A Vectorizing SUIF Compiler: 
Implementation and 
Performance

by

Derek J. DeVries

A Thesis submitted in conformity with the requirements 
for the Degree of Master of Applied Science in the 
Department of Electrical and Computer Engineering 
University of Toronto

© Copyright by Derek J. DeVries 1997
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

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

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

Desktop computers are increasingly used for DSP, multi-media, and data visualization applications. These codes contain a high degree of loop level parallelism, which cannot be fully exploited by superscalar processors. Vector architectures are a viable alternative for increasing workstation performance.

Vector architectures require more compiler support to exploit the available parallelism in a program than superscalar architectures do. Development effort thus shifts from hardware to compiler design.

This thesis describes the development of a vectorizing compiler, implemented in SUIF, capable of targeting a wide variety of vector architectures. The development of a code generator for the T0 vector-microprocessor is also discussed. Performance of T0-like vector processors on a set of multimedia and data-filter applications is also shown to demonstrate the effectiveness of the compiler and to show the applicability of vector architectures to multi-media applications and other common work loads.
First and foremost, I must thank my supervisor, Corinna G. Lee, for her efforts in this work, for her guidance and encouragement, and for going out of her way to make my masters work an invaluable experience.

Thanks to everyone in the lab (Alex, Rob, Guy, Greg, Steve, Robin, Kelvin, Marcus, Mark) for a seemingly endless stream of inane banter, bad jokes, questionable statistics, and for listening to my rants on a regular basis. Special thanks to Rob, in particular for the statistics (how many cows was it?), and for walking the treacherous path of SUIF with me.

Thanks to Dan, who is also in the lab, but whined until I gave him his own paragraph.

To Karen, for her continual support, encouragement, and for showing me there is more to a weekend than this office.

To Nicole, Eric, Shawn, Kia, Ann, all da’ punks, the SkuleNite 9T5, 9T6, and 9T7 cast and crew, and to everyone else who has ever asked me “are you done yet?”: YES!

I would also like to thank my family for their support and encouragement.

EINSTEIN

drink beer — live lager
# Contents

Abstract .......................... ii

Acknowledgements .................. iii

Table of Contents .................. iv

List of Figures ...................... vi

List of Tables ....................... viii

1 Introduction ........................
   1.1 Motivation .......................... 1
   1.2 Objectives .......................... 2
   1.3 Overview ............................ 2

2 Background ........................
   2.1 Vector Architectures ................ 4
       2.1.1 Torrent Overview .................. 9
           2.1.1.1 The TO ...................... 9
           2.1.1.2 The SPERT board and Environment ........ 12
       2.1.2 New Vector Architectures .......... 12
   2.2 Vector Compilers ................... 13
   2.2.1 Vector FORTRAN .................... 14
   2.3 Superscalar Architectures .......... 14
   2.4 Vector and Superscalar Comparison 17
   2.5 Summary ............................ 18

3 The Compiler ....................... 19
   3.1 SUIF Infrastructure ................. 19
   3.2 SUIF Vectorizing Compiler Overview 20
   3.3 Pre-Vectorization Transforms ......... 23
       3.3.1 Induction Variable Elimination ....... 23
       3.3.2 Privatization and Live-Range Renaming 24
       3.3.3 Reduction Detection and Elimination 25
   3.4 Vectorization ...................... 25
       3.4.1 Dependence Analysis ............... 26
# Table of Contents

3.4.2 Partitioning and Strip mining
  3.4.2.1 Conditionals Within Vectorizable Loops

3.4.3 Expansion of applicability
  3.4.3.1 Non-structured Control Flow
  3.4.3.2 Index Vector

3.5 Post Vectorization
  3.5.1 Pointers and Strides

3.6 Code Generation
  3.6.1 MachSUIF
  3.6.2 General
  3.6.3 Conditionals
  3.6.4 Reductions
  3.6.5 Register Allocation
  3.6.6 T0 Quirks
  3.6.7 Unsupported Transforms
  3.6.8 Object Code Generation

3.7 Summary

4 Experimental Results
  4.1 Correctness and Verification
  4.2 Effectiveness Relative to Other Compilers
    4.2.1 Summary
  4.3 Vector Performance
    4.3.1 Simulation Environment
      4.3.1.1 Cello
      4.3.1.2 Viola
    4.3.2 Memory Systems
  4.4 Performance Comparisons
    4.4.1 Simple Loops
      4.4.1.1 Straight line Loops
      4.4.1.2 Control Flow
    4.4.2 MPEG Kernels
    4.4.3 PGP and Others
  4.5 Summary

5 Conclusions
  5.1 Contributions
  5.2 Future Work

Bibliography
# List of Figures

2.1 Sample Loop ................................................. 5
2.2 T0 Block Diagram ........................................... 10
2.3 T0 Timing Diagram .......................................... 11
2.4 Block Diagram of a Super Scalar ......................... 16

3.1 Compiler Overview .......................................... 21
3.2 Scalar Expansion ........................................... 24
3.3 Vectorizer Overview ........................................ 26
3.4 Sample dependence vectors and Graphs .................... 28
3.5 Tarjan's algorithm ......................................... 30
3.6 Vectorization of a simple loop ............................ 35
3.7 If Conversion ............................................... 38
3.8 Breaks in loops ............................................ 41
3.9 Index vector transform ..................................... 43
3.10 Induction Variable Transformations ....................... 45
3.11 reduction algorithm ...................................... 49
3.12 Multiplication ............................................. 52

4.1 Simulation Environment Overview ......................... 66
4.2 Simulation Environment Parameters ....................... 70
4.3 Simple Loops Code .......................................... 74
4.4 Addition .................................................... 74
4.5 strided add ................................................ 75
4.6 Addition timing diagram ................................... 77
4.7 Multiply VV ................................................ 78
4.8 Multiply SV ................................................ 79
4.9 Control Flow Loops ......................................... 80
4.10 Simple IF .................................................. 80
4.11 Nested IF .................................................. 82
4.12 Fast IF ..................................................... 83
4.13 FDCT ....................................................... 85
4.14 Motion Test ................................................ 86
4.15 Motion Add ................................................ 86
4.16 IDCT ....................................................... 88
4.17 Dither ....................................................... 90
4.18 Matrix Multiply loops ..................................... 91
List of Tables

4.1 Summary of ANL loops ........................................... 58
4.2 Summary of Benchmarks ......................................... 73
Chapter 1

Introduction

1.1 Motivation

Despite advances in IC fabrication technologies, increasing the issue widths of superscalar processors, while dropping cycle time, cannot go on indefinitely. Design and validation complexity, physical limits on signal propagation across a die, and diminishing returns for increased issue width all point to a need for architectural alternatives to (super)scalar execution. Vector architectures provide a viable option for increasing performance without adversely affecting cycle time or time to market.

Vector architectures permit the exploitation of large amounts of parallelism within loops, without the need for the extensive control logic required for a superscalar machine to exploit the same parallelism. Vector machines have typically been used for large scientific and engineering applications due to the high cost of operation. Increased transistor density allows the creation of a vector microprocessor, allowing this kind of computation power to be brought to the desktop.

Desktop machines are increasingly utilized with DSP, multi-media, and data visualization applications. Such codes contain a high degree of loop level parallelism, which can be exploited by a vector processor.

Using multiple processors is another option for bringing improved performance to a computer system. These machines exploit parallelism within a program by partitioning work across the available processors. This approach is orthogonal to
1.2 Objectives

This thesis has three important goals: the development of an architecture independent tool for locating vector parallelism within a program, the development of a complementary assembly code generator for the vectorizer, and a performance study to explore the effectiveness of this compiler.

A vectorization tool, using a minimum of assumptions about the target architecture, will be able to provide a common front-end for a wide variety of target architectures, ranging from a traditional vector architecture with large vector register files, to new processors, containing multi-media instructions, introduced by all major processor design companies.

Assembly code generation is important to provide a mechanism for measuring the effectiveness of the vectorization tool. The target processor of the work in this thesis is the T0, a machine developed at Berkeley for which no vectorizing compiler exists. The performance study is conducted in two directions: in terms of both the compiler's ability to locate and exploit vectorizable constructs within source code and in terms of the performance of its emitted code on a number of Torrent-like hardware configurations as compared to a variety of superscalar configurations.

1.3 Overview

This thesis is organized as follows. Chapter 2 presents an overview of both vector and superscalar architectures, outlines some of the strengths and weaknesses of these two architectural approaches, and introduces some of the problems faced in compiling for vector machines. Chapter 3 presents the SUIF vectorizing compiler and assembly code generator developed in this thesis, describing their implementation and presenting the strengths, weaknesses of this software. Chapter 4 presents two sets of performance results. The first set characterizes the effectiveness of the compiler at locating
vectorizable code sequences and is based on a set of published vector-compiler test
loops. The second set of performance numbers show the effectiveness of the compiler
at exploiting vector parallelism by applying it to a number of simple, multimedia,
and filter kernels and simulating the resultant code on a variety of superscalar and
vector processor configurations. Chapter 5 contains some concluding remarks.
Chapter 2

Background

Vector processing is a style of program execution where loop level data parallelism is found and exploited within a program. This is similar to doall parallelism, where each iteration of a loop body could be executed independently of all others without synchronization, but is not as constraining. Rather, a vectorizing compiler locates expressions within a loop which can be evaluated for the entire iteration space, reordering statements within a loop so that such execution is legal. In this chapter, the main ideas of vector processing will be introduced and compared to superscalar approaches of exploiting parallelism within a loop. Some of the difficulties encountered when compiling for a vector machine are also introduced.

2.1 Vector Architectures

Vector processors are computers which provide special hardware for operating on vectors of data. The way these machines operate is best described by example. Figure 2.1(a) shows a simple, vectorizable loop. It can be seen that all reads of the b and c arrays could be performed at the same time, as could the addition and store once the data dependencies from the loads are resolved. Vector processors provide special hardware for exploiting this kind of parallelism. Vector processors contain a special register file for holding sets of data. Typically 32-128 elements can be held within a single vector register. Instructions issued with vector registers as operands
Figure 2.1: Sample Loop. Part (a) shows a simple array based loop. Part (b) shows a pseudo-assembly implementation of the loop for a (super)scalar processor. Part (c) shows the corresponding implementation on a vector machine.

operate on all elements in the specified registers.

Figure 2.1(b) and (c) show pseudo-assembly code for scalar and vector execution of the loop. The two code fragments are almost identical. Instructions S[1-4] and V[1-4] initialize 4 scalar registers with pointer values, locating the read and write positions for the current loop iteration and the limit of the loop. Instruction V4a is unique to the vector code sequence. The Vector Length Register (VLR) is a control register within a vector processor which sets the number of elements operated on by a vector instruction. Each vector instruction reads the VLR on issue and performs the specified operation on the first VLR elements of the vector registers specified as operands. In this example assume a vector register file containing registers of 32 elements each. The loop limit has been chosen so as to be an integer multiple of the maximum vector length, \( VLRMAX \).

Instructions S[5-6,8] and V[5-6,8] perform the memory operations in the loop. The vector memory operations load(or store) VLR elements of the source(target) array into(from) a vector register. These operations require one extra operand compared to the scalar memory operation: the stride through memory must be specified. The stride value specifies the distance in memory between target elements of the array. Unit stride accesses are shown in this loop; adjacent elements of the array are loaded(stored). Strided memory operation would load every \( n^{th} \) element of the array into adjacent elements of the vector register. Many vector processors provide unit
Instructions S7 and V7 perform the addition in the loop. V7 initiates VLR adds, while S7 initiates only one. Instructions S[9-12] and V[9-12] update all pointer values and perform the loop back test.

This short code sequence shows one of the advantages of vector execution:

*Vector processing can greatly reduce instruction bandwidth required to perform a set of operations.*

Executing the scalar loop would require issuing $4 + 96 \times 8 = 772$ instructions, while vector execution requires only $5 + 3 \times 8 = 29$ instructions to be issued, since the vector loop is only executed three times. Hardware within the processor effectively performs the loop-back test and pointer updates for the VLR operations initiated with each vector instruction.

Processor designers can also exploit another property of vector instructions:

*Operations specified within a vector instruction are guaranteed to be independent.*

This property can be exploited in several ways. Deeply pipelined functional units can greatly reduce the cycle time of a processor, at the cost of increased latency between the time of issue and the issue of a data dependent instruction. The vector execution model allows operations to be issued to the pipeline each cycle from the same instruction without the need for dependence checks between all elements operated on by the instruction. This prevent bubbles in the pipeline without added hardware costs. Data dependencies can exist between instructions, and dependent instructions must be stalled until the first element is available and any other latency constraints are met, but, assuming a second functional unit is available, may issue one operation per cycle after that.
plementation contains $N$ pipelines within a single functional unit. The elements of vector registers are striped across these pipes. Thus $\lceil VLR / N \rceil$ cycles are needed to initiate $VLR$ operations. The amount of parallelism that can be exploited when using parallel pipes is limited by $N$ per functional unit, where $N \leq VL_{MAX}$.

In cases where the data size is not a multiple of the natural size of the machine the loop must be strip mined [BGS93]. The iteration space must be broken up into sections of length $VL_{MAX}$. A small amount of additional overhead must be added to perform an extra iteration of the loop body with a VLR value less than $VL_{MAX}$. The amount of overhead is dependent on implementation, but as little as one scalar instruction per strip mined loop iteration could be introduced.

Pipelined scalar processors often employ data forwarding to reduce pipeline stalls due to data dependencies. Within a multi-stage pipeline there are usually some number of pipe-stages between the calculation of a value and its appearance in the register file. Forwarding makes the calculated value available at the head of the execute stage as soon as it is available. This allows dependent instructions, which would otherwise have to wait until the value was written to the register file, to be issued sooner.

Within vector processors this technique is called chaining. Due to the deep pipelining or wide parallel pipes involved forwarding across functional units is difficult without the addition of extra buffers. Chaining occurs through the register-file instead. Instructions are issued as soon as instructions on which they depend have written back enough values for the second instruction to finish without encountering a data hazard. The first instruction completes while the second instruction is executing.

The latency incurred by chaining is dependent on implementation. If reads and writes are possible to the same elements of the vector register in the same cycle, no latency would be present, otherwise at least one cycle of added latency would be incurred.

For example, in a parallel-pipe-style vector processor where all functional units return four elements per cycle, for a vector length of 64 and chaining and functional
register file) in 18 cycles. A processor without chaining would have to wait that long for the data before issuing a dependent integer operation, taking a total of 36 cycles to perform a load-and-add sequence of operations. With chaining, the integer operation could be issued two cycles after the load, allowing the load-and-add to complete in only 20 cycles.

In addition to hardware to support vector execution, scalar support is also required. Many vector operations require some scalar operands, such as the base address for loads. Other programming constructs, such as branches and function calls are inherently scalar, so a scalar functional unit must be supplied to support them. Finally, some algorithms (or components of algorithms) cannot be vectorized, so full scalar support must be provided to execute them.

Vector processors were first introduced in the early 1970’s with the CDC STAR-100 and the TI ASC both introduced in 1972. These both suffered from poor scalar performance and high vector start-up overhead caused by a memory-memory organization. All instructions operated on memory elements, rather than registers. Cray Research introduced the CRAY-1 in 1976. This was the first vector-register based machine and also the first machine capable of chaining vector operations. The CRAY X-MP in 1983 and the Y-MP in 1988 improved upon this design and incorporated the latest IC fabrication technology. These machines were all aimed at the supercomputer market and were used to solve large scientific and engineering problems.

The early eighties also saw the introduction of “minisupercomputers” such as the CONVEX C1; machines built along similar lines to the supercomputers, but with roughly one tenth the cost, making them viable for use in smaller science and engineering applications.

These machines all provided high performance for highly parallel applications, but increased research into superscalar machines and multiprocessors, and the promise of good performance on a broader class of applications led superscalar machines to a dominant position in the market place.

The rising importance of multimedia applications such as video players, real-time
parallel applications into the marketplace. Processor design companies have introduced the latest generation of vector machines, microprocessors with multi-media instruction sets, to compete in this growing market.

2.1.1 Torrent Overview

The Torrent vector processor [AJ96] is the target architecture of the compiler developed in this thesis. The Torrent Instruction Set Architecture (ISA) is based on the industry standard MIPS-II ISA, with the vector unit implemented as coprocessor 2. The ISA specifies a 32-bit machine and a maximum of 32 vector registers along with a set of vector and vector-unit control instructions. The ISA does not constrain the implementation, and does not guarantee an execution order for data elements within a vector operation. Implementations are permitted to extend, but not otherwise change the ISA specification.

2.1.1.1 The TO

A block diagram of the current version of this machine, the TO [AB95], is shown in Figure 2.2. The vector unit contains 15 32-element, 32-bit general purpose registers, and a sixteenth zero-vector register, connected to two vector functional units (VP0 and VP1) and one vector memory unit with a 128-bit data path to memory. A scalar register set and functional unit is also present. Scalar memory operations execute in the vector memory pipeline. The TO is a single-instruction, in-order-issue machine.

The TO is a parallel-pipe style machine. Each vector functional unit is composed of eight parallel pipes, with the vector register file striped across these pipes as shown by the shaded area in Figure 2.2. This allows each vector math instruction to initiate 8 operations per cycle. Each functional unit supports simple integer arithmetic and logical operations. In addition one of the vector units, VP0, contains a 16-bit fixed

---

1 Reads from this vector register always return zero in each element, while writes result in the data being discarded. This is useful as a fast mechanism for obtaining a common constant and for executing instructions for which only side effects are of interest.
point multiplier. Instructions are preferentially issued to VP1. They are only issued to VP0 if they are a multiply instruction or if VP1 is busy.

Memory access is supplied to both the scalar and vector units through the vector memory unit. Only one set of address lines go to memory, limiting memory performance. Scalar and strided memory accesses complete at a rate of one word per cycle, since an address must be specified for each element individually. Unit-stride vector memory accesses return four elements per cycle for word accesses and eight elements per cycle for half-word and byte accesses.

The stall time for instructions dependent on memory operations is dependent on both the value in the VLR and the type of access being performed. Assuming the VLR is set to 32, instructions dependent upon unit stride, 32-bit, memory accesses
Figure 2.3: T0 timing diagram. Execution of one iteration of the loop in Figure 2.1 must stall for five cycles, while those dependent on unit stride byte (or half word) accesses need to stall for two. Instructions dependent on strided accesses must stall for 29 cycles, regardless of the data size.

Memory performance can also be degraded by I-cache misses. The T0 contains a 1KB I-cache, which must be filled through the single memory port. Processor stalls of up to 3 cycles can be incurred by I-cache misses. Math operations can proceed in this time, but memory operations are suspended to permit the cache fill to occur.

The T0 [WAK+96] is implemented in 1.0um CMOS on a 17mm×17mm die in two layers of metal. It has a maximum clock rate of 45MHz.

The T0 execution of the loop in Figure 2.1 is shown in Figure 2.3. A strip mined loop iteration is executed in 13 cycles. This corresponds to 32 iterations of the scalar loop, giving 0.4 cycles/loop iteration. This pattern would be replicated for each loop iteration. Unit stride memory operations on the T0 can optionally increment the register containing the base address. Using this feature would remove instructions V[9-11], and permit the loop-back overhead to be completely masked by the vector store operation, reducing execution to 12 cycles, or 0.375 cycles/loop iteration. Even without this feature, V9 could be executed as early as cycle t1, where there is an available issue slot, again reducing execution to 12 cycles.
The T0 is currently available on a S-bus expansion card for SUN workstations called SPERT-II. This board runs at 40MHz and can contain 8MB of wave-pipelined SRAM capable of returning addressed values in 1.5 cycles. The T0 is mounted with chip-on-board (COB) technology.

The SPERT development environment is based on the GNU tool set. This provides a workstation-like environment to ease the porting existing applications to the SPERT hardware and the development of new applications. Assemblers, linkers and debuggers have all been developed which work with the SPERT board, allowing programmers to include vector-assembly language sequences in their applications, and interactively debug these applications on the SPERT board. These tools also work with two simulators included in the environment.

An ISA and RTL (Register Transfer Level) simulator have been developed for the SPERT board. The ISA simulator is a simulator for verifying correctness of an application. It simulates the T0 and the SPERT board at the ISA level. Each instruction executes in one cycle. The RTL simulator provides cycle-accurate timing information, at the cost of simulation speed. All vector and memory pipelines are modeled, including all interactions with the host machine, instruction cache behaviour, and interference between I-cache loads and data references. The ISA simulator can complete several thousand instructions per second, while the RTL simulator is limited to about 100 instructions per second.

GCC has also been ported to the SPERT environment. This compiler supports vector instructions through assembly language programming and through calls to a hand-coded vector library. It is not a vectorizing compiler.

2.1.2 New Vector Architectures

The preceding sections have described classic vector architectures such as the T0. A new variety of vector machines have been introduced by most major processor developers. Multi-media instruction sets, such as the SPARC VIS [KMT+95] instructions,
2 [Lee96] extensions utilize existing, wide datapaths to perform multiple operations in parallel on short data types. A 32-bit data-path, for example, could be split to perform four eight-bit additions with a very small cost in die area. SUN reports less than a 3% increase in die area [KMT+95] in its implementation of VIS. These machines typically use the existing floating point registers as a “vector register file”, relying on separating floating point and integer operations to avoid register conflicts. While these instructions cannot exploit as much parallelism as traditional vector processors, they can provide speed-up in many applications, and are aimed specifically at reducing the execution time of multi-media applications which typically contain algorithms operating on small integer data-types.

### 2.2 Vector Compilers

Compilers for vector machines, in addition to performing the usual optimizations performed for scalar execution, face the problem of identifying vectorizable loops written in scalar languages. Aggressive dependence analysis is required to identify memory access patterns suitable for vector execution. This limits programs to array-based data structures, or any other structure which can have uniqueness of data elements proven statically.

In addition to being able to prove the disjointness of memory operations, the compiler must use dependence analysis to determine a safe ordering for execution. Vector execution can tolerate loop-carried dependencies, unlike a do-all execution style, but it cannot tolerate dependence cycles. Dependence cycles occur when a statement requires data in iteration \( i \) which it calculated in iteration \( i - n, n > 0 \). This constrain includes indirect dependence cycles, where statement A depends on statement B, which in turn depends on A.

Compilers must also try to emit the highest possible quality of code. This means a number of loop transformations are often incorporated into the compiler, to both increase the number of loops which are vectorizable and to increase the execution
memory access into a unit stride access can have dramatic effect on performance.

2.2.1 Vector FORTRAN

FORTRAN, and other programming languages, have been extended to include statements with vector semantics. These extensions allow the programmer to write applications in a manner which is easily mapped onto vector machines. The problems faced by compilers for vector languages are the reverse of those faced by vector compilers for scalar languages. Code with vector semantics must still be strip mined to fit on the available hardware, but the semantics of the language are such that an entire vector statement is completed before the next one is executed. Thus each program statement must be strip mined separately. Numerous loop aligning transforms are applied in order to perform loop fusion on as many strip mined vector statements as possible, thus reducing scalar overhead.

2.3 Superscalar Architectures

A block-diagram of a superscalar machine is shown in Figure 2.4. These machines contain multiple copies of functional units and a complex instruction scheduling mechanism. The instruction fetcher loads some number of instructions from the I-cache and issues them to positions in the reorder buffer and the reservation stations for the required functional unit. An instruction moves from a reservation station into the functional unit when its source operands become available. These operands are retrieved from the register file, the output of other functional units, or from the special rename registers within the reorder buffer.

Superscalar processors exploit instruction-level parallelism by issuing multiple scalar instructions simultaneously. These instructions could be issued either in or out of order. In-order machines have simpler control structures, but cannot exploit as much parallelism without the compiler scheduling the instructions. This scheduling will not produce good performance on machines with different pipeline configurations.
it to exploit the available parallelism. They are able to reorder instructions to execute on available functional units. They can even reorder instructions for which dependence relationships are unresolvable at compile time, since all run-time information can be taken into account. In either case, the machine will have a limit on the number of instructions that can be issued and committed in each cycle.

To supply the large number of instructions needed to fully utilize a superscalar machine, branches in the code are predicted and fetching and execution proceeds along the predicted path. If this path is determined to be false all speculatively issue instructions must be flushed. The speculative nature of the instructions makes it impossible for these instructions to write results to the register file, so a set of renaming registers is provided, to act as targets for speculative operations. References to registers defined during speculative execution are filled from the rename register pool, rather than the real register file.

The re-order buffer contains the rename registers and performs the bookkeeping required to commit only those speculatively executed instructions for which it has been proven that they execute (i.e., all previous branches have been resolved). This mechanism allows for out-of-order execution of the program, but ensures in-order commitment of results, thus preserving correctness. Modern processors issue between two and six instructions per cycle.

Multiple memory units, which are desirable for performance, cause complications in these processors. In an instruction stream containing both loads and stores, the data dependencies restricting the issue of a load following a store may be resolved before the dependencies constraining the store are resolved. It is desirable to execute the load, making code dependent on it available for execution, but load cannot be performed safely until the address of the store operation is known. The address of the store isn’t available until it is actually issued, because of the address calculations implicit in the instruction. The load must therefore be stalled.

---

2 The base address of the load has been calculated before the base address and the data element for the store are available.
buffer (MAB) can be used to alleviate this problem.

Loads and stores are issued as soon as their respective addresses operands become available. Stores wait in the MAB until their data is available, while loads wait until the addresses of all stores preceding them in the dynamic instruction stream have been resolved and any possible conflict identified. Operations to identical addresses can also be eliminated. Speculative loads can be performed without difficulty, since they do not change the contents of memory, but speculative stores cannot be issued to the memory system until the instruction is committed in the ROB.

To avoid serialization through the memory system, multi-ported memory from the MAB must be provided, with each port being able to access an arbitrary address.

A real superscalar may not contain all of these features or may incorporate them into other organizations. For a detail discussion of these machines see [HP96].

Returning to Figure 2.1 the performance of a superscalar machine can be approximated. Assuming a large number of functional units, but only one branch unit, and a good scheduling mechanism a processor without an MAB, but with multiple memory ports, could execute the loop in 3 cycles: one to load the two source operands, one to add and one to store the result. The processor cannot perform any memory operations after a store in the dynamic instruction stream until after the store has been
increases to one cycle per iteration: the time required to evaluate the branch.

2.4 Vector and Superscalar Comparison

Vector and superscalar machines both attempt to exploit parallelism within a program. Vector machines operate at the loop level, executing multiple iterations of a loop concurrently, while superscalars attempt to locate independent instructions within a scalar instruction stream and issue them simultaneously.

These differences in style enable the vector machine to execute far fewer instruction in order to perform the same set of operations, and the execution of each instruction can be easily pipelined or parallelized. The set of applications that can exploit the vector machine's abilities is seen to be limited, however. Superscalar machines cannot exploit as much parallelism as vector machines, since their issue rates cannot be scaled to match the effective issue rates of vector processors. In the case of the T0 up to 25 operations could be “issued” simultaneously. On the other hand, superscalar processors can utilize parallelism from applications which cannot be run on vector machines.

Vector machines require complex compilers to exploit their abilities, but, for codes which exhibit regular memory access patterns, can provide larger speedups than superscalar processors. Superscalars can have simpler compilers, but have complex control logic requirements.

A common belief is that vector machines require complex memory systems to support the parallelism available in their functional units. As the T0 shows (see Chapter 4), a wide data-path is important, but good performance can be had with only one address port if memory traffic is dominated by unit-stride accesses. This is in contrast to superscalars, which require a separate address port for each memory functional unit and a memory access buffer to allow loads to bypass stores.

Another point of comparison for for the two architectures is in terms die area and
Given that the the R10000 is fabricated in a process technology which allows for more dense circuit layout than the unscaled T0's process, the size of the vector machine could be reduced even further through process migration. The clock rate of the T0 is comparable to small issue super-scalar machines implemented in the same process, so it would be expected that the scaled T0 would achieve a clock rate at least as high as the R10000. In this area, the T0 effectively issues up to 25 scalar operations per cycle (three eight-element vector operations plus one scalar operation), making it 100 times as area efficient as the superscalar machine $(25 \div 4) \times 16$.

Performance comparisons for these machine types are presented in Chapter 4.

2.5 Summary

This chapter has presented an outline of both vector and superscalar processor architectures and the strengths and weaknesses of each. Compiler requirements for each where also touched upon. Subsequent chapters will explore the compiler requirements in detail and present a performance comparison of a variety of vector and superscalar configurations.

---

The R10000 is a 4-way issue superscalar processor.
Chapter 3

The Compiler

This chapter describes the implementation of the SUIF vectorizing compiler and the infrastructure on which it is based. Its strengths and limitations are discussed and the complete flow of compilation is detailed.

An important point throughout this chapter is the idea of architecture independence. This compiler has two distinct sections: the high level transformation section and the assembly code generator. The goal of the high level transformer is to produce vectorized code which can be mapped onto any vector architecture: a traditional one, like the T0, or newer microprocessors with multi-media extensions, such as the SPARC VIS or Intel MMX instructions. Code generation is an architecture dependent step, but is portable, if not optimal, for a family of processors sharing the same ISA.

3.1 SUIF Infrastructure

The vectorizing compiler is based on the SUIF\textsuperscript{1}-1.1.0 infrastructure [WFW+94] with the \textit{machSUIF-1.0.0} [Smi96] extensions for specialized code generation and machine-dependent optimization. This approach allowed effort to be concentrated on the specifics of code transformation and assembly code generation using the extensible framework provided by these packages and the leveraging of existing functionality

\footnote{\textsuperscript{1}SUIF stands for Stanford University Intermediate Form.}
The SUIF compiler is built from a set of discrete executables, or passes, with each pass communicating with other passes via the file system. Internally SUIF represents a program as an object-based hierarchy of symbol tables and lists of abstract syntax trees (ASTs). Each expression tree represents one program statement encoded in a generalized, RISC-like instruction set. More complex encodings are also supplied, such as array-instructions, that encapsulate entire ASTs within themselves, and ensure that subscript expressions are clearly separated from statement level expressions.

A passes communicates information to a subsequent pass in two ways: code transformation and annotations. Annotations are markers that can be attached to almost all objects in the intermediate form, instructions and symbol table entries being the most common targets. Annotations contain lists of simple data items, such as strings and integers, or SUIF objects, such as symbols and types. More complex structured annotations, containing full C structures, can also be created. These annotations can be read and re-written by a pass to obtain or expand on previously determined information.

Communication through the file system results in slower compile times, but it permits the compilation sequence to be varied and custom passes inserted with little effort.

In addition to basic scalar compiler functionality, a number of passes and libraries are provided for detecting constructs such as reductions and induction variables, for performing unimodular and other code transformations, and for performing dependence analysis.

### 3.2 SUIF Vectorizing Compiler Overview

Vectorization involves a number of high-level transformations implemented within SUIF, followed by assembly code generation. The flow of compilation is shown in Figure 3.1. The extent of changes and additions to SUIF are also depicted in the figure. Italicized text indicates places where existing code was modified to provide
Figure 3.1: Data flow diagram for the SUIF vectorizing compiler developed in this thesis. Italicized text indicates places where code was modified to provide functionality needed by the vectorizer. Shaded bubbles indicate where new passes were added. The remainder of this chapter describes the process of compilation and the changes made in detail.

The flow of compilation is as follows:

**Source Code Translation:** Before analysis and transformation can begin, the source code for the program must be translated into the compiler’s intermediate form. C is the only language which can be directly translated, but in principle any source language for which a language-to-C translator is available could be vectorized without changes to subsequent steps. In this thesis both
C and FORTRAN are used as source languages. Translation from C to FORTRAN results in a loss of information: functionally equivalent C code may not express all of the semantics expressed in the original FORTRAN source. SUIF does contain a pass, fixfortran, used during the compilation of FORTRAN programs to rebuild as much of the information that was lost during the translation to C as possible.

Pre-vectorization Transforms: A number of transforms and analyses must be performed before vectorization can begin. These simplify the contents of loops, reducing the overall work done by the program being compiled, as well as simplifying the work to be done by subsequent compiler passes. In addition, special programming constructs, such as reductions and induction variables, are identified and, if possible, eliminated.

Vectorizing Transforms: A new pass, the vectorizer, was added to the compilation sequence. This pass performs a number of analyses to determine which loops within a program can be vectorized and performs all the required code restructuring. This pass is almost entirely architecture independent; only the maximum vector length of the target architecture, $V L_{M A X}$, is required. The output of this pass, when translated into C, resembles the original loop nest, with the step size of the vectorized loops increased by a factor of $V L_{M A X}$. Statements within this loop are annotated, indicating that they have been vectorized. Many other code transformations, described in Section 3.4 are also performed at this time. The value of $V L_{M A X}$ is supplied as a command line argument.

Post-vectorization Transforms: Once vectorization is complete, a number of transformations and optimizations, including determining memory access strides, are applied.

Code Generation: The final step of compilation is the translation of the intermediate form to an assembly language program. This process includes register al-
The next stage, object code generation, is left to a native assembler of the target architecture.

3.3 Pre-Vectorization Transforms

Before vectorization begins a number of traditional optimizations are performed, the most important of which are outlined below.

3.3.1 Induction Variable Elimination

Induction variables are scalars which, like the loop index variable, change value through the execution of the loop. Removing them, and replacing them with expressions involving the loop index variable has two major advantages.

Firstly, dependence analysis is greatly simplified by the removal of extra variables if they participate in array subscript expressions. Fewer conservative dependencies have to be assumed and the chance that a loop can be found to be vectorizable is greatly improved.

Secondly, as shown in Section 3.4.3.2, the index variable can participate in vectorizable expressions, however other induction variables cannot without more detailed analysis. Replacing induction variables with expressions involving the loop index removes scalar dependence constraints from the loop, again increasing the likelihood that vectorizable statements within the loop can be found.

A simple induction variable eliminator is distributed with SUIF and was used in an unmodified form in this compiler. It is effective at finding simple induction variables in many loops, but more complex induction variables, such as ones that are updated multiple times within a loop iteration, are ignored.
for(i=0; i<100; i++)
{
    s=b[i]+c[i];
    a[i]=s+2;
    s=d[i]+c[i];
    d[i]=s+2;
}

(a)

for(i=0; i<100; i++)
{
    privatized_ssa_s13[i]=b[i] + c[i];
    a[i]=privatized_ssa_s13[i] + 2;
    privatized_ssa_s24[i]=d[i] + c[i];
    d[i]=privatized_ssa_s24[i] + 2;
}

(b)

Figure 3.2: Applying scalar renaming and expansion to a simple loop

3.3.2 Privatization and Live-Range Renaming

Scalar expansion or scalar privatization[BGS93] is one of the most important transformations to be applied before vectorization. Scalar variables that are written and then read within one iteration of the scalar loop can be replaced with an array, thereby giving each iteration of the loop its own copy of the scalar variable and permitting vectorization. Scalar variables which carry data across an iteration boundary or into the loop cannot be handled easily with this scheme.

The SUIF environment supplies only a detection and marking pass. The marking pass was modified to perform the actual array replacement of the scalar variable. As an extension to this, scalar renaming is simultaneously applied.

Common programming practice is to reuse a variable once the value it contains is no longer useful. As is shown in Figure 3.2(a), a scalar variable is reused over two live ranges. Flow dependencies exist between each definition and subsequent use, but the anti-dependencies between the final use of a live range and the following definition introduce artificial ordering constraints on the loop which may result in a dependence cycle (see Section 2.2 for a definition of dependence cycles), and thus large portions of the loop becoming non-vectorizable. Renaming privatizable scalar variables at the end of each live range removes these dependencies, broadening the set of possible vectorizable loops and increasing the flexibility that exists for scheduling a vectorized loop.

Array expansion [BGS93] is not performed in the compiler.
Reductions are an important class of program construct that can be efficiently handled on a vector processor if the reduction can be detected. Some reductions can be completely eliminated from a loop. Ones of the form \( s = s + a \), where \( a \) is a constant, can be replaced by a summation outside the loop. This replacement is done before privatization occurs. More complex reductions of the form \( s = s + f(i) \), where \( i \) is the loop index, must be calculated within the loop. Some vector machines, like the IBM System/370 [cC91], have hardware support for the reduction of a vector (with a limited set of operators). Failing that, efficient reduction algorithms can also be implemented using vector hardware that are faster than simple scalar execution.

Reductions of this form are identified and marked with an annotation by a reduction detection pass distributed with the SUIF package. This pass is limited to locating single-statement reductions using addition, multiplication, maximum, or minimum operators.

### 3.4 Vectorization

Vectorization is the process whereby serial code is transformed into a form that will utilize the abilities of a vector processor (see Section 2.1). This requires that all inner loops\(^2\) be examined and vectorizable code sequences extracted. The extracted code must be transformed such that all dependence constraints be met and the loop fits on the hardware executing the code. At this stage of compilation, the only hardware-dependent parameter that needs to be known is the maximum vector length. This compiler obtains the value for \( VL_{MAX} \) from the command line, enabling it to target machines of any size. If the run-time environment were allowed to supply this value, all transformations could be done symbolically. This would increase scalar overhead, since no optimizations based on known constant values could be performed, but no assumptions would need to be made in compilation.

\(^2\)Some vector architectures such as the Burroughs BSP and Texas Instruments ASC support multi-dimensional vectorization, so the inner \( n \) loops could be vectorized.
The vectorization process used by this compiler is shown in Figure 3.3. In this figure the process is loosely divided into two distinct phases: analysis, where the input code is examined and all required transformations are determined, and transformation, where the loop is restructured into a vectorized form.

### 3.4.1 Dependence Analysis

In order to determine the set of vectorizable statements within a loop, a *dependence graph* must be built and analyzed. A dependence graph encapsulates all data flow information available about a loop in a directed graph. Edges in the graph indicate the flow of data between array references within the program. Both the type (*true*, *anti* or *output*) and the dependence distance in loop iterations are recorded for each edge in the graph. RAR dependencies are ignored. The nodes in the graph determine the finest granularity at which the graph can be analysed. The graph could be built with each node representing the references themselves, whole statements, or even sets of statements.

To build the graph, a list of all array references within an inner loop is gathered and iterated over to find the dependence relationship between each pair. The SUIF dependence library [Sta94] reports all dependence vectors flowing *from* one ref-
are produced. From each set at most one vector, the most restrictive, is chosen to characterize the relationship between the two. Since the vectorizer only works on the innermost loop, many vectors in the sets reported by the library for references within nested loops can be disregarded. Dependence vectors with non-zero distances in higher nesting levels are disregarded since the dependence is carried by these outer loops. Vectors which report a self-anti-dependence⁢ are also discarded since the load is assured to be performed before the store. Distances that are known constant values are normalized by the loop index increment, and values that are not whole numbers are discarded since these distances are not aligned to possible values of the loop index variable. Normalized distances greater than the maximum vector length can also be discarded. Of the remaining dependence vectors, only the most general is used to characterize the relationship between the two references. In use, a general dependence vector produces the most conservative representation of the dependence relationship between two references, and so is the most restrictive to code transformations.

The algorithm used to determine the vectorizability of a code fragment operates at the statement level, so in the case of this vectorizer the nodes of the graph represent statements. This sacrifices some detail, but simplifies the algorithm and removes the need to perform node splitting. Node splitting is a transformation which breaks a single statement into subexpressions, some of which may be vectorizable. Each dependence vector, and its associated bookkeeping information, is stored as a structured annotation at the root node of the expression tree containing the tail of the dependence vector. This process produces a conservative dependence graph of the program containing all true, anti, and output dependencies for use in the tests for vectorizability.

Figure 3.4 shows the dependence vectors and resultant graphs produced for two loops. The first singly-nested loop contains one pair of constraining references: \( e[i] \) in statement \( S1 \) and \( e[i+t] \) in \( S2 \). The read of \( e[i+t] \) occurs to a point an unknown displacement from the position of the write. This results in three vectors being

⁢The source and sink of the anti-dependence appear in the same statement.
for(i=0;i<100;i++)
{
    S1:  e[i]=f[i]+i0;
    S2:  g[i]=e[i]+i+b[i];
}

for(i=0;i<100;i++)
for(j=0;j<100;j++)
{
    S1:  a[i][j]=b[i][j]+c[i][j];
    S2:  d[i][j]=a[i][j]+u[i-1][j-1]*
        a[i][j-1];
}

Figure 3.4: Sample loops and the resultant graphs, with dependence vectors, are shown. Gray dependence arcs indicate vectors which are reported but are not included in the pruned dependence graph.

reported by the library: true dependence vectors (+) and (0) and an anti-dependence vector (+). A (+) vector relating two references means that the same memory location is accessed by the second reference some number of iterations after the first reference, the exact number being unknown. A (0) means the two references access the same location in the same iteration. Applying the criteria above to these vectors, the anti-dependence vector and the (+) flow-dependence vector are entered into the graph. The true dependence vector (0), shown with a shaded arrow, is disregarded.

In the second loop there are three pairs of constraining references, all involving the array a. In this case only one vector is reported for each reference pair. Again applying the qualification criteria, the (11) vector is disregarded since the dependence is not carried by the inner loop, while the other two edges are entered into the graph.

Tarjan's algorithm [Wol89], as depicted in Figure 3.5 is used to determine which statements within the loop may be executed on a vector processor. This is a depth-first search, iterating over the dependence graph, looking for nodes that do not participate in dependence cycles. Statements that participate in dependence cycles cannot
vectorizable by this algorithm.

The STACK and LOWLINK data structures are the key components of this algorithm. 
STACK records the forward path through the graph to the current node, while LOWLINK records the smallest statement number on which the current statement depends. INSTACK is simply a fast lookup mechanism to determine if a node is in the stack.

The NUMBER construct is a work-saving addition. If a node is examined that has already been numbered, the graph below that node has already been traversed, so the result of that traversal is simply looked up.

This algorithm reports vectorizable statements and cycles as it finds them. C-style comments mark the place in the FIND_CYCLE procedure where this determination is made. As these statements are found, pointers to them are stored in a list. This list is used to guide subsequent transformations.

During the recursive decent of the dependence graph, the search will return to a particular node (statement) if a cycle exists or the search is returning through the search path. The second half of the FIND_CYCLE procedure differentiates between these two cases. If the search has returned to a node, and that node is not at the top of the stack, a cycle has been found, and the contents of the stack indicate the member statements of that cycle. In this compiler they are reported in the compiler trace information and discarded, but the contents of each cycle could be examined and a number of techniques such as loop peeling [Wol89], node splitting, or idiom recognition, could be applied in an effort to break them. Once broken, the statements in the cycle could be added to the list of vectorizable statements.

If the current statement is at the top of the stack when the search returns to the node, two cases are possible. One is that the statement is vectorizable, the other is that a self-dependence exists. The test for a dependence from the current node to the current node differentiates the two cases. Again, self-cycle statements are reported in the compiler trace and discarded, while vectorizable statements are entered into the list containing pointers to those expression trees.

Tarjan’s algorithm considers only the array-based dependencies within the code.
Figure 3.5: Tarjan's algorithm

There are a number of other constraints that will prevent a statement from being vectorized even though it is identified as vectorizable at this point. These constraints are used to remove entries from the list of vectorizable statements. The following is a list of the vector-inhibiting constraints:

**Non-constant Scalars:** Scalar references within a vectorizable statement prevent vectorization if the scalar is not loop invariant. Simple reductions, identified by a previous pass, are one exception to this, and statements annotated as such are not removed from the list of vectorizable statements by application of this constraint. See Section 3.4.3.2 for other exceptions.

**Some IFs:** Condition tests that are vectorizable but that contain no vectorizable
Statements in any of the controlled clauses are not vectorized to reduce the total work performed by the loop. See Section 3.4.2.1 for a full explanation.

**Functions:** Assuming all arguments to a function call meet the conditions for vectorizability, Tarjan’s algorithm will report the function call as vectorizable. Statements containing function calls are not vectorized (see Section 3.4.1.1).

**Cycle-limited Vectorizable Statements:** Vectorizable statements with dependence constraints that place their execution between two cycles or other non-vectorizable constructs are not vectorized. This is a ramification of the transformation model used for subsequent steps (see Section 3.4.2) and is not a restriction on vectorization in general.

Once these criteria have been applied to the list of vectorizable statements calculated by Tarjan’s algorithm, code transformation can begin.

Returning to Figure 3.4, loop (a) can not be vectorized, since it contains a dependence cycle, but the loop shown in (c) can be.

### 3.4.1.1 Simplifying Assumptions

A number of assumptions were made about the nature of the code being compiled. Because the SUIF infrastructure does not support inter-procedural analysis, these assumptions were needed in order to perform vectorizing transformations on a broader set of loops. Most of these assumptions could be weakened or eliminated entirely if a strong interprocedural dependence analyzer became available. Such an analyzer is currently in use by SUIF's developers [HAA+96], but has not been released.

**Pointers in Argument List:** Many functions containing vectorizable loops have the arrays they are working on passed to the function as arguments. In the absence of interprocedural analysis, the safest assumption that can be made is that these arrays overlap in some unknown way. This conservative assumption would prevent

---

4 *Controlled clause* is a general way of referring to any set of statements that are control dependent on a condition test.
assumed between all references to arrays passed as arguments to the function. The vectorizer assumes that array parameters always locate memory that can be safely accessed by the vectorized loops.

**Functions in Inner Loops:** Functions serve to hide potential memory accesses within a loop from the dependence analyzer since they may access global variables or treat pointer parameters as arrays. In this scenario it is possible for vector code executed before the function to have prematurely modified data needed by the function, or for the function to prematurely modify data needed by vector statements executed after it. Accesses of this kind would violate the serial semantics of the program. Under the most conservative assumptions any function call within an inner loop will prevent vectorization of the entire loop.

However, in many cases a function call will not perform such accesses. Any function that performs only scalar operations on its arguments, that accesses global variables that are not participating in the current vector loop, or that performs read-only accesses to locations that are also read-only in the vector portions of the loop may be executed within a vectorized loop body without violating program correctness.

The vectorizer assumes that all inner-loop function calls are of the second form, increasing the set of available loops to be vectorized at some risk. Before privatization, all functions are moved out of expressions and into their own statements. The return value is assigned to a scalar temporary, which is expanded into an array by the following passes. All functions are then run in scalar mode, with the expanded scalars communicating the returned values to the vectorized statements.

**Sparse Arrays:** In cases where data would sparsely populate an array, indirect array accesses of the form $a[b[i]]=\ldots$ are often used to reduce the total storage requirements of the program. During dependence analysis, the most conservative assumption would be that all references to $a[b[i]]$ map into the same element of the $a$ array. This would prevent the vectorization of even a single assignment to such an array, since no knowledge of the order in which an indexed vector memory access
this constraint, via a command line argument, permitting such accesses to be vectorized. If there are multiple references, including at least one write, to the indirectly accesses array, normal dependence analysis unconditionally prevents vectorization of the statements.

### 3.4.2 Partitioning and Strip mining

Once dependence analysis has been completed, and the set of all vectorizable statements has been found, loop transformation can occur. For the following discussion Figure 3.6 is taken as an example. The loop to be vectorized contains multiple statements, two of which are participating in a dependence cycle. This cycle is data-dependent on one of the remaining statements, while the last statement is data-dependent on the cycle. The dependence edge between the first and fourth statement cannot be ignored, but does not further constrain the scheduling of the loop.

The execution model used for vectorizable loops dictates the transformations required. Assuming that some dependence cycle or other non-vectorizable component exists in the loop, each vectorizable loop will be broken into three sections. The first section, referred to as the pre-scalar body, contains all vectorizable statements on which the non-vectorizable statements depend, or that are independent of the cycle. The second section contains all the statements that participate in the cycle or are otherwise not vectorizable. These statements will be executed in a scalar loop (scalar body). The remaining section contains all vectorizable statements that depend on the cycle (referred to as the post-scalar body). This is in contrast to an execution style which would permit multiple scalar loops to appear in the code sequence.

The single scalar body model was chosen for two important reasons. Having only one scalar section limits the amount of scalar overhead present within the loop. It also simplifies determining the correct place in the code sequence to insert the scalar body. The drawback to this occurs in cases where dependence constraints force the scheduling of vectorizable statements between two non-vectorizable constructs. In this case the vectorizable statements must be run in the scalar loop as well.
vectorizable statements into statements that must be executed before the scalar body and those that must be executed after it. Statements that have no ordering constraint with respect to the scalar body are placed in the pre-scalar body partition. These two lists contain the vectorizable statements in arbitrary order. A topological sort based on the dependence arcs must be performed in order to determine the correct execution order for each vectorizable partition. This sort ensures that all dependence constraints are met.

The sort algorithm is very simple. Statements are moved from the unordered list of statements to the ordered list whenever one is found that has no incoming dependence arcs from other statements in the unordered list. This ensures that all values on which a statement depends have been calculated (or have not been overwritten) when the statement is executed. Since all dependence cycles have been removed from the list previously, this approach is guaranteed to find a correct execution order. As a performance consideration, the sorted list is post-processed to shorten the live ranges of compiler-introduced array temporaries.

It is at this point that the only architecturally dependent assumption is made. The loop must be strip mined [BGS93] into pieces that fit in the register file of the target machine. This compiler assumes that the target machine will have a maximum vector length as defined by a command line argument. This constant value can be used in subsequent passes to perform scalar optimizations on many of the control constructs introduced by the strip mining transformation. Leaving $VL_{MAX}$ as a symbolic value would remove this architectural assumption, but scalar optimizations would be less effective, and compiler-temporaries would have to be dynamically allocated. Code created by this compiler will not function on machines with vector registers smaller than the specified $VL_{MAX}$, but would perform correctly (though not as efficiently as correctly sized code) on machines with longer registers.

Two approaches could be taken when strip mining. The two vector sections could be strip mined separately, producing three disjoint loops. This would increase scalar overhead and memory usage as the three loops communicated with each other.
Figure 3.6: Vectorization of a simple loop. Part (a) shows the original loop. Part (b) shows the statement-level dependence graph. Part (c) shows the logical result of vectorization. Part (d) shows the result of vectorization after strip mining.
The second approach is to create an outer strip mined loop containing all three sections and modifying the scalar loop to iterate only over the current strip. This approach allows for compiler-introduced vector temporaries to be only the maximum vector length long, minimizing memory requirements and access time since the entire temporary can remain in a register in many cases. This compiler takes the second approach.

The strip mined loop is built by using the pre- and post-scalar statement lists to extract statements from the statement list of the original loop and move them into the body of a strip-control loop. If any statements remain to be executed in a scalar fashion, the scalar loop is also inserted into the strip-control loop, otherwise it is discarded. As each statement is moved into the strip mined loop it is marked with an annotation indicating that it has been vectorized. The annotation appears only at the top of the expression tree. A subsequent pass propagates this down through the expression tree, to each node which performs an operation on vectors.

Loop headers and array subscripts are also modified at this time to be consistent with the new execution order and the scalar overhead of strip-size calculation is added.

Figure 3.6(d) shows the results of strip mining. First the number of iterations performed by the original scalar loop are calculated and stored in iterations3. The current vector length, $VL_2$, is then set to the minimum of this value and the maximum vector length of the target machine. The `set_vl_reg()` function call is a place holder for the instruction sequence needed to set the vector length register of the target machine; it does not result in the performance of a function call in the assembly code version of the program. The strip mined outer loop steps with an index $VL_{MAX}$ times greater than the original scalar loop while the inner loop iterates with the original increment over the current strip range. Once the strip's work is completed, iterations3 is updated to reflect the number of iterations remaining and the vector length register is reset if required.

Performing the “short” iteration first, followed by some number of iterations at the full vector length would reduce scalar overhead within the strip mined loop,
interfere with a subsequent pass, so the slightly less efficient implementation was used.

Two additional transformations are performed at this time. In the absence of a scalar loop body, the original loop index variable would not contain the same value it would have under scalar execution, so an assignment to the variable is added to the end of the loop. Also, compiler temporaries are limited to arrays of $VL_{MAX}$ elements long. Accesses to these values within the scalar loop body, such as would occur if a function call appeared in the loop, would require complex index calculation to map the value of the scalar loop's index variable into the correct range. An extra induction variable, ranging from 0 to $VL_{MAX} - 1$ is added to the scalar loop to remove the need for these calculations. They have been removed for clarity in this example.

3.4.2.1 Conditionals Within Vectorizable Loops

Many loops contain conditionals: structured control flow where a condition test controls the execution of sets of statements or clauses. Because Tarjan's algorithm (Figure 3.5) does not consider the type of dependence relating two nodes in the graph, simply adding control dependence edges to the dependence graph—a zero-distance edge going from the condition test to each statement controlled by it— is sufficient for testing for the vectorizability of the construct. However, some mechanism must be used for communicating the control information from the test to the controlled clauses if they are being executed on vector hardware. If-conversion [Wol89] is the process whereby control information is translated into data. Conceptually, if-conversion replaces the test-and-branch that begins a if construct with an assignment to a temporary variable. Each statement controlled by the branch then uses the variable as an extra operand, performing its operations only if the variable holds a true value (or false if the statement was in the complementary clause of the if). This provides the basis of vectorizing structured control flow.

As can be seen in Figure 3.7(a) and (b), the condition test and control-flow is replaced with an assignment of the test result to an array temporary cv_then6 and
Figure 3.7: If Conversion: parts (a) and (b) show the transformation of a simple conditional operation. Parts (c) and (d) show a nested control flow example. The masking of vectors with 1 in part (d) ensure that logical semantics are maintained when bit-wise operations are applied to 32-bit representations of logical values.
statement is annotated with the symbol table entry for the cv_then6 array, indicating the condition vector that controls it. Use of the condition vector is dependent on the architecture of the target processor and is not addressed in these high level transformations.

Figures 3.7(c) and (d) show a more complex nested control flow example. The then and else clauses of both condition tests are populated, so two condition vectors are generated for each test, with the vectors for the nested test masked by the appropriate vector from the outer condition test. The additions of the bitwise maskings of “1” ensure that logical semantics are maintained with the complement bitwise operation. By explicitly calculating the condition vectors in this manner the compiler (potentially) reduces the live ranges of the condition vectors and speeds the use of these vectors at the cost of extra array temporaries. See Section 3.6.3 for a discussion on how these vectors are used.

Partial vectorization causes special problems. When some statements participating in the if can be run in vector mode but the rest must be run in the scalar loop some mechanism must be used to communicate control information into or out of the scalar loop body or redundant calculation must be performed. Since the scalar body would be required to perform condition tests to use any test results developed in the vector body, it was decided to have the scalar body perform redundant calculation. If the condition test is vectorizable and at least one controlled statement is also vectorizable, the condition test is evaluated in the vector body for use by the statements in the vector body. The same condition testing is performed within the scalar body for use by the scalar statements. In the case where the condition test is vectorizable but none of the controlled statements are, the condition test is only evaluated in the scalar body. In the reverse case, where the test is non-vectorizable but the controlled statements are, the controlled statements are not vectorized.

Switch statements are another form of structured control flow, which could be implemented in a manner similar to if-style control flow. A condition vector could be generated for each clause in the construct, then used to control the execution of
amount of work performed by the vectorized loop, as compared to scalar execution, grows quickly with the number of clauses in the switch statement and the number of registers needed to hold a unique condition vector for each clause would quickly exceed those available for use.

This compiler does not support switch-based control flow at this time.

3.4.3 Expansion of applicability

There are a number of additional analyses and transformations which the compiler performs during vectorization to expand the set of loops that can be vectorized.

3.4.3.1 Non-structured Control Flow

It has been shown in Section 3.4.2.1 that structured control flow can be handled in a straightforward manner. Non-structured control flow, introduced by break, continue and return statements, can also be handled.

Continue statements result in the return of control to the top of the loop being executed. This could be viewed as all statements in the loop appearing after the continue being control-dependent on the condition test controlling the continue statement. The number of loop iterations performed is not changed. This situation is best handled before vectorization by transforming the code to make this control flow structured, then allowing the normal control flow mechanisms to work. This restructuring is not part of the SUIF distribution and was not implemented in this thesis.

Return and break statements both cause premature exit from the loop. For purposes of vectorization they are identical, and throughout the discussion break will be used to refer to both.

Figure 3.8 shows a simple loop containing a break before and after vectorization. The original loop calculates a quotient and exits the loop if a divide-by-zero would be performed. The assignment within the break clause is arbitrary and simply represents some amount of cleanup work needed when the break is taken. The assignment
introduces a dependence constraint that would inhibit vectorization if it were included in the graph.

Breaks can help to simplify the dependence analysis portion of vectorization. Any code within the clause containing the break can be viewed as appearing outside the loop, since it is only executed once. As such it does not need to participate in the dependence graph. In this case this permits vectorization of the loop.

The loop is then scheduled as if the break condition does not exist, with the test

Figure 3.8: Breaks in loops: (a) the original loop, (b) the transformed loop.
are added to the break clause to allow correct vector execution.

At run time, the pre-scalar body is speculatively executed at full vector length, with the provision that no writes are performed. The break is evaluated for the current strip within the scalar body. If no break occurs, the outstanding writes must be performed, and the post-scalar body executed, at full vector length. If the break condition is met within the scalar body, a number of flags are set, and vector lengths recorded, to allow only the writes that would have been performed during scalar execution of the program to be executed.

An important observation of the scalar loop is that statements that appear before the break in the original code are executed one more time than statements which appear after it. This requires that two vector lengths are known, one for pre-break statements and one for post-break statements. These lengths are recorded in the break clause within the scalar body. The deferred writes and all statements appearing after the scalar body execute with the vector length appropriate to their position in the original code. Which length to use is determined by examining the two flags set within the break clause, one, break_taken8, is used as a fast test to determine that some break has been taken, while the other, this_break_taken9, is used to determine which of (possibly) multiple breaks in the loop have been taken.

For the current implementation, the vectorizer relies on the code generator to use this information to manage the deferring of stores and to insert vector length modifying instructions, including the test to see which break was taken at each vector instruction after the scalar body. This scheme adds considerable complexity to code generation and would slow down the code considerably even in the case of the break not being taken.

During the implementation of the code generator, it was decided a better strategy would be to emit a separate post-scalar-body clause for each break and one for the no break taken case in the loop and perform condition tests once to decide on the clause to execute. This alternate approach has not been implemented.
{ 
a[i]=b[i] + i;
index_ar5[i] = i;
}
iterations3 = 100;
VL2 = 32;
set_vl_reg(VL2);
for(Vindx1=0; Vindx1<100; Vindx1+=32)
{
a[Vindx1] = b[Vindx1] + index_ar5[Vindx1];
index_ar5[Vindx1] = 32 + index_ar5[Vindx1];
iterations3 = iterations3 - 32;
if(iterations3 < 32)
{
    VL2 = iterations3;
    set_vl_reg(VL2);
}
}

Figure 3.9: Index vector transformation showing the extra initialization loop and the index vector update.

3.4.3.2 Index Vector

A common programming construct is shown in Figure 3.9 (a): the index variable of the loop is used for both array index calculation and as a value in the right hand side of a calculation. Normally scalar variables which have non-constant values will prevent vectorization of the statement. However, as the transformed code shows, an index vector can be generated before the vector loop and be used to replace all references to the original loop index variable outside array index calculations. Assuming this vector resides in a vector register for the duration of the loop, a single vector instruction is needed at the end of each iteration to prepare the vector for the next iteration.

The induction variable elimination pass, run before vectorization, replaces redundant induction variables within the loop with expressions in the loop index, increasing the applicability of this transformation.

3.5 Post Vectorization

Once vectorization has been accomplished, a number of additional transformations must be run before code generation can begin, including a variety of scalar optimizations and a set of passes dedicated to exposing array access patterns to the code
A program statements which has been vectorized is marked with an annotation at the root of the expression tree representing it. For subsequent passes, and code generation in particular, it is useful to have each vectorizable instruction within the statement marked as such. The SUIF pass containing most of the scalar optimizations perform in a compiler, porky, was modified to propagate the vectorization annotations to each appropriate node in the tree. This is run before all other post-vectorization operations.

### 3.5.1 Pointers and Strides

In general, vector memory instructions require three arguments: the source/destination register, the base address of the access, and the *stride* of the access through memory. Each vector memory instruction accesses multiple data elements in memory: the first is located at the base address; the stride is the offset used to locate each subsequent element in memory. A second kind of memory operation, *scatter-gather*, uses a vector of addresses to load or store its elements.

Reducing scalar overhead within a loop is always desirable. A pass developed by Sanjay Pujare[Puj95] converts array references to induction variables, initialized outside the loop and updated at the end of the loop body. This pass was modified to also insert code to calculate the stride needed for each memory access.

The result of this transformation is shown in Figure 3.10. The original loop contains three array references: the unit stride write to a[i]; the stride-two, offset read from a[2*i+5]; and the strided read of unknown distance from b[s*i]. Part (c) shows the results of the transformation applied to the vectorized code. The three references have been replaced with pointer accesses, and the stride calculations, which give the stride in bytes, can be seen in the loop preamble. SUIF annotations, which have no representation in C, connect the stride values to their respective loads and stores.

A number of simple scalar optimizations are run following this pass to improve the code emitted by this pass and the vectorizer. Dead code elimination and constant
Figure 3.10: Induction variable transformation: (a) original code (b) vectorized loop (c) after index and stride generation.

Propagation where made aware of vectorized statements and their use of stride values attached to memory operations with annotations. For most memory references the stride will be a known integer constant, thus these optimizations can completely eliminate the calculation of the stride values and the need for registers to store the stride values.
In the context of this compiler, code generation is the process of changing the SUIF intermediate representation into assembly code for the use of a native assembler and linker, rather than direct translation into an object file. An extension to SUIF, \textit{machSUIF} [Smi96] is used to perform this mapping.

Up to this point in the compilation sequence, the only assumption about the target architecture has been that it has a particular maximum vector length\footnote{This assumption could be lifted if the run-time environment or code generator where to supply the value.}. A code generator for any vector architecture fitting these loose assumptions could be inserted at this point without modification to previous steps.

### 3.6.1 MachSUIF

MachSUIF is aimed at providing a framework for machine-dependent optimizations. It provides an intermediate representation related to SUIF's, but is aimed at capturing low level machine details, rather than high-level program constructs. MachSUIF, like SUIF, can be partitioned into two sections: the support library and a set of passes that make use of the library. At the present time the only passes available with the machSUIF distribution are code generators, which map the SUIF intermediate representation of the program to the machSUIF representation, and a machine language printer, for translating the machSUIF intermediate form to ASCII assembly language files suitable for the target machine's native assembler. Any number of optimization and scheduling passes could be inserted between code generation and ASCII file generation, but none were implemented in this thesis.

The target machine for this thesis was the Berkeley SPERT board containing the T0 vector microprocessor (see section 2.1.1.1). The T0 contains a MIPS-II scalar core with a vector co-processor implemented as co-processor two. MachSUIF library support for the MIPS architecture was extended to include T0-specific vector instructions, and the existing MIPS code generator, MGEN, was augmented to utilize these
needed to support a full range of (integer) vector operations.

3.6.2 General

Translation of the SUIF representation of the program into the machSUIF representation is straightforward. SUIF represents the program as a list of expression trees, with the nodes of each expression tree made up of symbol table entries and RISC-like instructions. Because previous SUIF passes have propagated all information needed for code generation to the nodes at which that information is needed, the appropriate instruction sequence is determined solely by a recursive post-order decent down each expression tree. Within the SUIF program data structure, vector instructions are identified by simple annotations; anything not marked as a vector operation is a scalar operation. Within machSUIF the opcodes and extensions differentiate between different versions of a an operation type.

The only complications arise when a machine-dependent construct, such as control-flow and reductions, or hardware limitations are encountered, or when register allocation for performance is added.

3.6.3 Conditionals

Generation of condition vectors is an architecture-independent mechanism for handling conditionals in inner loops. The way that condition vectors are used by controlled statements, however, is dependent on the target machine architecture.

In controlled-store machines, memory operations, and often all operations, are masked by a condition vector. Only the subset of elements needed for the current strip are loaded, operated upon and stored. This has the advantage of having no operation performed during vector execution that would not be performed under scalar execution, but complicates the memory system if more than one element is to be written from a vector to memory at the same time, as is the case for T0-style unit stride operations.
In the case of the Torrent, only conditional moves between registers are provided. This facilitates executing both the then and else clauses of a conditional and masking them together. This method works well for the assembly language programmer, but requires some effort on the part of the compiler to exploit.

In the most general case, the controlled execution of $a[i] = b[i] + c[i]$ involves four accesses to memory. First the two source arrays are loaded from memory and the expression is evaluated. All locations in the $a$ array, which are possibly written, must then be loaded into a register. The result of the RHS evaluation is conditionally moved over this loaded data under control of the condition vector and the entire vector is written out into memory. This mechanism protects the integrity of memory but, depending on the dynamic behaviour of the control construct, can be extremely wasteful. In the worst case, where none of the condition vector elements are true, none of the $4 \times VL_{MAX}$ memory operations would have been performed during scalar execution.

Assuming a complementary clause exists and the results are merged before being written to memory, slightly more than twice as much work was done for vector execution than was done in the scalar case.

Within the code generator, programmer-specified array accesses are all treated in the general manner outlined above. The register allocation scheme described in Section 3.6.5 attempts to eliminate redundant accesses to privatized scalar variables and other compiler temporaries, completely eliminating unnecessary memory accesses if enough registers are available.

Calculations of the RHS values are speculative, since the condition vector is used only to mask results, not to inhibit the operations. Any exceptions, such as a division by zero or a overflow, must be qualified by the condition vector before they are processed. The question of exceptions is not addressed in this implementation.

### 3.6.4 Reductions

The mechanism used for executing simple reductions of the form $s = s + a[i]$ is also dependent on the target architecture. Some machines, such as those in the HITACHI
Figure 3.11: A simple summation reduction of a 16 element vector using recursive doubling. Step 1 splits the source vector into two equal sections, which are added in step 2. This cycle is repeated in subsequent steps. Only two vector registers are used.

line, support reductions in hardware, while others, such as the Torrent must execute a sequence of vector and scalar operations to perform the reduction. The code generator produces code that performs a "recursive doubling" [KS73] evaluation of the vector as depicted in Figure 3.11. First the vector is padded out to full length with the identity value for the operation. It is then partitioned into two equal sections, as shown in Figure 3.11 and the reduction operator is applied to these two vectors with half the full vector length. This process is repeated with the length of the operation divided by two in each iteration until the operation at vector length 1 is performed. A final scalar operation combines the vector-reduced value with the scalar reduction variable.

The padding to maximum vector length is performed unconditionally. Varying the execution based on vector length would be possible, but comparing the cost of control flow with the cost of simple full-length executions shows no benefit.
A good register allocation scheme is important to obtain good performance from any architecture. A simple scheme is used in the code generator which eliminates most memory references to compiler temporaries such as privatized arrays, condition vectors, and the index vector if it is present, and reduces the amount of code needed to manage spilled variables. Allocation does not attempt to reduce memory accesses to programmer-specified arrays. This optimizes for a certain style of source code, where arrays are accessed only once or are reused through references to a scalar temporary.

Ideally a temporary will never need to be written to memory, but two conditions will force it to be. If at the time of calculation there are not enough registers available to give the values a register home for their lifetime, the values must be spilled. Privatized arrays that are referenced within the scalar body also need to be written out to memory to communicate the values to the scalar code correctly.

Before code generation begins, the program is traversed and the live ranges of all compiler temporaries are determined. Each definition and use point receives an annotation indicating that the values should be held in a register if possible. The last use of the temporary has no annotations. If the live range of the temporary crosses into the scalar body, with at least one use before the scalar body, a second annotation is added at the definition point, to indicate that the values must be written to memory. Forcing the write at the definition sight ensure that data is available to scalar memory accesses within the scalar body without the need for explicit spill code to be produced before the scalar body is entered. In the special case of a definition occurring before the scalar body, but all uses occurring after the scalar body, no annotations are made at the definition point (the default bahaviour produces a write of the value to memory), but uses will still receive any annotations that are appropriate.

During code generation, when the first store of an array temporary is encountered, the annotations are read. If a store is required by the annotations, it is performed. If the hint to keep the values in registers is found, and enough registers are available, the register holding the values is reserved and no memory operation is performed (unless forced by an annotation). If there are not enough registers, the store is performed
regardless of the annotations and the source register is returned to the free register pool.

Each time a load is required, a look-up is performed; the load is not emitted if the required values already have a register home. If a load was required to retrieve the temporary, but a hint to keep the value in register is found at the use point, and registers are available, the values remain in register, otherwise the register holding the values is returned to the free pool after use. It does not have to be rewritten to memory.

The index vector is also treated in a special way. The loop which generates the initial values (see Figure 3.9) writes directly into a vector register, guaranteeing that these values have a register home for the duration of the vectorized loop. Accesses to these values never require access to memory.

Programmer-defined arrays *always* have the memory copy as the only valid copy of the data.

### 3.6.6 T0 Quirks

The current implementation of the Torrent ISA, the T0[AB95], has some limitations with respect to general codes (see Section 2.1.1.1). It was aimed at the neural network research area, where short data-type, fixed-point calculations dominate. As such, and to reduce design complexity and die size, only a 16-bit fixed-point multiplier is supplied beyond the basic math and logical operations. The code generator recognizes 32-bit multiplications and emits emulation code to perform these operations at full precision using the available vector hardware.

Figure 3.12 shows the algorithm used to implement the multiply. The high- and low-order half-words are shifted and multiplied separately, then added together to obtain the full precision result. Both signed and unsigned multiplications are supported.

Integer division and modulus, and floating point operations, which are not supported in the T0’s vector unit, could be supplied in a similar manner but were not implemented. Compilation halts with an error message when such constructs are
Figure 3.12: Algorithm for 2n-bit multiplication with half precision (i.e. n-bit) multiplier. The prime and double-prime components are the high- and low-order n bits of the multiplicands respectively. For signed operations, an extra bit must be used to hold the sign of the operand. Shift by X shifts by $X \times n$ bits.

3.6.7 Unsupported Transforms

Several constructs are supported in the vectorizer but are not yet supported by code generation.

**Scatter/Gather Memory Operations:** Scatter/gather memory operations of the form $a[b[i]]=\ldots$ are currently (optionally) permitted by the vectorizer so long as no other references to that array are made. The passes dedicated to determining the base address and stride values of vector memory accesses are unable to process this style of access effectively, preventing the generation of assembly code for references of this form at this time.

**Conditional Reductions:** Conditional reductions of the form if($a[i]>0$) $s=s+a[i]$ are also allowed by the vectorizer, but reduction support does not currently include this construct. Masking the identity value for the reduction operator onto the reduction data based on the condition vector then reducing normally is the simplest mechanism for supporting this construct.

**Breaks:** Support for breaks and returns is more easily and efficiently implemented in the vectorizer than in code generation. Support for the code generator solution proposed in Section 3.4.3.1 has not been implemented.
Once MGEN has mapped the SUIF program representation into the machSUIF representation a number of machine dependent optimizations could be applied to it. At this time no machSUIF optimization passes are available, so compilation proceeds directly to assembly code generation. The machSUIF pass printmachine is used to produce the file.

The SUIF vectorizing compiler emits assembly language programs suitable for compilation with the target machine's native assembler. In the case of the T0, a full set of GNU-based tools are supplied, including a linker, debugger, assembler and GCC\textsuperscript{6}. These tools are used to assemble the SUIF file directly into an executable or to link a set of vectorized functions created by the SUIF vectorizing compiler with C source files containing driver routines.

### 3.7 Summary

This chapter has presented the compiler and code generation framework developed in this thesis. A wide variety of programming constructs have been shown to be supported, and importantly, the goal of architectural independence has been met. In the interest of simplicity in this implementation, a fixed maximum vector length was assumed, but this is not overly restrictive and can easily be eliminated.

Code generation for vector machines has been shown to be a straightforward process even when the architecturally specific transformations have been deferred to that time. There is, however, room for improvement.

Improved register allocation and could be implemented to improve cases where there is reuse of programmer-defined arrays. The supplied register allocation mechanism does however handle the common cases of privatized arrays and condition vectors effectively.

\textsuperscript{6}SPERT0-GCC is not a vectorizing compiler, it is a cross compiler for scalar programs to be run on the SPERT daughter card. A library of hand-coded vector functions is provided (see section 2.1.1.2).
to expose vector parallelism in a set of vector compiler test loops and the performance of a number of multimedia and filter loops when compiler with the SUIF vectorizing compiler.
Chapter 4

Experimental Results

This chapter presents a discussion of the mechanisms used to verify the correctness of the compiler presented in Chapter 3, followed by the results of applying that compiler to a number of benchmarks. First the effectiveness and robustness of the compiler are demonstrated by explaining its performance on the Argonne National Laboratory (ANL) vectorizable loop benchmark suite. Then actual performance results are presented.

Performance of a number of simple loops, multimedia kernels, and other “general purpose” kernels is evaluated for a variety of vector hardware configurations simulated with a variable configuration vector hardware timing simulator, Viola, which was developed for this thesis. These results are compared to a set of superscalar hardware configurations simulated with Cello, a reconfigurable superscalar simulator.

4.1 Correctness and Verification

The SUIF vectorizing compiler was extensively tested during its development. While developing the vectorization pass, facilities for translating SUIF files into text representations of the data structures and C-language representations were used to verify the correctness of transformations by inspection. The results of vector transformation produce legal C syntax from the SUIF-to-C converter, but the resultant code does not execute correctly when compiled with a scalar compiler, so comparison of
After the implementation of code generation, testing procedures could be automated. All benchmarks are augmented with optional I/O calls. Parallel compilation of a loop, using both a scalar compiler and the SUIF vectorizing compiler, allowed the comparison of the results of the two executions. Several programs were made self-diagnostic: they report the correctness of execution rather than emitting result data.

Loops of increasing complexity were compiled and tested in this manner to minimize the number of cases being tested by a particular loop, while ensuring that all features were tested. At this time the SUIF vectorizing compiler is a stable tool, capable of handling loop nests of arbitrary depth and control complexity.

## 4.2 Effectiveness Relative to Other Compilers

The ANL benchmark suite [LCD92] was used to test the versatility of this compiler. This is a set of 122 FORTRAN loops taken from a number of scientific and linear algebra codes, and other loops supplied by hardware vendors. These loops contain a wide variety of constructs, all of which are at least partially vectorizable, and are used to test the transformation and analysis abilities of a compiler and the quality of the code produced. These loops are not an exhaustive test of a compilers abilities but give a strong indication as to where its strengths and weaknesses may lie.

Table 4.1 shows the results of compiling the suite with the SUIF vectorizing compiler and with the IBM 3090-600J, CCC CRAY-2, and CONVEX C210 compilers. The IBM and CRAY compilers were chosen because they represent the best and the worst data points for the set of compilers presented in [LCD92]. The CONVEX was chosen as a intermediate point. All data for the three commercial compilers is derived from [LCD92] and is as of 1992.

Compilers are scored on their ability to vectorize each loop. Loops are considered fully vectorized if all statements within the original loop are vectorized, while partially vectorized loops need only perform one vector operation in the execution of the test
Some compilers conditionally vectorize a loop: determining whether or not to execute a vectorized loop body, or at which vector length to execute the body, at run time. Conditionally vectorized loops are considered fully vectorized if the vector loop was executed or if vector lengths greater than one were used at run-time. Loops which perform calls to optimized libraries, such as for handling reduction or recurrence constructs or calls to vectorized library functions, are also considered fully vectorizable with respect to that construct. Finally, some compilers do not vectorize loops that are partially vectorizable if a cost-benefit heuristic decides that no performance benefit will be attained by partially vectorizing the loop. In cases where the compiler output has reported that a candidate for partial vectorization has not been vectorized for this reason, the loop is considered partially vectorized.

A number of control constructs in these loops use the GOTO statement. A restructuring pass is run within the SUIF vectorizing compiler to transform this “spaghetti code” into structured control flow. Any loops which retain non-structured control flow after this pass are deemed non-vectorizable and are discarded. The parenthesized number in the column showing the number of loops in each category indicates the number of loops for which vectorization was actually attempted by the SUIF compiler based on this constraint. It has no bearing on results for the other three compilers.

SUIF compilers support FORTRAN through an intermediate stage of FORTRAN-to-C conversion. This mapping has the disadvantage of transforming any multi-dimensional arrays found in the FORTRAN source into single dimensional C arrays, with the sizes of the dimensions unresolved until run-time. In some cases this change complicates dependence analysis to the point where vectorization cannot be performed (see Figure 3.4 (a) for a similar case\(^1\)). For these cases, the loop was manually translated into C and compiled separately. Parenthesized values in the SUIF results

---

\(^1\)The \( t \) variable is representative of all calculations involving the higher dimensions of the multi-dimensional array. The sign of this variable is not known, so dependence vectors reflecting both signs must be assumed.
<table>
<thead>
<tr>
<th>Dependency Analysis</th>
<th>full</th>
<th>partial</th>
<th>full</th>
<th>partial</th>
<th>full</th>
<th>partial</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear dependence testing</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>induction variable recognition</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>global data flow analysis</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nonlinear dependence testing</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>interprocedural data flow analysis</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>control flow</td>
<td>2(1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>symbolics</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Section Totals</td>
<td>29</td>
<td>25</td>
<td>0</td>
<td>19</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vectorization</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>statement reordering</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>loop distribution</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>loop interchange</td>
<td>5(4)</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>node splitting</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>scalar and array expansion</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>scalar renaming</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>control flow</td>
<td>12(9)</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>crossing thresholds</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>loop peeling</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>diagonals</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>wavefronts</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Section Totals</td>
<td>41</td>
<td>31</td>
<td>5</td>
<td>30</td>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idiom</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction</td>
<td>13</td>
<td>8</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>recurrences</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>search loops</td>
<td>2(1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>packing</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>loop rerolling</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Section Totals</td>
<td>24</td>
<td>14</td>
<td>0</td>
<td>16</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language Completeness</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loop recognition</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>storage class equivalence</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>parameters</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>non-logical if's</td>
<td>3(0)</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>intrinsic functions</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>call statements</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>non-local gotos</td>
<td>2(0)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>indirect addressing</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>statement functions</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Section Totals</td>
<td>28</td>
<td>25</td>
<td>1</td>
<td>18</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>TOTALS</td>
<td>122</td>
<td>95</td>
<td>6</td>
<td>83</td>
<td>13</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of ANL loops for this and a number of other commercial compilers.

column show the results when these loops are included in the totals.

The suite can be divided into 4 major categories:

1. Program analysis: loops in this section tests the analysis and transformation abilities of the compiler in the presence of both array and scalar dependence
Linear Dependence Testing: This section tests the dependence analysis capabilities of a compiler. Cycle breaking and other loop transformations are required to vectorize all loops in this section.

Induction Variable Recognition: Tests the ability of the compiler to detect and replace redundant induction variables within a loop.

Global Data-Flow Analysis: Tests the ability to propagate information through-out a procedure to improve dependence analysis.

Nonlinear Dependence Testing: Tests for the ability to resolve non-linear dependencies.

Interprocedural Data-Flow Analysis: Tests the ability of the compiler to use information from a functions call sight to improve dependence analysis of the function. Both inter-procedural propagation and in-lining are valid options.

Control Flow: Loops in this section test the dependence analysers ability to qualify dependence vectors with control flow information from outside the loop.

Symbolics: Tests the ability to vectorize statements which contain symbolic values (function parameters) in loop bound expressions.

Linear dependence testing appears to be an area where the SUIF compiler does not perform well compared to the IBM and CONVEX compilers. The three loops which were not vectorized all contained dependence cycles that required some loop or statement transformation to be applied in order for vectorization to be performed.

Induction variable recognition within the SUIF vectorizing compiler depends on the detection pass distributed with SUIF, outlined in Section 3.3.1. This pass detects only simple induction variables, and many of the loops in this section
vectorizable. Any improvement made to this pass would directly influence the ability of the vectorizer to handle these loops.

Loops in which symbolic values affect the dependence relationships are difficult to vectorize because arcs must be included in the graph for each dependence that might exist. Both the CRAY and IBM compilers make use of some form of conditional vectorization to address these loops. This is a mechanism where a run-time test is generated to dynamically decide the way the code is to be executed. The IBM compiler prefers to select between a vectorized loop and a copy of the original scalar loop, while the CRAY calculates a safe vector length. The SUIF vectorizing compiler does not conditionally vectorize loops, but addition of this facility to the SUIF vectorizing compiler would permit many of these loops to be vectorized.

2. Vectorization: Loops in the section test the ability of the compiler to perform a variety of statement- and loop-level transformations to exploit vectorizable code.

**Statement Reordering:** Tests for ability to reorder statements to allow vectorization while maintaining dependence constraints.

**Loop Distribution:** Tests the compilers ability to introduce scalar bodies to handle non-vectorizable constructs (cycles and functions) in partially vectorizable loops.

**Loop Interchange:** Tests for ability to eliminate dependence cycles by interchanging loops.

**Node Splitting:** Tests for ability to break up individual statements in order to satisfy dependence constrains while vectorizing the loop.

**Scalar and Array Expansion:** Tests for the ability to recognize and exploit data structures which are not shared across instances of the loop body.
dependencies.

**Control Flow:** These loops test for the ability to handle structured control flow (i.e. if and switch-style constructs within the loop).

**Crossing Thresholds:** Tests for the ability to break a loop into multiple loops to remove dependence constraints in the presence of array references of the form \( a[i] = b[n-i] \).

**Loop Peeling:** Tests for the removal of \( n \) iterations from the start or finish of an iteration space to remove dependence contains.

**Diagonals** Tests for ability to vectorize accesses of the form \( a[i][i] \).

**Wavefronts** Tests for ability to restructure loops to perform diagonal accesses in order to remove dependence constraints.

The SUIF vectorizing compiler relies on the distribution code for detecting privatizable variables to perform array expansion of scalars. Three of these loops contain privatizable variables as defined in Section 3.3.2 and were vectorized to some extent. Of the remaining loops, three had scalars which carried information across loop iterations, and so were not vectorizable by the SUIF compiler. The remaining two required array expansion, an unsupported transform.

One of the control flow loops required loop interchange to be performed in order for vectorization to be possible. This transform is not supported, so the loop was rendered non-vectorizable. The other three loops implement switch-style control flow, which is not supported in the SUIF vectorizing compiler.

3. **Idiom Recognition:** Tests for a compilers ability to detect common program constructs which can have special vector implementations. Based on dependence testing alone code in this section (except for possibly loop rerolling) is not vectorizable. Better-than-scalar performance is possible if the *idiom* can be recognized and special purpose hardware or optimized library calls used.
Recurrences: Tests for the ability to vectorize special first- and second order recurrences which have can have logarithmically-faster software implementations or hardware support.

Search Loops: Search for occurrences of some value in a vector. Indices are recorded.

Packing: Tests for the ability to compress one vector into another under the control of a condition test (e.g. if( test ) b[j++]=a[i]).

Loop Rerolling: Tests for the ability to vectorize a loop which contains a fully unrolled inner loop.

Reductions are the only idiom currently exploited in the SUIF vectorizing compiler, but here it fails to perform up to expectations. All of the fully vectorized loops were summation reductions. Max/min reductions made up the remaining loops. Detection of these constructs relies on a special representation of the operation in the SUIF file using max and min instructions. Because these loops contained an explicit control-based representation of the operations, which could not be recognized as a reduction by the detector distributed with SUIF, the scalar variable dependencies introduced by the reduction could not be resolved. One max/min reduction loop, that calculates the maximum absolute value of a data set, was partially vectorized. The absolute value calculation was vectorized, while the reduction was left as scalar code.

One other loop saved the intermediate values of the reduction and was not vectorized.

The definition of “fully vectorized” within this test is important with respect to the loop rerolling set. These three loops are fully vectorizable in their supplied form, performing rerolling would only remove strided memory accesses, possibly improving performance. The unrolled loops are small, so machines with large vector startup times could experience performance degradation by re-rolling.
4. Language Completeness: Loops in the section test the compilers understanding of a number of FORTRAN constructs.

**Loop Recognition:** Loops in this section are implemented with explicit GOTOs. This tests the compilers ability to detect loops built in this manner.

**Storage Classes and Equivalence:** Tests for the compilers understanding of storage classes and equivalence.

**Parameters** Tests for vectorization of statements containing function parameters.

**Non-Logical IFs:** Support for vectorized multi-way branches and arithmetic if $s$ is tested.

**Intrinsic Functions:** Loops in this section contain intrinsic function calls.

**Call Statements:** Tests for vectorization of loops containing CALL statements.

**Non-Local GOTOs:** Test for RETURN and STOP construct support.

**Indirect Addressing:** Tests for scatter/gather support.

**Statement Functions:** Tests for recognition of STATEMENT functions.

The intrinsic function section contains three loops, one which contains calls to the sin and cos functions, while the other two loops contain only type-conversion operations, which are function calls in FORTRAN. The SUIF vectorizing compiler fully vectorized the type-conversion loops, since the function calls were removed when conversion to C occurred. It could only partially vectorize the loop containing trigonometric functions since no vectorized function library was created for this compiler. The other three compilers have such a library at their disposal, and so were able to vectorize the third loop completely.
called function had no body. Interprocedural analysis could remove the CALL from the loop without changing the result of the program.

FORTRAN-to-C conversion mapped all loops in the non-local GOTO category to GOTO-based control-flow in C. The SUIF vectorizing compiler did not attempt to vectorize these loops.

In addition to these 122 loops, thirteen control loop are also part of the suite. These should be vectorizable by all compilers, and are supplied to determine basic instruction rates for use in a performance comparison model proposed in [LCD92]. All four compilers fully vectorized these loops.

4.2.1 Summary

As can be seen from the summary lines in Table 4.1 the SUIF vectorizing compiler developed in this thesis appears to perform poorly in relation to the other three compilers. The SUIF compiler vectorizes 54% of the loops in the suite, while the best compiler, the 3090-600J vectorizes almost 83% of the available loops.

In many cases, a very specific solution, such as peeling or splitting of the loop, are the only mechanisms for vectorizing the loop, but some broader problems and weaknesses in SUIF support passes were also exposed. The use of stock SUIF passes at this time limits the applicability of this compiler, but as SUIF is refined and augmented the vectorizing extensions will become more effective without additional effort being spent on the vectorization pass itself. The framework of the vectorizer is such that additional loop transformations could easily be incorporated.

The ANL benchmark suite is a two-part test. This compiler has performed the code transformation portion, but at this time cannot proceed to the performance comparison section of the test. The suite's loops contain a number of constructs not yet supported through code-generation, notably indexed memory accesses. There is also a hardware limitation. The target machine, the T0 vector microprocessor, has no floating-point support. All benchmarks in this suite perform 64-bit floating
available, such a comparison could be performed.

### 4.3 Vector Performance

In addition to the ability of the compiler to locate and express the vector parallelism available in the code, the quality of the code produced must also be measured. To do this an extensive simulation environment was constructed and a number of benchmarks were run on both vector and superscalar hardware configurations. First a number of simple loops were run to show the behaviour on building-block cases. Then a number of important kernels from the MPEG-1 encoder and decoder [Gal91] were run to demonstrate effectiveness on multi-media workloads. Finally the effectiveness of the compiler on several miscellaneous loops, such as the PGP decryption kernel and the uuencode/decode pair, were examined.

#### 4.3.1 Simulation Environment

In order to compare the relative capabilities of the two architectures many different hardware configurations were simulated. This entailed the use of Cello, a reconfigurable superscalar simulator, and Viola, a reconfigurable vector simulator developed as part of this thesis. The simulation environment is shown in Figure 4.1.

As in the case of superscalar processors, vector processors demonstrate binary compatibility between ISA-compliant machines. All processor models used in this thesis execute the same object code. Re-compilation is not required despite the large difference between the smallest and largest models.

#### 4.3.1.1 Cello

Cello is a superscalar simulation package developed by Todd Mowry at the University of Toronto. It supports scalar processor configurations from simple in-order scalar machines up to wide out-of-order issue superscalars with aggressive scheduling and
Figure 4.1: Data flow diagram for the simulation environment developed in this thesis. The dotted arrow indicates an optional compilation scheme, used in some benchmarks, where only some components of a multi-file program are passed through the vectorizer.

branch prediction. A multi-level memory system is also simulated. Cello is a trace-driven simulator, whereby the simulator uses a pixie trace (or traces from several other sources) of the target executable to derive timing and utilization information based on a hardware model.

The parameters used for the five processors simulated in this thesis are detailed in Figure 4.2. The smallest processor, the 2I,8R model is representative of the smallest superscalars on the market. It contains a two-instruction, out-of-order instruction issue unit working with a eight-entry reorder buffer. The 4I,48R is representative of current technology: a 4-issue, 48-entry reorder buffer machine. Both of these machines have two memory units, which is the largest number of ports available on processors today. The large number of integer functional units was chosen so as not to present a resource limit. The 6I,NR machines are representative of large realistic superscalar processors. Three different reorder buffer sizes are simulated to examine the affects of ROB size on execution. Eight memory units were chosen to give these machines the same memory bandwidth as the T0 vector microprocessor.

In all machine models, a simple 2-bit counter branch prediction scheme was used with a 2K-entry table in the 6I machines, while the 2I,8R and 4I,48R machines contained only a 512K-entry table. The re-order buffer had access to a 256-entry physical register file (32 registers with 224 possible rename-registers).
Particularly, when loads and stores are both required to be performed, a load cannot be performed until the addresses of all previous stores have been resolved. This constraint can impose a huge penalty on the amount of parallelism that a superscalar processor can extract. To avoid this performance penalty, Cello was run such that loads could precede stores, with a 32-entry memory access buffer (see Section 2.3).

Programs to be run under Cello are first compiled with the SGI C compiler (version 5.3), then processed by pixie, a binary instrumentation tool. The instrumented version of the program supplies the execution trace used by Cello. The C compiler is run with the -02 flag set. This performs a number of optimizations on the code, including *loop unrolling*. Loop unrolling is a transformation which replicates the loop body \( n \) times, permitting \( n \) iterations of the original loop to be performed without the need for a loop-back test. An extra loop body is added to execute the up to \( n - 1 \) iterations which could not be executed in the unrolled loop. This transformation reduces the number of branches to be performed, and potentially increases the amount of parallelism which can be exploited. The SGI C compiler uses a heuristic to determine if this transformation is to be applied. Loops are unrolled by four, if it is determined to be worthwhile.

Another important optimization performed by the SGI compiler is global register allocation. This transformation reduces the number of memory-references within a procedure and minimizes the register-save and -restore overhead associated with a function call.

### 4.3.1.2 Viola

Viola is a cycle-accurate timing simulator for T0-like vector architectures developed as part of this thesis. The SPERT simulation package is the first component of this tool.

The SPERT development package, described in section 2.1.1.2 includes two simulators for the SPERT hardware. The first, an ISA simulator, executes all instructions at a rate of one per cycle. This is a useful tool for development because of its execu-
information. The RTL simulator provides cycle-accurate timing information for the T0 at the cost of simulation speed. Because neither of these simulators are capable of modeling different hardware configurations, a new simulation tool had to be developed to pursue the goals of this thesis.

The fast ISA simulator was used to develop a cycle-accurate, timing simulator capable of simulating a variety of vector hardware configurations. Viola uses the execution trace generated by the ISA simulator to model functional unit residence times and chaining latencies. This results in accurate timing information on a variety of vector-processor configurations.

Some hardware parameters are fixed because of the SPERT simulator being used. For example, both the maximum vector length and number of vector registers are fixed. However, the width and number of computational and memory functional units is fully configurable, as is the latencies on each instruction.

Several hardware limitations of the T0 (see 2.1.1.1) are also overcome. The T0 has unbalanced pipelines: only one of its two functional units contains a multiplier. This simulator assumes that all pipelines of the same category (vector math, memory, scalar) are identical, so multiplications can be issued to any available computational unit. The simulator can also optionally model hardware containing full 32-bit integer multiplication (despite the program being simulated containing emulation code) and a memory system that supports strided access to memory at the same rate as unit stride accesses. This last feature assumes a large number of address ports to memory, but is the complementary assumption to one made for the superscalar models. In the superscalar case, it was assumed that the peak memory bandwidth matched that of the T0, requiring the superscalar to have eight address ports. For the T0 this peak bandwidth, of eight elements per cycle, is attained for half-word (or byte) unit-stride memory accesses, needing only one address port. To balance the two machines, the enhanced addressing capability was added to the simulator. Viola further assumes that memory access time is independent of the size of the element (word, half word or byte).
Functional unit latency is the time between an instruction being issued to a vector pipeline, and the first set of results being available at the end of the functional unit. Elements appear at the end of the pipe at a rate of \( \text{pipewidth/cycle} \) after this time until \( VLR \) results have been produced. Chaining latency is the time required for a set of results at the end of a functional unit to propagate to the start of another functional unit. Because of the possible size mismatch between computational and memory functional-unit widths, chaining takes place through the vector register file. In the case of the T0, functional unit latency is two cycles, with no chaining latency\(^2\). All simulator models have a chaining latency of one cycle.

The machine configurations used for this study are shown in Figure 4.2. All machines are assumed to have one scalar unit and two integer vector pipelines. Functional unit width indicates the number of parallel pipes in each functional unit of that type. Depending on the operating mode of the simulator, either one address port per memory unit or one address port per memory pipe will be simulated. Pipe widths range from very small machines, 2 or 4 wide, representative of small, traditional vector processors or new multi-media extended instruction set machines, to larger models that reflect more aggressive traditional vector processor configurations. For each machine type, two different sets of functional-unit latencies were assumed.

### 4.3.2 Memory Systems

The memory systems of the superscalar and vector processor models were kept as similar as possible. It was assumed that an ideal memory system was being used, and all memory accesses experience only functional unit latency. Non-interfering instruction fetches were also assumed.

Normally I-cache fills would take place through one of the available memory units, degrading memory performance. In the case of superscalar models the I-cache fill time was set to zero, while in the vector case a separate instruction memory with its own "instruction port" was used: instructions were presented to the simulator from

\(^2\)The T0 writes the register file in the first half of a cycle and reads it during the second half.
### Figure 4.2: Simulation environment parameters for the Cello and Viola hardware models.

outside the memory system.

## 4.4 Performance Comparisons

This section presents the performance results for a number of simple loops and kernel loops from multimedia and filter applications. Due to differences in the superscalar and vector simulators, the loops were run on different data sizes in some cases. The SPERT simulator could isolate single functions, so results are based on a single call
Cello reports results for whole-program execution, so for superscalar simulations data sizes were increased, or functions called multiple times, to amortize program start overhead over the work being performed in the benchmark. Sizes were chosen so over 99% of execution time was spent in the function(s) to be measured.

The benchmarks are summarized in Table 4.2. Details of each benchmark are presented with its performance results. In cases where source code was modified, the modified code was run in both the vector and superscalar simulators.

First simulation results are presented for a number of simple vectorizable loops, followed by kernels from both MPEG encoding and decoding, and several other kernels.

Performance graphs in this chapter show normalized execution times, in cycles, for the basic unit of work in the loop under test. In most cases this is cycles per loop iteration, but larger units are used in some cases. An example graph can be found in Figure 4.4. The graphs present superscalar results at the left of the graph, with four bars for each vector machine model on the right. For the superscalar bars, gradations in the bars indicate the percentage of time at various issue rates, the lowest section indicating the time at maximum issue rate. For the vector simulations, the lower section of each bar indicates time spent with at least one vector unit busy, while the top bar indicates time spent with only the scalar unit active or a scalar memory operation in progress. The left pair of bars in each vector machine category provide results for machines with unit latencies on all operations, this closely reflects the behaviour of the T0. The right pair of bars show performance with non-unit latencies in some instructions. All latencies are shown in Figure 4.2.

In each pair of vector simulation results, the left bar, labeled “fm” (for Fast Memory) shows result for an aggressive memory system, one which has as many address ports to memory as the memory functional unit is wide. The right bar, labeled “sm” (slow memory) shows a T0-like memory system, with only one address port. This distinction affects only strided memory accesses, which must execute at one access per cycle for the slower memory model, but execute at the same speed as unit stride
Simulations were performed using two multiplier models: a T0-like, 16-bit fixed-point multiplier with (vector based) emulation code to perform integer 32-bit multiplies, and using the simulator augmented to perform "real" 32-bit multiplies. Only results from 32-bit multiplier simulations are presented. This puts the integer-math hardware present in the vector processor model on par with the superscalar models, which perform 32-bit integer multiplies.

A reference line, showing the best superscalar performance for the benchmark, is also shown on each graph.

4.4.1 Simple Loops

Loops in this section show vector and superscalar performance for a number of small loops, similar to those found in many applications. First several "straight line" loops, ones with no internal control flow, are examined, then a set of three loops containing control flow are examined.

4.4.1.1 Straight line Loops

Loops in this section show vector performance on several small inner loops, shown in Figure 4.3. Loops (a) and (b) perform simple summation across a large array, with loop (b) working on only every other element. Loop (c) contains a simple vector multiplication. The final simple kernel, (d), is the innermost loop from a matrix-multiply-like loop.

Figure 4.4 shows performance for the unit-stride addition loop. In general, increased functional unit width and memory bandwidth provide proportional improvements in execution. Examining the vector execution times of the 2M,2C, 4M,4C, 8M,8C, 16M,16C and 32M,32C models shows the execution time dropping by a factor of two for each doubling of the pipeline widths. Total execution time does not decrease at the same rate due to the presence of scalar overhead, which is not executed faster on the large machines. Comparing these to the 2X4M,8C, 2X8,16C and 2X16M,32C models, cases where memory bandwidth is equal, but is split between
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size/Size Data</th>
<th>Description</th>
<th>(features)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>10000/4999</td>
<td>vector + vector → vector</td>
<td>unit stride</td>
</tr>
<tr>
<td>addition</td>
<td>10000/2500</td>
<td>vector + vector → vector</td>
<td>strided</td>
</tr>
<tr>
<td>multiply</td>
<td>10000/4999</td>
<td>vector * scalar → vector</td>
<td>unit stride</td>
</tr>
<tr>
<td>multiply</td>
<td>10000/4999</td>
<td>vector * vector → vector</td>
<td>unit stride</td>
</tr>
<tr>
<td>simple if</td>
<td>100000/1000</td>
<td>conditional assignment to two different vectors</td>
<td>unit stride</td>
</tr>
<tr>
<td>nested if</td>
<td>100000/1000</td>
<td>nested if construct assignment to different arrays</td>
<td>unit stride</td>
</tr>
<tr>
<td>fast nested if</td>
<td>100000/1000</td>
<td>nested if controlling assignment to scalar</td>
<td>unit stride</td>
</tr>
<tr>
<td>dither</td>
<td>128 × 160 frame</td>
<td>colour space translation from YCbCr to RGB coordinates</td>
<td>mixed</td>
</tr>
<tr>
<td>FDCT</td>
<td>8 × 8 block</td>
<td>forward discrete cosine transform</td>
<td>mixed</td>
</tr>
<tr>
<td>IDCT</td>
<td>8 × 8 block</td>
<td>inverse discrete cosine transform</td>
<td>mixed</td>
</tr>
<tr>
<td>IDCT.hand</td>
<td>8 × 8 block</td>
<td>hand-written assembly version of IDCT loop</td>
<td>mixed</td>
</tr>
<tr>
<td>motiontest</td>
<td>16 × 16 block</td>
<td>computes mean absolute difference(MAD) between a 16 × 16 block in the current frame and a motion-compensated block in the previous frame</td>
<td>unit stride</td>
</tr>
<tr>
<td>motionadd</td>
<td>16 × 16 block</td>
<td>computes the MAD and average of a 16 × 16 block from a future frame and the motion-compensated block of a previous frame</td>
<td>unit stride</td>
</tr>
<tr>
<td>MM1</td>
<td>100 × 100/64 × 64</td>
<td>naive matrix multiply calculation</td>
<td>mixed</td>
</tr>
<tr>
<td>MM2</td>
<td>100 × 100/64 × 64</td>
<td>loop interchanged matrix multiple (no scalar reduction)</td>
<td>mixed</td>
</tr>
<tr>
<td>UUENCODE</td>
<td>60/1</td>
<td>encodes a 4096 byte binary file (one NFS block)</td>
<td>mixed</td>
</tr>
<tr>
<td>UUDECODE</td>
<td>60/1</td>
<td>decodes output of UUENCODE benchmark</td>
<td>mixed</td>
</tr>
<tr>
<td>decryption1</td>
<td>8000 blocks</td>
<td>IDEA decryption algorithm with unrolled inner loop</td>
<td>mixed</td>
</tr>
<tr>
<td>decryption2</td>
<td>8000 blocks</td>
<td>IDEA decryption algorithm with interchanged loops</td>
<td>mixed</td>
</tr>
<tr>
<td>decryption.hand</td>
<td>8000 blocks</td>
<td>hand written assembly code version of IDEA decryption, similar to interchanged code</td>
<td>mixed</td>
</tr>
</tbody>
</table>

Table 4.2: A summary of the benchmarks examined in this thesis.
for(i=0;i<length;i++)
{
    a[i]=b[i]+c[i];
}

(a) for(i=0;i<length;i+=2)
{
    a[i]=b[i]+c[i];
}

(b) for(i=0;i<length;i++)
{
    a[i]=length * b[i]+a[i];
}

(c) for(i=0;i<length;i++)
{
    a[i]=b[i]+c[i];
}

(d) for(i=0;i<length;i++)
{
    a[i]=length * b[i]+a[i];
}

Figure 4.3: Code fragments for simple loops.

Figure 4.4: Unit stride vector addition.

two functional units shows the single functional unit to have a distinct advantage.

Taking the 8M,8C and 2X4M,8C models as an example, the 8M,8C model can be running the addition 6 cycles (4 cycles to complete the first operand load, plus two cycles to start and chain the second load) after the first operand load is initiated. The store can proceed as soon as the final load completes. In cases where the memory pipe is half the width of the math pipe, the math pipe must stall \(32/(2 \times (memory\text{pipe}width) + 1 + latency)\) cycles for the memory unit to be far enough ahead for the math operation to complete without encountering a data hazard. This
forces the addition to be started nine cycles after the first load is initiated (one cycle to launch the first load plus 8 cycles for the second load to have enough data ready to start the addition), and prevents the store from being launched as soon as the first load completes. For half of the execution the memory system is supplying only half its possible bandwidth.

In all cases there is a significant amount of time spent executing only scalar code. This points out the need for aggressive scalar execution or more optimized scalar code generation to obtain the full benefits of vector execution.

Increased memory latency has a small impact on performance. The latency cost is paid only for the first element (or set of elements) in vector memory operations. Thus each cycle of latency adds only $1/32^{nd}$ of a cycle to the time per iteration for strided memory accesses, and $1/8^{th}$ of a cycle to unit stride word accesses.

The SGI C compiler unrolled this loop four times, giving superscalar performance a theoretical limit of 0.25 cycles per loop iteration with one branch unit. Superscalar architectures simulated in this thesis could not exploit all the parallelism available in this loop, achieving at best 1.014 CPLI. This is 33% better than the smallest vector machine (model 2M,2C), but 10% worse than the next smallest vector machine.
machines spend a large portion of their execution time, most over 88%, at full issue rate, but none of the functional unit sets are overly utilized. A large functional-unit set is important, however, so that full issue width can be maintained.

An interesting affect of latency can also be seen in this graph. Looking at the execution times for the 8M,8C and 8M,16C models, we see that the smaller machine perform better. As described in Section 2.1, the latency is broken into two components: the functional unit latency, the time between an instruction being issued to a functional unit and the first set of values being ready, and the chaining latency, the time required for the results from a functional unit to be fed to the input of another functional unit. This simulator uses a chaining latency of one for all models. A timing diagram for loop execution is shown in Figure 4.6. Timing for the second vector load, the addition, and the store is shown.

In the 8M,8C case, the adder consumes data at the same rate as the memory system can provide it, hence the add can start two cycles after the load is initiated. The store operation cannot be launched until the two cycle latency constraint from the add is met and the memory unit is free. The load completes, freeing the memory unit, two cycles after the load starts, allowing the store to start immediately, thus the memory system is kept active throughout execution.

In the 8M,16C case, the add must wait for four cycles after the launch of the load before it can start, causing the load/store unit to become idle. The store must also wait for two cycles after the start of the addition before it can be issued. This introduces a two cycle bubble into the memory pipeline. This behaviour is noticeable in many of the memory-bound benchmarks.

Wider functional units will be seen to improve performance when independent vector computational instructions are available.

Figure 4.5 underlines the need for an aggressive memory system capable of providing good bandwidth for strided memory accesses. Execution with vector processor models assuming multiple addresses to memory per memory functional unit execute the strided memory operations as fast as the unit-stride loop, while machines retriev-
ing one element per cycle per functional unit perform very badly. Machines with two memory units perform significantly better than similarly configured single memory unit machines. Where the single-unit machines require 64 cycles to load all elements for the calculation, the two-unit machines require only 33, allowing the addition to start after thirty cycles, rather than after 61 cycles. The scalar execution time is almost completely masked by vector memory operations.

The SGI C compiler decided that loop unrolling was not applicable to this loop, likely because of the non-unit step size and dynamic trip count. The work performed by the loop body is no more complex than that performed in loop (a), however the number of instructions needed to perform the work was 66% greater. The largest superscalar machine was capable of supporting the available parallelism to the same degree as the three largest machines could in loop (a) because of its large reorder buffer. It took 66% longer to execute the loop due to increased instruction count only. The smaller machines, because of their small buffers, could not look far enough ahead in the dynamic instruction stream to exploit the available parallelism, and so performed much worse.

Figure 4.7 shows vector performance on a simple multiplication. Structurally the loop is identical to the simple addition loop. Thus, under the assumption of unit
latency in the multiplier, performance is identical to addition. Increasing the vector multiplier latency to that of the superscalar multiplier (along with a 2-cycle load latency) has a small impact on execution time.

In the superscalar case, increased latency, of the multiply as compared to the add, improves integer functional unit utilization, but increased instruction count for performing mf10 operations degraded performance.

Figure 4.8 shows the performance on the loop shown in Figure 4.3 (d), which is similar to the innermost loop of a matrix multiplication. Because there are no strided memory accesses in this loop, the only differences in vector execution times lie between unit and non-unit instruction latencies. This loop exhibits slightly better performance than simple addition. The multiplication is between a scalar operand and a vector. Thus the multiplication can be initiated as soon as the memory unit has fetched enough of the first operand for the multiplication to be issued safely. The load of the second vector is thus run concurrently with a larger portion of the computation workload. There is sufficient scalar overhead to completely mask the small increase in the time spent running in the vector unit, thus it is expected that no performance degradation would be found. In addition, the reuse of the a[i] array
reduces scalar overhead slightly, providing the small improvement.

For the alternate latency cases, this masking and reduced instruction count act to decrease the impact of the higher latency multiply, compared to the vector multiply case shown in Figure 4.7.

In the superscalar case, once again, performance is limited by issue width and reorder buffer size.

4.4.1.2 Control Flow

Control flow within a loop can cause special problems for vector processors. Different approaches for handling it are outlined in Section 3.6.3. This compiler uses a mask vector to conditionally combine the values calculated for the entire vector length with the values in memory. This results in the insertion of an extra load before each conditionally controlled store to retrieve the values from memory. This also results in the vector processor performing much more work than the superscalar processor to achieve the same goal. The loops in this section, shown in Figure 4.9 are used to characterize the behaviour of the two types of machines in the presence of control flow.
if (a[i]==f[i])
    b[i]=c[i];
else
    f[i]=d[i]+a[i];
}

if (a[i]==f[i])
    if (b[i]==1)
        s=a[i];
    else
        f[i]=c[i];
else
    if (s>0)
        s=255;
else
    s=s;


\[ \text{Figure 4.9: Control Flow Loops} \]

Figure 4.10 shows performance results for the loop shown in Figure 4.9 (a). This loop was run on three data sets, one which always selected the then clause, one which always selected the else clause, and one which alternated between the two clauses. Superscalar performance is shown for each data set. A unique feature of the vector execution of control flow is that execution time is the same for all data sets, so only one set of bars are presented. The horizontal lines indicate best superscalar performance on each data set.

The first observation to be made on vector performance in this graph is the large scalar overheads incurred, and the inability for even the slow 2M,2C machine to
mask it. The large number of arrays in this loop have produced an undesirable amount of register spilling, requiring scalar loads of address operands before most of the vector memory operations and extra load-store pairs to perform loop-overhead calculations. Scalar code generated by this compiler relies on the register allocation mechanism distributed which the machsuif package. An improved register allocator could reduce scalar overhead down to levels comparable with the simpler loops of the previous section. This kind of change would bring vector performance numbers for most configurations ahead of the superscalar performance numbers for the realistic data sets which exercise all branches of the nested if construct.

The superscalar numbers are almost identical for the zero and fifty cases, but the hundred case is markedly different. For all three data sets the single branch unit is fully utilized, but in the hundred case, the peak instruction issue rate is five, rather than six as in the other two cases. The small amount of work in the then clause of the loop quickly saturates the memory access buffer, slowing the processor down. Smaller superscalar machines, which cannot exploit the same parallelism as the larger machines, never saturate the buffer, and behave the same way across all three data sets.

In the nested loop, whose performance is shown in Figure 4.11, zero indicates the outer else clause is always executed, the hundred case indicates the outer then clause is always taken, with a 33%/66% split between the inner then and else clauses. Fifty indicates the outer else was taken half the time, with a 33%/66% split between the inner clauses the other half of the time.

The superscalar machine exhibits the same kind of behavior in the nested loop as it does in the single-if loop. The zero data set performs the best out of the three. It contains reasonable work and requires evaluation of only one branch to commit results to memory. The fifty data set has intermediate performance, much worse than the zero data set, because two branches must be evaluated fifty percent of the time. This branch evaluation also reduces the rate at which memory operations can be committed, reducing over-all instruction issue rate. The hundred data set suffers from the same problems as the fifty set, except that it must always evaluate two
Figure 4.11: Nested IF. The zero-group indicates the outer else clause was always executed. The hundred-group indicates that the outer then was always taken, with a 33%/66% split between the inner then and else clauses. The fifty indicates the outer else was taken half the time, with a 33%/66% split within the clause.

The vector processor has the same relative performance in the zero data set case as it did in the case of the simple loop. Roughly three times as much work is done by the vector processor as is done by the superscalar, so vector performance lags the superscalar considerably. In cases where more than one branch needs to be evaluated it compares much more favorably. It still suffers from significant scalar overhead, however.

The largest single performance penalty experienced by the vector processor in these two loops is its increased memory bandwidth requirements. This increase is attributed to both the need to execute all clauses of the loop, requiring more operands to be loaded, and the inability to exploit data reuse across all clauses of the if constructs. The problem of operand loading is significant in cases where all iterations through a loop take one path, or when costly strided accesses are performed, but in cases with unit stride and a data set which exercises all control paths the added penalty is less significant. The forced preload of the VLR possible target elements of
the conditionally controlled store, memory operations not present in the superscalar code, is a much larger factor. The impact of these instructions is highly dependent on the machine configuration and the kind of access the store is (strided or unit-stride).

The loop depicted in Figure 4.9 (c) is a typical clamping routine, where a set of values are clamped between a upper and lower bound. Three data sets were used: the zero data set contains all zeros, the big data set contains values greater than the maximum allowed value, and the mixed set has one third of its values set to zero, one third in range, and one third above the allowed range.

The important thing about this loop is that no memory operation is required within the if construct. The SUIF vectorizing compiler can exploit the scalar temporary, eliminating all preloads before conditional stores to (the scalar expanded array) s, and only performing the program-specified write to the array b[i].

Performance results are shown in Figure 4.12. Superscalar performance for the zero data set is similar to that of previous benchmarks, while that of the hundred data set is slower. The low instruction count of the loop, combined with the need to evaluate two branch tests prevents a wide superscalar from issuing the maximum
is a pathological case for the branch predictor. A different path is taken on each iteration of the loop, resulting in poor prediction accuracy (56%) and a proportional loss in performance.

The vector processor does extremely well in this example. Except for the two smallest models, all vector processors perform at about the same speed as the superscalar on the zero data set. If the scalar overhead is ignored most models perform significantly faster.

### 4.4.2 MPEG Kernels

Loops in this section where taken from the MPEG-I package distributed by Berkeley. Five loops are examined. The forward discrete cosine transform (FDCT), motion-test, and motion-add loops are all taken from the MPEG encoder, while the inverse discrete cosine transform (IDCT) and dither loops were taken from the MPEG decoder.

The FDCT performs a decimation-in-space transformation on $8 \times 8$ pixel blocks of a source frame from a video stream. The source image is tiled, and the function called on each block. This function was written to perform fixed point operations for speed, and contains no control flow. Performance results are shown in Figure 4.13, normalized to cycles per pixel.

The most obvious feature of this graph is the large scalar overhead associated with all vector models. There are two important reasons for this. Firstly, there are sixteen separate pointer variables required to control the array accesses in this loop. Poor scalar register allocation introduces register spilling on the base address determination of most array references. The second relates to the size of the data being worked on in the loop.

All vector loops contain scalar code to initialize the base address variables, followed by the vector loop body and a scalar code sequence for updating the pointers after each loop iteration. In the case of loops examined thus far, the initialization code gets amortized over multiple iterations of the vector loop, making it effectively take no time, given a sufficiently large vector loop trip count. The pointer update cost
is paid on each vector loop iteration, so is amortized only over the $VLR$ iterations of the original loop performed by the single pass through the vector code.

In the case of FDCT, it is iterating over an $8 \times 8$ block of data using a single loop with a fixed trip count of eight. When vectorized, this low trip-count loop produces a strip mined loop which is executed only once, thus the initialization overhead is not amortized out, and the pointer-update cost is paid without the updated pointers being used. Another penalty paid by these low trip-count loops is in terms of function call overhead. As part of the -O2 option set while compiling programs for Cello simulation, global register allocation is performed, eliminating extraneous register saving. The SUIF scalar register allocator does not perform this optimization, so full register file saves and restores are performed as part of the function call.

A further effect of the small data size can also be seen in the vector performance numbers. Machines supporting functional unit widths of eight or greater exhibit identical performance. Since the $VLR$ is never set above eight in this benchmark, having more then eight pipelines within a functional unit has no effect.

This loop contains strided memory accesses almost exclusively, so multiple memory unit configurations with simple memory systems supporting one-element-per-
cycle strided loads show an advantage. This is not a large effect in this case, however, since strided loads of eight elements do not have a large performance penalty.

The only anomaly in superscalar performance is in the change from the 4I,48R to 6I,32R models. The decreased reorder buffer size cannot support the higher issue machine at a rate comparable to the smaller issue width machine.
When performing MPEG encoding, the current frame can be described in terms of motion with respect to the previous frame or future frame. These two loops participate in that encoding. Each uses a doubly nested loop to iterate over a 16 × 16 block of the current frame and the corresponding block in the previous frame (or the previous and future frame in the motion-add loop). Processing proceeds one row at a time, and involves computation based on corresponding elements in the source arrays and a reduction of these computed values. After each row is processed, the program conditionally breaks out of the outer loop based on the value of the reduction variable. Computation requires approximately twice as many operations in the motion-add case as it does in the motion-test case. Data for this loop was chosen so that full execution of the loop nest occurred.

The code distributed with the MPEG-I source package was not vectorizable by the SUIF vectorizing compile in its original form. The inner loop had been fully unrolled for performance reasons, resulting in a single loop with a break condition at its end. This loop is vectorizable, but full support for break constructs has not been implemented. The inner loop was manually re-rolled, and the re-rolled version of the loop run on both the vector and superscalar models. Performance results for these runs are shown in Figures 4.14 and 4.15.

Both loops show significant scalar overhead. Like the FDCT case, this is caused by the small data set being worked upon. Vector instructions are issued with a length of 16, and the vector loop is executed only once. Thus amortization of the scalar loop overhead is small, and amortization of the function initialization and clean-up code is non-existent. The time spent in vector execution is small compared to all superscalar implementations, indicating a stronger implementation of the scalar component of the compiler would easily push vector performance past superscalar performance in these loops.

Similar to the FDCT case, performance is capped by the short vector length. Machines with pipelines wider than 16 elements show no performance improvement.

Superscalar performance is poor for this loop, wide machines cannot maintain
Figure 4.16: IDCT. Graph (a) shows performance for the hand-coded loop, while (b) shows compiler generated code.

A high issue rate because the small size of the inner loops, iterating over only 16 elements at a time, cannot supply the parallelism need to to support a sustained high issue rate. Issue rate is better in the motion-add loop, because of the larger workload.

These three loops perform a total of 86% of total instructions executed in performing MPEG encoding. Performance gains in these procedures will have a significant impact on the sustainable rate of frame encoding for real-time MPEG applications.
of the FDCT loop. This function is called for each $8 \times 8$ block in the current frame which contains non-zero FDCT coefficients.

For this loop, two versions are available, the output of the SUIF vectorizing compiler (IDCT), and a hand-coded assembly version (IDCT.hand) [For94]. The performance results for both versions of the loop are presented in Figure 4.16. Superscalar performance, based on the output of the SGI-C compiler, is presented in both graphs for convenience.

There are two important observation when comparing the hand-written code with the compiler-generated code. The first is the scalar overhead caused by the large number of arrays present in the code. These introduce a large scalar penalty to the compiler code. This coupled with the loops small vector length, which allows for only a single iteration of the vector loop, prevents amortization of the scalar portions of the code. The second thing to notice is that the vector execution times of the two are almost the same. This similarity lends justification to comparing just the vector execution portion of the other graphs to the superscalar performance.

The source code is structured as a set of array loads to scalar temporaries, followed by computation on those temporaries and the writing out of the result to the target array. These scalars are expanded into temporary arrays during compilation. The similarity between hand coded and compiler generated vector execution also shows that the methods for allocating array temporaries to registers is effective.

Like the other small data set loops, increasing machine size shows performance gains only up to the maximum loop length.

Compared to superscalar performance, the vector processor performs about as well superscalar implementations.

The final MPEG loop, dither, is a colour-space mapping loop. It translates YCrCb colour coordinates to RGB colour coordinates. This loop iterates over the entire $n \times m$ frame. Input data consists of three arrays: an $n \times m$ array containing all Y values, and two $(n/2) \times (m/2)$ arrays containing the Cr and Cb values respectively. Each value from the quarter-sized arrays is used to calculate the values of four pixels in
the final image. The quarter-sized arrays are iterated over using unit-stride accesses, while accesses to the full sized array and the output array are strided. Each iteration through the loop calculates the RGB values for four adjacent pixels.

This loop contains scalar computation on scalar temporaries, followed by "clamping" of the co-ordinate values to the range of an unsigned character. This clamping is identical to the control flow presented in Figure 4.9 (c), replicated three times per pixel (twelve nested ifs total). This structure requires the evaluation of 6 branches per pixel. Performance results are shown in Figure 4.17.

Again in this loop, for most configurations, scalar overhead drops vector performance below that of the superscalar processor, but if scalar optimizations were powerful enough to eliminate or hide the scalar overhead effectively, vector performance would exceed that of the superscalar for most vector hardware configurations.

The single branch unit in the superscalar models puts a lower bound of six cycles per pixel on execution. The largest superscalars almost reach this bound.
for(j=0; j<B_COL; j++) {
    for(k=0; k<B_ROW; ++k) {
        sum+=a_matrix[i][k] * b_matrix[k][j];
    }
    c_matrix[i][j] = sum;
}

Figure 4.18: Matrix Multiply Loops. (a) naive implementation of matrix multiplication (MM1). (b) loop interchanged version (MM2).

### 4.4.3 PGP and Others

A number of applications other than those in the multi-media area can benefit from a vector processor. The following three benchmarks have been selected to show vector performance on a broader range of programs. Matrix multiply is a typical linear algebra operation, found in many image-transformation and modeling/simulation applications. The UU tools (encode and decode) are standard data-translation utilities used for communicating binary data through a text-based channel. The IDEA decryption algorithm is part of the PGP (Pretty Good Protection) data-encryption suite. Speed of data encryption and decryption is of growing importance as more transactions are performed across insecure networks.

Matrix multiplication is a common linear algebra kernel. Figure 4.19 shows performance for two implementation of the kernel. Figure 4.18 shows the two loops under examination. Loop (a) is the naive implementation: the dot product of a row and column vector is determined and stored in the corresponding position of the target array. This inner loop produces vector code performing one unit-stride load (for a_matrix) and one strided load (for b_matrix), as well as the reduction emulation code. Loop nest (b) has interchanged the loop nest, producing a inner loop performing three unit-stride memory accesses but requiring no reduction operation.

Looking at Figure 4.19 (a), performance data for the naive loop, it can be seen the vector processors with aggressive memory systems, a system as good as the superscalars, perform as well as the superscalar if scalar overhead is ignored. Even under the high latency models, performance is approximately equal to the mid-sized
Figure 4.19: Matrix Multiply. (a) naive implementation of matrix multiplication (MM1). (b) loop interchanged version (MM2).
In the loop interchanged case, shown in Figure 4.19 (b), even the small 2M,2C vector machine almost beats superscalar performance (ignoring scalar overhead). The superscalar experiences a slowdown in this case. The increased loop complexity, requiring three memory operations, has aggravated the problems of a loop already running at peak issue rate. For the vector execution case, the unit-stride memory access pattern eliminates the need for a highly addressable memory system, which is needed for the naive loop, and the removal of the reduction code in the inner loop reduced the total amount of work needed to be performed within the loop, allowing for continuous full-bandwidth memory use.

The encoder/decoder loops from the UU tool set were also benchmarked. These tools are used to translate a binary file into a representation guaranteed to contain only printable characters. In real terms this means taking three byte sets and expanding them out to four printable characters, with these four character sets emitted in lines of up to sixty characters (plus one control character, for a total of 61 characters). In their original form, these loops operate on 45-byte blocks of the input data, or single lines of encoded data. I/O is performed to retrieve the input data as a block and then as each output byte is determined. The encoder/decoder loops were rewritten to operate on 4KB blocks of data without affecting the encoding format. The modified tools read in 4KB of data to be encoded (or enough encoded data to decode to 4KB), and write output into a buffer, which is flushed after the en/decoding is complete. Standard implementations can decode vector-encoded files, and the vector decoder functions on standard encoded files.

Figure 4.20 shows vector performance reaching a maximum at 16-wide functional units. Despite the modification to the I/O portions of the program, encoding and decoding functions still work on 45/61-byte blocks of data from the input buffer. This results in a strip length of 15 in the vectorized loops. Performance peaks when the strip length of the inner most loop matches the width of the vector functional units. Providing wider units does not improve performance. Both the encoder and decode loops perform strided memory accesses, and the dependence on aggressive memory
Figure 4.20: UU Tool kernels.

<table>
<thead>
<tr>
<th>Cycles-per-decoded-byte</th>
<th>Cycles-per-data-byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2.000</td>
<td>2.000</td>
</tr>
<tr>
<td>3.000</td>
<td>3.000</td>
</tr>
<tr>
<td>4.000</td>
<td>4.000</td>
</tr>
<tr>
<td>5.000</td>
<td>5.000</td>
</tr>
<tr>
<td>6.000</td>
<td>6.000</td>
</tr>
<tr>
<td>7.000</td>
<td>7.000</td>
</tr>
<tr>
<td>8.000</td>
<td>8.000</td>
</tr>
<tr>
<td>9.000</td>
<td>9.000</td>
</tr>
<tr>
<td>10.000</td>
<td>10.000</td>
</tr>
<tr>
<td>11.000</td>
<td>11.000</td>
</tr>
<tr>
<td>12.000</td>
<td>12.000</td>
</tr>
<tr>
<td>13.000</td>
<td>13.000</td>
</tr>
<tr>
<td>14.000</td>
<td>14.000</td>
</tr>
<tr>
<td>15.000</td>
<td>15.000</td>
</tr>
</tbody>
</table>
is evident in the graphs.

Encoding involves some conditional operations, so superscalar performance is degraded by the need to evaluate many branch operations. Superscalar decoding executes at close to peak issue rate.

In both loops, assuming a fast memory system, vector performance for all but the 2M,2C machine exceed superscalar performance (disregarding scalar overhead). The 2M,2C machine exceeds the performance of the similarly equipped superscalar machine.

The final member of the benchmark suite is the decryption procedure from the IDEA encryption suite [Lai91]. This is used in the popular PGP data encryption package. In this scheme, the encrypted data is partitioned into blocks of sixteen bytes. This block is processed with the decryption key, resulting in a “decrypted block”. This decrypted block is XORed with the encrypted block to produce the decrypted output. Sample implementation code distributed by the authors implement this functionality with “decrypt” and “XOR” functions working at the block level. This implementation facilitates low memory requirements, since I/O could be performed on a per-block basis, but provides little parallelism for the superscalar to exploit and presents no vectorizable loops to the SUIF vectorizing compiler. The inner loop within the “decrypt” function utilizes “carry-around” scalar variables: scalars which carry data from the end of the loop body back to the head of the loop. This prevents the vectorization of this loop unless some additional transformations are performed.

For the performance study the code was re-written so that all blocks in the message are processed by the “decrypt” function in a single call, followed by the XORing of all blocks to provide the decrypted text. In this re-written form, the “decrypt” function consists of an outer loop iterating over all blocks, and an inner loop performing the decryption of a single block. The code structured in this manner shows small improvement in superscalar simulation, due to reduced instruction call overhead, but is still not vectorizable since the “carry-around” scalar problem has not been addressed.

---

[1] The XOR operations are vectorizable at this time.
able vectorization. In the first case, loop unrolling was applied to the inner loop, removing the carry-around scalars, however, this also increased the size of an already long inner loop by a factor of eight. In the second case, loop interchange was applied, which expanded the problematic scalars into array temporaries and allowed vectorization. Only these last two versions, and a third a hand-coded assembly version, were fully simulated.

The performance results for the final three versions of the IDEA loops are presented in Figure 4.21 (a), (b), and (c): the interchanged loop, unrolled loop, and hand coded loop, respectively. The performance of the original code is not show, but is worse than the unrolled loop case. The vector processor shows dramatic performance advantages over superscalars in this loop. These loops are all compute-bound, containing few memory references.

Poor superscalar performance can be attributed to the size of the decryption loop. This loop contains a long instruction sequence consisting of a single long dependence chain. This prevents the superscalar from selecting independent instructions from within a single loop iteration. It could exploit cross-iteration parallelism, but the loop body is large compared to the size of the re-order buffer. In the case of the largest superscalar, with a 128-element buffer, running on the interchanged loop nest, enough of the next iteration is present in the buffer for some cross-iteration parallelism to be found. In this one case superscalar performance exceeds the smallest vector configuration. In all other cases the buffer is too small to hold enough of the loop for parallelism to be found. This can be seen in the issue rates of the superscalar processors, which spend little of their execution issuing at peak rate. Superscalar performance before the vectorizing transformations were applied to the “decrypt” function were similar to the unrolled results.

A further hindrance to performance is the amount of work need to decrypt the data. To process the test data set 7 607 181 instructions are executed by the superscalar processor. Even if peak issue rate could be achieved, 9.9 cycles-per-byte would be the best possible performance achievable by one of the 6I machines. This is
comparable to the worst of the vector machines working with the loop-interchanged version of the algorithm, but is not even close to the performance achieved on the unrolled or hand-assembled versions.

The vector compiler-generated code compares favorably with the hand assembled version of the algorithm. Even excessive scalar overhead is not present.

As an additional experiment, a number of unrealistically large superscalar machines were simulated with the goal of developing a superscalar capable of approximating vector performance on this benchmark. The results of this experiment are shown in Figure 4.22. These simulations were run on the same hardware configuration as the 61 superscalar machines, but with increasing issue widths and reorder buffer sizes. Numbers from the previous graph set are also shown for comparison.

As can be seen, the large loop created by the loop unrolling transformation could never be run at a reasonable speed by even the 24-way superscalar. The interchanged loop could be run at a speed comparable with the vector models on the 24-way machine, but only with a ROB large enough to hold eight iterations of the loop: one of 768 entries. It is expected that a ROB large enough to hold eight instances of the unrolled loop would enable superscalar execution of it in a time comparable to the vector machines as well.

4.5 Summary

Throughout this chapter it has been seen the SUIF vectorizing compiler is capable of handling a wide variety of vectorizable loops constructs. A number of areas for improvement were identified through the ANL benchmark suite, but the current capabilities were shown to be effective on many benchmarks from applications of growing importance today.

The compiler was shown to produce poor scalar support code for vector code sequences, but the vector code sequences were shown to have almost identical behaviour to hand-coded versions of the same loop. Examination of the scalar code sequences shows wide room for optimization.
Figure 4.21: DEA decryption. (a) Interchangeable (b) unrolled (c) hand coded
Performance of the vector machine models to superscalar models consistently showed the superscalars to be issue-limited or unable to issue at full rate, while the vector machines consistently performed as well as the largest, most complex superscalars.

An interesting vector configuration trade off was also identified. Loops containing large quantities of computation benefit from wide math pipelines regardless of the width of their memory pipes, while memory-limited applications are slowed down by the required instruction timing if the math units are widened than the memory units.

In the last loop, the IDEA decryption kernel, it was shown that superscalars cannot exploit parallelism from loops with long dependence chains, even if the loops are extremely parallel, unless large reorder buffers are supplied. Even the smallest vector machine was able to significantly outperform the superscalar implementations on this loop.
Chapter 5

Conclusions

This thesis has presented the SUIF vectorizing compiler and machSUIF code generation for the Torrent ISA. Preliminary performance comparisons of the vector and superscalar architectures have also been presented.

The vectorization pass accepts scalar loops nests and performs a set of analyses and transformations to locate vector parallelism. Loops containing IF-based structured control flow, break-based control flow, reductions and complex, acyclic dependence relationships are all supported. Loops containing a mixture of vectorizable and non-vectorizable components are partially vectorized. The SUIF vectorizing compiler performs these functions with no assumptions about the target architecture other than the maximum vector length.

Code generation supports most transformations supported within the vectorization pass: only breaks, conditional reductions and indirect memory accesses are not fully supported. In addition, emulation code is produced for operations not supported in the target hardware, such as 32 bit multiplication and reductions. A simple register allocation scheme, implemented in the code generator, removes unnecessary memory operations to compiler-created arrays. All programmer-specified memory operations are performed.

Performance analysis has shown that vector processors are not limited in application to large scientific and engineering numerical applications. Vector parallelism can be located and exploited in processor-hungry multimedia and filter applications and,
given that round overheads can be brought within manageable limits, vector architectures are able to exploit more of the available parallelism than realizable superscalar configurations.

5.1 Contributions

The primary contributions of this thesis have been the development of an architecture independent vectorization pass to augment the SUIF compiler, the implementation of a code generation pass for the Berkeley Torrent ISA, the implementation of a configurable hardware simulator for vector architectures, and a preliminary evaluation of the feasibility of using vector architectures to multimedia workloads.

The vectorization pass provides a mechanism for exposing vector parallelism within an application to subsequent compiler passes in a way independent of vector architecture being targeted. This facilitates the rapid development of compilers for new architectures, while amortizing the investment in developing analysis and transformation routines within the vectorizer over all target vector architectures.

The code generator targets the T0 microprocessor, and fills a significant void in the software development environment which accompanies SPERT boards. It also shows that it is reasonable to defer many architecture-dependent operations, such as the use of condition vectors, to the code generation phase.

The hardware simulator provides a simple mechanism for comparing different vector processor configurations and for evaluating the impact of compiler transformations on execution.

5.2 Future Work

Future work can proceed along several routes: extension of the vectorizer, addition of code generation facilities for other architectures, and refinement and extension of the performance studies.

The compiler, as presented in this thesis, is capable of vectorizing a wide variety of
many pointers to places where extensions to the analysis and transformations supported by the compiler can be made. Many loop transformations are possible within the framework supported by this compiler, but the addition of conditional vectorization would have the broadest reaching impact, since it serves to augment all other loop transforms being applied by potentially mitigating the affects of dependence constraints that are not resolvable at compile time.

In loops where hand-coded versions of the routines under examination are available, comparing execution times shows that the vector portions of code produced by the SUIF vectorizing compiler is of good quality, but a large additional scalar overhead is incurred by the compiled code. A two-pronged approach, consisting of improved high-level scalar management coupled with a post-generation (but pre-ASCII generation) optimization and scheduling pass, would alleviate this problem considerably.

Several vectorizing transformations are supported for which code-generator support has not been implemented. This implementation would be a logical first step to extending the compiler.

The development of code generators for other other architectures would prove the architecture independence of this compiler. The new multi-media processors, such as the ULTRA-SPARC with VIS extensions, would be an excellent candidate for this research. Processor vendors for most of these machines have already developed powerful scheduling passes within their compilers in support of these instructions, but application programmers are typically forced to use a macro-based API to access the functionality. Using the SUIF vectorizing compiler as a source-to-source translator would remove the burden of vectorization from the programmer. The code generator presented in this thesis could be replaced with a macro insertion pass. SUIF facilities for translating its intermediate form into C could then be used to export the transformed code to the native compiler for translation and optimization. Such a pass could even be inserted immediately after vectorization, using a mixture of type casting techniques around the array-based references to translate the normal data-typed
(character, integer) information in the original loop into the multi-media data-types (pack_char8, pack_short4) used by the macros. Elimination of strided memory accesses, and efficient mechanisms for performing those which could not be eliminated, will be particularly important for these machines.

Extensions to the performance study could include the addition of new benchmarks, extensions of vector hardware to support floating point operations, the modeling of real memory systems, and modeling of machines containing shorter or fewer vector registers. Compiler passes could easily be modified to use fewer resources than the hardware simulator provides, creating an instruction trace consistent with smaller hardware to be processed by Viola. Memories in this thesis where assumed to be ideal. In the superscalar case this is optimistic, since in real machines the processor runs much faster than the memory system and caches are required to achieve best possible performance. Vector machines typically do not access memory though a cache, so the ideal model is less optimistic for these machines. In the case of the SPERT board, the system on which the simulator is based, memory and the processor run at the same speed, so the memory model is accurate, a processor running at twice the speed of the memory system was also modeled. These latencies should be increased to reflect vector processors running at clock speeds comparable to superscalar processors.
Bibliography


