**Compiler Optimizations For DSP
Using Genetic Algorithms**

by

David W. Horton, B.Eng.

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

Master of Engineering

Department of Systems and Computer Engineering
Faculty of Engineering

Carleton University
Ottawa, Ontario
January 5, 1997
©copyright
1997, David W. Horton

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - votre référence

Our file - notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-612-19518-X

Name David W Horton

Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

Compiler Optimizations For DSP using Genetic Algorithms

SUBJECT TERM

U S 4 4 UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS	
Architecture	0729
Art History	0577
Cinema	0900
Dance	0578
Fine Arts	0257
Information Science	0723
Journalism	0591
Library Science	0599
Mass Communications	0708
Music	0613
Speech Communication	0459
Theater	0465

EDUCATION	
General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Arts and Multicultural	0582
Business	0688
Community College	0275
Curriculum and Instruction	0277
Early Childhood	0518
Family	0519
Foreign	0517
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Human Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Nature of	0596
Philosophy of	0523

Psychology	0525
Reading	0533
Religions	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS	
Language	
General	0679
Ancient	0289
Indo-European	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Modern	0297
Modern	0298
African	0316
American	0391
Asian	0305
Canadian (English)	0352
Canadian (French)	0353
Chinese	0392
German	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY	
Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0489
SOCIAL SCIENCES	

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
Accounting	0310
Banking	0372
Marketing	0654
Management	0338
Merchandising	0385
Canadian Studies	
Geography	0501
History	0503
Literature	0505
Political Science	0506
Religion	0509
Sociology	0510
Theater	0511
Visual Arts	0528
Women's Studies	0544
Language	0551
General	0578

Ancient	0579
Medieval	0581
Modern	0582
Black	0528
African	0501
Asia, Australia and Oceania	0532
Canadian	0534
European	0535
Latin American	0536
Middle Eastern	0533
United States	0537
History of Science	

Law	0398
Political Science	
General	04151
Comparative Law and	
Religion	04161
Public Administration	0417
Recreation	0414
Social Work	0452
Sociology	
General	0426
Criminology and Penology	0427
Demography	0708
Developmental Studies	0431
Individual and Family	
Studies	0428
Industrial and Labor	
Relations	0429
Public and Social Welfare	0430
Social Structure and	
Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0432
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES	
Agriculture	
General	0673
Aquaculture	0285
Animal Culture and Nutrition	0675
Animal Pathology	0676
Food Science and Technology	0339
Forestry and Wildlife	0678
Plant Culture	0679
Plant Pathology	0680
Plant Physiology	0617
Rangeland Management	0777
Wood Technology	0746

General	0306
Acoustics	0307
Statistics	0308
Biology	0309
Cell	0379
Ecology	0329
Endocrinology	0353
Genetics	0363
Microbiology	0393
Neurology	0410
Molecular	0307
Neuroscience	0317
Optics and Optometry	0416
Physiology	0433
Rehabilitation	0621
Veterinary Science	0778
Psychology	0472
General	0784
Medicine	0780

Geology	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0445
Paleocology	0426
Paleontology	0418
Paleozoology	0485
Polymers	0427
Physical Geography	0348
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES	
Environmental Sciences	0768
Health Sciences	
General	0566
Anesthesiology	0300
Chemotherapy	0572
Dentistry	0572
Education	0350
Hospital Management	0769
Human Development	0738
Immunology	0566
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Safety	0354
Pharmacology	0381
Physiology	0319
Physiotherapy	0572
Podiatric Therapy	0382
Public Health	0573
Radiation	0575
Recreation	0575

Speech Pathology 0460
Tonicology 0383
Home Economics 0386

PHYSICAL SCIENCES	
Pure Sciences	
Chemistry	
General	0.685
Agricultural	0.779
Applied	0.684
Biological	0.684
Physics	0.687
Mathematic	0.688
Medical	0.728
Other	0.700
Engineering	0.691
Technology	0.694
Earth Sciences	0.695
Health Sciences	0.755

Physics	0603
General	0605
Astrophysics	0984
Astronomy and	
Astrophysics	0606
Atmospheric Sciences	0628
Atomic	0748
Electronic and Electricity	0607
Elementary Particles and	
High Energy	0798
Fields and Plasmas	0739
Mechanics	0609
Nuclear	0610
Optics	0782
Radiation	0756
Solid State	0611
Statistics	0463
Applied Sciences	
Applied Mechanics	0346
Computer Science	0986

Engineering	
General	0.037
Aerospace	0.038
Agricultural	0.039
Automotive	0.040
Biomedical	0.041
Chemical	0.042
Civil	0.043
Electronics and Electrical	0.044
Food and Thermal Systems	0.045
Hydraulic	0.045
Industrial	0.046
Materials	0.047
Marine Systems	0.048
Mechanical	0.048
Mathematical	0.043
Mineral	0.051
Nuclear	0.052
Packaging	0.049
Petroleum	0.045
Plastics and Municipal	0.046
System Sciences	0.050
Geotechnical	0.028
Operations Research	0.056
Systems Science	0.055
Teaching Technology	0.054

PSYCHOLOGY	
General	0421
Behavioral	0324
Clinical	0422
Developmental	0430
Experimental	0433
Industrial	0423
Personality	0425
Physiological	0439
Psychiatry	0349
Psychometrics	0432
Social	0451

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

Compiler Optimizations For DSP Using Genetic Algorithms

submitted by David W. Horton, B.Eng.

in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair

Department

Thesis Supervisors

External Examiner

CARLETON UNIVERSITY

Date April 15 / 97

Abstract

The problem of optimizing code generated by a high-level language can be difficult when the target architecture is that of a digital signal processor (DSP). These machines are often very specialized complex instruction set computers (CISC), whose architectures have many conflicting constraints. Traditional divide-and-conquer methods used in compilers targeted for a general purpose processor (GPP) may do a poor job in this environment. Furthermore, porting a compiler from one DSP architecture to another is hampered by their great dissimilarities.

A genetic algorithm (GA) is often a successful optimizing algorithm in the presence of a large number of constraints. Furthermore, the algorithm itself does not change if presented with a new list of constraints to work with. These two facts suggest that a GA could do well as the optimization engine for a DSP compiler when compared to traditional techniques, especially if portability is an important consideration.

This work takes an existing C compiler targeted for the Motorola 56000 processor family, and improves upon it by adding support for parallel memory moves and knowledge of the dual memory architecture of the M56K. With this as a base, a genetic algorithm is introduced into the optimizer to handle instruction combination, register assignment, and memory partitioning simultaneously, in an attempt to provide the optimizer with a more global perspective on this rather difficult optimization problem.

Acknowledgments

This work is a perfect example of Douglas R. Hofstadter's Law:

“Things always take longer than you think, even when taking
into account Hofstadter's Law.”

I've received lots of help during this long process. Many thanks to my parents for leaving me alone when I needed it, and for bugging me when I didn't. Thanks as well to friends, especially Sara, for keeping life interesting.

The people at NORTEL also deserves credit for making tools and generous funding support available.

Finally, thanks to my supervisor, John Knight, for making it all happen. If you need something done, just ask Professor Knight. Everyone else does.

Later...

Contents

List of Figures	vii
List of Tables	viii
List of Abbreviations	ix

Chapters

1	Introduction	1
1.1	The Need for DSP Compilers	1
1.2	Optimization in a DSP Compiler	2
1.3	Genetic Algorithms as Optimizers	3
1.4	Thesis Contributions	4
1.4.1	G56K Improvements	4
1.4.2	GA Driven Optimization	5
1.5	Thesis Organization	6
2	Digital Signal Processing	8
2.1	Overview of DSP	8

2.2	Digital Signal Processor Architecture	9
2.2.1	Memory Organization	9
2.2.2	Multiply Accumulate Instruction	10
2.2.3	Low Overhead Looping	10
2.2.4	Additional Addressing Modes	11
2.2.5	Fixed-Point or Fractional Arithmetic	12
3	Compilers	14
3.1	Purpose Of Compilers	14
3.2	Compiler Organization	15
3.2.1	Front-End Processing	17
3.2.2	Back-End Processing	19
3.3	Optimization	20
3.3.1	Initial Analysis	21
3.3.2	Peephole Optimization	23
3.3.3	Common Subexpression Elimination (CSE)	26
3.3.4	Constant Folding	27
3.3.5	Loop Optimization	28
3.3.6	Register Allocation	30
3.3.7	Dead Code Elimination	31
4	Genetic Algorithms	32
4.1	Biological Inspiration	32
4.2	Basic GA Components	33
4.2.1	Encoding	34
4.2.2	Fitness Evaluation	36

4.2.3	Reproduction Mechanism	36
4.2.4	Simple GA Pseudo-code	40
4.3	Schema Theorem	40
4.4	Simple Improvements	41
4.4.1	Selection Variations	42
4.4.2	Crossover Variations	43
4.4.3	Fitness Variations	44
4.4.4	Parallel Fitness Evaluation	47
5	Optimized DSP Code Generation	48
5.1	DSP Code Optimization Problem	48
5.2	Current DSP Programming Practice	50
5.3	Current DSP HLL Optimizing Approaches	51
5.4	A GA Approach to Optimization	52
6	Implementation	55
6.1	Choice of Tools	55
6.2	M56K Assembly Syntax	56
6.3	Goals	57
6.3.1	Parallel Memory Architecture Usage	58
6.3.2	GA Driven Optimization	59
6.4	G56K C Compiler	61
6.4.1	Important G56K Passes	62
6.4.2	Internal Representation	62
6.4.3	Machine Description (.md) File	63
6.4.4	N-Register Tracking	65

6.5	Parallel Combine Pass	66
6.5.1	New <code>x_memory</code> Keyword	67
6.5.2	Machine Description Modifications	67
6.5.3	<code>PAR_MOVE</code> / <code>PAR_ALU</code> Matching	69
6.6	GA Passes	69
6.6.1	Appropriate Passes For GA Control	70
6.6.2	Basic GA Choices	71
6.6.3	Encoding	72
6.6.4	Fitness Evaluation	74
7	Testing and Results	78
7.1	Parallel Combine Test	78
7.1.1	Simple Windowing Function	79
7.1.2	Simple Filtering Function	82
7.2	Dual Memory Space Test	85
7.2.1	Manual Memory Partitioning	85
7.2.2	Automatic Memory Partitioning	87
7.3	GA Combiner and Allocator Testing	87
7.3.1	GA Combiner Results	89
7.3.2	GA Allocator Results	90
7.3.3	Overall GA Results	91
8	Conclusions and Further Work	93
8.1	Thesis Contributions	93
8.1.1	Parallel Memory Moves	94
8.1.2	Manual Memory Partitioning	94

S.1.3	GA Instruction Combiner	94
S.1.4	GA Register Allocator	95
S.1.5	GA Memory Space Partitioner	96
S.2	Thesis Limitations	96
S.2.1	Computational Expense	96
S.2.2	Reproducibility and Randomness	97
S.2.3	Overlapping Memory Addresses	97
S.3	Appropriateness of the GNU Compiler	98
S.4	Final Conclusions	99
S.5	Future Work	100
References		106

Appendices

A	Sample DSP C Code	107
A.1	<code>fir.c</code>	107
A.2	<code>iir.c</code>	109
A.3	<code>dft.c</code>	111
A.4	<code>fft.c</code>	113
A.5	<code>minv.c</code>	117
A.6	<code>mdet.c</code>	121

List of Figures

3.1 Basic Compiler Organization	15
3.2 Front-End Processing	17
3.3 Parse tree for $A \times B + C$	18
3.4 Back-End Processing	19
3.5 Intermediate Processing	21
4.1 A Simple Bit-String Encoding for x and y	35
4.2 Roulette Wheel Selection Method	37
4.3 1-Point Crossover Mechanism	38
4.4 Baker's Multi-arm Roulette Wheel Selection Method	43
4.5 2-Point Crossover Mechanism	44
4.6 Uniform Crossover Mechanism	45

List of Tables

2.1	Bit-Reversal Patterns	11
4.1	Premature Convergence Example	46
4.2	Fitness Normalization Example	46
7.1	GA Combiner Results	89
7.2	GA Allocator Results	91
7.3	Overall GA Results	92

List of Abbreviations

ALU	Arithmetic and Logic Unit
C	The C programming Language
CC	Program name of the standard C compiler on UNIX systems
CSE	Common Subexpression Elimination
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor or Digital Signal Processing
FIR	Finite Impulse Response (Filter)
FFT	Fast Fourier Transform
frac	Fractional data type C-extension keyword
G56K	Motorola's port of GNU CC for the 56000 DSP
GA	Genetic Algorithm
GCC	GNU CC
GNU	Gnu's Not Unix (UNIX operating system clone)
GPP	General Purpose Processor or General Purpose Processing
HLL	High-Level Language
IIR	Infinite Impulse Response (Filter)

<code>inc</code>	Increment machine instruction
<code>INSN</code>	Basic unit of IR used by GCC
<code>IR</code>	Intermediate Representation
<code>M56K</code>	Motorola 56000 digital signal processor
<code>M68K</code>	Motorola 68000 general purpose processor
<code>mac</code>	Multiply-accumulate machine instruction
<code>.md</code>	Machine description file
<code>MDET</code>	Matrix Determinant
<code>MINV</code>	Matrix Inversion
<code>NORTEL</code>	Northern Telecom
p_c	Probability of Crossover
p_m	Probability of Mutation
<code>RTL</code>	Register Transfer Language
<code>SUS</code>	Stochastic Universal Sampling
<code>UNIX</code>	A common real-time operating system

Chapter 1

Introduction

This thesis considers the problems associated with building a good optimizing compiler for a digital signal processor. This chapter introduces these problems, and the suggestion that a genetic algorithm would be a useful tool in solving some of them. A listing of the thesis contributions is included, as well as a short summary of the following chapters.

1.1 The Need for DSP Compilers

Digital signal processors grew out of a growing need for special-purpose, performance-oriented computing devices capable of handling the speed requirements of digital computing. As performance was such an important issue, and program size fairly small, all programming was done by hand in assembly code.

When digital signal processing was still a novel concept this hand-coding was acceptable practice. As is typical, though, the demands made of these

processors has grown enormously. The use of DSPs has become commonplace, while their applications have become more complicated. To meet these new demands, designers using DSPs are looking towards DSP targeted compilers. A high-level language (HLL), such as C, provides greater flexibility, maintainability, and portability of software.

1.2 Optimization in a DSP Compiler

While performance is important in all software projects to some degree or another, in DSP programming insufficient performance means, in effect, that the software is useless. DSPs are found in embedded real-time systems, typically with very demanding timing constraints. Failure to meet those constraints at all times is often unacceptable. Yet because they are appearing in so many products, this performance is demanded with the most inexpensive hardware possible, meaning simple processors and limited memory.

Thus if software development for DSP is to move towards the domain of a HLL, the compilers for them must have good optimizers. While the theory and implementation of compiler optimizers for general purpose processors (GPP) is quite mature, this is not the case for DSP. The architecture of a digital signal processor is very different from today's general purpose processor. For example, DSP architectures normally include separately addressable memory spaces, not the single flat memory space of a GPP. DSPs have unique addressing modes and a complex instruction set, both of which directly support the particular algorithms that appear in DSP work.

1.3 Genetic Algorithms as Optimizers

The main thrust of this work is to explore solutions to some of the problems of compiler optimizers for DSP by using a genetic algorithm (GA). A GA is an optimization technique inspired by the process of natural selection. GAs form good solutions to problems by starting with a randomly generated population of multiple solutions. Pairs of these solutions are then mated, with the effect that good pieces of one solution are combined with those of a second, and a new solution, better than that of either parent, is generated. New generations of individuals are repeatedly produced until ultimately the best solution of a particular generation is chosen as the optimized solution.

Some of the characteristics of GAs address the problems encountered when implementing an optimizing compiler for a digital signal processor. Specifically, GAs can deal with the many constraints of a DSP architecture simultaneously, and allow for a global view of the optimization process. Specific issues, such as data partitioning between memory spaces, should be easy to introduce into a GA. GAs also tend to be quite flexible, which should prove useful for exploring tradeoffs between execution speed and memory consumption.

One common criticism of GAs is that they are computationally expensive when compared to many other optimization techniques. Thus GA applications have tended towards one-time only optimizations and other situations when speed is not critical. During writing and testing of a software project, however, a fast compiler is required as recompilations are frequently needed. From the outset it was assumed that a compiler with a GA optimizer would not be used during these stages of software development. Rather, the GA op-

timizer would only be enabled, and the entire project recompiled once, when the software was finalized.

1.4 Thesis Contributions

This work involves modifications to an existing C compiler, a port of GNU¹ CC called G56K, targeted for the Motorola 56000 (M56K) family of digital signal processors. There are two areas of change, namely improvements to G56K's knowledge of the target architecture, and the replacement of several optimizations passes of the compiler with a GA-based optimizer. The validity of this GA approach was then considered by using the new compiler.

1.4.1 G56k Improvements

The M56K hardware is capable of addressing dual-memory banks, called X-MEMORY and Y-MEMORY. Furthermore, it can perform memory moves to or from both of these banks at the same time it performs arithmetic and logical calculations on registers in the ALU. However, the original G56K compiler was not able to take full advantage of either of these features, severely restricting its practical use. This was addressed in the following ways:

Manual Memory Partitioning

In order to make use of the second memory bank, a new `x_memory` keyword was introduced into the compiler's understanding of the C syntax. By using

¹GNU stands for "Gnu's Not Unix", an implementation of a Unix-compatible software system, originated by Richard Stallman.

this language extension, the programmer is able to flag certain variables to be placed into the alternate memory space.

Parallel Memory Moves

A new pass was added to the end of the compiler's optimization process in order to rearrange code to perform data moves from one or both memory spaces in parallel with arithmetic and logical operations, subject to all restrictions of the M56k's architecture.

1.4.2 GA Driven Optimization

A genetic algorithm was added to G56k's optimizer. Replacement passes under GA control were developed for the original instruction combiner and register allocator, as well as an automatic memory partitioner for the newly added dual-memory support described in Section 1.4.1. The purpose of these passes in the compiler are as follows:

- **Instruction Combiner**

The combiner organizes the internal representation of the program being compiled into a series of instructions that tries to best match the target architecture's instruction set.

- **Register Allocator**

The allocator manages the distribution of the limited number of registers of the target architecture amongst the quantities appearing in the program being compiled.

- Automatic Memory Partitioner

Automatic memory partitioning relieves the programmer from needing to use the new `x_memory` keyword by choosing an appropriate memory space for certain quantities in the program, based on that quantity's usage.

Any combination of the three replacement passes can be performed at once. By placing some optimization passes under GA control, exploration of tradeoffs between choices made in each pass are possible. Certain uses of the GA, notably for instruction combination, produced consistently better compilation results over a variety of common DSP algorithms.

1.5 Thesis Organization

This thesis is organized as follow:

Chapter 2 describes some of the unique features of digital signal processors, stressing those elements that make them different from general purpose processors.

Chapter 3 provides background material on compilers and their organization, and includes a description of many common techniques they use to optimize their output.

Chapter 4 describes the essentials of genetic algorithms, and also presents some common improvements that can be made to them to adapt them better to the problem being solved.

Chapter 5 goes into greater detail about the problems associated with optimized DSP code generation, and details how the use of a genetic algorithm

is felt to be an appropriate mechanism to deal with them.

Chapter 6 provides implementation details about how the various changes were made to G56K, covering the parallel combiner, `x_memory` keyword, and genetic algorithm.

Chapter 7 shows results from the testing of the changes made to G56K, and compares the new compiler results to that of the original.

Chapter 8 presents a summary and final conclusion, as well as suggesting areas for further study.

Appendix A includes the C-coded DSP algorithms used in the testing of Chapter 7.

Chapter 2

Digital Signal Processing

This chapter provides background material on digital signal processing, and digital signal processors. The general organization of such a processor is discussed, with emphasis on those areas in which it differs from general purpose processors.

2.1 Overview of DSP

Digital signal processing¹, as a branch of signal processing, only became practical during 1970's with the coming of fast, relatively cheap digital computers. In DSP, digital data (discrete in quantity and time) can be transformed by computer software in the same way that analog data can be transformed with hardware.

This approach provides several advantages. Not only is the software

¹The abbreviation DSP is used to mean both the field of digital signal processing as well as the sort of specialized processor that performs such computations.

easily changeable, but it does not suffer from the inherent inaccuracy and degradation of analog hardware components. Furthermore, transformations not available in the analog domain (such as data compression) can be performed. Extensive coverage of the numerical theory of DSP can be found in texts such as [OS89] or [SK88].

2.2 Digital Signal Processor Architecture

DSP machines are optimized to do “number crunching” for specific types of algorithms while meeting regular hard real-time constraints. To support this their architecture differs greatly from that of a general purpose processor (GPP) [Lee88, Lee89]. Today’s GPP, while capable of the performance required for some DSP, is infeasible for many embedded applications due to its larger price-tag.

2.2.1 Memory Organization

The most evident difference between DSPs and GPPs is their memory organization. GPPs normally have the standard *von Neumann* architecture in which data and program memory are consolidated into a single flat space. This causes a memory bottleneck which in modern GPPs is solved with instruction and data caches and multi-port memory banks [HP90]. But high-performance memories are expensive and space consuming, thus inappropriate for DSPs. Furthermore, relying on caches is inherently unpredictable, making the hard timing constraints of DSP difficult to guarantee.

To prevent this memory bottleneck without resorting to such schemes

DSPs adopt a *Harvard* memory architecture with separate program and data memories, each with its own bus. Furthermore, many DSPs have two separate data memories, each being autonomous from the ALU. This allows for the simultaneous fetch of two pieces of data and one program instruction, an activity that can be taken advantage of in most DSP algorithms.

A further complication is that DSPs often have both on-chip and off-chip memory. Due to its proximity to the processor, the on-chip memory is faster (but limited in size). Programmers must divide data between the memories so as to minimize run-time while still satisfying size restrictions.

2.2.2 Multiply Accumulate Instruction

Another important architectural feature of DSPs is their direct hardware support for a multiply-accumulate (*mac*) instruction. Common DSP algorithms such as filtering or convolution involve repeatedly taking the sum of the product of two data values. This makes a single-cycle *mac* instruction almost mandatory for a good DSP chip.

2.2.3 Low Overhead Looping

DSP algorithms are also characterized by having extremely tight, fixed length loops. This is normally directly supported by the CPU with specialized loop counter registers which can be used to perform short loops with essentially no overhead.

Offset	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Offset
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Table 2.1: Bit-Reversal Patterns

2.2.4 Additional Addressing Modes

In addition to the addressing modes found in a typical GPP, a DSP often has special support for *bit-reversed* and *modulo* addressing. Efficient, direct support for these techniques by the machine can prove extremely useful as they are frequently found in often used tight loops.

Bit-Reversed Addressing

Enabling bit-reversed addressing on a DSP causes normal memory offset values to be mapped to their bit-reversed equivalents. Table 2.1 shows this pattern for four-bit wide offsets. This form of addressing is useful for walking

through arrays stored in the non-sequential order needed for the calculation of the *fast Fourier transform*.²

Modulo Addressing

DSP algorithms often access arrays of data in a circular fashion. Thus for array elements numbered 0 to $(N-1)$, indexing by any value n should actually access element $(n \bmod N)$. Modulo addressing supports this directly in the hardware.

2.2.5 Fixed-Point or Fractional Arithmetic

As a tradeoff between the added complexity of true floating-point calculations and simpler integer calculations, many DSP algorithms operate on what are called *fixed point* or *fractional* numbers, so named as the position of the decimal point is fixed so that only the fractional part of a number may be represented.

The motivation for such a data type is as follows. DSP applications demand that numbers with fractional parts be used. Simple integer arithmetic, then, is not sufficient for many applications. On the other hand, the hardware required for floating-point support is slow and expensive. It turns out that simpler fixed-point arithmetic and scaling can be substituted for floating-point, providing that certain overflow cases are considered.³ All of the hardware needed to support fixed-point is the same as that for integer

²See [OS89] pp. 594.

³Limiting *saturation* values are normally adopted to keep the propagation of errors to a minimum if fixed-point numbers overflow.

arithmetic. Thus the need for floating-point is provided with a minimum of hardware.

Digital signal processors, then, need only one set of hardware to do both integer and fractional arithmetic. It is up to the programmer, or sometimes the compiler to properly scale values, depending whether they are to represent integers or fractional numbers, and which of the two the target processor assumes.

For example, the M56K assumes all data to be fractional. Thus moving the hexadecimal value \$3F into a 24-bit data register does not yield \$00003F as would be expected for an integer value, but rather \$3F0000. If \$3F is to be interpreted as an integer, the programmer or compiler must scale it first by shifting the decimal place (in this case) 24 places to the left.

Chapter 3

Compilers

This chapter provides background material on compilers for high-level languages such as C. The organization of a typical compiler is described, as well as the reasoning behind that organization. The optimization process is described in some depth. A discussion of data and control flow is included, as well as detail about various possible optimization passes.

3.1 Purpose Of Compilers

A compiler is nothing more than program that translates one language into another. The compilers that we are interested in here are used to write computer programs. The source is a high-level language that allows programmers to describe algorithms in an easily readable fashion, while the target language is that of the computer the program is to run on. Compilers are the basic tool of the computer programmer.

Being so important, the basics of compilers are well understood. Fur-

Figure 3.1: Basic Compiler Organization

thermore, many tools have been developed to help in the fabrication of new compilers, especially ones to generate the *front-end*. A thorough discussion of all elements of compiler design can be found in [ASU86]. The following will discuss those elements of design that are common to most compilers, regardless of the language or target machine.

3.2 Compiler Organization

At the very least, compilers are organized into two distinct stages as depicted in Figure 3.1, a *front-end* and a *back-end*. The front-end is responsible for translating the programming language presented to it into an *intermediate representation*, or IR, that it passed to the back-end. That back-end takes an IR and translates it into assembly code appropriate for the target architecture. Optimizations may take place anywhere in the entire process, but most

commonly during the final stage or optional middle stage.

By introducing a standard IR, front and back-ends have an agreed upon means of communication. This allows for front-ends and back-ends to be interchanged to provide new compilers for new architectures. For example, a single back-end, targeted for the Motorola 68020, could be used with a C-to-IR front-end to make a complete C compiler, or a Pascal-to-IR front end to make a complete Pascal compiler for the same machine. This organization makes the support of new programming languages, and the porting of a compiler to a new architecture, greatly simplified through code reuse.

In reality, individual compiler vendors define their own IRs, making inter-vendor front and back-end swapping impossible. Also, given that some machine architectures vary greatly from others, IRs may not be expressive enough to describe the low-level concepts that the back-end needs to efficiently program the target CPU.

Figure 3.1 contains an optional third stage for *intermediate processing* between front and back-ends. If it exists it is used for code optimization, and is of particular interest for this work. While performing optimization on pure IR yields a more portable compiler, there are limits to what can be done as many common optimizations are very machine specific, requiring back-end knowledge. Optimization shall be treated in some detail in Section 3.3, while the following sections briefly detail the front and back-end. Further general discussion can be found in [Lem92], and [Hol90] contains detailed implementation information of a compiler written in the C language.

Figure 3.2: Front-End Processing

3.2.1 Front-End Processing

It is the job of the front-end to do the initial translation of the programming language to the IR of the compiler. Figure 3.2 depicts the two processes involved in this processing, *scanning* and *parsing*. The scanner reads in the program line by line, breaking it down into *tokens*. A token is a basic unit of language, such as a keyword or variable name. For example, consider the C-code:

```
if (x != 23) x += y;
```


Figure 3.3: Parse tree for $A \times B + C$

This statement contains the keyword tokens `if`, `(`, `!=`, `)`, `+=`, and `;`, two variable tokens `x` and `y`, and a constant token `23`.

Interpreting these tokens is the job of the parser. It groups tokens, ensures their validity, and forms them into a *parse tree*. An example of this for the expression $A \times B + C$ is shown in Figure 3.3. The parser recognizes the string of tokens that make up $A \times B + C$ as an expression containing two factors, one of which is itself an expression containing two factors. The parse tree, translated into some form of data structure, is the IR used by the compiler.

Figure 3.4: Back-End Processing

3.2.2 Back-End Processing

Figure 3.4 suggests that back-end processing is responsible only for *code generation*, which here is used to mean strictly the process of translating the purely logical form of IR into real code targeted for the machine. This may involve several choices, such as:

- Translation of IR to real machine instructions
- Register allocation
- When to use on-chip and when to use off-chip memory

- Which memory space to store variable in
- Stack space management

Should intermediate processing be performed by a separate stage of the compiler, many of these choices may have already been made, and code generation becomes extremely simple.

3.3 Optimization

Optimizations can be either machine dependent or independent. Independent optimizations can be performed directly on the IR used by the compiler, and handled in the separate intermediate processing stage. Other optimizations may have to be deferred until the back-end, unless the intermediate stage is given information to describe the target architecture.

Figure 3.5 shows the intermediate processing stage broken down into a series of optimizing passes. Each pass performs some transformation on the IR to remove useless information, do some form of compression on it, or perhaps reorganize it to better suit the overall program. These transformations must, of course, preserve the original ideas expressed by the programmer (although not necessarily in the form originally used).

By splitting the optimization problem into distinct, separate passes, an extremely complicated problem is broken down into manageable pieces. The drawback is that in making the problem of optimizing more tenable the global viewpoint of optimization is lost. This shall be further discussed when implementation of a GA driven optimizer is discussed in Section 6.3.

Figure 3.5: Intermediate Processing

3.3.1 Initial Analysis

One of the first steps in optimizing is to do some analysis on the IR. Generally this analysis consists of gathering information about both of control and data flow through the program.

Tracking Control Flow

To track control flow (the various paths that execution may take through a program) it is useful to first divide the IR into *basic blocks*. A basic block can

be defined as “a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end”.¹

Often optimizations are much easier to perform within a basic block due to the guarantee that the code involved must execute sequentially. It can be far more difficult or even impossible to apply the same optimization techniques when more than one basic block is affected. An example of this complication is given in the discussion of constant subexpression elimination in Section 3.3.3.

Control flow information can sometimes be used as an aid in guiding the optimizer to areas to concentrate on. For example, it might be more beneficial to optimize a heavily used loop at the expense of other more seldomly used code. In practice, it is very difficult for compilers to make such a decision, due to the data-dependency of control flow. Consider the pseudo code below:

```
if (cond1 is true) do
    action1
else do
    action2
```

Suppose that with typical data, `cond1` is true 99% of the time. If `action1` can be optimized at the expense of `action2` it should be. However, `cond1` could depend on user input, or perhaps previous calculations, neither of which the compiler can predict. Without some other external knowledge, it must treat both `action1` and `action2` equally.

¹From [ASU86], page 528.

Tracking Data Flow

Along with control flow certain optimization have need to know about data flow through the program. Data flow analysis involves knowing when and how data is used during the program. Essentially it is a matter of *scheduling*, that is deciding in what order what needs to be done. In the simplest case this involves knowing when a particular piece of data, such as a variable or a constant, becomes needed (is born), and when it is no longer needed (dies). Coupled with possible control flow paths, this information allows the compiler to track what resources it needs at any particular time, and perform *register allocation* (see Section 3.3.6).

Additional usage information, detailing exactly when data is used, can help when the question of code rearrangement is considered. Frequency of usage can also be looked at to ensure that often used data gets placed in faster memory, or in registers, in preference to other data.

3.3.2 Peephole Optimization

The simplest forms of optimizations, which are grouped with the name *peephole* optimizations, do not require any control or data flow information to perform. The term peephole optimization is used to describe transformations that are seen by considering small pieces of the program at once, as if viewed through a tiny peephole. Peephole optimizations can be done during intermediate processing, but are sometime more appropriately done during the back-end. The following sections detail some common techniques.

Algebraic Simplification

Algebraic simplification is just the application of simple algebraic relationships to the program code. Take as examples the statements $x \leftarrow x + 0$ or $x \leftarrow x \times 1$. Both can be removed without affecting the program. Such statements can arise during IR generation from the parser. In theory an endless number of such relationships could be tested for in a program. In practice, since these are the only ones likely to occur they are the only ones tested for.

Strength Reduction

The term *strength reduction* refers to replacing one operation, or set of operations, with others that yield the same values but are cheaper to execute. There is often a great deal of machine dependency involved.

For example, the calculation of $x \times 2^n$, with x and n integers, can be replaced by a left or right bit shift operation. This is invariably cheaper than straight multiplication. Another example would be the calculation of $x \times n$. On a machine with poor support for multiplication this might be better evaluated as n additions for small values of n . On another machine, with lots of dedicated hardware, multiplication might be as fast or faster than addition.

Jump Optimization

There are a variety of simple ways to remove extra jumps from code. Take the following two examples:

<pre> : jmp label1 : label1: jmp label2 : label2: : </pre>	<pre> : jeq label1 jmp label2 label1: : label2: : </pre>
--	--

In the first case the two chained unconditional jumps can be replaced by a single jump straight to `label2`. In the second case, the logic of the first conditional jump can be reversed to eliminate the second jump completely. There are many similar scenarios where certain adjacent or chained jumps can be eliminated from code. Often these situations arise after other optimizations rearrange or eliminate code that originally occurred between jumps or labels.

Machine Idioms

Particular machines have special purpose hardware instructions devised to make common operations faster. It is often of great advantage to specifically look for instances where they can be used. An example is auto increment or decrement addressing for walking through an array of values. For example, on the M68K, the following are equivalent:

<code>move.b (A1), D1</code>	
<code>addi #1, A1</code>	<code>move.b (A1)+, D1</code>
<code>move.w (A2), D2</code>	<code>move.w (A2)+, D2</code>
<code>addi #2, A2</code>	<code>move.l (A3)+, D3</code>
<code>move.l (A3), D3</code>	
<code>addi #4, A3</code>	

Special looping instruction or support for delayed branches could also be considered as machine idioms.

3.3.3 Common Subexpression Elimination (CSE)

Consider the following expression to evaluate x :

$$x \leftarrow c1 \times (a + b) - c2 \div (a + b);$$

The value of $(a+b)$ is called a *common subexpression* as it occurs more than once as a temporarily needed sub-computation in the calculating of variables. When translated directly to some IR representation, this expression for x might become:

```
t1 = a + b
t2 = c1 × t1
t3 = a + b
t4 = c2 ÷ t3
x  = t2 - t4
```

Here the t 's represent temporary storage, probably registers in the CPU. Obviously the calculation of $t3$ is redundant, as any subsequent use of it can be replaced with $t1$. This process is called *common subexpression elimination*. But now consider the following code:

```
if (cond1 is true) do
  x ← c1 ÷ (a + b);
x ← x - c2 ÷ (a + b);
```

Translated to an IR, this might become:

```

    cmp cond1 to 0
    jeq label1
    t1 = a + b
    x = c1 × t1
label1:
    t2 = a + b
    t3 = c2 ÷ t2
    x = x - t3

```

In this case, the common subexpression $(a+b)$ must be evaluated the second time, even though its value may already have been calculated. The complication is that the subexpression appears in two separate basic blocks (the presence of `label1:` indicates a possible control flow change).

3.3.4 Constant Folding

If the compiler can deduce at compile time that the value of an expression is in fact constant throughout the program, and it knows its value, then it can replace references to that expression by the appropriate constant. In C, for example, one might find the following:

```

#define NUM_COLUMNS 10
#define NUM_ROWS 100
#define NUM_ELEMENTS (NUM_COLUMNS*NUM_ROWS)

for (i=0; i<NUM_ELEMENTS; i++)
{ /* do something */ }

```

Instead of evaluating `NUM_ELEMENTS` as $(10 * 100)$ every time it is used, the multiplication can be done once at compile time and the result of 1000 used throughout the code. This procedure is called *constant folding* or *constant propagation*.

3.3.5 Loop Optimization

The presence of a loop often indicates a high usage area of code, and thus optimization done inside them are well worth while. They generally follow this form:

```

while (condition) do     $\Leftarrow$  loop logic
  whatever               $\Leftarrow$  loop body

```

Loops contain some body of code to be repeated, controlled by bracketing loop logic that determines the end condition for the loop. Loop logic also controls the values of *induction variables*, identifiers whose values depend on the number of iterations of the loop that have passed.

Movement of code from inside the loop to the outside is often considered. A common occurrence is with temporary calculations. Consider the following C fragments:

```

for (i=0; i<10; i++)    for (i=0, tmp=x/y; i<10; i++)
{
  a = f(i);              {
  sum = a+(x/y);          a = f(i);
  /* use sum */           sum = a+tmp;
  dif = a-(x/y);          /* use sum */
  /* use dif */           dif = a-tmp;
                          /* use dif */
}                          }

```

As shown, the value (x/y) can be calculated once outside of the loop instead of twenty times inside. Simple optimizations like this can be done directly by the programmer (if they notice it). But a good compiler can take over this responsibility from the programmer. Having the compiler do the movement itself allows the programmer to concentrate on other things, such as overall design, as well as retaining the better expressiveness of the

first version. Maintainability is also improved (consider the case of a later revision of the code changing the value of `x` or `y` inside the loop).

Furthermore, the compiler may be able to move code that the programmer does not have access to. Consider the following less obvious code movement possibility that occurs frequently in C++:

```
for (i=0; i<10; i++)
{
    someclass myvar;
    /* use "myvar" */
}
```

The programmer has followed good coding practice and placed the declaration of a new variable `myvar` inside the loop, restricting its scope to the loop body itself. But this causes the call to the constructor for class `someclass` to be performed at the beginning of every repetition through the loop. By knowing the specifics of this constructor call the compiler may be able to move some or all of it outside of the loop. The programmer may not even have access to this code at the source level.

Another simple means to speed up a loop is to unwind it. If the number of repetitions of a loop are known beforehand and constant, then the body can simply be repeated the appropriate number of times. This avoids the overhead of the loop logic itself, at the expense of memory space. This approach is only worthwhile for short, tight loops (i.e. the loop overhead is significant compared to the computation of the body).

Finally, the use of induction variables may be reduced or eliminated in a variety of ways. For instance, multiple induction variables that can be expressed in terms of one another may be collapsed into a single variable. In

other instances complicated array subscripting can often be expressed with simpler pointer arithmetic.

3.3.6 Register Allocation

Register allocation involves the mapping of the IR to the actual hardware of the machine being targeted, and can be the most complicated of all optimization techniques. While commonly done in the back-end, intermediate processing can handle most allocation so long as the register set of the target is fully described to it. Given that each register type may have different widths, and restrictions on its use, an IR must be carefully defined to allow for the description of multiple target architectures.

Register allocation itself can be done with a variety of algorithms, often differing depending upon how much of the program under compilation is being considered at one time. A common starting point is to assume in the IR an infinite register set, with every variable or quantity having its own register. Register assignment then becomes a problem of mapping this infinite set of registers to a finite set in the “best” way possible.

One possibility is to minimize the number of *memory spills* required for the mapping. A spill occurs when a particular register must be used that already contains a quantity yet to be used. This quantity must be temporarily “spilled” to memory, and restored later on. Using this idea, allocation within a basic block (local allocation) is a fairly straightforward procedure, and can be accomplished by tracking variable usage counts and using a technique called *graph colouring*.²

²See [Lem92] for details of this algorithm

Unfortunately, when considering variables that span multiple basic blocks (global allocation) the problem is complicated due to the possibility of changes in control flow. One approach taken is to treat local and global allocation completely separately, dividing all available registers amongst the two. This can cause an excessive number of memory spills, as even though registers are available they have been reserved for the other of local or global allocation. Furthermore, some special purpose registers will always be needed for both, and will always be in conflict.

A better approach is to prioritize certain registers for global and others for local allocation. Local is done first, followed by global. If global allocation requires a register already in use from local allocation, the local quantity must be relocated. If other local registers are available then it goes there. Otherwise, it may be spilled or, if more highly used than yet another local quantity it may take over the second local's register, causing a new quantity to need relocation. This procedure is repeated until all quantities have been found a home throughout all of the program.

3.3.7 Dead Code Elimination

As a final note on possible optimizations, code that is never used need never be generated. Unreachable code can develop as a result of some of the techniques described above. It is also common practice to place code in a program that is never intended to be in the final product, such as debugging code or extensions for future changes. Such unreachable code is easy to detect and eliminate from the output by looking for code beyond unconditional jumps or conditional jumps that can never be satisfied.

Chapter 4

Genetic Algorithms

This chapter provides background material on genetic algorithms. The basics of a genetic algorithm (chromosome encoding, fitness evaluation, and reproduction mechanisms) are described. The schema theorem is presented, along with a short discussion of its importance. Finally, some common extensions to the basic GA are described.

4.1 Biological Inspiration

Genetic Algorithms are one of a number of techniques belonging to a group of numerical methods called Evolutionary Computing. These techniques are inspired by the process of natural selection.

Natural selection is the process by which living beings evolve and change over time to better meet the requirements of their environment. Those individuals that are stronger and better suited tend to live, whereas the weaker tend to die. Thus, in the long run, it is the stronger individuals that are

successful and reproduce. The children of these individuals tend to have characteristics similar to their parents, most importantly those that allow the parents to survive. Thus the fitter the individuals, the more likely they are to reproduce and have similarly fit children.

In living beings, the information that goes into making up these characteristics is stored in chromosomes. These are made of long chains of complex molecules, called genes. Each gene, or set of genes, goes into defining some characteristic of the being. The chromosome of a child is made up from genes from both parents.

This sharing of genetic material means that the child will typically have some traits similar to its parents. But at the same time it also allows for new combinations of genes to occur in the chromosome, combinations that existed in neither parent. If these new traits are beneficial to the child, then it may be more fit overall, and have a better chance of survival, and thus a better chance of passing on these new genes to its children. If the new traits prove detrimental, the child's fitness will be lessened along with its chances for survival. Over time, the average fitness of individuals in a population will increase.

4.2 Basic GA Components

In [Hol75], genetic algorithms were presented using the mechanisms of evolution as a means to find good solutions to a problem. In a GA, an individual represents a particular solution to the problem being solved. The fitness of that individual corresponds to how well it solves that problem in comparison

to other individuals.

In the simplest case the GA starts with a collection of many randomly generated individuals, and pairs them off into parents. This pairing process is called *selection*. Each pair is then combined using two genetic operators, *crossover* and *mutation*, detailed in Section 4.2.3, to produce two new individuals (children) which replace the two parents in the next generation of the population. This process is continued, and over time the average fitness of the population grows. The process can be stopped for a variety of reasons, such as:

- a fixed number of generations occur
- no significant average fitness improvement
- a given length of computation time

The solution to the problem is chosen to be the individual in the final generation with the best fitness value.

An individual's encoding is represented by a string of bits, that is a series of 1's and 0's. This must be done in such a way so that the relative fitness of all possible strings can be evaluated. The processes of *encoding* and *fitness evaluation* are the only problem-specific parts of a GA. The *reproductive mechanism* is generic to all problems. These three elements of GAs are described in the following sections.

4.2.1 Encoding

The first problem in developing a simple GA as an optimizer for a problem is to map solutions into bit strings. Consider the simple example of optimization

Figure 4.1: A Simple Bit-String Encoding for x and y

a function of two variables, such as:

$$z = \left(\frac{10 - x^2}{x} \right) + \left(\frac{6 - y^2}{y} \right) \quad (4.1)$$

Restrict the values of x and y such that $-10 < x < 10$ and $-6 < y < 6$. To properly encode this, a bit string must be mapped into all possible values of x and y .

A simple way, as depicted in Figure 4.1 would be to use 6 bits for each of x and y such that the first bit represents the sign, and the next 5 a fraction. Then scale by 10 or 6 as appropriate to yield values for x and y . Notice that the choice of bit-length for each number determines its precision.

In general, problems can often be encoded in terms of numbers, each with its own mapping into a bit-string. Each of the bit-strings put end-to-end form the chromosome that represents a particular solution.

4.2.2 Fitness Evaluation

Given a chromosome, the GA needs a means to evaluate its fitness relative to other chromosomes. This is done with a fitness function, or objective function, which returns some number for a given chromosome. The greater the number, the greater the fitness.

In the example of the function of equation 4.1 the objective function would simply decode the chromosome for values x and y , evaluate z , and use that as the fitness.

It is not important that the objective function return a fitness relating directly to the problem being solved, as it does here (in fact it is the answer to the problem!). Rather, the objective function must only differentiate between chromosomes as to which is a better solution.

4.2.3 Reproduction Mechanism

The reproductive mechanism follows three notions. Firstly, there must be selection pressure to ensure that the more fit the individual the better its chance are at participating in reproduction. Secondly, there must be an element of heredity, where characteristics of a parent are passed down to its offspring. Thirdly there must be exploration, so that the fitness of individuals has a chance of increasing over time.

Two parameters are needed by the GA during reproduction. They are both probabilities, one of crossover (p_c) and the other of mutation (p_m), and will be explained in their corresponding sections.

Figure 4.2: Roulette Wheel Selection Method

Selection

Selection is the process by which individuals are chosen for mating. It must be random, yet still favour highly fit chromosomes. A simple sampling technique, commonly referred to as *Roulette Wheel Selection*, works as follows. Imagine all individuals being given a slice of a wheel, the size of which is proportional to their fitness, as depicted in Figure 4.2. A simple spinner is then used when a random individual is needed.

This ensures that, given a population of size N , the expected number of times that an individual i will be chosen in N spins is given by $f_i/\bar{f}$, where f_i is the fitness value of individual i , and $\bar{f}$ is the average fitness value of all individuals. Thus an individual that is twice as fit as the average is expected to be chosen two times during N selections, twice as often as the average individual.

While roulette wheel selection works, it is not perfect. For example, given that each spin is randomly determined, it is possible that a particularly

Figure 4.3: 1-Point Crossover Mechanism

poor individual may be over-selected. More dangerous is that even though highly fit an individual is never selected and is lost forever from the population. Section 4.4.2 will discuss some of the limitations of this strategy, and suggest both better sampling strategies and other completely different alternatives.

Crossover

The crossover operator takes two chromosomes and produces from them two children. With probability $1-p_c$ the children are simply the duplicates of each of their parents. p_c is normally kept fairly high, so that the majority of the time crossover is actually performed. As depicted in 4.3, this is done by randomly choosing a point in the chromosomes, splitting them into two, and

exchanging the corresponding parts of each. This is simple 1-point crossover. Variations will be considered in Section 4.4.2.

Heredity is achieved as the children each contain exact copies of much of the parent's chromosome. Yet at the same time, there is exploration of new chromosome possibilities taking place. The concept is that useful building blocks (called *schema*) from either parent will be combined in the child to form a hybrid, while at the same time new schema are being formed and explored.

Consider the schema (110*10), where the "*" character represents a "don't care". This schema of parent 1 is preserved in child 2. However, due to crossover, child 2 contains the new schema ("101011") which was found in neither parent. If this proves to be a beneficial schema it will be reflected in child 2's fitness value, and tend to carry on into future generations.

Mutation

Due to the fact that the initial population is seeded randomly, and that over time individuals will die off and leave the population, it is possible that certain schema will not exist in the population, nor be able to be created with the crossover operator. This limits the GA in its ability to reach all regions of the function space.

The mutation operator is provided to correct this problem. Unlike crossover, it is purely exploratory. To mutate a chromosome each bit is changed from a 1 to a 0 and vice versa with a probability p_m . p_m is usually very small, so that most of the exploratory work is performed by the crossover operator.

4.2.4 Simple GA Pseudo-code

In summary of the preceding sections, the pseudo-code for a simple GA goes as follows (assuming a fixed maximum number of generations):

```

generate random initial oldpop of size popsize
for gen from 1 to maxgen do
  for i from 1 to  $\frac{\text{popsize}}{2}$  do
    select 2 chromosomes from oldpop for parents
    with probability  $p_c$ 
    crossover parents to produce two children
  else
    copy parents to produce two children
    mutate bits of children with probability  $p_m$ 
    add children to newpop
  endfor
  replace oldpop with newpop
endfor
pick best chromosome from newpop as solution

```

4.3 Schema Theorem

Consider the schema theory as stated in [Gol89]:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m \right] \quad (4.2)$$

where:

H is a particular schema

$m(H, t)$ is the expected number of schema H in generation t

$f(H)$ is the average fitness of the strings representing schema H

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

l is the total length of H (number of 1's, 0's, and *'s)

$o(H)$ the order of H (number of 1's and 0's only)

$\delta(H)$ is the length of H (the distance between the furthest separated 1 and 0)

p_c is the probability of crossover

p_m is the probability of mutation per bit

Thus for a schema H_1 of (*10*011*), $l(H_1)$ is 8, $o(H_1)$ is 5, and $\delta(H_1)$ is 6.

The schema theorem suggests that the number of a particular schema that is short, of low-order, and above-average in fitness value will increase exponentially in successive generations.

When designing a GA, it is important to keep this in mind, particularly when designing the encoding for a problem. The GA will be more successful if related elements of a problem map tightly into local regions of the chromosome, lessening their chance for disturbance by the genetic operators.

4.4 Simple Improvements

The GA as described will work fine for many diverse problems. Unfortunately GAs are very difficult to formally analyze, making definite improvements difficult. What works with a certain type of problem often is detrimental for a different one. Many implementors rely on a large bag of tricks developed over time from one problem to the next, trying all in an attempt to better the optimization for a new problem. Some techniques that require little more in the way of complication are discussed here.

4.4.1 Selection Variations

Many different selection schemes have been experimented with for a variety of problems. Analysis of the effectiveness of various popular methods, including other sampling as well as tournament techniques can be found in [GD91]. As was suggested in 4.2.3, the simple roulette wheel technique is limited in effectiveness.

Two terms are used in describing the effectiveness of sampling—*bias* and *spread*. Bias is defined as the absolute difference between an individual's sampling probability and its expected value ($f_i/\bar{f}$). Spread is a measure of the range of possible values of n_i , where n_i is the number of identical individuals i in a particular generation. The optimal, minimum spread is therefore:

$$\left\lfloor \frac{f_i}{\bar{f}} \right\rfloor \leq n_i \leq \left\lceil \frac{f_i}{\bar{f}} \right\rceil \quad (4.3)$$

For example, if individual 1 has a $f_1/\bar{f}$ of 2.4, then a sampling technique that guaranteed it to be chosen either 2 or 3 times would exhibit the property of minimum spread.

Consider the roulette wheel sampling technique from Section 4.2.3. While it is clear that such sampling has zero bias, which is considered optimal, it does not have minimum spread. There is no mechanism to prevent the over-selection of poor individuals, or the under-selection of good ones.

A better sampling technique with respect to bias and spread, first introduced in [Bak87], is called *Stochastic Universal Sampling* (SUS) or *Baker's Multi-arm Roulette Wheel*. As shown in Figure 4.4, the single arm of the roulette wheel is replaced by N equally spaced arms, where N is the popula-

Figure 4.4: Baker's Multi-arm Roulette Wheel Selection Method

tion size. This is spun giving all N selections at once. This selection scheme guarantees both zero bias and minimum spread.

4.4.2 Crossover Variations

A natural extension of the 1-point crossover of Section 4.2.3 is to consider 2-point crossover, as depicted in Figure 4.5. One motivation for moving to a 2-point crossover would be that single point crossover is limiting in the sort of combinations it can make of chromosomes. 2-point is better able to create new schemas from parents, and is thus more exploratory.

A further extreme called *uniform crossover* is depicted in Figure 4.6. Under uniform crossover the bits of each child are chosen from the corresponding bits of one of their parents, each having a equal chance of contribution. This is depicted by an additional string called a bit mask, with 1's and 0's indicating the choice of parent for the bit in that position.

The price paid for greater exploration is the risk of greater disruption

Figure 4.5: 2-Point Crossover Mechanism

of existing schema, and a lessening of the effects of heredity. Whether this is the case or not depends heavily of the order and length of the schema, and the length of the chromosomes being affected. A discussion of disruption and other effects of 2-point and uniform crossover can be found in [Sys89], and further analysis of the general case of n-point crossover in [SJ91].

4.4.3 Fitness Variations

A common problem in implementing a GA for a particular problem is that of *premature convergence*. This is most commonly caused by the population becoming dominated too quickly by the best individuals. Other chromosomes that have completely died out, while having poorer fitness values, may still have contained useful schema that could have been exploited if given enough

Figure 4.6: Uniform Crossover Mechanism

time.

The most common mechanism to correct premature convergence is some form of *fitness normalization*. Consider the simple population of chromosomes show in Table 4.1. Assuming zero bias and spread, the table suggests that individual 3 could be the only chromosome selected for reproduction, and crossover will no longer have any affect.

To correct this problem, normalization is used. All individuals are ranked in fitness, and their ranking used to set a new fitness value. For example, the most fit might be given a new fitness of 100, with each successively ranked individual assigned a value 10 lower than the previous. Re-examining our example yields Table 4.2, where there is a much more even distribution of fitness values. Further discussion of ensuring diversity in GA

Individual i	Fitness $f(i)$	$f(i)/\bar{f}$
1	20	0.40
2	5	0.10
3	173	3.46
4	2	0.04

Table 4.1: Premature Convergence Example

Individual i	Fitness $f(i)$	Ranking	New Fitness $f_n(i)$	$f_n(i)/\bar{f}_n$
1	20	2	90	1.06
2	5	3	80	0.94
3	173	1	100	1.18
4	2	4	70	0.82

Table 4.2: Fitness Normalization Example

populations with normalization techniques, and other mechanisms, can be found in [Bak88] and [ES91].

Fitness normalization can also be used to solve what could be considered the opposite problem, when the fitness evaluation yields answers that are too similar in value. If all fitness values lie within a small range, the selection process may not be able to distinguish well between the good and the bad. Normalization can be used under these circumstances to provide more differentiation between individuals.

4.4.4 Parallel Fitness Evaluation

One practical limitation of GAs is that for problems of reasonable complexity they require a large amount of computing time. This limits their applicability to problems that require answers quickly.

Conveniently, though, the objective function, which is the most time-consuming element of most GAs, needs to be run over and over on a number of independent individuals. As each run of the objective function is independent on the others, they can be done simultaneously on separate CPUs. This could be a simple network of nodes, or a special-purpose multiple-cpu supercomputer.

Chapter 5

Optimized DSP Code Generation

This chapter presents in detail the problem addressed by this thesis, namely that of the poor performance of DSP compilers. Current DSP programming practice is analyzed, and existing techniques to deal with this problem are discussed. Finally, the reasoning behind the use of a genetic algorithm for DSP code optimization is presented.

5.1 DSP Code Optimization Problem

DSPs are appearing everywhere, from inside kitchen appliances to components of high-speed communication equipment. Due to high volume cost is vitally important, forcing the need for the maximum of performance (speed and memory) with the minimum of hardware. At the same time, this high volume demands the overall advantages of high-level language design—speed and accuracy of development.

For a GPP, designers of today are likely to develop software entirely in

a HLL to take advantage of the greater flexibility, maintainability, and portability that it gives over hand-coded assembly. While in the past performance was only obtained through hand-coding it is now provided by highly optimizing compiler support. Unfortunately, while the field of compilers for GPPs is quite advanced, the performance of the state of the art in DSP targeted compilers is still far from being acceptable for their applications [SCL94b]. The reasons for this appear to be twofold:

1. A mismatch between the underlying assumptions that common high-level languages make about the target architecture, and the actual architecture of the target machine.
2. An inability to deal with confining and conflicting architectural constraints.

In the first case, HLLs designed for general purpose processing assume a uniform memory structure with consecutive addressing, making it difficult to properly handle the multi-memory banks of a DSP. Decisions such as in which memory space to place variables does not exist for GPPs. Pointer manipulation, as seen by the HLL, does not consider the problems that arise from a single memory address existing in multiple memory banks.

The second problem is that of difficult constraints. Today's optimizing compilers are most effective when the target architecture has both a regular structure and orthogonal instruction set [HP90]. That is, compilers do well with many general purpose registers and when different elements of the instructions themselves (operations, addressing modes and registers, data types, etc.) are independent of one another. But in DSPs this is not the case.

They have many special purpose registers for data accumulation and memory fetching. Complicated addressing and looping registers are further added to the mixture. In some cases registers may only be used in certain orders, and sometimes contents may only be valid after pipeline delays.

5.2 Current DSP Programming Practice

In DSP today, designers most often hand-code the time-critical pieces of their applications, and use a HLL such as C for other less critical areas. This approach is supported by studies such as in [SCL94a], in which it was found that the most time and opportunity for parallelism in a variety of DSP applications (e.g. image enhancement, spectral analysis) was to be found in a number of common DSP algorithms contained within the application. The overall control and setup code contained less opportunity for optimization.

Unfortunately, programming the machine directly is both error-prone and time consuming. Hand-coded programs, or program fragments, are not easily portable. Applications are becoming more complex and their components harder to develop. It is awkward to interface the HLL and hand-coded algorithms without the HLL really understanding the machine architecture. The reality is that better techniques are needed to develop applications for DSP.

5.3 Current DSP HLL Optimizing Approaches

Several approaches have been taken in providing a high-level optimized approach to programming DSPs. The most straight-forward is to augment an existing compiler (generally for the C language) to specifically take advantage of a DSP architecture by adding a post-optimizer. [LLC94] describes such an approach for the Texas Instruments TMS320C25 processor. This approach works well enough in finding places to introduce *mac* instructions, and to take advantage of special looping hardware, but does not deal well with memory partitioning or special addressing modes.

Another approach meeting with some success is using a traditional HLL, such as C, with language extensions to support DSP concepts [Pro94, Kre94]. This approach focuses on the first problem stated above in Section 5.1 of a mismatch between current HLLs and DSP architectures. For example, such a language may allow a variable to be declared as being in a particular memory space through a new keyword or compiler pragma, or directly support a new data type such as a modulo memory pointer. Unfortunately, many such language extensions are vendor-specific, meaning that one of the major advantages of HLLs, portability, is lost.

Such language extension schemes rely on the programmer to perform memory partitioning by hand. Some automatic partitioning algorithms have been suggested in [SCL95, SCL96], but only targeting the authors' theoretical DSP machine model.

A completely different approach is that taken by such tools as Ptolemy [PHLB95] or Gabriel [Lee89] which use a graphical approach to code generation. These environments take advantage of the fact that most DSP applic-

ations can be viewed as simple signal propagations through a system. The data flow of the application is pieced together with high-level block diagrams, each containing useful transformation to apply to the input signals. Code for the block diagrams may be implemented as hand-coded assembly, or in a traditional HLL. While this provides abstraction from the machine for the programmer, they are restricted to what blocks are provided. Furthermore, these or new blocks must still be implemented in some fashion.

5.4 A GA Approach to Optimization

None of the above approaches consider the second common problem of designing a compiler for DSP machines, namely that of confining and conflicting constraints, and the difficulties this imposes on a traditional compiler. Neither do they very satisfactorily deal with the problem of memory space partitioning without programmer intervention. In this work the use of a genetic algorithm is considered as an alternative approach to code optimization when the target is a DSP.

Assuming one can map the problem of the choices made when optimizing code to the familiar bit strings of a genetic algorithm, this work was motivated by the following observations:

1. **GAs perform well in the presence of many constraints**

Genetic algorithms have been shown to be very robust, and still able to perform well with problems that are too complex for more conventional optimization schemes. Their multi-point exploratory nature ensures that the entire problem space is efficiently investigated.

2. GAs allow for a more global view of optimization

Using a GA should allow for a more holistic approach to code optimization, as the effect of different passes can be considered at once. If an individual's encoding is made to represent choices made during multiple passes, then the evaluation of this individual's fitness takes into account the choices made during all passes at once. This allows for the exploration of tradeoffs between choices made at each stage of optimization, an ability that is not practical when considering each pass separately.

3. Automatic data partitioning is easy to introduce

Once a GA is involved in the optimizer, it is easy to use some portion of the chromosome for making the memory partition choice of variables for the programmer. This way no extensions need to be made to the target language to support a dual-memory architecture.

4. Portability of a GA driven optimizer should be relatively easy

The optimizing algorithm itself does not change with the introduction of a new set of constraints to represent another target architecture. Furthermore, no language additions are required in order to support automatic memory partitioning. Thus it seems reasonable that a GA driven optimizer should be relatively portable between DSPs so long as the constraints of the machine can be expressed in some form the GA can follow.

5. Fitness evaluation allows for flexibility of optimization concerns

Certain optimizations involve a tradeoff between speed and memory consumption. For example, loop unrolling and function inlining increase speed of execution at the expense of program size. Such decisions are typically left to manual programmer intervention by compile-time switches, as different applications or products may need to favour one over the other. If the fitness evaluation of a chromosome takes both into account separately, then a simple user-supplied weighting favouring speed or memory will cause the GA to automatically prefer the optimizations that reflect the designer's needs.

Observations 1-3 are explored in this thesis. It was hoped that the use of a GA optimizer would prove to generate superior code than the original optimizer, and indeed improvement was achieved for certain types of optimizations. The compiler was also augmented to support parallel dual-memory moves, and by using the GA it can automatically partition variables into appropriate memory spaces.

Chapter 6

Implementation

This chapter describes the implementation of the changes made to the G56K compiler. Examples of what these changes are to be capable of doing are shown. In order to understand some of the specifics, explanation of the implementation of the original compiler is needed, in particular its internal representation for code. Separate sections then detail the work needed for the parallel combiner and GA replacement pass.

6.1 Choice of Tools

This work was performed with the assistance of Northern Telecom Limited (NORTEL), and thus to some extent the choice of target architectures and base compiler was dictated by their needs. NORTEL's interest was with the Motorola 56000 family of digital signal processors, and an appropriate port of the GNU C compiler provided by Motorola called G56K.

The M56K's architecture is very much like that described in Chapter 2.

However, more extensive details of this processor can be found in both [m56] and [Klo86].

G56K, as supplied by Motorola, supports multiply-accumulate, or `mac` instructions fairly well. Furthermore, NORTEL has extended the compiler to recognize a new data type called a `frac`, which indicates to the compiler to use the variable as a fixed-point number. However, G56K makes very poor use of the parallel memory architecture of the M56K. All code generated by the compiler occurs serially. That is, the ability of the G56K to simultaneously operate on data in the ALU and fetch new data from memory is ignored by the compiler. Furthermore, aside from the limited access provided by the L-MEMORY model, the dual-memory architecture of the M56K is unknown to the compiler—either X-MEMORY or Y-MEMORY is targeted, but never both.

6.2 M56k Assembly Syntax

The following sections introduce examples of the syntax used by the M56k's assembler, which may be unfamiliar. Instructions may have 1, 2 or 3 operands, with destination always being listed last. For example, the following is a multiply-accumulate of registers `x1` and `y1` into register `a`:

```
mac x1,y1,a
```

Memory or register moves are performed with the `move` instruction. For example, moving the contents of register `x0` to the X-MEMORY location pointed to by register `r2` would appear as:

```
move x0,x:(r2)
```

Many arithmetic operations, such as `mac`, can have certain data moves

performed in parallel with the ALU operation. In some cases, these instructions can take up to two parallel data moves, subject to certain register use restrictions. Here are examples of an immediate value parallel move with an addition, and the classic FIR filter multiply-accumulate with two memory fetches and auto-incrementing address registers:

```
add x0,a      #10,r1
mac x0,y0,b    x:(x0)+,x0  y:(x4)+,y0
```

Further detail can be found in [m56].

6.3 Goals

The implemented changes to the G56K compiler were twofold. Foremost was the addition of a working GA driven optimizer for the compiler. At the same time the compiler was made more suitable for the M56K by better supporting its parallel dual-memory architecture. Not only would this be directly useful for NORTEL regardless of the GA optimizer, but it provided a greater search space for the GA to investigate.

In both cases, the desire was to keep the changes as transparent for the programmer as possible. Thus a major consideration was to try to require no changes to existing source code in order to use either better dual-memory support or a GA optimizer.

The following two sections show specific examples of the what the two changes made to the compiler should be capable of doing.

6.3.1 Parallel Memory Architecture Usage

First consider the use of only a single memory space, and the possibility of parallel memory moves. The following C code calculates the sum of the contents of three memory locations. Also shown is the results of the original compiler, targeting X-MEMORY.

```

frac* val1ptr;
frac* val2ptr;
frac* val3ptr;
frac total;

total = *val1ptr;
total += *val2ptr;
total += *val3ptr;

```

```

move x:(r1),a
move x:(r4),x0
add x0,a
move x:(r3),x0
add x0,a

```

The code ensuring that `r1`, `r4`, and `r3` point to the appropriate memory locations, plus the placement of data there, is omitted for simplicity. While considering only a single memory space there is the possibility of moving the third `move` instruction in parallel with the preceding `add` as follows:

```

move x:(r1),a
move x:(r4),x0
add x0,a      x:(r3),x0
add x0,a

```

Furthermore, if the compiler were allowed to place the value pointed to by `r4` into the second memory space, the first two `move` instructions could be performed in parallel:

```

move x:(r1),a    y:(r4),x0
add x0,a        x:(r3),x0
add x0,a

```

Such use of the parallel architecture of the M56K is achieved by adding

a new *parallel combine* pass to the G56K compiler. This pass is detailed in Section 6.5.

6.3.2 GA Driven Optimization

By breaking the problem of optimization into distinct steps, each on its own is simple enough to be dealt with well. However, one stage of optimization can conflict with another. For example, consider the following series of M56K instructions:

```
move #1,x0
add x0,a
move x:(r0),x1
```

Strength reduction, and knowledge of the M56K's instruction set, suggest that the first two instructions should be combined in a single *inc* instruction. On the other hand, a move can be performed in parallel with an *add* (but not an *inc*). Thus these three instructions are equivalent to both of the following:

```
inc a           move #1,x0
move x:(r0),x1  add x0,a    x:(r0),x1
```

It might appear at first that the increment version, as it requires neither the use of register *x0* nor the encoding of an immediate value of 1, is the better choice. But this is not always the case. For example, if later code requires the constant value 1 and will generate it in a register anyway, the second version could produce shorter code:

```

inc a
move x:(r0),x1      move #1,x0
                    add x0,a      x:(r0),x1
:
                    :
move #1,x0          ;; use value 1 in x0
;; use value 1 in x0

```

Register assignment is another area which can develop such conflicts.

Consider the following set of instructions:

```

mac x0,y0,a
move x:(r1),x0
move y:(r2),y0

```

Here the register allocator has chosen to use the address registers `r1` and `r2`. This choice will have been based on some heuristic that minimizes the number of register conflicts. Later, when trying to make data moves in parallel, the potential exists for the three instructions to be collapsed into one `mac` with two parallel move's:

```

mac x0,y0,a      x:(r1),x0 y:(r2),y0

```

This turns out to be illegal, as the M56K only allows dual memory accesses when one address register comes from the low bank (`r0-r3`) and the other the high bank (`r4-r7`). The G56K's register allocator is not sophisticated enough to take this into account for all possible cases.¹ As with the previous conflict, the optimization that would ultimately result in better code may in some cases be the choice made by the register allocator and in others that of the parallel combiner.

¹The allocator does try to spread out address register usage by alternately choosing from each bank. This does not guarantee, however, that the choices it makes best support parallel data moves.

Both of these examples illustrate the same problem, namely that each individual optimization pass can only to a limited extent take into account the actions of all preceding and following passes. Without such a division of concerns each pass becomes too complicated to practically implement. The GA approach is to have encoded in the chromosome the choices made at each optimization stage. Then the fitness evaluation of a chromosome take into account how good those choices were from the viewpoint of the entire optimization process. This pass is detailed in Section 6.6.

6.4 G56k C Compiler

This section gives a small amount of detail of the implementation of the G56k C compiler, ported by Motorola from GNU's GCC (v1.37.1). Extensive implementation detail of GCC, and how ports of it are developed, can be found in [Sta92].

While a large C program in itself (about 120 000 lines) much of the implementation detail of G56k is well documented. G56k is organized into a front and back end, with communication between the two done via a LISP-like IR called RTL (Register Transfer Language). In order to promote portability, most all optimizations are done on the RTL itself. However, the distinction between optimization and back-end processing is somewhat unclear, as the optimizer is provided a great amount of knowledge about the machine architecture through the .md file (see Section 6.4.3).

6.4.1 Important G56k Passes

After some initial work preprocessing an entire file, G56k then parses each function separately to completion before going on to the next. Thus all optimizations are done on a function basis, each function being treated with the exclusion of the others. For our purposes, the important passes made on the function under compilation are as follows:

- RTL Generation
- Jump Optimization
- Register Scan
- Common Subexpression Elimination
- Loop Optimization
- Data Flow Analysis
- Instruction Combination
- Register Allocation
- Final Assembly Output

6.4.2 Internal Representation

Internally, a function is represented as a doubly-linked chain of instructions called *insns*, each of which is in turn made up of RTL. An *insn* takes on the following form when viewed as text:

```
(<insn_type> <uid> <prev_uid> <next_uid> (<pattern>)  
  <insn_code> (<log_links>) (<reg_notes>))
```

The meaning of the fields are as follows:

insn_type The type of insn (e.g. “insn”, “jump_insn”, “code_label”).

uid Unique id number of this insn.

prev_uid The uid of the previous insn in the chain.

next_uid The uid of the next insn in the chain.

pattern The RTL code that represents the action of this insn.

insn_code Code number of this insn in the .md file (see Section 6.4.3 below).

log_links Logical links to other insns (its use is discussed more fully in Section 6.6.3).

reg_notes Interesting notes about this insn.

As an example, the following is an insn with uid 21 that assigns the constant value of 1 to register 29. It has not been found in the .md file as of yet, and has no logical links to subsequent insns. It is also to be considered equivalent to the constant value of 1.

```
(insn 21 19 22 (set (reg:SI 29) (const_int 1))
  -1 (nil) (expr_list:REG_EQUIV (const_int 1) (nil)))
```

6.4.3 Machine Description (.md) File

The machine description file (or .md file) provides the mechanism to map insns into actual machine instructions supported by the target architecture. The RTL pass changes the tree structure generated by the parser into a set of “named” insn patterns that must appear in the .md file of all target machines.

Such named patterns provide mechanisms for register and memory moves, basic arithmetic, and control flow instructions for all supported data types.

For example, the following is the description of the 2 operand single integer negation insn targeted for the M56K:

```
( define_insn "negsi2"
  [ ( set ( match_operand:SI 0 "register_operand" "=D" )
    ( neg:SI ( match_operand:SI 1 "register_operand" "0" ))) ]
  ""
  "neg %0" "" )
```

The part surrounded by “[” and “]” characters is called the *pattern* of the insn. The insn must match this pattern in order to be a valid `negsi2` instruction. The `match_operand` construct is a place-holder that describes to the compiler the type of operand that can take this place in the pattern. For example, both operands in the `negsi2` pattern must be single integer (SI) registers (not memory locations).

The “`neg %0`” string is called a *template* and tells the compiler exactly what to output into the assembly file for any insn matching this pattern. In this case, it consists of the string “`neg`”, followed by operand 0. The `%0` is known as a *directive*, and tells the compiler what text to substitute into the assembly file.

In general, the pattern does not have to be a simple string but instead can be an arbitrarily complex C function that returns the template string. This allows for a single insn to have multiple template possibilities. The choice of pattern is dictated by the C code, and typically depends upon the actual values of the operands in the insn.

The code number of an `insn` indicates to the compiler which entry in the `.md` file it has been matched to. Not all `.md` entries are named. The ones that are form a complete enough internal representation so that the RTL generation pass is able to make the initial translation from high-level language to RTL. However, unnamed entries, which better support the target architecture, are also allowed. The combine pass of the compiler is responsible for fitting the best choice of patterns from the `.md` file to the chain of `insns` generated by the RTL pass.

For complete detail about the format of entries in the `.md` file it is necessary to consult the source code of GNU CC. For this discussion it is enough to know the purpose of both the pattern and the template.

6.4.4 N-Register Tracking

One “simplification” made in the G56K compiler is the lack of proper tracking of the N-REGs of the M56K. The architecture is such that offset register `Nx` can only be used with address register `Rx`. Thus the decision of which R-REG is used dictates which N-REG must be used with it.

The approach taken is as follows. Firstly, N-REGs are ignored throughout the entire compiler, until the final `insn` scan that generates the actual output assembly file. During the final scan, as they are scanned sequentially, an `insn` whose template will use an N-REG consults a table to see what value is presently in it. If the value is correct for the operands of the `insn` then a template is chosen that simply uses the N-REG. Otherwise, a template that first loads the correct value is chosen, and the table is updated to reflect this new value. The effects of this approach are twofold:

1. N-REG usage information only exists within the templates of an insn.
2. This information is not valid until final insn scanning is being performed and the specific template for an insn is selected.

The parallel combine pass needs complete information on N-REG usage throughout the entire program before it can run, as it must ensure that any code rearranging it performs is legal. For example, a move instruction that involves an N-REG offset cannot be moved earlier in the code than when the N-REG in question is set to the correct value.

In order to support the new parallel combine pass, more N-REG usage information is needed than originally supplied in the compiler. To collect it, the insn chain is prescanned before the parallel combine pass, which finalizes all template choices for each insn. At the same time information on when there is a read from or write to an N-REG is collected, to be used later, along with all other data-flow information, by the parallel combiner.

6.5 Parallel Combine Pass

The new parallel combine pass is responsible for introducing parallelism into the output code. Whether targeting a single memory space or both X-MEMORY and Y-MEMORY, the issue of parallel execution is only considered after other optimization passes are complete. This way other passes do not need to be changed to account for the parallelism, nor need to be aware of the dual-memory architecture.

The parallel combine pass is enabled by passing the `-mparallel-combine` switch to the G56K program. Dual memory support is enabled with the

`-md-memory` switch. In total it represents an addition of approximately 1700 lines of code to the original compiler.

6.5.1 New `x_memory` Keyword

As a stepping stone to automatic memory partitioning by the compiler itself, a new keyword, `x_memory`, was added to the C compiler. This choice for a keyword is really a misnomer, as it indicates to the parallel combine pass that the variable is to be placed in the alternate memory space to that being targeted, and not necessarily X-MEMORY.² One complication of this is that memory pointers have to be flagged as to which memory space they refer to, and this information propagated through the `insn` code as they are used.

As discussed earlier, this sort of language extension is problematic as the programmer must now make a major performance-affecting decision, that of memory partitioning, by hand. As will be seen, though, partitioning will be performed automatically by the compiler once a GA is involved in the optimization process.

6.5.2 Machine Description Modifications

Each `insn` in the `.md` file is given one of the following designations via the machine specific field:

`NO_PAR` No parallelization is possible with this `insn`

`PAR_ALU` `Ins`n template contains a parallel ALU instruction

²The keyword `x_memory` was used only as a matter of convenience as Motorola had actually added it to the parser already, although it was not used to implement anything.

PAR_MOVE Insn template *may* contain a parallel memory move instruction

To support parallel moves, new %P and %Q directives are defined for insn templates. The templates of a PAR_ALU insn have a %P directive in them to indicate exactly where the PAR_MOVE template is to be joined in order to form the parallel move assembly instruction. For example, the addsi3 insn, a three-operand single-integer addition, could appear as follows:³

```
( define_insn "addsi3"
  [ ( set ( match_operand:SI 0 "register_operand" "" )
    ( plus:SI ( match_operand:SI 1 "register_operand" "" )
      ( match_operand:SI 2 "register_operand" "" ))) ]
{
  if ( REGNO ( operands[2] ) == REGNO ( operands[0] ))
  {
    return "asl    %0 %P";
  }
  else
  {
    return "add    %2,%0 %P";
  }
} "PAR_ALU" )
```

Similarly, all appropriate templates of a PAR_MOVE insn have a %Q directive to indicate the joining point. By finding a legal match of a PAR_MOVE and one or two PAR_MOVE of insns, and using these new directives, an appropriate string can be formed for the assembly file that performs all appropriate operations in parallel.

³The actual addsi3 insn for the M56K is much more complicated, but for simplicity much of the irrelevant detail is omitted here.

6.5.3 PAR_MOVE/PAR_ALU Matching

A simple greedy algorithm is used to find matching PAR_MOVEs for a PAR_ALU insn. The insn chain is scanned from the head, and when a PAR_ALU insn is found it is matched with the farthest forward PAR_MOVE possible, subject to the following:

1. The two insns are in the same basic block.
2. The template of the PAR_MOVE must be verified to have a %Q directive in it.
3. The PAR_MOVE has not been placed in parallel with another PAR_ALU.
4. Moving the PAR_MOVE does not break any data flow restrictions.
5. If dual-memory support is enabled, and this is the second PAR_MOVE to be put in parallel with the PAR_ALU, then the two PAR_MOVEs must be compatible (e.g. they must abide by the R-REG bank restrictions as described in Section 6.3.2, and many other restrictions dictated by the M56k).

6.6 GA Passes

The new GA passes replace a number of existing optimizing passes, yet provide the same types of transformations. Some of the original code is reused, but the GA provides overall control. In total, additions to the compiler to support the GA passes amounted to approximately 3500 lines of code.

6.6.1 Appropriate Passes For GA Control

In order for there to be an advantage in using the GA as opposed to the original pass, there must be some potential advantage of exploration of ordering within the pass itself, plus the possibility of tradeoffs between one pass and another. The following areas were identified as appropriate for GA control:

Insn Combination

Insn combination removes redundant loads, stores, and register to register moves that are generated as a result of a simple translation of the C code to RTL. However, it also provides for algebraic simplification, strength reduction, and exploitation of machine idioms. The original combine pass searches through the insn chain from the beginning, exhaustively trying to combine each insn with all following insns with which it shares some common element. The `.md` file describes those combinations that are legal for the target machine.

In some respects, the combine pass performs as a simple scheduler, because it tends to reorder and group insns in the insn chain. But it does not explore the benefits of different schedules. For example, given the first example of Section 6.3.2, it might always elect to combine the move and add into an `inc`, as most often this will prove the best optimization. By having a GA dictate which groups of insns to combine, better exploration should be possible.

GA controlled combination is enabled by passing the `-muse-ga-combine` switch to G56K.

Register Allocation

Register allocation is most commonly performed by a graph colouring algorithm, and produces good results. However, such an approach cannot practically take into account all the effects its choices have on other optimization passes, such as shown in the second example in Section 6.3.2. By using a GA to consider multiple passes, including register allocation, such exploration can be done without building more knowledge into the allocator itself.

GA controlled register allocation is enabled by passing the `-muse-ga-alloc` switch to G56K.

Memory Partitioning

Strictly speaking, memory partitioning is not a separate pass. The use of the `x_memory` extension keyword allows the programmer to manually place variables into either X-MEMORY or Y-MEMORY as desired, and this is taken into account during the new parallel combine pass. It is a simple matter, though, once a GA is involved in the compilation, to have it make this decision for the programmer automatically.

GA controlled memory partitioning is enabled by passing the `-muse-ga-dual-mem` switch to G56K.

6.6.2 Basic GA Choices

The code used for the GA engine was a simple modification of SGA-C, a C implementation of the genetic algorithm described in [Gol89], originally writ-

ten in Pascal. SGA-C allows for simple parameter changes (population size, crossover and mutation probabilities, etc.), and selection method changes.

This implementation used a SUS sampling scheme based on that developed for [Mar95]. As all encodings (see Section 6.6.3) involve entire bytes, crossover was restricted to byte boundaries. Furthermore, mutation was also performed on a byte basis, not bit. The single-point crossover scheme was also changed to a two-point mechanism. These changes were all suggested from either GA theory or early experimentation.

6.6.3 Encoding

A chromosome is broken into three separate pieces, one for each of the combiner, allocator and partitioner. The encoding and decoding of each part is independent on the others.

Combiner

For each `insn` in the function being compiled, four groups of three bytes (called a “chunk”) are used to encode how it is to be combined with its neighbours. The first two bits of the first byte in each chunk tells the optimizer the `combine_type` for this `insn`, these being:

`NO_COMBINE` Don’t try to combine this `insn`

`DOUBLE_COMBINE` Try to combine this `insn` with one following `insn`

`TRIPLE_COMBINE` Try to combine this `insn` with two following `insns`

Each `insn` contains a linked list of pointers (called `LOG_LINKS`) to subsequent target `insns` in the same basic block that use at least one of the same

operands as it does. `LOG_LINKS` is ordered starting from the nearest target `insn` to the furthest. The original combiner uses the `LOG_LINKS` information to exhaustively try to combine each `insn` with its neighbours. Due to the ordering of `LOG_LINKS`, the order of combine attempts is always fixed. The GA breaks this ordering by using the second (in the case of a `DOUBLE_COMBINE`) and third (in the case of a `TRIPLE_COMBINE`) bytes to pick `insns` from the `LOG_LINKS` list. Each of these two bytes is interpreted as a fractional value between 0 and 1, and is scaled to an integer value from 1 to the length of `LOG_LINKS`. A combine is then attempted with one or both of these and the original `insn`.

Each attempt at a combine may fail if there is no legal way to do so (as determined by the `.md` file). However, there are 4 chunks encoded per `insn`, which allows for multiple combine attempts to be made.

Allocator

G56K assumes an infinite number of “pseudo-registers” when generating its RTL. The problem of register allocation then becomes a matter of mapping these pseudos to real registers. The original G56K approach was to consider local allocation first (pseudos whose lifespans do not exist over multiple basic blocks) followed by global allocation (pseudos whose lifespans do exist over multiple basic blocks), each using a graph colouring approach. Any pseudos that could not be mapped to a real register without some conflict are left unmapped, and a subsequent reload pass performs the necessary memory spilling to relieve the conflicts.

The GA allocator considers local and global allocation together. One

byte is generated in the chromosome for each pseudo register in the function being compiled. The value in this byte determines which real register the pseudo will be assigned to. If this register is inappropriate (of the wrong type or is already in use during the pseudo's lifetime) then the pseudo is left unmapped and the reloader handles it afterwards as in the original allocator.

Partitioner

To allow for the GA to control data partitioning, one byte is generated in the chromosome for each variable local to the function. A single bit of this byte is used to make the choice of between X-MEMORY or Y-MEMORY, and this reflected in the data structure describing that variable to other passes.

6.6.4 Fitness Evaluation

In order to evaluate the fitness of an individual's encoding, that encoding must be applied to the function being compiled. Doing so changes the existing insn chain and other data held in the compiler that represents the function being compiled. As the GA will have to evaluate the fitness of many individuals, the original data must somehow be preserved.

Due to the extremely complicated nature of these data structures, it was found that the easiest way to do the duplication of data was to fork the compiler to have two separate processes running. The child process performs the compilation and evaluates the fitness of the resulting function, and returns that value to the parent via shared memory. The drawback of this approach is that for every individual in a population a system call to `fork()` is needed,

which is computationally expensive.⁴

The child process, with its own copy of the `insn` chain, decodes the chromosome and applies the appropriate transformations to the `insn` chain. Furthermore, all other non-GA controlled optimization passes (such as jump optimization, and the new parallel combiner) were applied to the function before the fitness of the chromosome was determined. This way the chromosome's fitness reflects how well the `insn` combines, register allocations, and memory partition choices worked with other optimization passes.

Calculating Fitness From Insn Cost

No attempt was made to consider memory usage in the fitness evaluation. Rather, the fitness value was based only upon a simple measure of performance related to the length of the code resulting from compilation. Considering the IR of the `insn` chain only, and not the resulting object code, function performance should be inversely proportional to the number of `insns` remaining after optimization. Good `insn` combination tends to eliminate redundant `insns` from the chain, whereas poor register allocation tends to add extra `insns` to represent the loads and stores of memory spills.

Following this analysis, the cost of the entire `insn` function was determined by summing the cost of each `insn`. An `insn`'s cost is determined by the number of machine instructions it produces. For example, the template of the `iorsi3` `insn` (3 operand single-integer inclusive OR) contains 2 M56K instructions, whereas the `iordi3` (3 operand double-integer inclusive OR) template

⁴A better implementation could actually use this sort of approach to its advantage to have fitness evaluation performed by multiple processes at once on multiple workstations, with the original parent acting only to organize the work and collect the results.

generates 7, and their costs would be assigned as 2 and 7 respectively. The cost is not determined until the exact template string for an insn is chosen, as insns can have different template patterns for different data (and hence different costs).

The actual fitness value of an individual was determined by:

$$fitness = \frac{1}{insn\ cost + 1} \quad (6.1)$$

Thus the higher the fitness value the better the individual.

Fitness Ranking

Ultimately, it was determined that some sort of fitness ranking scheme was required in order to prevent premature convergence from occurring. After some initial experimentation, the following was found to produce good results over a variety of DSP functions. Population sizes of 200 were used for all GA experiments, with the top 40 given a ranked fitness value of 200 down to 161. All other individuals were given the ranked fitness value of 160.

Branches and Loops

The simple assigning of weighted costs to each insn for an overall cost does not take into account of the effects of conditional branches in the code. Branches that are taken more often are better choices for optimization. This could be reflected in an individual's fitness evaluation by a greater weighting of the insns along the more frequently taken paths.

In general this behavior cannot be predicted by the compiler as it is data

dependent. One solution would be to have the programmer provide compiler directives to indicate relative branching probabilities given their knowledge of the program's intended behavior. This was not implemented, as it would have meant, in effect, extending the C language.

Looping suffers from similar unpredictable behavior. However, just the presence of a loop indicates an area that, relative to the immediately adjacent code, has at least the potential for more frequent usage. Thus it makes sense to weight the cost of insns inside of loops more heavily than the surrounding code. This could be implemented by scaling the cost of insns inside of loops by their loop depth. However, this remains unimplemented.

Chapter 7

Testing and Results

This chapter describes the testing of the performance of the new compiler. The first section deals strictly with the new parallel combiner. Next memory partitioning is considered, using both the new `x_memory` keyword and also the GA. Finally the GA combiner and allocator are tested individually, and together, on a series of six common DSP algorithms.

7.1 Parallel Combine Test

The first addition to G56K that was tested here was the parallel combiner, without trying to take advantage of the dual memory architecture of the M56K. Two sample DSP algorithms are presented, a windowing function and a simple portion of a FIR filter.

7.1.1 Simple Windowing Function

The following `window1()` function could be used to implement any windowing algorithm when initialized with the proper coefficients. It is intended to show how the parallel combiner can make a significant performance improvement in a tight loop situation. `window1()` uses array subscripting for all its memory accesses, whereas later implementations will use pointers for more efficient data access.

```
/*
   window1() takes a pointer to an array of fracs of size ARRAY_SIZE.

   If "reset" is non-zero, it uses these values to initialize its
   static "coeff" array for future use.

   If "reset" is zero, each value in the given array is scaled by the
   corresponding value stored in the "coeff" array.
*/

void
window1( frac* in, int reset )
{
    int i;
    static frac coeff[ARRAY_SIZE];

    if( reset )
    {
        for( i=0; i<ARRAY_SIZE; i++ )
        {
            coeff[i] = in[i];
        }
    }
    else
    {
        for( i=0; i<ARRAY_SIZE; i++ )
        {
```

```

        in[i] *= coeff[i];
    }
}
}

```

Assuming that `ARRAY_SIZE` is large, it is really the performance of the second for loop that is most important to optimize. The original compiler generates the following four instructions to form this second loop (`ARRAY_SIZE` is 1024):

```

do      #1024,L13
L11      move    y:(r1)+,x0
         move    y:(r2)+,y0
         mpyr    y0,x0,a
         move    a1,y:(r4)+
L13

```

The parallel combiner can do no better with the C code structured as it is. However, the presence of the `mpyr` (multiply with round), which can take parallel moves, suggests that an improvement is possible. In order to force a situation where the `mpyr` has an independent move that can be performed in parallel, the function was rewritten as `window2()`:

```

/*
  window2() is as window1(), but tries to preload array values to
  allow for parallel move possibilities in the loops.
*/

void
window2( frac* in, int reset )
{
    int i;

```

```

static frac coeff[ARRAY_SIZE];
frac* coeff_ptr = coeff;
frac tmp_frac;

if( reset )
{
    tmp_frac = *in++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        *coeff_ptr++ = tmp_frac;
        tmp_frac = *in++;
    }
}
else
{
    tmp_frac = *coeff_ptr++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        *in++ **= tmp_frac;
        tmp_frac = *coeff_ptr++;
    }
}
}

```

The resulting assembly code for the second for loop is as follows:

```

do      #1024,L13
L11     move    y:(r1),x0
        mpyr    x0,x1,a      y:(r4)+,x1
        move    a1,y:(r1)+
L13

```

Thus with a careful rearrangement of the C code and the parallel combiner, a 33% increase in the performance of this loop was obtained.

7.1.2 Simple Filtering Function

This second example to demonstrate the parallel combiner is a single stage of a FIR filter. While such an implementation by itself is not very useful, it is a demonstration of the effect of the parallel combiner with the important `mac` instruction.

```
/*
  filter1() takes a pointer to an array of fracs of size ARRAY_SIZE.

  If "reset" is non-zero, it uses these values to initialize its
  static "coeff" array for future use.

  If "reset" is zero, the sum of the product of corresponding values
  of the "in" and "coeff" arrays is calculated and returned to the
  caller.
*/

frac
filter1( frac* in, int reset )
{
  int i;
  static frac coeff[ARRAY_SIZE];

  if( reset )
  {
    for( i=0; i<ARRAY_SIZE; i++ )
    {
      coeff[i] = in[i];
    }
    return 0;
  }
  else
  {
    frac acc = 0;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
```

```

        acc += in[i] * coeff[i];
    }
    return acc;
}

```

As before, the most important part of the generated assembly code is the implementation of the second loop:

```

        do      #1024,L13
L11      move    y:(x1)+,x0
        move    y:(x4)+,y0
        macr    +x0,y0,a
L13

```

The parallel combiner is no better here than it was with the original code for `window1()`, due to the fact that there are no memory moves available following the `macr` (multiply-accumulate with round) instruction. But similar restructuring of the C code, to produce `filter2()`, yields an improvement:

```

/*
  filter2() is as filter1(), but tries to preload array values to
  allow for parallel move possibilities in the loops.
*/

frac
filter2( frac* in, int reset )
{
    int i;
    static frac coeff[ARRAY_SIZE];
    frac* coeff_ptr = coeff;
    frac tmp_frac1, tmp_frac2;

```

```

if( reset )
{
    tmp_frac1 = *in++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        *coeff_ptr++ = tmp_frac1;
        tmp_frac1 = *in++;
    }
    return 0;
}
else
{
    frac acc = 0;
    tmp_frac1 = *in++;
    tmp_frac2 = *coeff_ptr++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        acc += tmp_frac1 * tmp_frac2;
        tmp_frac1 = *in++;
        tmp_frac2 = *coeff_ptr++;
    }
    return acc;
}
}

```

The resulting loop generated by the compiler is shorter by one instruction, a 50% improvement in its performance:

```

L11      do      #1024,L13
          macr    +x0,y0,a      y:(r4)+,y0
          move    y:(r1)+,x0
L13

```

7.2 Dual Memory Space Test

While the preceding examples did demonstrate parallel data moves, neither took advantage of the dual memory of the M56K. In the filtering example, further improvement is possible if the coefficient array is stored in the second memory space. This can be accomplished with manual programmer intervention by use of the added `x_memory` keyword (shown in Section 7.2.1), or by using the GA to perform automatic data partitioning (shown in Section 7.2.2).

7.2.1 Manual Memory Partitioning

An experienced DSP programmer may be familiar enough with the nature of how their program uses data to be able to make intelligent choices for the compiler when it comes to data partitioning. For instance, in the filtering example, it is clear that if the coefficient array is stored in an alternate memory space, parallel memory fetches should be possible. The function `filter3()` introduces the `x_memory` keyword as a modifier for the `coeff` array declaration:

```
/*
   filter3() is identical to filter2(), but with the "coeff" array
   placed into the alternate memory space.
*/

frac
filter3( frac* in, int reset )
{
    int i;
    x_memory static frac coeff[ARRAY_SIZE];
    frac* coeff_ptr = coeff;
    frac tmp_frac1, tmp_frac2;
```

```

if( reset )
{
    tmp_frac1 = *in++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        *coeff_ptr++ = tmp_frac1;
        tmp_frac1 = *in++;
    }
    return 0;
}
else
{
    frac acc = 0;
    tmp_frac1 = *in++;
    tmp_frac2 = *coeff_ptr++;
    for( i=0; i<ARRAY_SIZE; i++ )
    {
        acc += tmp_frac1 * tmp_frac2;
        tmp_frac1 = *in++;
        tmp_frac2 = *coeff_ptr++;
    }
    return acc;
}
}

```

The resulting multiply-accumulate loop is reduced to a single instruction, and improvement of 300% when compared to the original `filter1()` example:

```

do      #1024,L13
L11      macr    +x0,y0,a      x:(r4)+,y0      y:(r1)+,x0
L13

```

7.2.2 Automatic Memory Partitioning

Because G56K's optimizer works on a function-by-function basis, it can only make memory space choices for variables that are local to the function under compilation. Notice that the implementations of both the windowing and filter functions use a static local variable for the coefficient array. It may be more common practice with C programmers to declare the coefficient array outside of the function, which would prove far less awkward as other functions can then have access to it for their own reasons (e.g. for initialization). However, both functions were written with automatic memory partitioning in mind, and thus use a local static data array for storage.

For the simple `filter2()` function, a trivially short GA run is needed in order to produce the same results as the `filter3()` code, with its manual placement of the `coeff` array into the alternate memory space. What is more interesting is the possibility of using the GA to perform other optimizations at the same time. Testing of the GA combiner and allocator are treated in the following section.

7.3 GA Combiner and Allocator Testing

The preceding example DSP algorithms are somewhat limited in their complexity and real usefulness. Furthermore, they had to be "tweaked" in order to get the best performance out of the parallel combiner. This section, which shows results when using the GA optimizer for instruction combination and register allocation, uses six more complicated and complete C algorithms to test the GA combiner and allocator described in Chapter 6. These algorithms

are as follows:

fir.c Full FIR filter implementation with scaling factor, modified to operate on G56K's fractional data type.

iir.c Sample-by-sample IIR filter implemented by cascaded direct form II second order sections, modified to operate on G56K's fractional data type. Samples are stored in a history array for future filter processing.

dft.c Straightforward DFT implementation, with complex numbers modified to be represented by a pair of G56K's fractional data type.

fft.c In-place radix 2 decimation in time FFT, modified to operate on G56K's fractional data type.

minv.c Matrix inversion with Gauss-Jordan elimination, supporting `int`, `float`, and `double` matrices of arbitrary size.

mdet.c Matrix determinant calculation, supporting `int`, `float`, and `double` matrices of arbitrary size.

The code for each of these algorithms can be found in Appendix A. The intent was to test the GA optimizer on some real code examples to compare its performance to the performance of the original compiler.

The following three sections test the GA driven combiner by itself (with the original allocator), the GA allocator by itself (with the original combiner), and then the combination of the two together, in all cases with the parallel combiner turned on. The resulting assembly code is compared to that of the original compiler with the parallel combiner turned on.

Function Name	Original Size	New Size	New Size as % of Original
fir.c	142	140	98.6%
iir.c	114	108	94.7%
dft.c	278	266	95.7%
fft.c	719	697	96.9%
mdet.c	661	655	99.1%
minv.c	872	865	99.1%

Table 7.1: GA Combiner Results

7.3.1 GA Combiner Results

Table 7.1 shows a comparison of the number of instructions produced by the original compiler and the compiler with the GA driven combiner replacing the normal combine pass of the optimizer. In all cases there is a reduction in total number of instructions, ranging from under 1% for the matrix functions to just over 5% for the iir filter.

These results suggest that, as discussed in Section 6.6, the original compiler limited itself by only allowing exproation of insn combines in a particular order. As it stands, this new combiner does not have any patterns available to it that the original combiner lacks, but merely does matching in a different order. Further improvement may be possible by augmenting the .md file with patterns which were never considered for the original combiner because they would never be used.

Following is an example from `iir.c` of a typical combiner improvement. The original combiner is set up to favour the `mac` instruction in all cases, and introduces it whenever possible. This results in the following code generated by the original compiler plus parallel combiner during `iir.c`:


```

tfr    b,a
mac    -x1,y0,a      x0,y:(r2)+
move   y:(r1)+,y0
tfr    a,b

```

Two transfer register (`tfr`) instructions are needed in order to set up the `mac`, and to get the result into the proper register. In this case the accumulation is used to perform subtraction, not addition (note the minus sign). However, if a straightforward `mpy` and `sub` were used instead, the following code, as generated by the GA combiner, is possible:

```

mpy    y0,x1,a      x0,y:(r2)+
sub    a,b          y:(r1)+,y0

```

No `tfr` instructions are needed, and as an added bonus the additional `sub` instruction can take a parallel data move that before was performed serially. The original combiner will never produce such code as it always tries to generate `mac` instructions. While this preference normally yields better results, there are always exceptions similar to this one.

7.3.2 GA Allocator Results

Table 7.2 gives results of using the GA allocator as a replacement for the `local-alloc` and `global-alloc` passes of the original compiler. Unlike the GA combiner, the GA allocator seems to cause a degradation of performance in almost every case, nearly as high as 20% for `dft.c`.

The reason for this poor performance is the use of the existing reload pass of the original compiler. The `reload()` function, which is responsible

Function Name	Original Size	New Size	New Size as % of Original
fir.c	142	155	109.2%
iir.c	114	132	115.8%
dft.c	278	331	119.1%
fft.c	719	702	97.6%
mdet.c	661	671	101.5%
minv.c	872	876	100.5%

Table 7.2: GA Allocator Results

for satisfying the requirements of any unsuccessfully allocated registers, is called by the GA combiner to clean up any problems. `reload()` prefers a certain set of the M56K's registers as reload registers, into which it attempts to put all pseudo registers that the register allocator could not pair with an available real register. Should these reload registers already be in use, their contents must be spilled. In effect, then, the allocator and reloader compete for registers.

The original allocator is synchronized with the reloader in that it preferences all registers not designated as reload registers. This tends to reduce any conflicts between the two passes. However, the GA allocator has no knowledge of reload registers, and treats all equally. The result can be that when calling upon the reloader to fix up any missing allocations, work that the GA allocator has performed gets undone, and excessive memory spills result.

7.3.3 Overall GA Results

The testing results shown in Table 7.3 include both the GA combiner and allocator working together. Not surprisingly, given the poor performance

Function Name	Original Size	New Size	New Size as % of Original
fir.c	142	155	109.2%
iir.c	114	133	116.7%
dft.c	278	337	121.2%
fft.c	719	724	100.7%
mdet.c	661	702	106.2%
minv.c	872	943	108.1%

Table 7.3: Overall GA Results

of the GA allocator, the overall results are poor in comparison to those of the original compiler. In all cases the results are poorer than with just the GA allocator alone, suggesting that the poor register allocation removes the possibility of the improved GA combiner having any beneficial effect.

Comparing Table 7.2 with Table 7.3, the degradation of the optimizer appear to be greater with the larger functions (`fft.c`, `mdet.c`, and `minv.c`). This is probably due to the relative length of the parts of the chromosome used to encode combine and allocation information. The longer the function being compiled, the longer the chromosome needs to be in order to encode insn combine information. However, the allocator component remains the same length, as it only depends on the number of registers in the target architecture (which remains the same regardless). Thus the larger the function, the more the combine encodings begin to dominate the chromosome. This causes the struggling GA allocator to suffer even more, as most selection pressure comes from improving insn combination, not register allocation.

Chapter 8

Conclusions and Further Work

This chapter summarizes the work performed, presents conclusions, and provides some suggestions for future research. A discussion of the limitations of this approach to compiler optimization is also included.

8.1 Thesis Contributions

In order to examine the possibility of using a genetic algorithm as the optimization mechanism in a DSP compiler, several changes were made to an existing C compiler. G56K targeted for the M56K. To make G56K a more practical high-level language for programming of DSP, support for parallel memory moves were introduced into the compiler. A genetic algorithm, used to control several optimization passes, was also added. The specific contributions are as follows:

8.1.1 Parallel Memory Moves

The original compiler ignored the major ability of the M56K to perform data moves in parallel with arithmetic optimizations. Not taking advantage of this feature means that the compiler makes very inefficient use of the M56K's hardware, and limits its use in implementing any significant real-time DSP algorithm. A new parallel combine optimization pass was introduced which rearranges the object code such that, when possible, data move instructions are combined in parallel with arithmetic instructions to reduce overall execution time.

8.1.2 Manual Memory Partitioning

The architecture of the M56K includes dual memory spaces that can be accessed simultaneously. However, the original compiler only uses both memory spaces for simple stack manipulations. To offset the limited ability of the original compiler to deal with multiple memory spaces, the `x.memory` keyword was added to the syntax of the compiler to allow the programmer to manually designate that variables are to be placed into either X-MEMORY or Y-MEMORY. This, along with the parallel combiner, was shown (see Section 7.2.1) to provide the potential for great performance gains (300%) in common tight-loop situations.

8.1.3 GA Instruction Combiner

The original combine pass of the compiler, which is responsible for reducing the number of instructions required to implement a function by combining

multiple instructions into a single instruction, was replaced with a new combiner driven by a genetic algorithm. The original technique could only attempt to combine instructions in a particular order, and always preferred certain patterns over others when matching.

However, the best ordering and set of patterns changes from function to function, and is difficult to predict beforehand. By using a genetic algorithm to evolve good combine attempts, a consistent improvement in the combine pass was demonstrated (see Section 7.3.1) over a variety of stock DSP algorithms, in some cases resulting in more than a 5% reduction in required code length.

8.1.4 GA Register Allocator

Using the same genetic algorithm routines developed for the combiner, a GA based register allocator was added to the compiler to replace the separate local and global allocation passes. When tested on the same DSP algorithms as the combiner, mixed results were found (see Section 7.3.2). For one algorithm, a 2.4% decrease in code length was obtained. This improvement was the exception, however, and compilation of all other algorithms with the GA allocator resulted in worse than original results, with as high as a 19.1% increase in code length.

The problem with the allocator was identified to be a conflict with a subsequent compiler pass. Further investigation would be needed to correct the problem.

8.1.5 GA Memory Space Partitioner

Using the same genetic algorithm routines developed for the combiner, a GA based memory space partitioner was developed to automatically choose either the X-MEMORY or Y-MEMORY space for variables. This relieves the programmer of having to use the `x_memory` keyword extension to take full advantage of the dual memory of the M56K. As G56K has a function-oriented optimizer, automatic memory space partitioning is limited to variables whose scope does not extend beyond an individual function.

8.2 Thesis Limitations

Adapting the C language to use dual memory spaces, and the use of a genetic algorithm as a compiler's optimizer, proves to introduce certain unique problems, described in the following sections.

8.2.1 Computational Expense

One practical consideration when building a compiler is the time taken for it to generate usable object code. Conventional compiler wisdom says that the rate of compilation should be in the order of thousands of lines per minute.

Genetic algorithms, by their nature, tend to be computationally very expensive. The objective function, and sometimes encoding and decoding processes, tend to be complex operations, and need to be repeated many times over. The particular implementation described here uses an expensive `fork()` system call in the objective function as a means to duplicate the internal representation of the function being compiled. This makes the performance

of the GA very poor. Compilation of functions mere hundreds of lines long took several hours to complete.

Obviously such a compilation rate is unacceptable when developing and testing software. However, if the optimization benefits of the genetic algorithm are considered worthwhile, then using the improved optimizer a single time for an entire project, once the code for that project has been finalized, should be acceptable.

8.2.2 Reproducibility and Randomness

A genetic algorithm is a process driven by random numbers. While successive compilations of the same code will be identical so long as the random number generator seeds are kept the same, any code change at all may completely alter the types of optimizations performed, regardless of consistent seeding.

This becomes an issue when considering program maintenance, as a simple code change in a function will potentially cause the optimization choices for that entire function to be made differently. Applying a patch to existing software is then complicated, as the entire function may need to be replaced. This could cause problems due to lack, or rearrangement, of existing memory space.

8.2.3 Overlapping Memory Addresses

The organization of the second memory space has not yet been discussed. In this implementation, while separate memory space addressing is generated properly by the compiler (the `x:` or `y:` before an direct or indirect memory

access), the compiler does not reuse any address ranges. Thus if one address range is used in X-MEMORY it will be skipped in Y-MEMORY, and vice-versa.

8.3 Appropriateness of the GNU Compiler

Ultimately the G56K compiler did not prove to provide a convenient environment in which to explore the use of GAs in DSP. The original GNU compiler, of which G56K is a port, was written with the intent that it be appropriate for machine architectures the GNU system is to run on. Generally, this means 32-bit single memory space architectures with many general purpose registers. Clearly this is not typical of a DSP, including the M56K.

Most notably, the GNU compiler was set up assuming that the target machine has a memory indirect with immediate offset addressing mode. All accesses to stack variables are generated with a single frame-pointer register and multiple offsets. But the M56K does not allow for immediate offsets, rather it requires that offset values be loaded into an N-REG first. This causes the frame-pointer address register (r0), and its sole offset register (n0), to be overused while other registers are still available.

Hampering this further is the fact that the use of N-REGs is not tracked fully. Not doing so was an unfortunate decision on the part of the original implementors of G56K. This restricts the effectiveness of the any register allocator, whether controlled by a genetic algorithm or not. Also, much time and effort had to go into making up for this missing information when it came to the new parallel combiner.

8.4 Final Conclusions

In general, the reduction in code size provided by using a genetic algorithm in the compiler was not spectacular, especially considering the increase in compilation time. On the other hand, when considering only the results of the GA combiner, there would appear to be value in the adaptive way in which the GA was able to best match the M56K's machine instruction set to the intermediate representation of the compiler.

We feel that this is due to the ability of the genetic algorithm to explore tradeoffs, both inside this particular optimization pass, and also between it and other passes, in a way that the original combine pass was not able. It is our contention that this validates the idea that a more global view of the optimization process in a DSP compiler is worthwhile, and can be provided by a genetic algorithm. A better implementation of the GA register allocator would only further the GA's ability to view optimization from this more global perspective, and allow for very significant optimization improvements.

The introduction of the parallel combiner into the compiler seemed to work fairly well. However, to make best use of it the programmer needs to have some feeling for the underlying architecture to ensure that data is available to be moved in parallel with an arithmetic instruction. This is non-ideal, as it means that the programmer is restricted in the way that the high-level language can efficiently express algorithms.

In conjunction with the parallel combiner, the new `x_memory` keyword as a mean of data partitioning is also non-ideal, as it requires an extension to the C language. This limitation goes away if the genetic algorithm is used for data partitioning. Given, though, that the second memory space is not

properly managed, its use is not really practical at this time.

8.5 Future Work

There is a great deal that could be done to continue the work presented in this thesis. One recommendation would be to speed up the rate of compilation. As mentioned, if the GA optimizer is only used once at the end of software development, then speed is not necessarily important. However, for very large programs, even one GA optimization run might prove impractical. Furthermore, this slow performance makes any new change to the GA optimizer difficult to test as results take so long to produce.

Improving on the efficiency of the objective function by removing the `fork()` call seems an obvious start. The `fork()` was only used as a convenience in order to avoid having to untangle the complicated set of linked-lists and pointer tables used to represent the IR of the function being compiled. While complex, this data could be duplicated without going to the extreme of using `fork()`. Based on profiling, and an estimate of its replacement duplication code, this would allow for a doubling of execution speed. However, for a more significant performance increase it would be worthwhile to consider executing the GA in a multi-processor environment. As each individual's fitness is independent of the others, N processors can compute N fitness costs simultaneously. As profiling of the compiler shows that most time is spent in the calculation of the objective function, a performance increase approaching N times faster execution should be possible, so long as N is large yet less than the population size (which in these experiments was 200).

Another means of improving efficiency would be to store the best chromosome found for each function once it is compiled, so that unless a code change to that function is required it need not be re-optimized during the recompilation of a file.

The implementation of the GA driven register allocator should also be re-examined, and made to work more consistently. With a working GA allocator, more study could then be done into the exploration of tradeoffs between instruction combination and register allocation. Giving each separate optimization pass its own chromosome, instead of concatenating them together, is also suggested. By using multiple chromosomes for each individual the problem of relative lengths causing one part to overshadow the other is eliminated.

The existing objective function is extremely simple, and quite naive in its assumption that shorter code means superior optimization. Better treatment of branches, in particular looping constructs, is needed. Furthermore, the objective function should make more realistic assumptions about the differences in access time of on-chip and off-chip memory. It would also be nice to consider aspects of execution speed and memory consumption tradeoffs.

Proper management of the second memory space, such as means to specify its size and to correct the limitations described in Section 8.2.3 are needed.

Finally the issues introduced by the genetic algorithm in maintenance and in-the-field patching need to be further thought through.

References

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [Bak87] James Edward Baker. “Reducing Bias and Inefficiency in the Selection Algorithm”. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–21, 1987.
- [Bak88] James Edward Baker. “Adaptive Selection Methods for Genetic Algorithms”. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 101–111, 1988.
- [ES91] Larry J. Eshelman and J. David Schaffer. “Preventing Premature Convergence in Genetic Algorithms by Preventing Incest”. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122, 1991.
- [GD91] David E. Goldberg and Kalyanmoy Deb. “A Comparative Analysis of Selection Schemes Used in Genetic Algorithms”. In *Foundations of Genetic Algorithms*, pages 69–91, 1991.

- [Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
- [Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.
- [Hol90] Allen I. Holub. *Compiler Design in C*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.
- [HP90] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1990.
- [Klo86] Kevin L. Kloker. "The Motorola DSP56000 Digital Signal Processor". *IEEE Micro*, pages 29–48, December 1986.
- [Kre94] Benjamin Krepp. "DSP-Oriented Extensions to ANSI C". In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 695–702, October 1994.
- [Lee88] Edward A. Lee. "Programmable DSP Architectures: Part I". *IEEE Acoustics, Speech, and Signal Processing Society Magazine*, October 1988.
- [Lee89] Edward A. Lee. "Programmable DSP Architectures: Part II". *IEEE Acoustics, Speech, and Signal Processing Society Magazine*, January 1989.

- [Lem92] Karen A. Lemone. *Design of Compilers: Techniques of Programming Language Translation*. CRC Press, Inc., Boca Raton, Florida, 1992.
- [LLC94] Wen-Yen Lin, Corinna G. Lee, and Paul Chow. "An Optimizing Compiler for the TMS320C25". In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 689–694, October 1994.
- [m56] *DSP56000 Digital Signal Processor Family Manual*.
- [Mar95] Raul San Martin. *Optimizing Power Consumption, Area, and Delay in Behavioral Synthesis*. PhD thesis, Carleton University, 1995.
- [OS89] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [PHLB95] Jose Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck. "Software Synthesis for DSP Using Ptolemy". *Journal of VLSI Signal Processing*, pages 7–21, January 1995.
- [Pro94] Carla Procaskey. "Improving Compiled DSP Code Through Language Extensions". In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 846–850, October 1994.
- [SCL94a] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. "Application-Driven Design of DSP Architectures and Com-

- ilers". In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 437–440, April 1994.
- [SCL94b] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. "Towards Better DSP Architectures and Compilers". In *Proceedings of the Fifth International Conference on Signal Processing Applications and Technology*, pages 658–664, October 1994.
- [SCL95] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. "Automatic Data Partitioning for HLL DSP Compilers". In *Proceedings of the Sixth International Conference on Signal Processing Applications and Technology*, pages 866–871, October 1995.
- [SCL96] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. "Exploiting Dual Data-Memory Banks in Digital Signal Processors". In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, October 1996.
- [SJ91] William M. Spears and Kenneth A. De Jong. "An Analysis of Multi-Point Crossover". In *Foundations of Genetic Algorithms*, pages 301–315, 1991.
- [SK88] Robert D. Strum and Donald E. Kirk. *First Principles of Discrete Systems and Digital Signal Processing*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [Sta92] Richard M. Stallman. Using and Porting GNU CC. Technical report, Free Software Foundation, 1992.

REFERENCES

106

- [Sys89] Gilbert Syswerda. "Uniform Crossover in Genetic Algorithms".
In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2-9, 1989.

Appendix A

Sample DSP C Code

A.1 fir.c

```
/******
```

```
fir_filter_int - Filters int data array based on passed int coefficients.
```

```
The length of the input and output arrays are equal  
and are allocated by the caller.
```

```
The length of the coefficient array is passed.
```

```
An integer scale factor (passed) is used to divide the accumulation result.
```

```
void fir_filter_int(int *in,int *out,int in_len,  
                    int *coef,int coef_len,int scale)
```

```
    in        frac pointer to input array  
    out       frac pointer to output array  
    in_len    length of input and output arrays  
    coef      frac pointer to coefficient array  
    coef_len  length of coefficient array  
    scale     scale factor to divide after accumulation
```

```
No return value.
```

```

*****/

void fir_filter_int(in,out,in_len,coef,coef_len,scale)
    frac *in,*out,*coef;
    int coef_len,in_len,scale;
{
    int i,j,coef_len2,acc_length;
    frac acc;
    frac *in_ptr,*data_ptr,*coef_start,*coef_ptr,*in_end;

    /* set up for coefficients */
    coef_start = coef;
    coef_len2 = (coef_len + 1)/2;

    /* set up input data pointers */
    in_end = in + in_len - 1;
    in_ptr = in + coef_len2 - 1;

    /* initial value of accumulation length for startup */
    acc_length = coef_len2;

    for(i = 0 ; i < in_len ; i++) {

    /* set up pointer for accumulation */

        data_ptr = in_ptr;
        coef_ptr = coef_start;

    /* do accumulation and write result with scale factor */

        acc = (*coef_ptr++) * (*data_ptr--);
        for(j = 1 ; j < acc_length ; j++)
            acc += (*coef_ptr++) * (*data_ptr--);
        *out++ = (acc/scale);

    /* check for end case */

```

```

        if(in_ptr == in_end) {
            acc_length--;      /* one shorter each time */
            coef_start++;      /* next coefficient each time */
        }

/* if not at end, then check for startup, add to input pointer */

        else {
            if(acc_length < coef_len) acc_length++;
            in_ptr++;
        }
    }
}

```

A.2 iir.c

/* FILTER INFORMATION STRUCTURE FOR FILTER ROUTINES */

```

typedef struct {
    unsigned int length;      /* size of filter */
    frac *history;           /* pointer to history in filter */
    frac *coef;              /* pointer to coefficients of filter */
} FILTER;

```

/******:

iir_filter - Perform IIR filtering sample by sample on floats

Implements cascaded direct form II second order sections.
 Requires FILTER structure for history and coefficients.
 The length in the filter structure specifies the number of sections.
 The size of the history array is 2*iir->length.
 The size of the coefficient array is 4*iir->length + 1 because
 the first coefficient is the overall scale factor for the filter.

Returns one output sample for each input sample. Allocates history array if not previously allocated.

`frac iir_filter(frac input, FILTER *iir)`

`frac input` new float input sample
`FILTER *iir` pointer to FILTER structure

Returns `frac` value giving the current output.

Allocation errors cause an error message and a call to `exit`.

*****/

```

frac iir_filter(input,iir)
    frac input;      /* new input sample */
    FILTER *iir;      /* pointer to FILTER structure */
{
    int i;
    frac *hist1_ptr,*hist2_ptr,*coef_ptr;
    frac output,new_hist,history1,history2;

/* allocate history array if different size than last call */

    if(!iir->history) {
        iir->history = (int *) calloc(2*iir->length,sizeof(frac));
        if(!iir->history) {
            exit(1);
        }
    }

    coef_ptr = iir->coef;          /* coefficient pointer */

    hist1_ptr = iir->history;       /* first history */
    hist2_ptr = hist1_ptr + 1;     /* next history */

    output = input * (*coef_ptr++); /* overall input scale factor */

```

```

for(i = 0 ; i < iir->length ; i++) {
    history1 = *hist1_ptr;          /* history values */
    history2 = *hist2_ptr;

    output = output - history1 * (*coef_ptr++);
    new_hist = output - history2 * (*coef_ptr++);    /* poles */

    output = new_hist + history1 * (*coef_ptr++);
    output = output + history2 * (*coef_ptr++);      /* zeros */

    *hist2_ptr++ = *hist1_ptr;
    *hist1_ptr++ = new_hist;
    hist1_ptr++;
    hist2_ptr++;
}

return(output);
}

```

A.3 dft.c

/* COMPLEX STRUCTURE */

```

typedef struct {
    frac real, imag;
} COMPLEX;

```

/******

dft - Discrete Fourier Transform

This function performs a straight DFT of N points on an array of complex numbers whose first member is pointed to by Datain. The output is placed in an array pointed to by Dataout.

```

void dft(COMPLEX *Datain, COMPLEX *Dataout, int N)
*****/

void dft(Datain, Dataout, N)
    COMPLEX *Datain, *Dataout;
    int N;
{
    int i,k,n,p;
    static int nstore = 0;      /* store N for future use */
    static COMPLEX *cf;         /* coefficient storage */
    COMPLEX *cfptr,*Dinptr;
    double arg;

    /* Create the coefficients if N has changed */

    if(N != nstore) {
        if(nstore != 0) free((char *) cf);    /* free previous */

        cf = (COMPLEX *) calloc(N, sizeof(COMPLEX));
        if (!cf) {
            exit((int)1);
        }

        arg = 8.0*atan(1.0)/N;
        for (i=0 ; i<N ; i++) {
            cf[i].real = (frac)cos(arg*i);
            cf[i].imag = -(frac)sin(arg*i);
        }
    }

    /* Perform the DFT calculation */

    for (k=0 ; k<N ; k++) {
        Dinptr = Datain;
        Dataout->real = Dinptr->real;

```

```

    Dataout->imag = Dinptr->imag;
    Dinptr++;
    for (n=1; n<N; n++) {

        p = (int)((long)n*k % N);
        cfptr = cf + p;      /* pointer to cf modulo N */

        Dataout->real += Dinptr->real * cfptr->real
                        - Dinptr->imag * cfptr->imag;

        Dataout->imag += Dinptr->real * cfptr->imag
                        + Dinptr->imag * cfptr->real;

        Dinptr++;
    }
    Dataout++;      /* next output */
}

```

A.4 fft.c

```

/* COMPLEX STRUCTURE */

typedef struct {
    frac real, imag;
} COMPLEX;

/*****

fft - In-place radix 2 decimation in time FFT

Requires pointer to complex array, x and power of 2 size of FFT, m
(size of FFT = 2**m). Places FFT output on top of input COMPLEX array.

void fft(COMPLEX *x, int m)

```



```

*****/

void fft(x,m)
    COMPLEX *x;
    int m;
{
    static COMPLEX *w;          /* used to store the w complex array */
    static int mstore = 0;      /* stores m for future reference */
    static int n = 1;           /* length of fft stored for future */

    COMPLEX u,temp,tm;
    COMPLEX *xi,*xip,*xj,*wptr;

    int i,j,k,l,le,windex;

    double arg,w_real,w_imag,wrecur_real,wrecur_imag,wtemp_real;

    if(m != mstore) {
/* free previously allocated storage and set new m */

        if(mstore != 0) free(w);
        mstore = m;
        if(m == 0) return;      /* if m=0 then done */

/* n = 2**m = fft length */

        n = 1 << m;
        le = n/2;

/* allocate the storage for w */

        w = (COMPLEX *) calloc(le-1,sizeof(COMPLEX));
        if(!w) {
            exit((int)1);
        }
    }
}

```

```

/* calculate the w values recursively */

    arg = 4.0*atan(1.0)/le;          /* PI/le calculation */
    wrecur_real = w_real = cos(arg);
    wrecur_imag = w_imag = -sin(arg);
    xj = w;
    for (j = 1 ; j < le ; j++) {
        xj->real = (float)wrecur_real;
        xj->imag = (float)wrecur_imag;
        xj++;
        wtemp_real = wrecur_real*w_real - wrecur_imag*w_imag;
        wrecur_imag = wrecur_real*w_imag + wrecur_imag*w_real;
        wrecur_real = wtemp_real;
    }
}

/* start fft */

    le = n;
    windex = 1;
    for (l = 0 ; l < m ; l++) {
        le = le/2;

/* first iteration with no multiplies */

        for(i = 0 ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp_real = xi->real + xip->real;
            temp_imag = xi->imag + xip->imag;
            xip->real = xi->real - xip->real;
            xip->imag = xi->imag - xip->imag;
            *xi = temp;
        }

/* remaining iterations use stored w */

        wptr = w + windex - 1;

```

```

    for (j = 1 ; j < le ; j++) {
        u = *wptr;
        for (i = j ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp.real = xi->real + xip->real;
            temp.imag = xi->imag + xip->imag;
            tm.real = xi->real - xip->real;
            tm.imag = xi->imag - xip->imag;
            xip->real = tm.real*u.real - tm.imag*u.imag;
            xip->imag = tm.real*u.imag + tm.imag*u.real;
            *xi = temp;
        }
        wptr = wptr + windex;
    }
    windex = 2*windex;
}

/* rearrange data by bit reversing */

j = 0;
for (i = 1 ; i < (n-1) ; i++) {
    k = n/2;
    while(k <= j) {
        j = j - k;
        k = k/2;
    }
    j = j + k;
    if (i < j) {
        xi = x + i;
        xj = x + j;
        temp = *xj;
        *xj = *xi;
        *xi = temp;
    }
}
}

```

A.5 minv.c

```
/* structure used by all matrix routines */
```

```
typedef struct {
    int element_size;
    unsigned int rows;
    unsigned int cols;
    char **ptr;
} MATRIX;

#define SCALE_MAT(a,b,s,rows,cols,rowoff,coloff,typea,typeb) { \
    typea **_AMX = (typea **)a->ptr; \
    typeb **_BMX = (typeb **)b->ptr; \
    typea *_PTA; \
    typeb *_PTB; \
    int _IX,_JX; \
    for(_IX = 0 ; _IX < rows ; _IX++) { \
        _PTB = _BMX[_IX]; \
        _PTA = _AMX[_IX + rowoff] + coloff; \
        for(_JX = 0 ; _JX < cols ; _JX++) \
            *_PTB++ = (typeb)(s * (*_PTA++)); \
    } \
}
```

```
/******
```

matrix_invert - Gauss-Jordan elimination with full pivoting.

The returned matrix is always a double matrix. Exits and prints error message with singular matrices or bad MATRIX structures.

```
MATRIX *matrix_invert(MATRIX *A)
```

```
*****/
```

```
MATRIX *matrix_invert(A)
    MATRIX *A;
```

```

{
    MATRIX *matrix_allocate();

    MATRIX *Ai;
    double **a;
    double big,pivot_inverse,temp,abs_element;
    int *pivot_flag,*swap_col,*swap_row;
    int i,n,row,col,swap,irow=0,icol=0;

    /* check for square matrix */
    if(A->rows != A->cols) {
        exit(1);
    }

    /* check pointer */
    if(!A->ptr) {
        exit(1);
    }

    /* size of square matrix */
    n = A->rows;

    /* allocate space for the inverse */
    Ai = matrix_allocate(n,n,sizeof(double));

    /* copy to double matrix */
    switch(A->element_size) {
        case 2:
            SCALE_MAT(A,Ai,1,n,n,0,0,short,double)
            break;
        case 4:
            SCALE_MAT(A,Ai,1,n,n,0,0,float,double)
            break;
        case 8:
            SCALE_MAT(A,Ai,1,n,n,0,0,double,double)
            break;
        default:
            exit(1);
    }
}

```

```

    }

    a = (double **)Ai->ptr;

/* allocate index arrays and set to zero */
    pivot_flag = (int *) calloc(n,sizeof(int));
    swap_row = (int *) calloc(n,sizeof(int));
    swap_col = (int *) calloc(n,sizeof(int));

    if(!pivot_flag || !swap_row || !swap_col) {
        exit(1);
    }

    for(i = 0 ; i < n ; i++) { /* n iterations of pivoting */

/* find the biggest pivot element */
        big = 0.0;
        for(row = 0 ; row < n ; row++) {
            if(!pivot_flag[row]) { /* only unused pivots */
                for(col = 0 ; col < n ; col++) {
                    if(!pivot_flag[col]) {
                        abs_element = fabs(a[row][col]);
                        if(abs_element >= big) {
                            big = abs_element;
                            irow = row;
                            icol = col;
                        }
                    }
                }
            }
        }
        pivot_flag[icol]++; /* mark this pivot as used */

/* swap rows to make this diagonal the biggest absolute pivot */
        if(irow != icol) {
            for(col = 0 ; col < n ; col++) {
                temp = a[irow][col];
                a[irow][col] = a[icol][col];

```

```

        a[icol][col] = temp;
    }
}

/* store what we swaped */
swap_row[i] = irow;
swap_col[i] = icol;

/* bad news if the pivot is zero */
if(a[icol][icol] == 0.0) {
    exit(1);
}

/* divide the row by the pivot */
pivot_inverse = 1.0/a[icol][icol];
a[icol][icol] = 1.0;      /* pivot = 1 to avoid round off */
for(col = 0 ; col < n ; col++)
    a[icol][col] = a[icol][col]*pivot_inverse;

/* fix the other rows by subtracting */
for(row = 0 ; row < n ; row++) {
    if(row != icol) {
        temp = a[row][icol];
        a[row][icol] = 0.0;
        for(col = 0 ; col < n ; col++)
            a[row][col] = a[row][col]-a[icol][col]*temp;
    }
}

/* fix the affect of all the swaps for final answer */
for(swap = n-1 ; swap >= 0 ; swap--) {
    if(swap_row[swap] != swap_col[swap]) {
        for(row = 0 ; row < n ; row++) {
            temp = a[row][swap_row[swap]];
            a[row][swap_row[swap]] = a[row][swap_col[swap]];
            a[row][swap_col[swap]] = temp;
        }
    }
}

```

```

    }
}

/* free up all the index arrays */
free((char *)pivot_flag);
free((char *)swap_row);
free((char *)swap_col);

return(Ai);
}

```

A.6 mdet.c

```

/* structure used by all matrix routines */

typedef struct {
    int element_size;
    unsigned int rows;
    unsigned int cols;
    char **ptr;
} MATRIX;

#define SCALE_MAT(a,b,s,rows,cols,rowoff,coloff,typea,typeb) { \
    typea **_AMX = (typea **)a->ptr; \
    typeb **_BMX = (typeb **)b->ptr; \
    typea *_PTA; \
    typeb *_PTB; \
    int _IX,_JX; \
    for(_IX = 0 ; _IX < rows ; _IX++) { \
        _PTB = _BMX[_IX]; \
        _PTA = _AMX[_IX + rowoff] + coloff; \
        for(_JX = 0 ; _JX < cols ; _JX++) \
            *_PTB++ = (typeb)(s * (*_PTA++)); \
    } \
}

```



```

    }

/*****

matrix_det - Determinant of the input matrix.

Returns a double value equal to the determinant of the matrix.
Allocates and frees a matrix for the determinant calculation.
Exits and prints error message on invalid MATRIX structure.

    double matrix_det(MATRIX *A)

*****/

double matrix_det(A)
    MATRIX *A;
{
    MATRIX *Adet;
    double **a;
    double *a_ptr;
    double det,big,pivot_inverse,temp,abs_element;
    int n,row,col,swap_row,pivot;

    if(A->rows != A->cols) {
        exit(1);
    }

/* check pointer */
    if(!A->ptr) {
    }

/* size of square matrix */
    n = A->rows;

/* allocate space for the determinant calculation matrix */

```

```

    Adet = matrix_allocate(n,n,sizeof(double));

/* copy to double matrix for calculations */
switch(A->element_size) {
    case 2:
        SCALE_MAT(A,Adet,1,n,n,0,0,short,double)
        break;
    case 4:
        SCALE_MAT(A,Adet,1,n,n,0,0,float,double)
        break;
    case 8:
        SCALE_MAT(A,Adet,1,n,n,0,0,double,double)
        break;
    default:
        exit(1);
}

a = (double **)Adet->ptr;

/* initialize the answer */
det = 1.0;

for(pivot = 0 ; pivot < n-1 ; pivot++) {

/* find the biggest absolute pivot */
    big = fabs(a[pivot][pivot]);
    swap_row = 0; /* initialize for no swap */
    for(row = pivot + 1; row < n ; row++) {
        abs_element = fabs(a[row][pivot]);
        if(abs_element > big) {
            swap_row = row;
            big = abs_element;
        }
    }

/* unless swap_row is still zero we must swap two rows */
    if(swap_row != 0) {
        a_ptr = a[pivot];

```

```
        a[pivot] = a[swap_row];
        a[swap_row] = a_ptr;
/* change the sign of determinant because of swap */
        det = -det*a[pivot][pivot];
    }
    else {
/* calculate the determinant by the product of the pivots */
        det = det*a[pivot][pivot];
    }

/* if almost singular matrix, give up now */
    if(fabs(det) < 1.0e-50) return(det);

    pivot_inverse = 1.0/a[pivot][pivot];
    for(col = pivot + 1 ; col < n ; col++) {
        a[pivot][col] = a[pivot][col]*pivot_inverse;
    }

    for(row = pivot + 1 ; row < n ; row++) {
        temp = a[row][pivot];
        for(col = pivot + 1 ; col < n ; col++) {
            a[row][col] = a[row][col] - a[pivot][col]*temp;
        }
    }
}

/* last pivot, no reduction required */
det = det*a[n-1][n-1];

/* free up the calculation matrix */
matrix_free(Adet);

return(det);
}
```