 8 4 8	
1000	1101	4 8 8 4	4 8 4 8	
1001	1100	4 8 8 8	8 12 8 12	
1010	1000	4 4 8 4	4 8 4 12	
1011	1101	8 4 8 4	4 8 0 8	

Table 9.3: Gradient coefficients for the fifth encoding

9.4. Benchmark Results

This section presents the results of the application of the gradient method to a sample of MCNS benchmarks [49]. These benchmarks are combinatorial. These results are displayed graphically in the next section.

Table 9.4 shows the results obtained for the two-level benchmarks from the MCNC benchmark set.

Ins is the number of inputs

Outs is the number of outputs

Patts is the number of patterns or vectors in the description

Lits is the number of literals before simplification

SOP refers to the number of literals obtained after synthesis using *SIS* [69] (*script.rugged* followed by a *full_simplify*)

FAC is the number of factored literals obtained after synthesis using *SIS*

$\ln(\mu')$ shows the results of the *autocorrelation* method from [72]

∇ is the gradient which is simply the sum of the input and output gradients as presented earlier. A $[-1, 0, 1]$ encoding was used.

These benchmarks offer a variety of sizes, input and output vector sizes and numbers of patterns in the cover.

	Ins	Outs	Patts	Lits (SOP)	SOP	FAC	$\ln(\xi')$ [72]	∇
5xp1	7	10	75	296	146	113	5.69	3058
9sym	9	1	87	522	287	211	6.91	3614
Z5xp1	7	10	128	4032	142	116		11520
Z9sym	9	1	420	3780	239	183		20200
alu4	14	8	1028	7483	244	191		75656
apex1	45	45	206	9133	1688	1532		34760
apex5	117	88	1227	7106	815	771		133988
b12	15	9	432	292	96	77		26014
bw	5	28	87	413	168	157	4.06	3536
clip	9	5	167	888	170	129	7.3	9436
con1	7	2	9	23	21	20	5.46	190
cordic	23	2	1206	18369	104	74		88844
cps	24	109	1308	7156	1361	1227		90728
duke2	22	29	87	2174	512	439		8468
e64	65	65	65	2145	253	253		14980
ex4	128	28	1860	4404	868	463		95366
ex5	8	63	256	60960	358	347		30628
inc	7	9	34	562	114	99		2604
misex1	8	7	32	122	58	51	6.2	1406
misex2	25	18	29	188	112	103		2134
misex3	14	14	1848	17971	867	683		134414
misex3c	14	14	305	1764	579	450		20444
pdc	16	40	2810	186972	456	421		391218
rd53	5	3	32	144	42	37	4.54	1056
rd73	7	3	141	840	71	68	6.30	5706

Table 9.4: PLA benchmark results

	Ins	Outs	Patts	Lits (SOP)	SOP	FAC	$\ln(\xi')$ [72]	∇
rd84	8	4	256	3288	350	340	6.8	16352
sao2	10	4	58	532	176	125	6.65	3278
spla	16	46	2307	17823	809	642		470010
squar5	5	8	32	425	65	59		2172
t481	16	1	481	4752	36	36		29308
table3	14	14	175	7381	944	779		22820
table5	17	15	158	7259	1001	910		22150
vg2	25	8	110	804	106	87		8048
xor5	5	1	16	80	16	16	5.77	472

Table 9.4: PLA benchmark results

9.4.1. Gradient versus Autocorrelation

Figure 9.4 compares the gradient estimate with the autocorrelation versus the resulting number of *SOP* literals. The benchmark results are sorted by increasing *SOP* value. All values (from Table 9.4) are normalized. The straight line is the ideal (normalized *SOP* vs. *SOP*). It can be seen that both have problems but the gradient is a much better estimate than the log of the *autocorrelation*.

The normalization is performed to enable the comparison of the different methods on the same graph. It is important for the estimate to be monotonically increasing. It is best if it approaches the ideal line when normalized.

9.4.2. Gradient versus literals

Figure 9.5 shows the difference between using the gradient estimate and the number of literals for increasing *SOP*. The x-axis shows the number of *SOP* literals after

Figure 9.4: Gradient and Autocorrelation estimate vs. the reduced SOP
Ideal is the normalized SOP literals after simplification.

▽ is the gradient estimate

SIS synthesis. The y-axis shows the normalized estimates for the two estimates.

One would like to see $finalsize = k \times estimate$. Normalization attempts to make all the constants k equal.

It is not clear from these experiments that the gradient estimate is better than the number of literals even though some of the estimates produced by the number of literals are clearly bad. The reason is probably due to the effect of the number of patterns on the gradient. The size of the cover might be skewing the results. This is due to the fact that we simply sum all the gradients in a cover which is clearly a gross estimate and is a function of the size of the cover.

If we use the measure $\zeta = \frac{\nabla}{\Pi}$ with Π being the number of patterns, then we get the results in Figure 9.6. This shows that the dependency on the number of patterns is

Figure 9.5: Gradient and literals with respect to the reduced SOP

Figure 9.6: Gradient/pattern vs. lits with respect to SOP

dramatically reduced. We note, however that the estimate is still not excellent for the larger circuits. These are circuits with small number of literals that the synthesis tool could not reduce. This might be a limitation of the synthesis more than a limitation of the estimate. Regardless, the gradient seems to follow the shape of the ideal line much more closely than the number of literals. Note that while this is not shown, dividing the number of literals (*Lits*) by the number of patterns does not improve the estimate.

Figure 9.7: Gradient/pattern vs. lits with respect to SOP for small circuits

Figure 9.7 shows the first 10 benchmarks (which were not clear in Figure 9.6). Again, we are comparing the number of literals (*Lits*) to the *gradient/patts*. We can clearly see that in most instances the gradient measure is closer to ideal than the number of literals.

Figure 9.8 shows the same first 10 benchmarks but used the *Lits/patts* rather than the *Lits* measure. The large errors in the Lits estimate are now eliminated but the accuracy of this complexity measure has not improved much. It is again clear in here that the

gradient measure outperforms the number of literals measure.

Figure 9.8: Gradient/pattern vs. lits/pattern for small circuits

9.4.3. FSM size estimation

While Section 9.4.2 presented the results on combinatorial benchmarks, this section will present the result of using the gradient estimate as a complexity indicator in the state encoding problem in FSM synthesis.

A FSM can be described by a quintuple $Q=(I,O,S,\omega,\lambda)$ where:

$I = \{x_1, \dots, x_m\}$ is the set of inputs,

$O = \{z_1, \dots, z_k\}$ is the set of outputs,

$S = \{s_1, \dots, s_n\}$ is the set of states,

$\delta\omega: SxI \rightarrow \mathcal{S}$ is the state transition function

and $\lambda: SxI \rightarrow \mathcal{O}$ is the output function.

We then construct the cover:

$$C_f = [[I, S_{n-1}] \quad [S_n, O]].$$

The complexity of realization is then estimated to be proportional to $\nabla(C_f)$.

To estimate the size of FSMs for different encoding the *dk14* MCNS benchmark [49] was used. It contains 56 patterns with 3 input bits, 5 output bits and 7 states.

14 different state encodings were used to produce the estimates shown in Figure 9.9. The encodings used binary and 1-hot encodings. After the encodings are selected, the FSMs are synthesized in the manner shown in Section 9.3.4.

Figure 9.9: Gradient and Literals vs. SOP for multiple FSM encodings

The values shown are normalized. It is again clear that the gradient provides a much better estimate than the number of literals before simplification. The straight line is the ideal estimate (normalized SOP).

FSM	inputs/ outputs	% Don't care	K(n)	H(P)	lits L(n,d,p)	Lits Random	Lits Mustang	∇
dk15	5/7	0.0	23	3.97	92	96	92	1860
dk16	7/8	15.6	80	6.03	400	383	307	8560
dk17	5/6	0	23	3.87	89	83	84	2564
dk27	4/6	12.5	12	3.18	33	29	29	684
dk512	5/7	6.3	23	4.34	93	82	84	1904
bbtas	5/5	25	23	2.99	51	35	36	1124
beecount	6/7	20.3	47	2.35	88	56	50	1502
red	7/6	70.3	84	2.39	60	60	65	- ^a
modulo12	5/5	25	23	3.59	62	42	43	1240
malati	9/9	91.6	338	4.25	120	193	181	- ^a
s8	7/4	84.4	84	2.26	30	31	27	1008
lion9	6/5	60.9	47	3.37	62	49	22	1216

Table 9.5: Gradient comparison with Entropy

a. Benchmark not found.

9.5. Comparison with Entropy

To compare with the entropy measure, we use the results from [11] on the following FSM benchmarks: dk15, dk16, dk17, dk27, dk512, bbtas, beecount, modulo12, s8 and lion9. *Red* and *malati* were not found in benchmark archives.

In [11], the entropy estimate was calculated and the FSMs were synthesized using a random encoding as well as an encoding by Mustang [16]. The results are shown in

Figure 9.10. The x-axis shows the entropy estimate, the y-axis shows the SOP literals obtained after synthesis using either a *random* encoding or the *Mustang* encoding. The *ideal* estimate line is also shown.

Figure 9.10: Entropy versus literals for random and Mustang encoded FSMs

To compare the gradient estimate with these results, we obtain the gradient estimate for the same FSMs and we synthesize them with an arbitrary encoding as well as an encoding using NOVA [75] which is part of SIS [69]. Figure 9.11 shows the results for the arbitrary encoding. Both the SOP and FAC number of literals are shown. We note that these estimates are closer to the ideal than the entropy estimates.

Figure 9.12 shows the results for the NOVA encoding. Again, we obtain a better linearity of the estimate than was obtained with the entropy measure. It also seems that the NOVA encoding results in slightly better estimates than the arbitrary encoding.

This section shows that the gradient estimate seems to outperform the entropy estimate for the FSM benchmarks used.

Figure 9.11: Gradient versus literals for an arbitrary encoding

Figure 9.12: Gradient versus literals for NOVA encoding

Figure 9.13: Permutation of the input and output variable ordering
Each point corresponds to a permutation showing an estimate versus the synthesis result

9.6. Permutations

Here we examine the effect of ordering on the gradient results. We know that a limitation of the gradient heuristic is that its dependency on the variables ordering. We generate multiple permutations of the PLA benchmarks, calculate their respective gradients and synthesize them. We can perform this with respect to the variable ordering (column) or pattern ordering (row) or both.

Ideally, the synthesis would obtain the 58 term answer for all cases. One sees from Figure 9.14 that it actually gates four answers.

9.6.1. Permutation of input and output variables

For the arbitrarily picked benchmark PLA `misex1.pla`, we generate 576 permutations of the input and output ordering and evaluate the gradient and synthesize them. Figure 9.13 shows the results of this experiment. SOP here is the sum of products literals after synthesis using SIS.

Figure 9.14: Normalized results for the permutation of the input and output variable ordering

Figure 9.15: Permutation of the input and output variable ordering

Figure 9.14 shows the same results in a normalized form. We note the variation in the gradient value is $\pm 5\%$ and the variation in the SOP synthesis value is $-2/+10\%$. There is more variation in the synthesized results than the gradient estimate. This indicates that, while the variable ordering affects the gradient, the effect is within the tolerance of the synthesis tools. This essentially eliminates the disadvantage in the use of the gradient.

We now perform the same comparison with the factored literals. This is shown in Figure 9.15 (non-normalized) and Figure 9.16 (normalized). The variation of the factored literals value is smaller than the SOP value at $-2/+8\%$ but this is still the same as the gradient variation.

Figure 9.16: Normalized results for the input and output permutations

Figure 9.17: Normalized results for the input, output and pattern permutations

Figure 9.18: Normalized results for the input, output and pattern permutations

9.6.2. Permutation of variables and patterns

Re-ordering the input variables is not the only option we have. The permutation in the row ordering is also of interest. More than 5000 permutations of the MISEX1 benchmark were tested.

Figure 9.17 shows the normalized gradient variation with respect to the SOP variation. The gradient is within $\pm 8\%$. The SOP value varies by more than $+10/-2\%$. Figure 9.18 shows the normalized gradient variation with respect to the FAC variation. The gradient is within $\pm 8\%$ and the FAC variation is within $+8/-2\%$.

We note, that while FAC and SOP variation is not affected by the re-ordering of the input patterns, the gradient is. The effect is, however minimal.

9.7. Conclusion

Using the spatial properties of boolean functions in a cover form, one can estimate the resulting hardware complexity of their realizations. The linear complexity *gradient* introduced in this thesis can be an effective method for general complexity estimation.

We showed here that the gradient is reasonable heuristic based on the neighborhood information in a cover.

We also showed that the gradient outperforms the autocorrelation and entropy methods for the PLA and FSM benchmarks presented.

We also showed that the gradient is a better estimate than the number of literals prior to simplification.

We examined the disadvantage of the gradient which is its dependence on the variable and pattern ordering and showed that the resulting error is on the same order as the error introduced by synthesis for the same ordering change.

Experiments were also run on finite state machines specifically for the state encoding problem. The gradient seems to track the efficiency of the different encodings rather well.

As a result we can conclude that the *gradient* is a good heuristic for estimating the controller area in high-level synthesis. Its simplicity is such that we can estimate the area for all generated schedules (and architectures) in an iterative method such as the GA synthesis method presented in this thesis without incurring any complexity penalty.

Chapter 10: Conclusion and Future Work

-

10.1. Conclusion

In this thesis, we described the need for a high-level synthesis tool that is flexible and adaptable due to the ever changing technology features and requirements. We also required the tool to be comprehensive allowing many optimizations that are not possible when the high-level synthesis problem is partitioned into subproblems or when certain subproblems are overly simplified. The tool needs to have a good global optimization strategy and should find good engineering solutions within a reasonable amount of time.

We then presented such a tool. It used genetic algorithms without application specific heuristics, thus satisfying the flexibility and adaptability requirement. the formulation used was comprehensive allowing general CDFG graphs to be synthesized. The synthesis comprised simultaneous allocation, binding and scheduling with clock rate selection and the use of real library functional units without the need to overly simplify any aspect. The GAs allowed for easy global optimization in a reasonable time.

We also presented a novel technique for estimating the area of controllers in the resulting architectures that is simple and was demonstrated to be effective. This method was experimentally shown to be effective in the estimation of the circuit area for general functions and to evaluate the state encoding in finite state machines.

We presented results supporting all the above claims.

10.2. Contributions

This thesis presented the following contributions (repeated from the introduction):

- **Using a GA formulation for the whole synthesis process not just scheduling:** Previous approaches either addressed the scheduling and allocation portions of straight line code only or they used extensive heuristics in the implementation. This method performs other synthesis tasks like control-dependent and control synthesis, register and mux allocation, and clock selection in a single GA framework.
- **Merging an exact coloring method with the GA synthesis formulation to implement CDFGs:** Previous methods either added constraints for conditional operations, transformed the graphs into non-conditional ones or used some coloring techniques to guide the heuristic schedulers. This method uses the coloring scheme to evaluate the solutions and relies on the GA for the optimization.
- **Providing a novel heuristic for the estimation of the area cost of controllers:** Previous methods used the area of a microcode ROM as the estimate. Most methods did not use the controller area during the optimization at all. This method integrates the area calculation with the optimization resulting in synthesized architectures that have better controllers. The heuristic is an improvement over complexity estimates in many logic design fields and can be used for area estimation of any logic function or state encoding problems in Finite State Machines¹. This thesis develops the method but does not integrated it with the synthesis program which is relegated to future work.
- **Implementing an exact clock selection method suitable for CDFGs:** Previous methods implemented clock selection in a heuristic manner or at very high cost. No method has considered the use of multiple clocks for different paths in a CDFG. Here, we present such a method that is well integrated in the same GA framework.
- **Providing a scheduling technique that improves existing heuristics:** A mobility reduction heuristic is presented. We prove that this technique can be applied to existing mobility-based heuristics and will improve their results. A good application for this

1. FSM state encoding was run on one example producing good results but further exploration is required and constitutes a major area for future work.

method is as an enhancement to force-directed scheduling that will produce more global solutions. The method can be easily integrated with an evolutionary technique to form a complete synthesis approach.

10.3. Future Work

- Implementing the mobility reduction evolutionary approach: This approach needs to be implemented and compared with the method used in this thesis. We suspect it will be more efficient. Note that implementing the mobility reduction approach without evolutionary techniques might also be useful, for instance extending Force-directed scheduling beyond the limited mobility reduction that has been implemented already.
- Integrate the gradient within the GA synthesis framework: The gradient method that was introduced in this thesis should be integrated with the GA synthesis methodology to estimate the controller sizes. This simply implies generating the FSM description of the controller from each GA individual. The gradient is then run on the FSM and the controller area is added in the objective function calculation.
- Improve the gradient estimate for incompletely specified functions: Some of the inaccuracy observed in the gradient estimate might be a result of the crude don't care representation used. A better representation may improve the quality of the gradient.
- Developing a different gradient based on finite difference differentiating techniques that result in better solutions and find a more theoretical basis to explain how the gradient works.
- Implement methods to improve the speed and scaling of GA synthesis: While the GA synthesis produced good results for relatively large problems in a relatively short time, it can still benefit from improvements in the speed and scaling capability. It is not clear how this can be achieved. Other than the potential of combining GA with heuristics

which could limit their generality, this should be pursued using different encoding approaches like the mobility reduction suggestion or by relying on GA research to provide improvements to the basic algorithm.

Bibliography

- [1] I. Ahmad, M. K. Dhodhi and C. Y. R. Chen, "Integrated Scheduling, Allocation and Module Selection for Design-Space Exploration in High-Level Synthesis," IEE Proceedings on Computers and Digital Technology, vol. 142, no. 1, January 1995, pp. 65-71.
- [2] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," IEEE Transactions on Computers, vol. C-23, January 1974, pp. 88-93.
- [3] Amir Wadi Al-Khafaji and John R. Tooley, *Numerical Methods in Engineering Practice*, HRW, Toronto, 1986.
- [4] Y. Arai, T. Agui and M. Nakajima, "A fast DCT-SQ scheme for images," Transactions of the Institute of Electronics, Information and Communication Engineers, vol. E71, no. 11, 1988, pp. 1095-1097.
- [5] Thomas Bäck, Ulrich Hammel and Hans-Paul Schwefel, "Evolutionary Computation: Comments on the History and Current State," IEEE transactions on Evolutionary Computation, vol. 1, no. 1, April 1997, pp: 3-17.
- [6] E. J. Baker, "Reducing Bias and Inefficiency in the Selection Algorithm", Proceedings of the Second International Conference on Genetic Algorithms and their Applications, Morgan Kaufmann, San Mateo, CA, June 1987.
- [7] Reinaldo A. Bergamaschi, Salil Raje, Indira Nair and Louise Trevillyan, "Control-Flow Versus Data-Flow-Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System," IEEE Transactions on VLSI Systems, vol. 5, no. 1, March 1997, pp. 82-100.

- [8] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Transactions on Computer Aided Design, vol. CAD-10, no. 1, January 1990, pp. 85-93.
- [9] En-Shou Chang, Daniel D. Gajski and Sanjiv Narayan, "An Optimal Clock Period Selection Method Based on Slack Minimization Criteria," ACM Transactions on Design Automation of Electronics Systems, vol. 1, no. 3, July 1996, pp. 352-370.
- [10] S. Chaudhuri, S. A. Blythe and R. A. Walker, "A Solution Methodology for Exact Design Space Exploration in a Three-Dimensional Design Space," IEEE Transactions on VLSI Systems, vol. 5, no. 1, March 1997, pp. 69-81.
- [11] Kwang-Ting Cheng and Vishwani D. Agrawal, "An Entropy Measure for the Complexity of Multi-Output Boolean Functions," Design Automation Conference, June 1990, pp. 302-305.
- [12] R. W. Cook and M. J. Flynn, "Logical Network Cost and Entropy," IEEE Transactions on Computers, vol. C-22, pp. 823-826, September 1973.
- [13] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, Montreal, 1990.
- [14] Giovanni De Michelli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.
- [15] T. C. Denk and K. K. Parhi, "Exhaustive Scheduling and Retiming of Digital Signal Processing Systems," IEEE Transactions on Circuits and Systems II, vol. 45, no. 7 July 1998, pp. 821-838.
- [16] Srinivas Devadas et al, "MUSTANG: State Assignment of Finite State Machines Targeting Multi-level Logic Implementations," IEEE Transactions on Computer Aided Design, vol. CAD-7, December 1988, pp. 1290-1300.

- [17] Srinivas Devadas and Richard Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Transactions on Computer Aided-Design*, vol. 8, no. 7, July 1989, pp. 768-781.
- [18] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhasker, "Datapath Synthesis Using a Problem-Space Genetic Algorithm," *IEEE Transactions on Computer Aided Design*, vol. 14, no. 7, August 1995, pp. 934-944.
- [19] R. O. Duda and P. E. Hart, *Pattern Recognition and Scene Analysis*, Wiley-Interscience, Toronto, 1973.
- [20] Nikil Dutt and Champaka Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop," Technical Report #92-107, University of California, Irvine, October 10th, 1992.
- [21] J. A. Fisher, "Trace Scheduling: A Technique For Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, July 1981, pp. 478-490.
- [22] L. J. Fogel, A.J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*, Wiley, New York, 1966.
- [23] M.R. Garey and D. S. Johnson, *Computers and Intractability: A Guide for the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
- [24] C. H. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, Boston, Ma, 1992.
- [25] C.H. Gebotys, "Optimal Scheduling and Allocation of Embedded VLSI Chips," *Design Automation Conference*, Anaheim, CA, June 1992, pp. 116-119.
- [26] E.F. Girczyc and J.P. Knight, "An ADA to Standard Cell Hardware Compiler Based

on Graph Grammars and Scheduling,” IEEE International Conference on Computer Design, Port Chester, NY, October 1984, pp. 726-731.

[27] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Ma, 1989.

[28] Guerra and Rabaey, “A Partitioning Scheme For Optimizing Interconnect Power,” IEEE Journal Of Solid-State Circuits, vol.32 no.3, March 1997.

[29] B. Haroun and M. I. Elmasry, “Architectural Synthesis for DSP Silicon Compilers,” IEEE Transactions on Computer Aided Design, vol. 8, April 1989, pp. 431-447.

[30] M.J.M. Heijligers, *The Application of Genetic Algorithms to High-Level Synthesis*, Eindhoven Technical University, Eindhoven, 1996.

[31] L. Hellerman, “A Measure of Computational Work,” IEEE Transactions on Computers, vol. C-21, May 1972, pp. 439-446.

[32] A. Hemani and A. Postula, “A Neural Net based Self Organizing Scheduling Algorithm,” European Design Automation Conference, Glasgow, March 1990, pp. 136-139.

[33] C. Y. Hitchcock and D. E. Thomas, “A Method of Automatic Data Path Synthesis,” Design Automation Conference, June 1983, pp. 484-489.

[34] J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, MA, 1975.

[35] C. T. Hwang, J. H. Lee and Y. C. Hsu, “A formal Approach to the Scheduling Problem in High-Level Synthesis,” IEEE Transactions on Computer Aided Design, vol. 10, no.4, April 1991, pp. 464-475.

- [36] R. Jain, M. Mlinar, and A. Parker, "Area-Time Model for Synthesis of Nonpipelined Designs," IEEE International Conference on Computer Aided Design, Santa Clara, CA, 1988, pp. 48-51.
- [37] M. Jamoussi, B. Kaminska, "Controllability and Observability Measures for Functional-Level Testability Evaluation," VLSI Test Symposium, 1993, pp. 154-157.
- [38] M. G. Karpovsky, *Finite Orthogonal series in the design of digital devices*, John Wiley, New York, 1976.
- [39] E. Kellerman, "A Formula for Logical Network Cost," IEEE Transactions on Computers, vol. C-17, September 1968, pp. 881-884.
- [40] Taewhan Kim, Noritake Yonezawa, Jane W. S. Liu, and C. L. Liu, "A scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach," IEEE Transactions on Computer Aided Design, vol. 13, no. 4, April 1994, pp. 425-438.
- [41] M. Kovac and N. Ranganathan, "JAGUAR: A fully Pipelined VLSI Architecture for JPEG Image Compression Standard," Proceedings of the IEEE, vol. 83, no. 2, February 1995, pp. 247-258.
- [42] J. R. Koza, *Genetic Programming*, MIT Press, Cambridge, MA, 1991.
- [43] D. Ku and G. DeMicheli, "Relative Scheduling under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," IEEE Transactions on Computer Aided Design, June 1992, pp. 696-718.
- [44] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, New Jersey, 1985.
- [45] B. G. Lee, "A new algorithm to compute the discrete cosine transform," IEEE

Transactions on ASSP, vol. 32, December 1984, pp. 1243-1245.

[46] J. H. Lee, Y. C. Hsu and Y. L. Lin, "A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis," IEEE International Conference on Computer Aided Design, Santa Clara, CA, November 1989, pp. 20-23.

[47] Youn-Long Lim, "Recent Developments in High-Level Synthesis," ACM Transaction on Design Automation of Electronic Systems, vol. 2, no. 1, January 1997, pp. 2-21.

[48] J. Lis and D. Gajski, "Synthesis from VHDL," International Conference on Computer Design, 1988, pp. 378-381.

[49] R. Lisanke, *Logic Synthesis Benchmark Circuits for the International Workshop on Logic Synthesis*, Microelectronics Center of North Carolina, May 1989.

[50] Tai A. Ly and Jack T. Mowchenko, "Applying Simulated Evolution to High-Level Synthesis," Transactions on Computer Aided Design, vol. 12, March 1993, pp. 389-409.

[51] C. A. Mandal, P. P. Chakrabarti, S. Ghose, "Complexity of Scheduling in High-Level Synthesis," VLSI Design, vol. 7, no. 4, 1998, pp. 337-346.

[52] Michael C. McFarland, Alice Parker and Raul Camposano, "The High-Level Synthesis of Digital Systems," Proceedings of the IEEE, vol. 78, no. 2, February 1990, pp. 301-317.

[53] MCNC High-Level Synthesis Workshops, HLSynth92, http://www.cbl.ncsu.edu/CBL_Docs/hls92.html, 1992.

[54] Zbigniew Michalewicz, *Genetic Algorithms + Data Structures = Evolution programs*, Springer-Verlag, New York, 1992.

- [55] S. Narayan and D. D. Gajski, "System Clock Estimation Based on Clock Slack Minimization," European Design Automation Conference, Hamburg, Germany, February 1992, pp. 66-71.
- [56] John A. Nestor and Ganesh Krishnamoorthy, "SALSA: A New Approach to Scheduling with Timing Constraints," IEEE Transactions on Computer Aided Design, vol. 12, no. 8, August 1993, pp. 1107-1122.
- [57] N. Park and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioural Specifications," IEEE Transactions on Computer Aided Design, vol. 7, no. 3, March 1988, pp. 356-370.
- [58] N. Park and A. C. Parker, "Synthesis of Optimal Clocking Schemes," Design Automation Conference, Las Vegas, NV, June 1986, pp. 454-460.
- [59] Alice C. Parker, Jorge T. Pizzaro and Mitch Milnar, "MAHA: A program for Datapath Synthesis," Design Automation Conference, Las Vegas, NV, 1986, pp. 461 - 466.
- [60] Pierre Paulin and John Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs," IEEE Transactions on Computer Aided Design, vol. CAD-8, no. 6, July 1989, pp. 661-679.
- [61] R. Potasman, J. Lis, A. Nicolau and D. Gajski, "Percolation Based Scheduling," Design Automation Conference, Orlando, FL, June 1990, pp. 444-449.
- [62] I. Radivojevic and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling," IEEE Transactions on Computer Aided Design, vol. 15, no. 1, January 1996, pp. 45-57.
- [63] K.R. Rao and J.J. Hwang. Techniques and Standards for Image, Video and Audio Coding, Prentice Hall.

[64] M. Rim and R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications," Design Automation Conference, Anaheim, CA, June 1992.

[65] Simon Ronald, "Robust Encodings in Genetic Algorithms," in D. Dasgupta, Z. Michalewicz, ed., *Evolutionary Algorithms in Engineering Applications*, Springer-Verlag, New York, 1997, pp. 29-44.

[66] R. San Martin, *Optimizing Power Consumption, Area and Delay in Behavioral Synthesis*, Carleton University Thesis 1995.

[67] R. San Martin and J. P. Knight, "Genetic Algorithms for Optimization of Integrated Circuit Synthesis," International Conference on Genetic Algorithms and their Applications, Morgan Kaufman, San Mateo, CA, 1993, pp. 432-438.

[68] H.-P. Schwefel, *Numerical Optimization for Computer Models*, Wiley, Chirchester, UK, 1981.

[69] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[70] Ravi Singh and John Knight, "Concurrent Testing in High-Level Synthesis," International Symposium on High-Level Synthesis, Niagara on the Lake, May 1994.

[71] Leon Stok, *Architectural Synthesis and Optimization of Digital Systems*, CIP_Gegevens Koninklijke Bibliotheek, Den Haag, 1991.

[72] D. Varma, E. A. Trachtenberg, "On the Estimation of Logic Complexity for Design Automation Applications," International Conference on Computer Design, Cambridge, MA, September 1990, pp. 368-371.

APPENDIX A: Algorithms and Derivations

- This appendix contains many of the algorithms used in this thesis and some of the basic algorithms in literature. It also contains the derivation of one of the gradient estimates.

A.1. List scheduling algorithm

Figure A.1 shows the basic algorithm for List Scheduling.

```
List_Scheduler(V,E)
  for all operations V{
    for all hw types {
      Determine scheduling candidate list L
      Determine remaining operation List R=V-L
      Select candidates with scheduled predecessors S
      S ⊆ L. These can now be scheduled
      Schedule operations in S subject to constraints
    }
  }
```

Figure A.1: List Scheduling Algorithm

A.2. Simulated Annealing algorithm

Figure A.2 shows the simulated annealing algorithm used in Section 8.6.

```

Initialize  $T$  (e.g.  $T = 100$ )
-    $T = 100$ ;
    generate new individual newpop[gen];
    While  $T > 0$  do
        evaluate newpop[gen]
         $\delta = \text{newfitness} - \text{oldfitness}$ ;
        if  $\delta < 0$  or  $\text{random} < \text{prob} = e^{-\delta E/T}$  then
            newpop[gen]
        else
            newpop[gen] = oldpop[gen]
        end if
        reduce  $T$  (e.g.  $T = 0.9T$  or  $T = T-1$ )
    end while

```

Figure A.2: Simulated Annealing

A.3. GA Synthesis algorithms

Figure A.3 shows the main GA synthesis algorithm.

```

GA basic algorithm
initialize a random population of individuals
evaluate fitness of all initial individuals of population
while not done do
    select a sub-population for offspring creation
    recombine the genes of selected parents
    randomly mutate new population
    evaluate fitness of the new population
select most fit individual

```

Figure A.3: basic GA Synthesis algorithm

Figure A.4 shows the main objective function calculation function.

```
- objfunc(critter)
    for all operations in operation list
        find number of bits assigned for hw and cc
        read hw bits and decode values
        use hw value to determine hw type
        get corresponding area, power, latency etc., from cost file
        adjust chromosome pointer and read cc value
        find latest precedent from precedence list
        add cc value to the absolute cycle of precedent
        insert token in vertical list (for resource cost)
    add up horizontal list token
    Evaluate cost (fitness) of chromosome
```

Figure A.4: Objective function calculation

Figure A.5 is the latest precedence routine that guarantees no operation is scheduled before its precedences are completed.

```
find latest precedence routine:
    Search for all operations in precedence list
    If op in list match current op
        then assign precedent as max
    if op in list does not match current op go to next op in list
    if no precedent, abs clock cycle = rel clock cycle
    Output abs cc is Input abs cc + latency - 1
    cc_abs_in = clock_cycle+searchccccurrent->cc_abs;
    cc_abs_out = clock_cycle+searchccccurrent->cc_abs +latency - 1;
```

Figure A.5: Find latest precedence routine

Figure A.6 shows the main algorithm for vertical traversal of the graph.

- *Vertical list processing routine:*

For all operations on the same HW unit and ID and in the same clock cycle, increment FU token count in list.

When list is complete, traverse and multiply occurrence by cost.

Figure A.6: Vertical List processing

There are, of course many other algorithms used in the GA synthesis that are not included here. Most of these were explained within the thesis. The algorithms included here are meant to give an idea of the type of processing involved.

A.4. Second order derivative

This section shows how to derive the second order derivative mentioned in Section 9.3.

Consider the following polynomial approximation:

$$f(x) = a_0 + a_1x + a_2x^2 \quad (\text{A.1})$$

The second order derivative is then

$$f^{(2)}(x) = 2a_2$$

Then from Equation (A.1) we have

$$(x_i = 0) \Rightarrow f_i = a_0$$

$$(x_{i+1} = h) \Rightarrow f_{i+1} = a_0 + ha_1 + h^2 a_2$$

$$(x_{i+2} = 2h) \Rightarrow f_{i+2} = a_0 + 2ha_1 + 4h^2 a_2$$

these equations can be solved to provide

$$\begin{Bmatrix} a_0 \\ a_1 h \\ a_2 h^2 \end{Bmatrix} = \frac{1}{2} \begin{bmatrix} 2 & 0 & 0 \\ -3 & 4 & -1 \\ 1 & -2 & 1 \end{bmatrix} \begin{Bmatrix} f_i \\ f_{i+1} \\ f_{i+2} \end{Bmatrix}$$

resulting in

$$f^{(2)}(x) = 2a_2 = \frac{1}{h^2}(f_i - 2f_{i+1} + f_{i+2}) \quad (\text{A.2})$$

Now, we derive the second order partial derivative in two dimensions.

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i} = \frac{\partial^2}{\partial x^2} \left(\frac{\partial^2 f}{\partial y^2} \right)$$

And from Equation (A.2) in the y direction,

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i} = \frac{\partial^2}{\partial x^2} \left[\frac{1}{h^2} (f_{i,j-1} - 2f_{ij} + f_{i,j+1}) \right]$$

$$= \frac{1}{h^2} \left(\frac{\partial^2 f_{i,j-1}}{\partial x^2} - 2 \frac{\partial^2 f_{ij}}{\partial x^2} + \frac{\partial^2 f_{i,j+1}}{\partial x^2} \right)$$

$$= \frac{1}{h^2} \left[\frac{1}{h^2} (f_{i-1,j-1} - 2f_{i,j} + f_{i+1,j-1}) - \frac{2}{h^2} (f_{i-1,j} - 2f_{i,j} + f_{i+1,j}) + \frac{1}{h^2} (f_{i-1,j+1} - (2f_{i,j+1} + f_{i+1,j+1})) \right]$$

-

which simplifies to

$$\left. \frac{\partial^4 f}{\partial x^2 \partial y^2} \right|_{x_i, y_i} = \frac{1}{h^4} [f_{i-1,j-1} - 2f_{i,j-1} + f_{i+1,j-1} - 2f_{i-1,j} + 4f_{i,j} - 2f_{i+1,j} + f_{i-1,j+1} - 2f_{i,j+1} + f_{i+1,j+1}]$$

V1 register refers to the registers connected to the first input of the ALU at each cycle.

V2 register refers to the registers connected to the second input of the ALU at each cycle.

Output register refers to the registers connected to the output of the ALU.

Type refers to the type of operation executed by the ALU at each cycle.

In refers to the input port.

CC	V1 register	V2 register	Output register	Type
	1,2,3	In,5,6,7,8	1,2,3,4,5,6,9	+
6	3	In	3	+
7	2	In	2	+
8	1	In	9	+
9	1	5	6	+
11	2	6	5	+
17	1	7	9	+
18	3	7	4	+
19	3	8	1	+

Table B.1: ALU1 connections

Table B.2 shows the connections to *ALU2* and all the register transfers. Here the type of operation is either an addition or a subtraction.

CC	V1 register	V2 register	Output register	Type
	1,3,4,7	In,2,5,6,7	1,2,3,4,5,7	+/-
5	4	In	5	+
7	4	6	7	+
8	1	In	4	-
9	3	2	2	+

Table B.2: ALU2 Connections

CC	V1 register	V2 register	Output register	Type
-	1,3,4,7	In,2,5,6,7	1,2,3,4,5,7	+/-
10	7	5	5	-
17	4	2	1	-
18	3	7	2	-
19	1	2	3	-

Table B.2: ALU2 Connections

Table B.3 shows the connections to *ALU3* and all the register transfers. Note that *ALU3* is being used strictly as a subtractor.

CC	V1 register	V2 register	Output. register	Type
	2,3,6,9	In,2,3,5,9	1,3,4,6,7	-
5	3	In	6	-
6	3	In	4	-
7	2	In	6	-
8	2	3	1	-
9	9	5	3	-
12	6	2	6	-
19	3	9	7	-

Table B.3: ALU3 Connections

Table B.4 shows the connections to *ALU4* and all the register transfers.

CC	V1 register	V2 register	Output Register	Type
	1,2,3,6,8	4,7,9	1,3,5,8	+
8	6	4	8	+
9	6	4	5	+

Table B.4: ALU4 connections

CC	V1 register	V2 register	Output Register	Type
-	1,2,3,6,8	4,7,9	1,3,5,8	+
10	3	9	1	+
17	8	7	8	+
18	2	4	3	+
19	1	9	8	+

Table B.4: ALU4 connections

Table B.5 shows the connections to *MUL1* and all the register transfers. Note that all the connections to port 2 come from the constants.

CC	V1 register	V2 register	Output Register	Type
	1,5,7,8	m1,m2,m3,m4	1,2,7,8	*
8	7	m3	1	*
9	8	m1	2	*
10	5	m4	8	*
11	5	m2	7	*
12	1	m1	7	*

Table B.5: MUL1 connections

B.3. Scheduled 8-point DCT DFG

Figure B.2 shows the scheduled graph for the original 8-point DCT DFG. The operations performed in the same clock cycles are shown in the same background colors. The clock cycles in which the operations are executed are displayed using arrows. Note that the input port scheduling is not shown.

Figure B.2: Example time-shared schedule for a 8-point 1D DCT

B.4. Resource scheduling

Table B.6 shows the resource scheduling table for a 1-D 8-point DCT.

cc	ALU ₁	ALU ₂	ALU ₃	ALU ₄	MUL ₁	I ₁	r ₁	r ₂	r ₃	r ₄	r ₅	r ₆	r ₇	r ₈	r ₉
0						a ₀									
1						a ₁	a ₀								
2						a ₂	a ₀	a ₁							
3						a ₃	a ₀	a ₁	a ₂						
4						a ₄	a ₀	a ₁	a ₂	a ₃					
5		b ₅ =a ₃ +a ₄	b ₂ =a ₂ -a ₄			a ₅	a ₀	a ₁	a ₂	b ₂	b ₅				
6	b ₄ =a ₂ +a ₅		b ₆ =a ₂ -a ₅			a ₆	a ₀	a ₁	b ₄	b ₆	b ₅				
7	b ₁ =a ₁ +a ₆	c ₂ =b ₂ +b ₆	b ₃ =a ₁ -a ₆			a ₇	a ₀	b ₁	b ₄	b ₆	b ₅	b ₃	c ₂		
8	b ₀ =a ₀ +a ₇	b ₇ =a ₀ -a ₇	c ₁ =b ₁ -b ₄	c ₆ =b ₃ +b ₆	e ₂ =m ₃ *c ₂		b ₀	b ₁	b ₄	b ₇	b ₅	b ₃	c ₂	c ₆	c ₁
9	c ₀ =b ₀ +b ₅	c ₃ =b ₁ +b ₄	c ₄ =b ₀ -b ₅	c ₅ =b ₃ +b ₇	e ₃ =m ₁ *c ₆ e ₂ =m ₃ *c ₂		b ₀	c ₃	c ₄	b ₇	c ₅	c ₀	c ₂		c ₁
10		d ₄ =c ₂ -c ₅		d ₃ =c ₁ +c ₄	e ₄ =m ₄ *c ₅ e ₃ =m ₁ *c ₆ e ₂ =m ₃ *c ₂		d ₃	c ₃	c ₄	b ₇	d ₄	c ₀			
11	S ₀ =c ₀ +c ₃				e ₇ =m ₂ *d ₄ e ₄ =m ₄ *c ₅ e ₃ =m ₁ *c ₆ e ₂ =m ₃ *c ₂		d ₃	c ₃	c ₄	b ₇	s ₀	c ₀			
12			S ₄ =c ₀ -c ₃		e ₆ =m ₁ *d ₃ e ₇ =m ₂ *d ₄ e ₄ =m ₄ *c ₅ e ₃ =m ₁ *c ₆ e ₂ =m ₃ *c ₂				c ₄	b ₇	s ₀	s ₄			
13					e ₆ =m ₁ *d ₃ e ₇ =m ₂ *d ₄ e ₄ =m ₄ *c ₅ e ₃ =m ₁ *c ₆ e ₂ =m ₃ *c ₂				c ₄	b ₇	s ₀	s ₄			
14					e ₆ =m ₁ *d ₃ e ₇ =m ₂ *d ₄ e ₄ =m ₄ *c ₅ e ₃ =m ₁ *c ₆		e ₂	e ₃	c ₄	b ₇	s ₀	s ₄			

Table B.6: Resource scheduling for 1-D DCT

cc	ALU ₁	ALU ₂	ALU ₃	ALU ₄	MUL ₁	I ₁	r ₁	r ₂	r ₃	r ₄	r ₅	r ₆	r ₇	r ₈	r ₉
15	-				$e_6=m_1*d_3$ $e_7=m_2*d_4$ $e_4=m_4*c_5$		e_2	e_3	c_4	b_7	s_0	s_4		e_4	
16					$e_6=m_1*d_3$ $e_7=m_2*d_4$		e_2	e_3	c_4	b_7	s_0	s_4	e_7	e_4	
17	$f_6=e_2+e_7$	$f_5=b_7-e_3$	$f_7=e_4+e_7$		$e_6=m_1*d_3$		f_5	e_3	c_4	b_7	s_0	s_4	e_6	f_7	f_6
18	$S_2=c_4+e_6$	$S_6=c_4-e_6$	$f_4=e_3+b_7$				f_5	s_6	f_4	s_2	s_0	s_4		f_7	
19	$S_1=f_4+f_7$	$S_3=f_5-f_6$	$S_7=f_4-f_7$	$S_5=f_5+f_6$			s_1	s_6	s_3	s_2	s_0	s_4	s_7	s_5	

Table B.6: Resource scheduling for 1-D DCT

B.5. 2-D DCT

Table B.7 shows the completion times of a 2-D 8-point DCT.

	0	1	2	3	4	5	6	7
0	89	96	101	115	82	120	134	139
1	85	104	109	123	90	128	142	147
2	84	103	108	122	89	127	141	146
3	85	104	109	123	90	127	142	147
4	78	97	102	116	83	121	135	140
5	85	104	109	123	90	127	142	147
6	84	103	108	122	89	126	141	146
7	85	104	109	123	90	127	142	159

Table B.7: Completion times of 2-D 8 point DCT

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved