CARNEGIE MELLON

Department of Electrical and Computer Engineering

Architecture Synthesis of High-Performance

Application-Specific Processors

Maurício Breternitz Júnior

April, 1991

CARNEGIE MELLON UNIVERSITY

ARCHITECTURE SYNTHESIS OF HIGH-PERFORMANCE APPLICATION-SPECIFIC PROCESSORS

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

by

MAURÍCIO BRETERNITZ JÚNIOR

Pittsburgh, Pennsylvania
April, 1991

Abstract

A new method to design Application-Specific Processors (ASP) for computation-intensive scientific and/or embedded applications is presented. Target application areas include scientific and engineering programs and mission-oriented signal-processing systems requiring very high numerical computation and memory bandwidths. The application code in conventional HLL such as Fortan or C is the input to the synthesis process. Latest powerful VLSI chips are used as the primitive building blocks for design implementation. The eventual performance of the application-specific processor in executing the application code is the primary goal of the synthesis task. Advanced code scheduling techniques that go beyond basic block boundaries are employed to achieve high performance via exploitation of fine-grain parallelism. The Application-Specific Processor Design (ASPD) method divides the task of designing an special-purpose processor architecture into Specification Optimization (behavioral) and Implementation Optimization (structural) phases. An architectural template resembling a scalable Very Long Instruction Word (VLIW) processor and a suite of compilation tools are used to generate an optimized processor specification. The designer quickly explores various cost versus performance tradeoff points by performing repeated compilation for scaled architectures. The powerful microcode compilation techniques of Percolation Scheduling and Enhanced Pipeline Scheduling extract and enhance parallelism in the application object code to generate highly parallelized code, which serves as the optimized specification for the architecture. Further performance/efficiency enhancement is obtained in Implementation Optimization by tailoring the implementation template to the execution requirements of the optimized processor specification. A scalable implementation template constrains the implementation style. Graph-coloring algorithms that exploit special graph characteristics are used to minimize the amount of hardware to support execution of the optimized application microcode without impairing code performance. Compilation techniques to allocate data over multiple memory banks are used to enhance concurrent access. The entire architecture synthesis procedure has been implemented and applied to numerous examples. Speedups in the range of 2.6 to 7.7 over contemporary RISC processors have been obtained. The computation times needed for the synthesis of these examples are on the order of a few seconds.

Dedication

To him who is able to keep you from falling and to present you before his glorious presence without fault and with great joy — to the only God our Saviour be glory, majesty, power and authority, through Jesus Christ our Lord, before all ages now and forevermore. Amem. [Jude 24:25]

Ora, àquele que é poderoso para vos guardar de tropeços e para vos apresentar com exultação, imaculados diante da sua glória, ao único Deus, nosso Salvador, mediante Jesus Cristo, Senhor nosso, glória, majestade, império e soberania, antes de todas as eras e agora, e por todos os séculos. Amem. [Judas 24:25]

Acknowledgements

First, I would like to thank my advisor, John Paul Shen, for his leadership during this thesis. I benefitted and learned from John's ability to help clarify rough ideas, identify key conceptual issues and express them in concise text. John's enthusiasm and open-minded researh approach, along with his clear example of Christian living challenged and taught me a number of things about life and myself.

I would also like to thank professors Daniel P. Siewiorek, Donald E. Thomas and Alex Nicolau for serving in my thesis committee. Dan and Don are credited for their leadership in the area of digital systems systhesis. Alex invented Percolation Scheduling, a fine-grained compilation technique which is a key element in this thesis. Alex's accessibility and willingness to answer questions helped smooth the way. Roni Potasman and Haigeng Wang worked on our VLIW compiler project as participants of the Percolation Scheduling group at U.C.Irvine under Alex's leadership.

Andy Wolfe, Chriss Stephens and Ron Bianchini were key members of the White Dwarf project. Chriss also helped with the VLIW compiler graphic interface, originally developed by Dan Nydick. Chris Holt and Greg Palmer created the graph coloring software package.

During my work on the VLIW project at the IBM T.J.Watson lab, I interacted with Dave George, Mickey Tsao and Kemal Ebcioglu. Dave's project leadership helped me come onboard and Mickey helped deal with IBM internals. I learned a lot while interacting with Kemal during our development of the VLIW compiler prototype.

My dear friend Jim McInerney deserves more than a paragraph, perhaps a chapter. Jim and I agree to disagree on must subjects, which makes our weekly Saturday breakfasts even more enjoyable. This helped me come to a better appreciation and understanding of this great country along with its many shortcomings. Jim has been a great *companheiro* during the inevitable ups and downs of life during Ph.D. research.

The friendship of Alberto and Noemia Elfes, Eleri Cardozo, Hudson and Susana Ribas, Haroldo and Otilia Costa Lima and Edna and Marco "Gubi" Gubitoso of the Pittsburgh brazilian community provided much needed home base support. I'll remember with *saudades* all those years of shared life and times together.

Finally, thanks to my wife Martha Rubio Breternitz for almost everything else. Martha temporarily forsaked her career in architecture so that I could perform this work. This research would not have been completed without Martha's patient and loving support. I thank her for being there during the hard times and for sharing with joy the good times. The arrival of our daughter Raquel helped me understand how much more can love be extended without thinning.

This work was supported in part by CNPQ of Brazil and in part by IBM.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Background . . . 4
- 1.3 Architecture Synthesis Approach . . . 7
- 1.4 Thesis Overview . . . 9

2 Architecture Synthesis 10
- 2.1 White Dwarf Experimental Project . . . 10
  - 2.1.1 FEM Application . . . 11
  - 2.1.2 White Dwarf Design process . . . 12
  - 2.1.3 White Dwarf Processor Architecture . . . 13
  - 2.1.4 White Dwarf Performance Evaluation . . . 17
- 2.2 Architectural and Implementation Models . . . 18
  - 2.2.1 The VLIW Architecture . . . 19
  - 2.2.2 Implementation Architectural Template . . . 23
- 2.3 Application-Specific Processor Design (ASPD) . . . 25

3 Specification Optimization 29
- 3.1 Specification Optimization Procedure . . . 29
- 3.2 Intra-Iteration Techniques . . . 32
  - 3.2.1 Percolation Scheduling Core Transformations . . . 34
  - 3.2.2 Percolation Scheduling with Register Renaming . . . 37
  - 3.2.3 Computation of Unifiable-ops . . . 39
  - 3.2.4 Memory-Reference Disambiguation . . . 41
  - 3.2.5 Detection of Induction Variables via Symbolic Substitution . . . 43
- 3.3 Inter-Iteration Techniques . . . 45
  - 3.3.1 Enhanced Software Pipelining . . . 46
  - 3.3.2 Removing Extraneous Copy Operations from Pipelining . . . 49

4 Retargetable Optimizing Compiler 54
- 4.1 Fine-Grain/Microcode Compilation Task . . . 54
- 4.2 Compilation Framework . . . 55
- 4.3 Graphics Interface . . . 59
- 4.4 Code Parallelization Techniques . . . 64
- 4.5 Data Structures . . . 67

4.6 Compilation Effectiveness 70
4.7 Compiler Status and Future Direction 72

5 Implementation Optimization **74**
5.1 Implementation Optimization Procedure 74
5.2 Register Files Allocation 76
5.2.1 Graph Coloring Algorithm 78
5.2.2 Performance of ADJUST-GRAPH 86
5.2.3 Discussion 90

6 Memory Organization for Concurrent Access **92**
6.1 Parallel Memory Access Techniques 92
6.2 Bank Allocation by Graph Coloring 95
6.3 Concurrent Access of Array Elements 96
6.4 Implementation and Application 102
6.5 Examples 105
6.6 Extensions 109

7 Example Applications of ASPD Method **112**
7.1 Synthesis, Validation and Evaluation Procedure 112
7.2 Detailed Example: MIN (Livermore Kernel 24) 113
7.3 Detailed Example: Vector Scaling 117
7.4 Synthesis Benchmark: Elliptic Filter 118
7.5 Synthesis Benchmark: GCD 123
7.6 Linear-Phase B-Spline Filter Example 127
7.7 White Dwarf FEM Solver Example 127
7.8 RISC Instruction Set Processor Example 135
7.9 FFT Processor Example 138

8 Summary, Conclusions and Extensions **142**
8.1 Thesis Summary 142
8.2 Key Contributions 143
8.3 Future Work 145

A FEM Program Source Code **147**
A.1 BackSubstitution Routine 147
A.2 VLIW code for BackSubstitution Routine 149

List of Figures

2.1 White Dwarf System Organization 14
2.2 White Dwarf Data Path Organization 14
2.3 White Dwarf Floating-Point Unit 15
2.4 White Dwarf Integer Unit 16
2.5 White Dwarf Control Path 17
2.6 Wide Instruction Word Architectural Template 22
2.7 Implementation Template 24
2.8 Detail of Sparse Interconnect 25
2.9 Application-Specific Processor Design Method. 26

3.1 Move-op Example 36
3.2 Move-cj Example 37
3.3 Example of Move-op with Register Renaming 38
3.4 Induction Variable Detection in Loop Body With Conditional 45
3.5 Software Pipelining Example 46
3.6 Enhanced Software Pipelining Algorithm 47
3.7 Avoidance of COPY operations Introduced by Pipelining 53

4.1 Compilation Flow 55
4.2 Example: min.c source code and intermediate code 56
4.3 Structure of Control Flow in VLIW Parallelizer 57
4.4 Graphics Interface - Program and Instruction Windows for min.c 60
4.5 Graphics Interface - min.c after first Move-op 61
4.6 min.c after Percolation Scheduling intra-iteration optimization 62
4.7 min.c after Software Pipelining inter-iteration optimization 64
4.8 Data Structures - Instruction and Operation Nodes 69
4.9 Data Structures - Example Instruction 71
4.10 Compilation Effectiveness: Speedup, Resources, Compilation Time 72

5.1 Definition reaching multiple uses 80
5.2 Multiple possible definitions reaching an use 81
5.3 $K_{1,3}$ graph and non-claw-free graph 83
5.4 Claw Graph Resulting from ADJUST-GRAPH 85
5.5 Linear-Phase filter experiment: Register Files Needed 86
5.6 Example of straight-line program that generates an odd cycle in G+ 89

6.1 Memory Subsystem with Multiple Memory Banks. 93
6.2 An Example Interference Graph 95
6.3 Modulated Skewed Distribution Example - MSD(3,4). 97
6.4 Values of K' for which MSD(K,M) is not valid for PAS(i,i+K'). 101
6.5 FFT Algorithm 107
6.6 FFT Algorithm With Code to Compute Memory Bank Number 111
6.7 FFT Inner Loop Code With Memory Bank Control 111

7.1 min.c VLIW code with Reaching Definitions and Register File Allocation . . 115
7.2 Example: min.c Interference Graph 116
7.3 Example: min.c - ASP Design 117
7.4 Vector Scaling Example: C Source Code and NADDR Code 118
7.5 Vector Scaling: VLIW Code With Register File Allocation 118
7.6 Vector Scaling Example - ASP Design 119
7.7 Vector Scaling: VLIW Code With MSD(2,2) Data Allocation for Concurrent Access 120
7.8 Example: Fifth Order Ellipt Filter - NADDR code 121
7.9 Example: Fifth Order Elliptic Filter VLIW code - Part 1 122
7.10 Example: Fifth Order Elliptic Filter VLIW code - Part 2 123
7.11 Example: Fifth Order Elliptic Filter - Interference Graph 124
7.12 Example: Fifth Order Elliptic Filter - ASP Design 125
7.13 Elliptic Filter ASP Design - Register File Allocation 125
7.14 Example: Greatest Common Divisor - C source code and NADDR Code . . 127
7.15 Example: Greatest Common Divisor - Interference Graph 128
7.16 Example: Greatest Common Divisor - VLIW code and Register File Allocation 129
7.17 Example: Greatest Common Divisor - ASP Design 129
7.18 Linear-Phase B-Spline Filter Experiment 130
7.19 Linear-Phase B-Spline Filter Algorithm 130
7.20 Example: Linear-Phase B-Spline Filter VLIW code 131
7.21 Example: Linear-Phase B-Spline Filter 132
7.22 Linear-Phase B-Spline Filter - Register File Allocation 132
7.23 Finite Element Matrix Solver Computation 133
7.24 Back Substitution experiment - simulation results 134
7.25 BackSubstitution Example - Widest VLIW instruction 136
7.26 Simple RISC Processor Specification 138
7.27 Intra-iteration Optimized RISC Specification: 4 cycles/instruction. 138
7.28 Optimized RISC Specification: 1 cycle/RISC instr 139
7.29 Example: FFT Inner Loop Code With Memory Bank Control 141

Chapter 1

Introduction

This thesis presents an automated approach, called *architecture synthesis*, to design application-specific processors (ASP) that are performance efficient and cost effective. Applications of interest include problem-specific algorithms in scientific and engineering computation, and mission-oriented/embedded signal and image processing systems. Architecture synthesis can be viewed as a specialized form of high-level digital system synthesis[56], and integrates and leverages recent research results from other research areas, namely application-specific integrated circuit (ASIC) CAD, fine-grain parallel architectures, and optimizing microcode compilers.

1.1 Motivation

Current approaches to design special-purpose or application-specific systems involve the use of either off-the-shelf general-purpose processors (GPP) or custom-designed special-purpose processors (SPP). The GPP-based approach selects an available GPP that is the best suited to the application, and programs the GPP to carry out the application algorithm. This approach minimizes the system development cost and time, and involves little or no custom hardware design. Such systems are easy to use and can be readily adapted to other applications. Typical strategies to achieve high-performance in a general-purpose computing system involve the use of:

- technology, i.e. faster devices and higher levels of integration
- parallelism in the system architecture to allow concurrent execution of tasks

- algorithm selection and improvement.

However, the GPP-based approach cannot meet the performance requirements of many computation-intensive applications, and/or frequently involves very inefficient utilization of available resources. For example, a recent high-performance microprocessor with fine-grained parallelism, the Intel i860, is rated at 80MFLOPS. However, performance close to peak is unfrequently achieved on specific tasks. In a digital-signal processing task, processor utilization ranges between 10% to 20% of the processor's peak performance[73] due to architectural mismatch with the application and difficulties in compilation. Furthermore, the typical strategies for achieving high-performance are gradually becoming more tedious and costly. Device cycle times are slowly reaching a technology limit, and the programming and run-time control of systems with many processors are extremely difficult tasks[45].

For those applications for which existing GPPs do not provide acceptable performance or performance/cost, SPPs are developed which exploit characteristics of the application to enhance the performance for given applications. The SPP-based approach involves custom architecture and hardware design, and results in well-balanced and highly-efficient architectures. However this is usually a costly and time-consuming process, and system reusability and flexibility are usually quite limited.

An alternative approach is to take advantage of more application-specific information and develop processors that are adapted to the needs of the particular task at hand, having better synergy between hardware and software. High performance is a requirement of many scientific and engineering computation, and mission-oriented/embedded signal and image processing systems. Many application areas require high-performance special-purpose processors[58], including hardware accelerators for computation-intensive algorithms, hard-wired processors for signal processing, coprocessors for scientific computation, and embedded processors for many mission-oriented systems. Special-purpose processors have been designed to solve problems in areas such as numerical simulation, digital signal processing, image processing[74], speech understanding, and engineering design using finite-element analysis[52].

The need for high-performance processors can be cost-effectively satisfied by ASPs. These systems are prime candidates for performance enhancement by a hardware coprocessor ded-

icated to execute the task at hand. The approach described in this work provides powerful tools which generate efficient special-purpose systems by combining the use of high-performance components with the use of fine-grained parallelism in the hardware. This allows the system designer to concentrate his efforts on the development of efficient algorithms.

Furthermore, it is not uncommon that the primary use of a general-purpose scientific engineering workstation is to run a single compute-intensive application, e.g. finite-element analysis or circuit simulation[91], not to mention the computational requirements of high-quality graphics. The current trend towards standard binary interfaces and standardized busses make feasible the inclusion of hardware accelerator cards with software packages. This trend, already noticeable in the personal computer world[75], may become increasingly common in case of reduction of the cost of designing an ASP.

This work is intends to fill a gap in the range of cost vs. performance alternatives from low-cost general purpose processors that have limited processing power, e.g. the Motorola MC68000 family, to full custom SPP design. It aims at providing the performance of special-purpose processors at costs comparable with those of the general-purpose processors.

Cost-effectiveness is achieved by the use of off-the-shelf parts that are manufactured in high volume. High-performance VLSI building blocks are becoming available. These off-the-shelf building blocks include: fast array multipliers, floating-point processors, integer processors, multi-ported register files and other data path functional modules [28, 86]. These fully-custom VLSI chips can function as standard building blocks, or macros, in a semi-custom design approach. What is needed is a semi-custom design and implementation methodology for architectural-level design of high-performance application-specific computing systems. Automated tools reduce both the cost of the processor design phase, e.g: CAD software, and the cost of using the special-purpose processors, e.g. smart compilers and Operating Systems. High performance is achieved via a processor structure customized to exploit the fine-grained parallelism inherent to the target application.

Automated software tools to support the proposed semi-custom design and implementation approach are presented. Results from a number of active research areas are leveraged for the development of such software tools. These research areas include the computer-

aided design of fully-custom and semi-custom VLSI circuits, design of optimizing compilers for microcode optimization[37, 59, 62] and automated synthesis of general-purpose processors[29, 80, 3]. With the richness of research results in these related areas, the development of software tools to support the automated design of ASPs is quite promising. This thesis presents a semi-custom design methodology, associated techniques and software tools for architectural synthesis of application-specific processors.

1.2 Background

Behavioral synthesis tools, also known as structural design tools, are concerned with the task of transforming the primitives of the algorithmic specification (behavior) of the system to the primitives of structural representation[56]. The result of structural design is a detailed description of an implementation in the form of a register-transfer level structure. Structural design tools are similar in spirit with architecture synthesis, and constitute a very active field of research[29, 83, 82, 17, 42, 56]. A number of relevant past and ongoing structural design projects are briefly reviewed.

The pionering work of Barbacci[12] in the EXPL system used a behavioral specification in ISP and a register-transfer module set for the target implementation of the design. This system proposed a number of heuristics to exploit parallelism through transformation on a graph representation of the design. Interestingly, this early project included means for automated exploration of the design space, a task which has been relegated to the user by many of the later tools.

The CMU-DA project[29] produced a number of datapath generators. EMUCS[80] uses a heuristic approach to design the datapath. The procedure takes as input a dataflow graph generated from an ISP description of the target application. The dataflow graph is called Value Trace[80], and is similar to dataflow graphs commonly used by programming language compilers with some extensions to describe control sequencing. EMUCS considers each operation in the graph in succession, binding graph elements to hardware and generating appropriate interconnect or creating new functional elements according to table-driven cost estimates. The control step allocator assigns operations to control steps before hardware binding.

The DAA project takes a knowledge-based approach[49]. A number of patterns commonly encountered in datapath design are encoded in DAA's knowledge base. The synthesis tasks are performed by a number of temporally ordered subtasks. It uses ASAP scheduling to generate a parallel design. Example DAA designs were judged of adequate quality by expert human designers.

Facet[84] takes an algorithmic approach based on clique-partitioning. Facet uses the notion of compatibility among elements like registers or operations in the dataflow representation of the target application. A graph is built with vertices corresponding to elements in the dataflow graph. Vertices corresponding to compatible elements are linked by an edge. A clique partitioning heuristic is used to find cliques of compatible elements in the graph. Each clique may then be bound to a hardware structure. For example, during register allocation, variables whose lifetimes do not overlap are found to be compatible and may be allocated into the same physical register. The same notion is employed to allocate operations to datapath operators, and to schedule the use of busses in the datapath. Facet allocates hardware elements after scheduling has been performed. Scheduling of basic blocks is performed using a simple ASAP scheduling technique. *Basic blocks* are sequences of code which contain no branches except for the last instruction in the block, i.e. code with a single entry and a single exit point. The Facet implementation allows user intervention to modify the code sequence for a basic block, thus allowing manual exploration of the design space.

The System's Architect Workbench(SAW)[27] converts a behavioral description of a piece of hardware into a set of register-transfer components plus a control sequence table. It supports two methodologies to design: a general approach using design algorithms that support designs of many styles, and an approach tuned to microprocessor design. A number of behavioral transformations are performed on the Value-Trace representation of the design, such as procedure inlining, recombination of multiway jumps (SELECT), motion of operations in/out of SELECT branches. These transformations allow exploration of system-level design alternatives, enhance fine-grained parallelism and achieve some inter-basic-block optimization. Automated partitioning of large designs is also supported. SAW implements combined scheduling, allocation and mapping via force-directed scheduling[22]. SAW's advanced hardware allocation techniques[78] under cost constraints is of particular relevance

to this work.

The IBM synthesis project[17] is one of the few systems capable of scheduling parallel operations which come from different basic blocks. In the IBM system, all operations in the dataflow graph for a loop body are initially scheduled to be performed in a single clock cycle. This violates the assumption that a register may only be written once per cycle. Repeated passes are then made to break paths in the loop into different states, thus removing violations of the single-write restriction. This is a powerful technique which has generated good results, and has been capable of producing a design for a 32-bit RISC CPU with quality comparable to that of manual design. However, this system does not have the capability of automatically generating pipelined designs; this feature is left to the user. Furthermore, in a worst case it may perform complete enumeration of all paths in the loop body, and thus may be of exponential complexity. This technique is applied only to innermost loops. This approach is useful in generating instruction set processors where the number of distinct alternatives (instructions) is not too large, but becomes too costly if the input is an arbitrary program loop including nested loops. More recently, the IBM system has been extended with *path-based scheduling*[18]. This approach considers the possible sequences of operations in a control-flow graph, stressing optimization across conditional branches. Enumeration of all paths in a control-flow graph may require time that grows exponentially with the number of nodes in the graph. Experimental results have found that computation times for small to medium-sized problems are still manageable.

The Flamel[82] system is also capable of scheduling concurrent operations from multiple basic blocks by using extended techniques for tree height reduction[2]. Limited software pipelining effects are achievable in Flamel only through loop unrolling. Other behavioral synthesis projects include MIMOLA[90], Chippe[14], ADAM[3, 66], Bridge[21], and Become[71].

Most foregoing synthesis approaches perform scheduling optimization only within basic block boundaries. Scheduling optimization dictates performance, because it determines the number of clock cycles required for the execution of the application code. Scheduling limited to basic blocks is not capable of achieving high-performance through exploitation of parallelism because of the limited parallelism in basic blocks. Early experiments[39], later confirmed by [64], have found that the speedup achievable by exploiting parallelism in basic

blocks is bounded by a factor of three. The architecture synthesis method presented in this thesis achieved high performance by performing scheduling optimization beyond basic block boundaries.

Recent progress in compilers that extract fine-grain parallelism has addressed the problem of compilation beyond basic blocks. Trace Scheduling[37] (TS) is a technique capable of extracting parallelism beyond the basic block boundaries. TS relies on identifying the most probable execution paths in the application code, and on optimizing the execution of these paths. Each path may span a number of basic blocks. Code is generated by scheduling concurrently the execution of the operations on each path. Scheduling constraints and additional code for correction are necessary in case the path is not followed in its entirety during the execution of the application. Experiments with TS in scientific FORTRAN programs have found speedups of up to one or possibly two orders of magnitude. However, the effectiveness of trace scheduling is limited to those applications with a predictable flow of control. Furthermore, TS entails extra overhead in the case the predicted execution paths are not taken. Percolation Scheduling[62] (PS), a technique which evolved from trace scheduling, overcomes these limitations and is able to subsume trace scheduling. PS allows the concurrent execution of operations which belong to different paths in the execution of the application, thus benefiting a class of applications not amenable to traditional techniques of concurrency extraction. Software Pipelining[30, 5, 54] (SP) is a technique to speed up execution of loops by initiating the execution of future iterations while the current iteration is still in progress. Modern PS techniques are applicable to loops which contain conditional statements in the loop body. The combination of both PS and SP techniques is a key factor in achieving considerable optimization of programs with unpredictable flow of control. Speedups on the range of five to tenfold on a VLIW architecture with 16 ALUs have been observed for large realistic programs, such as some UNIX utilities and parts of the SPEC benchmark suite[59], thus overcoming basic block limitations on available parallelism.

1.3 Architecture Synthesis Approach

The approach proposed in this thesis addresses the architecture synthesis problem by exploiting recent advances in compilation techniques and leveraging latest VLSIC and hardware

technology. Attributes which characterize this approach are as follows:

- **Targeted Application.**

  1. The effort focuses on the synthesis of special-purpose processors for specialized application areas, namely scientific and engineering computations, and mission-oriented signal-processing systems. General purpose functionality is sacrificed for performance and efficiency.
  2. These applications frequently require very high numerical computation and memory bandwidths, e.g. multiple hundreds of MFLOPS and comparable I/O rates for each processor. The targeted performance range is beyond that of most single, or small number of, off-the-shelf general purpose processors.

- **Design Style.**

  1. The behavioral description, i.e. input to the synthesis process, is the actual application code written in a conventional HLL such as Fortran or C. The description serves as the architecture specification as well as the application source code.
  2. Latest powerful VLSI chips are assumed as the primitive building blocks for design implementation. Examples include 32-bit integer processors, 64-bit floating-point processors, multi-ported register files, and 32-bit complex address generators. Using the large-grain building blocks alleviates some of the optimization complexity and produces realistic and practical designs.
  3. The approach addresses the design of the complete processor including the datapath, control path, and the data memory, with regards to achieving a balanced processor architecture. Advanced techniques to perform parallel memory access from a multi-banked memory subsystem are crucial in achieving high memory bandwidth and in turn high computation rate.

- **Design Optimization.**

  1. The eventual performance of the application-specific processor in executing the application code is the primary goal of the synthesis task. The secondary consideration is the efficient utilization of hardware resources.

2. Advanced code scheduling techniques that go beyond basic block boundaries are employed to achieve high performance via exploitation of fine-grain parallelism. The scheduling algorithms are rigorous and automated.

3. The optimized code produced by the compiler serves as the optimized specification for the architecture. Second level optimization is performed, without using arbitrary and ad-hoc metrics, to ensure efficient utilization of hardware resources.

1.4 Thesis Overview

This dissertation presents a new approach for architecture synthesis. Chapter 2 describes the architecture synthesis method, which divides the design task into Specification Optimization and Implementation Optimization phases. Appropriate architectural and implementation templates for supporting semi-custom design optimization are introduced. Chapter 3 describes the techniques for specification optimization. Advanced microcode scheduling techniques that go beyond basic blocks, their characterization for use in architecture synthesis and extensions are presented. Chapter 4 describes a retargetable microcode compiler that implements specification optimization techniques, accepts high-level C source code and produces optimized VLIW code for a range of target architectures. Chapter 5 describes implementation optimization via the use of graph-coloring techniques, and special characteristics of conflict graphs to allocate data over low-cost distributed register files. Memory bandwidth between CPU and the memory sybsystem is a key factor to achieving overall system performance. Chapter 6 describes compilation and memory allocation techniques used to enhance memory bandwidth in a memory organization with multiple banks. Chapter 7 demonstrates the application of the architecture synthesis method to a number of examples. The thesis is summarized in Chapter 8 along with conclusions and directions for further research.

Chapter 2

Architecture Synthesis

The basic tenet of the ASPD approach is to exploit characteristics of the application programs in the design of highly-efficient special-purpose processors. This research starts with the formulation of a preliminary design method. The key features identified in the preliminary method are the use of fine-grain parallelism to obtain high performance, the use of powerful VLSI components for efficient and low-cost design, and the provision of connectivity between components to streamline data transfers. A trial implementation of the preliminary method ensued via the hand-design of an application-specific processor[87]. The selected target application is finite-element analysis. Finite-element algorithms are widely used, and are extremely compute-intensive; hence, constitute ideal candidates for hardware/firmware acceleration.

Experience with the finite-element project led to refinement of the preliminary method. This includes the identification of a good architectural template, with provision for suitable memory and I/O bandwidths. Development of techniques and tools evolved from experience in the development and implementation of compilation techniques for the IBM VLIW architecture[31]. This helped identify desirable characteristics of compilation techniques for performance optimization, architectural models suitable for scalability, and performance bottlenecks.

2.1 White Dwarf Experimental Project

This section documents an experimental project, called the White Dwarf[87], which explores the feasibility of the architecture synthesis approach. The results and experiences from this

project served as the basis of and the model for the architecture synthesis method presented herein, called Application-Specific Processor Design (ASPD). The White Dwarf ASP is a multiple board, single user coprocessor for a SUN 3/160C workstation. It is designed, using the architecture synthesis approach, to accelerate a particular finite element analysis application[91] which performs two dimensional magnetic field analysis using the Incomplete Choleski preconditioned Conjugate Gradient (ICCG) method. The application program requires a large number of floating-point operations, is heavily used and well understood, and uses a unique sparse matrix data format.

2.1.1 FEM Application

The Finite Element Method is a powerful numerical procedure for the simulation of a wide range of engineering problems. It is widely used in engineering and scientific applications, extremely computation intensive and the existing algorithms are mature and stable. FEM seeks a numerical solution to a given problem by subdividing the problem space into a mesh of similar geometric shapes called finite elements. By selecting simple, easily analyzed elements, the governing equations for the system are approximated over each finite element. The analogous descriptions are used to approximate the equations over a large number of elements by imposing boundary conditions.

Finite Element discretization creates a large set of simultaneous equations to be solved. These equations are expressed as a single large matrix equation of the form $S\tilde{x} = \tilde{y}$ where the matrix S and the vector $\tilde{y}$ are known, and S is sparse. The FEM system at CMU is used for design and simulation of a wide range of two dimensional magnetic and electronic devices[91]. The user describes problems via an interactive graphics interface. A partitioning algorithm creates a mesh of triangular elements to cover the design, increasing the density of elements in areas where more detail and accuracy are needed. The program then builds a system of equations based on the geometry of the elements and the known boundary conditions of the problem. This system of equations is solved using the Incomplete Choleski preconditioned Conjugate Gradient (ICCG) algorithm[9]. The most time consuming part of the procedure is the solution of the system of equations. Runtimes on a dedicated workstation range from minutes to solve simple problems to hours for larger and more detailed problems. The ASP

processor replaces this algorithm in the FEM code, and is transparent to the user. It is called the White Dwarf and is composed of two VME boards in a Sun workstation.

2.1.2 White Dwarf Design process

The White Dwarf design process is dictated by the commercial environment. Performance is considered the primary criterion for evaluating design decisions, however in cases where a significant cost saving could be realized only with a slight loss in performance, cost considerations are allowed to determine design decisions. The application algorithm is well defined before the design procedure begins, and it is possible to estimate the performance of a proposed processor design with reasonable precision. The control flow of the finite element algorithm is seldom affected by the run time values of data items. The distribution of sparse data in the matrices is well known. Therefore, the primary effect of data on problem solving time is the number of iterations through the main loop. It is possible to estimate overall system performance with reasonable accuracy by estimating the time required to execute the main loop on a proposed design. Cost evaluations are less precise. Estimates are made of the number of parts involved in each board. As design modifications are made, it is necessary to reassess physical resources such as board space and power consumption. Any decision that increases the system cost from original estimate is required to enhance performance by a similar percentage.

Using the conventional intuitive design approach, the problem is analyzed with the assumption that no resources are available. As each operation is analyzed, it is first determined whether the operation can possibly be completed with the existing resources. If not, then additional resources must be added to the architecture. If the operation can be completed, then the designer must determine whether or not the addition of resources will improve performance and allow the operation to complete faster. This approach forces the designer to justify cost/performance benefits of each addition to the design. A potential failing of this constructive method is that it is difficult to establish that all of the potential parallelism has been located. This procedure is likely to repeatedly identify trouble spots in the application code, indicating the possible benefit of restructuring the problem at the algorithmic level. Furthermore, the granularity of design changes may influence the process. For example,

there are situations where the addition of a single resource, e.g. an adder or multiplier, does not improve performance but the combined effect of the addition of a number of such resources provides a substantial performance boost[76]. The combinatorial nature of the kind of search required to cover these cases may prove too time consuming and tedious for a human, or even automated, designer.

Given the above considerations, the alternative design approach adopted in ASPD assumes a machine with a large number of resources and unconstrained connectivity between resources. The entire application is analyzed; maximum parallelism is extracted and operations are assigned to resources as needed to achieve maximum computation throughput. Once the analysis is complete, all unused resources, either operators or communication paths, are eliminated. The remaining architecture identifies a starting point which may exceed performance, as well as cost, requirements. By gradually constraining the architectural parallelism, additional scheduling techniques are employed to remap the algorithm to a further constrained architecture.

2.1.3 White Dwarf Processor Architecture

The White Dwarf processor, illustrated in Figure 2.1, is partitioned into a number of boards connected by a dedicated high-speed bus called Dwarfbus. The CPU board contains all the data path logic, microcode memory and timing control unit. The system board contains the interface to the VME bus and the logic to download microcode and data memories. The Dwarfbus connects these boards to the memory subsystem. The memory subsystem is expandable with up to eight memory boards, but a basic White Dwarf system may contain only one memory board.

The data path is composed of separate integer and floating-point units. The integer unit can read and write pointer data from three integer memory banks. These banks correspond respectively to the row, column and link fields of an element in the data structure that represents the sparse matrix. The floating-point unit can access three floating-point data banks, which store the floating point elements of the ICCG algorithm data structures. Figure 2.2 illustrates the structure of the White Dwarf data path.

The floating-point unit, illustrated in Figure 2.3, has two AMD Am29325 32-bit floating-

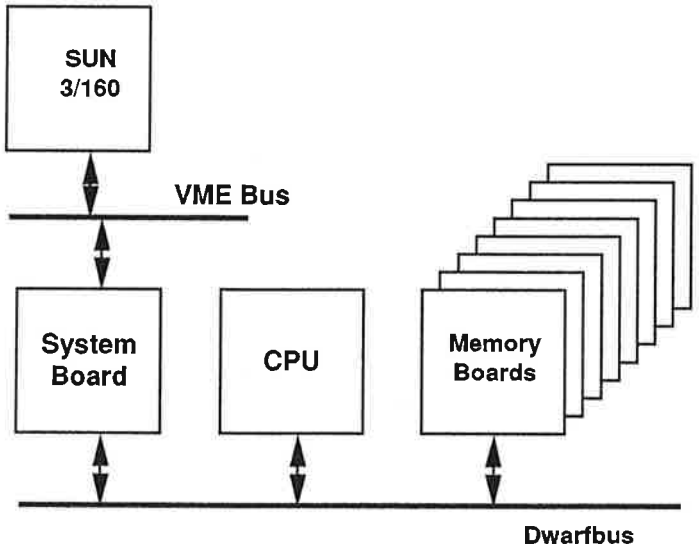


Figure 2.1: White Dwarf System Organization

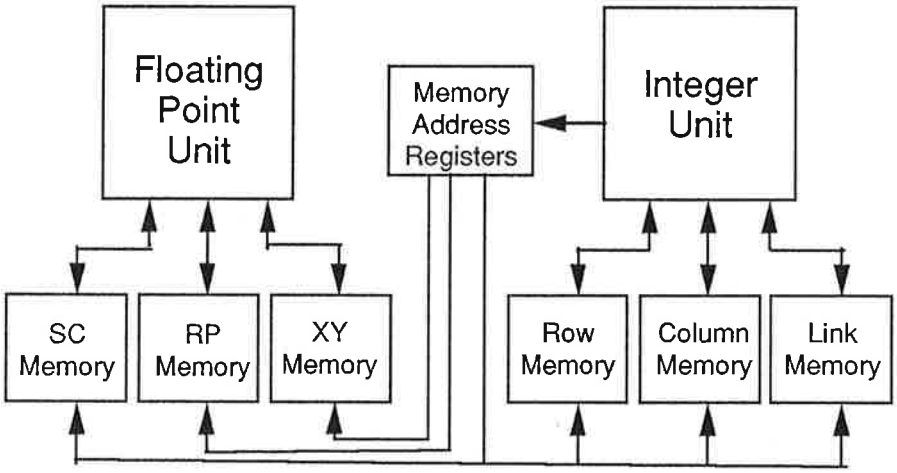


Figure 2.2: White Dwarf Data Path Organization

point ALUs. Each device executes addition, multiplication or subtraction in a single cycle. In fully clocked mode, each operation is completed in 100ns. The two floating-point ALUs have a sparsely interconnected system of 4-ported register files. Each register file supports two reads and two writes per cycle. Each floating point unit has a register file for temporary data and a memory buffer, also implemented by a register file. ALU results are always written

into the same register in both register files. Alternatively, the ALUs can take their inputs from the memory buffers. The memory buffers serve as memory data registers, receiving data from memory and storing it until the next cycle. Results are transferred between the two ALUs by writing them back into both register files. Data arrives from memory on three separate data busses. The implicit connectivity of the memory buffers is used to route data from the busses to the ALU input ports. A 2x3 crossbar switch routes data returning to memory. Each ALU generates a result per cycle. The two results can be routed to any of the three floating-point data busses to be stored in memory. This interconnection structure allows all FEM procedures to operate effectively and is much cheaper and faster than full connectivity, e.g. a full crossbar. Division, a relatively rare operation in the FEM algorithm, is available in only one of the ALUs. It is implemented using the Newton-Raphson iterative method.

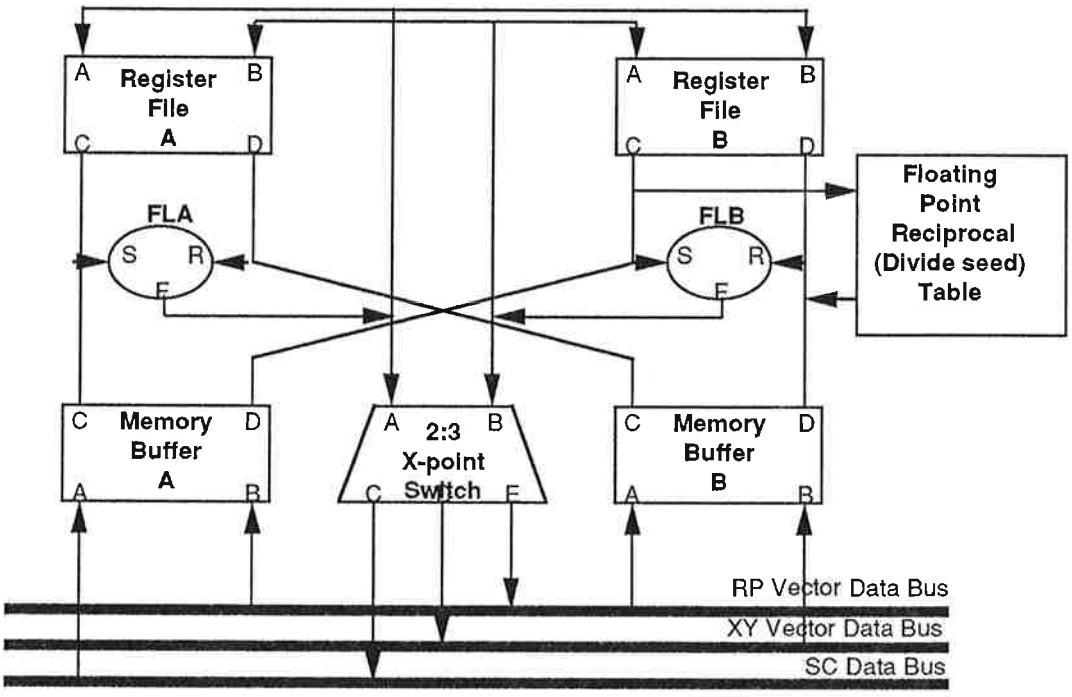


Figure 2.3: White Dwarf Floating-Point Unit

The integer unit, illustrated in Figure 2.4, generates addresses for data in memory, and is used to test for terminating conditions in matrix operations. The integer unit has a single AMD Am29332 32-bit integer ALU. It has a pair of register files which serve as both general purpose register and memory buffers. A complete register to register operation is executed

in one cycle. The equality tester is used to monitor matrix addresses in order to detect diagonals or ends of rows and columns.

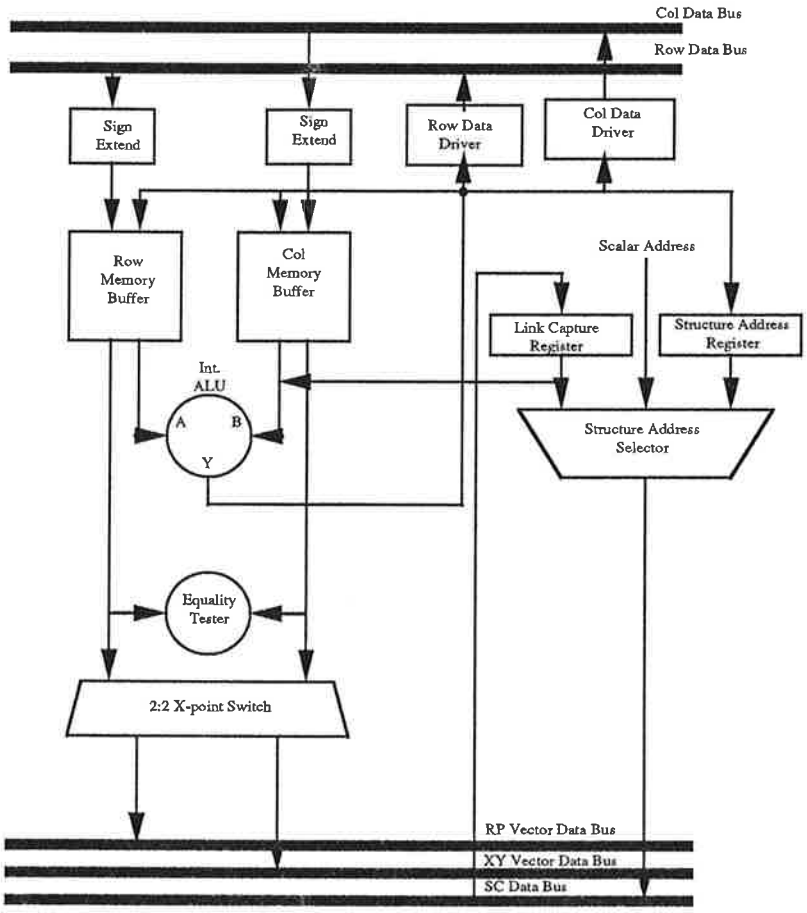


Figure 2.4: White Dwarf Integer Unit

The White Dwarf processor employs a wide instruction word. All of the control fields for the ALU's register files, data path routing, memory control and microsequencing are contained explicitly in each microinstruction word. The finite element algorithm takes a relatively small number of microinstructions. Thus, chip cost is not a factor in the design of the control memory. Figure 2.5 illustrates the control path. The Am29331 microsequencer provides control flow features, conditional branching, nested subroutine calls and loop counters. Each microinstruction contains a microsequencer parcel.

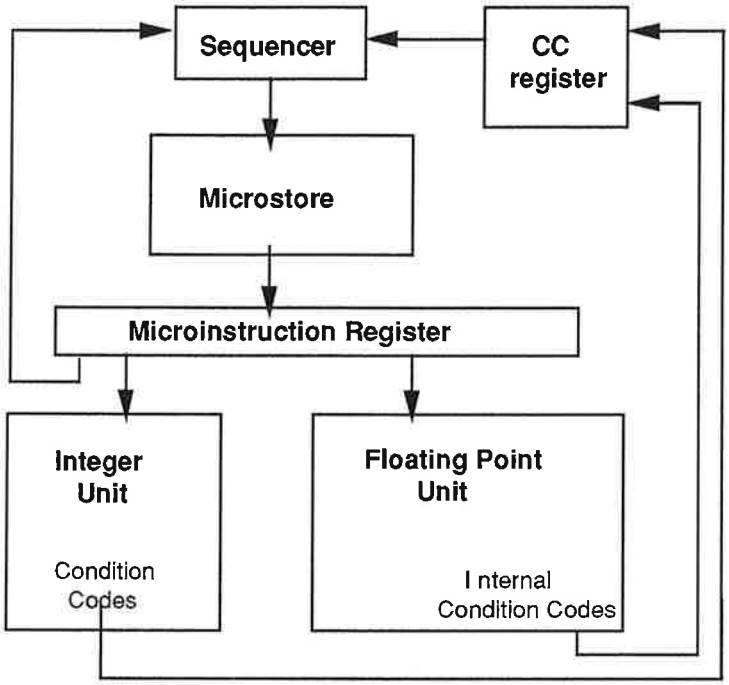


Figure 2.5: White Dwarf Control Path

2.1.4 White Dwarf Performance Evaluation

The White Dwarf prototype system has been designed to operate with a 112ns cycle time. The two floating-point unit can perform two floating-point operations per cycles, therefore the White Dwarf can reach a peak processing rate of 17.8 MFLOPS. Several of the finite element procedures, such as the vector inner product, operate at close to peak speed. The four sparse matrix procedures that form the bulk of the processing operate with 90% utilization of the floating-point unit. Some vector triad operations require reading and writing into the same memory bank per iteration, and thus reduce ALU utilization to 29%. These routines comprise a small portion of the computational load. Overall, an average rate of 80% utilization is maintained, giving the system a useful sustained throughput in the 15 MFLOPS range. For comparison, the execution of simple gather operations on sparse matrices has been reported at sustained throughput of 5 to 11MFLOPS on the Cray-1S[15]. The CMU WARP systolic array has achieved sustained performance of 12.5 MFLOPS[85]. Compared to the Sun 3/160 host, the White Dwarf executes the compute-intensive portion about 40 times faster.

The White Dwarf experience has validated the use of a wide-instruction word architecture and the effectiveness of the synchronous execution model for the efficient use of compile-time extracted parallelism. A key point in the White Dwarf project involves the importance of having adequate memory bandwidth to keep multiple functional units busy. Such bandwidth is achievable with memory organization with multiple banks. Limited connectivity among functional units provides the required data transfers at low cost and fast cycle time.

2.2 Architectural and Implementation Models

The architectural template is further elucidated in this section; specification optimization and architecture generation techniques are presented and illustrated in subsequent chapters. The architectural template is a highly scalable processor architecture model that can be customized according to the specific requirements of the application code. The emphasis of the customization process is to achieve high performance with efficient use of hardware.

The key architectural issues of parallelism, interconnection, and the use of specialized components were identified from experience with the White Dwarf project. Parallelism in the architecture enables high performance by the concurrent execution of the operations required to execute the program. Furthermore, the interconnection structure among the components of a high performance architecture must be adapted to perform quickly the data transfers required by the target application program. Ideally, the communication paths (e.g. busses and multiplexers) between functional units that exchange data must be as simple and direct as possible. Full connectivity, such as that provided by cross-bar switches, is simple and direct but entails a high cost both in hardware requirements and longer cycle times.

As seen above, one of the keys to high-performance is the use of fine-grained, or instruction-level, parallelism. The architecture is capable of concurrent execution of several ALU operations by having multiple functional units. However, the existence and use of parallelism in the architecture introduces the problem of synchronization, the time correlation of related activities. The coordination of operation execution and data transfers must be such that all operations and data transfers are executed and completed in timely fashion according to the data dependencies in the application algorithm.

The extraction of parallelism may be performed either at runtime, by mechanisms to

issue operations concurrently such as scoreboarding, dispatch stacks and dataflow engines, or at compile time. At run time all data dependencies are explicit, and there is no ambiguity between indirect references. However, the extraction of parallelism at run time incurs potentially significant overheads in terms of increased cycle time and cost of hardware for scheduling of operations and interlocking mechanisms to ensure dependency preservation. The alternative approach is compile-time code parallelization. This approach eliminates runtime overheads by performing the scheduling work at compile time. This yields simpler, and potentially faster and cheaper, machines. Furthermore, this approach can exploit parallelism that is not readily available at coarser levels of granularity, and is far too costly to be expressed explicitly at the user application level. The compiler performs extensive analysis of the program to achieve efficient hardware utilization. Such an analysis is not always feasible, or economically viable, at runtime. For example, for a given point in the program, the compiler may use global information about the code characteristics and control flow of computations forthcoming from that point. This "view of the future" is usually not available to purely hardware-based mechanisms.

Code parallelization at compile time is in line with the ASPD philosophy because, by using extensive code analysis, the compiler produces efficient code schedules that make uniform use of hardware resources. The goal is to find code schedules to keep the available functional units busy most of the time. This allows efficient exploitation of the available parallelism by a hardware mechanism that is tailored to and efficiently used by the application program.

2.2.1 The VLIW Architecture

Modern VLIW machines[36, 46, 31] combine the advantages of RISC architectures with the speed benefits of parallel machines. The canonical VLIW model has a load/store, register-to-register instruction set, with unit execution time and uniform and conflict free data-access for all operations. In this model, n operations are issued synchronously to n ALUs that operate on a shared data file and complete execution concurrently in one cycle. Upon completion of one instruction, the next instruction issues and the process repeats. One, or usually more, of the operations in the instruction can combine to dynamically determine the next instruction to execute i.e, conditional jumps. While multiple conditionals may occur in one instruction,

they always combine, in more or less general fashion depending on the implementation, to form a single multiway jump. Thus a characteristic of VLIW machines is that there is a single thread of flow of control from instruction to instruction, even though each instruction may have a multitude of potential successors.

The synchronous execution model, the uniform data-access, combined with the ability to execute multiple and different machine language level operations in every cycle make the VLIW model effective in exploiting very fine grain parallelism not available to traditional MIMD/SIMD multiprocessors, due to the absence of synchronization, communication and runtime scheduling overheads. While the uniformity of the model would prevent arbitrary scalability in a real-world implementation (notably due to the difficulty to provide uniform and conflict-free access time for a large number of processors), the model has still been shown as viable for implementation on a medium scale (tens of processors) notably in the Multiflow [36] and IBM VLIW [31] machines. Of the two, the IBM machine is the one closest to the "pure" VLIW model. It can be thought of as executing *program graphs*, one node at a time. Each node in the program graph corresponds to a VLIW instruction, and contains a *tree* formed from RISC like operations with control-flow operations defining the tree, and non-control-flow operations such as multiply, load, etc, populating the branches.

The execution of a VLIW instruction can be thought of as a three-steps process. In the first step, all operands for all operations in the instruction are read. In the second step, all conditions are evaluated and a path to the unique successor instruction is chosen. This yields the next VLIW instruction to execute. In step three, results of the operations on the path chosen in step two are written into their destinations and control is transferred to the next instruction which is then executed. Note that, as an intentional side-effect of this three-step process, operations in a given instruction can only read the results computed by a previous instruction, since all reads occur before any writes take place. If several operations on the chosen path in the instruction have the same destination, the last operation on the path takes precedence i.e. gets to write to the location. In practice, on the IBM VLIW machine, these three steps are efficiently executed as part of the basic machine cycle. Note that all operations in one instruction will have completed before the next instruction is issued. State-of-the-art technology is used to avoid undully lengthening the cycle time when

performing traditionally costly operations e.g. floating point. The cycle time of the IBM VLIW machine will vary with the precise technology used, but is comparable with that of state-of-the-art RISC processors implemented in the same technology.

The chosen architectural template is a scalable VLIW-like architecture with conditional execution of operations[31]. The VLIW architecture, inspired by the IBM VLIW project and illustrated in Figure 2.6, is characterized by the following attributes:

1. Single Instruction Stream with Application Software implemented directly in wide instructions.

2. N functional units and M memory banks supported by a global register file, and all controlled by a wide instruction.

3. Conditional execution is supported by the functional units; explicit conditional branching is carried out by the sequencer.

The actual values for N and M are scaled to match the optimized application code, Possible decomposition of the global register file into multiple distributed register files with sparse interconnection is applied to match the requirements of the optimized code. The VLIW architecture is an effective template for specialized scientific processors because it effectively achieves high performance by exploiting parallelism, as exemplified by the use of VLIW machines for scientific computation[36, 81, 31], and because of its flexibility for scaling and customization. Furthermore, it exploits fine-grain parallelism at the level of individual ALU (micro)operations via compile-time synchronization. This kind of parallelism is more commonly present in applications than the coarse-grain parallelism that is efficiently used by other architectural models such as MIMD and SIMD. Existing multiprocessors cannot effectively exploit fine-grain parallelism because the relative cost of synchronization would easily overcome the benefits of parallelism. Even a few cycles of communication and synchronization overhead are much longer than the execution time of a single operation. SIMD architectures, while not having the synchronization overhead, are only applicable to a certain class of tasks in which a number of operations is applied regularly over large sets of data.

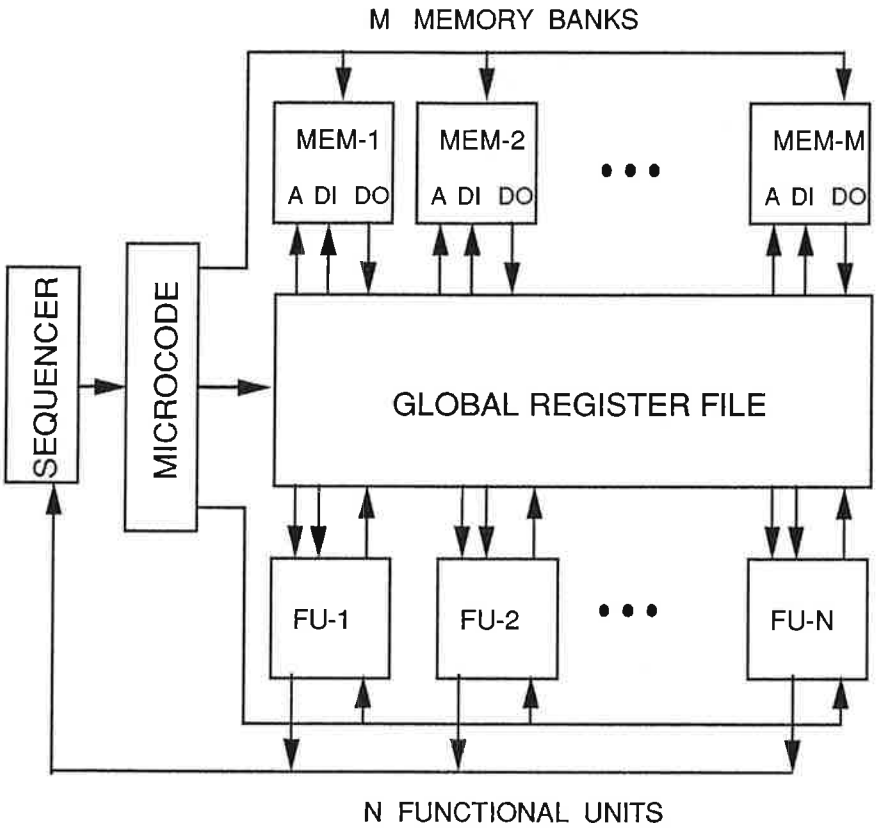


Figure 2.6: Wide Instruction Word Architectural Template

The use of conditional execution of operations in the VLIW architecture provides additional performance enhancement by reducing the performance penalty involved in executing conditional statements. In a conditional-execution model, condition code bits in the processor determine whether the result of an operation performed conditionally is stored into the register file or discarded. For example, in order to execute a conditional operation in a RISC processor, a minimum sequence of three instructions is necessary. The instructions compute the condition code, branch conditionally to an instruction to perform the operation and, if the branch is taken, perform the operation. A conditional-execution architecture reduces this critical path to two instructions that compute the condition code and execute the operation conditionally. Furthermore, the conditional-execution feature allows the execution of operations concurrently with the conditional branch. In case the branch is taken, operations in the branch-taken path that depend on the results of operations executed conditionally are ready for execution. Due to this fact, the conditional-execution model is particularly use-

ful to speed-up branch-intensive programs. Examples of this kind of programs are systems programs such as editors, operating systems and compilers, and programs that use dynamic pointer-based data structures and sparse matrices.

The VLIW architectural template serves as a virtual and scalable target architecture during, possibly repeated, application code compilation for specification optimization. This template is scaled by constraining the resources and used as candidate target architectures. During architecture generation the scaled architecture is used as a framework to hardware allocation. Optimization is performed during this phase to ensure efficient hardware implementation of the scaled architecture.

2.2.2 Implementation Architectural Template

In addition to the architectural template, an implementation template is used to permit further customization for achieving efficient utilization of hardware resources. The implementation template of the data section is the multiple-bus organization illustrated in Figure 2.7. This organization, inspired by the one presented in [42], is composed of a number of single-ported register files and functional units joined by a sparsely interconnected multiple-bus organization. Data are organized into the distributed register files to support the specific requirements of concurrent access of this code. Sparse interconnect is allocated to support the required data transfers between the multiple register files and functional units.

The architecture operates on a two phase clock. In the first phase, data are transferred from the register files to functional unit inputs. On the second phase, data from the functional unit outputs are stored back into the register files. The multiple bus structure provides sparse connectivity between register files and functional units. In Figure 2.7, each horizontal bus is associated with a register file, and vertical busses are connected to functional unit inputs and outputs. Tri-state buffers and multiplexers at some crosspoints in Figure 2.7 provide sparse connectivity between register files and functional units. Each register file is organized as memory with a single read/write port. This organization has both lower cost and higher speed than multiported memories[13].

The set of register files allows concurrent access to data allocated into distinct register files and thus implements the function of the global shared register file of the VLIW architecture

at much lower cost. This is done by careful allocation of data that are concurrently accessed into distinct register files and by providing the interconnection paths to the functional units that operate on these data.

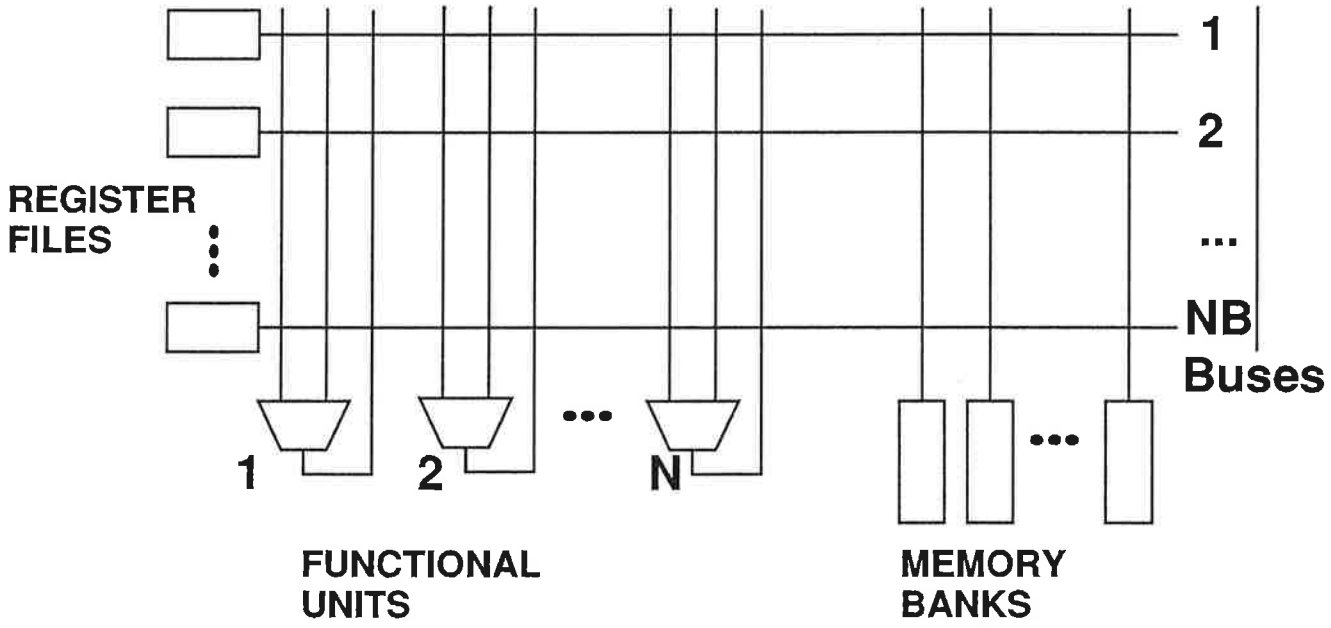


Figure 2.7: Implementation Template

The bus associated with a register file allows broadcast of values to the input busses of functional units. Connections are directional: functional unit input busses may only read from the busses associated with a register file, while functional unit output busses may only write into those busses. This is illustrated in Figure 2.8, for a realization technology that uses busses and tri-state drivers. To read a value from a register file R into an input of a functional unit there must be a connection from R's bus (horizontal) to the (vertical) bus associated with that functional unit input. Similarly, to store the result generated by a functional unit into register file R there must be a connection from that functional unit's output bus to the (horizontal) bus associated with R. Control for the tri-state buffers at crosspoints is issued by the instruction.

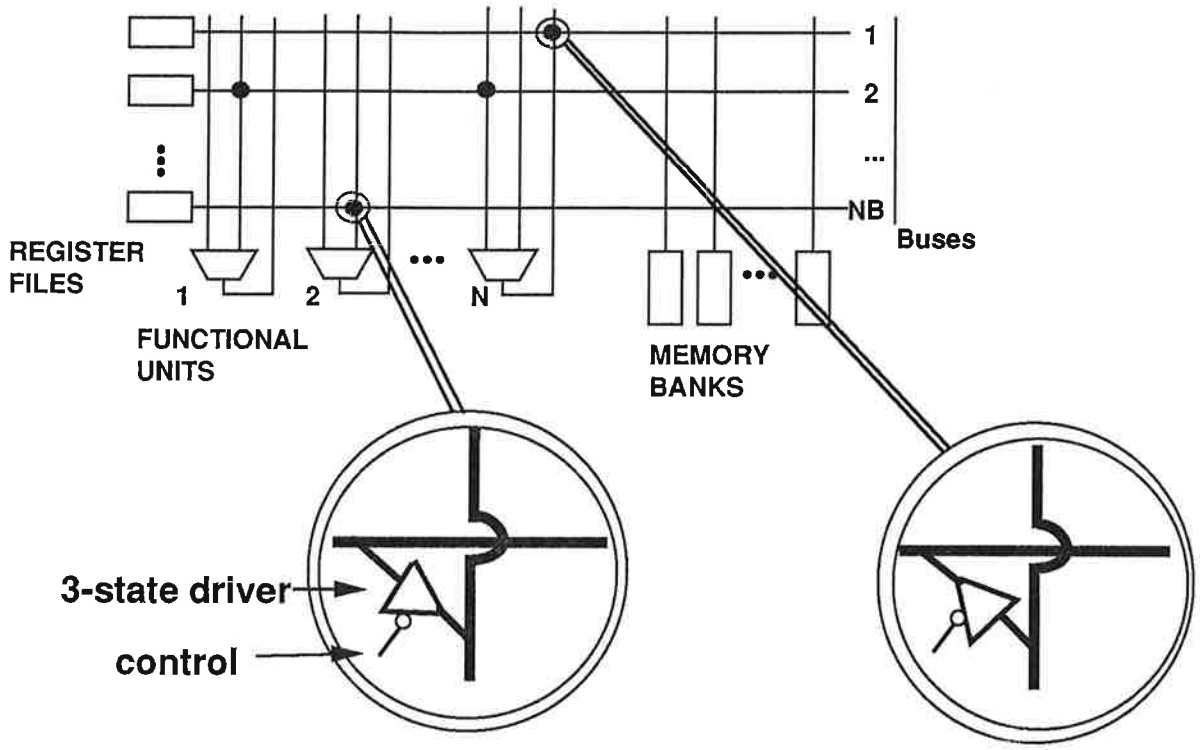


Figure 2.8: Detail of Sparse Interconnect

2.3 Application-Specific Processor Design (ASPD)

Based on the White Dwarf experience, the Application-Specific Processor Design (ASPD) method has been developed. A semi-custom framework is used to reduce the complexity and effort of the design task. This is analogous to and can be viewed as an architecture-level extension of the semi-custom design of ASICs. The input to the synthesis process is the actual application source code written in a conventional HLL such as Fortran or C. The description serves as the architecture specification. The method produces as a result a netlist description of the generated ASP, along with the associated application code.

The ASPD framework is characterized by:

1. the use of an architecture model and simplified design rules to constrain the design style and reduce the design task complexity;

2. the use of predesigned and well-characterized large-grain building blocks as primitives; and

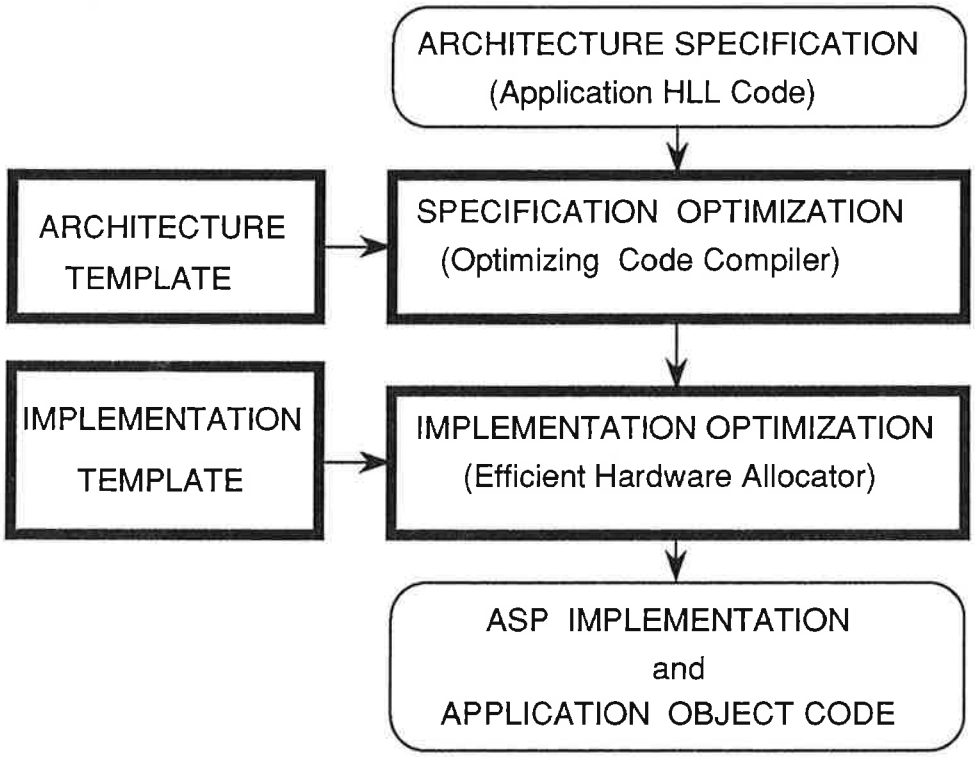


Figure 2.9: Application-Specific Processor Design Method.

3. the extensive use of sophisticated software tools to perform design customization and optimization.

The ASPD method achieves high performance by extracting and exploiting fine-grain parallelism, beyond basic-block boundaries, in the application code and implementing the necessary hardware resources to support its execution. The overall ASPD method is illustrated in Figure 2.9 and involves the following four key components:

1. **Architectural Template**: The architectural template implements the semi-custom design framework, defines a scalable architecture model, and is used to constrain the design style. The range of scalability of the template constitutes the entire design space. The template can be scaled via the imposition of various resource constraints.

2. **Specification Optimization**: The first of two optimization phases, specification optimization has as its objective the maximization of application code performance. It employs a powerful retargetable VLIW compiler to translate the application source code into highly optimized object code containing much fine-grain parallelism. An

efficient architecture is generated by scaling the architectural template and performing repeated compilation of the application code with varying resource constraints. Based on the compiled object code, detailed hardware resources are allocated using systematic procedures. The optimized code serves as the optimized specification of the ASP architecture.

3. **Implementation Template:** The implementation template addresses the issues of efficiency in the architecture design, while maintaining downwards code compatibility with the architecture template. Given one specific instance of the optimized code generated for an scaled version of the architectural template, the implementation template is further scaled in terms of resource utilization by allowing detailed allocation of data and connectivity to satisfy the resource and communication requirements of this object code at a reduced cost.

4. **Implementation Optimization:** The aim of this second optimization phase is to achieve efficiency, or high utilization of hardware resources. The direct mapping of the optimized specification into hardware produces a canonical implementation. This canonical implementation is then pruned, via hardware allocation algorithms, to produce the optimized efficient implementation. Graphs are used to represent relationships of concurrent activation of the hardware resources by the program. Graph-coloring algorithms are employed to map non-overlapping activities to the same hardware unit, thus saving hardware resources. Characteristics of the target application and architecture template are used to identify special kinds of graphs for which efficient algorithms exist.

The system-level interfacing for the generated ASP is as a coprocessor similar to the White Dwarf. The processor is viewed by the host as an efficient "hardware subroutine". Data is loaded under control of the host processor into the processor's data memory, via DMA, prior to execution and transferred back to the host after processing is finished. The multiple bank organization of data memory allows high-bandwidth data transfers, thus minimizing overheads. For example, this organization is suitable for high-bandwidth DSP algorithms[43]. This coprocessor model does not take interrupts or context switches, which

are handled by the host.

The principal input to the ASPD design process is the high-level language code for the application. Under control of the designer, repeated compilation for scaled versions of the VLIW template is performed to search the design space. The performance attained for a given scaled architecture is obtained via execution on a simulator, or may be estimated, for simple programs, by code inspection. Once a design point is selected, the result of specification optimization is presented in the form of optimized code for a scaled VLIW architecture. The VLIW architectural template is then scaled to allow execution of this application code. Operations are assigned for execution by functional units and data are distributed into multiple register files to allow concurrent access to values in registers as required by the application code. Communication paths are provided to support the required data transfers. The implementation template is then pruned to eliminate unnecessary resources without degrading the performance. The resulting ASP design describes the number, sizes, and data allocation of register files, and their interconnection to the functional units and memory banks. The object code to execute the application is also provided.

Chapter 3

Specification Optimization

The specification optimization phase is responsible for achieving performance optimization in the ASPD method. It takes as input the application program, in C or FORTRAN, and produces highly optimized (in terms of extracted parallelism) horizontal microcode, which serves as the optimized architecture specification. The primary workhorse in this phase is a highly retargetable optimizing VLIW, or microcode, compiler.

During the specification optimization phase, the retargetable compiler is first invoked on the input application code using the VLIW architecture template, without any resource constraints, as the target architecture. The resultant parallelized code serves as the initial optimized specification for the ASP architecture. The number of functional units and memory banks needed are determined by the resource requirements of this parallelized code. If the resources required are too impractical to implement or too inefficient in their utilization, the compilation is repeated with resource constraints imposed. This process of repeated compilation with gradual imposition of resource constraints leads to the final optimized specification. The powerful compilation techniques employed in this project and their use in Specification Optimization are presented in this chapter.

3.1 Specification Optimization Procedure

The keyword for the specification optimization phase of the ASPD method is performance. During this phase the designer is concerned with locating the performance levels which are attainable for the target application. This performance is to be achieved under cost constraints. The goal is to find quickly the cost versus performance tradeoff point that

matches the designer's requirements.

The ASPD method makes use of the architectural model to explore the design space. This is achieved by compiling the high-level application code for various scaled versions of the ideal VLIW architecture. The designer quickly explores various cost versus performance tradeoff points by performing repeated compilation for scaled architectures. Initially, the compiler is allowed to schedule without resource constraints as many parallel operations as it can find, and is requested to report the amount of resources required. The result of this compilation locates the point of maximum obtainable performance. By repeatedly scaling down resources available to the compiler, the designer generates a set of possible design options and performs tradeoffs between cost and performance.

The process of repeated compilation places certain requirements on the compilation techniques to be used. In order to have broad application and usefulness, the compilation techniques must be *general*, *scalable* and *monotonic*. A *general* fine-grain parallelization technique is applicable to a wide range of programs. Such a technique is not pattern-sensitive, i.e. only applicable to programs that exhibit definite patterns of data and control actions. Examples of parallelization techniques that are not general include vectorization techniques, which require regular data access patterns, and systolization, which places strict requirements on data access and recurrences. Early software pipelining techniques[81] are also not general, because they only apply to single-statement loops having simple recurrences. The non-general techniques exhibit a form of all-or-nothing behavior: if the target application and the compilation technique have a good match, extremely high performance may be obtained, but benefits from the same technique for any other kind of application may be almost nil.

The parallelization technique must be *scalable*: the speedups obtained with the technique must be well correlated with the amount of resources required. Trace Scheduling[37] is an example of a technique that is neither general nor scalable. It is not general because it requires target applications that have highly predictable flow of control. It is also not scalable because a wide range of cost and performance tradeoffs is only possible if the target application has long traces. Furthermore, the presence of conditional jumps in the code may cause non-linear growth in the resulting code size. This breaks the correlation be-

tween resource requirements and performance. It is known[37] that many subroutines for computation-intensive scientific applications have characteristics suitable for trace scheduling. However, present day scientific applications also include many kinds of code beyond the computation-intensive kernels, e.g. user communication and system interfaces. These codes do not usually exhibit characteristics that match trace scheduling. A compilation technique is not *monotonic* if the code produced by using more resources executes slower than a code schedule with less resources.

A general parallelization technique achieves a broad range of applicability for ASPD. It must also be scalable, to simplify exploration of the design space, due to the good correlation between performance obtained and amount of resources available. Furthermore, due to consistent speedup across various kinds of code, a general and scalable technique achieves speedup for the whole application as opposed to only benefiting the highly parallel kernels in the program. The PS transformation finds parallelism among operations from beyond basic blocks. Some operations from beyond conditional branchs whose operands are ready are scheduled speculatively. A possible source of non-monotonicity is the case in which the compiler is overly aggressive at the early scheduling of speculative operations. This may waste some resources to execute operations whose results are not used most of the time. The heuristic choice functions are carefully chosen to avoid such situation.

The intended application domain of ASPD is scientific and engineering embedded computation. In this application domain, a substantial fraction of the computation time is spent executing loops. Therefore a general and scalable optimization mechanism that handles loops efficiently is a key factor to achieve efficient performance. The specification optimization phase implements both intra-loop-iteration and inter-loop-iteration scheduling techniques. *Intra-iteration* techniques find parallelism among operations belonging to the same loop iteration. This optimization is also applicable to code belonging to outer loops or in between loops. Examples of intra-iteration techniques are trace scheduling[36] and percolation scheduling[62]. *Inter-iteration* techniques are capable of scheduling concurrently operations belonging to distinct iterations of the loop. Speedup is obtained by achieving overlapped execution of loop iterations. This is done by starting the execution of a future iteration before execution of the current iteration is completed. Examples of techniques for

inter-iteration optimizations are loop unrolling and software pipelining techniques [54, 5, 33].

3.2 Intra-Iteration Techniques

Traditional microcode compilation techniques are limited to finding parallelism among operations from a single basic block[1]. However, experiments have shown that the maximum speedup attainable from parallelism in basic blocks is usually only a factor of two to three[64]. To increase the amount of parallelism achievable it is necessary to go beyond conditional jumps by concurrently executing operations from multiple basic blocks. Trace scheduling[37] is a technique to extract parallelism beyond basic block boundaries. Experiments with trace scheduling in scientific FORTRAN programs have found speedups of up to one and possibly two orders of magnitude[64]. However, trace scheduling has a reduced scope because it requires programs that exhibit a highly predictable flow of control. Another limitation of trace scheduling is that it cannot achieve inter-iteration optimization by fully pipelining loop iterations (see [4]). Trace scheduling achieves a limited form of inter-iteration optimization via loop unrolling [35], but this approach entails pipeline startup and flush overheads at every group of unrolled loop iterations.

To extract fine-grain parallelism, an enhanced version of *Percolation Scheduling* [62, 34] is adopted in this work. Percolation Scheduling (PS) is a code parallelization technique that evolved from experience with trace scheduling[37]. It is applicable to more general classes of code than trace scheduling, being particularly good at finding parallelism in branch-intensive code such as systems programs. Furthermore, it has been demonstrated to obtain consistent speedups for many classes of code [59]. Percolation Scheduling is also the compilation technique selected for the IBM VLIW project, which is a general-purpose VLIW processor for efficient execution of systems and general-purpose applications.

Percolation Scheduling is composed of a set of semantics-preserving transformations that convert an original program graph into a more parallel one, globally rearranging the code to extract parallelism. The core transformations regulate the conditions under which an operation or conditional jump may be moved between adjacent microinstructions. The optimization process starts with serial code containing one operation or conditional jump per instruction. By repeated application of core transformations it is possible to move, or per-

colate, operations and conditional jumps to preceding instructions to achieve more parallel code. The treatment of operations and conditional jumps is unbiased and regulated by data dependencies. Percolation Scheduling approximates, at compile time, the execution schedules of operations in a dataflow processor. The set of candidate operations that may be scheduled in a given instruction are those operations whose input arguments are available. For example, after one operation o is moved to a preceding instruction, other operations which use values produced by o become eligible for motion to the instruction previously occupied by o. Because operation scheduling is done at compile time, it is feasible to perform extensive code analysis to determine the priority of execution of operations. This may avoid the waste of resources which happens in a dataflow processor, where operations, whose inputs are ready but whose outputs are not immediately needed or not needed at all, are executed too soon. Therefore, Percolation Scheduling achieves efficient extraction of parallelism without requiring the expensive hardware mechanisms of dataflow processors.

Percolation Scheduling is not restricted to optimizing one execution path at the possible expense of other execution paths, as in trace scheduling. Operations belonging to distinct execution paths are treated uniformly, and the effects of parallelization can benefit multiple execution paths. This feature is particularly useful for branch-intensive applications. In [33], a variation on Percolation Scheduling, called Extended Percolation Scheduling, has been adapted to the conditional-execution model and extended to avoid blocking of the motion of operations due to data dependencies other than direct flow dependency. Percolation Scheduling generates high-performance code for branch-intensive code which is very difficult to speed up with other conventional techniques. Performance improvements on the order of tenfold for a 16-ALU VLIW have been obtained[33]. The Percolation Scheduling transformations can expose large amounts of parallelism even in the presence of conditional jumps.

In ASPD the main concern is not only absolute performance, but also efficient hardware usage. Nicolau and Ebcioglu[34] extended Percolation Scheduling to handle target architectures with constrained resources by defining the notion of *unifiable-ops*. For each instruction L, *unifiable-ops(L)* is the set of operations and conditional jumps in the program that could be moved to L without requiring other operations to move first. Algorithms to compute

and maintain *unifiable-ops* information during the compilation process are presented. A *choice function* assigns priorities to the operations and is used to decide which operations in *unifiable-ops(L)* are actually moved to Instruction n in the presence of limited resources. This approach provides a separation of concerns: the computation of *unifiable-ops* is algorithmic and rigorous, and the Percolation Scheduling transformations that perform the motion are semantics-preserving and provably correct. The problem of optimal code scheduling with constrained resources is NP-complete, thus the heuristic choice function is necessarily sub-optimal[34]. The use of a sub-optimal heuristic may introduce non-monotonicity in the resulting code schedule.

In our implementation, the algorithms in [34] have been extended to account for the possibility of renaming destination registers of operations during percolation. Furthermore, an efficient representation of operation sets using bit-sets has been proposed and implemented in what is, to the best of our knowledge, the first and only implementation so far of the algorithms presented in [34]. The use of force-directed scheduling[69] is proposed in this work as the choice function for Percolation Scheduling. *Force-directed* scheduling[69] is a technique which attempts to achieve uniform use of resources and has produced good results in practice. As initially proposed, force-directed scheduling applies only to straight-line code or simple conditional constructs. Unifiable-ops provides information about where each operation could be scheduled (across basic block boundaries). The force-directed heuristic is used as the choice function to select where each operation should go. This approach effectively extends the applicability of force-directed scheduling beyond straight-line code.

3.2.1 Percolation Scheduling Core Transformations

Percolation Scheduling is composed of three primitive transformations: *move-op, move-cj* and *delete*. These transformations are local, involving only two consecutive microcode instructions. By repeated application of primitive core transformations, powerful code motions are performed. These transformations are atomic, i.e. involve the motion of a single operation or conditional jump, which is the smallest quantum of execution on the VLIW architecture. After each transformation the code is semantically equivalent to the original program[4]. This allows detailed control of the fine-grain (operation-level) parallelization

process and a wide range of tradeoffs between resource requirements and performance obtained from concurrency. This feature makes Percolation Scheduling a *scalable* fine-grain parallelization technique. Similarly, Percolation Scheduling is a *general* parallelization technique, because it schedules concurrent execution at the fine granularity level of concurrent execution of operations and conditional jumps. Therefore, it is able to capture highly irregular forms of parallelism not visible at coarser levels. Parallelism at coarser grains e.g. iteration-level or subprogram level, may also be found at the fine-grain level (e.g, [8]). The importance of fine-grain parallelism exploitation has already been recognized to some extent, and is reflected in the use of horizontal microcode in many high-performance application-specific processors[75, 81, 51]. Furthermore, [4] shows that the Percolation Scheduling core transformations are *complete* with respect to the set of all possible local, dependency-preserving transformation on program trees. This means that no alternate parallelization system that also has locality of application and is dependency-preserving is capable of exposing more parallelism at this level.

Figure 3.1 illustrates the *move-op* transformation. In this example, Operation a in Instruction $L2$ is moved to a predecessor instruction $L1$. After the move, the successor of $L1$ on that path is a new instruction $L2\_p$ which is a copy of $L2$ without the moved operation. The motion is allowed only if no data-dependency constraints are violated. These constraints are direct dependency or write-live dependency. A *direct dependency* exists if some operation in L1 computes an input value for a. A *write-live* dependency[1] exists if a writes a register or memory location which is either read by some other operation in L2 or live at Instruction Ly. After the motion of Operation a, L2 is preserved because it has other predecessors. This guarantees semantic correctness because the upward motion of a on the path L1-L2 does not affect other execution paths that traverse L2 but not L1. Other invocations of move-op might move a to the other predecessors of Instruction L2. If L2 has no other predecessors and all operations have been moved from it, it is removed by the *delete* transformation.

Similarly, Figure 3.2 illustrates the *move-cj* transformation. In this example Instruction L1 is followed by Instruction La which executes a conditional jump. If the condition code which determines the jump condition is available at Instruction L1, it may execute the

[1] In some texts, this dependency is called "write-after-read" or "anti-dependency".

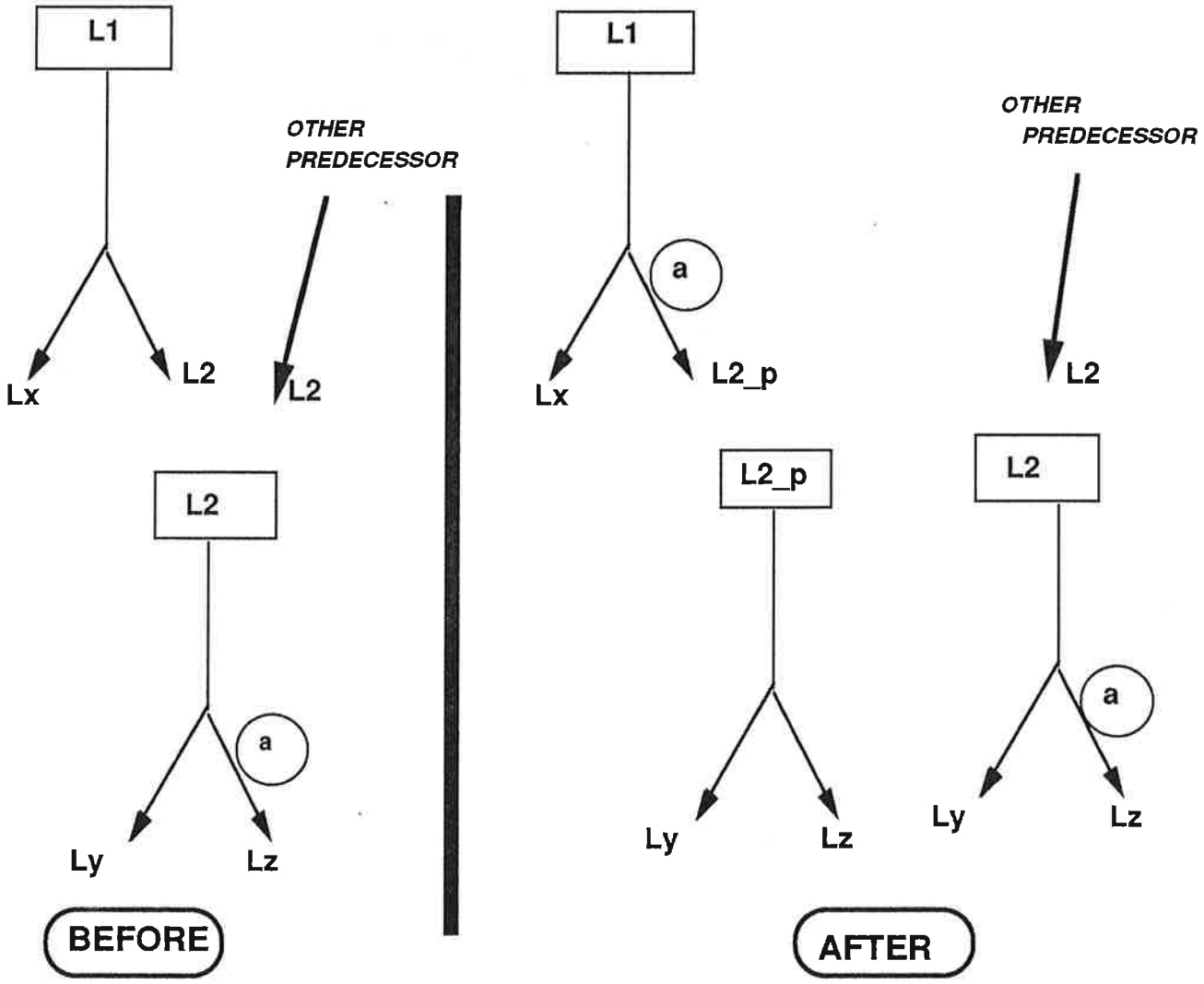


Figure 3.1: Move-op Example

conditional jump itself. The conditional jump is moved into Instruction L1 which is changed to target two new unconditional instructions Lx and Ly. Lx is a copy of La that behaves as if it is known that the condition code is true, and Ly is a copy of La that behaves as if it is known that the condition code is false. As in the previous case, La is preserved in its original form if it has predecessors other than L1. If the conditional jump may be moved to all predecessors of La, it becomes an empty instruction and is deleted. The move-cj mechanism allows for early resolution of conditional jumps. This is desirable to allow motion of operations from paths after the conditional jump whose motion is otherwise

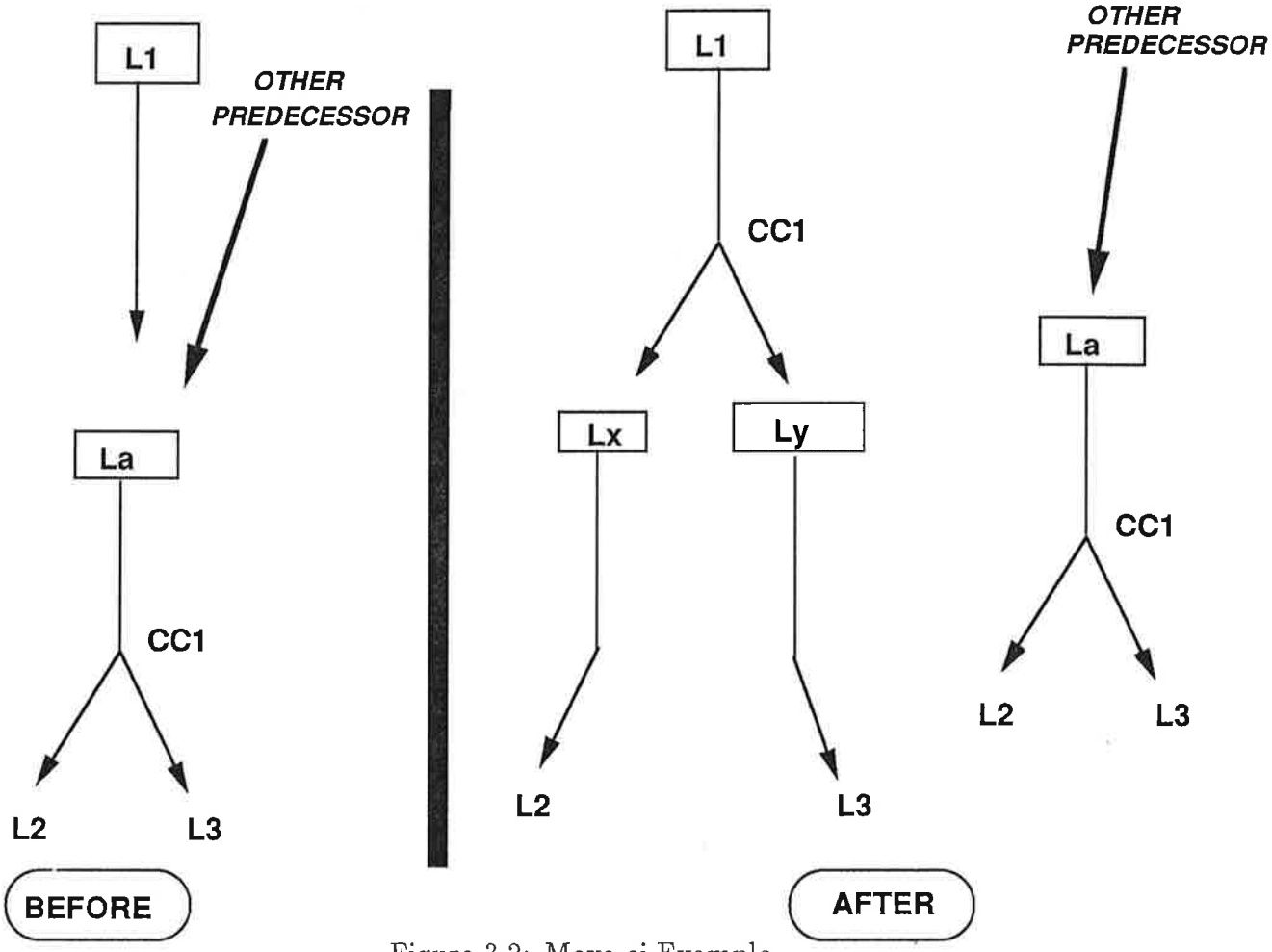


Figure 3.2: Move-cj Example

blocked by write-live dependencies on the other path.

3.2.2 Percolation Scheduling with Register Renaming

In many situations, the upward motion of an operation *o* is blocked only by a write-live dependency. This happens if *o* is moved from some Instruction L2 into Instruction L1 and the old value of the destination register of *o*, say register `z` is needed at some other path starting at the tip of L1. In these cases, the motion can still be performed, by using a new destination register for *o*. The opcode for *o* is changed to store its result into a new register `z'`. A copy operation to move the contents of `z'` into `z` is left in L1 in the place of *o*. Figure 3.3 illustrates Move-op with renaming. The addition of `x` and `y` is moved from the right branch of L2 to L1. However, the old contents of register `z` are required by a `load` operation

in the left branch of L2. The destination register of the addition is changed to write the result into z', and a copy from z' into z is left in L2.

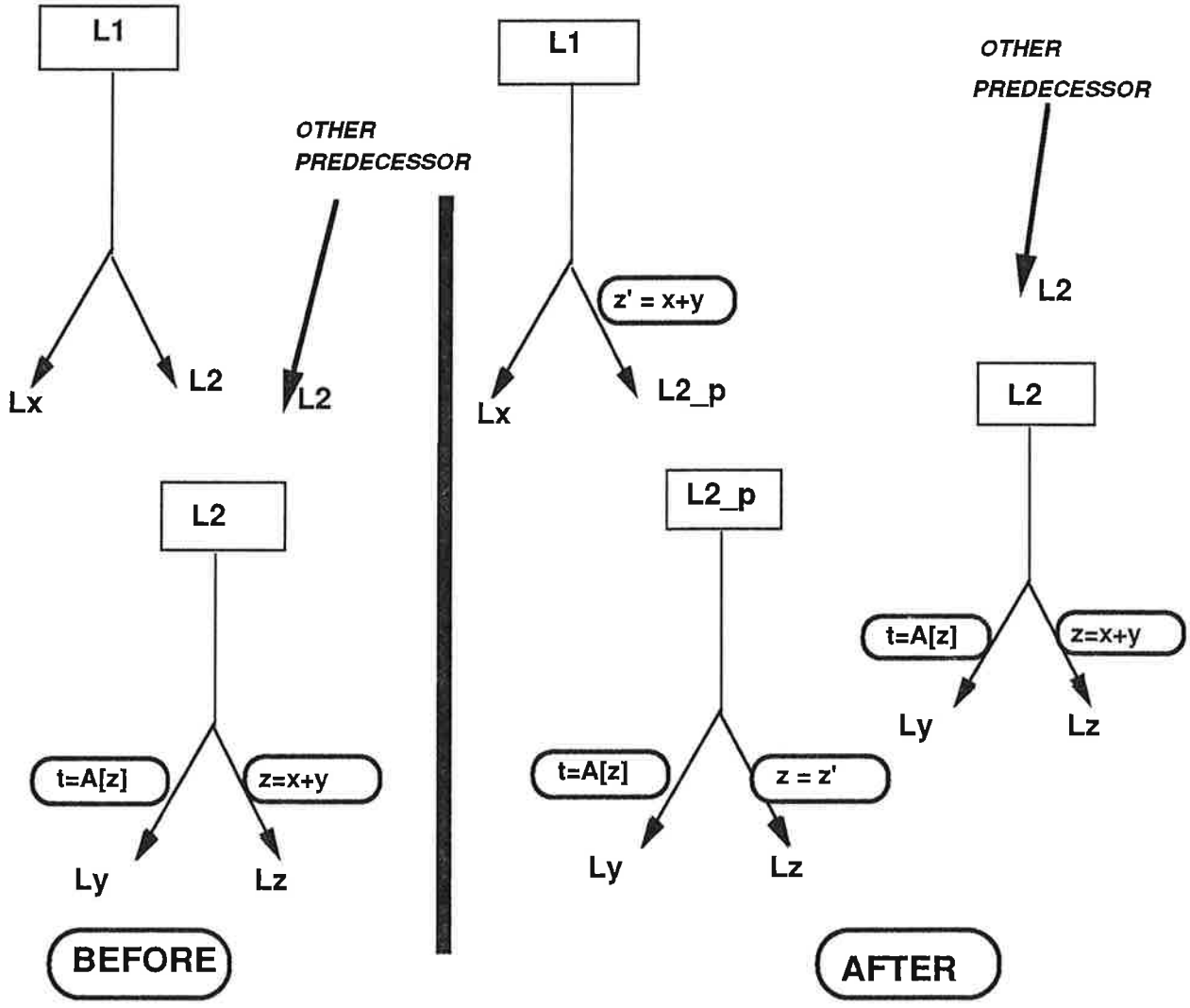


Figure 3.3: Example of Move-op with Register Renaming

The move-op transformation with renaming has the drawback of introducing an extra copy operation in the code. It is desirable, however, because the result of operation *o* becomes available at least one cycle earlier, inducing further possibilities of code motion. Other operations that require the new value of z computed by the addition may be moved into L2, and thus execute earlier, by reading the required value from register z' instead. Furthermore, if all such operations are changed to read directly from z', the copy operation

becomes dead and may be removed from the code.

The move-op transformation with renaming is a key to achieving efficient loop pipelining. Frequently, each loop iteration computes a value in some register which is last used some cycles later in the same iteration. Without the renaming transformation, future loop iterations that would overwrite the same register location are delayed until the value is last used. This reduces the amount of pipelining achievable. By using register renaming, the copy operations preserve the value until it is last used. This allows more overlap between loop iterations. Percolation Scheduling with register renaming is inspired by [25], which demonstrates its usefulness for enhanced extraction of parallelism. The use of register renaming in Percolation Scheduling is first implemented in the VLIW compiler for the IBM VLIW prototype[33].

3.2.3 Computation of Unifiable-ops

For each instruction L, *unifiable-ops(L)* is the set of operations or conditional jumps that can be immediately moved from anywhere in the program to the top of L by a sequence of core PS transformations. In other words, if there exists in the program some operation or conditional jump *o* such that *o* could be moved to an imaginary empty predecessor of L by a series of PS transformations involving only *o* or its copies, then *o* is present in *unifiable-ops(L)* . *unifiable-ops(L)* is a dataflow attribute, similar to live-variables or reaching definitions information.

The algorithm to compute unifiable-ops presented in [34] is intuitively described as a kind of "simulated Percolation Scheduling". Consider every operation or conditional jump *o* in some instruction L of the program. Initially the algorithm finds the data-dependency conditions under which *o* could be moved to each predecessor of L. If the data dependency constraints are satisfied for some attempted motion, the operation or conditional jump is annotated into the unifiable-ops set of that instruction. The algorithm then proceeds recursively, considering as candidates for motion not only the operations originally present in each instruction but also the operations in the unifiable-ops sets. The direction of the motion of operations of conditional jumps is against the control flow, attempting to move operations "closer" to the program's entry point. The program's control flow graph is a directed graph and motion along loop back-edges is prevented.

If some instruction L contains an operation that computes a new value for register a, then the motion into L of an operation that uses the value of a as an argument is prevented by data dependencies. In other words, the motion of the set of all operations that use a as one input argument is *killed* at L. For each instruction L, the set of all operations that require values computed at L is constructed. This is the set of operations whose motion through L is killed due to direct dependency. The set of operations whose motion through L is killed due to write-live dependency is computed similarly. For an instruction L that has a successor Instruction S, candidate operations to the unifiable-ops set are operations in the unifiable-ops set of S and operations in L on the path to S. Candidate operations that are not killed on the path in L to S are in the unifiable-ops set of L. The unifiable-ops set of L is computed by considering all of its successors. The process of simulated percolation scheduling is performed quickly by manipulating sets of operations. Care is taken to avoid blocking the motion of operations that are available on all successors and whose motion is killed only by their write-live dependencies. The unifiable-ops sets is computed in one pass by visiting instructions in reverse depth-first order starting from the entry point of the program. The algorithm in [34] uses these sets to simulate the percolation of many operations quickly. It also suggests the use of a bit set representation of sets of operations to allow efficient implementation.

Furthermore, unifiable-ops information may be maintained incrementally and locally with the PS transformations. This is possible if each instruction is only considered as a target for motion of operations after all of its predecessors have been considered. We have implemented and extended the algorithm of [34] to use the renaming of destination registers. This is done by using an alternative definition of the set of *killed* operations that ignores write-live dependencies. For each instruction L, two kinds of unifiable-ops information are computed: *unifiable-ops(L)* and *runifiable-ops(L)*. Unifiable-ops(L) is the set of operations that could be immediately moved to L, as before. Runifiable-ops(L) is the set of operations that could be immediately moved to L by a series of PS transformations, but would require renaming of the destination register. This information is used to find code schedules that require fewer registers, while maintaining parallelism. During Percolation Scheduling, operations that do not require renaming are moved first, to attempt to use available functional units without

requiring extra registers.

3.2.4 Memory-Reference Disambiguation

Indirect memory references due to the use of array indices and pointers pose problems to the computation of data dependencies. For example, assume that a memory `store` operation is followed by a `load` operation. To determine whether the `load` operation could be performed either concurrently or before the `store`, it is necessary to check whether the memory locations accessed by the two memory references may be the same. A conservative approach would assume that there is always a data dependency. This serializes the memory access and preserves the original order specified in the program. This approach may cause considerable reduction in the achievable performance. The alternative approach is to perform *memory-reference disambiguation*[2]. The compiler attempts to determine whether two memory references may access the same memory location. This is done by constructing symbolic expressions for the memory addresses, and then checking if the expressions may ever have the same value. For example, for array accesses `A[i]` and `A[i+1]` it is not difficult to determine that the memory locations involved are never the same. However, in memory references such as `A[i]` and `A[j]`, where `i` and `j` are input variables, no disambiguation is possible and the memory accesses must be performed serially. Memory-reference disambiguation is crucial for high-performance architectures because of the higher memory bandwidth required to keep the multiple functional units busy.

Our algorithm for Memory-reference Disambiguation is an enhanced version of the algorithm sketched above, which is fully described in [35]. To build symbolic expressions, the compiler expresses the address of every memory access in terms of loop invariants, loop indices, and definitions of other variables. Initially, the symbolic expression of every operation *o* is initialized to *o*. Each operation *o* corresponds to a free variable that represents its result in a symbolic expression. For example, let us assume that operation *o1* writes variable `a` which is an argument of some other operation *o2*, that *o2* is `m := a + 1`, and that the definition of `a` in *o1* is the only one that reaches *o2*. By substituting the symbolic expressions of the operations that define inputs of *o2*, the symbolic expression for the value

[2]In some texts, memory-reference disambiguation is called "memory anti-aliasing".

of the result of *o2* is *o1 + 1* . To complete the symbolic expression for m, the algorithm is applied recursively to find the symbolic expression for the result of *o1*. This is done by creating an expression in which the operator is the opcode of *o1* and substituting in that expression the symbolic expressions for the operations that define input arguments for *o1*. The goal is to build linear expressions of the form $k_1 * o_1 + k_2 * o_2 + k_3 * o_3 + k_4 * o_4 + ... + k_n$, where the o_i are variables and the k_i are integer constants. An operation may be reached by more than one definition of an input argument. Here, the symbolic expressions contains the "OR" operator to express alternative values for that argument. In the above example, if there are two definitions of a reaching *o2*, say *o1* and *o3*, the symbolic expression for the result of *o2* will initially be *(o1 + 1) OR (o3 + 1)* . Substitution of the symbolic expressions for *o1* and *o3* completes the expression. If the number of alternatives makes an expression impractically large, it is feasible to substitute the values represented by those alternatives by a newly created free variable. The resulting expression contains less detailed, but still useful, information at a reasonable cost.

To compare two memory accesses whose addresses are expressed by such symbolic expressions e_1 and e_2, the compiler checks if the symbolic equation $e_1 - e_2 = 0$ has a solution. Expressions of this form are called *linear diophantine equations.* The process of substitution stops for operations other than addition, subtraction or multiplication because, in this case, the resulting expression would contain operators other than addition and multiplication. This leads to complex non-linear equations that cannot be handled by the mechanism to solve diophantine equations. In such cases, the compiler takes the conservative approach and schedules the memory accesses serially.

Similarly, the process of substitution must stop if the symbolic expression of *o* is required while deriving the symbolic expression for *o* itself, otherwise infinite recursion follows. This happens if operation *o* may use a value previously computed by itself, possibly in a previous loop iteration. In this case, the result of operation *o* may be a *loop induction variable*[2]. A simple example of an induction variable is the loop counter in

```
for(i=j,i<10,i=i+3){
            ..v[i].. v[i+1]
}
```

Loop induction variables are detected before the creation of symbolic expressions. The

induction variable in a symbolic expression is substituted by a closed form expression in terms of an imaginary loop counter. In the example loop above, `i` is an induction variable, and its symbolic expression at the beginning of the loop is $i_0 + \alpha * 3$. The value of `i` at the entry point before the loop entry is represented by i_0. The imaginary counter associated with the loop, α, starts at zero and is initialized by one on every loop iteration. In the above loop, the memory address referenced by `v[i]` is expressed by $K_{address-of-v} + i_0 + \alpha * 3$, where the constant $K_{address-of-v}$ is the memory address of the start of vector `v`. Similarly, the memory address accessed by `v[i+1]` is given by $K_{address-of-v} + i_0 + \alpha * 3 + 1$. The difference of the two symbolic expressions is *1* and thus the corresponding memory addresses in these memory references are never the same.

3.2.5 Detection of Induction Variables via Symbolic Substitution

In this subsection, an algorithm for the detection of induction variables is presented. It is inspired by the "variable folding" method presented in [60]. The traditional algorithm for detection of induction variables is described in [2]. That algorithm works by identifying those variables whose only definition in the loop is an operation of the form `var = var + K` for some constant `K`. Notice that `K` may also be a loop invariant. These are called *basic induction variables*. Next, variables whose value is a linear function of basic induction variables are detected. This is done by performing repeated passes over the loop body, which may be expensive computationally. Furthermore, the simple algorithm cannot detect the case of mutually referencing recurrences in which there are no basic induction variables. One example is the following loop:

```
for(..){
    var1 = var2 + K1;
    var2 = var1 + K2;
    }
```

The proposed algorithm works by identifying loop variables which are *candidates* to be induction variables. Such are variables that:

- are live at the entry of the loop, and
- have a definition inside the loop that reaches the loop entry.

The algorithm proceeds by computing the symbolic expression for the values of candidate induction variables at the loop entry. The symbolic expression for the result of an imaginary copy operation that copy the candidate induction variable into itself is computed. To compute the symbolic expression, only definitions that reach the loop entry from inside the loop are considered. A candidate variable a is found to be a loop induction variable if the resulting symbolic expression for result of the copy operation a = a is of the form $a_0 + K$ where K is a constant and a_0 is the value of a at the start of the loop.

The closed form expression for an induction variable a at the loop entry is $a_0 + \alpha * K$ where a_0 is the symbolic expression for a before the loop, α is the imaginary counter associated with the loop, and K is the constant value by which a is incremented on every loop iteration. The algorithm continues recursively to find the symbolic expression for a_0 by tracing back definitions of a that reach the loop entry. The algorithm is applicable to loops that have multiple entry points. In this case, the symbolic expression for an induction variable must be of the form $a_0 + K$ with the same value for K on every path in the loop from one entry point to another entry point.

This technique handles naturally the chains of operations that define induction variables. In this case, the symbolic expressions built for the results of operations in a chain are expressed in terms of the result of the operation in the chain that is closer to the loop entry. When the variable associated with that expression is found to be an induction variable and its value is expressed in closed form, the other expressions that refer to it also use the closed form. Furthermore, this algorithm handles the case of loops that have conditional statements and the induction variable is incremented in distinct ways in the paths inside the loop. For example, the loop in Figure 3.4 has two paths in the loop body. The induction variable, i, is incremented by 4 in one path through the loop and is incremented by 2, and then by 2 again, in the other path. Traditional induction variable detection algorithms do not detect such cases.

The use of symbolic expressions to detect induction variables requires the powerful mechanisms of reaching definitions flow analysis and construction of symbolic expressions. Therefore, the above algorithm is probably too costly for implementation in a simple compiler for sequential processors, e.g. CISC. However, a powerful compiler with advanced scheduling

```
for(..) {
        if(..) then
               i = i + 4;
        else
              {
               i = i + 2;
               ...
               i = i + 2;
              }
        }
```

Figure 3.4: Induction Variable Detection in Loop Body With Conditional

capabilities such as required by highly pipelined RISC processors and modern superscalar architectures requires these mechanisms for other reasons. In this case, the improved induction variable detection capabilities, faster execution speed and simplicity of implementation make the use of this algorithm desirable for the compiler implementation.

3.3 Inter-Iteration Techniques

The speed at which inner loops are executed is critical for the total run time of an algorithm. If each of the inner loop iterations can be independently executed then vector instructions can be used in a supercomputer, or iterations can be allocated to distinct processors in a MIMD or SIMD architecture. The alternative case happens if an iteration depends on input values which are produced by some previous iteration. In this case the best performance is obtained with overlapped execution of loop iterations, when loop iterations are started without waiting for previous iterations to complete. Ideally, each loop iteration should be scheduled to produce the values needed for the subsequent iterations as quickly as possible. The subsequent iterations would then be started and execution would proceed concurrently. Multiple loop iterations, in different stages of execution, are in progress simultaneously. In a VLIW and horizontally microcode architectures, the schedule for overlapped execution of iterations is generated statically. This is called *software pipelining*[81, 30, 54, 4], which consists of scheduling the operations in each iteration such that the iterations can be continuously initiated in pipelined fashion to yield optimal throughput. Figure 3.5 illustrates Software Pipelining. Assume that the loop in the figure has no inter-iteration dependencies and that Operation `A` produces a value used by Operation `B` which in turn produces a value

for Operation C. The overlapped execution of loop iterations with an issue rate of one iteration per cycle is illustrated at the center of the figure. A software pipelined schedule to achieve this execution rate is illustrated at the right. In this case, the software pipelined schedule has a steady state composed of one instruction that jumps back to itself.

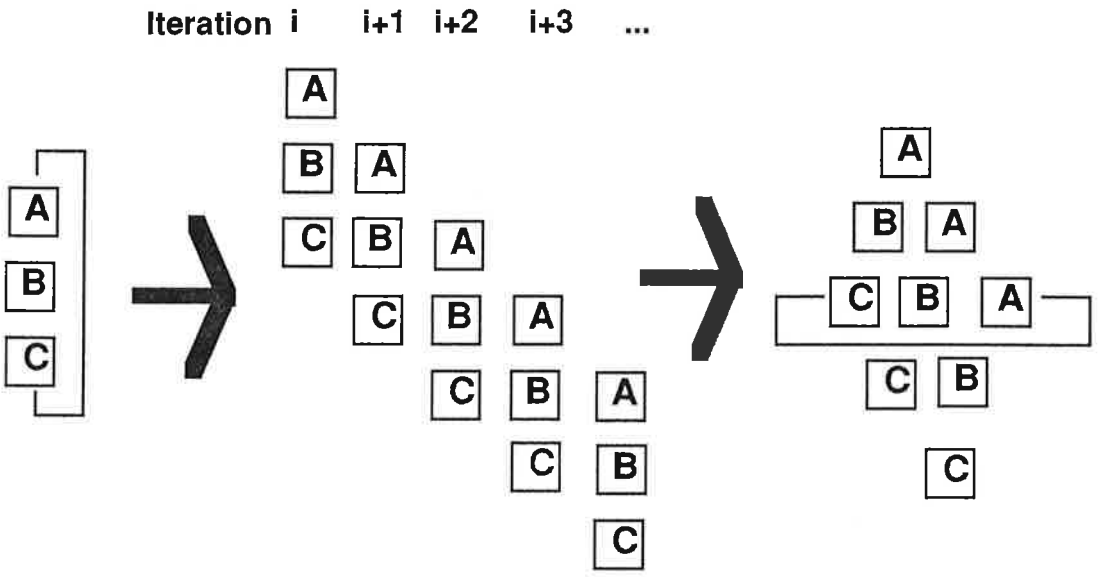


Figure 3.5: Software Pipelining Example

3.3.1 Enhanced Software Pipelining

The inter-iteration optimization technique adopted in this work, called *Enhanced Software Pipelining*[33], is capable of handling loops that have conditional statements inside the loop body. Both "*while*" loops and "*for*" loops are treated in the same way. The resulting schedule for these loops may have distinct iteration issue rates for each path through the loop body. Other software pipelining techniques[24, 54] are restricted to a single issue rate for iterations, by constraining all paths through the loop body to the same length. Enhanced Software Pipelining is capable of pipelining loops with conditionals so that distinct iteration issue rates are achieved for the distinct paths in the loop. The generated schedule is capable of handling the case in which, say, iteration $i + 2$ finishes before iteration i. This may happen if iteration i takes a "long" path through the loop body, and future iterations take shorter

paths. Enhanced Software Pipelining was developed and first implemented in the compiler for the IBM VLIW prototype.

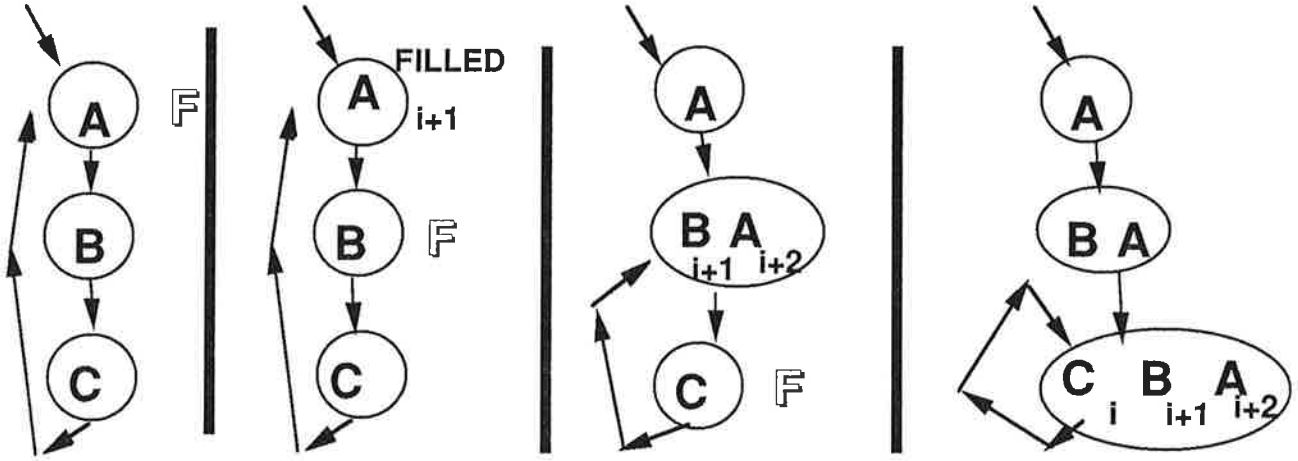


Figure 3.6: Enhanced Software Pipelining Algorithm

The algorithm for Enhanced Software Pipelining is best understood by observing the idealized characteristics of overlapped execution of loops. For simplicity of presentation, the following discussion assumes single-cycle execution of operations. Let us assume that a target issue rate of one iteration per cycle is desired. During execution of the loop, an idealized pipelined schedule should issue all operations of the first loop iteration whose input arguments are available. These are called "Level 1" operations. The execution of these operations computes arguments for some other operations in the first loop iteration. In the second cycle of execution, the execution of the first loop iteration continues by issuing those operations that require arguments computed by Level 1 operations and become ready for execution after the first cycle. These are called Level 2 operations. The idealized execution of the first loop iteration continues in this manner by issuing operations in Level $i+1$ in the cycle immediately after operations in Level i. Operations from the first iteration are issued until cycle N, where N is the length of the longest dependency chain in the first iteration. Similarly, in this idealized execution schedule with pipelining, the Level 1 operations of the second loop iterations are also issued in the second cycle concurrently with Level 2 operations of the first iteration. In the next cycle, Level 1 operations of the third iteration

are issued concurrently with Level 2 operations of the second iteration and Level 3 operations of the first iteration. The overlapped execution of loop iteration continues in this way until there are no more iterations to be issued and the remaining operations are issued until the loop execution is completed. In the example in Figure 3.5, operation `A` is the only Level 1 operation. Similarly, `B` is the only Level 2 operation, and `C` is the is the only Level 3 operation.

The Enhanced Software Pipelining algorithm simulates the above process. It works by performing controlled Percolation Scheduling of operations through the loop back edge. First, it uses the Percolation Scheduling transformations to fill the instruction at the loop entry with operations that are ready for execution (Level 1 operations). The search for Level 1 operations spans the whole loop body, stopping at the loop backedge. Next, the algorithm fills the successors of the instruction just filled with Level 2 operations. Before doing any code motion, the algorithm also adjusts the compiler's view of the position of the loop backedge, to include the entry instructions of the loop, which are the instructions just filled. In this manner, data-ready operations (Level 1) of the next iteration are also visible to the compiler and are scheduled concurrently with Level 2 instructions of the current iteration. The process is repeated by moving the loop back edge past the instructions just filled. Initially, a loop back edge is defined by the last instruction in a path through the loop body that jumps to the instruction at the loop entry. A path through the loop forms a cycle of directed edges. By moving the back edge to the successors of the loop entry instructions, a search for data ready instructions that starts at those instructions will also include the operations in the loop entry instructions just filled. In other words, the Percolation Scheduling mechanism will move operations through the original loop back edge. This will cause the instructions containing those operations to be replicated, because they are located in instructions that have predecessors outside of the loop. The replicated instructions will compose the loop prologue in the pipelined schedule. The algorithm continues by moving the loop back edge and filling up the instructions after the back edge with operations in the loop body that are data ready. The process stops when there are no more operations from the first iteration to be scheduled. The process eventually terminates because there is a finite number of operations in the first loop iteration.

It is important to notice that the algorithm works by controlled application of Percolation Scheduling transformations. Because each PS transformation is atomic, the code after each transformation is semantically equivalent to the original program. Therefore, the process of compilation can be stopped at any time. This provides the scheduling mechanism with considerable flexibility to satisfy various requirements of resource constraints and memory bandwidth during the scheduling process, and is a key factor in the efficient resource utilization achieved. For example, if the target architecture has limited resources, e.g. N ALUs, the search for Level 1 operations only percolates upwards the N most important operations before moving the loop back edge. In this case the pipelined schedule achieves a lower issue rate as compared with an idealized dataflow schedule. The amount of overlap among iterations, i.e. the amount of parallelism, is effectively matched to and controlled by the amount of available resources. This feature makes Enhanced Pipeline Scheduling a *scalable* fine-grain parallelization technique.

3.3.2 Removing Extraneous Copy Operations from Pipelining

When the minimum distance between the definition of a value and the last use of that same value in a loop iteration is greater than one cycle, the pipelining procedure uses register renaming to achieved increased iteration issue rate. This introduces copy operations to preserve the value until its last use, because future iterations that execute in overlapped fashion overwrite the register originally holding the value. It is possible to avoid these copy operations having each overlapped loop iteration using its own register to hold the value. This approach involves code replication: the loop steady-state code must represent the fact that the overlapped iterations using distinct register names lead to distinct code sequences to represent the states of execution of those iterations. These COPY operations require the use of functional units for their execution. To reduce the program's requirements for functional units, it is possible to reduce the functional units usage at the cost of having increased code size. Figure 3.7 illustrates this situation. Figure 3.7 (a) shows a loop in which overlapped execution of successive iterations on each cycle will cause overwriting of the contents of register a before it is last used. By preserving register a into a', via a copy operation, the next loop iteration is allowed to start in overlapped fashion, thus enhancing

throughput. However, in this example, the value of a must be preserved for two cycles, and the copy operation in the next iteration will overwrite register a'. Thus, another copy operation a' ← a" is necessary, as illustrated in Figure 3.7 (b).

The copy-avoidance mechanism in the IBM VLIW project unrolls the loop before the pipelining algorithm, and writes the unrolled loop iterations in static single-assignment form[32] so that each iteration uses its own registers. Therefore, iteration overlap may occur without requiring COPY operations to save intermediate values. The amount of loop unrolling is a function of the longest distance between the creation and last use of a value in a single loop iteration, and also of the iteration issue rate. The code is compiled twice. The first compilation is used to determine the amount of loop unrolling, and code is again submitted to the compiler after unrolling is performed. A drawback of this procedure happens for loops having conditional execution, and therefore may have distinct iteration issue rates on different paths through the loop. Unrolling loops prior to compaction and pipelining results is both increased compilation times and longer static code size.

An alternative approach inspired by the object code unrolling method of[54] is presented here. The idea is to unroll the object code on loop paths exhibiting chains of copy operations. Such chains of copy operations, in time, preserve a series of values of a renamed variable. The transformation is applied to the loop steady-state object code after compaction and pipelining has been performed. Loop unrolling is performed on each path through the loop steady state code. The amount of unrolling on each path is determined by the length of the longest chain of copy operations along that path. Unrolling of execution paths in the loop steady state code is followed by register renaming to eliminate copy operations. Figure 3.7 (c) illustrates the chain of copy operations copy a' ← a and copy a'' ← a' in the loop steady state VLIW instruction. That instruction holds the two COPY operations and the operations that define register a and use register a''. Copy removal substitutes that VLIW instruction for a series of three instructions that define and use registers a, b and c. Figure 3.7 (d) illustrates the loop execution after unrolling and register renaming is applied, thus removing the need for copy operations.

Since this procedure is applied to loop steady state code, some copy operations may still remain in the loop prologue. However, loop prologue is not likely to exceed resource

limitations. Furthermore, because the number of executions of the loop prologue instructions is much lower than the number of executions of the loop steady state instructions, simple solutions such as splitting loop prologue instructions to fit available resources may be applied without considerable loss in performance.

The algorithm works by identifying chain of copy operations. These chains take the form of:

```
L: a' <- op
a <- a'
f(a)
goto L
```

in the same instruction (assuming one-cycle steady-state code). In the above example, all operations in the same VLIW instruction are listed vertically.

By unrolling once the (only) path through the loop and renaming registers, the resulting code does not require copy operations. For example, after unrolling and renaming of the distinct incarnations of register a to a and b the above loop becomes:

```
L: a <- op
f(a)
goto L1

L1: b <- op
f(b)
goto L
```

Instructions in the loop steady state code contain the code that controls overlapped execution of multiple loop iterations. Programs in which the inner loop has multiple paths compound the the problem of unrolling to remove copies because consistency in the use of register names must be maintained. For example, let us consider the execution of pipelined iterations proceeding through distinct paths, and assume that iteration i assigns a value to program variable x in instruction L, and this value is reused in the same iteration two cycles later. All instructions reachable from L in two cycles (including L itself) must contain code that refers to the correct location for the value of x. For example, in a loop with three paths, a possible execution scenario is iteration 1 taking path 1, iteration 2 also taking path 1 and iteration 3 taking path 2. A software pipeline state corresponds to a number of overlapped loop iterations, e.g. iterations i..j taking respectively execution paths p_i,p_{i+1},..,p_j through the loop.

The unrolling algorithm must be such that register names are reused as much as possible, to avoid code explosion. The algorithm in [30], originally proposed as a form of software pipelining, may be extended for loop unrolling. The algorithm constructs a software pipelined scheduled by overlapping loop iterations that are in distinct phases of execution. The algorithm attempts to reduce the number of possible states in a pipelined loop execution while maintaining execution throughput. This algorithm identifies opportunities to reuse pipeline states, and therefore is applicable to reduce the number of loop unrollings in the removal of extraneous copy operations.

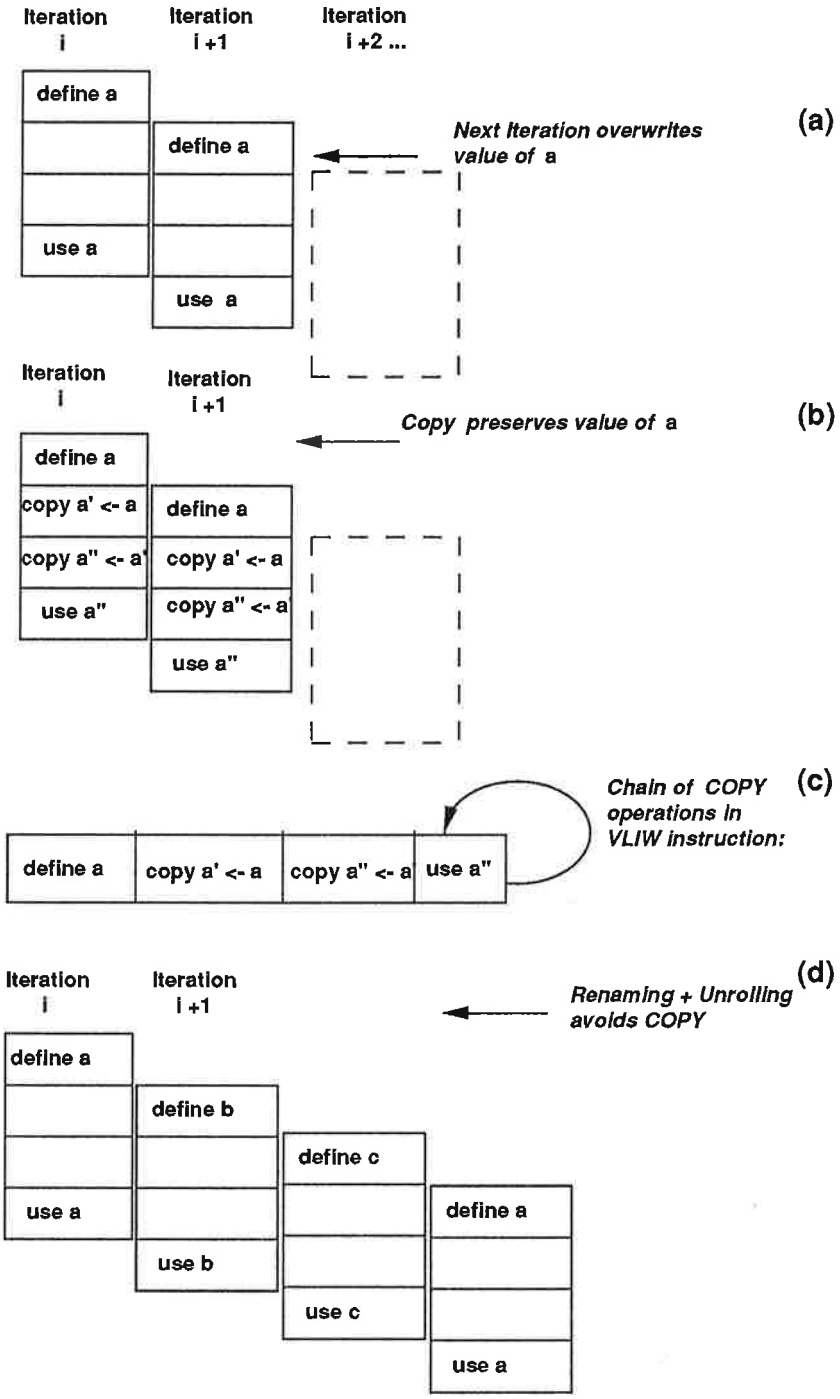


Figure 3.7: Avoidance of COPY operations Introduced by Pipelining

Chapter 4

Retargetable Optimizing Compiler

As part of this research, a retargetable microcode compiler has been developed. It is capable of generating code for a spectrum of specialized target architectures within the range of the VLIW architecture template, and implements the foregoing powerful intra-iteration and inter-iteration optimization techniques.

4.1 Fine-Grain/Microcode Compilation Task

A parallelizing compiler for a general purpose and widely used programming language such as C is necessary to help demonstrate the effectiveness of the specification optimization techniques. During development, the possibility of integrating the parallelization techniques into a previously existing compiler was investigated. The cost involved in adapting and enhancing the trace-scheduling compiler Bulldog[35] with percolation scheduling was considered. It was discarded because of the difficulty involved in handling and absorbing a highly complex piece of software that evolved over a long period of time and was developed by many researchers. Furthermore, a secondary goal of the compiler development is to create a framework, available to other researchers, to support and stimulate further research in compilation for architectures with fine-grained parallelism such as VLIW and superscalar architectures. Therefore, the choice of C as the compiler implementation language eases integration in UNIX platforms thus enhancing portability of the system, as opposed to the distinct dialects of LISP used in the Bulldog and IBM VLIW compilers. Presently, this compiler development is a joint effort with researchers led by professor Alex Nicolau at U.C.Irvine. Initial design and data structures definition was performed at CMU, based on

experiences from the research and development of the VLIW compiler at the T.J.Watson Research Center.

To reduce development effort, pre-existing compilation tools are used as much as possible. The retargetable compiler is composed of two parts: the front-end compiler, which translates the C source files into optimized code for an idealized RISC-like uniprocessor architecture, and a back-end parallelizing scheduler, which rearranges the operations of the RISC-like code for concurrent execution on the target VLIW architecture. This is illustrated in Figure 4.1.

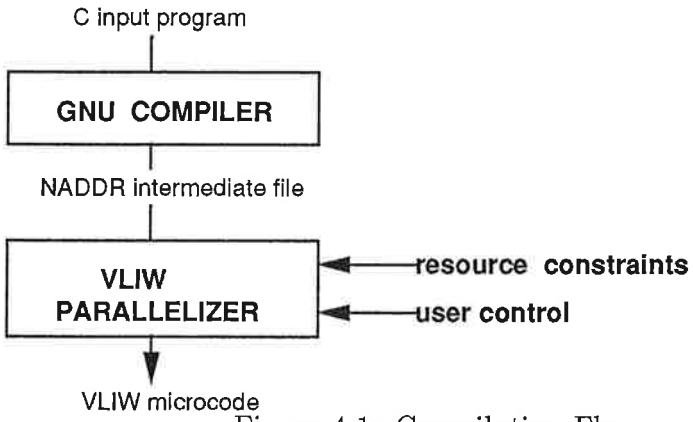


Figure 4.1: Compilation Flow

4.2 Compilation Framework

The front-end is based on the GNU public-domain optimizing compiler. This compiler is robust, supports code optimization, and generates code of good quality. At U.C.Irvine, it has been retargeted to generate code for the idealized RISC architecture. The processor description file for the target RISC architecture is adapted from the GNU configuration file for the MIPS processor[79]. The standard scalar optimizations[2] are performed by the GNU front-end.

The intermediate format is inspired by the NADDR[35] format used in the Bulldog trace-scheduling compiler. The general form of an intermediate-code operation is a list

```
(OPCODE destination arg-1 arg-2 ... arg-N )
```

The first element in the list is the operation specifier. The second element designates the destination of the result of the operation. The remaining list of elements designate the required input arguments for the operation. Except for jumps and memory LOAD and STORE operations, the argument designate either registers or immediate constants. Control flow operations include jumps, immediate and conditional. Some assembly-level pseudo-operations are used to designate instruction labels and procedures. The C source code, and the corresponding NADDR code for a simple program to find the minimum of an array are illustrated in Figure 4.2.

```
int min;

min (A)
int A[];
{

for(i=0; i< N; i++)
     {
     if ( A[i] < min)
       min = A[i];
     }
}
```

```
(PROC_BEGIN   LOOP

(LABEL LOOP)
  (IGE $cc0 $i $n)
  (IF  $cc0 (LABEL exit))
  (IVLOAD  $u 100 $i)
  (IGE $cc1 $u $min)
  (IF  $cc1  (LABEL L5))
  (IASSIGN $min  $u)
(LABEL L5)
  (IADD $i 2 $i)
  (GOTO (LABEL LOOP))
(LABEL exit)
  (IGOTO $31)
  (PROC_END LOOP)
)
```

Figure 4.2: Example: min.c Source Code and Intermediate Code

The retargetable compiler is structured to have two modes of operation: a batch mode and an interactive mode. In batch mode, the user creates a specification file with the sequence of optimizations and parallelizing transformations to be applied to the code. The use of

the specification file been structured to allow selection between alternative parallelization techniques without the need for the creation of multiple versions of the compiler. This allows easy experimentation with distinct compilation techniques in a single integrated framework.

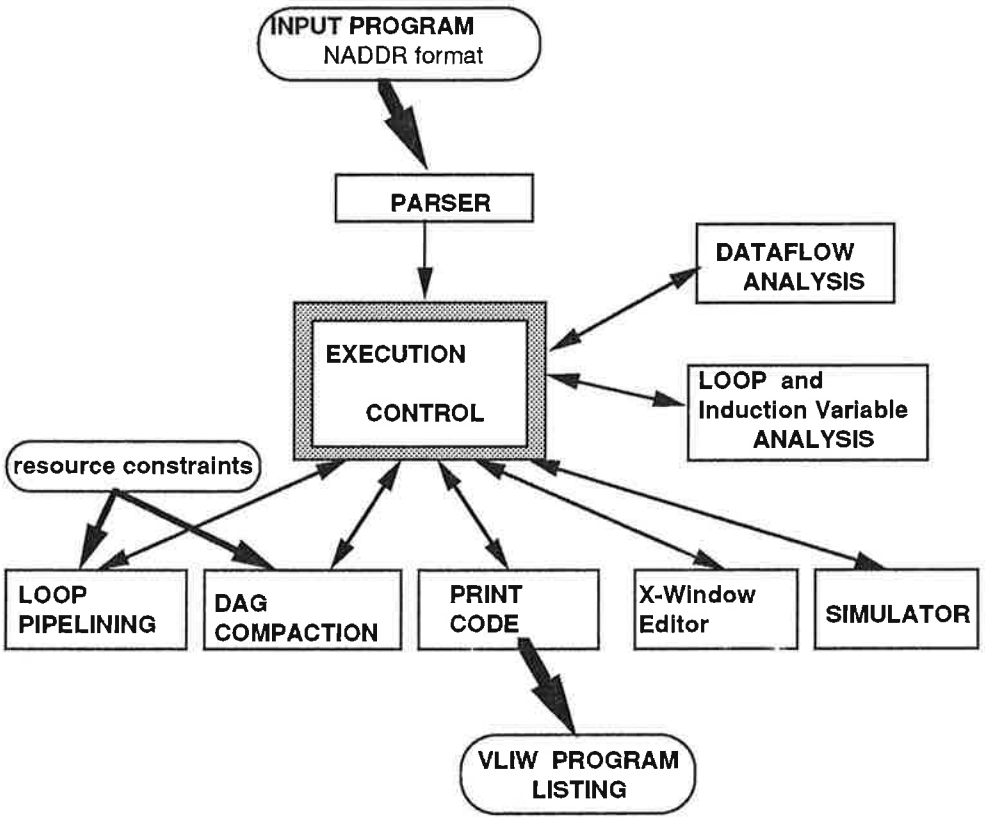


Figure 4.3: Structure of Control Flow in VLIW Parallelizer

The interactive mode is used to display, under user control, the program at distinct stages of parallelization. It presents the program's control flow graph on the screen and allows interactive inspection of the contents of each VLIW instruction. This feature has been extensively used during development of the compiler itself, to help debug the compiler by visual inspection of the generated code. There is also provision, not fully implemented, to interactively control the application of the parallelization optimizations: operation motion, compaction of the selected section of code via Percolation Scheduling, and loop pipelining. These features allow the use of a mouse and graphics interface to choose a parallelization transformation, e.g. loop pipelining, and select individual operations or sections of code to which the chosen transformation is applied. This feature, when fully implemented, should

allow the compiler to be used as an interactive microcode editor similar to the one described in [7]. It is expected that with this feature the user can enhance parallelization by performing the optimizations interactively. Furthermore, the interactive graphics interface is also useful for research on compilation techniques, since it may be used to help identify good heuristics for parallelization with constrained resources. Presently, the graphics interface displays the control flow graph of the parallelized program, along with detailed description of selected instructions. There is provision to extend it to display information on resource utilization.

The input to the system is the set of C source files of the application program. The user invokes the front-end GNU C compiler to generate optimized code for the idealized RISC architecture, producing a file with the NADDR intermediate code. The parallelizing back-end is subsequently invoked. A specification file, called `standard.ops`, contains a list of procedures to be invoked. In the implementation, each of these procedures receives as argument a pointer to the global data structure describing the program. A file called `vliw.resources` describes the available resources. Presently, the user specifies the maximum number of ALU operations and the maximum number of conditional jumps allowed in each VLIW instruction. There is provision to provide a detailed description of resources by distinguishing between resources such as adders, multipliers, etc.

A number of parallelizing transformations are then performed under user control via the specification file. Conditional execution of operations may be disabled by an specification argument to support VLIW implementations with simpler sequencing mechanisms. With this option activated, all operations that are scheduled in a VLIW instruction are executed unconditionally. The parallelizing transformations update the global data structure. The user may choose to apply pipeline scheduling to all loops, or to apply percolation scheduling on a single iteration of these loops. In the figure, intra-iteration optimization is referred to as `DAG Compaction`. The loop body is viewed as a Directed Acyclic Graph, and Percolation Scheduling transformations are applied within the loop boundary. Percolation Scheduling of the code between loops may also be invoked by a command in the specification file. The user may elect to visually inspect the code by invoking graphics display interface at any point between transformations. Figure 4.3 illustrates the above alternatives. In the Figure, the first three stages of parsing, dataflow analysis, and loop and Induction Variable

analysis are mandatory. Once the analysis is complete, the user may select alternative code parallelization techniques.

4.3 Graphics Interface

When invoked, the graphics interface starts by displaying a window with the program control flow graph. This is called the program window and is identified by the window name `vliw` at the top, as illustrated in Figure 4.4. The program window has the title "Code View" right above a horizontal scroll bar. Scroll bars on the sides of the window allow viewing of large programs and large instructions.

Each instruction is represented by a unique label, assigned by the program. The mouse selects instructions to be displayed in more detail. By using the mouse to click on an instruction in the program window, the user opens a window which displays the selected instruction in detail. This window, called an instruction window, is identified by the "instr" window name at the top. The instruction window has the title "Instruction" followed by the unique instruction identifier above the horizontal scroll bar. Instructions are displayed in a tree-shaped format. The unique label for the instruction is presented, underlined, at the top. Each instruction which contains conditional jumps and conditionally executed operations is displayed as a tree. The code of each operation, its destination and operands are presented.

The tree-like display of an instruction is not a canonical representation. Instructions that contain conditionally executed operations may be represented in several alternative ways. The instruction tree displayed corresponds closely to the internal representation of the instruction. The ordering of operations on a path through a given instruction is one that bears correspondence with the initial ordering of operations in the original NADDR program, and reflects the way in which Percolation Scheduling transformations are used to populate the instructions. The adoption of this representation is helpful to help debug the compiler. It suffers the drawback of possibly presenting multiple times an operation which is present on multiple paths through the instruction. Alternatively, a canonical representation for each instruction is considered. Its use has been discarded because the savings in memory use and execution time did not offset the benefits of helping debug the highly complex parallelizer.

The graphic display of the program of Figure 4.4, immediately after parsing and before

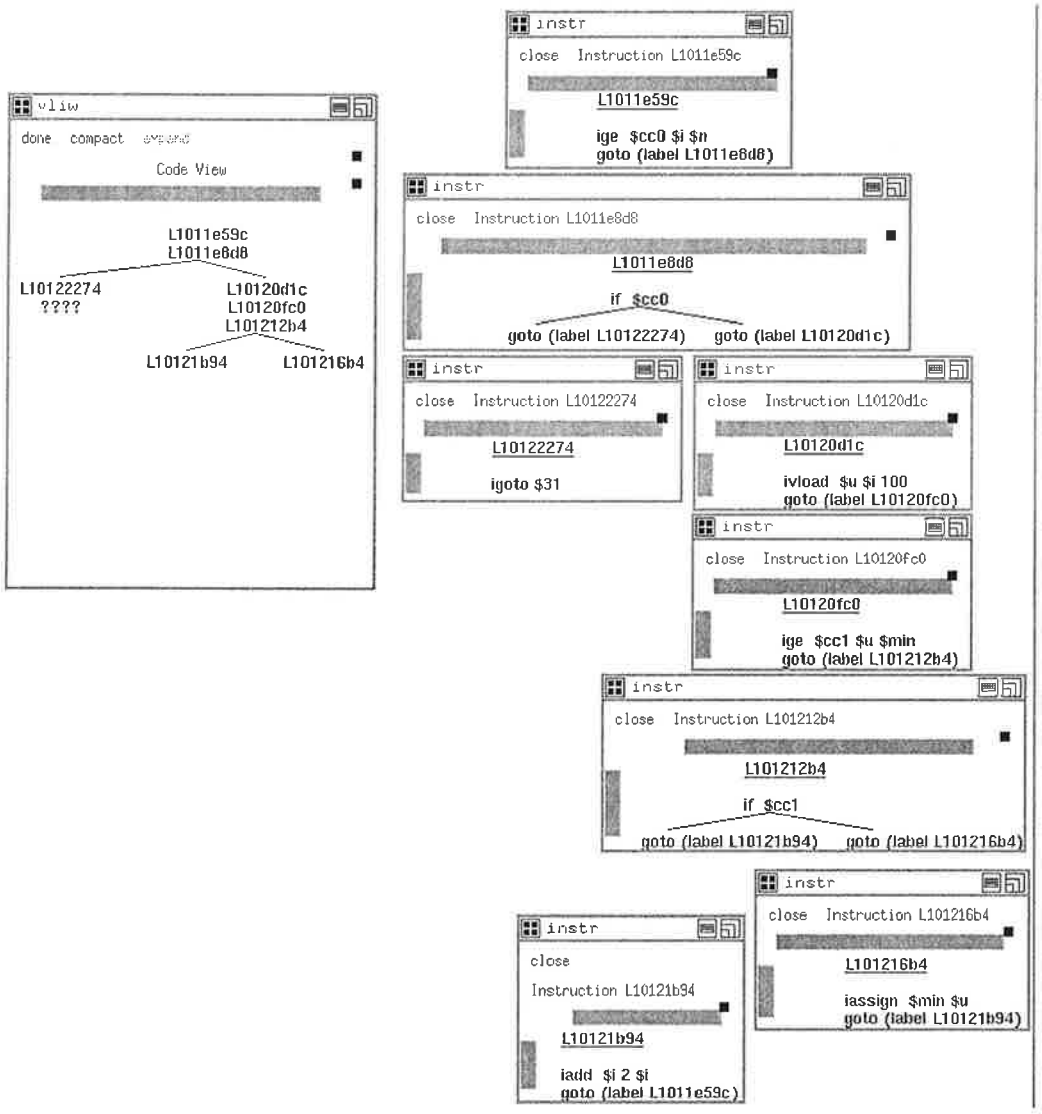


Figure 4.4: Graphics Interface - Program and Instruction Windows for min.c

any parallelizing transformation has been applied, is illustrated in Figure 4.4. This figure has been obtained via a screen dump of the graphics interface. The program window is shown at the left, all instructions are selected, and instruction windows are placed on the screen accordingly to execution flow. The instruction window at the top represents the entry instruction of the program and successive instructions are placed below it. Each instruction

contains a single RISC-like operation. For example, Instruction L10122274 executes the `igoto $31` operation. This operation implements the procedure return, via an indirect jump to a register.

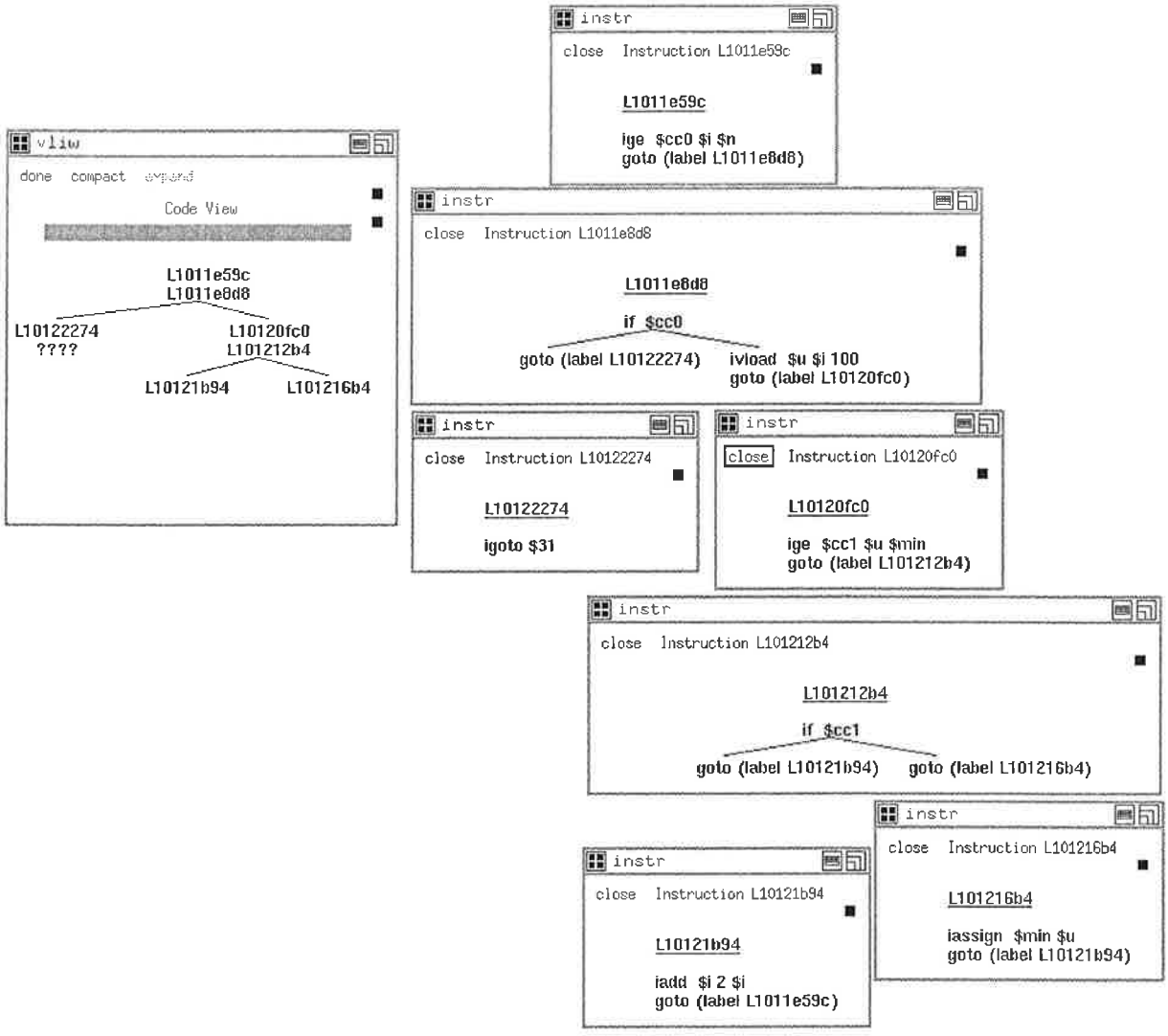


Figure 4.5: Graphics Interface - min.c after first Move-op

Figure 4.5 displays the same program after the first Percolation Scheduling transformation, in this case *move-op*, has been performed. Instruction L10120d1c originally contained a

memory load operation (`IVLOAD $u 100 $i`). This operation loads the contents of memory location addressed by `$i` plus offset `100` into register `$u`. In other words it may be described as `$u = Memory[ 100 + $i ]`. The load operation is moved to the preceding instruction, Instruction L1011e8d8. After the move, Instruction L10120d1c became empty and has been removed from the program. After the move, the load operation is executed conditionally by Instruction L1011e8d8.

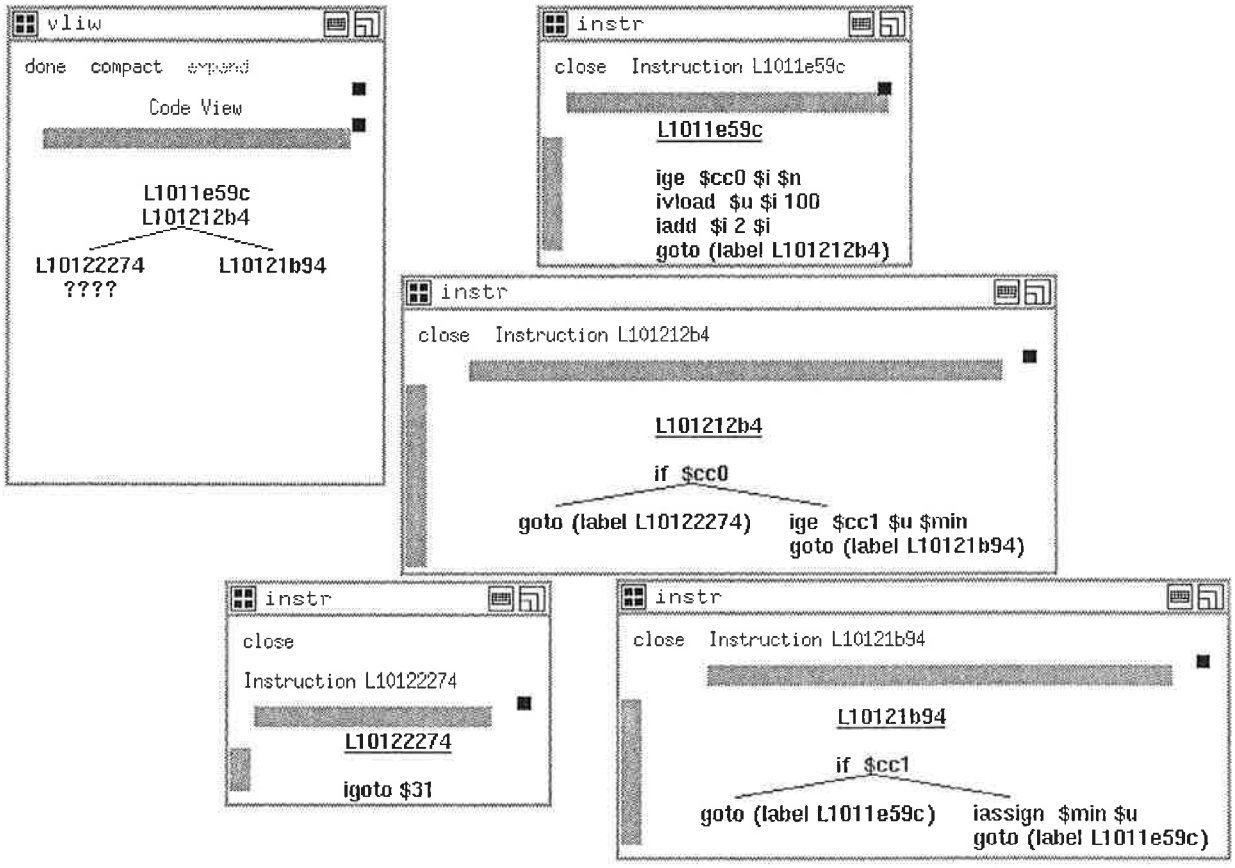


Figure 4.6: min.c after Intra-iteration Optimization

Figure 4.6 illustrates the same program after intra-iteration optimization has been applied. Here the compiler is allowed to schedule the operations belonging to a loop iteration as soon as possible, without concern with resource constraints. For one loop iteration, the longest path through the loop body happens when a new value for the minimum of the array is found. It has been reduced from 7 instructions to 3 instructions. The shortest path through the loop body has been reduced from 6 instructions to 3 instructions. The speedup achieved for this code ranges between 2 and 2.3 depending on which path through the loop is more frequently taken. The parallelized code requires a VLIW CPU with the ability to perform 2 ALU operations, one memory load, and a conditional jump concurrently.

Figure 4.7 illustrates the program after inter-iteration optimization. Instructions L1011e59c and L101212b4 represent the pipeline prologue. These instructions start execution of the first and second loop iterations in overlapped fashion. Instruction L10120d1c represents the loop steady state code. Here a throughput of one iteration per cycle is obtained. This happens because on some execution paths Instruction L10120d1c jumps back to itself. In this example, exit from the loop does not require epilogue instructions because future loop iterations that are started in overlapped fashion do not alter variables that are required outside of the loop. This assumes that the only live variable after the loop is `$min`. The parallelized code requires a VLIW CPU with the ability to perform 3 ALU operations, 1 memory load and 2 register-to-register copy operations concurrently. The speedup over the RISC code ranges between 6 and 7. It should be noticed that the sequencer is also considered a resource while computing the amount of parallelism in the VLIW architecture. The NADDR original code contains explicit `GOTO` operations, whereas the VLIW instructions carry a next-address field.

In Figure 4.6, it is seen that variable $u is defined by a given loop iteration in its first cycle of execution and is last used by that iteration two cycles later. Therefore, register renaming is required when overlapped iterations are started on every cycle. As a new iteration starts, it attempts to overwrite the value of `$u`, and therefore value of `$u` must be preserved. The compiler renames the destination register in the operation attempting to overwrite `$u`, and the starting iteration defines a new variable (in the example the `ivload` operation defines variable `$u'`). In the following cycle, the value just defined is copied into place by the

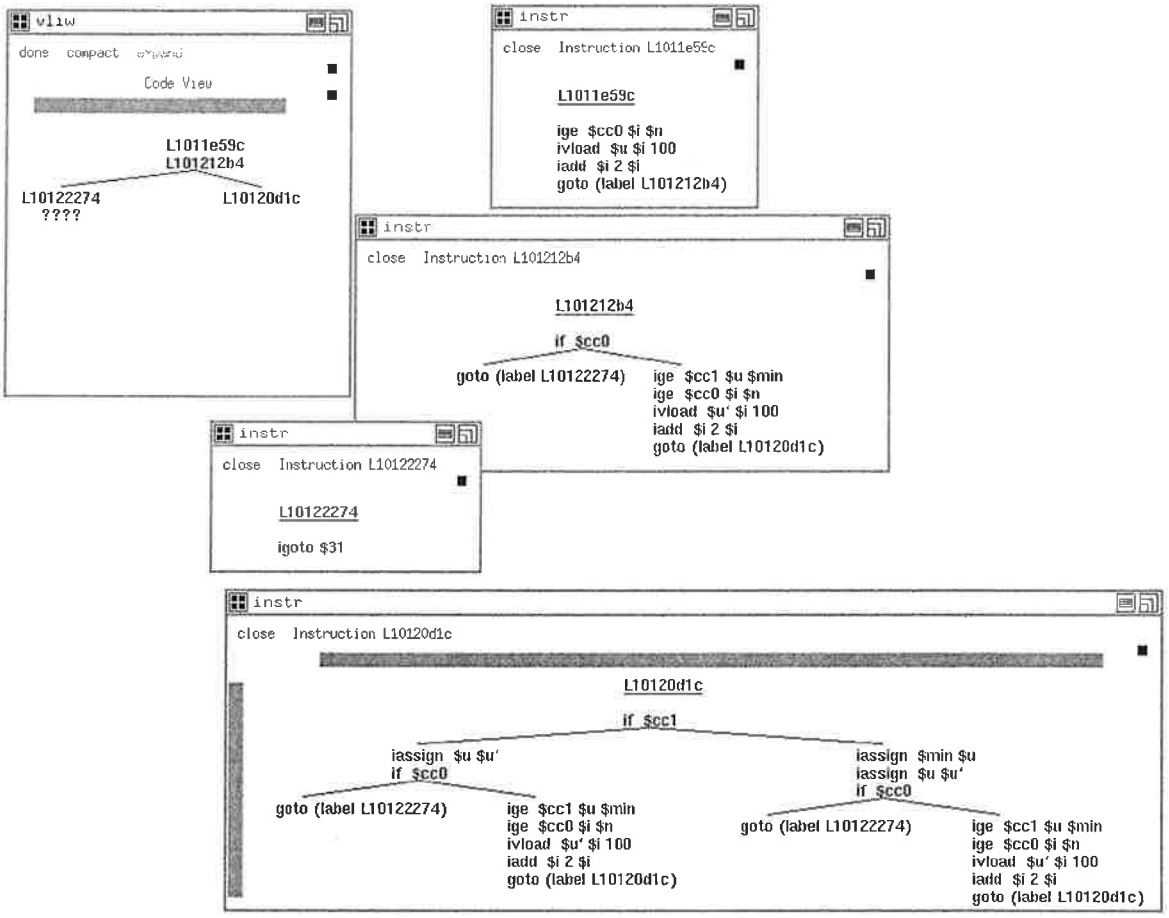


Figure 4.7: min.c after Software Pipelining inter-iteration optimization

copy operation `iassign $u $u'` and the old value of `$u` is last used. The copy happens concurrently with the last use of the original value.

4.4 Code Parallelization Techniques

The VLIW parallelizing back-end is composed of 25 thousand lines of C code. It reads a program in the RISC-like intermediate format and organizes RISC operations into VLIW instructions. The features implemented in the back-end compiler are:

- enhanced percolation scheduling with register renaming
- software pipelining
- computation of unifiable-ops with renaming
- memory disambiguation
- compilation with resource constraints
- an X-window based interactive code editor

A number of the standard dataflow analysis to support parallelization are also implemented. These include live variables flow analysis, reaching definitions, loop analysis and detection of induction variables. The back-end initially parses the NADDR-like intermediate code and creates a global data structure describing the input program. Subsequently, dataflow analysis is performed. The information acquired during this analysis is stored into the global data structure. This is followed by loop analysis and detection of induction variables to allow memory disambiguation. This completes the analysis phase, and parallelization transformations ensue.

The files which compose the back end are classified, according to their function, in the following categories:

- Core Percolation Scheduling
- Guidance Layer Routines
- Computation of Auxiliary Dataflow Information
- Loop Analysis, Induction Variable Analysis and Disambiguation
- User Interface
- Auxiliary Routines

The Core Percolation Scheduling routines *move-op*, *move-cj*, and *delete* are contained in files `rmovop.c`, `movecj.c` and `delete.c`, respectively. The first two routines receive as arguments a pointer to an operation, the instruction where it is located, and a pointer

to the destination in the preceding instruction. The data-dependency conditions for the move are checked and, if satisfied, the move is performed. The program data structure is updated accordingly. The routine for the *delete* transformation receives a pointer to an instruction, checks whether it is composed entirely of no-operations and in that case removes the instruction from program.

Guidance Layer routines, contained in file `rmaxcomp.c`, use the core Percolation Scheduling routines to achieve inter-iteration and intra-iteration optimization. The *maxcomp* routine performs intra-iteration optimization of a loop body, under resource constraints. It uses routine *fill-instr* to fully populate a target VLIW instruction until resource constraints are exceeded. It heuristically selects operations in the `unifiable-ops` set of the target instruction and uses *migrate* to perform the move. *fill-instr* uses the *migrate* transformation, which applies a series of core PS transformations to bring a designated operation to an instruction. Similarly, routine *pipeline-ilist* receives as argument the list of operations in the loop body and implements the Enhanced Pipeline Scheduling transformation. It proceeds by using *fill-instr* to fill the instructions at the entry of the loop and changing attributes in the data structure that show the position of the loop back edges.

Computation of Auxiliary Dataflow Info in file `live.c` contains routines that perform live variables dataflow analysis. Similarly, Reaching Definitions flow analysis is performed by routines in files `reach.c`, `attreach.c`, `genreach.c`, `killreach.c`. These routines implement the standard algorithms presented in [2]. The computation and incremental update of unifiable-ops information, with and without renaming, is found in `runifiable.c`.

Loop Analysis is found in files `loop.c`, `bdloops.c`, `backedge.c`, `dominat.c`, `innerloop.c` and `dfnumber.c`. These files contain routines to identify loops in the program, and construct a list of the instructions in an inner loop. This is performed by computing dominators[2] in the control flow graph of the program. Induction Variable Analysis, implemented at U.C.Irvine and contained in file `getivs.c`, uses algorithms from [60] and [2]. The result of Induction Variable Analysis is required to initialize symbolic derivation of loop induction variables, which is required to perform Memory-Reference Disambiguation.

Memory-Reference Disambiguation is contained in files `fold.c`, `depend.c`, `normadd.c`, `normmul.c`, `interf.c`. The key routine is *check-dependency*, which receives two operations

including memory load and store, as argument and informs whether there is a data dependency. A data dependency exists if the result of the first operation, which may be a `store` into memory, may be an input argument of the second operation, possibly a memory `load` operation. User Interface routines are contained in files `executive.c`, which interprets the batch control file `standard.ops` and activates appropriate routines as specified. The X-Window graphics interface, initially developed by Dan Nydick of Carnegie-Mellon University, is contained in files `editcode.c` and `Graphic.c`. Output of program listings is performed by routine *pr-program* in file `print.c`.

The parser routines translate the input NADDR code into program representation in the global data structure. File `stlex.y` contains the `yylex` driver routine for the lexical analysis routines. File `opcodes.c` handles NADDR instruction and builds the program data-structure

Auxiliary Routines to traverse the program data structures, update links between elements of the data structure, allocate and release dynamic memory are contained in file `aux.c`. Sets are represented as bit vectors. Routines for set operations such as set creation, intersection and union are found in file `sets.c`.

4.5 Data Structures

The program data structures are allocated dynamically, under programmer control. This approach may save memory at run time and result in potentially faster compiler execution. The key data structures in the compiler are those describing VLIW instructions and their operations. The program is represented as a linked list of instructions. This representation contains information about the successors and predecessors of each instruction, and corresponds closely to the Control Flow Graph[2] of the program. Data-dependency information is not stored explicitly in the data structure, and is computed dynamically as required. The description of each instruction contains pointers to the operations it performs. Auxiliary data structures are used to represent arguments of operations, assign unique identification numbers to operations (to allow bitsets of ops) and loop identification numbers. The key data structures of the compiler are defined in file `microcode.h`.

Figure 4.8 illustrates the data structure nodes that represent instruction and operations.

Each instruction in the VLIW program is represented by an Instruction type node. The program is represented by the linked list of Instruction nodes. An Instruction node contains links to its predecessors and successors in the list. Other fields in an Instruction node are pointers to bitsets that represent dataflow information: the sets of live variables and the sets of definitions that reach the instruction. Similarly, each operation is represented by an Operation node. Operation nodes are linked in tree structures, reflecting the tree representation of the conditional-execution VLIW instruction. Each Instruction node contains a pointer to its corresponding tree of Operation nodes. Furthermore, there is a path descriptor node associated with each path through the instruction. The path descriptors contain summary information about each execution path of the instruction. For a given path, the path descriptor node points to the successor instruction on that path in the control flow graph, and contains the sets of values that are read and written on that path. This summary information is used to compute data dependency during Percolation Scheduling. Furthermore, a Path Descriptor node describes all copy operations on the corresponding path. This is used during the Percolation Scheduling transformations to allow the motion of operations whose input arguments are defined by copy operations. If the operation being added to a path uses an input argument that is the destination register of a copy operation, it is changed to read directly the source of the copy operation. This is done by scanning the list of copy operations on a path.

An Operation node contains the unique id associated with the operation, the operation type and opcode. Operation types are ALU operation, control flow, or dummy. Dummy operation nodes are used as list headers. An operation node points to a data structure describing each argument. The sets of variables read and written by the operation are also included, to allow quick checks of data dependency via set intersection. Memory operations also include a pointer to a data structure describing the symbolic derivation of the target memory address. The length of the dependency chain, in a loop iteration, starting at the operation is also included. It is used in the heuristic choice of operations to move in case of constrained resources. Depth-first numbering of the operations in the original NADDR program is also used as a heuristic. This information is also used by the heuristic to choose operations to move[59].

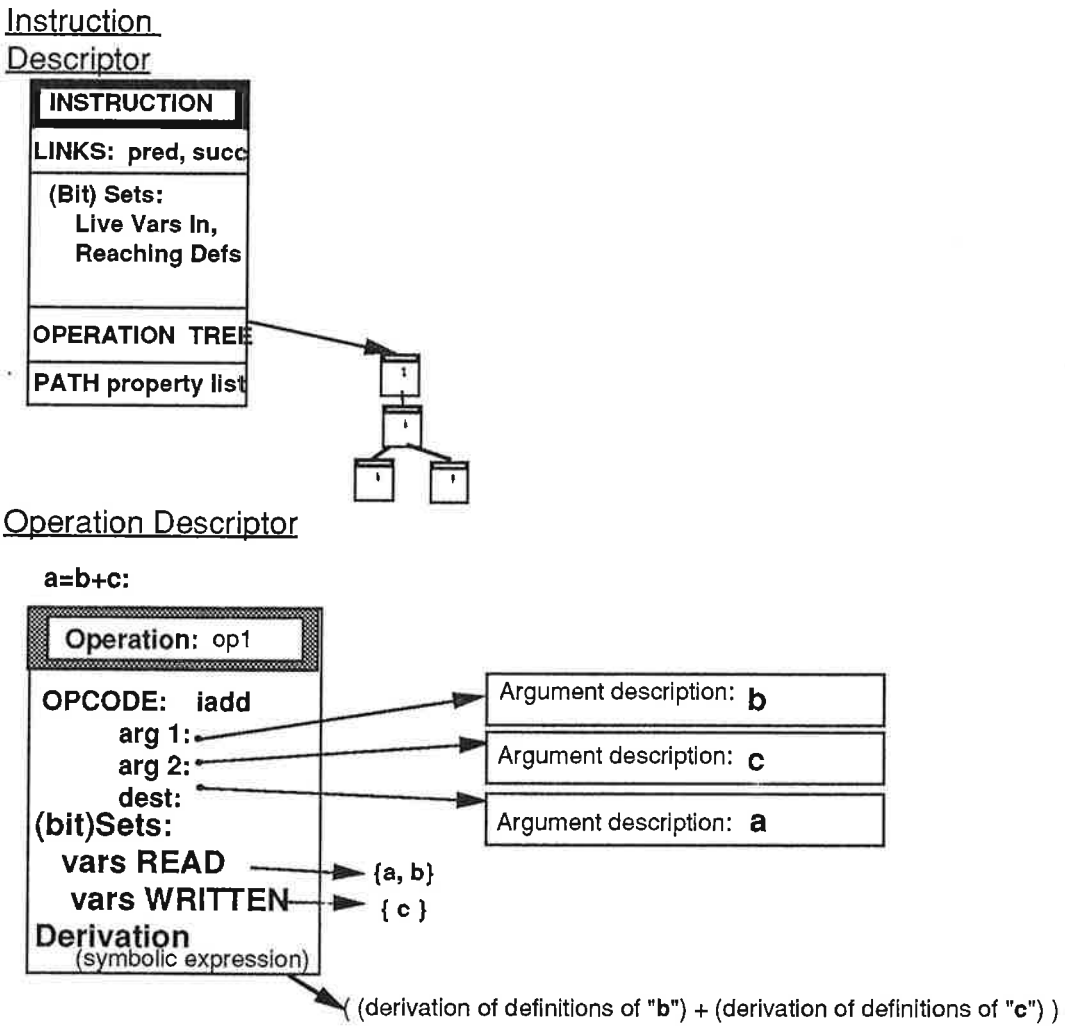


Figure 4.8: Data Structures - Instruction and Operation Nodes

Figure 4.9 illustrates the data structures corresponding to a VLIW instruction that con-

tains two operations. A textual description of the instruction is presented at the top. One of the operations, `a = b + c` , is executed unconditionally. The second operation, `i = i + 1` , is only executed if the condition code `cc1` is true. In the figure, the Instruction node is presented at the left, with a pointer to the associated tree of Operation nodes and the list of Path Descriptor nodes. Dummy nodes used as list headers are not shown. Here this list has two elements, corresponding to the two possible values of the conditional jump.

4.6 Compilation Effectiveness

This section presents compilation time and speedup obtained for a number of small benchmarks, as illustrated in Figure 4.10. Times are measured in CPU seconds on a DecStation 5000. The benchmarks include Livermore loops 1 to 12 and a number of selected small benchmarks which have previously been reported on the literature[31]. The fourth column indicates the amount of resources for execution of the compiled VLIW program. The first number is the number of operations in the widest VLIW instruction, and the second number if the maximum number of conditional jumps in an instruction. Speedup is given by the ratio between the number of execution cycles measured by simulation of the serial code to the number of cycles for VLIW execution. All programs are compiled to achieve maximum speedup. As expected, compilation times are relatively large in comparison to a typical workstation compiler. This is a consequence of the additional number of optimizations and data flow computations in a parallelizing compiler. Furthermore, it is seen from Figure 4.10 that the increase of compilation times is correlated with increased size of the input program and with the amount of speedup obtained. A great number of Percolation Scheduling transformations are involved when a large amount of parallelism is uncovered, e.g, in the case of a large inner loop that achieves an Inter-Iteration optimized throughput of one cycle per iteration. Here, this happens for Livermore loops 1 and 7. Another cause of increased compilation time is the presence of complex array index expressions. This requires extra computational effort for memory-reference disambiguation. Finally, removal of extraneous copy operations is not applied, and the aggressive scheduling heuristics result in the large number of copy operations for some of the examples. For example, the steady-state code of Livermore Loop 7 contains 34 such COPY operations.

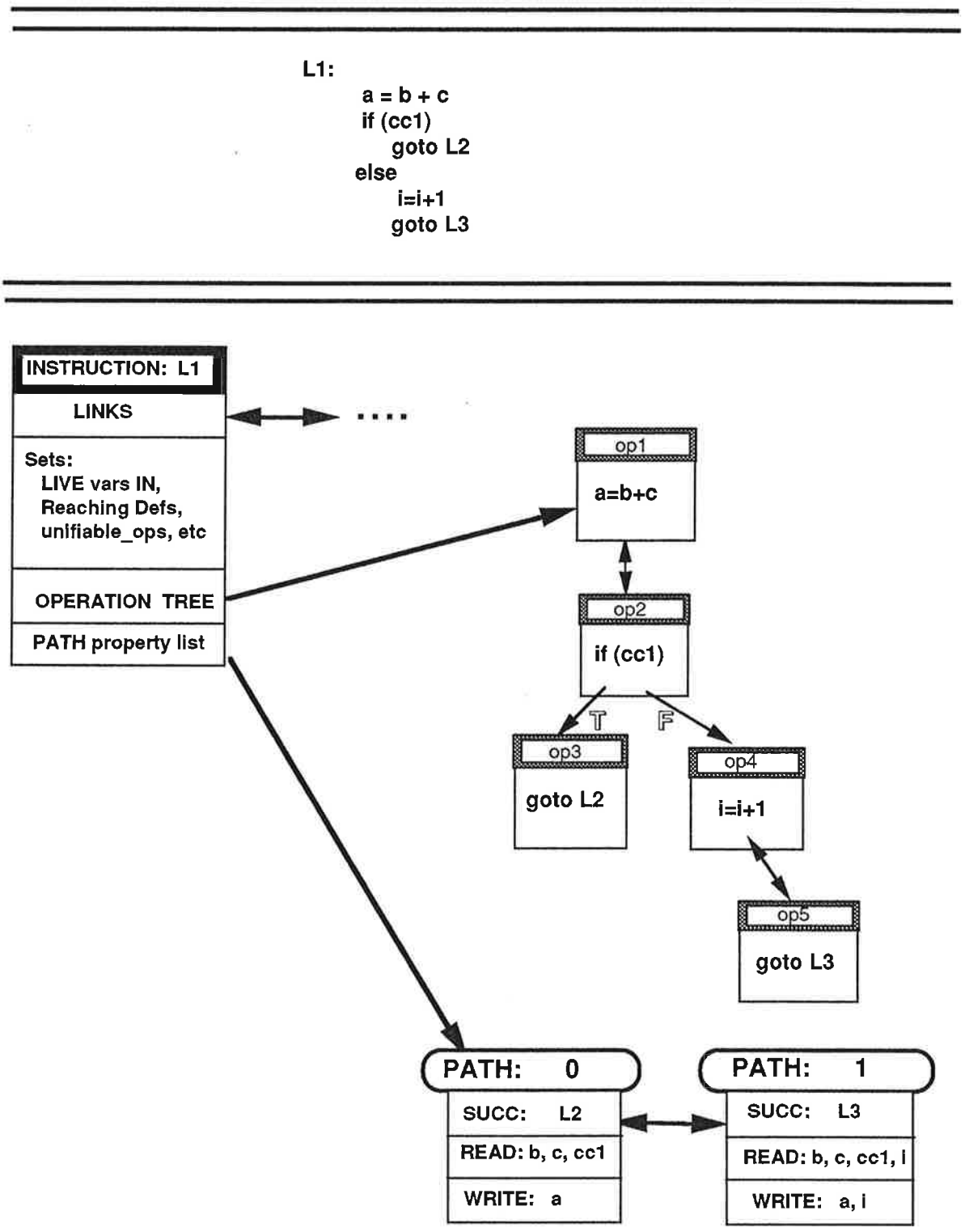


Figure 4.9: Data Structures - Example Instruction

Benchmark	Size (NADDR lines)	Compilation Time seconds	Resources (ALU, Cond. Jump)	Speedup
LL1	31	21.2	(21,1)	13.5
LL2	55	1.8	(11,3)	3.0
LL3	26	1.1	(8,1)	8.9
LL4	45	1.5	(6,1)	3.0
LL5	25	1.7	(9,1)	5.5
LL6	46	2.1	(10,3)	3.6
LL7	48	24.9	(65,1)	23.6
LL9	80	5.6	(9,1)	3.6
LL11	25	2.7	(10,1)	8.9
LL12	23	1.5	(10,1)	8.9
inserc	31	0.9	(11,2)	8.8
minmaxc	32	1.1	(21,15)	12.7

Figure 4.10: Compilation Effectiveness: Speedup, Resources, Compilation Time

4.7 Compiler Status and Future Direction

Increased compiler code size is a consequence of the choice of C as the implementation language. In our experience, a LISP implementation with roughly the same capabilities requires one half or one third of the number of code source lines. This agrees with the claims about the ease of implementing experimental compilers in LISP of [35]. In a LISP implementation, memory allocation and deallocation is performed by the runtime system. This results in an increased cost in terms of compiler execution time and memory space requirements. The chosen approach of programmer-controlled memory allocation has led to bugs during compiler development primarily because of programming errors in pointer references and maintaining consistency in a highly interconnected the data structure. However, it is believed that the resulting savings in execution speed and memory space requirements should make it the approach of choice for an industrial-strength compiler implementation.

The VLIW compiler is presently functional, and the full cycle from source C code to NADDR intermediate code, and then to optimized VLIW microcode has been exercised and verified by simulation on some examples. Presently unimplemented are the capabilities to handle multiple procedures, procedure calls and multiple input files. There is plan to extend the compiler with techniques to handle pipelining of outer loops[61]. The compiler has been extended at U.C.Irvine with a Percolation Scheduling technique capable of supporting

pipelined and multi-cycle latency ALU operators[65], and with an intra-iteration scheduling technique that does not require unifiable-ops information. This scheduling technique is directed by the mobility of operations on a critical dependence path. The inter-iteration optimization routines may convert the input program from a reducible flow graph into an irreducible one, thus preventing further applications of loop analysis. There is plan to enhance the present loop detection analysis routines to handle programs with non-reducible flow graphs[2]. Our Induction Variable detection algorithm presented in Chapter 3 is first implemented in the IBM VLIW compiler, and there is plan to extend this compiler which implements the traditional algorithms from [2].

Chapter 5

Implementation Optimization

The specification optimization phase produces highly parallelized application microcode which in turn serves as the optimized, in terms of code performance, specification for the ASP architecture. The number of functional units and the number of memory banks needed to support the parallelized code execution are specified. However, the straightforward mapping of this specification to hardware may not produce the most efficient implementation. Cost and performance factors reduce the justifiable usefulness of the canonical VLIW to execute a single application. The global shared register file has a cost that grows proportionally to the square of the number of ports[13]. Furthermore, for a given implementation technology, the increased fan-out of memory elements in the register file causes an increase in the minimum cycle times achievable. This chapter describes algorithms used during the Implementation Optimization phase to efficiently allocate hardware resources to support execution of the optimized application microcode without impairing code performance.

5.1 Implementation Optimization Procedure

The architecture template used during specification optimization assumes a global register file. Direct implementation of a global register file in hardware may be too costly or too inefficient. However, for a given application, it is possible to allocate data into distinct single-ported register files such that concurrent accesses, as required by the application, can still be performed. During the Implementation Optimization phase, the register utilization in the optimized microcode is analyzed to produce a more efficient implementation involving multiple single-ported register files.

The data section implementation template uses a number of single-ported distributed register files and sparse connectivity(see Figure 2.7). Thus, it is likely to have lower cost than the canonical VLIW because of lower connectivity. It also has faster cycle times due to the use of single ported register files and lower fanout load on register file elements.

The problem of implementation optimization involves finding the appropriate number of distributed register files, their sizes, and their connections to the functional units. Heuristics to allocate variables into multiple distinct single-ported register files are presented in [42]. These heuristics are only applicable to straight-line microcode without branches. The algorithms in this chapter are applicable to more realistic programs with arbitrary flow of control. Furthermore, for some classes of programs, it is demonstrated that these algorithms are capable of allocating the globally optimum (minimal) number of register files.

The total number of functional units is constrained by the designer during Implementation Optimization. This, in turn, constrains the number of possible functional units a register file may connect to (vertical busses in Figure 2.7). To further reduce possible interconnect cost, it is necessary to pack variables in the smallest number of single-ported register files that will still allow concurrent accesses as required by the microcode.

Implementation Optimization first allocates values into register files, in order to minimize their total number (horizontal busses in Figure 2.7). Then, operations are assigned to the functional units. This assignment determines the required interconnection points between functional units and register files: a connection is created between a given register file bus and a functional unit's input (output) if the functional unit reads(writes) a value from(into) that register file.

Operations allocation should be performed so that the total number of connections between register files and functional units (dots at crosspoints in Figure 2.7) is reduced. There are two objective functions to be minimized during the process of allocating operations to functional units:

- *Total number of connections to busses*: each functional unit connection to a bus is either a multiplexer input, or a tri-state buffer. Therefore, by minimizing the number of connections, total hardware cost is reduced.

- *Bus loading:* Bus load is proportional to the total number of connections to a given

bus. The final clock cycle of the architecture depends on the maximum delay on each bus, which is strongly affected by the number of connections. Therefore, to maximize speed, a good balance must be found to avoid over-loading a single bus. Unbalanced organizations with less interconnection points may have longer cycle times, even though total hardware cost is reduced.

The allocation process to simultaneously minimize the above objective functions is a difficult task, and is related to another research project at CMU. It is a separate research task to find efficient heuristics to solve this problem, as well as an adequate formulation to allow adequate expression of the distinct objective functions. The Least-Cost Clique Partitioning Procedure (LCCPP) of Springer[77] may be adaptable to perform this task. A second order consideration may be cost reduction on the functional units. For example, if all operations assigned to a given functional unit are additions, then that functional unit may be implemented by an adder, instead of using a general-purpose functional unit. These second-order cost considerations are likely to be of lesser importance for our target application area of board-level ASPD. However, this possibility should be considered if the techniques in this thesis are extended for application in large chip design.

In this work, a guided locally greedy approach is adopted. The least-cost allocation is approximated on an instruction-per-instruction basis. Allocation is performed first on instructions that are likely to be the most constrained in terms of interconnection requirements. These instructions are identified from our knowledge of the algorithm for inter-iteration optimization. In a software pipelined loop, the instruction sequences for pipeline startup (prologue) and pipeline wind-down (epilogue) contain a subset of the operations in the loop steady state. Therefore, operation allocation is performed first on instructions belonging to the loop steady state. This allocation is likely to be a good match for loop prologue and epilogue instructions.

5.2 Register Files Allocation

For the target application, the final architecture must be capable of emulating the canonical VLIW's global shared register file with a minimum number of single-ported register files. In the implementation template, it should be noticed that the interconnection of N register

files to M functional units may have cost proportional to $N * M$. Furthermore, each register file creates a logical "bus" and, therefore, it is important to reduce the total number of register files used. Register files allocation allocates values into positions in single-ported register files. In the implementation template, the execution of each microcode instruction proceeds through the three phases of: reading input values from register files, execution of the operations, and writing the results back into the register files. Because register files are single-ported, distinct values that are accessed concurrently either for reading or writing by some microcode instruction cannot be allocated in the same register file.

At the Implementation Optimization stage, the optimized specification for the target application is expressed in the form of microcode for a canonical VLIW architecture with a multiported register file. Microcode operations in the optimized specification are expressed in terms of symbolic references to register file positions in the canonical VLIW. In the following discussion, positions in the canonical VLIW's register file are referred to as *variables*. A microcode operation `x := y + z` is said to *define* variable `x` and to *use* variables `y` and `z`. If there is a possible execution path in the program such that the value of variable `x` defined at instruction M_1 may be used as an operand for some operation in instruction M_2, the definition of `x` in M_1 is said to *reach* M_2.

For example, assume that canonical VLIW register file position \$1 is assigned a value by some microcode operation $O1$ and this value is subsequently last used by a distinct operation $O2$; and later, the same position is reused to hold a value created by operation $O3$ which is last used by operation $O4$. Here, $O1$ *defines* a value of \$1 which is *used* by $O2$ and $O3$ *defines* another value of \$1 which is *used* by $O4$. In the following discussion, the pair {definition,use} of variable \$1 is called a *name* of \$1.

The mapping from variables to names may be one-to-many due to reuse of each logical register. In the above example, register files allocation is capable of allocating physical single-ported register file 1, position 1, to hold the first name of \$1, and later using another register file and position to hold the second name for \$1. This is done if the allocation results in a lower total number of register files needed. Furthermore, busses in the implementation template allows broadcast of a value from a functional unit to multiple register files, as well as from a register file output to several functional unit inputs. Therefore, it is possible to have

one operation that defines a variable x create multiple names of x, by using this broadcast feature. By allocating *names* instead of *variables* into positions in the distributed register-file, register files allocation is able to better exploit the flexibility of the multiple single-ported register files to achieve a lower overall cost. For example, assume that variables x, y and z are defined in distinct instructions, but x and y are concurrently read in some instruction, x and z are concurrently read in some other instruction and that y and z are concurrently read in a third instruction. Due to concurrent-access conflicts, a one-to-one mapping between names and variables would require three single-ported register files. However, by using the bus broadcast feature, it is possible to write two copies of x into two register files, write y into one of these register files and write z into the other register file. This organization supports the above concurrent accesses while requiring only two register files. The broadcast feature gives more flexibility in the scheduling of busses and register files, and allows the multiple single-ported register files to efficiently emulate the ideal multi-ported global register file at a much lower cost.

In the above example, one extra register file position is used to avoid the need for a new register file. Tolerating register file growth to reduce the total number of register files is a desirable tradeoff, because a reduction in the total number of register files results in considerable reduction in the interconnection cost. In the worst case, interconnection cost may grow proportionally to the square of the number of register files, and because register files are implemented as regular silicon structures, their cost grows proportionally to the number of positions they hold.

Allocation of names into register files is done using a graph-coloring algorithm. In the colored graph, colors correspond to register files. After allocation of names into register files, the number of positions in each register file is also minimized. This is achieved by using standard register allocation techniques[2] to reuse register file positions. A register file position may be reused to hold names that are not concurrently alive.

5.2.1 Graph Coloring Algorithm

A graph G is constructed having two vertices for each microcode instruction M_i in the optimized specification. The vertices are called the read-vertex M_i, denoted as M_i^R, and the

write-vertex of M_i, denoted as M_i^W. If there is a possible execution path in the program such that variable v defined at instruction M_1 may be used as an operand for some operation in instruction M_2, Graph G has an edge labeled v between write-vertex M_1^W and read-vertex M_2^R. The edge is denoted $(M_1, M_2)_{\mathrm{v}}$. In other words, M_2 uses the name v defined at M_1. The relationship of concurrent-access of names in register files is represented in Graph G by edges that share vertices: if variables x and y are concurrently written by some microcode instruction M, Graph G has two edges labeled x and y emanating from the write-vertex of M. Similarly, if x and y are concurrently read by microcode instruction M, there will be two edges labeled x and y emanating from the read-vertex of M.

To construct graph G, *reaching definitions*[2] flow analysis is performed. A definition of variable v in instruction M that reaches instruction N creates in Graph G an edge labeled v between the read-vertex of N and the write-vertex of M. Because graph G summarizes the requirements for concurrent access, the obvious approach of applying edge coloring to graph G can be used to assign names to register files. This was suggested in [42]. *Edge coloring* of a graph is an assignment of colors to the edges in the graph such that no two edges which have a common vertex are assigned the same color. Each color corresponds to a register file. If two names x and y share a vertex in G, they are assigned distinct colors and thus located in distinct register files. Therefore, such an allocation satisfies single-ported register file constraints, allowing concurrent access from multiple single-ported register files as required by the application microcode. The minimum number of colors should be used, to reduce the overall number of register files needed. Obviously, graph G is bipartite since each edge is between a read-vertex and a write-vertex. Minimum edge coloring of bipartite graphs may be done efficiently in $\mathcal{O}(N log N)$ time by the efficient algorithm of Cole and Hopcroft [23].

However, the edge-coloring method has two important drawbacks. First, straightforward application of this method will unnecessarily store copies of a new definition in two register files when the definition reaches two distinct microcode instructions. For example, writing a name for v at instruction M and reading that name at instructions $N1$ and $N2$ creates an edge $(M, N1)_{\mathrm{v}}$ and another edge $(M, N2)_{\mathrm{v}}$. Because these two edges share the write-vertex of M they are assigned distinct colors, which correspond to distinct register files. This will

force the new value for variable v generated at M to be stored into two distinct register files, thus generating unnecessary copies (see Figure 5.1). In [42], this drawback is alleviated by changing the graph coloring subroutine to ignore the interference between two edges labeled with the same variable if they only have a write-vertex in common. This seemingly arbitrary change makes algorithmic performance hard to characterize. It is not clear if this method still produces optimal results, even for straight-line programs.

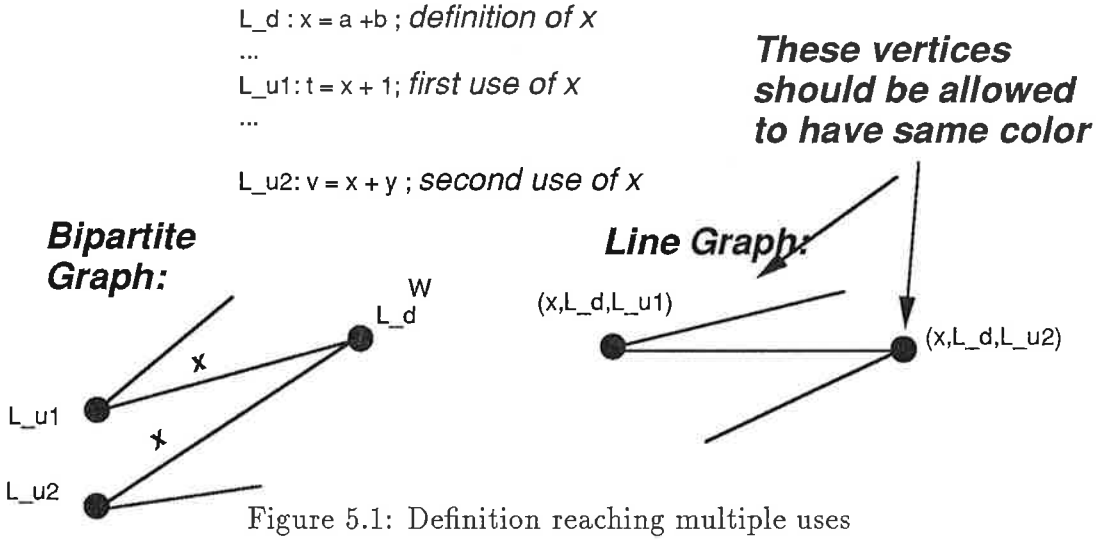


Figure 5.1: Definition reaching multiple uses

Secondly, this simple edge-coloring method as presented above is only applicable for the case of straight line programs. A problem happens if some use of a variable is reached by multiple definitions. For example, assume that distinct branches of an if-then-else statement assign new values to variable x and the variable is later used at the rejoin. In the simple edge-coloring method, each pair {definition,use} of x generates an edge in Graph G. Because the edges share the read-vertex, the corresponding names are allocated into two distinct register files. Consequently, the definition of x in the **then** side is stored in one register file, and the definition of x in the **else** side is written into another register file. When program execution reaches the use of x at the rejoin point it is not known which register file holds the proper value of x, since it depends on past history of the execution of the program. (See Figure 5.2).

A two-step solution to both of the above problems has been derived and is presented below. The following problem cases are solved:

- **Problem A:** definition reaching multiple uses

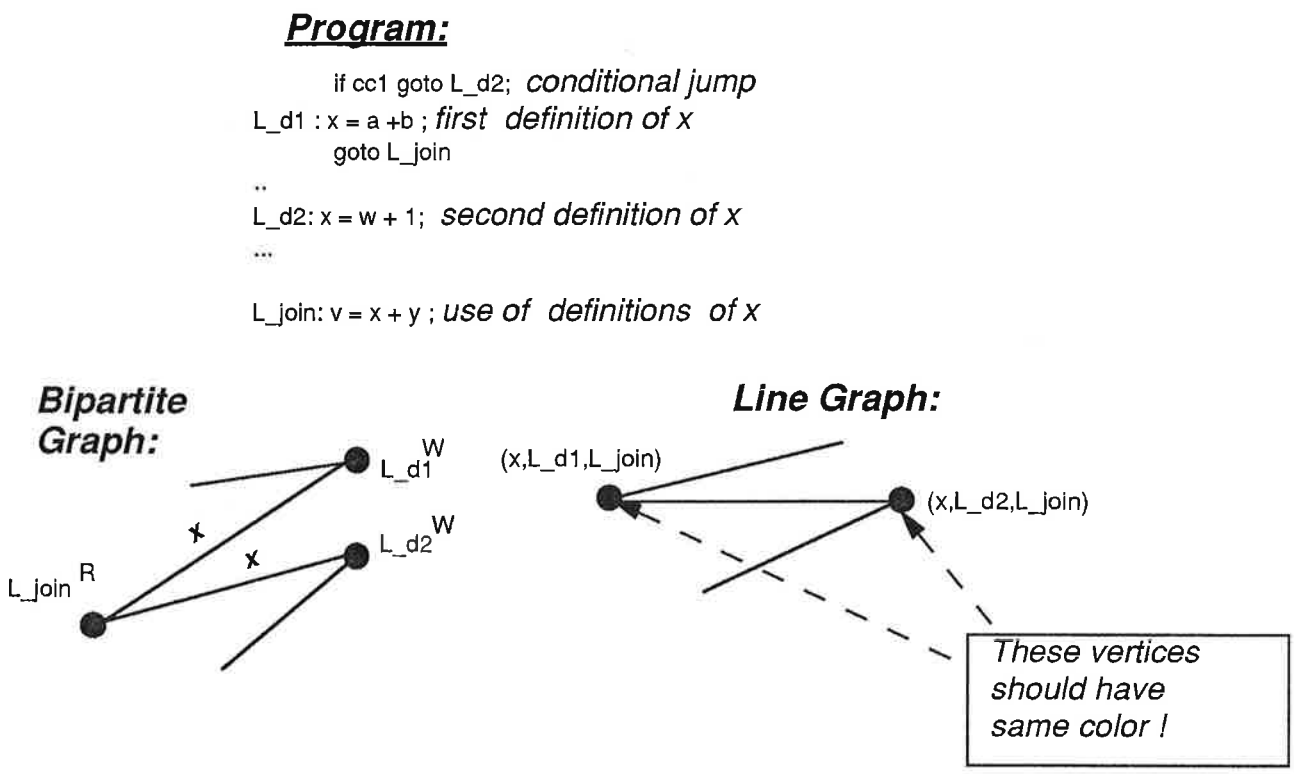


Figure 5.2: Multiple possible definitions reaching an use

- **Problem B:** use reached by multiple definitions

Graph G is transformed into another graph representation that conveys the same concurrent-access requirement, but is amenable to efficient coloring algorithms. The above problem cases are identified and corrected by adjusting the transformed graph. It is shown that this is often done without losing algorithmic performance, and the optimal number of register files is obtained.

First, the *line graph* L(G) of Graph G is generated. The line graph[53] L(G) of a graph G is a graph which has a vertex corresponding to each edge of G. Vertices of L(G) are connected by an edge iff the corresponding edges of G have a vertex in common. Notice that a vertex in L(G), denoted as $(L1, L2)_x$, represents the communication of a value between instructions L1 and L2, i.e, the value of variable x computed by an operation in microcode instruction L1 may be used as an operand in some operation in microcode instruction L2. In other words, the definition of variable x at instruction L1 reaches an use of x in instruction L2.

Furthermore, Graph L(G) conveys the same concurrent-access requirement as graph G.

For example, assume that values for two distinct program variables, `v1` and `v2` are concurrently computed by some instruction L. In Graph G, there is an edge labeled `v1` between the write-vertex L^W and the read vertex Lx^R, assuming the definition of `v1` is used at Lx. This is denoted as $(L, Lx)_{v1}$. Similarly, Graph G also has edge $(L, Lz)_{v2}$, assuming that the value of `v2` computed at instruction L is used at instruction Lz [1]. These edges share the write-vertex of L. This represents the fact that both names are concurrently defined at L.

Graph L(G) has two vertices corresponding respectively to edges $(L, Lx)_{v1}$ and $(L, Lz)_{v2}$ of Graph G, and these vertices are connected by an edge. This edge in L(G) represents the fact that both `v1` and `v2` are defined concurrently in instruction L, and the corresponding edges in Graph G share the write-vertex of L. The case of two variables `y1` and `y2` being concurrently read by an instruction L is analogous. The presence of an edge between any two vertices of L(G) indicates that the corresponding names have a concurrent-access conflict, and therefore should not be allocated into the same register file.

The above discussion illustrates how Graph L(G) represents the same requirements about concurrent-accesses as Graph G, hence vertex-coloring of L(G) can be used to allocate names into register files. *Vertex-coloring* of a graph is an assignment of colors to its vertices such that no two vertices connected by an edge receive the same color. Colors correspond to register files, and a minimum number of colors should be used. Because L(G) is the *line graph* of G, edge-coloring of G corresponds to vertex-coloring of L(G). Furthermore, because line graphs of bipartite graphs belong to the class of *claw-free perfect* graphs, vertex-coloring may be performed in polynomial time by the algorithm of Hsu[44]. Therefore, algorithmic efficiency is not lost. *Claw-free* graphs are graphs which do not contain the graph $K_{(1,3)}$ (the "claw") as a vertex-induced subgraph (see Figure 5.3). *Perfect* graphs are graphs in which the coloring number is the same as the size of the maximal induced clique for the graph itself and any of its induced subgraphs.

Before vertex-coloring, L(G) is adjusted to account for the problem cases of definition reaching multiple uses, and uses reached by multiple definitions. The procedure to adjust L(G), resulting in graph G+, is:

[1] This discussion assumes that both names computed in instruction L are *alive*, i.e., will be eventually used[2].

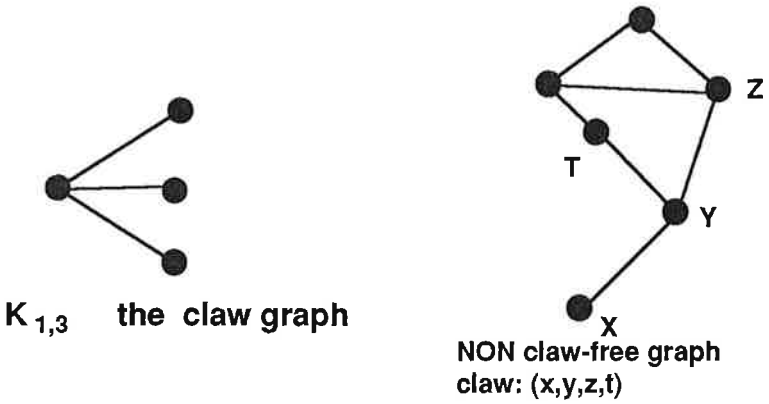


Figure 5.3: $K_{1,3}$ graph and non-claw-free graph

Procedure 5.1 *ADJUST-GRAPH:*

- **Step A:** *remove edges in L(G) that correspond to some definition of a variable reaching multiple uses.*
- **Step B:** *merge those vertices in L(G) that correspond to multiple definitions of a variable reaching the same use.*

Step A in procedure ADJUST-GRAPH above avoids the unnecessary writing of the same value into distinct register files, because by removing the edge between the vertices affected, they are allowed to receive the same color. Note, however, that the flexibility of writing copies of this value into multiple register files is preserved, because it is still possible to assign distinct colors to these vertices. Similarly, Step B constrains multiple definitions reaching a common use of a variable to use the same register file by merging the corresponding vertices in L(G).

ADJUST-GRAPH works by scanning once the vertices in the bipartite graph G: whenever two edges emanating from a write-vertex correspond to a definition of some variable reaching two uses, the edge between the corresponding vertices in L(G) is removed. Similarly, if two incoming edges of a read-vertex correspond to two definitions of the same variable, the corresponding vertices in L(G) are merged. The procedure involves visiting once each vertex of Graph G, and performing pairwise comparison between all edges on that vertex. Scanning the vertices in Graph G takes time proportional to the number of instructions in the program.

Comparing edges takes time proportional to the square of the number of edges incident on a vertex. To compute the maximum number of edges on a vertex in Graph G, let us assume that the microcode resulting from specification optimization is composed of L instructions, and is compiled for a canonical VLIW that has N functional units. Therefore, Graph G has L read-vertices and L write-vertices. Further, each write vertex may define at most N variables (since there are N functional units). In the worst case, these definitions may reach all instructions in the program. Thus the maximum number of vertices in a write vertex is $L * N$. Similarly, a read-vertex may use at most $2 * N$ distinct variables (assuming dyadic operations). Assuming these variables are defined in all instructions of the program, there are at most L definitions for each variable. Therefore, the maximum number of edges in a read-vertex is $2*L*N$. Thus, the worst-case complexity of ADJUST-GRAPH is proportional to $\mathcal{O}(L^3 * N^2)$. Experimentally the worst case performance is rarely experienced because Graph G is usually sparse.

A possible problem with procedure ADJUST-GRAPH is that it may alter graph L(G) into G+ so that G+ no longer belongs to the claw-free perfect class of graphs. This indeed happens for some programs and in this situation, a general graph-coloring algorithm is required. Figure 5.4 illustrates one such case. In that example, graph G+ has a claw. The definition of variable a reaches multiple uses. The edges between the three names of a are removed from L(G).

A check for the claw-free perfect graph property may be performed efficiently in polynomial time by the algorithm of Chvatál and Sbihi [20]. If the graph is claw-free-perfect, the check also informs the minimum number of colors required. Here, the optimal (minimum) number of single-ported register files is used. If this property is not present in G+, a coloring heuristic must be used because the graph coloring problem on general graphs is NP-complete. A good heuristic for an approximate solution to the vertex-coloring problem has been proposed by Chaitin[19]. This heuristic frequently achieves the optimal solution for sparse graphs. Experimental evidence indicates that Chaitin's algorithm is particularly good for sparse graphs[70]. Figure 5.5 below shows a simple experiment with a linear-phase B-spline filter. In that experiment, the number of functional units available for specification optimization ranges from one (single operation per cycle, a la RISC) until saturation is

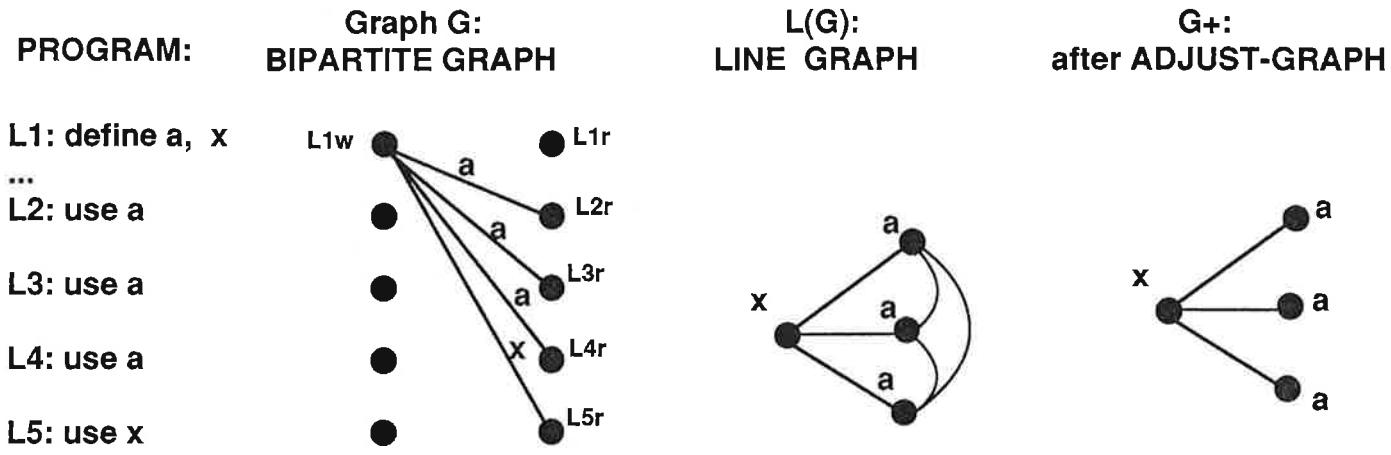


Figure 5.4: Claw Graph Resulting from ADJUST-GRAPH

reached with 16 functional units, when maximal parallelism is achieved and further increases on resource availability do not result in reduced execution time. The number of colors, corresponding to number of register-files used by Chaitin's heuristic is compared against the optimal number of colors computed by the procedure to check for the claw-free-perfect graph property.

In this experiment, all graphs are claw-free perfect. From this experiment it is seen that Chaitin's heuristic frequently achieves the optimal number of colors for claw-free perfect graphs. Our implementation applies the test for the claw-free perfect graph property to G+ and uses Chaitin's heuristic for coloring. Because the test also produces the optimal number of colors required if the graph passes the test, it is known how far from optimality is the result of Chaitin's heuristic.

After variables have been grouped into register files, minimization of the size of each individual register file is performed by standard graph-coloring-based register allocation. Register allocation was proposed by Chaitin for use in optimizing compilers[19]. It attempts to reduce the number of registers required for program execution by reusing registers. Similarly, it is possible to reuse a register file position to hold names whose lifetimes do not overlap. For each register file, an interference graph is built. This graph has a vertex corresponding to each name allocated to that register file. There is an edge between any two

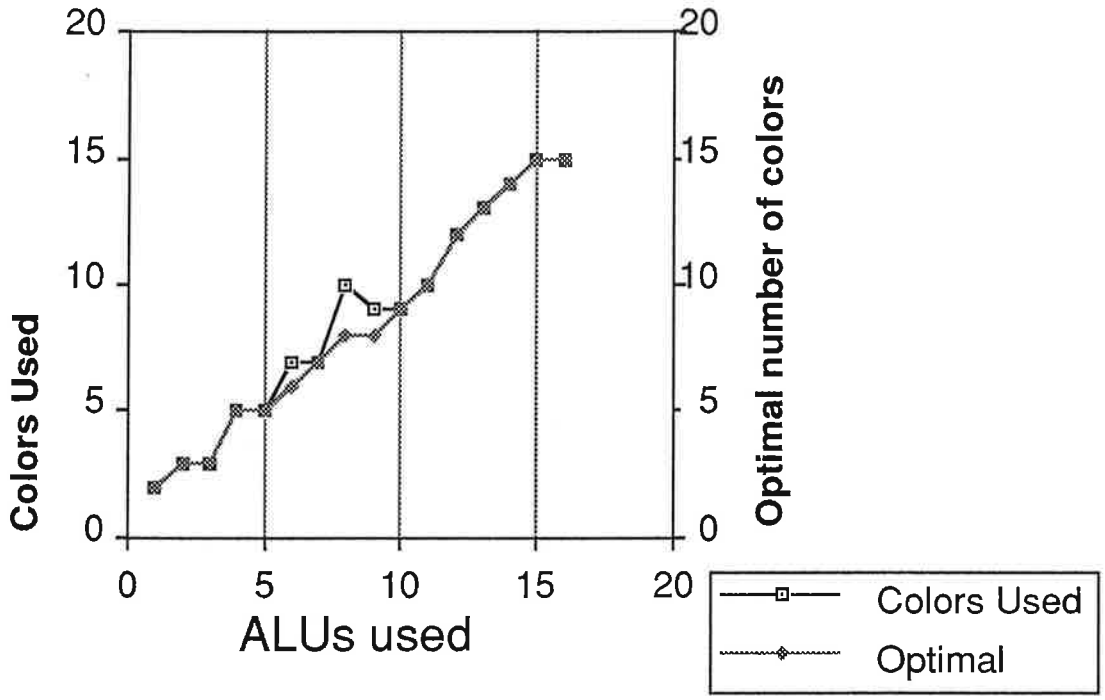


Figure 5.5: Linear-Phase filter experiment: Register Files Needed

names whose lifetimes overlap. Vertex-coloring of the interference graph assigns colors, corresponding to positions in the register file, to names. Names with non-overlapping lifetimes may receive the same color, and thus reuse the same register file position. If the minimum number of colors is used, the minimum-size register file is obtained.

5.2.2 Performance of ADJUST-GRAPH

This section investigates characteristics of programs that preserve the claw-free-perfect graph property of Graph L(G) even with application of ADJUST-GRAPH.

The Strong Perfect Graph Conjecture is valid for claw-free graphs [67]. The Strong Perfect Graph Conjecture[41] states that G+ is perfect if and only if:

Condition i: G+ does not have odd holes (odd cycles) of length greater than 3.

Condition ii: G+ does not contain odd anti-holes

A hole is a vertex-induced subgraph of C+ which is a cycle. A anti-hole is the complement of a hole. For claw-free graphs, Ben-Rebea's lemma simplifies the above conditions:

Lemma 5.1 (Ben-Rebea) *Let G be a connected claw-free graph with chromatic number $\alpha(G) >= 3$. If G contains an odd antihole then it contains a hole of length five.[20]*

Therefore, according to Ben-Rebea's lemma, if Graph G+ satisfies condition *i*, then it will also satisfy condition *ii*. If Graph G+ has a chromatic number equal to 2 or less, it obviously cannot contain a cycle, because K_3, the smallest cycle, has chromatic number equal to 3. The revised set of conditions of the Strong Perfect Graph Conjecture for claw-free graphs is:

Condition a: G+ is claw-free (so [67] applies)

Condition b: G+ does not have odd holes (odd cycles) of length greater than 3.

We proceed to investigate situations in which claws and odd holes are created in G+.

Observation 1: *The case of Step B of ADJUST-GRAPH never occurs for straight-line programs.*
This is true because, in straight line programs, the use of a variable may only be reached by a single definition. If two operations define new values for variable v consecutively before an use of v, only the last definition reaches the use.

Observation 2: *For straight-line programs, if vertices $(Lwa, Lra)_a$ and $(Lwb, Lrb)_b$ are adjacent in Graph G+ , then either $Lwa = Lwb$ or $Lra = Lrb$.*

Informally, vertices in G+ correspond to definition-use pairs of some variable. Each vertex in Graph G+ corresponds to one edge in Graph G, because ADJUST-GRAPH does not execute the vertex-merging step on L(G) for straight-line programs. Therefore, two vertices are adjacent in G+ only if the corresponding edges share a vertex in G. That vertex may either be a read-vertex of a write-vertex. If the two vertices of G+ are labeled with variables x and y this means that x and y are either both assigned values in some microcode instruction, or are simultaneously used in some microcode instruction.

Observation 3: *For straight-line programs, a claw in G+ involves vertices labeled with at most three variables*

This is verifyed by analyzing all possible cases. Let us assume that G+ has a claw composed of vertices labeled with variables `x`, `y`, `z` and `t`, and edges (`xy`), (`xz`), (`xt`). Assume that `x` is distinct from `y`. Without loss of generality, the proof consists in showing that either `z` or `t` is the same as `y`.

Edge (`xy`) implies that names `x` and `y` are either concurrently read or written (Observation 2). Assume, without loss of generality, that they are concurrently read (the proof of the alternative case is symmetrical) and that `x` is distinct from `y`. Edge (`xz`) implies that names `x` and `z` also are either concurrently read or written, or are the same name. If they are the same name, the observation is proved. `x` and `z` must be concurrently written, otherwise edge (`zy`) would exist, and the subgraph induced by vertices {x,y,z,t} would not be a claw. Edge (`xt`) implies that names `x` and `t` are either concurrently read or written, or the same name. If both are distinct and concurrently written, edge (t,z) would exist and the graph is not a claw.

If both are distinct and concurrently read, edge (`ty`) would exist and the graph is not a claw. Therefore, here `x` and `t` must be the same name, and the observation holds.

Observation 4: *Graph G may only contain even holes*

This results from the fact that Graph G is bipartite. Each edge in Graph G is between a write-vertex and a read-vertex. Since the hole is a closed path that connects all vertices, the number of vertices must be even.

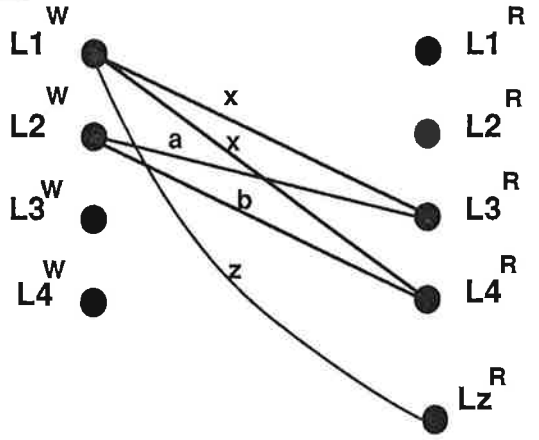
A consequence of Observation 4 is that holes in L(G) are also even [41]. This happens because the line graph of cycle C_n, the cycle with n vertices, is also C_n. An odd hole is created in G+ if ADJUST-GRAPH disconnects an edge of a hole, and the vertices connected by that edge are adjacent to a third vertex. An example is shown in Figure 5.6.

If all consecutive edges of a hole in Graph G are labeled with distinct variables, the hole is preserved in G+, because no edge is ever removed from that hole (edges are only removed between vertices labeled with the same variable). A problem happens when consecutive edges of a hole in Graph G are labeled with the same variable. This is the case when a definition reaches multiple uses, and another definition in the same instruction reaches some use. From Observation 1, the case of an use reached by multiple definitions does not happen

PROGRAM:

```
L1: x= t+1;  z=l-m ; define x and z
L2: a=w/3; b=n*p   ; define a and b
L3: u= x - a       ; use x and a
L4: v= x * b       ; use x and b
Lz: ...            ; use z
```

BIPARTITE GRAPH:



CORRECTED LINE GRAPH G+:

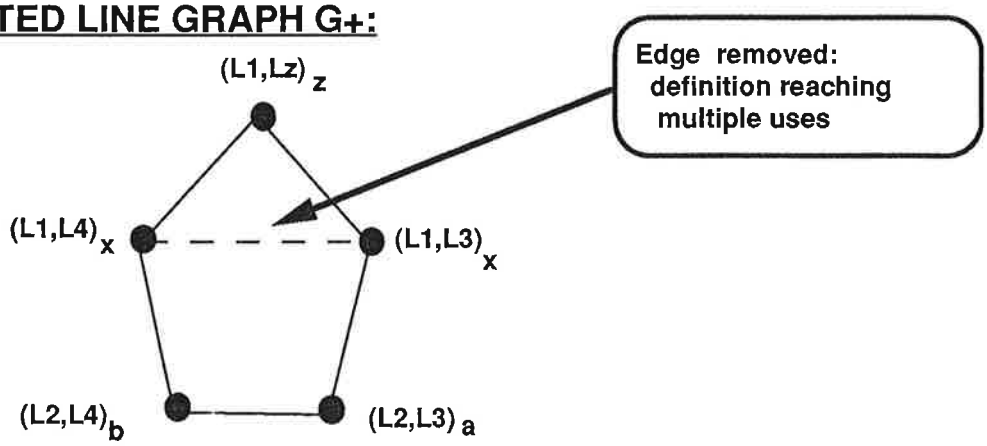


Figure 5.6: Example of straight-line program that generates an odd cycle in G+

for straight line programs. If a hole in Graph G contains an even number of such cases, an

even number of edges is added to the corresponding hole in G+ and the claw-free perfect graph property is preserved.

In the example of Figure 5.6, the optimal number of colors is the same for the 5-cycle as it would be if the edge between vertices $(L1, L4)_x$ and $(L1, L3)_x$ had not been removed. One might be tempted to imagine that the removal of an edge that creates an odd hole would produce a graph which requires the same number of colors as the graph in which the edge has not been removed. Avoiding removal of the edge might be useful, because the graph without the edges removed is claw-free-perfect and thus the polynomial-time coloring algorithm is applicable. One possible implementation is to recognize those cases when the removal of an edge in G+ introduces an odd hole. ADJUST-GRAPH can be altered to identify the creation of odd holes, and leave the edge to preserve the claw-free perfect graph property. However, the above is not the case for all graphs, as is shown next.

Let us concentrate in the case for edge removal when consecutive vertices of L(G) corresponding to the same definition reaching two uses concurrently with another definition happens twice. Assume that Graph G has an even hole, and the above case of edge removal happens twice. The removal of the first edge introduces an odd hole, and later removal of the second edge transforms the hole into an even hole. It may be easily seen that the resulting graph is an even cycle, and only 2 colors are necessary, whereas 3 colors would be needed in case the edges (which compose 3-cliques) were left.

5.2.3 Discussion

The preceding section illustrates the situations in which the claw-free-perfect graph property is lost in the application of ADJUST-GRAPH. This may be used during Specification Optimization to create schedules in which the property is most likely to be preserved. These are situations in which there are free slots in the microcode, and it is possible to schedule a given operation in one of multiple instructions, without impairing code performance. Preference is given to schedules in which definitions that reach multiple uses are performed in isolation. It is also possible to replicate the execution of the operation. If there are empty slots in the microcode, this is done for free.

To keep things in perspective, however, it must be realized that a resulting graph G+

may not be claw-free-perfect but still be amenable to a specialized coloring algorithm. For example, the claw is a bipartite graph, and thus can obviously be colored with two colors.

A future possibility is to integrate the information about graphs, claws and odd holes with the scheduling algorithm. Graph G and L(G) could be incrementally maintained during Percolation Scheduling. The possibility of preserving the claw-free-perfect graph property may be considered as yet another scheduling parameter.

Chapter 6

Memory Organization for Concurrent Access

The communication bandwidth between the CPU and the memory is a key factor in determining the overall performance of a computer system[10, 45]. This interface, if not properly designed, can become a serious bottleneck and, as VLSI technology progresses and CPUs become faster[68], the problem can get even worse. Traditionally, this problem is overcome by the use of caching and memory interleaving techniques[45, 50]. These techniques have provided acceptable solutions.

In horizontally microcoded engines and VLIW computers[38], each wide instruction specifies several ALU operations and memory accesses that are executed concurrently. This capability requires even more memory bandwidth to keep the multiple functional units busy. A VLIW computer is capable of making multiple simultaneous memory accesses. This characteristic, coupled with the relationship between the memory addresses referenced in the same instruction, may be used at compile time to achieve higher memory bandwidth. By appropriately allocating data over multiple memory banks at compile time, concurrent accesses can be enhanced at run time. This chapter describes compilation and memory allocation techniques to take advantage of commonly-occurring memory access patterns to achieve enhanced memory bandwidth.

6.1 Parallel Memory Access Techniques

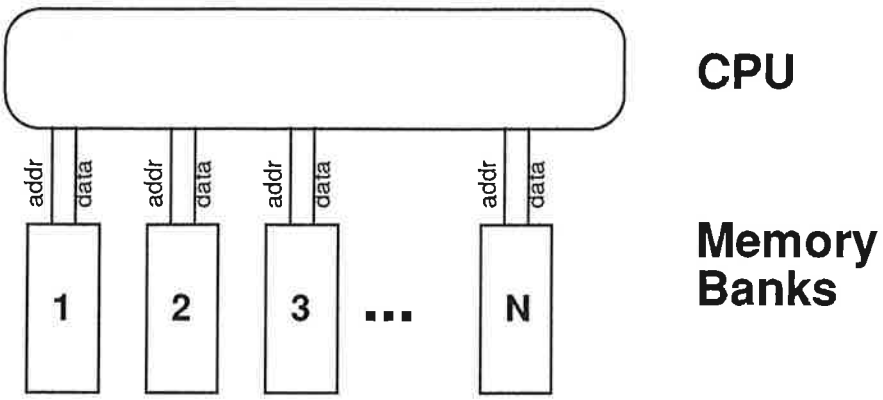


Figure 6.1: Memory Subsystem with Multiple Memory Banks.

The memory subsystem is composed of multiple banks, each of which can be independently accessed through a dedicated path to the CPU, as illustrated in Figure 6.1. For a given memory technology, one such organization with N memory banks has a potential bandwidth increase by a factor of N. However, the increase may not be achievable due to occasional concurrent accesses of data located in the same memory bank. True multiported main memory is expensive. The practical solution generally adopted is to stretch the cycle time when there is a memory bank collision. Performance is degraded if collisions are frequent. Furthermore, the amount of hardware needed to detect bank collision is proportional to the square of the number of memory banks.

The key idea behind the proposed method is to use information, available at compile time, to allocate into separate memory banks data that are concurrently accessed. The compiler has the freedom to decide at compile time the starting address and the memory bank where each array variable is stored. This fact is used to enhance the utilization of memory bandwidth provided by multiple banks. A key motivation is the observation access to data structures in scientific programs is structured, as demonstrated by the success of vector architectures[26]. The proposed method concentrates on optimizing concurrent accesses of array variables in FORTRAN-like DO loops, because arrays (possibly multi-dimensional) usually constitute the most important data structure in scientific programs. Furthermore, there exists the opportunity to enhance the performance of a large body of existing scientific FORTRAN programs.

Most indirect references in inner loops of scientific code are made to array variables. The

predictable and structured nature of DO loops provides excellent opportunity for successful compile-time analysis and determination of the access patterns to the array variables involved. In this context, the nature of memory accesses generated by the VLIW instruction word can be efficiently exploited.

The above is not true in the general case, for example, of memory accesses to arbitrary linked data structures. Such memory references, including following pointers into a list of indirect references, depend on runtime information which cannot be easily predicted at compile-time. This renders compile-time analysis of memory access patterns very difficult if not impossible. However, techniques like memory reference disambiguation[60] have been successfully employed, even though their scope of application concentrates only on handling array references. This shows that opportunities exist and dramatic performance gains are achievable by optimizing memory access of array variables.

Several researchers proposed hardware techniques to streamline access to data in multiple memory banks. Budnick and Kuck[16] proposed an interleaved memory organization in which having a prime number of memory banks reduces the probability of concurrent accesses to the same bank. Modern architectures like Cydrome's CYDRA 5[24] computer use a hardware hashing scheme to spread out the address of words over multiple memory banks. These techniques require extra hardware for decoding memory addresses and increase the latency of memory access. Shapiro[72] studied the theoretical limitations on concurrent accesses to elements of matrices in SIMD architectures.

In the following section, a graph-coloring technique to allocate entire arrays into memory banks is described. This is done to allow concurrent access of elements of different arrays. When several elements of an array are referenced by same VLIW instruction, the array must be distributed across several memory banks to allow concurrent access of these elements. In Section 6.3 a schema to distribute elements of an array over multiple memory banks to accommodate a class of frequently occurring memory accesses is presented. Theorems characterizing the conditions to allow concurrent access are presented. Section 6.4 describes the associated compilation techniques along with their application to a number of examples. The last section describes the possible future extensions of this method.

6.2 Bank Allocation by Graph Coloring

It is always possible to make concurrent access of elements of two different arrays if the arrays are allocated into different banks. By extending this idea, concurrent access of elements of several distinct arrays is possible by allocating each array into its own memory bank. This may not be a practical solution if there are many array variables because an excessive number of memory banks may be required.

Arrays that are not concurrently accessed may be stored in the same memory bank, thus reducing the number of memory banks needed. A graph coloring procedure is employed to efficiently allocate the arrays into memory banks. Two array variables X and Y are said to *interfere* if elements from `X` and `Y` are concurrently accessed by the same instruction in the application code. The set of all interferences among array variables is found by scanning the application code. An *interference graph* is constructed in which each vertex corresponds to an array variable, and edges connect array variables that interfere. An example is shown in Figure 6.2. In this example, arrays `A` and `B` are accessed by the first instruction and thus interfere. Similarly, arrays `B`, `C`, and `D` also interfere because the second instruction accesses one element from each.

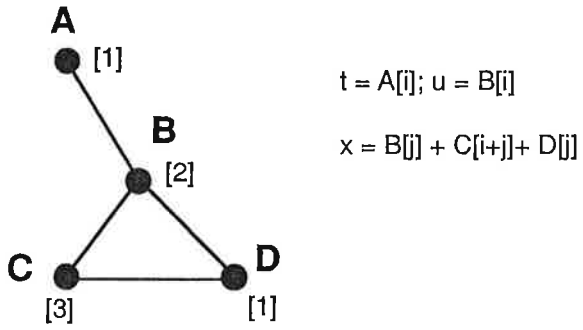


Figure 6.2: An Example Interference Graph

Vertex-coloring of the interference graph is used to allocate the array variables into the memory banks to allow conflict-free access. The approach of allocating entire arrays into memory banks may provide an acceptable solution for several applications, and the same technique may also be employed to allocate scalar variables into memory banks. However, it

is still a problem when multiple concurrent accesses are made to elements of the same array variable. This situation is important in practice, because multiple accesses to the same array variable are frequently present in the original code, or are introduced by code transformation techniques like loop unrolling, loop quantization[61] or inter-iteration optimization[81, 30]. For example, assume that elements `X[i]` and `X[i+1]` of vector `X` are accessed concurrently. In this case, it is necessary to partition the elements of `X` over several memory banks so that references to concurrently accessed elements are made to distinct memory banks.

One example solution is to split vector `X` into two sub-vectors, `X-odd` and `X-even`. `X-odd` is composed of the odd-indexed elements of `X`, and `X-even` is composed of the even-indexed elements of `X`. By storing `X-odd` in one memory bank, and `X-even` into another, `X[i]` and `X[i+1]` always refer to data stored in different memory banks and can be accessed concurrently, thus doubling the effective memory bandwidth.

The above approach may be phrased as: rewrite all references to vector `X` in the original program in terms of two new vectors, `X-odd` and `X-even` such that the transformed program is equivalent to the original, but has no concurrent accesses to multiple elements of the same array variable. Then, the graph coloring technique described before can be used to allocate the new array variables of the transformed program into memory banks.

The problem of mapping data over the memory banks, then, reduces to one of determining how to partition the original program's array variables to allow concurrent access, and then rewriting the program in terms of these new variables. The next section presents the procedure to solve this latter problem by analyzing concurrent accesses to elements of an array variable for a class of memory access patterns.

6.3 Concurrent Access of Array Elements

In this section, the notion of distributing an array over several memory banks and the requirements for concurrent memory access are formalized.

Definition: A *distribution schema* for an array A over M memory banks is a function

$$D{:}\{Integers\} \rightarrow \{0, 1, 2, \ldots M-1\},$$

where $D(i) = k$ means that array element A_i is stored in memory bank k.

is still a problem when multiple concurrent accesses are made to elements of the same array variable. This situation is important in practice, because multiple accesses to the same array variable are frequently present in the original code, or are introduced by code transformation techniques like loop unrolling, loop quantization[61] or inter-iteration optimization[81, 30]. For example, assume that elements `X[i]` and `X[i+1]` of vector `X` are accessed concurrently. In this case, it is necessary to partition the elements of `X` over several memory banks so that references to concurrently accessed elements are made to distinct memory banks.

One example solution is to split vector `X` into two sub-vectors, `X-odd` and `X-even`. `X-odd` is composed of the odd-indexed elements of `X`, and `X-even` is composed of the even-indexed elements of `X`. By storing `X-odd` in one memory bank, and `X-even` into another, `X[i]` and `X[i+1]` always refer to data stored in different memory banks and can be accessed concurrently, thus doubling the effective memory bandwidth.

The above approach may be phrased as: rewrite all references to vector `X` in the original program in terms of two new vectors, `X-odd` and `X-even` such that the transformed program is equivalent to the original, but has no concurrent accesses to multiple elements of the same array variable. Then, the graph coloring technique described before can be used to allocate the new array variables of the transformed program into memory banks.

The problem of mapping data over the memory banks, then, reduces to one of determining how to partition the original program's array variables to allow concurrent access, and then rewriting the program in terms of these new variables. The next section presents the procedure to solve this latter problem by analyzing concurrent accesses to elements of an array variable for a class of memory access patterns.

6.3 Concurrent Access of Array Elements

In this section, the notion of distributing an array over several memory banks and the requirements for concurrent memory access are formalized.

Definition: A *distribution schema* for an array A over M memory banks is a function

$$D{:}\{Integers\} \rightarrow \{0, 1, 2, \ldots M-1\},$$

where $D(i) = k$ means that array element A_i is stored in memory bank k.

In the definitions and proofs below, infinite-sized arrays are assumed to avoid the special cases introduced by the fact that some elements do not exist when array sizes are limited. This assumption makes the proofs less cumbersome. The general results derived in this section apply to finite-size arrays as well.

Definition: A *modulated skewed distribution* with modulus K, denoted MSD(K,M), is a distribution schema for M memory banks where

$$D(i) = \lfloor i \div K \rfloor mod M.$$

for K > 0 and M > 1. An MSD(K,M) organization is similar to the traditional memory interleaving organization, but allocates contiguous blocks of K elements of the array into each memory bank. This is illustrated in Figure 6.3.

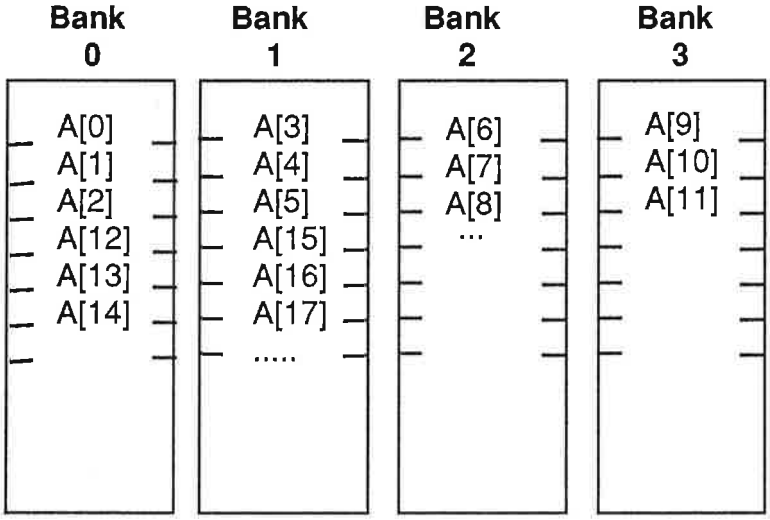


Figure 6.3: Modulated Skewed Distribution Example - MSD(3,4).

Definition: A *parallel access sequence* is a set of P functions of a variable i, $(f_1, f_2, f_3, \ldots f_P)$, where f_1 is the identity function, i.e. $f_1(i)=i$, and $f_j(i) \neq f_k(i)$, for all j,k with $1 \leq j < k \leq P$, i.e. for each i, all the $f_\alpha(i)$ $1 \leq \alpha \leq P$, are different. A parallel access sequence is also denoted PAS($f_1(i)$, $f_2(i)$, $f_3(i)$, ..., $f_P(i)$). For example, PAS(i,i+3) means $f_1(i)=i$ and $f_2(i)=i+3$.

For each value of i, a parallel access sequence describes a P-element subset of an array. The elements in the subset are concurrently accessed from memory during the execution of a loop in a program. Intuitively, i indicates the loop iteration count.

The P functions f_α, $1 \leq \alpha \leq P$, in a PAS, called the *access functions* of the PAS, designate the indices of the array elements as a function of i, that are referenced during iteration i of the loop. In this study, the focus is on f_α that are linear functions of i, namely of the form $\beta\ i + \delta$, with both β and δ being integers, because these access functions occur frequently in many applications[11].

Definition: A distribution schema D is *valid* for a parallel access sequence *PAS* if $D(f_j(i)) \neq D(f_k(i))$, for all *i,j,k* where $i > 0$ and $1 \leq j < k \leq P$, i.e. for any value of i (loop iteration counter), the P array elements designated by the access functions are all stored in different memory banks.

A valid distribution schema ensures that the array elements referenced by the PAS during each iteration of the loop may be accessed concurrently from memory. For example, suppose that in the program there is a DO loop with i as the iteration counter, and that array elements A[i] and A[i+3] are added in every iteration. Assume that the elements of array A are organized into two memory banks in alternating groups of three consecutive elements, so that the first group of three elements of the array is stored in Bank 0, the next three elements are stored in Bank 1, the next three elements are again stored in Bank 0, and so forth, as prescribed by MSD(3,2). It can be easily seen, by analogy with Figure 6.3 that, for any element i, if element i is stored in Bank 0, then element $i+3$ is stored in Bank 1, and conversely, if i is in Bank 1, then $i+3$ is in Bank 0. Thus, with this MSD (3,2) mapping of the array over the memory banks, elements i and $i+3$ can always be accessed concurrently from memory.

Theorem 6.1 *MSD(K,M) is valid for PAS(i,i+K), for any integers K > 0 and M > 1.*

Proof: Consider D(i) and D(i+K): from the definition of MSD(K,M), D(i) $= \lfloor i \div K \rfloor \bmod M$, and D(i+K) $= \lfloor (i+K) \div K \rfloor \bmod M$.

Let $\alpha = \lfloor i \div K \rfloor$. Then,

D(i) $= \alpha \bmod M$, and

D(i+K)$= (\alpha + 1) \bmod M$

Thus, D(i)$\neq$D(i+K) , and consequently, MSD(K,M) is valid for PAS (i,i+K).

•

This theorem shows that a MSD organization of an array allows concurrent memory access for a pair of accesses of the form PAS(i,i+K). However, this is not the only possible organization of the array that allows concurrent access. Theorem 6.2, below, is a generalization of Theorem 6.1, and states the conditions under which it is still possible to make concurrent access to elements i and $i+K'$ if the array is organized as MSD(K,M), with K$\neq$K'. This is important because an array may be referenced in several loops in the program, in each loop the accesses are of the form (i,i+K_a) for different K_a, and it is necessary to choose one organization for the array that is valid for all, possibly different, access patterns. Note that different values of K_a may result, for example, from a two-dimensional matrix being accessed by A[i,j]+A[i,j+1] in one loop (fetching consecutive elements by rows), and by A[i,j]+A[i+1,j] in another loop (fetching consecutive elements by columns). Assuming that A is a 256*256 matrix stored in row-major order, the access to A[i,j] is a reference to element (i*256+j) of A, while A[i,j+1] refers to element (i*256+j+1), and thus these two accesses refer to elements that are 1 element apart. The access to A[i+1,j] refers to element ((i+1)*256+j) = (i*256+j+256), which is 256 elements apart from the element referenced by A[i,j].

Lemma 1: Given MSD(K,M) and an integer α, D(i) = D(i+αKM).

Proof: From the definition,

D(i+αKM) = $\lfloor \{i + \alpha KM\} \div K \rfloor$ mod M = $\lfloor i \div K + \alpha M \rfloor$ mod M

= $\lfloor i \div K \rfloor mod M$ = D(i). •

Lemma 1 helps simplify the subsequent proofs because the allocation of elements over the memory banks is regular, exhibiting a pattern that repeats with period KM. The next lemma is a special case of Theorem 6.2, i.e. for values of K' in the range [0 : KM], and will be used in the proof of Theorem 6.2.

Lemma 2: Given integers K,M>0 and 0$\leq$K'<KM, MSD(K,M) is valid for PAS(i,i+K') iff K$\leq$K'$\leq$(M-1)K.

Proof: (<=) Let K$\leq$K'$\leq$(M-1)K, want to show MSD(K,M) is valid for PAS(i,i+K'). Prove by contradiction.

Assume MSD(K,M) is not valid for PAS(i,i+K').

=> ∃ i such that D(i) = D(i+K') or $\lfloor i \div K \rfloor modM = \lfloor (i+K') \div K \rfloor modM$ or $\lfloor i \div K \rfloor modM = \lfloor i \div K + K' \div K \rfloor modM$ (1)

Case I: If K' mod K = 0

=> $\lfloor i \div K + K' \div K \rfloor modM = \{\lfloor i \div K \rfloor + \lfloor K' \div K \rfloor\} modM = \lfloor i \div K \rfloor modM + (K' \div K) modM$

(1) becomes => $\lfloor i \div K \rfloor modM = \lfloor i \div K \rfloor modM + (K' \div K) modM$ => $(K' \div K) modM = 0$ => $K' = 0 or K' = KM$ => Contradiction.

Case II: If K' mod K ≠ 0

=> $\lfloor i \div K + K' \div K \rfloor modM =$

$\lfloor i \div K \rfloor modM + \lfloor K' \div K \rfloor modM$ (Case a) OR $\lfloor i \div K \rfloor modM + \lfloor K' \div K \rfloor modM + 1$ (Case b)

Case a: (1) becomes => $\lfloor i \div K \rfloor modM = \lfloor i \div K \rfloor modM + \lfloor K' \div K \rfloor modM$ => $\lfloor K' \div K \rfloor modM = 0$ => $K' < K or K' > KM$ => Contradiction.

Case b: (1) becomes => $\lfloor i \div K \rfloor modM = \lfloor i \div K \rfloor modM + \lfloor K' \div K \rfloor modM + 1$ => $\lfloor K' \div K \rfloor modM + 1 = 0$ => $\lfloor K' \div K \rfloor modM = M - 1$ => $K' > (M-1)K$ => Contradiction.

(=>) Let 0<K'<K (Case I), or (M-1)K<K'<MK (Case II).

Case I: 0<K'<K Let i = 0 $D(i) = \lfloor i \div K \rfloor modM = 0$, and

$D(i+K') = \lfloor (i+K') \div K \rfloor modM = \lfloor K' \div K \rfloor modM = 0$

Since ∃ i such that D(i)=D(i+K'), MSD(K,M) is not valid for PAS(i,i+K').

Case II: (M-1)K<K'<MK Let i = K-1

$D(i) = \lfloor (K-1) \div K \rfloor modM = 0$, and $D(i+K') = \lfloor (i+K') \div K \rfloor modM = \lfloor (K-1+K') \div K \rfloor modM$

Since (M-1)K<K'<MK, let K'=(M-1)K+d, where 1≤d≤K-1

=> $D(i+K') = \lfloor (K-1+d) \div K + M - 1 \rfloor modM = \{\lfloor (K-1+d) \div K \rfloor + M - 1\} modM, since 1 \le d \le K-1 = (1+M-1) modM = 0$

Since ∃ i such that D(i)=D(i+K'), MSD(K,M) is not valid for PAS(i,i+K'). **Q.E.D.**

Figure 6.4 illustrates Lemma 2, showing the cases where D(i) would be equal to D(i+K'). Theorem 6.2 is a consequence of this lemma, and of the regularity in the allocation of array elements over the memory banks, as in Lemma 1.

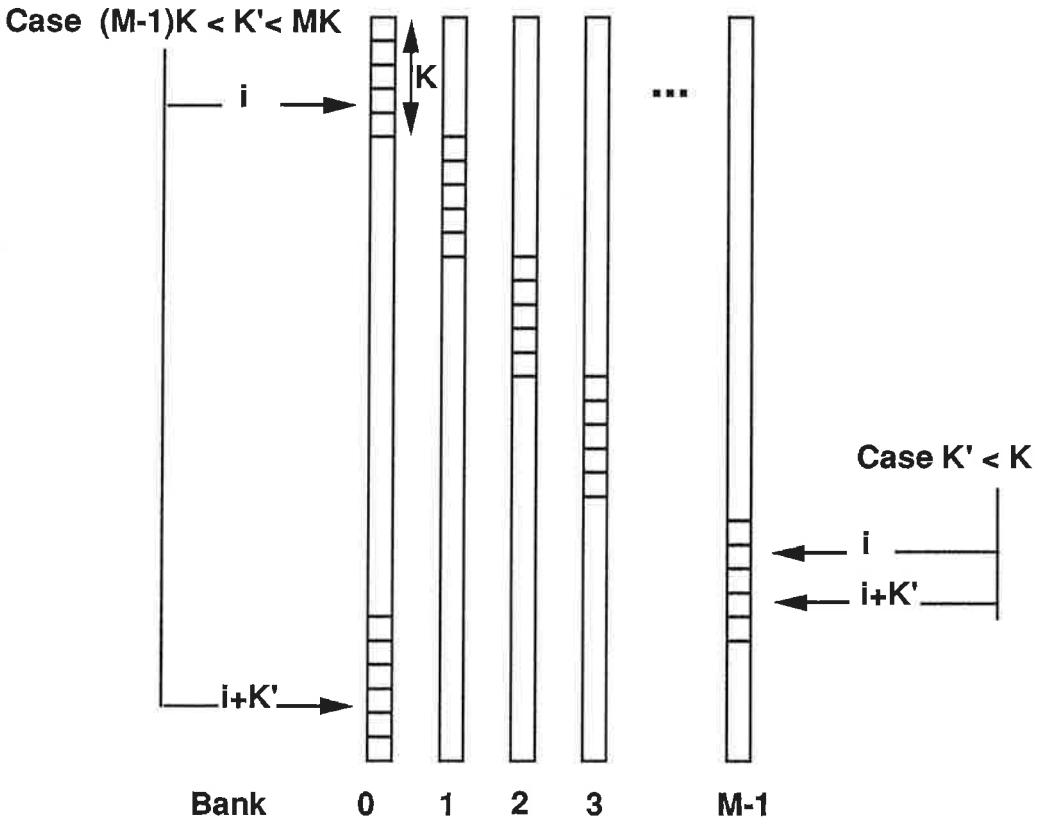


Figure 6.4: Values of K' for which MSD(K,M) is not valid for PAS(i,i+K').

Theorem 6.2 *For any integers K, K' >0 and M >1, MSD(K,M) is valid for PAS(i,i+K') iff K ≤K' mod (KM) ≤ (M-1)K.*

Proof: *Let i' = i mod KM, and K" = K' mod KM*

>From Lemma 1, D(i) = D(i'), and (2)

D(i + K') = D(i mod KM + K' mod KM) = D(i' + K") (3)

>From Lemma 2,

D(i')≠D(i'+K") iff K ≤ K" ≤ (M-1)K. (4)

Substituting (2) and (3) in (4), we have

D(i)≠D(i+K') iff K ≤K' mod (KM) ≤(M-1)K.

Q.E.D. •

Frequently in real applications, a given array is referenced inside several different loops in the program. This introduces the important problem of finding a distribution schema which is valid for several PAS of the form (i, i+K_i), where all K_i are integers. Each of the loops in which the array is referenced imposes a different PAS on the distribution schema. This problem can be formalized as follows: given a set of h PAS of the form (i, i+K_j), j=1,...,h find K and minimum M, such that MSD(K,M) is valid for {PAS(i,i+K_j)}, for j=1,...,h.

Applying the conditions required by Theorem 6.2, the solution to this problem reduces to finding two integers K,M >0, with M minimal, that satisfy the following system of equations:

$K \leq K_1 \bmod KM \leq (M-1)*K$

$K \leq K_2 \bmod KM \leq (M-1)*K$

$K \leq K_3 \bmod KM \leq (M-1)*K$

...

$K \leq K_h \bmod KM \leq (M-1)*K$

The above system admits a trivial solution (K,M) = (1,N), where N is the number of elements in the array, which corresponds to the case in which every array element is stored by itself in its own memory bank. This solution is only practical when N is small. M is the number of memory banks, and is required to be the smallest M such that K and M satisfy the above system of equations. A simple technique to solve the system of equations is to search for solutions starting from M=2, and searching for K starting at 1 up to the smallest K_h. Since number of memory banks is related to the maximum number of concurrent accesses being performed, this complete enumeration procedure is still practical.

6.4 Implementation and Application

Code generation to achieve concurrent memory accesses can be viewed as rewriting the code in terms of new array variables so that memory accesses are made

to arrays located in different memory banks. A simple example of array access is:

```
     DO 10 I = 0, N
10 ...A[I]+ A[I+K]
```

for some constant K. Assuming array A is organized as MSD(K,M), the above code is rewritten as:

```
    DO 10 base=0, N/K, K

        DO 20 ii= base, base+K-1
        20        A_in_bank_1[ii] + A_in_bank_2[ii]


        DO 21 ii= base, base+K-1
        21        A_in_bank_2[ii] + A_in_bank_3[ii]

        ...

        DO 2K ii= base, base+K-1
        2K        A_in_bank_M[ii] + A_in_bank_1[ii+K]

10    CONTINUE
```

To allow concurrent memory accesses, it is necessary to recognize array access sequences of the above form during compilation, and generate the code as exemplified above. The above code sequence can be seen as a template for the code to be generated, by varying the number of inner DO loops and the ranges of the loop iteration counters. To recognize access sequences amenable to this technique, it is sufficient to note that the array index expression is an induction variable[2] of the loop. Interestingly, much of the information needed for memory reference disambiguation is also useful to perform this kind of data allocation to

enhance concurrent memory access. The technique is applicable whenever the symbolic expressions for two memory references to array elements inside of a loop differ by a constant K. Furthermore, the compiler may use the symbolic expression describing the target address of each memory reference to make decisions about concurrent scheduling of memory operations. If the initial value of the loop iteration counter is known at compile time, the generated code sequence is similar to the one above. In the case where the index of the first memory access is not known at compile time, the generated code would have to initiate execution at one of the inner DO loops above, with the appropriate value for the variable `base`. In the above example, if the loop should initiate execution at i= 2*K, this implies that, in the generated code, execution should begin at the third inner DO loop, with `base` = 0.

In the most general case, the compiler does not have any information about the initial value of the loop counter. For example, it may be an input variable to the program. In this case, it is necessary to generate a prolog sequence of code that is executed before the loop, which correctly initializes `base` as a function of the initial value of the loop counter, and then jumps to the appropriate inner DO loop, with the inner loop counter `ii` properly initialized. In this case, extra code must be executed before execution of the loop proper begins. However, since each loop is to be executed many times, the benefits obtained by achieving higher effective memory bandwidth easily offset the overhead of executing extra prolog code. Furthermore, in horizontally microcoded and VLIW architectures this prolog code can frequently be executed in parallel with the code that precedes the loop. For the class of scientific programs, the compiler will likely have enough information at compile time so that loop prolog code will not be necessary.

The examples shown below, in increasing order of complexity, can be handled by small variations of the compilation techniques described above to allow concurrent access to A[i] and A[i+K]:

Case 1:

```
do 10 i=1,N
```

```
10     ..A[i]+A[i+K]     for MSD[K,M]
```

Case 2:

```
    do 10 i=1,N
10     ..A[i]+A[i+K]     for MSD[K',M]
```

Case 3:

```
    do 10 i=1,N,STEP
10     ..A[i]+A[i+K]     for MSD[K,M]
```

Case 4:

```
    do 10 i=1,N,STEP
10     ..A[i]+A[i+K]     for MSD[K',M]
```

6.5 Examples

In is section, the application of this method to a simple inner loop is illustrated. Extension and application of this loop to an FFT example is illustrated in Chapter 7. Assume the following FORTRAN loop:

```
DIMENSION A[256,256]
                  DO 10 I = 1, 256
                  DO 10 J = 1, 256
  10              X + A[I,J] + A[I+1,J] + A[I, J+1]
                        (1)         (2)         (3)
```

Assuming A is stored in row-major organization, the index of element A[I,J] can be expressed as I*256+J. By making `I'= I*256 + J`, the above array references (1), (2) and (3) become:

(1) `A[I']`

(2) `A[I' + 256]`

(3) `A[I' + 1]`

A distribution schema for this array can be found by using the method described earlier. The system of equations to be solved is:

$K \leq 1 \bmod K*M \leq (M-1)*K$
$K \leq 255 \bmod K*M \leq (M-1)*K$
$K \leq 256 \bmod K*M \leq (M-1)*K$

It is easily seen that (K,M) = (1,6) is a solution by searching for solutions starting from the minimum number of banks, M=3, because three concurrent accesses are performed.

This example assumes that the matrix is organized in row-major order, so element `A[i,j]` is indexed by `I*256+J`. Another possibility is to have the array organized in column-major order. In that case, element `A[i,j]` would be addressed as `J*256+I`. The choice of organization for the array influences the number of memory banks that are required for concurrent access. For example, for a 256*256 matrix `A`, an MSD distribution that allows concurrent access to elements A[I,J] and A[I+1,J] requires three memory banks if the matrix is organized in row-major order, but only two memory banks if the array is organized in column-major order. In general, the compiler must find the best organization for each array by examining all PAS that reference that array, and choosing the organization which will require the least number of memory banks. This problem is similar to and may use techniques from the choice of vector 'shapes' in compilation for vector machines[57].

The application and extension of this technique to the innermost loop of the FFT algorithm is now presented. The FFT algorithm is composed of two phases: a bit-reversal transformation and a combining phase. The focus here is on the combining phase of this algorithm. The overall structure of the FFT algorithm is illustrated in Figure 6.5.

The VLIW code for the innermost loop is presented below. This code is generated during specification optimization by the retargetable optimizing microcode

```
shuffle(); /* perform bit-reversal transformation */
FOR i:=1, 2, 4, 8, ... , n/2{ /* combining phase*/
    FOR j=0 to n-1 by 2*i{
    compute W=f(i);      -- Trigonometric Recurrence
      FOR k:= 0 to i-1{
  Z    =v[k+j+i]*W -- Butterfly operation
  v[k+j]  =v[k+j]+Z
  v[k+j+i]=v[k+j]-Z }}}
```

Figure 6.5: FFT Algorithm

compiler from a sequential description of the FFT innermost loop which contains 22 instructions. In this figure, each VLIW instruction starts with a numeric label followed by the list of operations in the instruction.

```
; FFT combining phase: Danielson-Lanczos section of FFT routine
; Assumes that the Pre-loop code  initializes the following registers:
;         L1 as the address of v[K+J],
;         L2 as the address of v[K+J+I],
;         W as exp(2* PI/I)  "twiddle factor"
;         F_J_I as the address of v[K+J+I] (loop termination condition)
;
1: F1.r=BR(L1)   ; F1.i=BI(L1)   ; Load array elements v[K+J],v[K+J+I]
   F2.r=BR(L2)   ; F2.i=BI(L2)   ; from memory
2: T1=F1.r*W.r; T2=F1.r*W.i; Complex Multiply
   T3=F1.i*W.r; T4=F1.i*W.i;
3: Z.r = T1 - T4; Z.i = T2 + T3 ; Butterfly operation
4: F2.r=F2.r+Z.r;  F1.r=F2.r-Z.r; Store new v[K+J],v[K+J+I] back into memory
   F2.i=F2.i+Z.i;  F1.i=F2.i-Z.i; cc1=(L1<F_J_I)
5: BR(L1)=F1.r   ; BR(L2)= F2.r;
   BI(L1)=F1.i   ; BI(L2)= F2.i; L1=L1+1; L2=L2+1   ; IF cc1 GOTO 1:
```

In the above VLIW code, Instruction 1 loads the array elements into Registers L1 and L2; Instructions 2 and 3 perform the complex multiplication by the twiddle factor, Instruction 4 performs the butterfly operation (2-point FFT), and Instruction 5 stores the new values of the array elements.

Let l=k+j. In the innermost loop of the FFT, the array elements referenced in the butterfly operation are v[l] and v[l+i], for values of i which are powers of two. Let B be the number of memory banks in the machine. It is easily seen that the traditional techniques for memory interleaving cause bank collisions on the memory reads in Instruction 2 and in the memory writes in Instruction 5 above, whenever the value of i is a multiple of B. In this case, registers L1 and L2 contain the addresses of array elements `v[k+j]` and `v[k+j+i]`, which are located in the same memory bank because their addresses differ by `i`. Thus, even though the above innermost loop has only five instructions, collision in memory access

by the memory fetch operations in Instruction 1, and by the store operations in Instruction 5, cause each loop iteration to take 7 cycles, assuming 1-cycle memory accesses. This reduces the achievable performance for this loop down to 71%. Furthermore, the degradation in performance is even larger in systems with more realistic multi-cycle memory access times. Note that, if the memory access time is similar to the time for the complex multiplication in Instruction 2, the load of A[i+k] might be delayed to Instruction 2, thus avoiding the conflict in memory access for the memory fetch operations. However, for longer memory latency, and for the store operations in Instruction 5, the conflict persists.

For the FFT code, the value of n must be a power of two, and for ease of hardware implementation the number of memory banks is also a power of two. Because of this fact, the above memory access collisions will always happen in real machines. Thus, the memory accesses in the butterfly operation will be performed serially, even though the data dependencies allow the operations to be done in parallel.

The MSD method can be used to solve this problem, by finding an MSD(K,M) that allows concurrent access to PAS(i,i+K_i) for all values of K_i equal to 1, 2, 4, 8, ..., n/2, leading to an improvement of about 30% in the performance. In this case, MSD(1,3) is a good organization. This is confirmed by verifying the conditions of Theorem 6.2 for all expected values of K_i. Alternatively, it suffices to notice that MSD(1,3) stores element i in bank *i mod 3*. Element *i+K*, where *K* is a power of two, is stored in bank *(i + K) mod 3*. Because *K* is a power of two, *K mod 3* is never zero, and thus elements *i* and *i+K* are stored in distinct memory banks.

Note that, for this FFT example, this straightforward application of the MSD method is able to achieve memory bandwidth comparable to that of the work of Lin and Ho [55], which was obtained after careful study of characteristics of the FFT algorithm. They noticed that the indices of the elements involved in the butterfly operation have different parity. By separating the vector in two memory banks, the first containing those elements whose indices have even parity and the

second with the elements whose indices have odd parity, the memory accesses can be performed concurrently. For large FFTs, the bandwidth of memory access is responsible for a large portion of the performance, and thus our method provides considerable improvement, because the effective memory bandwidth is doubled at the innermost loop of the FFT.

6.6 Extensions

If the target memory bank of each memory reference operation is not known at compile time, concurrent memory access requires an expensive interconnection structure to route the data from the referenced memory bank to the appropriate functional unit. The capability to detect and handle access collisions to a single memory bank is also necessary. Alternatively, the compiler may avoid scheduling multiple memory accesses in case of doubt, thus sacrificing potential performance. Many classes of loop exhibit access patterns such that it is possible to know the memory bank of each memory reference. This is called *bank-disambiguation* by [35]. To achieve bank disambiguation with the MSD method, [48] suggests unrolling the loop. However, unrolling loops may not be a desirable solution due to reasons such as increased code size. An alternative solution which avoids the need for loop unrolling is now presented.

The key idea is to keep track of the memory banks referenced in each iteration, and to enhance the switches connecting memory busses and register files busses. This target memory bank number of each memory access is computed concurrently during program execution. The enhanced switch takes the number of the target memory bank as part of the switch control,

Figure 6.6 illustrates the FFT source code with bank control. Figure 6.7 illustrates the corresponding VLIW code. In Figure 6.7, `[BR(L1,B0)=F1.r]` denotes: store register F1.r into location L1, bank B0. The switches connecting the register holding the target memory address are set up to perform this access conditionally on a bank number supplied as a control argument. (see Figure 7.29). A register provides the switch with run-time computed bank number;

in this example, registers `B0` and `B1`. Only the memory bank whose number matches the bank number selection register is activated by the switch. In the FFT example this requires the computation of the recurrences for `(j+k)mod 3` and `(j+k+i)mod 3` in the innermost loop. The operations to keep track of the memory bank number are executed in parallel with the loop iteration.

The compilation algorithms required to generate this code are similar to the algorithms to perform loop unrolling to achieve bank disambiguation. This can be done automatically for those loops whose indices evolve via simple recurrences. Furthermore, for loops where the initial values of the indices, which are the basis for recurrence, are not known at compile time, a pre-loop may be added to compute the initial memory banks dynamically, prior to the loop proper. This is inspired by the pre-loop technique suggested in [35]. This computation adds a small overhead which is executed only once before the loop, but provides much enhanced memory bandwidth at low cost. Furthermore, the addition of such pre-loop extends the applicability of the MSD technique to a much larger class of loops.

```
for(i=1,B1i=1; i<=(n-1); B1i = (B1i+i)mod 3,i=i+i){
for(j=0,B0j=0,B1j=B1i;
j<=(n-1);(j=j+2*i),B0j=((B0j+2*i)mod 3),B1j=((B1j+(2*i))mod 3)){
for(k=0,B0=B0j,B1k=B1j;
k<=(i-1);
k=k+1,B0=((B0+1) mod 3),B1=((B1k+1) mod 3)){

....B0[L0] is v[k+j], and
....B1[L1] is v[k+j+i]
..butterfly operation ..

  /* update bank number and array index    */
  /* this code is executed in parallel with loop control*/
  /* control     */
B0=B0+1;if(B0>2){B0=0;L1=L1+1}
B1=B1+1;if(B1>2){B1=0;L2=L2+1}
}
}
```

Figure 6.6: FFT Algorithm With Code to Compute Memory Bank Number

```
$1: F0.r=BR(L0,B0)   ; F0.i=BI(L0,B0)   -- Load  v[K+J],v[K+J+I]
    F1.r=BR(L1,B1)   ; F1.i=BI(L1,B1)   -- from memory
    New_B0 = B0 + 1 ; New_B1 = B1 + 1 -- keep track of bank number
$2: T1=F0.r*W.r; T2=F0.r*W.i        -- Complex Multiply
    T3=F0.i*W.r; T4=F0.i*W.i;
    New_L0 = L0 +1   ;        -- prepare to increment
$3: Z.r = T1 - T4; Z.i = T2 + T3       -- Butterfly operation
    cc3= New_B0 > 2; cc4= New_B1 > 2; -- bank number wrap-around ?
    New_L1 = L1 +1   ;        -- prepare to increment
$4: F1.r=F1.r+Z.r;  F0.r=F1.r-Z.r;
    F1.i=F1.i+Z.i;  F0.i=F1.i-Z.i; cc1=(L0<F_J_I);
$5: BR(L0,B0)=F0.r   ; BR(L1,B1)= F1.r -- Store new v[K+J],v[K+J+I]
    BI(L0,B0)=F0.i   ; BI(L1,B1)= F1.i -- back into memory
    if(cc3) B0 = New_B0 else {B0 = 0; L0 = New_L0;}
    if(cc4) B1 = New_B1 else {B1 = 0; L1 = New_L1;}
    IF cc1 GOTO $1:
```

Figure 6.7: FFT Inner Loop Code With Memory Bank Control

Chapter 7

Example Applications of ASPD Method

This chapter illustrates the ASPD Method by describing experimental application on a number of selected examples. The benchmarks have been chosen to illustrate characteristics of the ASPD method and to allow comparison with previous designs such as the White Dwarf project and some existing High-Level Synthesis benchmarks.

7.1 Synthesis, Validation and Evaluation Procedure

This section describes the procedure to verify correctness and evaluate performance of the example designs. The VLIW code has been valided by a simulator, by comparison of the results of simulation of the serial NADDR program with the results produced by a VLIW architecture simulator. The simulator, integrated with the compiler code, works from the compiler's data structures. Speedup is given by the ratio of the cycle times required to simulate the serial execution to the number of cycles for VLIW execution. The simulation is favorable for the serial execution because it assumes that unconditional jumps in the serial NADDR code can be executed in zero cycles. To check correctness of the simulation, some of the examples in this chapter have been further validated by simulation by two other VLIW simulators. These simulator are obtained from independent

research projects at CMU. One is the simulator for the XIMD architecture[88], an extension of the canonical VLIW. It is capable of simulating VLIW code that does not require conditional execution of operations. The second simulator is a result of a class project to use VLIW compilation to investigate fine-grained parallelism and super-scalar architectures.

The results of Implementation Optimization are validated by checking for resource contention. A simple AWK script is used to verify correctness of graph coloring. A "C" simulator is constructed, by hand, for one small example. A proposed validation approach is to change the VLIW program listing module to output valid "C" source code that simulates VLIW execution. Each VLIW instruction is associated with a labeled sequence of C code. The code contains two declarations for each register in the VLIW program, corresponding to the global and local values of the variable. As each VLIW operation is simulated, it reads its arguments from the global set of variables, and stores its result in the local copy. Before the simulated VLIW instruction jumps to next instruction, the local values are copied, in order, into the global values to preserve the VLIW program semantics. This alternative is advantageous because it allows simulation of large input programs at the system's speed. Furthermore, the system's debugging system may be used to inspect the customized VLIW simulator code. This approach has been explored preliminarily, by recognizing a subset of NADDR operations. It remains to be implemented code to translate all known NADDR instructions into C code. An interesting variant of this alternative is to output code for a high-level simulation language such as Verilog. This language allows creation of instances of library hardware modules, specification of their connectivity and simulation of the resulting design.

7.2 Detailed Example: MIN (Livermore Kernel 24)

This section presents the ASP design for a simple program that finds the minimum of an array. This program is Livermore Kernel 24. The C source code

and NADDR intermediate code for this example are illustrated in Figure 4.2 in Chapter 4. Figure 7.1 illustrates the result of Specification Optimization which achieves throughput of 1 cycle/iteration. Each VLIW instruction is printed with one operation per line. The definition number corresponding to each operation is presented in the same line after the operation code in square brackets. For example, the opcode for definition 16, which is found in Instruction 1014a3a0, is

```
(ige $cc1[ 1 ]  $u'[3 ]  $u[4 ]  )[16]
```

The number of the register file holding each argument is listed in square brackets. The first argument `$u'` is in register file 3, and the second argument, `$u`, is in register file 4. The resulting value of `$cc1` is written into register file 1. In case the operation writes more than one destination, multiple register file values are listed. The set of reaching definitions is listed after each instruction.

Figure 7.2 presents the result of applying ADJUST-GRAPH, Graph G+, to this program. Each node in the graph is labeled with the corresponding variable name, the color assigned to the node, followed by a list of definition numbers and the instruction where the defined value is used. For example, the top node in Figure 7.2 illustrates definition 2 of variable `$min` that is used at Instruction 101477d8. This is not a claw-free-perfect graph, because the subgraph induced by the nodes that have a * symbol preceding the variable name in Figure 7.2 is a claw. The graph-coloring heuristic used 6 colors for vertex-coloring of this graph, which is optimal because the graph contains a 6-clique. This clique is composed of the 6 nodes at the center of the figure, identified by the + symbol after the variable name. This example illustrates how condition codes are handled in the same manner as other register values. Condition code are allocated uniformly in the register file, and may be operated upon as any other data. Alternatively, a realization technology may elect to store condition code values into separate single-bit register banks. This is easily accomplished by enhancing the ADJUST-GRAPH procedure to introduce interference edges in G+ between vertices corresponding to condition codes and vertices corresponding to integer or floating point register values. This causes allocation of condition code values

```
101467bc:
    (iconstant $0[ 1 ]   0 )[0]
    (iconstant $i[ 5 ]   0 )[1]
    (goto 10146cd0)
Reaching Defs:  {  }

10146cd0:
    (ivload $min[ 6 ]   0 $i[5 ]  )[2]
    (ivload $u[ 4 ]   0 $i[5 ]  )[3]
    (iadd $i[ 5 ]  $i[5 ]   4 )[4]
    (goto 101477d8)
Reaching Defs:  { 0 1  }

101477d8:
    (ige $cc1[ 1 ]  $u[4 ]  $min[6 ]  )[5]
    (ile $cc2[ 2 ]  $i[5 ]   10 )[6]
    (ivload $u'[ 3 ]   0 $i[5 ]  )[7]
    (iadd $i[ 5 ]  $i[5 ]   4 )[8]
    (goto 1014a3a0)
Reaching Defs:  { 0 2 3 4  }

1014a3a0:
    (if $cc1[1 ]  L15
      (if $cc2[2 ]  Lloop
        (iassign $u[ 4 ]  $u'[3 ]  )[9]
        (ige $cc1[ 1 ]  $u'[3 ]  $min[6 ]  )[10]
        (ile $cc2[ 2 ]  $i[5 ]   10 )[11]
        (ivload $u'[ 3 ]   0 $i[5 ]  )[12]
        (iadd $i[ 5 ]  $i[5 ]   4 )[13]
        (goto 1014a3a0)
      ELSE
        (goto 1014acec))
    ELSE
      (iassign $min[ 6 ]  $u[4 ]  )[14]
      (if $cc2[2 ]  Lloop
        (iassign $u[ 4 ]  $u'[3 ]  )[15]
        (ige $cc1[ 1 ]  $u'[3 ]  $u[4 ]  )[16]
        (ile $cc2[ 2 ]  $i[5 ]   10 )[17]
        (ivload $u'[ 3 ]   0 $i[5 ]  )[18]
        (iadd $i[ 5 ]  $i[5 ]   4 )[19]
        (goto 1014a3a0)
      ELSE
        (goto 1014acec)))
Reaching Defs:  { 0 2 3 5 6 7 8 9 10 11 12
   13 14 15 16 17 18 19  }

1014acec:
    (ivstore  0 $0[1 ]  $min[6 ]  )[20]
    (goto 1014af94)
Reaching Defs:
 { 0 2 3 5 6 7 8 9 10 11 12
   13 14 15 16 17 18 19  }

1014af94:
    (igoto 10144170)
Reaching Defs:
 { 0 2 3 5 6 7 8 9 10 11 12
   13 14 15 16 17 18 19 20  }
```

Figure 7.1: min.c VLIW code with Reaching Definitions and Register File Allocation

into banks separate from the register file banks. Consequently, condition code banks can be realized as single-bit wide banks.

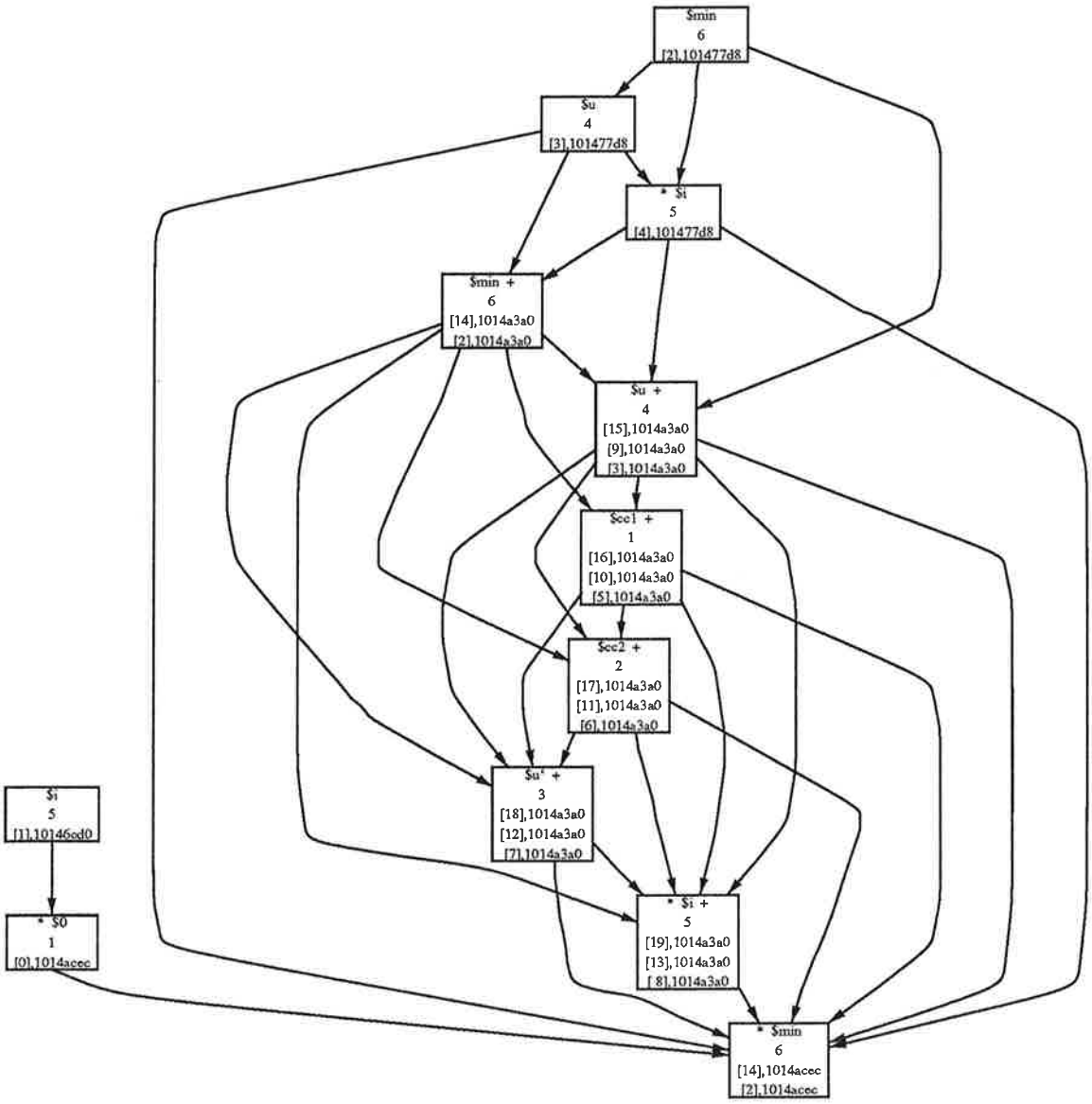


Figure 7.2: Example: min.c Interference Graph

Following register files allocation, operations are assigned to functional units. The resulting design is illustrated in Figure 7.3. The list of operations assigned to each functional unit is presented under it. It requires one memory bank, two comparators, an adder, and two latches to perform COPY operations. Simulated execution of this loop using as input the short vector composed of the 6 elements {99,5,99,3,99,4} takes 8 cycles. The serial NADDR execution requires 24 cy-

cles; there is a speedup of 3 over the serial NADDR execution by the optimized code. An speedup of 5.84 is obtained with simulated execution of the same loop on a larger array with 100 randomly generated elements. Specification optimization of this example takes 0.21 CPU seconds on a DECStation 5000. CPU time is measured with the UNIX "time" command. With the added execution of register files allocation and graph coloring routines, the total runtime is 0.39 seconds.

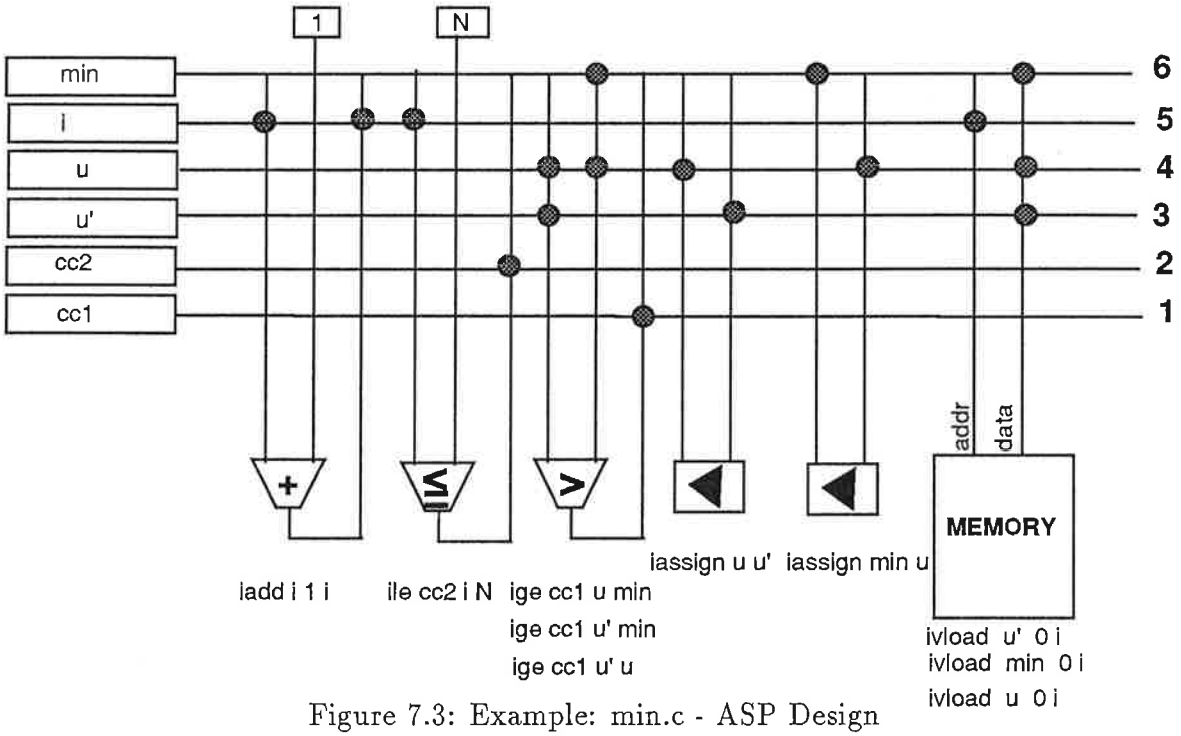


Figure 7.3: Example: min.c - ASP Design

7.3 Detailed Example: Vector Scaling

This section presents the ASP design for a simple program to multiply all elements of an array by a constant. This simple design is chosen to illustrate the memory allocation technique. The loop source code in C and the corresponding NADDR code are illustrated in Figure 7.4.

Figure 7.5 presents the VLIW code that achieves throughput of 1 iteration/cycle, for an speedup of 5 over the serial NADDR code. This code requires one multiplier, one adder, two latches to execute copy operations, and a memory system that supports two concurrent accesses. In the loop steady state instruc-

```
while(TRUE)
    {
        i++;
        A[i]=A[i]*2;
    }
```

```
(PROC_BEGIN x
(LABEL lup)
(IADD $a 4 $a)
(IVLOAD $x 0 $a)
(IMUL $x 2 $x)
(IVSTORE 0 $a $x)
(GOTO (LABEL lup))
(PROC_END x)
)
```

Figure 7.4: Vector Scaling Example: C Source Code and NADDR Code

```
101467b4:
    (iconstant $i[ 4 ]   1 )[0]
    (goto 10146ca0)
Reaching Defs:
 {  }
10146ca0:
    (iadd $i[ 4 ]   4 $i[4 ]  )[1]
    (goto 10146f48)
Reaching Defs:
 { 0  }
10146f48:
    (ivload $a[ 2 ]   0 $i[4 ]  )[2]
    (iadd $i'[ 1 ]   4 $i[4 ]  )[3]
    (goto 101474b8)
Reaching Defs:
 { 1  }
```

```
101474b8:
    (imul $a[ 2 ]   2 $a[2 ]  )[4]
    (ivload $a'[ 3 ]   0 $i'[1 ]  )[5]
    (iadd $i''[ 5 ]   4 $i'[1 ]  )[6]
    (goto 10147770)
Reaching Defs:
 { 1 2 3  }
10147770:
    (ivstore  0 $i[4 ]  $a[2 ]  )[7]
    (iassign $i[ 4 ]  $i'[1 ]  )[8]
    (iassign $i'[ 1 ]  $i''[5 ]  )[9]
    (imul $a[ 2 ]   2 $a'[3 ]  )[10]
    (ivload $a'[ 3 ]   0 $i''[5 ]  )[11]
    (iadd $i''[ 5 ]   4 $i''[5 ]  )[12]
    (goto 10147770)
Reaching Defs:
 { 1 3 4 5 6 7 8 9 10 11 12     }
```

Figure 7.5: Vector Scaling: VLIW Code With Register File Allocation

tion, two memory accesses are performed, a `load` and a `store`. An ASP design for this loop is illustrated in Figure 7.6, assuming a memory system that supports two concurrent memory accesses.

The byte addresses of the two memory references in the loop steady state instruction always differ by 8. In this example, memory is byte addressable, with an word size of 32 bits. Thus, the MSD technique is applicable to this loop. Figure 7.7 illustrates the code after MSD is applied.

7.4 Synthesis Benchmark: Elliptic Filter

This filter is a benchmark from the 1988 High Level Synthesis Workshop. The loop presents a large number of inter-iteration dependencies. The NADDR code for the loop, composed of 34 operations, plus a jump, is illustrated in Figure 7.8. The inter-iteration optimized code requiring 4 adders and 2 multipliers, is illustrated in Figures 7.9 and 7.10.

The VLIW code executes one iteration every 13 cycles, thus achieving an

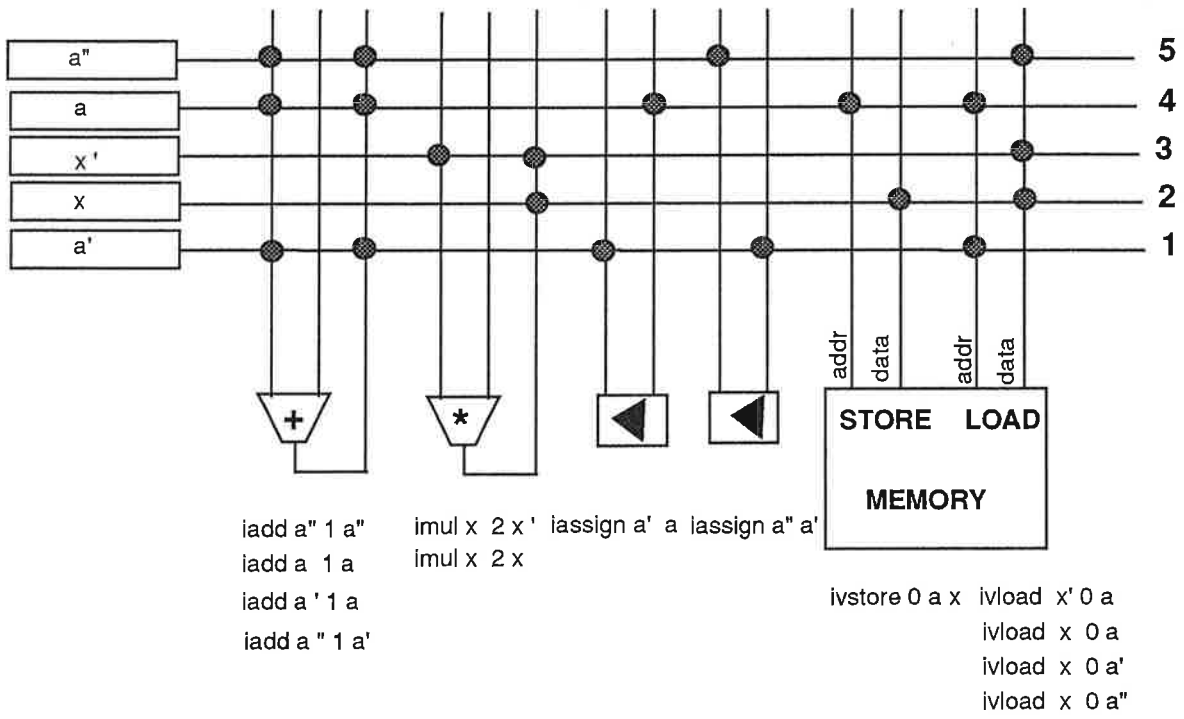


Figure 7.6: Vector Scaling Example - ASP Design

speedup of 2.61. The interference graph for this example, illustrated in Figure 7.11, is not claw-free perfect, and has been colored with 6 colors. This number of colors is is optimal because the graph contains a 6-clique. In this example, allocation of operations to operators is performed by a simple first-come first-served heuristic. Figure 7.12 illustrates the result of implementation optimization. The allocation of variable into register files is presented in Figure 7.13. Depending on architecture realization choices, if ALUs with the capability of execution of addition and multiplication are available, this filter may be implemented by using only 4 functional units.

The bus broadcast feature of the implementation architecture is exercised in this example. In Instruction 10124e94 of Figure 7.9, operation (`iadd $op8[ 1 2 ] $op3[6 ] $op9[2 ] )` writes the resulting value of $8 in register files 1 and 2. Later in the program, Instruction 1012545c uses the value stored into register file 2, as illustrated in Figure 7.10. (In that instruction, register file 1 is busy providing the value of `$sv9`). Instruction 10127718 uses the value of $8

```
Vector Scale Loop  pipelined code,
with MSD(2,2) applied

inner loop unrolled


101467b4:
    (iconstant $i[ 4 ]   1 )[0]
    (goto 10146ca0)
Reaching Defs:
 {  }
10146ca0:
    (iadd $i[ 4 ]   4 $i[4 ]  )[1]
    (goto 10146f48)
Reaching Defs:
 { 0  }
10146f48:
    (ivload $a[ 2 ]   0 $i[4 ]  )[2]  load from bank 0
    (iadd $i'[ 1 ]   4 $i[4 ]  )[3]
    (goto 101474b8)
Reaching Defs:
 { 1  }
101474b8:  load from bank 0
    (imul $a[ 2 ]   2 $a[2 ]  )[4]
    (ivload $a'[ 3 ]   0 $i'[1 ]  )[5]
%    (iadd $i''[ 5 ]   4 $i'[1 ]  )[6]
    (isub $i''[ 5 ]   4 $i'[1 ]  )[6]
    (goto 10147770)
Reaching Defs:
 { 1 2 3  }

after the preceding instr,
i'' must have "0" instead of 2, so
it gets decremented, instead of incremented


/* STORE always use i */

10147770:  load from bank 1, offset 0, use i
    (ivstore  0 $i[4 ]  $a[2 ]  )[7]
    (iassign $i[ 4 ]  $i'[1 ]  )[8]
    (iassign $i'[ 1 ]  $i''[5 ]  )[9]
    (imul $a[ 2 ]   2 $a'[3 ]  )[10]
%    (ivload $a'[ 3 ]   0 $i''[5 ]  )[11]
    (ivload $a'[ 3 ]   0 $i[5 ]  )[11]  bank 1
    (iadd $i''[ 5 ]   4 $i''[5 ]  )[12]
    (goto 1014777-Unroll1)

increment i

10147770-Unroll1: load from bank 1, use i
    (ivstore  0 $i[4 ]  $a[2 ]  )[7]
    (iassign $i[ 4 ]  $i'[1 ]  )[8]
    (iassign $i'[ 1 ]  $i''[5 ]  )[9]
    (imul $a[ 2 ]   2 $a'[3 ]  )[10]
%   (ivload $a'[ 3 ]   0 $i''[5 ]  )[11]
    (ivload $a'[ 3 ]   0 $i[5 ]  )[11]  bank 1
%    (iadd $i''[ 5 ]   4 $i''[5 ]  )[12]
    (isub $i''[ 5 ]   4 $i''[5 ]  )[12]
    (goto 1014777--Unroll2)

decrement i, reverse banks

10147770-Unroll2:  load from bank 0, offset 2, use i
    (ivstore  0 $i[4 ]  $a[2 ]  )[7]
    (iassign $i[ 4 ]  $i'[1 ]  )[8]
    (iassign $i'[ 1 ]  $i''[5 ]  )[9]
    (imul $a[ 2 ]   2 $a'[3 ]  )[10]
%    (ivload $a'[ 3 ]   0 $i[5 ]  )[11]  bank 1
    (ivload $a'[ 3 ]   2 $i[5 ]  )[11]  bank 1
    (iadd $i''[ 5 ]   4 $i''[5 ]  )[12]
    (goto 10147770-Unroll3)

/* increment i */

10147770-Unroll3:  load from bank 0,offset 2, use i
    (ivstore  0 $i[4 ]  $a[2 ]  )[7]
    (iassign $i[ 4 ]  $i'[1 ]  )[8]
    (iassign $i'[ 1 ]  $i''[5 ]  )[9]
    (imul $a[ 2 ]   2 $a'[3 ]  )[10]
%    (ivload $a'[ 3 ]   0 $i[5 ]  )[11]  bank 1
    (ivload $a'[ 3 ]   2 $i[5 ]  )[11]  bank 1
%    (iadd $i''[ 5 ]   4 $i''[5 ]  )[12]
    (isub $i''[ 5 ]   4 $i''[5 ]  )[12] ; DECREMENT i''
; so i gets decr later
    (goto 10147770)

/* increment i, go back to loop */
```

Figure 7.7: Vector Scaling: VLIW Code With MSD(2,2) Data Allocation for Concurrent Access

```
(PROC_BEGIN LOOP
(iconstant $dummy 3)
(LABEL LOOP)
(iadd $op3  $inp  $sv2)
(iadd $op32  $sv33  $sv39)
(iadd $op12  $op3  $sv13)
(iadd $op20  $op12  $sv26)
(iadd $op25  $op20  $op32)
(imul $op21  $op25  2)
(imul $op24  $op25  2)
(iadd $op19  $op12  $op21)
(iadd $op27  $op24  $op32)
(iadd $op11  $op12  $op19)
(iadd $op22  $op19  $op25)
(iadd $op29  $op27  $op32)
(imul $op9  $op11  2)
(iadd $sv26  $op22  $op27)
(imul $op30  $op29  2)
(iadd $op8  $op3  $op9)
(iadd $op31  $op30  $sv39)
(iadd $op7  $op3  $op8)
(iadd $op10  $op8  $op19)
(iadd $op28  $op27  $op31)
(iadd $op41  $op31  $sv39)
(imul $op6  $op7  2)
(iadd $op15  $op10  $sv18)
(iadd $op35  $sv38  $op28)
(imul $outp  $op41  2)
(iadd $op4  $inp  $op6)
(imul $op16  $op15  2)
(imul $op36  $op35  2)
(iadd $sv39  $op31  $outp)
(iadd $sv2  $op4  $op8)
(iadd $sv18  $op16  $sv18)
(iadd $sv38  $sv38  $op36)
(iadd $sv13  $op15  $sv18)
(iadd $sv33  $sv38  $op35)
(GOTO (LABEL LOOP))
    (PROC_END LOOP)
)
```

Figure 7.8: Example: Fifth Order Elliptic Filter - NADDR code

stored in register file 1, and register file 2 is used to provide the value of $op4. This illustrates the global view taken by the algorithm to make use of the bus broadcast feature and minimize the overall number of required register files. The CPU time for specification optimization of this example is 2.7 CPU seconds on a DECStation 5000. With the added execution of register files allocation and graph coloring routines the total runtime is 2.9 seconds. Here the incremental cost of register files allocation is small. The reason is that this is a simple loops and thus reaching definitions flow analysis, which is performed before graph coloring, converges rapidly in one pass. Furthermore, the amount of parallelism in each VLIW instruction is relatively small, and consequently the interference graph is sparse.

```
10122158:
    (iadd $op3[ 6 ]  $inp[6]  $sv2[  ]  )[1]
    (iadd $op32[ 1 ]  $sv33[]  $sv39[ 1 ]  )[2]
    (goto 101227b8)
Reaching Defs:  {  }

101227b8:
    (iadd $op12[ 2 ]  $op3[6 ]  $sv13[ ]  )[3]
    (goto 10122ad0)
Reaching Defs:  { 0 1 2  }

10122ad0:
    (iadd $op20[ 2 ]  $op12[2 ]  $sv26[1 ]  )[4]
    (goto 10122df0)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10122df0:
    (iadd $op25[ 5 ]  $op20[2 ]  $op32[1 ]  )[5]
    (goto 101230f0)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

101230f0:
    (imul $op21[ 4 ]  $op25[5 ]   2 )[6]
    (imul $op24[ 3 ]  $op25[5 ]   2 )[7]
    (goto 1012372c)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

1012372c:
    (iadd $op19[ 4 ]  $op12[2 ]  $op21[4 ]  )[8]
    (iadd $op27[ 3 ]  $op24[3 ]  $op32[1 ]  )[9]
    (goto 10123d14)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10123d14:
    (iadd $op11[ 1 ]  $op12[2 ]  $op19[4 ]  )[10]
    (iadd $op22[ 2 ]  $op19[4 ]  $op25[5 ]  )[11]
    (iadd $op29[ 4 ]  $op27[3 ]  $op32[1 ]  )[12]
    (goto 10124604)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10124604:
    (imul $op9[ 2 ]  $op11[1 ]   2 )[13]
    (iadd $sv26[ 1 ]  $op22[2 ]  $op27[3 ]  )[14]
    (imul $op30[ 3 ]  $op29[4 ]   2 )[15]
    (goto 10124e94)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10124e94:
    (iadd $op8[ 1 2 ]  $op3[6 ]  $op9[2 ]  )[16]
    (iadd $op31[ 5 ]  $op30[3 ]  $sv39[1 ]  )[17]
    (goto 1012545c)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }
```

Figure 7.9: Example: Fifth Order Elliptic Filter VLIW code - Part 1

```
1012545c:   *** Register File 2  provides $op8 ***

    (iadd $op7[ 6 ]  $op3[6 ]  $op8[1 2 ]  )[18]
    (iadd $op10[ 4 ]  $op8[1 2 ]  $op19[4 ]  )[19]
    (iadd $op28[ 1 ]  $op27[3 ]  $op31[5 ]  )[20]
    (iadd $op41[ 2 ]  $op31[5 ]  $sv39[1 ]  )[21]
    (goto 10125fb0)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10125fb0:
    (imul $op6[ 3 ]  $op7[6 ]   2 )[22]
    (iadd $op15[ 2 ]  $op10[4 ]  $sv18[3 ]  )[23]
    (iadd $op35[ 4 ]  $sv38[5 ]  $op28[1 ]  )[24]
    (imul $outp[ 1 ]  $op41[2 ]   2 )[25]
    (goto 10126ba8)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10126ba8:
    (iadd $op4[ 2 ]  $inp[6]  $op6[3 ]  )[26]
    (imul $op16[ 4 ]  $op15[2 ]   2 )[27]
    (imul $op36[ 6 ]  $op35[4 ]   2 )[28]
    (iadd $sv39[ 1 ]  $op31[5 ]  $outp[1 ]  )[29]
    (goto 10127718)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }
```

```
10127718: *** Register File 1 provides  $op8 ***

    (iadd $sv2[ 1 ]  $op4[2 ]  $op8[1 2 ]  )[30]
    (iadd $sv18[ 3 ]  $op16[4 ]  $sv18[3 ]  )[31]
    (iadd $sv38[ 5 ]  $sv38[5 ]  $op36[6 ]  )[32]
    (goto 10127f70)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10127f70:
    (iadd $sv13[ 2 ]  $op15[2 ]  $sv18[3 ]  )[33]
    (iadd $op3[ 6 ]  $inp[6]  $sv2[1 ]  )[34]
    (iadd $sv33[ 3 ]  $sv38[5 ]  $op35[4 ]  )[35]
    (goto 10128510)
Reaching Defs:  { 0 1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20 21
   22 23 24 25 26 27 28 29 30 31 32
   33 34 35 36 37  }

10128510:
    (iadd $op32[ 1 ]  $sv33[3 ]  $sv39[1 ]  )[36]
    (iadd $op12[ 2 ]  $op3[6 ]  $sv13[2 ]  )[37]
    (goto 10122ad0)
Reaching Defs:  { 0 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 18 19 20 21 22
   23 24 25 26 27 28 29 30 31 32 33
   34 35 36 37  }
```

Figure 7.10: Example: Fifth Order Elliptic Filter VLIW code - Part 2

7.5 Synthesis Benchmark: GCD

This program, another benchmark from the 1988 High Level Synthesis Workshop, is a simple loop that finds the gratest common divisor of two numbers. Figure 7.14 illustrates the C source code and the corresponding NADDR code for this benchmark.

This example illustrates how the conditional-execution capabilities of the architecture template are crucial to achieving high performance. Specification Optimization achieves steady-state throughput of 1 cycle/iteration in the innermost loop, for a 5-fold speedup over the serial NADDR code (this assumes that unconditional jumps take zero cycles). On each loop iteration, the values of `y-x` and `x-y` are concurrently computed, loop termination (condition code `$cc0`) is verified, and condition codes for the comparison of the new values of `x` and `y` are found.

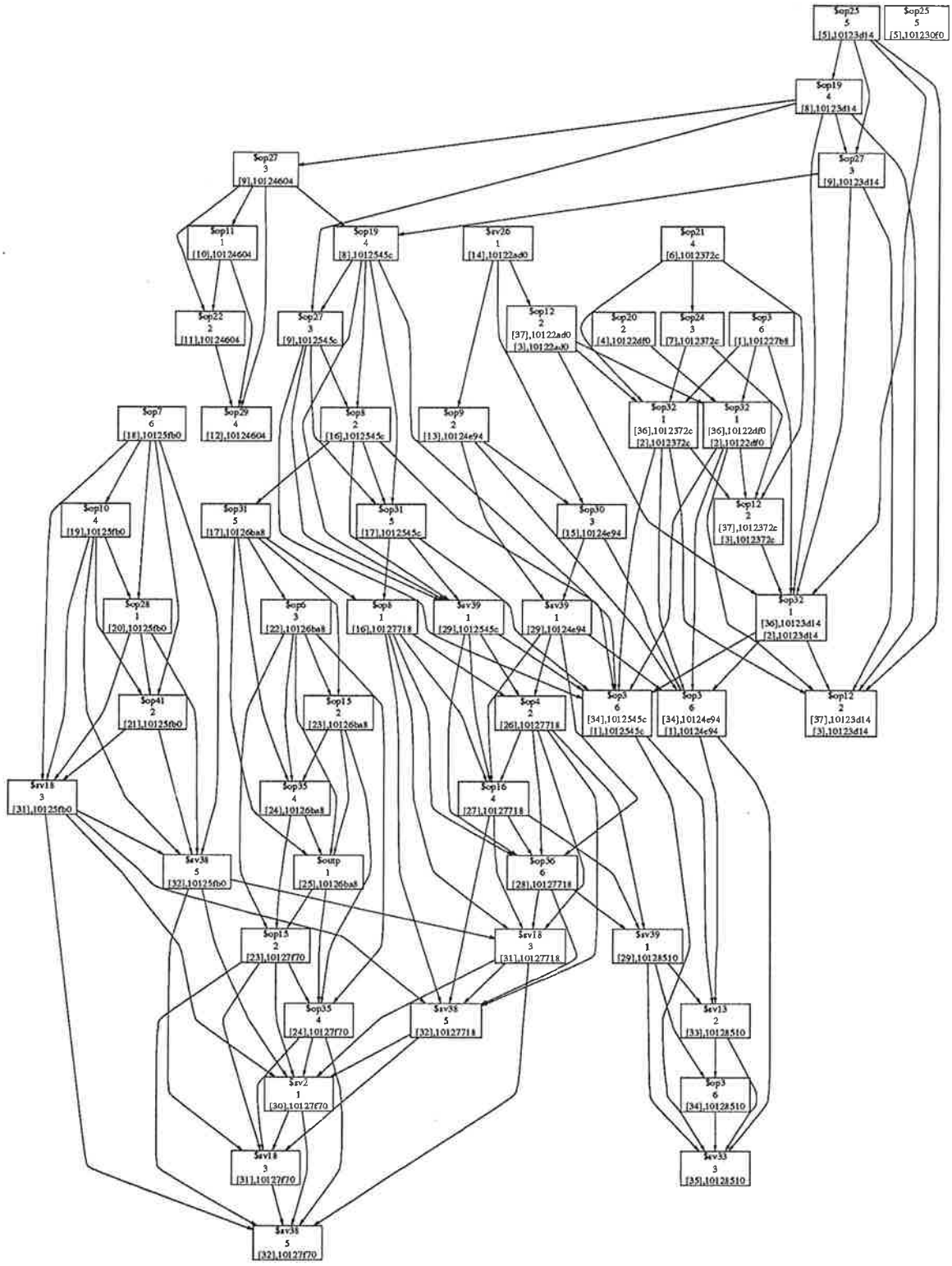


Figure 7.11: Example: Fifth Order Elliptic Filter - Interference Graph

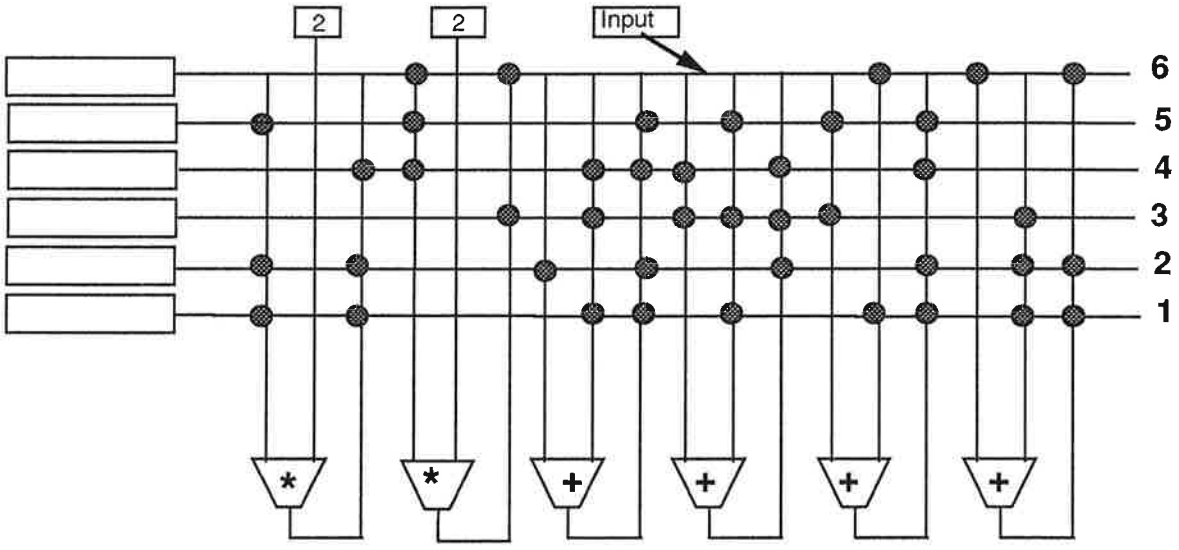


Figure 7.12: Example: Fifth Order Elliptic Filter - ASP Design

Register File	Variables Allocated
1	sv26, op32, op11, sv39, op28, op8, outp, sv2
2	op20, op12, op22, op9, op8, op41, op15, op4, sv13
3	op24, op27, op30, op6, sv18, sv33
4	op21, op19, op29, op10, op35, op16
5	op25, op35, sv38, op31
6	op3, op7, op36

Figure 7.13: Elliptic Filter ASP Design - Register File Allocation

The execution of each loop iteration has a latency of two cycles. In the first cycle, comparison of the current values of x and y is performed concurrently with the computation of y-x and x-y. In the second cycle, the condition codes assigned with the result of the comparisons determine the correct values of x and y for use by future iterations. If conditional-execution is not available, each iteration has a latency of three cycles to execute the comparison, the conditional jump to an appropriate code sequence, and the assignment of a new value to the appropriate register. Since each iteration requires the values of both x and y in its first cycle, a two-cycle delay between consecutive iterations ensues. For this example, the absence of conditional execution of operations reduces by one half the achievable performance.

The result of ADJUST-GRAPH, illustrated in Figure 7.15, is a claw-free graph, and is colored with the optimal number of colors, 6. Figure 7.16 illustrates the result of Specification Optimization. This design requires 8 ALUs and 2 latches to perform COPY operations. It should be noticed that, depending on the realization technology, the operations

```
(ieq $cc0[ 1 ]  $3'[3 ]  $2[4 ]  )
(ige $cc1[ 2 ]  $3'[3 ]  $2[4 ]  )
```

may be assigned to the same ALU. The first operation subtracts $3' from $2, and the second operation compares the same values for equality. This reduces cost to 6 ALUs and 2 COPY units. Figure 7.17 illustrates the result of Implementation Optimization. The CPU time for specification optimization of this example is 0.2 CPU seconds on a DECStation 5000. The added execution of register files allocation and graph coloring barely alter the same total runtime, the difference being close to the resolution of the workstation's timing mechanism. For comparison, the IBM synthesis system with path-based scheduling, which also achieves a throughput of 1 cycle/iteration[18], has an execution time of 0.3 seconds on an IBM 3090/200 mainframe.

```
/*
 * gcd(x,y)
 *
 * the result is the largest integer
 * that divides evenly x and y
 *
 */

int retval;

main()
{
register int x,y;

while (x!=y){
if(x < y) y=y-x;
else   x=x-y;
}
retval = x;
}
```

```
(PROC_BEGIN main
;  (ICONSTANT $0 0)
(LABEL main)
(LABEL LOOP)
(IEQ  $cc0 $3 $2 )
(IF  $cc0 (LABEL L7))
(IGE  $cc1 $3 $2 )
(IF  $cc1 (LABEL L4))
(ISUB $2 $2 $3)
(GOTO (LABEL LOOP))
(LABEL L4)
(ISUB $3 $3 $2)
(GOTO (LABEL LOOP))
(LABEL L7)
(IVSTORE 0 $0 $3)
(IGOTO $31)
(PROC_END main)
)
```

Figure 7.14: Example: Greatest Common Divisor - C source code and NADDR Code

7.6 Linear-Phase B-Spline Filter Example

This section presents the feasible design points for a simple linear-phase B-spline filter[89] loop, based on repeated compilation with varying resource constraints. This entire experiment requires 47.5 CPU seconds on a DECstation 3100. When the compiler is allowed to use as many ALUs as needed, it finds the "optimal" issue rate of one iteration per cycle by using 16 ALUs. Figure 7.18 shows the iteration issue rates obtained by scaling down the number of functional units available. The filter algorithm is illustrated in Figure 7.19. The design point at 6 ALUs is chosen for implementation, and Figure 7.20 illustrates the VLIW code obtained by using 6 ALUs. The steady state code of the loop is shown on the right side of the figure. A throughput of one loop iteration every 3 cycles is achieved, for an speedup of 5.3 over the serial NADDR execution. The graph resulting from ADJUST-GRAPH is not claw-free, and the coloring heuristic uses 12 colors. Figure 7.21 illustrates the resulting filter design, and Figure 7.22 presents the allocation of variable into register files.

7.7 White Dwarf FEM Solver Example

Figure 7.23 illustrates the algorithm of the Finite-Element Matrix solver.

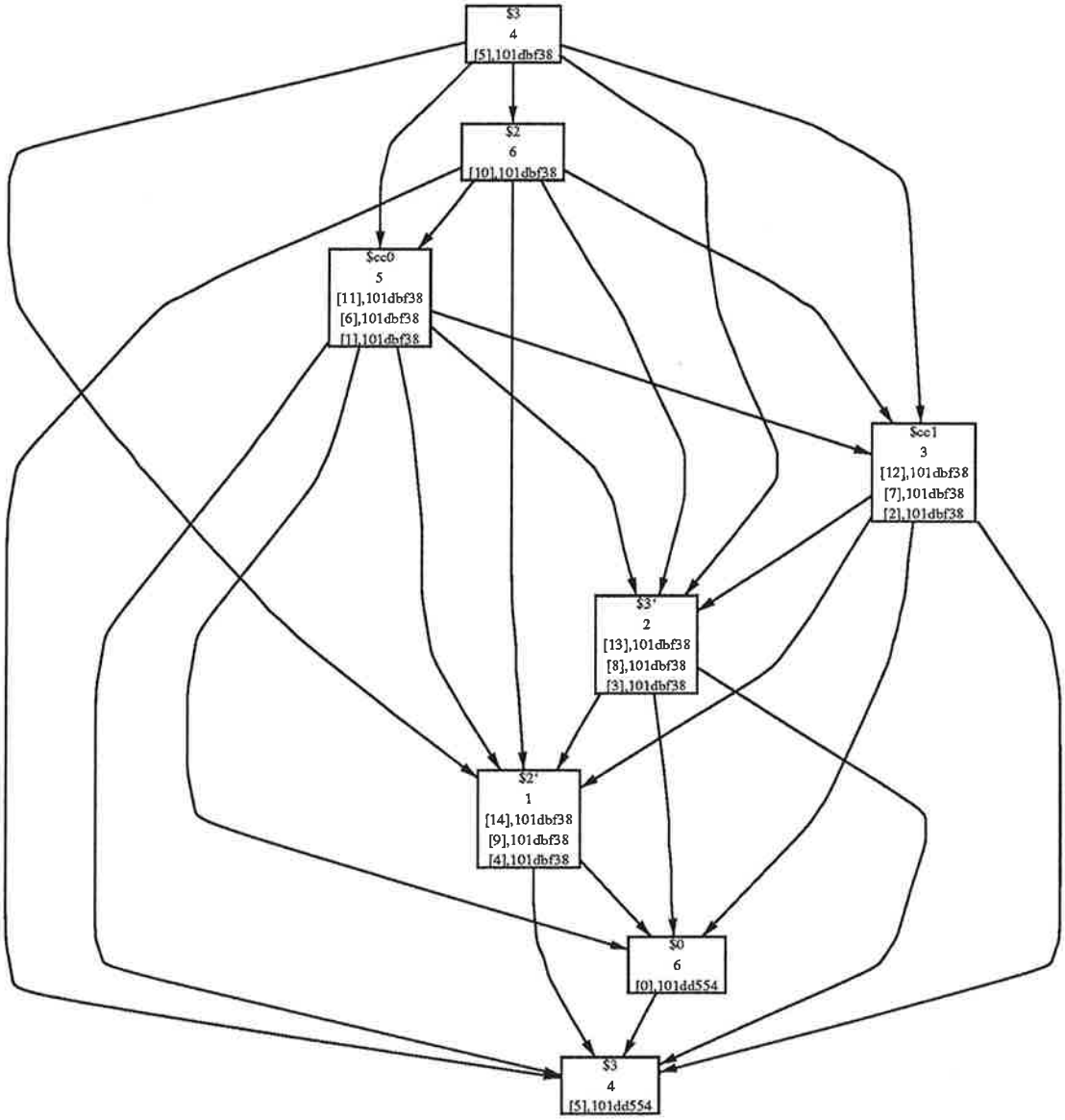


Figure 7.15: Example: Greatest Common Divisor - Interference Graph

```
101db460:
;   (iconstant $0[ 1 ]   0 )[1]
    (goto 101db71c)
Reaching Defs:
 {   }
101db71c:
    (iconstant $3[ 6 ]   21 )[2]
    (goto 101db9f8)
Reaching Defs:
 { 1  }
101db9f8:
    (iconstant $2[ 4 ]   27 )[3]
    (goto 101dbfa8)
Reaching Defs:
 { 1 2  }
101dbfa8:
    (ieq $cc0[ 1 ]  $3[6 ]  $2[4 ]  )[4]
    (ige $cc1[ 2 ]  $3[6 ]  $2[4 ]  )[5]
    (isub $3'[ 3 ]  $3[6 ]  $2[4 ]  )[6]
    (isub $2'[ 5 ]  $2[4 ]  $3[6 ]  )[7]
    (goto 101dc71c)
Reaching Defs:
 { 1 2 3  }
```

```
101dc71c:
    (if $cc0[1 ]  L17
      (goto 101ddd38)
    ELSE
      (if $cc1[2 ]  L14
        (iassign $3[ 6 ]  $3'[3 ]  )[8]
        (ieq $cc0[ 1 ]  $3'[3 ]  $2[4 ]  )[9]
        (ige $cc1[ 2 ]  $3'[3 ]  $2[4 ]  )[10]
        (isub $3'[ 3 ]  $3'[3 ]  $2[4 ]  )[11]
        (isub $2'[ 5 ]  $2[4 ]  $3'[3 ]  )[12]
        (goto 101dc71c)
      ELSE
        (iassign $2[ 4 ]  $2'[5 ]  )[13]
        (ieq $cc0[ 1 ]  $3[6 ]  $2'[5 ]  )[14]
        (ige $cc1[ 2 ]  $3[6 ]  $2'[5 ]  )[15]
        (isub $3'[ 3 ]  $3[6 ]  $2'[5 ]  )[16]
        (isub $2'[ 5 ]  $2'[5 ]  $3[6 ]  )[17]
        (goto 101dc71c)))
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17  }
101ddd38:
    (ivstore  0 $0[1 ]  $3[6 ]  )[18]
    (goto 101de00c)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17  }
101de00c:
    (igoto 101d8830)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 18  }
```

Figure 7.16: Example: Greatest Common Divisor - VLIW code and Register File Allocation

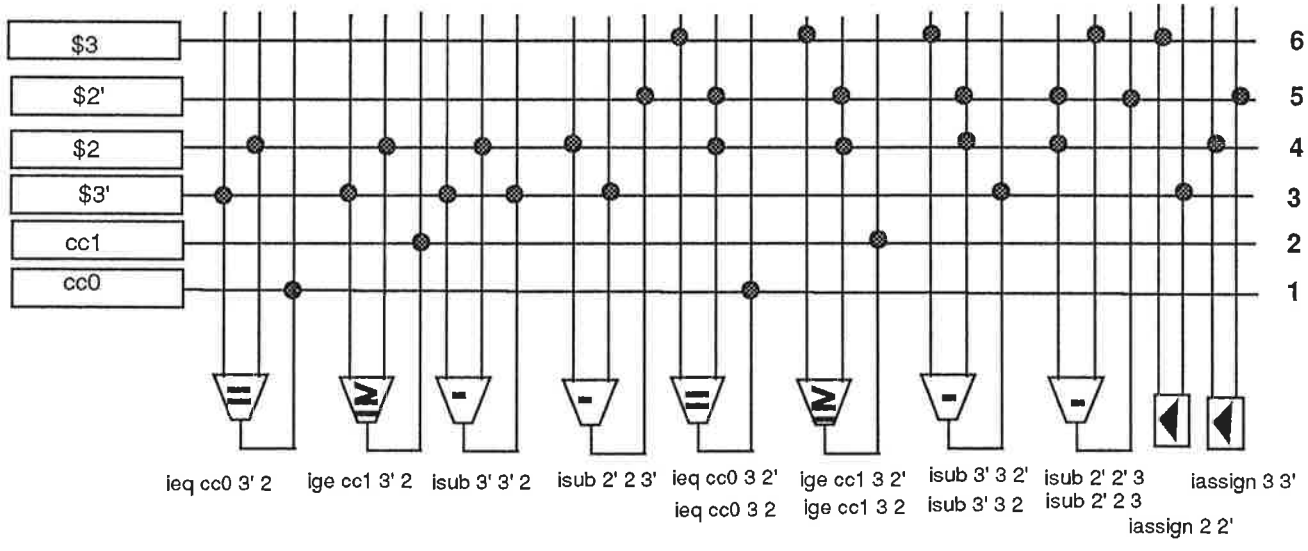


Figure 7.17: Example: Greatest Common Divisor - ASP Design

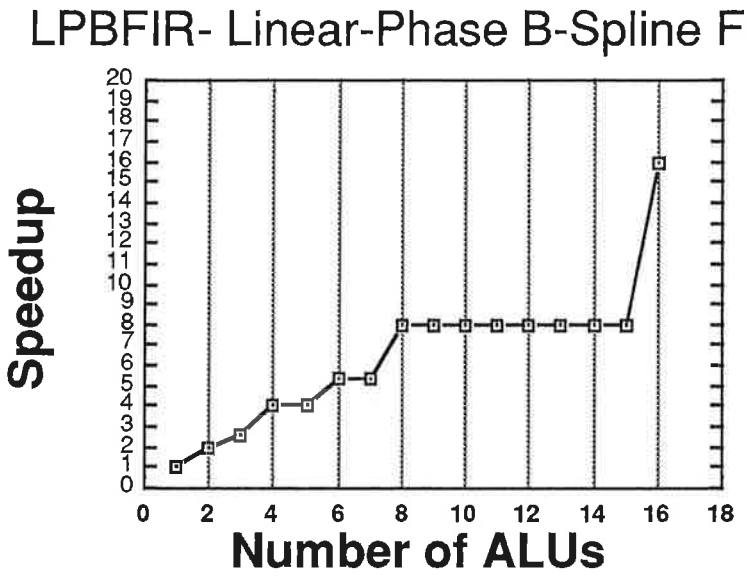


Figure 7.18: Linear-Phase B-Spline Filter Experiment

```
(PROC-BEGIN lpbfir
(LABEL lup)
    (IADD   $x0  $In-port  $a10)
    (IADD   $x1  $a1  $a9)
    (IADD   $x2  $a2  $a8)
    (IADD   $x3  $a3  $a7)
    (IADD   $x4  $a4  $a6)
    (IMUL   $y0  $m0  $x0)
    (IMUL   $y1  $m1  $x1)
    (IMUL   $y2  $m2  $x2)
    (IMUL   $y3  $m3  $x3)
    (IMUL   $y4  $m4  $x4)
    (IMUL   $y5  $m5  $a5)
    (IADD   $z1  $y0  $y1)
    (IADD   $z2  $z1  $y2)
    (IADD   $z3  $z2  $y3)
    (IADD   $z4  $z3  $y4)
    (IADD   $z5  $z4  $y5)
    (GOTO (label lup))
 (PROC-END       lpbfir))
```

Figure 7.19: Linear-Phase B-Spline Filter Algorithm

```
10124fd4:
    (iadd $x0[ 10 ]  $in_port[9 11 12 ] $a10[5 7 11 ] )[18]
    (iadd $x1[ 3 ]  $a1[12 ]  $a9[6 9 ]  )[19]
    (iadd $x2[ 1 ]  $a2[2 5 10 ]  $a8[3 7 ]  )[20]
    (iadd $x3[ 2 ]  $a3[4 9 ]  $a7[8 ]  )[21]
    (iadd $x4[ 5 ]  $a4[2 11 ]  $a6[1 10 ]  )[22]
    (imul $y5[ 4 ]  $m5[10 12 ]  $a5[1 2 ]  )[23]
    (goto 10126bd0)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17  }
10126bd0:
    (imul $y0[ 7 ]  $m0[6 ]  $x0[10 ]  )[24]
    (imul $y1[ 11 ]  $m1[8 ]  $x1[3 ]  )[25]
    (imul $y2[ 2 ]  $m2[9 ]  $x2[1 ]  )[26]
    (imul $y3[ 3 ]  $m3[4 ]  $x3[2 ]  )[27]
    (imul $y4[ 6 ]  $m4[7 ]  $x4[5 ]  )[28]
    (iadd $x0[ 10 ]  $in_port[9 11 12 ]  $a10[5 7 11 ] )[29]
    (goto 10127180)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 18 19 20 21 22
   23  }
10127180:
    (iadd $z1[ 4 ]  $y0[7 ]  $y1[11 ]  )[30]
    (iadd $x1[ 3 ]  $a1[12 ]  $a9[6 9 ]  )[31]
    (iadd $x2[ 1 ]  $a2[2 5 10 ]  $a8[3 7 ]  )[32]
    (iadd $x3[ 2 ]  $a3[4 9 ]  $a7[8 ]  )[33]
    (iadd $x4[ 5 ]  $a4[2 11 ]  $a6[1 10 ]  )[34]
    (imul $y0[ 7 ]  $m0[6 ]  $x0[10 ]  )[35]
    (goto 10126ea8)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 19 20 21 22 23
   24 25 26 27 28 29  }
10126ea8:
    (iadd $z2[ 1 ]  $z1[4 ]  $y2[2 ]  )[36]
    (iadd $x0[ 10 ]  $in_port[9 11 12 ] $a10[5 7 11 ])[37]
    (imul $y1[ 11 ]  $m1[8 ]  $x1[3 ]  )[38]
    (iadd $x1[ 2 ]  $a1[12 ]  $a9[6 9 ]  )[39]
    (imul $y2[ 3 ]  $m2[9 ]  $x2[1 ]  )[40]
    (iadd $x2[ 5 ]  $a2[2 5 10 ]  $a8[3 7 ]  )[41]
    (goto 10127cf0)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 23 25 26 27 28
   29 30 31 32 33 34 35  }
10127cf0:
    (iadd $z3[ 4 ]  $z2[1 ]  $y3[3 ]  )[42]
    (imul $y3[ 3 ]  $m3[4 ]  $x3[2 ]  )[43]
    (iadd $x3[ 2 ]  $a3[4 9 ]  $a7[8 ]  )[44]
    (iadd $z1[ 1 ]  $y0[7 ]  $y1[11 ]  )[45]
    (imul $y0[ 7 ]  $m0[6 ]  $x0[10 ]  )[46]
    (iadd $x0[ 10 ]  $in_port[9 11 12 ] $a10[5 7 11 ])[47]
    (goto 1012668c)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 23 27 28 30 33
   34 35 36 37 38 39 40 41  }
1012668c:  *** STEADY-STATE LOOP BEGINS **
    (iadd $z4[ 8 ]  $z3[4 ]  $y4[6 ]  )[48]
    (imul $y1[ 11 ]  $m1[8 ]  $x1[2 ]  )[49]
    (iadd $x1[ 2 ]  $a1[12 ]  $a9[6 9 ]  )[50]
    (imul $y4[ 6 ]  $m4[7 ]  $x4[5 ]  )[51]
    (iadd $x4[ 5 ]  $a4[2 11 ]  $a6[1 10 ]  )[52]
    (iadd $z2[ 1 ]  $z1[1 ]  $y2[3 ]  )[53]
    (goto 10125e04)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 23 28 34 36 38
   39 40 41 42 43 44 45 46 47 48 49
   50 51 52 53 54 55 56 57 58 59 60
   61 62 63  }
10125e04:
    (iadd $z5[ ]  $z4[8 ]  $y5[4 ]  )[54]
    (imul $y2[ 3 ]  $m2[9 ]  $x2[5 ]  )[55]
    (iadd $x2[ 5 ]  $a2[2 5 10 ]  $a8[3 7 ]  )[56]
    (imul $y5[ 4 ]  $m5[10 12 ]  $a5[1 2 ]  )[57]
    (iadd $z1[ 1 ]  $y0[7 ]  $y1[11 ]  )[58]
    (imul $y0[ 7 ]  $m0[6 ]  $x0[10 ]  )[59]
    (goto 101260dc)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 23 40 41 42 43
   44 45 46 47 48 49 50 51 52 53 54
   55 56 57 58 59 60 61 62 63  }
101260dc:
    (iadd $z3[ 4 ]  $z2[1 ]  $y3[3 ]  )[60]
    (imul $y3[ 3 ]  $m3[4 ]  $x3[2 ]  )[61]
    (iadd $x3[ 2 ]  $a3[4 9 ]  $a7[8 ]  )[62]
    (iadd $x0[ 10 ] $in_port[9 11 12 ] $a10[5 7 11 ])[63]
    (goto 1012668c)
Reaching Defs:
 { 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 15 16 17 42 43 44 47 48
   49 50 51 52 53 54 55 56 57 58 59
   60 61 62 63  }
```

Figure 7.20: Example: Linear-Phase B-Spline Filter VLIW code

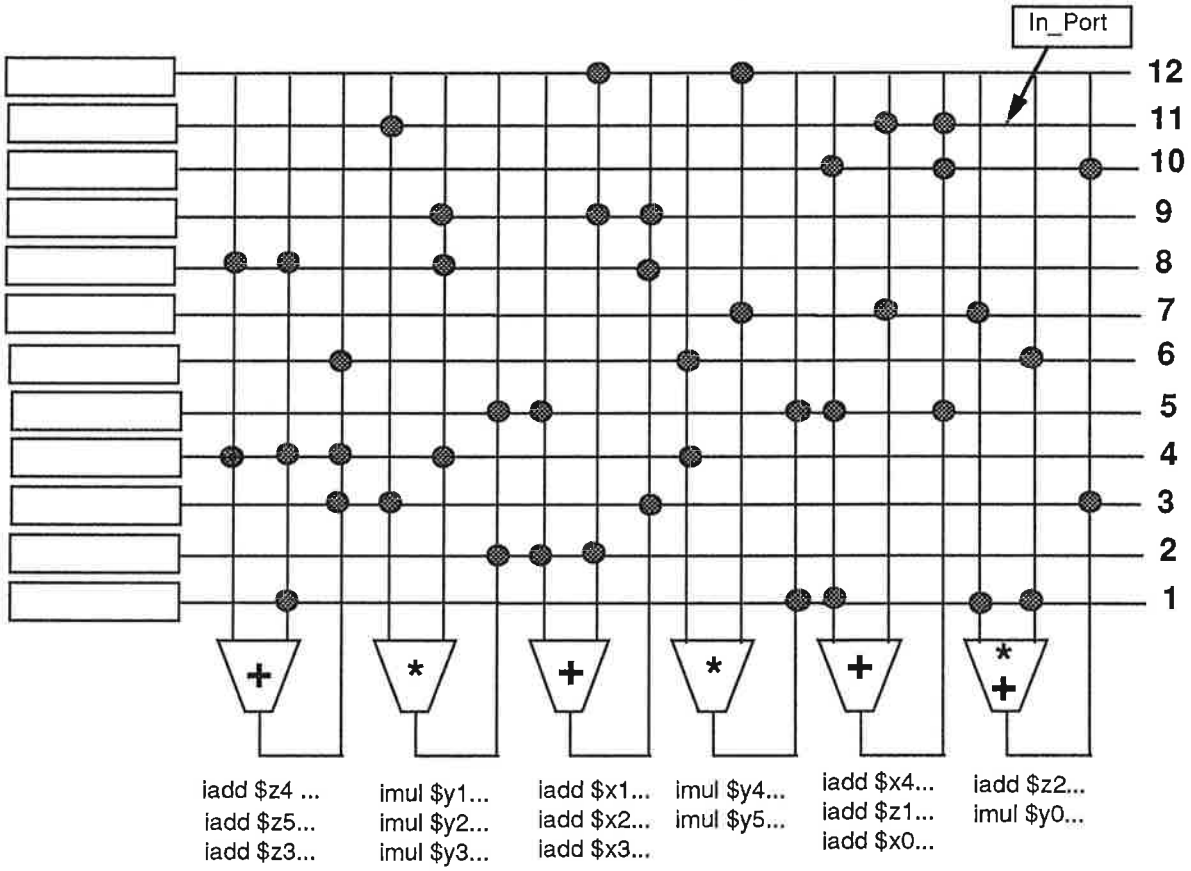


Figure 7.21: Example: Linear-Phase B-Spline Filter

Register File	Variables Allocated
1	z1, z2, a5
2	x1, a2, x3
3	y2, a8
4	z3, z5
5	x4, x2, a10
6	y4
7	m4, y0
8	z4, m1, a7
9	a9, m2, a3
10	a6, x0
11	in-port, y1, a4
12	a1, m5

Figure 7.22: Linear-Phase B-Spline Filter - Register File Allocation

while($\frac{norm(\tilde{r})}{norm(\tilde{y})} < \epsilon$) {	convergence ?
$\tilde{r} = C^{-1}.\tilde{r}$	BackSub(); /* perform Back Substitution */
$\theta \doteq (\tilde{r})^t.\tilde{r}$	theta = ThetComp(); /* get theta*/
$\tilde{r} = C^T.\tilde{r}$	SubBack(); /* second back substitution */
$\tilde{p} = \tilde{r} + \beta.\tilde{p}$	NewP(beta); /* get the new P vector */
$\delta = \tilde{p}.s.\tilde{p}^t$	delta = DeltaComp(); /* get delta*/
$\alpha = \frac{\gamma}{\delta}$	alpha = gamma/delta; /* compute alpha */
$\tilde{x} = \tilde{x} + \alpha.\tilde{p}$	NewX(alpha); /* get the new Xvector */
$\tilde{r} = \tilde{y} = S.\tilde{x}^t$	RES(); /* compute Residual: new error vector */
}	

Figure 7.23: Finite Element Matrix Solver Computation

`BackSub` is one of the most complex procedures in the Finite-Element code. It solves the equation $R_{(t+1)} = C^{-1} * R_{(t)}$, where R is a dense column vector of size N and C is an $N \times N$ lower triangular sparse matrix. Every diagonal element of C is non-zero. Instead of computing the inverse matrix, the procedure solves the system $C * R_{(t+1)} = R_{(t)}$ where $R_{(t)}$ is unknown. Because C is lower triangular, C[N,0] is the only element in row N. Therefore, $R_{(t)}[N] = C[0,N] * R_{(t+1)}[N]$, thus

$$R_{(t+1)}[N] = R_{(t)}[N] \div C[0,N] \tag{7.1}$$

The value of $R_{(t+1)}[N-1]$ is computed similarly. Because row N-1 has at most two elements and thus

$$R_t[N-1] = C[0,N-1] * R_{(t+1)}[N] + C[1,N-1] * R_{(t+1)}[N-1] \tag{7.2}$$

. The value of $R_{(t+1)}[N]$ is given by 7.1, and the equation 7.2 is solved for $R_{(t+1)}[N-1]$. The remaining values of $R_{(t+1)}$ are computed similarly.

The procedure starts at the top left of the matrix (row 0, column 0), using the formula for $k = 0$ to N:

$$R_{(t+1)}[k] = \frac{R_{(t)}[k] - \sum_{i=0}^{k-1} C[k,i] R_{(t+1)}[i]}{C_{(k,k)}}$$

The inner loop of the procedure implements the dot product, and the outer loop applies the formula for each row. The source code for the BackSubstitution routine is presented in Appendix A.

ALUs	Cond. Jumps	Speedup small data set	Speedup large data set
1	1	1.00	1.00
2	1	1.13	1.27
3	2	1.57	1.83
4	1	1.98	2.27
5	1	2.33	2.72
6	2	2.46	3.21
7	2	2.58	3.30
8	2	2.78	3.48
12	2	3.08	4.32
14	2	3.08	4.32
16	2	3.08	4.32
17	2	3.08	5.63

Figure 7.24: Back Substitution experiment - simulation results

Figure 7.24 illustrates Specification Optimization of the FEM BackSubstitution routine. The experiments are run with a small data set, for ease of simulation, and a larger and more realistic data set. Both matrix data sets have approximately the same distribution of sparseness typically found in electromagnetic FEM problems. With 17 ALUs and more, the maximum rate of one iteration per cycle is achieved. However, this rate is rarely exercised in simulation for this example, given the short vector characteristics of this FEM problem. Due to the short vector characteristics, the maximum speedup achieved in simulation is 5.63. From 12 to 16 ALUs, the inner loop executes at the maximum rate of two cycles per iteration. The maximum speedup achievable for this program is 13, because there are 12 operations and one conditional jump in the inner loop.

One of these operations, `(ICONSTANT $8 19940)`, loads a constant value into a register and is removed by optimization. The 1 cycle/iteration steady state instruction for this loop requires 17 ALUs, because of extra COPY operations introduced during pipelining. The steady state pipelining of 1 cycle/iteration is achieved, for this program that uses sparse pointer-based data structures, by using a compiler assertion to disable inter-iteration data dependency.

The VLIW program for the case of 1 cycle/iteration is presented in Appendix A. Figure 7.25 presents the largest instruction for the inner loop. That instruc-

tion, L1014c024, performs concurrently five COPY operations, two floating-point operations (`fadd` and `fmul`), four integer operations, and five loads from memory as well as one conditional jump. This example assumes a memory system similar to the White Dwarf's, which is capable of providing concurrently all elements of a record in the sparse matrix. As mentioned before, the COPY operations can be avoided by using loop unrolling and the algorithm of Section 3.3.2, thus reducing the instruction cost. The hand-coded White Dwarf BackSubstitution routine also achieves a maximum rate of one iteration per cycle. For comparison, the hand-generated microcode inner loop is composed of four instructions, because the loop is unrolled four times to avoid copy operations, and is optimized for the case of short vectors. The widest microinstruction in the White Dwarf routine performs concurrently two integer additions, one floating-point addition and one floating-point multiplication, a memory access to read the column, row and matrix entry elements of the sparse matrix, a memory read from the dense vector R, a test for loop termination, a number of register-to-register transfers and one conditional jump. This hand-generated microcode is comparable to the number of resources required by the compiler-generated VLIW code. The CPU time for specification optimization of this example is 1.5 CPU seconds on a DECStation 5000. With the added execution of register files allocation and graph coloring routines the total runtime is 2.4 seconds.

7.8 RISC Instruction Set Processor Example

This section illustrates the inter-iteration effects of the specification optimization techniques. In this example the application code given as input to the ASPD system is a program to interpret an instruction set. The resulting ASP is the design of an instruction set processor. The use of *run-time disambiguation* during specification optimization allows the automated generation of a high-performance pipelined processor with instruction prefetching. Furthermore, the resulting design is capable of achieving throughput of one instruction per cycle when data dependencies allow.

```
1014c024:
    (fadd $f6 $f6 $f4)
    (fmul $f4 $f8 $f10)
    (fassign $f8 $f8')
    (fassign $f8' $f8'')
    (fassign $f8'' $f8''')
    (fvload $f10  0 $2)
    (if $cc0 L17
      (iassign $4 $4')
      (iassign $4' $4'')
      (iadd $2 $6 $2')
      (ine $cc0 $8 $9)
      (iash $2' $8'  2)
      (ivload $8  4 $4'')
      (ivload $9  0 $3)
      (ivload $8'  0 $3)
      (fvload $f8'''  0 $4'')
      (iadd $3 $3  12)
      (iadd $4'' $4''  12)
      (goto 1014c024)
    ELSE
      (goto 1014c844))
```

Figure 7.25: BackSubstitution Example - Widest VLIW instruction

Run-time disambiguation is applicable if memory-reference disambiguation mechanism[63] does not have enough information about memory accesses and hence parallelization is prevented. The solution is to generate distinct versions of the code, one assuming that memory references do not interfere, and another assuming data dependency. These two versions of code are scheduled for concurrent execution. A check is added to determine at run-time which version is correct. Run-time disambiguation is able to achieve significant speedup, and alleviates memory access bottlenecks. Furthermore, the use of run-time disambiguation allows pipelining in cases where conservative assumption of data dependency would entail serial execution. This is the key to achieving pipelined implementations of instruction set processors with pre-fetching. Furthermore, in situations where the compiler does not have enough information to decide on memory-reference disambiguation it is frequently the case that no conflict occurs, e.g. pointer-based data structures[32]. A drawback of run-time disambiguation is the potentially large increase in code size. However, run-time disambiguation allows pipelining to occur, and thus is able to extract parallelism, in situations when more conventional approaches fail. Even though the implementation of instruction-set processors is not the target application area of ASPD, this example is included to illustrates the breadth of application of the ASPD method

and the application of run-time disambiguation. Here, the pipelined code for the RISC processor is initially generated by the VLIW compiler, manually enhanced with the addition of run-time disambiguation.

Figure 7.26 is a behavioral description of the simple RISC processor which, for the purpose of illustration, has only two kinds of instructions: ALU operation and branch. After percolation scheduling is applied, the operations in the inner loop, corresponding to RISC instruction fetch and execute cycle, are divided into four microinstructions corresponding to the phases of: instruction fetch, decode, execution, and write back results. This is illustrated in Figure 7.27.

By applying inter-iteration optimization, the result is a processor capable of executing a new RISC instruction every two cycles (assuming fast 1-cycle memory). Enhanced Software Pipelining is only able to start a new iteration, i.e. a new RISC instruction, concurrently with the third phase of instruction execution. This is because that there is a potential dependency between the microinstruction for execution, and the microinstruction for instruction fetch, if a branch instruction is being executed by the RISC processor.

By employing extended pipeline scheduling with run-time disambiguation, a new iteration is started conditionally at every cycle, because there are paths through the loop where the new value of the program counter is known. These paths correspond to execution of RISC instructions other than branch, without data dependency between consecutive instructions. In Figure 7.28, Instructions 1 to 4 represent the RISC instruction execution pipeline being filled on startup, and Instruction 5 represents the steady-state of concurrently executing 4 RISC instructions without data dependency. In Instruction 5, the results of RISC instruction i are being stored into the register file (`{RF[D]=DST}`); RISC instruction *(i+1)* is being executed (`(DST = ALU(OP,SRC1,SRC2))`; the arguments of RISC instruction *(i+2)* are being fetched from the register file (`<SRC1=RF[S1]; SRC2=RF[S2]>`); RISC instruction *(i+2)* is being decoded (`{ccd=OP'>1>}`); and RISC instruction *(i+3)* is being fetched from memory (`{S1=Mem[PC].s1, ...}`). With this technique, a pipelined RISC processor is generated naturally from

```
Simple RISC processor:
MEM[0..memsize]: memory
IR: Instruction Register,
PC: Program Counter
RF[0..31]: Register File>


LOOP  IR=MEM[PC];PC=PC+1;
      IF IR.opcode.class=ALU_OP /* ALU op ? */
      THEN  /* perform ALU operation */
        RF[IR.dest]=ALU(IR.opcode,RF[IR.src1],RF[IR.src2])
      ELSE  /* JUMP instruction */
        PC = M[PC];
ENDLOOP
```

Figure 7.26: Simple RISC Processor Specification

```
L1: IR=MEM[PC];PC=PC+1; /* Fetch */
L2: S1=RF[IR.src1]; S1=RF[IR.src2]; DECODE[IR.opcode(L3, L4)
/*Conditional jump to L3 or L4 */
L3: RF[IR.dst] = IR.opcode(S1, S2); GOTO L1 /* execute ALU op */
L4: PC=MEM[PC]; GOTO L1
```

Figure 7.27: Intra-iteration Optimized RISC Specification: 4 cycles/instruction.

the the original behavioral description of a RISC processor. The resultant design is capable of executing one RISC instruction per cycle when dependencies allow. Furthermore, the code for handling pipeline hazards is generated automatically. Additionally, this technique presents the interesting possibility of generating processors capable of executing more than one instruction per cycle (superscalar). This is done by unrolling the loop corresponding to the behavioral description of the RISC instruction fetch and execute cycle prior to input to the ASPD system. Care is required during the application of run-time disambiguation, to avoid large increases in code size and schedules that are wasteful of resources.

7.9 FFT Processor Example

This section illustrates an ASP for the FFT algorithm presented in Chapter 6. The VLIW code, with memory bank control, is illustrated in Figure 6.7. This FFT example uses memory organization for concurrent access and enhanced memory switches for memory bank control, as described in Section 6.6 of Chapter

```
(1) S1=Mem[PC].s1; S2=Mem[PC].s2; D =Mem[PC].d;
    OP=Mem[PC].op; PC=PC+1;GOTO(2)
(2) ccd=(OP>0); SRC1 = RF[S1];  SRC2 = RF[S2];
    S1=Mem[PC].s1; S2=Mem[PC].s2; D'=Mem[PC].d;
    OP'=Mem[PC].op; PC'=PC; PC=PC+1;    GOTO (3)
(3) OP=OP'; D=D'; PC''=PC' ; PC'=PC;
    if ccd /* ALU op */
DST  = ALU(OP,SRC1,SRC2);
ccd=(OP'>1); SRC1 = RF[S1]; SRC2 = RF[S2];
cc1=S1==D ; cc2=S2==D;
S1=Mem[PC].s1; S2=Mem[PC].s2; D''=Mem[PC].d;
OP=OP' ;  OP'=Mem[PC].op;
PC=PC+1; GOTO (4)
    else{PC=Mem[PC].Immed ;  GOTO (1) }
(4)if !(cc1 v cc2)
   RF[D]=DST ;    OP=OP'; D=D';
   if ccd /* ALU op */
DST  = ALU(OP,SRC1,SRC2);
ccd=(OP'>1); SRC1 = RF[S1];SRC2 = RF[S2];
cc1=S1==D ; cc2=S2==D;
cc3=S1==D' ;  cc4=S4==D''
S1=Mem[PC].s1 ; S2=Mem[PC].s2;
OP = OP' ; OP'=Mem[PC].op;
D=D'; D'=D'';  D''=Mem[PC].d;
PC''=PC'; PC'=PC; PC = PC + 1; GOTO (5)
   else {PC=Mem[PC].Immed ;  GOTO (1)}
else /* conflict */
    PC = PC'';    GOTO (1)
(5)if !(cc1 v cc2 v cc3 v cc4)
   RF[D]=DST
   if ccd /* ALU op */
DST  = ALU(OP,SRC1,SRC2);
ccd=(OP'>1); SRC1 = RF[S1];SRC2 = RF[S2];
cc1=S1==D ; cc2=S2==D;
cc3=S1==D' ;  cc4=S4==D''
S1=Mem[PC].s1; S2=Mem[PC].s2; D''=Mem[PC].d;
OP=OP'; OP'=Mem[PC].op;
PC''=PC'; PC'=PC; PC=PC+1; GOTO (5)
   else {PC=Mem[PC].Immed ;  GOTO (1)}
else /* conflict */
    PC = PC'';    GOTO (1)
```

Figure 7.28: Optimized RISC Specification: 1 cycle/RISC instr

6. This code takes 5 cycles per iteration, whereas a RISC processor would require 22 cycles, therefore achieving a speedup of about 4.4 (bank control is not needed for a serial RISC execution). This is the same speedup obtained by the VLIW architectural template, i.e. direct implementation of the optimized specification without implementation optimization. However, a canonical VLIW processor capable of executing that same code would require 5 ALUs, a 15-ported global register file and a 2-ported global memory system. On the other hand, the design in Figure 7.29 is built out of single-ported register files and memories. Furthermore, the interconnection between register files and functional units is much more sparse than the full connectivity in the canonical VLIW processor. It is expected that cycle time for the FFT ASP design in Figure 7.29 would be shorter than the cycle time of a canonical VLIW processor capable of executing the same code. Therefore, the FFT ASP achieves high performance as well as efficient utilization of the hardware resources.

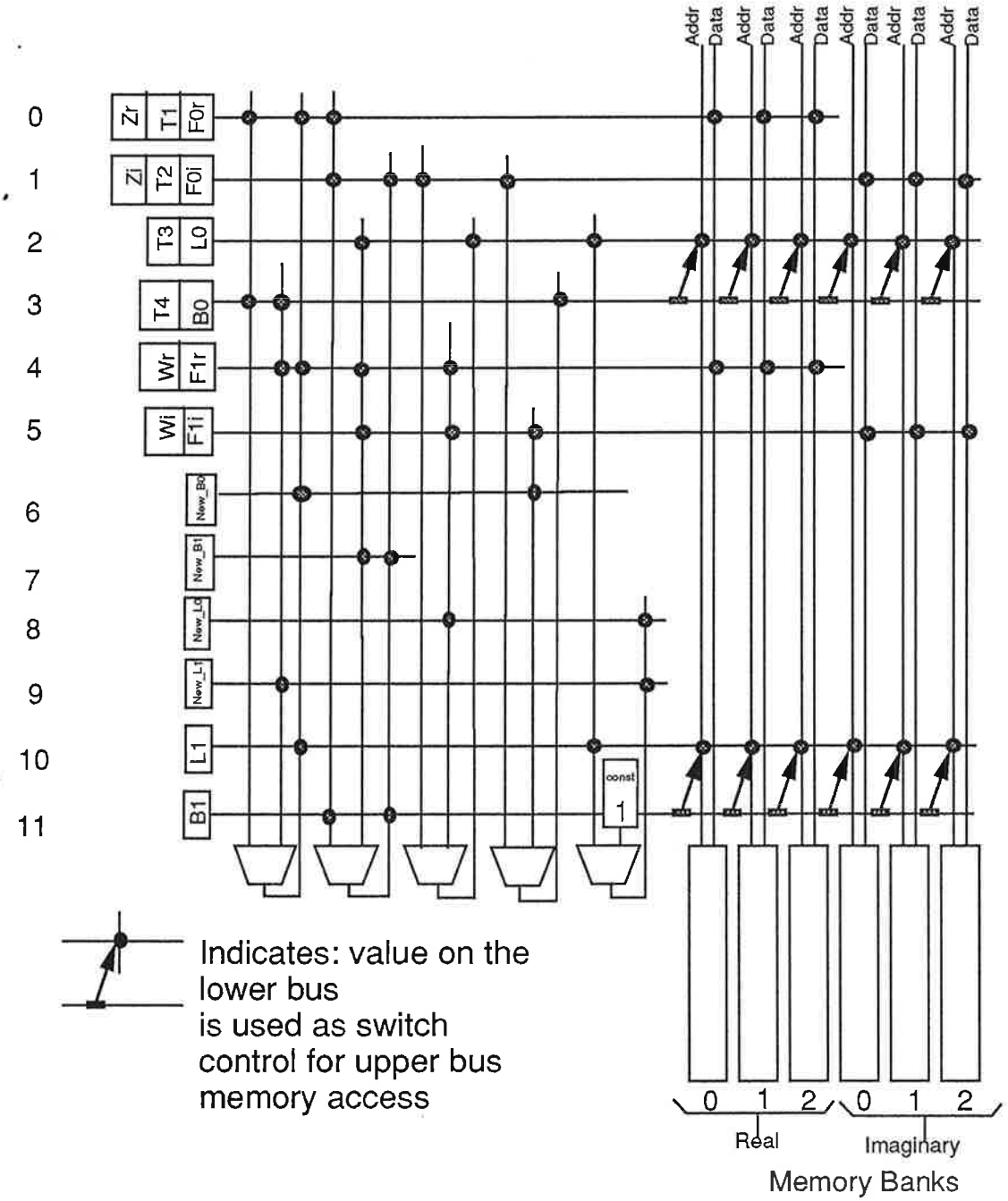


Figure 7.29: Example: FFT Inner Loop Code With Memory Bank Control

Chapter 8

Summary, Conclusions and Extensions

This chapter presents a summary of the thesis and describes key contributions along with suggestions for further research.

8.1 Thesis Summary

The ASPD architecture synthesis method divides the task of designing an special-purpose processor architecture into Specification Optimization and Implementation Optimization phases. Algorithms for the Specification Optimization and Implementation Optimization phases and their implementation are presented in this thesis. The use of advanced compilation techniques is a key to achieving high-performance processors. Instruction scheduling is a prime determinant of performance because it determines the number of clock cycles required for the execution of the target application. Once a schedule is fixed, remaining opportunities for performance enhancement are found in the hardware allocation phase by reducing the clock cycle. Scheduling limited to basic blocks is not capable of achieving high-performance, because there is not much opportunity for parallelism in basic blocks. Percolation Scheduling[62, 34] and Enhanced Pipeline Scheduling[33] are characterized as general and scalable compilation techniques that allow broad application and usefulness, and are employed in the ASPD method. High-performance schedules are obtained via compilation and code par-

allelization for an unconstrained architectural template. A VLIW architectural template is chosen as the scalable target architecture for Specification Optimization. This process of repeated compilation with gradual imposition of resource constraints leads to the final optimized specification.

Further performance enhancement is obtained by tailoring the implementation organization to the execution requirements of a specific code schedule. A scalable implementation template featuring multiple busses and distributed single-port register files is presented. Allocation of data into multiple single-ported register files is performed by using efficient graph-coloring algorithms. The identification of special graph characteristics allow the use of efficient polynomial-time coloring algorithms. Data allocation is performed to allow concurrent access to register file data as required by the optimized program. For some classes of programs, the optimal (minimum) number of register files is achieved. Implementation Optimization results in an sparsely interconnected structure which is capable of executing the VLIW parallelized code at a lower hardware cost. Example applications of the ASPD method illustrate the high performance obtained with the use of advanced compilation techniques at very reasonable computational costs.

8.2 Key Contributions

The key contributions of this thesis are divided into the following categories:

- **A New Method of Architecture Synthesis:**

  The ASPD method of architecture synthesis of high-performance application-specific processors brings together the two research areas of compilation for fine-grain parallelism and digital system synthesis. A clean framework for synthesis is established with the division of the optimization task into behavioral and structural realms.

- **New and Improved Compilation Techniques:**

A new algorithm to detect Induction Variables is described in Chapter 3. It is applicable beyond the VLIW realm, as recent superscalar and super-pipelined uniprocessors require powerful methods to exploit architectural parallelism.

A new memory allocation algorithm to allow concurrent access to memory variables is described in Chapter 6. The algorithm is useful in compilers for VLIW architectures with multiple memory banks. Furthermore, this algorithm is extensible to enhance the effectiveness of processors equipped with direct-mapped caches.

An efficient graph coloring algorithm for register file allocation is described in Chapter 5. The algorithm may be extended to allocate data into multiple register files in VLIW architectures which have register file access constraints such as [46].

- **Synthesis Tools Implementation:**

  The computation of the unifiable-ops attribute using bit sets is previously unimplemented. An implementation which enhances algorithmic performance is described in Chapter 3 along with extensions to handle register renaming.

  The Enhanced Software Pipelining algorithm has been extended to use the unifiable-ops attribute, and the implementation is described in Chapters 3.

  An advanced retargetable compiler for VLIW architectures has been implemented. This is the first public-accessible VLIW compiler.

- **Application of Architecture Synthesis Tools:**

  The architecture synthesis method and tools are applied to a number of examples. These examples confirm the feasibility of the architecture synthesis method and show that performance speed-up in the range of 2.6 to 7.7 over contemporary general-purpose RISC processors can be achieved.

8.3 Future Work

In [47] standard compiler optimizations[2] are also integrated in the percolation scheduling framework. The basic idea is to apply the standard optimizations as operations are being moved by percolation scheduling. For example, the actual opcode and arguments of an operation may be changed to reflect the optimization during the motion of the operation from an instruction to another. This approach is desirable since some of the standard high-level optimizations may actually reduce the eventual parallelism in the final object code. Therefore, it is beneficial to delay the application of optimization until the parallelization phase of compilation. Furthermore, new opportunities for optimization are encountered during the parallelization process which did not exist in the serial version of the program.

In the present implementation, pipelining is only applied to inner loops. The enhanced percolation scheduling technique is applicable to outer loops by considering (pipelined) inner loops as a single indivisible unit during parallelization[31]. However, this method does not benefit nested loops with short dependency vectors. Loop Quantization[61] is capable of extracting more parallelism by unrolling and compacting both the inner and outer loops. A possible extension is to extend the enhanced software pipelining algorithm to move operations across outer loop back-edge. This achieves loop-unrolling effects and thus emulates the Loop Quantization algorithm. This can be developed in an integrated framework for the parallelization of nested loops.

The specification optimization examples of Chapter 7 assume single-cycle latency execution of operations. Because data-dependency is a key controlling factor in percolation scheduling, it frequently finds schedules in which operations in consecutive instructions are data dependent. This is specifically important for operations in the critical dependency path. Single-cycle operation latency is used in the IBM VLIW prototype[31]. In the White Dwarf project, the small difference in operator latencies for the integer and floating-point units was such that the use of multiple cycle floating-point units would either unduly complicate the clocking

mechanisms or unnecessarily increase floating point operator latency. Because floating-point operations composed most of the critical dependency path, a single-cycle operator latency was adopted. The integer ALU operations take also a single pipeline stage in the recently announced MIPS R4000 RISC processor. A possible ASPD extension for architecture realization with multicycle pipelined operators is the notion of degree of concurrency[13], which combines multiplicity of functional units and pipeline depths. Specification optimization is performed with single-cycle latency operators and alternative realizations are searched in actual implementation with an equivalent degree of concurrency.

An alternative implementation of software pipelining that uses perfect pipelining[6] is planned. Perfect pipelining is guaranteed to find an optimal pipelined schedule for a special class of loops that have no conditional jumps in the loop body. The schedule is optimal in the sense that no transformation of the loop based on the data dependencies can yield a shorter running time for that loop.

The implementation template does not constrain the layout of the realization technology. Some of the algorithms in this thesis may be used to help hardware implementation. For example, a printed-circuit board ASP realization may use crossbar chips to cover the multiple-bus implementation architecture template. The algorithms for operation allocation can be extended to perform this task. Furthermore, a PLA-folding algorithm[40] may be used to allocate the register files and operators of the implementation architecture in rows and columns, to maximize sharing of rows and columns. The PLA-folded layout is provided as an improved input to the placement and routing tools to help achieve a more compact physical layout.

Appendix A

FEM Program Source Code

A.1 BackSubstitution Routine

```
#define N 34
#define MSIZE 11
#define COL item->col
#define ROW item->row
#define SVAL item->sval

typedef struct {
  float  sval;
  int    row;
  int    col;
}  matentry;

matentry  s[N] = {
  {    2.0,   0,   0}, {    3.0,   1,   0}, {    2.0,   1,   1},
  {   -3.0,   2,   0}, {    2.0,   2,   2}, {    2.0,   3,   1},
  {    3.0,   3,   3}, {    2.0,   4,   2}, {    3.0,   4,   3},
  {    2.0,   4,   4}, {   -3.0,   5,   0}, {    2.0,   5,   3},
  {    2.0,   5,   5}, {    2.0,   6,   2}, {   -3.0,   6,   3},
  {   -3.0,   6,   4}, {    3.0,   6,   6}, {    2.0,   7,   5},
  {    2.0,   7,   6}, {   -3.0,   7,   7}, {    2.0,   8,   0},
```

```
  {   3.0,   8,   1}, {   3.0,   8,   2}, {   2.0,   8,   7},
  {   3.0,   8,   8}, {   2.0,   9,   0}, {  -3.0,   9,   1},
  {   2.0,   9,   7}, {   3.0,   9,   9}, {  -3.0,  10,   0},
  {   2.0,  10,   4}, {  -3.0,  10,   5}, {   2.0,  10,   9},
  {   3.0,  10,  10} };

matentry *rowstart[MSIZE] = {  s+0,  s+1,  s+3,  s+5,  s+7,
                               s+10, s+13, s+17, s+20, s+25,
                               s+29  };

float z[MSIZE] = { 12.0,  4.0,  9.0, 16.0, 45.0,
                   20.0, 44.0, 24.0, 10.0, 90.0,
     50.0  };

float y[MSIZE] = {  0.0,  0.0,  0.0,  0.0,  0.0,
                    0.0,  0.0,  0.0,  0.0,  0.0,
      0.0  };

main()
{
  int      k;
  float    temp;
  matentry *item;


  for (k = 0; k < MSIZE; k++) {
    item = rowstart[k];
    temp = 0.0;
    while (ROW != COL) {
      temp += SVAL * y[COL];
```

```
        item++;
      }
      y[COL] = (z[COL] - temp) / SVAL;
    }
  }
```

A.2 VLIW code for BackSubstitution Routine

```
1014683c:
    (iconstant $0  0)
    (iconstant $2  19140)
    (iconstant $6  20740)
    (iconstant $8''  19940)
    (goto 1014e918)

1014e918:
    (iassign $5 $2)
    (iadd $7 $2  796)
    (ivload $4  0 $2)
    (fvload $f6  21540 $0)
    (iadd $5' $2  4)
    (goto 1014ba20)

10147aa0:
    (ivload $4  0 $5)
    (fvload $f6  21540 $0)
    (iconstant $8''  19940)
    (iadd $5' $5  4)
    (goto 1014ba20)

1014ba20:
    (ivload $8  4 $4)
    (ivload $9  8 $4)
    (ivload $8'  8 $4)
    (iadd $3 $4  8)
    (fvload $f8  0 $4)
    (iadd $4' $4  12)
    (fvload $f8''  0 $4)
    (ile $cc0' $5' $7)
    (goto 1014d930)

1014d930:
    (ieq $cc0 $8 $9)
    (iash $2 $8'  2)
    (ivload $8  0 $3)
    (iadd $3 $3  12)
    (ivload $8'  4 $4')
    (fvload $f8'  0 $4')
    (iadd $4'' $4'  12)
    (iassign $4 $4')
    (if $cc0' L18
      (goto 1014ebc8)
    ELSE
      (goto 10147548))

10147548:
    (if $cc0 L19
      (iadd $3 $6 $2)
      (iadd $2 $8'' $2)
      (fassign $f8' $f8'')
      (goto 101482f8)
    ELSE
      (iassign $4' $4'')
      (iash $2 $8  2)
      (ivload $9  0 $3)
      (ivload $8'  0 $3)
      (iadd $3 $3  12)
      (ivload $8''  4 $4'')
      (fvload $f8''  0 $4'')
      (iadd $4'' $4''  12)
      (iassign $8 $8')
      (goto 1014bcf8))

1014ebc8:
    (if $cc0 L19
      (iadd $3 $6 $2)
      (iadd $2 $8'' $2)
      (fassign $f8' $f8'')
      (iassign $5 $5')
      (goto 1014ee70)
    ELSE
      (iassign $4' $4'')
      (iash $2 $8  2)
      (ivload $9  0 $3)
      (ivload $8'  0 $3)
      (iadd $3 $3  12)
      (ivload $8''  4 $4'')
      (fvload $f8''  0 $4'')
      (iadd $4'' $4''  12)
      (iassign $8 $8')
      (goto 1014bcf8))

101482f8:
    (fvload $f8  0 $2)
    (goto 1014b220)

1014ee70:
    (fvload $f8  0 $2)
    (goto 101485c0)

1014b220:
    (fsub $f4 $f8 $f6)
    (fassign $f8 $f8')
    (goto 1014dbd8)

101485c0:
    (fsub $f4 $f8 $f6)
    (fassign $f8 $f8')
    (goto 1014f670)

1014dbd8:
    (fdiv $f4 $f4 $f8)
    (goto 1014d03c)

1014f670:
    (fdiv $f4 $f4 $f8)
    (goto 10148040)

1014d03c:
    (fvstore  0 $3 $f4)
    (goto 1014fd14)

10148040:
    (fvstore  0 $3 $f4)
    (goto 10147aa0)

1014bcf8:
    (iadd $2 $6 $2)
    (ine $cc0 $8 $9)
    (iash $2' $8'  2)
    (iassign $8 $8'')
    (ivload $9  0 $3)
    (ivload $8'  0 $3)
    (iadd $3 $3  12)
    (ivload $8''  4 $4'')
    (fvload $f8'''  0 $4'')
    (iadd $4''' $4''  12)
    (goto 101488b0)

101488b0:
    (fvload $f10  0 $2)
    (if $cc0 L17
```

```
      (iassign $4 $4')
      (iassign $4' $4'')
      (iadd $2 $6 $2')
      (ine $cc0 $8 $9)
      (iash $2' $8' 2)
      (iassign $8 $8'')
      (ivload $9 0 $3)
      (ivload $8' 0 $3)
      (iadd $3 $3 12)
      (iassign $4'' $4''')
      (ivload $8'' 4 $4''')
      (fvload $f8'''' 0 $4''')
      (iadd $4''' $4''' 12)
      (goto 1014e3c0)
    ELSE
      (goto 1014d30c))

1014e3c0:
    (fmul $f4 $f8 $f10)
    (fassign $f8 $f8')
    (fassign $f8' $f8'')
    (fassign $f8'' $f8''')
    (fvload $f10 0 $2)
    (if $cc0 L17
      (iassign $4 $4')
      (iassign $4' $4'')
      (iadd $2 $6 $2')
      (ine $cc0 $8 $9)
      (iash $2' $8' 2)
      (iassign $8 $8'')
      (ivload $9 0 $3)
      (ivload $8' 0 $3)
      (fassign $f8''' $f8'''')
      (iadd $3 $3 12)
      (iassign $4'' $4''')
      (goto 1014c024)
    ELSE
      (goto 1014c844))

1014c024:
    (fadd $f6 $f6 $f4)
    (fmul $f4 $f8 $f10)
    (fassign $f8 $f8')
    (fassign $f8' $f8'')
    (fassign $f8'' $f8''')
    (fvload $f10 0 $2)
    (if $cc0 L17
      (iassign $4 $4')
      (iassign $4' $4'')
      (iadd $2 $6 $2')
      (ine $cc0 $8 $9)
      (iash $2' $8' 2)
      (ivload $8 4 $4'')
      (ivload $9 0 $3)
      (ivload $8' 0 $3)
      (fvload $f8''' 0 $4'')
      (iadd $3 $3 12)
      (iadd $4'' $4'' 12)
      (goto 1014c024)
    ELSE
      (goto 1014c844))

1014c844:
    (fadd $f6 $f6 $f4)
    (goto 1014d30c)

1014d30c:
    (fmul $f4 $f8 $f10)
    (ivload $8 8 $4)
    (iconstant $8' 19940)
    (fvload $f8' 0 $4)
    (iadd $5 $5 4)
    (goto 1014e670)

1014e670:
    (fadd $f6 $f6 $f4)
    (iash $2 $8 2)
    (iassign $8 $8')
    (ile $cc0 $5 $7)
    (goto 1014caec)

1014caec:
    (iadd $3 $6 $2)
    (iadd $2 $8 $2)
    (if $cc0 L18
      (goto 1014de80)
    ELSE
      (goto 10147d78))

10147d78:
    (fvload $f8 0 $2)
    (goto 1014ace8)

1014de80:
    (fvload $f8 0 $2)
    (goto 1014f120)

1014ace8:
    (fsub $f4 $f8 $f6)
    (fassign $f8 $f8')
    (goto 1014b770)

1014f120:
    (fsub $f4 $f8 $f6)
    (fassign $f8 $f8')
    (goto 1014f3c8)

1014b770:
    (fdiv $f4 $f4 $f8)
    (goto 1014e130)

1014f3c8:
    (fdiv $f4 $f4 $f8)
    (goto 1014f940)

1014e130:
    (fvstore 0 $3 $f4)
    (goto 1014fd14)

1014f940:
    (fvstore 0 $3 $f4)
    (goto 10147aa0)

1014fd14:
    (igoto 101441f0)
```

Bibliography

[1] Landskov 80. Local microcode compaction techniques. *Computing Surveys*, Sept 1980.

[2] R. Sethi A. Aho and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.

[3] J. Pizzarro A. C. Parker and M. Milnar. MAHA: A program for datapath synthesis. *23rd Design Automation Conference*, pages 461–466, 1986.

[4] A. Aiken. *Compaction-Based Parallelization.* PhD thesis, Dept. of Computer Science, Cornell University, 1988.

[5] A. Aiken and A. Nicolau. Optimal loop parallelization. *Dept. of Computer Science - Cornell Tech. Report TR 88-905*, March 1988.

[6] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. *European Symposium on Programming, Springer-Verlag Lecture Notes in Computer Science no. 300*, 1988.

[7] Alexander Aiken and Alexandru Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, May 1988.

[8] Alexander Aiken and Alexandru Nicolau. Fine-grain parallelization and the wavefront method. In *2nd Workshop on Parallel Compilation, Urbana, Illinois*, 1989.

[9] O. Axelsson. *Solution of Linear Systems of Equations: Iterative Sparse Matrix Techniques.* Springer-Verlag, 1976.

[10] J. L. Baer. *Computer Systems Architecture.* Computer Press, 1980.

[11] U. Banerjee. Speedup of ordinary programs. *Tech. Report UIUCDS-R-79-989, U. of Illinois C.S. Dept.*, 1979.

[12] M. R. Barbacci. *Automated Exploration of the Design Space for Register Transfer Systems.* PhD thesis, Carnegie-Mellon University, Nov 1973.

[13] M. Breternitz and A. Nicolau. Tradeoffs between pipelining and multiple functional units in fine-grain parallelism exploitation. *ICS-90 International Conference on Supercomputing, Santa Clara CA*, April 1989.

[14] F.D. Brewer and D.D. Gajski. Knowledge-based control in micro-architecture design. *Proceedings of the 24th Design Automation Conference*, June 1987.

[15] I. Bucher. The computational speed of supercomputers. *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Aug 1983.

[16] P. Budnick and D. Kuck. The organization and use of parallel memories. *IEEE Trans. Comp.*, pages 1566–1569, December 1971.

[17] R. Camposano. A method for structural synthesis. *IBM T. J. Watson Research Report RC13081*, 1987.

[18] R. Camposano and Reinaldo A. Bergamaschi. Synthesis using path-based scheduling algorithms and exercises. *Proceedings of the 27th Design Automation Conference*, June 1990.

[19] G. J. Chaitin. Register allocation and spilling via graph coloring. *Proceedings of the SIGPLAN82 Symposium on Compiler Construction*, pages 201–207, June 1982.

[20] V. Chvatál and N. Sbihi. Recognizing claw-free perfect graphs. *Journal of Combinatorial Theory, Series B 44*, April 1988.

[21] C.J.Tseng, R.S. Wei, S.G. Rothweiler, M. M. Tong, and A. K. Bose. Bridge: A versatile behavioral synthesis system. *Proceedings of the 25th Design Automation Conference*, June 1988.

[22] R. J. Cloutier and D. E. Thomas. The combination of scheduling, allocation and mapping in a single algorithm. *Proceedings of the 27th Design Automation Conference*, June 1990.

[23] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM Trans. Comp.*, Aug 1982.

[24] Inc. Cydrome. CYDRA 5 departmental supercomputer. *Product Summary*, 1988.

[25] Ron Cytron and Jeanne Ferrante. What's in a name? or, the value of renaming for parallelism detection and storage allocation. *Research report RC12785. International Business Machines Corporation.*, 1987.

[26] B. Leasure D. J. Kuck, R. H. Kuhn and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. *Fourth International Computer Software and Applications Conference*, October 1980.

[27] D.E.Thomas, E.M.Dirkes, R.A.Walker, J.V.Rajan, J.A.Nestor, and R.L.Blackburn. The system architect's workbench. *Proceedings of the 25th Design Automation Conference*, June 1988.

[28] Advanced Micro Devices. *Hot new products catalog.* Advanced Micro Devides, Sunnyvale, California, 1984.

[29] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas. A design methodology and computer aids for digital VLSI systems. *Carnegie-Mellon University*, July 1981.

[30] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. *20th Annual Workshop on Microprogramming*, Dec 1987.

[31] K. Ebcioğlu. Some design ideas for a VLIW architecture for sequential-natured software. *Proceeding of the IFIP Working Conference on Parallel Processing*, April 1988.

[32] K. Ebcioğlu. *Personal Communication*, 1990.

[33] K. Ebcioğlu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *2nd Workshop on Parallel Compilation, Urbana, Illinois*, 1989.

[34] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *ICS-89, Greece*, 1989.

[35] J. R. Ellis. *Bulldog: A compiler for VLIW Architectures.* PhD thesis, Yale, 1985.

[36] R. P. Colwell et al. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comp.*, Aug 1988.

[37] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comp.*, C-30:pp. 478–490, July 1981.

[38] J. A. Fisher. Very long instruction word architectures and the ELI-512. *10th Symp. Computer Architecture*, 1983.

[39] C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comp.*, December 1972.

[40] A. R. Newton G. D. Hachtel and A. L. Sangiovanni-Vincentelli. An algorithm for optimal PLA folding. *1989 IEEE Transactions on CAD*, April 1982.

[41] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[42] B. S. Haroun and M. I. Elmasry. Automatic synthesis of a multi-bus architecture for DSP. *ICCAD*, Nov 1988.

[43] B. S. Haroun and M. I. Elmasry. Architectural synthesis for DSP silicon compilers. *IEEE Transactions on CAD*, April 1989.

[44] W. L. Hsu and G. L. Nemhauser. Algorithms for maximum weight cliques, minimum weighted clique covers and minimum colorings of claw-free perfect graphs. *Annals of Discrete Mathematics 21*, 1984.

[45] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Inc., 1984.

[46] G. Slavenburg J. Labrousse. CREATE-LIFE: A design system for high performance VLSI circuits. *ICCD*, 1988.

[47] M. Breternitz Jr. and A. Nicolau. Integrated code optimization and parallelization. *in preparation*, 1991.

[48] M. Breternitz Jr and J. P. Shen. Organization of array data for concurrent memory access. *21st Workshop on Microprogramming*, Dec 1988.

[49] T. J. Kowalski. *The VLSI Design Automation Assistant: A Knowledge-Based Expert System.* PhD thesis, ECE Dept., Carnegie-Mellon University, April 1984.

[50] D. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1977.

[51] B. McSweeneyand R. Woudsma L. Matterne, D. Chong. A flexible high performance 2-D discrete cosine transform IC. *1989 IEEE International Symposium on Circuits and Systems*, May 1989.

[52] D. B. Gannon L. Snyder, L. H. Jamieson and H. J. Siegel. *Algorithmically Specialized Parallel Computers.* Academic Press Inc, 1985.

[53] ed. L. W. Beineke, R. J. Wilson. *Selected Topics in Graph Theory.* Academic Press, 1978.

[54] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, CS Dept., Carnegie-Mellon University, 1988.

[55] W. T. Lin and C. Y. Ho. *A New FFT Mapping Algorithm for Reducing Traffic in a Processor Array.* VLSI Signal Processing II, edited by S.Kung, R.Owen and N.Nash, IEEE Press, 1986.

[56] R. Camposano M. McFarland SJ, A. C. Parker. Tutorial on high-level synthesis. *25th DAC*, Jun 1988.

[57] M. E. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic, 1987.

[58] S. Manohar and G. Baudet. VLSI supercomputing. *24th Allerton Conference on Communication, Control and Computing*, October 1986.

[59] T. Nakatani and K. Ebcioğlu. Using a lookahead window in a compaction-based parallelizing compiler. *Proceeding of the 23rd ACM/IEEE Workshop on Microprogramming*, November 1990.

[60] A. Nicolau. *Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace-Scheduling Compiler.* PhD thesis, Yale, December 1984.

[61] A. Nicolau. Loop quantization: Unwinding for fine-grain parallelism exploitation. *Dept. of Computer Science Cornell Tech. Report TR 85-709*, October 1985.

[62] A. Nicolau. Percolation scheduling: A parallel compilation technique. *Dept. of Computer Science Cornell Tech. Report TR 85-678*, May 1985.

[63] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Trans. Comp.*, May 1989.

[64] A. Nicolau and J. A. Fisher. Using an oracle to measure parallelism in single instruction stream programs. *14th Annual Workshop on Microprogramming*, October 1981.

[65] Alexandru Nicolau and Roni Potasman. Realistic scheduling: Compaction for pipelined architectures. *23rd ACM/IEEE Workshop on Microprogramming*, 1990.

[66] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Trans. CAD*, March 1988.

[67] K. R. Parthasarathy and G. Ravindra. The strong perfect-graph conjecture is true for $k_{1,3}$-free graphs. *Journal of Combinatorial Theory (B)* 21, 1976.

[68] D. Patterson. Reduced instruction set computers. *CACM*, January 1985.

[69] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. CAD*, June 1989.

[70] J. V. Rajan. *Automatic Synthesis of Microprocessors.* PhD thesis, Carnegie-Mellon University, Dec 1988.

[71] S.G. Rothweiler R.S. Wei and J.Y. Jou. BECOME: Behavior level circuit synthesis based on structure mapping. *Proceedings of the 25th Design Automation Conference*, June 1988.

[72] H. D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Trans. Comp.*, pages 421–428, May 1978.

[73] J. P. Shen. *Personal Communication*, 1990.

[74] H. J. Siegel and J. T. Kuehn. PASM: A partitionable SIMD/MIMD system, for parallel image processing research. *Algorithmically Specialized Parallel Computers*, pages 69–78, 1985.

[75] M. Slater. C& T enters processor market with PUMA. *Microprocessor Report, Vol 4, no. 4*, December 1991.

[76] G. Slavenburg. Signetics VLIW presentation, EE 849 invited presentation. *CMU*, Fall 1990.

[77] D. Springer. *Coloring and Clique Partitioning for Data Path Allocation.* PhD thesis, Carnegie-Mellon University, March 1991.

[78] D. L. Springer and D.E.Thomas. Exploring the special structure of conflict and compatibility graphs in high-level synthesis. *ICCD*, November 1990.

[79] R. M. Stallman. *Using and Porting GNU CC.* Free Software Foundation Inc, 1988.

[80] D. E. Thomas, C. Y. Hitchcock II, T .J. Kowalksi, J. V. Rajan, and R. A. Walker. Automatic data path synthesis. *IEEE Computer*, 16:59–73, December 1983.

[81] R. F. Touzeau. A fortran compiler for the FPS-164 scientific computer. *Proceeding of the SIGPLAN'84 Symposium on Compiler Construction*, June 1984.

[82] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Trans. CAD*, March 1987.

[83] C. J. Tseng. *Automated Synthesis of Data Paths in Digital Systems.* PhD thesis, Carnegie-Mellon University, 1984.

[84] C-J Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Trans. CAD*, CAD-5(3):379–395, July 1986.

[85] P. S. Tseng. Iterative sparse linear system solvers on the warp supercomputer. *Technical Report, Carnegie-Mellon University*, 1987.

[86] WEITEK. *WTL 3132/WTL 3332 Floating Point Processor, WTL 7137 Integer Processor Data Sheet.* WEITEK, June 1986.

[87] A. Wolfe, C. Stephens M. Breternitz J and, A. L. Ting, D. B. Kirk, R. P. Bianchini Jr, and J. P. Shen. The white dwarf: A high-performance application-specific processor. *15 Computer Architecture Conference*, June 1988.

[88] A. Wolfe and J. P. Shen. A variable instruction stream extension to the VLIW architecture. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

[89] D.Chengyu Y.Neuvo and S.K.Mitra. Interpolated finite impulse response filters. *IEEE Trans. ASSP*, June 1984.

[90] G. Zimmermman. MDS - the mimola design method. *Journal of Digital Systems*, 1980.

[91] Z.J.Cendes and D.N.Shenton. Adaptive mesh refinement in the finite element computation of magnetic fields. *IEEE Trans. Mag.*, 1985.