LOOP OPTIMIZATION TECHNIQUES ON MULTI-ISSUE ARCHITECTURES

by

Dan Richard Kaiser

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Communication Sciences)
in The University of Michigan
1994

Doctoral Committee:

Professor Trevor N. Mudge, Chair
Associate Professor Richard B. Brown
Professor Edward S. Davidson
Professor Ronald J. Lomax
Associate Professor Karem A. Sakallah

Dedicated to the memory of
Francis Marie Kaiser,
1911-1994.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Trevor Mudge for his continued support and encouragement. I would also like to thank my committee for their comments and suggestions. Thank you to the students and faculty of the Computer and Communication Sciences Department, where I began my graduate work, and to the students and faculty of the Aurora project. Thanks to my parents for their support during my school years. A special thanks to my family, Pam, Seth and Tadd, for their support and encouragement, and for bearing with me through the long process of finishing my dissertation.

This work was partially supported by the Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAL03-90-C-0028, and by Cadence Design Systems, Inc.

TABLE OF CONTENTS

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

CHAPTER I
INTRODUCTION

Considerable effort has been put into designing computer architectures which exploit instruction level parallelism in an attempt to achieve execution rates of greater than one instruction per cycle. A wide variety of architectures and accompanying compiler algorithms have been proposed and developed. The best examples have shown good performance improvements relative to scalar architectures constructed in similar technologies.

Much of the experimental work on new architectures has focused on just the hardware architecture, with perhaps one scheduling algorithm designed for the architecture. A new architecture is generally compared to a similar scalar architecture as a reference point. Few experiments have compared different architectures to each other or investigated compiler scheduling algorithms across architectures, because of the difficulty of retargeting the compiler.

This work is a first step towards a direct comparison of different architectures in conjunction with different scheduling algorithms. We compare loop scheduling techniques on several architectures, together with accompanying compiler optimization techniques. In particular, loop optimizations as performed by an optimizing compiler are implemented on a set of multi-issue architectures, allowing the interactions between the loop optimizations and the architectures to be studied.

1 Scheduling

Instruction scheduling is the process of determining an execution order for a set of operations. The instruction scheduler accepts a directed graph {V,E} of operations ($o_i \in V$) and dependencies between the operations ($<o_i,o_j> \in E$), and produces an ordered list of operations $L = <o_1, o_2,..., o_n>$. The ordered list L maintains the dependencies E of the original graph, i.e. if the graph contained a dependency $<o_i,o_j> \in E$, o_i appears in the list before o_j. The function computed by the scheduler is shown in (1).

$$s:\{V,E\} \rightarrow <o_1, o_2,..., o_n> \text{ where } \forall <o_i,o_j> \in E \Rightarrow i < j. \qquad (1)$$

After the ordered list of operations is produced, the code generator will transform the list of operations into a list of instructions which can be executed on a particular architecture.

Instruction scheduling must be correct: the order placed on the set of instructions must maintain the semantics of the original list of instructions. The semantics of the original list is called the *program order* or *in-order semantics*. The in-order semantics is dictated by the programming language. The ordering between instructions is determined by the control and data dependencies between the instructions and is encoded as the set of dependencies E in the program graph. This is generally a partial ordering, which allows some leeway for the scheduler to reorder the instructions to improve execution efficiency. For instance, in the program segment shown in Figure 1, statement 3 is dependent on statement 1 and statement 1 must be scheduled prior to statement 3. Statements 1 and 2 are independent and can be scheduled in any order. The relationship between statements 2 and 3 depends on the values of i and j and may be dependent.

```
1.  a = 5
2.  x[i] = 10 * x[j]
3.  d = x[j] * a
```

FIGURE 1. Source for a vector loop

Control dependencies are dependencies between conditional instructions and any other instructions whose execution depends on the conditional instructions. In-order semantics does not allow dependent instructions to execute until the conditional instruction has been executed and the result of the condition is known. Since control dependencies are often a significant performance limiting factor, some execution models relax the requirement that the result of the condition be known before the dependent instruction begins execution. This is usually referred to as *speculative execution*. If speculative execution is allowed, some method must be provided to undo the effects of executing the dependent instructions if the eventual resolution of the condition determines that they should not have been executed.

Data dependencies are formed by the sharing of data and memory locations between instructions. There are four types of dependencies. If A and B are two instructions with A preceding B in program order, then the input and output locations of the two instructions can be denoted by the sets {In_A, Out_A, In_B, and Out_B}, respectively. Furthermore, the possible dependencies between A and B can be defined as follows:

1. *Flow dependencies* are the locations in $\mathrm{In}_B \cap \mathrm{Out}_A$.
2. *Anti-flow dependencies* are the locations in $\mathrm{In}_A \cap \mathrm{Out}_B$.
3. *Output dependencies* are the locations in $\mathrm{Out}_A \cap \mathrm{Out}_B$.
4. *Input dependencies* are the locations in $\mathrm{In}_A \cap \mathrm{In}_B$.

In some sense, flow dependencies are the only true dependencies because they express the sharing of data between instructions. Flow dependencies must be honored to obtain correct execution semantics. Anti-flow and output dependencies arise due to sharing memory locations between different instructions. These dependencies can sometimes be removed by renaming memory locations. Input dependencies come from the sharing of memory locations between instructions. The discovery of input dependencies is not important for correctness, they do not impose an execution ordering, but they can be used to improve execution efficiency.

There is one other set of constraints that may be imposed on the instruction execution order: exceptions produced by the execution of instructions should be *precise*, i.e. the machine state following the handling of an exception should appear as though any instructions following the exceptional instruction had not executed. This requirement can be quite restrictive. Precise exceptions in effect introduce a control dependency between every instruction which can produce an exception and any following instructions. Implementing this in an aggressively scheduled machine requires hardware support to maintain and restore the correct machine state when an exception is encountered.

Implementing precise exceptions efficiently is a difficult problem. In the interest of providing performance, some architectures do not provide precise exceptions. Other architectures make precise exceptions optional, so that users only incur a performance penalty if they require precise exceptions. Allowing reordering while providing precise exceptions is a form of speculative execution, and processors designed with speculative execution in mind can usually provide precise exceptions with little extra overhead, having already incurred the hardware cost of implementing out-of-order execution.

If only correct execution were required, or the target architecture had no parallelism, the instruction scheduler would have nothing to do. There would be no performance advantage in reordering the instructions. The in-order semantics would be the final word on instruction ordering and the instruction list produced by the code generator would not need to be scheduled. However, most current high-performance commercial machines have some form of parallelism, usually in the form of multiple function units or pipelining, and we can expect that machines issuing 4 to 8 instructions per cycle will become much more common. On machines with the hardware to support even modest amounts of parallelism, exploiting the hardware to improve performance requires intelligent instruction scheduling.

Instruction scheduling can take place either in the compiler or the processor. Instruction scheduling at compile-time is called *static* instruction scheduling. *Dynamic* instruction schedule occurs at run-time, and is performed by the processor itself. There are advantages and disadvantageous to each type of scheduling. Static instruction scheduling has the advantage that the compiler can use a much wider scope to gather scheduling information. The disadvantage is that some information that can affect performance is not available until run-time. Dynamic schedulers have access to run-time information, but the scope of available information is typically much more limited than is available in the compiler. Dynamic instruction scheduling also requires more complex hardware, which can adversely affect performance.

We will investigate a set of scheduling algorithms by using them to schedule loops for a set of machine architectures and then comparing the relative performance of the code produced by each algorithm. As part of this investigation, we will examine several instruction issue policies and their interactions with the scheduling algorithms employed in the compiler.

2 Methodology.

The goal of this work is to compare the performance of different architecture-compiler *systems*, i.e., a computer architecture in conjunction with compiler scheduling algorithms. In particular we want to explore the interaction of various scheduling algorithms with different processor features such as instruction issue policies. To explore this design space requires a compiler in which the scheduling and other related algorithms can be modified, as well as simulators for the range of architectures under study.

A compiler which allows access to its scheduling algorithms is a difficult item to obtain. Few compilers are available which are well documented and allow access to their compilation process. In addition, some of the scheduling and other optimizations require support early in the compilation process. This means that modifying the code generator without access to the analysis portions of the compiler precludes important scheduling and transformation opportunities. GNU C is publicly available and it is well documented. Unfortunately it does not readily allow the user to experiment with optimizations prior to code generation.

Because of the difficulty of obtaining and modifying a compiler in conjunction with the other necessary work, much of the research into computer architecture is done without accompanying work on the compiler. This greatly reduces the amount of work involved an investigation, but it creates a tendency toward hardware based solutions.

Failing to investigate compiler algorithms along with architectural variations can lead to incorrect conclusions. Even though the architectural changes seem entirely hardware based, there is a strong software interaction which must be considered. For instance, superscalar machines seem like a good idea because they can provide performance improvement with an existing instruction set architecture (ISA), without new compiler support. However, this expectation of performance improvements without compiler support may prove to be optimistic. In a study of a SPARC superscalar architecture [102], Lee et al. write:

> "Unfortunately, an optimal scheduling policy is very hardware dependent. The base compiler we use was not targeted for superscalar hardware, and most of the optimizations must be applied manually. Our results confirm that superscalar hardware alone would gain little without support from an optimizing compiler."

Another possible method of investigation is to manually schedule the instructions for a new architecture. This is possible for small benchmarks and can quickly yield performance numbers for an architecture. The are two potential pitfalls with this approach: Manual coding is tedious and error prone, and thus hard to do consistently for even medium size benchmarks. The other more important problem is that it is impossible to assure ourselves that we are faithfully employing techniques available to a compiler. By building a compiler and insisting on using it to generate all the code schedules, we avoid these problems.

Because of its key role in providing computer performance, a central part of this investigation will involve compiler's optimization algorithms and scheduling techniques. The particular optimizations employed, and the analysis necessary to support the optimizations are presented and discussed in Chapter IV. The scheduling techniques used in the compiler are discussed in Chapter III and Chapter IV.

The choice to build an optimizing compiler is not without its own problems, however. Building an optimizing compiler is a huge undertaking. To mitigate this problem we have reduced the effort by limiting the benchmarks we execute to a limited set and also by allowing the compiler, referred to herein as Tortoise, to execute slowly. Narrowing the set of benchmarks reduces the magnitude of the programming task because we do not have to handle all the constructs in a rich language such as C or FORTRAN. Allowing the compiler to run slowly frees us to use a more powerful, interpreted language to construct the compiler in, and relieves us of the tuning and careful programming required to make a compiler run efficiently.

As with all computer performance studies, we would like to measure execution time. In [67], Hennessy and Patterson give equation (2) for the execution time:

$$T = I \times CPI \times Tc \tag{2}$$

where T is the execution time, I is the number of instructions executed, CPI is the number of cycles per instruction (for the set of instructions executed), and Tc is the cycle time of the processor. The number of instructions, I, is a function of the benchmarks chosen, and the compilation process. CPI is a function of the processor implementation and the instruction mix. The instruction mix is also a function of the benchmarks and the compilation process. The work reported in this thesis will concentrate on I and CPI in (2) and will assume a fixed Tc.

Tc is largely a function of the physical process used to implement the processor. However, Tc is not a completely independent variable and we should be aware of the dependencies here, even though we will not be able to measure them. Tc can be expressed as:

$$Tc = Tcp + Tcc \tag{3}$$

where Tcp is the portion of the cycle time which can be attributed to the physical characteristics of the logic devices used to implement the architecture, and Tcc is the portion of the cycle time arising from the complexity of the architecture. In general, a more complex function, or architecture, will have a longer cycle time due to longer logic paths and larger fan outs.

Tcp can be reduced by improvements in process technology. This effect is outside the scope of this work; accordingly we will assume that it is fixed. We will assume, however, that increasing the functionality of the architecture, as in more dynamic dependency checking and instruction scheduling, will result in greater complexity in the hardware implementation. This additional complexity will necessarily increase Tcc, adding to the cycle time and slowing down the system. Some of these effects are discussed in [130], [131], [132] and [185]. For this investigation we limit ourselves to noting that Tc, as shown in (3), increases with increasing architecture complexity without attempting to quantify the increase.

In terms of effort involved it would be beneficial to fix the instruction stream by selecting a set of benchmarks and a compiler. With the set of instructions fixed, this would leave only CPI to be measured to determine performance. However, when comparing architectures in conjunction with compiler algorithms, CPI is not an accurate performance metric. This is because the instruction mix is changing as well as the instruction cycles. For instance, replacing low latency instructions with fewer high latency instructions can raise the CPI, even though the total cycles executed may be reduced. For this reason, the total cycles to execute a given program will be used, which is the I x CPI part of (2).

The instruction mix is also effected by the set of benchmarks chosen for the performance evaluation. A wide variety of benchmark suites have been used to study performance. Ad hoc collections of programs have been used, more typically in early studies, as well as groups of programs explicitly designed as benchmarks. The most widely used benchmarks for processor performance are: the SPEC suite [184], the Livermore Loops [113], and for scientific machines: the Linpack Kernels. More recently the use of the PERFECT Club benchmarks [39] is also starting to appear.

This work uses the Livermore Loops for primarily two reasons: 1) The Livermore Loops have been widely reported in the literature and many of the machine architecture studies have been done using some or all of the Livermore Loops; and 2) The compiler scheduling techniques used here focus on loop optimization techniques.

In addition, the Livermore Loops are a small to medium size, relatively simple set of benchmarks to compile. This reduces some of the effort involved in getting an optimizing compiler debugged and running correctly. Focusing on a narrow set of operations and language constructs relieves us of the burden of handling every language construct in the source language and becoming distracted with the implementation details of writing a compiler.

Restricting ourselves to a narrow set of benchmarks also limits the set of applicable optimizations. On one hand this is unfortunate. It would be interesting to make a broad study of compiler algorithms and computer architectures on all types of language constructs. However, by limiting our range somewhat we can examine much more closely the interactions of a particular set of algorithms and computer architectures.

Selecting the Livermore Loops as our benchmark suite is contrary to the current trend of running larger benchmark suites such as the SPEC benchmark suite, the PERFECT Club benchmark suite, or even larger benchmarks. There are certainly some good reasons for running larger benchmarks. Capacity effects such as cache misses and some program behaviors will only show up when running very large benchmarks for very long times [18]. However, our focus is not memory system performance, and for exploring the performance of loop scheduling techniques and system performance, the Livermore Loops are still a viable benchmark suite.

The Livermore suite was developed in the 1970’s to study the code produced by the FORTRAN compilers for the CDC-6600 and CDC-7600 computers. The Livermore suite has since been widely used to compare the performance of numerous computer architectures, particularly high performance architectures. In addition, the Livermore Loops have been used to study and track the performance of different compilers and compiler versions for a single architecture.

A specific goal of the development of the Livermore Loops was to provide a small benchmark suite which would cover the range of commonly used Fortran constructs and provide accurate predictions of computer system performance under actual application loads. The developers of the Livermore Loops realized that the size of a benchmark is a trade-off. A very short benchmark, such as a dot-product function or even the Linpack Loops is not an accurate performance predictor because it does not cover the full range of computer system behaviors. On the other hand, very large benchmarks have problems, especially on new systems where their size makes them unwieldy to analyze, characterize and adapt to a new system. In addition, a large benchmark may not provide any additional performance information because it may spend most of its time in a small part of the code, as Knuth noted in an early paper [86]. For instance, the SPEC benchmark suite typically only exercises about 4000 lines of codes, even though it is much larger in total source lines.

The Livermore benchmark suite is composed of a set of floating-point computations taken from scientific applications. They are intended to cover the range of common Fortran program constructs found in these types of applications. The Livermore Loops have in general shown good correlation between their performance and the performance of the scientific codes they were designed to mimic [113].

The original 14 Loops were criticized because they were found to be more heavily weighted toward vectorizable codes than a typical scientific application. Because supercomputers tend to be vector architectures and have vastly superior performance on vector codes, this would tend to over-predict performance. An additional 10 larger loops were added to the suite to balance the scalar/vector ratio and to “challenge the vectorization capability of Fortran compilers.” Since we are not exploring vector architectures in this study and the original 14 loops were reported to be accurate predictors of scalar architecture performance, we use the original 14 kernels in this study.

Another important question that arises when beginning with this type of investigation is the type of machine to study. We wish to base our study on realistic machines. By this we mean architectures that might be commercially available in the next four-to-five years. Accordingly we have based our studies on multi-issue implementations of the MIPS R3000 ISA. In particular, we have used the Aurora III for a case study. The Aurora III is a prototype superscalar processor being developed in the Advanced Computer Architecture Laboratory at The University of Michigan [119][120][130][131] [132][133][185]. The Aurora III is a superscalar version of the MIPS R3000 ISA implemented in GaAs technology, and is scheduled for tape-out in the Fall of 1994.

We investigate the performance characteristics of a number of hardware features in the Aurora III, only some of which have been included in the final version. In order to investigate the performance of processors with different instruction issue policies, we construct several modified models of the Aurora III. Using the Aurora III as a base architecture, we build scalar, superscalar, VLIW, ad DAE processors, schedule code tailored to each architecture, and examine the performance of the combined compiler-processor system.

3 Research Contributions

We explore the performance of a set of scheduling algorithms applied to a set of machine architectures. We compile a set of benchmarks using block scheduling, loop unrolling, and software pipelining and evaluate their performance on scalar, VLIW, DAE and superscalar architectures.

In addition to evaluating the performance of different scheduling techniques, we look at the analysis and compiler support required to implement the scheduling techniques effectively. We also explore the machine independent optimizations required to obtain a good optimizing compiler. Some new algorithms for induction variable analysis and corresponding optimizations are presented.

We examine the interaction between operation latency and the scheduling technique employed. We also look at the effect of pipelining function units on performance and scheduling.

We look at the effects on scheduling and overall performance of some hardware features proposed for the Aurora III, a superscalar prototype being implemented in GaAs technology at The University of Michigan. The Aurora III incorporates internal 64 bit wide data paths, double precision floating point load and store instructions, dual instruction issue, decoupled integer and floating point units, fully pipelined function units, load queues, store queues and result reordering. We look at each of these features in turn and the effect of each feature on performance and scheduling.

We also examine instruction issue policies in the context of the Aurora III. Using a fixed set of function units and architectural components, we vary the instruction issue policy to model scalar, VLIW, DAE and superscalar architectures. The issue policy affects register naming and use, which in turn affects the compiler scheduling and register allocation algorithms. We examine the relationship between these features and algorithms and discuss some of the implications.

Different schedules use machine resources different ways. Registers and function units are obvious resources. Instruction and data cache are also resources and may be allocated by a compiler. We briefly investigate the effects caused by interaction of the scheduling algorithms with the cache and memory systems.

4 Thesis Organization

Chapter 2 presents previous work pertinent to this investigation. This includes studies on the amount of parallelism available in typical programs and an overview of a number of computer architectures designed to exploit instruction level parallelism.

Chapter 3 examines several scheduling techniques that have been used in high performance systems. Particular emphasis is given to loop scheduling techniques.

Chapter 4 discusses the internal workings of the Tortoise compiler developed for this investigation. The specific techniques employed and the justification for the techniques are also given.

Chapter 5 describes the experiments performed for this investigation and discusses their results.

Chapter 6 gives concluding remarks and suggestions for future work.

CHAPTER II
INSTRUCTION LEVEL PARALLELISM

1 Available Parallelism Analysis

Determining the amount of parallelism available in typical programs is one important aspect of the work on instruction level parallelism. The amount of parallelism available in a program is important to systems designers because it determines how much parallelism we should be attempting to achieve when designing a machine to exploit instruction level parallelism. If programs generally provide parallelism on the order of 10,000 operations per cycle, we would design a very different machine than if the available parallelism was closer to 10 operations per cycle.

Estimates of available parallelism range vary widely, depending on many factors, and the terminology used to describe parallelism also varies widely. Parallelism can be expressed as operations per cycle, instructions per cycle, FORTRAN or other high level language (HLL) statements per cycle. Some studies invert the this relation and express results in cycles per instruction (CPI). Other studies avoid the problem of defining an abstract metric and use speedup with respect to a base architecture. For small benchmarks, where the absolute best performance is known or can be found, *efficiency*, expressed as a percentage of the ideal performance, may be used. There is no standard metric for parallelism, the terminology used is determined by the goals and methodology being used by each research group.

We will not attempt to reconcile the terminology used in reporting the results of the studies examined here. We will report the results of each study using the author's terminology. Our intention is not to find a definitive value for the amount of parallelism present in general programs, but rather to show the general range of what other people have found when studying this issue.

There are some major tends in the study of available parallelism: The first is to study available parallelism given a particular model or device designed to exploit it. These studies tend to find relatively small amounts of available parallelism, with speedups on the order of 1.1 times to 10 times sequential machines. The second type of study is to examine the parallelism inherent in the execution of a program, without regard to an implementation which could exploit this parallelism. A third approach is to find the largest amount of parallelism available in any program. As could be expected, the last two types of studies tend to find much larger amounts of parallelism than the studies of parallelism on a particular machine.

The experiments which are not tied to an architectural model generally execute a program and save a trace of the instructions executed. A directed acyclic dependency graph (DAG) is then constructed from the trace. The DAG is examined to find its height and width. The height of this DAG divided by the number of operations in the sequential program is the speedup. The width of the DAG divided by the height is the average parallelism, and the maximum width of the DAG is the maximum amount of parallelism.

Studies using this methodology conclude that programs contain a large amount of parallelism. An early study by Kuck [91], finds a minimum of 16 processors useful and,

> “As the programs become more complex, 128 or more processors
> would be effective in executing our programs.”

However, this was an early study which ignored the problems presented by conditional code. The benchmarks used are very small by today’s standards, most “less than 200 cards,” and many “do not even contain DO-loops.”

In a study of very large benchmarks using parallelism-time profiles for programs [92], Kumar shows that the amount of parallelism varies widely during the course of execution. Both the ideal case of full knowledge of control and data dependencies, and the case where control and data knowledge is restricted, show approximately the same amounts of parallelism. Average parallelism was shown for the particular benchmarks studied, to be on the order of “500-3500 FORTRAN statements executing concurrently in each clock cycle.” In an experiment with restricted knowledge, the amount of parallelism was reduced by a factor of 10, but was still as much as 1000 FORTRAN statements per cycle.

The abstract studies on parallelism are encouraging because they indicate that common programs do have large amounts of parallelism. The problem is that it is not practical to exploit a large part of this parallelism. In a study by Wall [189], an execution trace was produced and used to find the amount of parallelism available under various machine models and software techniques. This study finds a large gap in performance between “perfect” and “good” techniques:

> “Our study shows a striking difference between assuming that the techniques we use are perfect and merely assuming that they are impossibly good. Even with impossibly good techniques, average parallelism rarely exceeds 7, with 5 more common”

Riseman and Foster found the same results [150]:

> “In fact, our results seem to indicate that even very large amounts of hardware applied to programs at run time do not generate hemibel [a factor of 3] improvements in execution speed. We are left, then, with three alternatives: extensive preprocessing of programs as suggested by Kuck et al; recasting algorithms to take advantage of machine parallelism as, for example, in the Goodyear STARAN or the Illiac IV; or just plain speeding up a conventional monoprocessor so it gets the job done faster.”

So while the abstract studies showing large amounts of parallelism are encouraging, it seems that the speedups which can actually be achieved will be modest.

In another similar study [12], Austin and Sohi use the MIPS program pixie to produce a trace, then the trace is fed into a dependency analysis tool. The dependency graph of an entire trace can then be used to investigate critical path length and parallelism profile. The average parallelism is found to be between 13 and 23,000 operations per cycle for the SPEC suite. Much of this parallelism is available only after renaming registers, and with an instruction window of almost the entire program. A small window with approximately 100 instructions only finds 10 to 20 operations per cycle.

The most aggressive experiments tied to architectural models provide for speculative execution with out-of-order issue and completion, register renaming and memory by-pass subsystems. In [59] Franklin and Sohi examine an architecture providing all these features and predict a 2 to 7 times speedup. Other studies of this type of system [114][124][182] indicate similar speedups.

The problem with these architectures is that they are large and complicated to implement. In [182], Uht estimates over a million gates just to implement the instruction window and ordering matrices. This brute force approach to exploiting instruction parallelism may not be the best means of providing performance. More complexity means a longer design and test cycle, and a potentially slower cycle time. These conflicting issues must be traded off against the gains from parallelism.

This is not to say that dynamically scheduled architectures are impractical. More modest designs using either instruction windows or Tomasulo style reservation stations [179] have been quite successful. Notable examples are the IBM 360/91 [10] and the RS/6000 [76]. The studies on this type of machine generally show that it is possible to achieve modest speedups with a small instruction window. Flynn reports to find most of the available parallelism with a window size of 2 or 4 instructions in [56]:

> “Under the constraint that instructions are not dispatched until all preceding conditional branches are resolved, stack[1] sizes as small

> as 2 or 4 achieve most of the parallelism that a hypothetically infinite stack would."

If control dependencies are strictly enforced, so that speculative execution is not allowed, speedups over scalar machines are modest. Without speculative execution, speedups tend to be a sub-linear function of the window size and performance benefits disappear for general benchmarks at window sizes in the range of 4 instructions. The average parallelism is limited to something on the order of 2 operations per cycle. Foster also reports the same modest amount of parallelism in [58]:

> "The limit on the parallelism that is achieved with an infinitely large stack was found to be slightly more that 1.72;..."

In [177], Tjaden and Flynn explored the parallelism provided if an instruction window was added to an IBM 7094. They found speedups between 1.2 and 3.2 times scalar on their benchmarks. In [178], Tjaden and Flynn try several versions of ordering matrices to encode instruction dependencies, including one with shadow buffers to provide some speculative execution. They find parallelism between 1.36 and 1.98 instructions per cycle (IPC).

In [138], Pleszkun and Sohi start with a set of Cray function units and study the effects of adding register renaming and multiple issue. Control dependencies were required to be resolved before dependent instructions could execute. They found that the best issue rates that could be achieved with the given set of function units were in the range 0.79 to 3.15 IPC.

In [80], Jouppi and Wall use a compiler and a machine simulator for the "MultiTitan," to explore superscalar and super-pipelined execution. The compiler and simulation system allowed the machine's function units and operation latencies to specified. The compiler provided basic block scheduling and loop unrolling. The authors found lit-

[1] They used the term instruction stack to refer to the buffer we are calling an instruction window.

tle difference between superscalar and super-pipelining, and an IPC limit of approximately 2. Furthermore, they show that for these types of machines, more parallelism in the form of additional function units is not useful. They also show a decrease in available parallelism when compiler optimizations are applied. They make an interesting comment: Cache misses impose a larger penalty for multi-issue and other parallel machines. This is because the number of instructions lost is magnified by the width of the instruction window.

Smith, Johnson and Horowitz study the available parallelism for a superscalar MIPS architecture in [161]. In this study, trace driven simulations were used to find the parallelism for variations of the MIPS architecture, including superscalar versions. The benchmarks used were non-scientific code, i.e. avoiding the Livermore Loops. They start with code optimized for the R2000 in this study. Pixie is used to generate instruction traces and a simulator is used to analyze the traces for the different machine configurations. A number of machine features are tried, along with superscalar execution: Register renaming, perfect branch prediction, 2 instruction or 4 instruction wide fetch and decode units, infinite instruction windows and fixed size instruction windows of up to 32 instructions were tried. Tomasulo style execution units with reservation stations are also used.

With an unlimited prefetch buffer, and an instruction window size of up to 32 instructions, they find speedups of 2.3 to 4.1 for prefect branch prediction and register renaming. This drops to a mean speedup of 2.0 with a branch prediction accuracy of 85%, with a prefetch buffer of 4 instructions.

The instruction window architecture requires a large number of busses and register ports. This can be mitigated by using Tomasulo style execution units and reservation stations. With reservation stations and prefect branch prediction, the speedup falls to 1.2. A 1 cycle delay for taken branches and a fetch decode width of 2 or 4 instructions, puts the speedup at approximately 1.3 for a 2 instruction window or 1.9 for a 4 instruction window. With branch prediction, the speedup is in the range of 1.6 for a 2 instruction window and 2.6 for a 4 instruction window.

As has been shown in these studies, the problem with superscalar architectures is not that they are impractical or that they fail to provide speedups. However, there is a potential problem with superscalar architectures: their complexity. The question is whether the additional complexity of a superscalar architecture outweighs the potential parallelism exploitable by these architectures.

In addition, there is a question of how well superscalar architectures work “off the shelf.” One reason these architectures are so popular is that they promise performance improvements running existing software and *without compiler support*. Superscalar architectures typically implement the same instruction set as some existing scalar architecture. Parallelism is detected and exploited by the hardware. This is a major advantage in the commercial world because a company producing a new superscalar processor would not have to also provide new compilers or other software.

However, while it is true that existing code will run on the new architecture, compiler support may be necessary to achieve enhanced performance. In a study of superscalar SPARC architectures [102], Lee et al. report:

> “Unfortunately, an optimal scheduling policy is very hardware dependent. The base compiler we use was not targeted for superscalar hardware, and most of the optimizations must be applied manually. Our results confirm that superscalar hardware alone would gain little without support from an optimizing compiler.”

Given that compiler support will be required to achieve maximum performance from an architecture, the natural question is whether better performance can be achieved using a less complicated architecture with compiler support.

2 Machine Architectures

Many varieties of machine architectures have been designed to exploit instruction level parallelism. The architectures presented here are designed to execute a few, e.g. less than ten, operations concurrently. The primary difference between these architectures is in how the operations to be executed each cycle are specified and/or discovered. One end of the spectrum is the superscalar machines like the IBM 360/91 and the RS/6000 [9][76] where the dependencies between instructions are resolved by the hardware. On the other end of the spectrum are the VLIW architectures [36][144] where parallelism between instructions must be discovered and specified by the compiler. Somewhere in between are the DAE architectures [156] where compiler support is used to provide the hardware with dependency information.

The basic architecture against which the other, more parallel architectures are compared, is the pipelined scalar architecture. This architecture is chosen as the basis for comparison because it is typical of today’s general purpose commercial computers. Instances of this architecture, such as the MIPS R3000, are readily available for use in these experiments. Comparison against a strictly scalar architecture, i.e. one without pipelining, would show better speedups, but such an architecture is sub-standard by current market driven criteria. Also, any architecture which employs parallelism in the form of a wider instruction path can also employ pipelining. In other words, pipelining is one type of parallelism, which has already been accepted and is in wide use in the computer industry.

In a pipelined scalar architecture, instructions are divided into several stages, where each stage performs one simple operation and requires one machine cycle to execute. One instruction is issued every cycle and the execution of the stages of different instructions is overlapped in time. A typical set of stages is: fetch, decode, issue, execute, and write. A functional block diagram of a pipelined processor is shown in Figure 2.

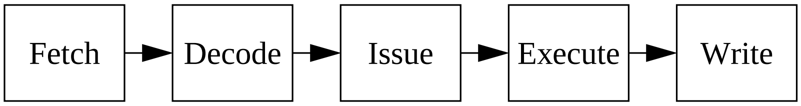


FIGURE 2. Block diagram of a pipelined scalar processor

A Very Long Instruction Word (VLIW) architecture looks functionally similar to a scalar processor. The difference is that each instruction can specify multiple operations. The stages of a VLIW processor operate in lock-step, including the execution stages in the function units. This means that if any of the stages stall, the entire processor stalls. A block diagram of a VLIW processor is shown in Figure 3.

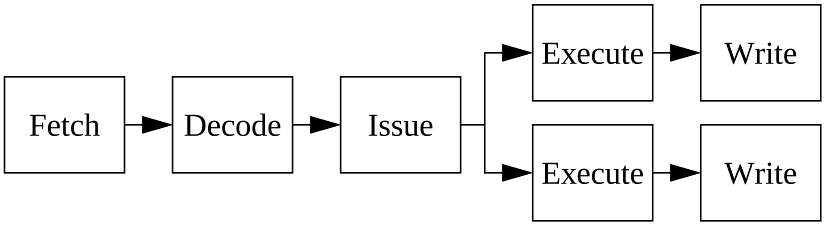


FIGURE 3. Block diagram of a VLIW processor

Superscalar architectures can also issue several operations each cycle. A superscalar architecture fetches and issues multiple instructions each cycle, where each instruction contains one operation, as in the scalar architecture. The instructions to be issued are selected from an instruction window and each cycle a superscalar architecture can examine at least as many instructions as can be issued. Every instruction in the instruction window is compared with every other instruction in the window to search for dependencies. The set of instructions which do not have dependencies and for which there are resources available are issued.

The functional diagram for a superscalar architecture looks very similar to the diagram of a VLIW architecture. The major difference between superscalar and VLIW architectures is that superscalar architectures check for dependencies between instructions in the instruction window and can issue instructions out of static order. To do this, superscalar architectures must provide some mechanism to track and control the out-of-order execution so that static semantics are maintained. A number of data structures have been used to maintain dependency information, include ordering matrices [178], register scoreboarding [174], and reservation stations [179]. Figure 4, shows the block diagram of a superscalar processor.

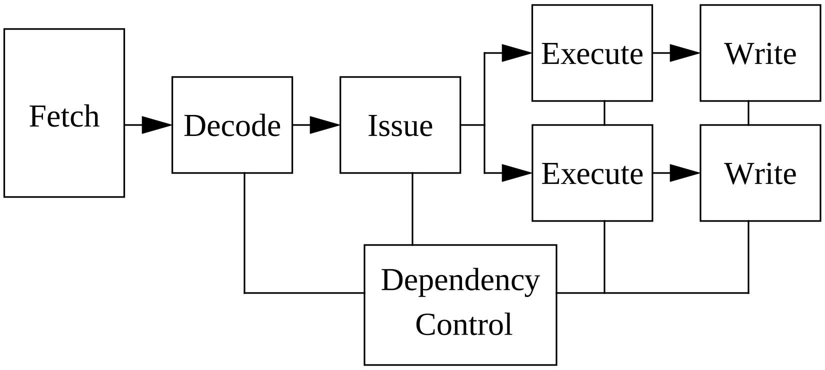


FIGURE 4. BLock diagram of a superscalar architecture

Ordering matrices are the most general mechanism for maintaining dependency information. Ordering matrices are hardware structures encoding the dependency relationship between all executing and pending instructions.This is a very explicit representation of dependency information, but it is also costly to implement. For this reason, more compact representations have been designed.

Register scoreboarding associates dependency information with the register set. The target register for each instruction is marked busy from the time the instruction issues until the result has been written to the register file. Instructions attempting to access a register are blocked while the register is busy. This is a very compact and efficient way to represent dependency information.

Another dependency representation is implicit in the use of reservation stations. Reservation stations are pending instruction queues placed in front of each function unit. An instruction enters a reservation station after its instruction window dependencies have been resolved. There are still register dependencies, as in scoreboarding. However, in a system with reservation stations, results can be directly forwarded to the reservation stations, rather than going through the register file.

A superscalar architecture may keep a larger set of instructions in the instruction window than it can simultaneously issue. This is done so that the probably of finding instructions to execute in parallel is increased. Dependencies between instructions may be resolved at the decode and issue stages, or there may be some mechanism for maintaining dependency information within the execution stages.

VLIW processors can track dependency information too. Pipeline hazards may be checked in a VLIW processor. If a hazard is found, the entire pipeline would stall. However, VLIW processors do not allow out of order issue so the control logic is simpler.

Decoupled Access/Execute or just decoupled (DAE), architectures fall somewhere between VLIW and superscalar architectures in terms of their dynamic behavior and complexity. Like superscalar architectures, DAE architectures fetch multiple instructions each cycle. The difference is that DAE instructions are split into separate instruction streams and the static execution order is maintained only within each instruction stream. The processors can only communicate with each other through a set of hardware queues. Dependency relationships between the instruction streams are specified by the queue operations encoded in the streams and enforced by the queue hardware. This logic is less complex than the superscalar implementation because all the pending instructions do not have to be checked for dependencies. The only dependency checking required is whether the queues are full or empty. A block diagram of a DAE architecture is shown in Figure 5.

The idea behind the DAE architecture and the reason for its name, is that there will be two instruction streams: the Access stream and the Execute stream. The A processor (Access) will perform address calculations to deliver addresses to the memory system. The E processor (Execute) will use the data from the memory system to execute the program. Hopefully, the Access processor will run far enough ahead of the Execute processor so that the memory latency is hidden.

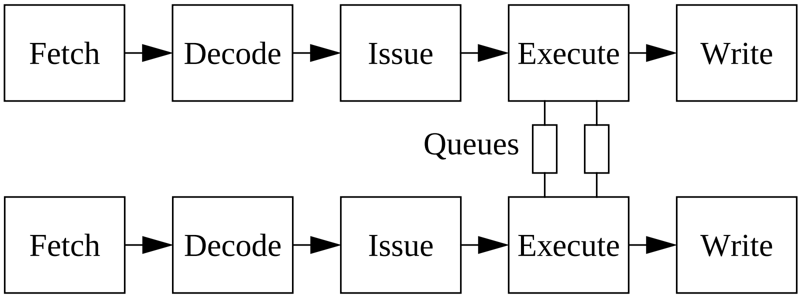


FIGURE 5. Block diagram of a DAE architecture

A range of architectures and features is available to the systems designer with respect to instruction issue and dependency control. One extreme is the VLIW architecture where little or no dependency checking and control is performed. With a VLIW architecture the onus is on the compiler to discover instruction level parallelism and schedule the instructions accordingly. The other extreme is the superscalar architecture where the hardware actively searches for parallelism between instructions. In between these extremes is a continuum of architectures, each reflecting different choices made about what the compiler should do and what hardware should do.

2.1 VLIW Architectures.

VLIW (Very Long Instruction Word) architectures grew out of work done on horizontally microcoded processors. In a microcoded architecture there is generally more parallelism available at the microcode level than there is in the instruction set. In a desire to gain access to this additional parallelism, some machines, such as floating point systems AP-120b and FPS-164, were designed to be programmed directly in horizontal microcode [26]. This leads to greater performance but at the cost of an architecture which is more difficult to program.

The difficulty of programming horizontal architectures has been attacked on two fronts. On the hardware side, the instruction set was made more regular. Extra data paths were proposed to remove arbitrary constraints and hardware idiosyncracies. This produced the first VLIW architectures [52][142]. VLIW machines are characterized as being able to execute multiple operations each cycle from one instruction, where each operation is similar to what would be found in a scalar processor. In other words, the low level hardware details, such as register file bypass, are hidden from the instruction set architecture, just as they would be in a scalar architecture. As the same time, a VLIW architecture is still able to issue multiple operations per cycle, as in a microcoded architecture.

On the software side, algorithms were proposed to allow code to be efficiently compiled for horizontal architectures. Trace scheduling [52] and software pipelining [142] were developed for these architectures. The small block size found in typical programs is an even more severe problem for VLIW architectures than for scalar architectures. Both trace scheduling and software pipelining attempt to alleviate this problem by scheduling operations across block boundaries. These scheduling techniques are key to achieving performance in a wide architecture. They are discussed further in Chapter III.

How well do VLIW architectures perform? Early work indicated great potential for VLIW architectures. In [126], Nicolau and Fisher found available parallelism for a VLIW from 3 to 988 times scalar. However, this study assumed perfect branch prediction, which leads to optimistic performance predictions.

On more realistic models, Ellis finds speedups up to 7.4 on using the BullDog trace scheduling compiler on a simulated VLIW, the ELI-512 [46]. In [35], Colwell et al. find speedups between 1.0 and 4.4 times scalar using a trace scheduling compiler on a single node Warp with a perfect cache.

Sohi and Vajapeyam provide an extensive study of VLIW architectures in [167]. They start with the assumption of 5 function units, integer alu, integer multiply, fp addition, fp multiply and memory, and vary the instruction width between one and four operations. They perform this experiment for both “modest” and “deep” pipelining.

They compile the first 14 Livermore Loops using loop unrolling. They find a good speedups for an architecture which can issue two operations per cycle: 1.57 times scalar for modest pipelines and 1.38 times scalar for deep pipelines. More operations per cycle are found to be not as useful. Performance in a system with two operations per instruction is found to be only 15% below the maximum obtainable performance. A third operation per cycle only adds 10% to the performance of a two operation per cycle system.

A constrained instruction format allowing one integer and one floating point operation, but not two operations of the same type, has less performance improvement than allowing arbitrary operations: 1.2 times scalar for modest pipelining and 1.13 times scalar for deep pipelining. This is a fairly modest performance improvement. However, the constrained instruction format allows a much simpler register file and bus format, which would be less costly to implement.

A study by Love, comparing a VLIW and a DAE architecture, found the two comparable in performance [105]. The benchmarks were a mix of small, hand compiled programs and this study begs the question of what effect the compilation process would have on performance.

The line between a VLIW and a superscalar architecture can be somewhat vague. To reduce the code size, the actual implementation of VLIW architectures can provide instruction formats to allow unused operation slots (NOPs) to be left out of the object code [167][36][35]. This decreases the size of the object code, but adds decoding overhead. The next step, somewhere between VLIW and superscalar, is the static superscalar Torch architecture, described Smith et al. in [162]. Torch executes instructions in the static order determined by the compiler. The architecture allows access to a set of shadow registers and buffers, allowing the compiler to speculatively schedule instructions across conditionals. Simulations of the architecture show performance in the 1.4 to 1.6 times scalar, even with a limited scheduling algorithm in the compiler. This compares favorably with perhaps 1.5 to 1.9 times scalar performance for a dynamically scheduled superscalar.

2.2 DAE Architectures

Decoupled Access/Execute (DAE) Architectures lie in complexity somewhere between VLIW and superscalar architectures. DAE architectures execute two or more instruction streams in a loosely coupled or decoupled fashion. The two instruction streams are independently executed and dependencies are only explicitly checked and enforced within an instruction stream. The streams are synchronized by the use of queues, which provide communications between the sub-processors.

The intent behind the DAE design is that one instruction stream will compute addresses which are delivered to the memory system (the Access processor) and the other instruction stream will use the data delivered from the memory system to execute the program (the Execute processor). Given a nicely behaved program with no recurrences or other dependencies between the two streams, the Access instruction stream will execute ahead of the Execute instruction stream. If the Access stream is far enough ahead of the Execute stream, the memory latency is entirely hidden. The two instruction streams can each execute at their own maximum rate and memory delays are removed from the schedule. According to James Smith in [156]:

> "The [DAE] architectures discussed in this paper permit improved scalar performance in two important ways. First, the Flynn bottleneck is sidestepped by using two instruction streams. This effectively doubles the maximum available instruction bandwidth. Second, because hardware queues are used for communication between the instruction streams, the streams can "slip" with respect to each other. This leads to what is essentially dynamic scheduling of instructions, previously provided only by the sophisticated issue methods used in the CDC 6600 and IBM 360/91. Moreover, the instruction issue logic used in each instruction stream remains simple."

The DAE architecture in [156] used the scalar part of the Cray instruction set as the starting point for the definition and modified it by adding queues and queue branch instructions [154][155][156]. Using the Cray instruction set allowed the Cray Fortran compiler to be used to generate code with only minor modifications.

A DAE architecture does provide improved performance over a scalar processor. In simulations comparing a DAE with a scalar Cray architecture, Smith et al. find that the DAE architecture shows a 1.5 mean speedup over a scalar Cray processor [157]. They also find, by increasing the memory latency, that the DAE machine is less sensitive to memory latency than the Cray-1. In a simulation of some of the Livermore Loops, the vectorizable loops show no memory effects. The non-vectorizable loops show memory effects equivalent to the Cray.

Smith’s DAE architecture was eventually implemented as the Astronautics ZS-1 [158][159]. An interesting note is found in [160]. A cache was not part of the DAE architecture, but was added to the Astronautics “late in the design cycle.” Apparently even though the access processor can execution in front of the execute processor to hide the memory latency, there was still a bandwidth problem of supplying enough instructions and data from memory to support a dual issue processor.

Smith and Kaminski discuss some other architectural trade-offs in [155]. In particular this paper discusses DAE machines with both combined or split instruction streams. They address the question of how early the streams are to be split. Designs with an early instruction split can have separate instruction streams all the way from the I-cache. The implementation in [155] uses separate caches and I-fetch units. It is also possible to divide the instruction streams later, after the I-fetch unit. The Astronautics ZS-1 was implemented with combined, i.e. late split, instruction streams.

In [109], Mangione-Smith, et al. study the performance of the Astronautics ZS-1. They develop an ideal performance model for vector and scalar loops, based on the available machine resources. They find that the Astronautics ZS-1 performance is between 60% and 80% of the ideal for vector loops and 90% of the ideal for loops with linear recurrences. They show the effects of memory latency and how allowing “slip” between the instruction streams can mitigate the effects of memory latency.

The Astronautics ZS-1 was not the only DAE architecture to be implemented. The MAP-200, by CSPI was an earlier DAE machine [34]. The MAP-200 contained two decoupled, wide processors. Each of its two processors could execute two operations per cycle, so an ideal speedup would be a factor of 4. In [34], Cohler and Storer found speedups between 1.9 and 2.9 on a small set of benchmarks.

Another DAE architecture is the PIPE [62][48][49]. This is an interesting variation because this is a symmetric architecture, i.e. both processors implement the same instruction set.

The PIPE was specifically intended to be implemented on a single chip using VLSI technology. Like the Berkeley RISC, the implementation was severely constrained by the technology of VLSI at the time of its implementation. PIPE was implemented as a 16 bit machine, with 16 registers and a 16 word I-cache. Floating point operations were performed by an off-chip co-processor.

Only a single processor version has been implemented. Since the DAE mode was not implemented, no performance results for DAE execution are available. However, the PIPE studies did show an interesting result which is applicable to pipelined and VLIW architectures. In [48], Farrens found that padding shorter operations so that all function units had the same latency produced shorter schedules than a variable latency implementation, because of contention for the result bus. This result was valid up to a latency of 4 cycles.

DAE architectures can be usefully extended to more than two processors. In [13], Benitez and Davidson propose adding a “vector execution unit” to the WM architecture. They identify linear access expressions and execute these expressions on the vector execution unit, which is a third execution unit separate from the access unit. They report performance improvements of 1% to 43% for a small set of benchmarks.

In a more ambitious use, decoupling is used on the Warp to make programming a systolic array easier [11][98][35]. This is a little different from the DAE architectures in that the queues between the processors are directly accessible to the user in the W2 language implemented for Warp.

2.3 Superscalar Architectures

The term *superscalar* is commonly used by the architecture community to refer to dynamically scheduled architectures which maintain scalar semantics and can issue more than one scalar instruction per cycle [80]. That a superscalar architecture maintains scalar semantics while issuing multiple instructions every cycle is probably the major reason for their popularity: A superscalar version of an architecture can improve performance of that architecture, while allowing existing programs to be run *without modification*. For a hardware company with an established customer/software base, this is an insurmountable argument to develope a superscalar architecture. Most computer manufacturers are producing, or will produce a superscalar machine in the next few years.

The defining feature of a superscalar architecture is the ability to issue multiple instructions each cycle. The processor must be able to fetch and examine multiple instructions for possible issue each cycle. This is accomplished via an instruction window and a wide path, to carry multiple instructions to the instruction cache. Each cycle, instructions are fetched into the instruction window. Each cycle the instructions in the instruction window are examined for dependencies and resource conflicts to determine which instructions can be issued.

To maintain an issue rate of greater than one instruction per cycle, the processor must also be able to execute and complete multiple instructions per cycle. Multiple execution units are generally present in scalar processors, so there is no additional cost associated with executing multiple function units. However, the ability to write multiple results to the register file is *not* usually present in scalar architectures and adding extra ports to the register file can be expensive. Extra read ports are also required to supply the execution units with the extra operands required each cycle. The register file congestion can be reduced by splitting the register file, at the cost of some performance, or by providing a buffer to accept results from the function units and funnel the results to the register file. As reported by Upton, et al. in [185], not every instruction will require a result to be written to the register file, and a result buffer can allow a register file with a single write port to keep up with a multiple issue architecture.

The problem of routing multiple results to the register file is one example of a general problem in multi-issue architectures. Because multiple instructions can produce multiple results going to multiple destinations, routing the data between sources and sinks is also a complex problem. Every place where multiple results can appear, e.g. the writeback port to the register file, requires a multiplexor to determine which source has produced a result and direct it to its proper destination. This is a much more complex operation than just moving the data. In addition, multiplexors are relatively costly structures in terms of area.

The required dependency checking is another fundamental problem with superscalar architectures, one which requires complex hardware to resolve. Because scalar semantics must be maintained, all the instructions currently being executed and all the instructions being considered for execution must be compared to determine potential conflicts. This means that all the instructions in the execution units and in the instruction window must be checked for dependencies. The complexity of this operation grows as a quadratic function of the number of instructions to be checked. The amount of hardware required to perform this check in a single cycle quickly becomes unwieldy.

A general approach for expressing and resolving instruction dependencies is to use ordering matrices. Tjaden investigates the use of ordering matrices in [178]. This data structure succinctly captures the relationships between instructions, but the implementation of ordering matrices requires a large amount of hardware. In [182], Uht estimates a cost of one million gates to implement an issue stage with a “reasonable” size window, e.g. 32 long by 8 issue ports. These types of issue mechanisms, i.e. large register widows with arbitrary dependency checking, have so far only been studied via simulations and have not been implemented, because of the hardware cost. The “cost” of dependency resolution hardware is not just in terms of chip area. The issue stage is likely to be in the critical path of the processor and large amounts of hardware at this point will also slow down the cycle time.

The high cost of the dependency checking has been mitigated somewhat in recent superscalar designs by reducing the number of instructions in the instruction window and/or only performing a partial dependency check. The instruction window in the next round of superscalar designs seems to be on the order of four or eight instructions [76][97][102].

The dependency checking complexity has also been reduced by restricting the type and number of dependency checks performed. One example of a simpler dependency structure is a register scoreboard, first used in the CDC-6600 [175], and more recently in the Motorola 88000 processor [117]. In a scoreboard, a bit is associated with each register. A register is marked “busy” if it is the target register for an instruction currently being executed. Instructions are blocked from execution if any of their registers are marked busy. Execution of other instructions is allowed to proceed. A register scoreboard is a relatively simple structure to implement and effectively maintains dependency information.

One well known method of supporting dynamic instruction scheduling are reservation stations with register renaming. This architecture was first used in the IBM 360/91 [9]. More recent proposals for this type of system are found in [79] and [70]. In a system with reservation stations, each function unit maintains a queue of instructions waiting to execute (*reservation stations*). The instructions in the reservation stations are ready to execute when the execution unit is free and all of their operands have arrived. Moving the instructions which are ready to execute out of the instruction window removes some of the congestion at the instruction window and simplifies its functionality.

In a Tomasulo architecture, dependencies between the instructions are tracked through a register renaming and forwarding mechanism [179]. Register renaming allows instructions with output (write-write) dependencies to execute simultaneously. Operand forwarding sends results from completing instructions directly to instructions waiting for the operands in the reservation stations. This relieves congestion at the register file, but requires a bus which can broadcast results to the reservation stations.

Another method of simplifying the task of instruction dependency resolution is to introduce instruction categories [187]. In this method instructions are grouped into sets (categories) which cannot interfere with each other during execution, e.g. integer and floating point instructions. Instructions within a category execute sequentially. Dependency resolution only requires checking the categories of the instructions in the instruction window, which requires less decoding than finding and checking all the registers on all the instructions in the window.

While the more aggressive superscalar architectures purport to provide large performance gains, they do so at the cost of additional hardware. The increased complexity of the hardware has several problems:

1. It becomes more difficult to design and implement the architecture.
2. The amount of hardware required by a superscalar implementation can make the design too large to be implemented on a single chip, introducing delays due to chip boundary crossings.
3. Testing becomes more difficult.
4. More complex hardware may slow down the clock cycle, mitigating any performance gains due to increased parallelism.

Because of the difficulties of implementing the complex hardware required for large superscalars, i.e. superscalars with large instruction windows, large reservation stations, and complete dependency checking and resolution; actual implementations have been considerably scaled back. Depending on the implementation, the speedup provided by a realistic superscalar architecture may become quite modest: Smith et al. report a 1.2 speedup over scalar in [164], Mahlke, et al. report 1.6 times scalar in [108], Lee et al. report 2.2 times scalar for a 4 instruction window and 1.7 times scalar for a 2 instruction window in [102]. Given these comparatively modest performance results, it seems important to ask whether a simpler architecture would not perform as well or better by allowing the clock cycle to be pushed further than is possible in a corresponding superscalar architecture.

2.4 Memory System Support

Some recent work on a memory system for a MIPS superscalar architecture has highlighted the problem produced from combined increasing processor performance with realtively decreasing memory performance. In [168], Sohi and Franklin show that a traditional blocking memory system with a 10% miss rate delivers a throughput of just 0.4 references per cycle. This throughput is enough to supply a scalar load-store processor, which would have a one instruction per cycle issue rate, and typically needs a throughput of 0.25 to 0.4 requests per cycle. However, this throughput will starve a superscalar processor, which will be attempting to issue more than one instruction per cycle. If a 1 IPC issue rate requires 0.4 requests per cycle, a 2 IPC issue rate would require 0.8 requests per cycle. If the memory system can only service 0.4 requests per cycle, the sustained issue rate drops to 1.67 IPC.

A similar result is shown in a study of superscalar SPARC architectures [102]. In this study, the performance improvement of a 4-scalar (a 4 instruction window) versus a scalar architecture drop from 2.0 times on a system with an infinite cache to 1.3 times on a system with a finite cache. This result was with a 128K byte direct-mapped cache with a 32 byte block. Given the drop in performance improvement, the memory system appears to be a major factor limiting performance in this system.

This problem of limited memory throughput has been attacked in several ways. Recently there has been interest in non-blocking loads, speculatively executed loads and prefetch instructions. Non-blocking loads allow multiple loads to execute until the result of an undelivered item is required. An early version of this type of system was designed for CDC Canada [89]. In [168], a non-block cache memory system of this type with a maximum of 4 pending loads and 8 ports was able to remove most cache stalls.

Non-blocking loads help, but they are limited in how far back in the schedule they can be moved by the small block size found in most programs. Loads are instructions which can raise exceptions. Attempting to load from a page which is not resident in memory, or out of the programs memory space will cause an exception to be raised. The conditional instruction which the load would cross is often a guard for the load instruction, i.e. the conditional determines whether the load *should* be executed. Generally, to be able to move loads past block boundaries, speculative capabilities such as delayed exceptions are required. Delaying the exception from the load until the use of the result of the load allows specious exceptions to be squashed. The MultiFlow architecture [36] had this feature. Details of it effectiveness are not available.

A study [151] adding speculative loads to the MIPS architecture shows good results for benchmarks with large data sets. In this study, speculative loads bypass the cache, going directly to memory. Executions with small data sets perform slightly worse with this system than if speculative loads were not used. This is because when the data sets fit entirely in the cache, some performance is gained due to reuse of data in the cache. Since the cache is entirely bypassed, the data is not available for reuse. However, performance improvements were reported for benchmarks where the data set does not fit in the cache. One advantage of speculative loads over prefetch instructions is that speculative loads do not consume additional instruction bandwidth.

Prefetch instructions are non-blocking, non-exceptional instructions which provide a hint to the memory system that a data item will be used soon. Callahan, et al. implement prefetch instructions in [23]. In this study, prefetch load instruction were provided along with standard loads. Both load instructions put data into a single unified cache. A compiler prepass was used to add prefetch instructions to the source code. A prefetch load was added for the following loop iteration to every simple array reference in an inner loop, i.e. references which make direct use of the loop induction variable. They report a 20% improvement for a 50 cycle memory, but with an estimated overhead of 28% for executing prefetch instructions and address calculation.

In [85], Klaiber and Levy add prefetch instructions to loops using a simple algorithm. Their prefetch instruction loads into a prefetch cache, which is separate from the normal load cache. This prevents the prefetch from interfering with normal load.

In [28], Chen and Baer study a system which includes both a prefetch mechanism and non-blocking loads. In this case the prefetch is provided via a hardware prediction mechanism. This has the advantage that extra instructions are not required. However, the prediction is not as accurate or general as a software mechanism could provide. They note that both prefetch and non-blocking loads are useful: “Prefetch instructions exploit pre-miss parallelism and non-blocking loads exploit post-miss parallelism.” They show that a combined approach has the highest performance.

In [27], Chen et al. use a combined software/hardware strategy handle load latency. Speculative loads are used to remove as much latency as possible. In addition, code is added to the schedule to allow loads to migrate past stores. The load is always performed and the value of the bypassed store is saved. The address of the load and store are checked and the proper datum is loaded. This strategy works well for certain codes on systems with long memory latencies. The difficulty is that the amount of code required to correctly implement this grows exponentially with the number of stores bypassed. Also, the conditional code required to select the correct result can be problematic on architectures with a large branch penalty. This can be mitigated by providing additional instructions such as conditional moves.

[118] examines compiler generated prefetching in detail. Loop pipelining with locality analysis is used to generate as few prefetches as possible. This gives very good results, removing 50 to 90% of the cache misses in their benchmarks. In all cases, the selective prefetching algorithm showed improved performance over no prefetching.

Memory latency hiding via prefetch and speculative load instructions has also been done in the context of superscalar architectures. This work raises some interesting questions about DAE architectures. One of the major advantages cited for DAE architectures is that they hide memory latency [157]:

> “Another important characteristic of decoupled architectures is a reduced sensitivity to memory access delays. This results from the ability of the access instructions to run ahead and fetch data in advance of when they are needed.”

It is not always possible to build a DAE schedule which does this, for instance where there are recurrence relations in a loop. In this case the performance of a DAE architecture is greatly reduced [34]:

> “An interesting commentary on the architecture is to note that once one has become used to the decoupling of the APS and the APU, the need to synchronize, as in the examples above, becomes quite disturbing. For example, one can see in the process above how the APS must wait for the APU to catch up; then, after SET(WI), the APU will in most cases be waiting until the APS gets the first address out and the IQ has data. Clearly, both of these waits represent idle hardware--and resulting inefficiency.”

Perhaps a combination of prefetch and speculative load instructions would provide a better means of hiding memory latency than a DAE architecture.

The possibility of adding prefetch instruction to a VLIW architecture is raised by Callahan and Kennedy in [23]. They speculate that a VLIW implementation may reduce the overhead, making prefetch instructions profitable:

> "Software prefetching should be particularly useful on high-performance systems that can issue more than one instruction per cycles -- if the costs of issuing the prefetch instruction and computing the prefetch address can be completely hidden under other instructions, the reduction in execution time can be substantial."

Prefetch and speculative load instructions have been shown to give substantial performance improvements on scalar machines. It is likely that these type of instructions would be even more useful on a VLIW architecture.

3 Similar Studies

One study which is closely related to our work is a comparison by Smith, et al. between a dynamically scheduled superscalar processor and a "static" superscalar [162][164]. In these studies, the dynamic superscalar architecture has a reservation station style execution mechanism. The static superscalar is a VLIW type architecture where instructions execution in-order. Support is included in the static architecture for speculative execution by providing delayed-exception instructions and explicitly referenced shadow registers and buffers. Both architectures have been simulated with instruction widows of size 2 and 4.

With an instruction window of size 4, the static superscalar shows a speedup of 1.6 over scalar as compared to a dynamic speedup of 1.9 over scalar, a difference of only 20% [162]. A 1.2 times speedup was available on the static architecture without speculative execution support. Most of the performance improvement with speculative executions was found with moving instructions across only one branch [164].

A performance improvement midway between that shown by the static architecture with and without speculative execution was found with a system with 64 registers, versus 32 registers and 32 shadow registers. The fact that more non-shadow registers is useful seems to suggest that the shadow register file may not be the correct organization. It is not clear that a full set of shadow registers will be effectively used and a better implementation of speculative results may be the reorder buffer found in the WISQ project [137]. Perhaps a reorder buffer would allow both a large register set and speculative execution.

Static versus dynamic instruction scheduling is studied in a comparison of a VLIW with a DAE architecture by Love [107]. In this study, a set of benchmark programs was hand compiled and hand optimized for both a VLIW and a DAE architecture. Simulations of the architectures showed little performance difference. The programs were equally split as to which architecture had better performance. The performance variation between the two architectures was also similar.

There were some problems with this study. One important area was that the programs were hand compiled and optimized for each of the architectures. How well a compiler can generate code for an architecture is a key part of the performance equation. Compiling the benchmarks by hand fails to answer this important question. Building a compiler to answer this question is time consuming, but it eventually must be done for the results to have validity on a system where most of the executed code is compiled.

CHAPTER III
LOOP OPTIMIZATIONS

When evaluating an architecture, which compiler “optimization” techniques are applied when generating code is critically important. Using no optimization techniques or only machine independent techniques can lead to an over-estimate of the amount of parallelism being exploited by the architecture [80] and an under-estimate of the performance of the architecture [102][167].

The question of compiler capabilities becomes paramount when exploring architectures with varying scheduling policies, because architectures with static scheduling rely heavily on the compiler for performance. Generating code by hand for a study of architecture performance begs the question of compiler behavior because a large part of how well the system performs is embodied in the algorithms in the compiler. This is a flaw in a previous study of static versus dynamic scheduling by Love [107].

Given the importance of compiler optimizations, we still have to decide which compiler techniques should be included in our investigation. All of the standard machine-independent optimizations should be performed, such as those described by Aho, et al. in [4]. Failure to perform these optimizations would skew the results toward showing larger amounts of simple address calculations, as shown in [80]. This is especially true in the benchmarks we will use, the Livermore Loops, which are composed of DO-loops containing array operations. For this type of code, traditional compiler optimizations are very effective. The common and important optimizations for loops containing array references are loop induction variable detection and reduction, forward substitution, code hosting, and dead code elimination.

In addition to generic optimizations, some machine dependent optimization and code generation techniques are available, which have been specifically targeted for statically scheduled architectures. Loop unrolling, trace scheduling, and software pipelining are scheduling techniques which have been used to improve performance on statically scheduled architectures. These techniques could be combined in a single compiler. Loop unrolling is generally used with trace scheduling [46], and loop unrolling has been used with software pipelining [149]. However, trace scheduling and software pipelining have not been combined, probably because of the complexity of these techniques.

1 Loop Unrolling

Loop unrolling works by replicating the body of a loop some (machine and code dependent) number of times and scheduling the resulting code as a single basic block. Replicating the loop body has a couple of performance advantages: Producing a larger loop body provides a larger block of instructions for the scheduler to work with, which gives the scheduler more options when positioning operations; Combining multiple iterations allows induction variable computations to be combined. These performance improvements are traded against the potential penalty caused by increased I-cache misses on the larger loop body.

1.1 An Example of Loop Unrolling

A schedule for a short vector loop provides a good demonstration of loop unrolling. This loop is shown in Figure 6.

```
Do I = 1, N
    X[i] = A * (Y[i] + Z[i])
end
```

FIGURE 6. Source for a vector loop

Assume we are scheduling for a scalar architecture with addition and multiplication function units, each with a 3 cycle latency. The loads, stores and loop control will be ignored and only the addition and multiplication will be scheduled. NOPs are also not shown. A simple schedule for the loop body, without unrolling is shown in Figure 7.

```
1: t1 = Y[i] + Z[i]
2:
3:
4:   X[i] = A * t1
```

FIGURE 7. The loop body without unrolling

A schedule where the loop has been unrolled three times is shown in Figure 8. There are no dependencies between iterations in this case, and unrolling the loop body produces very efficient code.

```
1: t1 = Y[i] + Z[i]
2: t2 = Y[i+1] + Z[i+1]
3: t3 = Y[i+2] + Z[i+2]
4:   X[i] = A * t1
5:   X[i+1] = A * t2
6:   X[i+2] = A * t3
```

FIGURE 8. The loop body with unrolling

1.2 Loop Unrolling Performance Benefits

Loop unrolling can be considered *the* standard optimization technique: It is in use in most commercial compilers, and loop unrolling is pervasive enough that its absence from a compiler’s repertoire is cause for comment [111].

Loop unrolling works by concatenating multiple copies of the original loop body to form a new, larger loop body. The number of copies made of the loop body is the *unroll count*. The loop bounds checking is not included in the copies of the loop body and the bounds checking on the new loop is modified to reflect the behavior of the new loop. For instance, if unrolling the loop in shown in Figure 6 four times would yield the loop shown in Figure 9.

```
Do I = 1, N/4, 4
    X[i] = A * (Y[i] + Z[i])
    X[i+1] = A * (Y[i+1] + Z[i+1])
    X[i+2] = A * (Y[i+2] + Z[i+2])
    X[i+3] = A * (Y[i+3] + Z[i+3])
End
```

FIGURE 9. Unrolled Loop

There is some overhead associated with loop unrolling. The number of loop iterations may not be an integral number of unroll count, so code must be generated to check for this case and execute any remaining iterations which cannot be executed in the unrolled loop body. The cleanup code will generally be less optimal than the unrolled code; if the loop typically executes few iterations, loop unrolling can be detrimental to performance. One way to overcome this problem is to add code specially designed to execute the loop a constant few iterations (Hwu calls this type of structure a *superblock* in [75]). There is also a secondary cost of loop unrolling in some architectures caused by the additional cache misses due to the increased code size [115][116][40][171].

The efficiency of loop unrolling quickly drops in relation to the size of original loop inefficiency and the unroll count. It is easy to see why this is the case. Each additional time the loop is unrolled, the idle portion of one iteration is removed. The idleness reduces at the rate 1-(unroll_count_idle_fraction). For short loops with a small initial efficiency, the loop may have to be unrolled a large number of times to significantly increase the efficiency. This makes loop unrolling not very effective at improving efficiency on a short loop with high initial overhead, e.g. unrolling a loop containing a single high latency operation.

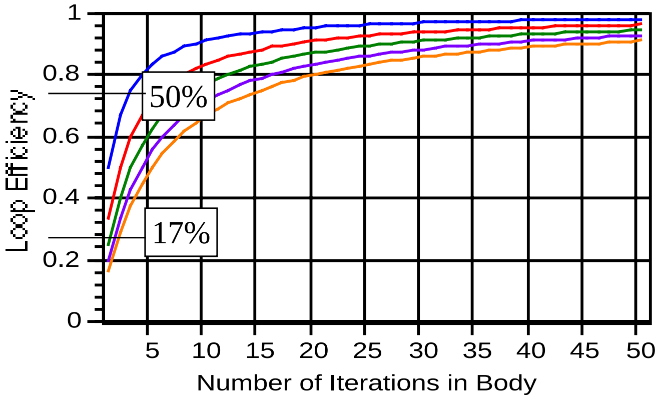


FIGURE 10. Loop Efficiency vs. Number of Iterations Unrolled

The top curve shown in Figure 10, is the efficiency curve for a loop with initially a 50% efficiency, e.g. 1 busy cycle and 1 idle cycle. The bottom curve is an initial 17% efficiency, e.g. 1 busy cycle and 5 idle cycles. As can be seen here a loop starting at 50% efficiency must be unrolled 9 times before the efficiency reaches 90% and a loop starting at 17% efficiency must be unrolled 45 times before reaching 90%.

The disadvantage of having to unroll the loop a large number of times is that more time is likely to be spent in unoptimized code sections. An unrolled loop is usually constructed with an unoptimized version of the loop to execute iterations which cannot be executed in the unrolled version. If the loop executes fewer iterations than the unroll count or the number of iterations is not a multiple of the unroll count, the unoptimized version is executed to handle these iterations. If loops are unrolled many times and executed few iterations, loop unrolling will provide no advantage because most of the time will be spent in the unoptimized version of the loop.

In spite of its drawbacks, loop unrolling is an effective optimization technique. In a study of parallelism on a VLIW architecture [167], Sohi and Vajapeyam find no speedup is provided by the architecture without loop unrolling. They do find a speedup of up to 1.6x scalar on a four operation VLIW architecture with loop unrolling. In [102], Lee, et al. find speedups between 1.5x and 9.0x scalar on a superscalar SPARC with an instruction window of four. This study also includes software pipelining. They find loop unrolling provides better performance than software pipelining, due to its ability to remove branches and index computations. However, they also note that loop unrolling can be combined with software pipelining to get the benefits of both techniques.

2 Trace Scheduling

Trace scheduling attempts to increase the size of the block of code presented to the scheduler by scheduling the blocks from one commonly executed path through the program (a *trace*). This was first proposed by Fisher in [52] as a way to increase the available parallelism at the microcode level. The technique has also been applied to horizontally microcoded architectures [96][104].

In [46], Ellis applies trace scheduling to an “8-cluster ELI” and finds good speedups. He was able to get speedups of up to 7.4 times scalar on some problems. Unfortunately, he does not determine how much of the improvement was due to trace scheduling and how much was due to other optimizations, namely loop unrolling. The trace scheduling compiler and ideas from the ELI project ultimately became the basis for the MultiFlow VLIW architecture of [123].

Trace scheduling works by allowing operations to migrate across conditional operations, which normally block code motion. The entire trace is treated as one basic block for scheduling purposes and operations in the trace can be scheduled in whatever order is most beneficial, limited only by data dependencies.

After the operations in the trace have been scheduled, clean-up code is added to the off-trace branch of every conditional to compensate for any operations which have made a block crossing. This is one of the main ideas behind trace scheduling: That code could be added to undo the effects of speculative execution of an operation when the guarding condition fails. For instance, suppose that a simple increment is to be moved above a conditional. This transformation is illustrated in Figure 11.

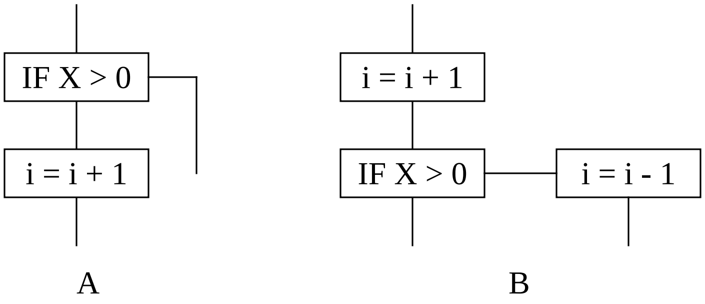


FIGURE 11. Trace Scheduling Example.

In this case, the effect of the increment instruction can be can be undone on the off-trace branch of the conditional by adding a decrement instruction. However, this assumes that moving the increment instruction above the conditional will not cause an extraneous overflow exception. This is one potential problem with trace scheduling: an instruction executed earlier than normal may raise an exception which it otherwise would not. For instance, loads are often advantageous instructions to move up in a schedule and they can normally produce exceptions. MultiFlow allowed loads to be trace scheduled by adding a non-exception raising load instruction [36].

Another problem with trace scheduling was that at first it was not clear whether the trace scheduling algorithm would always terminate. In [127], Nicolau showed that trace scheduling will terminate, but that there could be an exponential number of additional operations produced, which is practically the same as not terminating. Sequences of conditional constructs can cause this phenomenon. This is unfortunate because long sequences of conditional code are exactly where it would seem to be appropriate to apply trace scheduling. It seems this is not the case, as noted by Ellis in [46]:

> “But as discussed in chapter 8, even with the automatic profiler these programs had little available parallelism. This had many branches with probabilities close to half (branches that went each way about the same number of times). Trace scheduling will never do very well on such programs, because the core assumption of trace scheduling is that branches mostly go one way or the other.”

Trace scheduling is also limited in its ability to handle loops. A trace must be a linear sequence of operations with no cycles, so back edges of loops are excluded. Fisher speculates on some possibilities for extending trace scheduling to handle loops, but these were not implemented [52].

It is also not clear how much of a performance benefit is provided by trace scheduling when this technique is used on a VLIW architecture. In the Bulldog compiler, Ellis used trace scheduling with loop unrolling to improve the performance of loops. He reports good results, but leaves open the question of how much performance improvement came from trace scheduling and how much came from loop unrolling. [102] reports speedups of up to 9 times scalar for a superscalar SPARC using loop unrolling alone, so this is a real question.

Colwell reports the performance of the MultiFlow VLIW machine in [36]. Unlike the work in [46], which was a simulated architecture, the MultiFlow contained all the idiosyncracies of a real machine and the results are much more conservative than the results from the earlier ideal machine. The 14 operations wide system only achieves a speedup of 1.5 over the 7 operations wide system. While this is a respectable speedup on a real machine, this type of performance increase has been shown on systems with less resources, specifically less instruction width, e.g. in [162].

Trace scheduling is an interesting idea. Allowing operations to migrate across block boundaries can be a powerful technique for a compiler to have in its repertoire. And Ellis has demonstrated that a trace scheduling compiler can be constructed for a complicated architecture.

Trace scheduling long blocks of operations may be an overuse of a good idea. Allowing operations to migrate across block boundaries during static scheduling can be beneficial for performance. This idea has shown up in other systems which do not purport to be trace scheduling. In [162], Smith, et al. describe the Torch processor, which uses delayed exception instructions and shadow buffers to allow instructions to be scheduled across block boundaries. They report good results, even though their system only allows a single conditional to be crossed. In [27], Chen, et al. add compensating code to allow loads to migrate across stores in order to handle long memory latencies. They show good results for intermediate latency memory (20 cycle latency), even though only one store can be crossed. Systems employing memory prefetch instructions provide a non-blocking, non-exceptional load instruction to allow loads to be migrated to earlier than normal positions [85][23].

One of the ideas central to trace scheduling, allowing code to migrate across block boundaries with support to compensate for the effects of early instruction execution, has been incorporated into at least academic thought. Execution profiling to determine the most likely branch direction is also popular due to the performance improvement available by correctly predicting branches [72][103][112]. Even though trace scheduling compilers are uncommon, the ideas used in trace scheduling are still actively pursued.

3 Software Pipelining

Software pipelining first appeared in microcode [87] and was developed as a compiler scheduling technique by Rau, et al., along with VLIW architectures [142] [143] [144]. Software pipelining developed for the same reasons as trace scheduling and has the same effect, i.e. software pipelining looks for larger amounts of parallelism by scheduling operations across basic blocks. However, trace scheduling selects linear sequences of blocks without back arcs and specifically avoids loops. Software pipelining works specifically on loops. In trace scheduling, operations are allowed to migrate throughout the trace, potentially crossing block boundaries. In software pipelining operations are allowed to migrate between iterations, potentially crossing the block boundary at the end of the loop.

In sequential loop execution, each iteration begins execution after the completion of the previous iteration. The sequential execution of three iterations of a loop is illustrated in Figure 12.

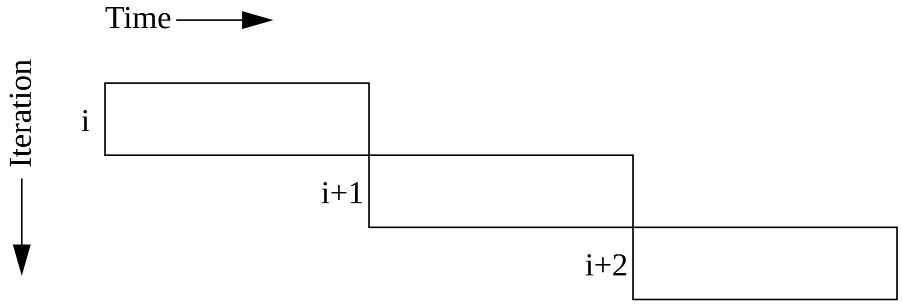


FIGURE 12. Sequential loop execution

In a software pipeline, successive iterations are allowed to begin execution before all the preceding iterations have completed execution. The pipelined execution of three iterations of a software pipeline is illustrated in Figure 13. Software pipelining provides a form of execution for the iterations which behaves like a standard hardware pipeline.

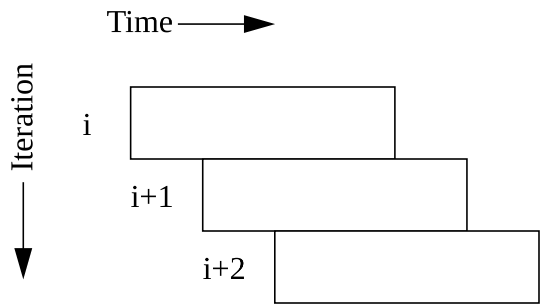


FIGURE 13. Pipelined Loop Execution

In software pipelining, a loop is treated as the basic unit of scheduling. Operations are allowed to migrate across the block boundary at the beginning and end of the loop, into previous iterations or out of the loop into prolog or epilog code. Iterations of the loop migrate into each other with subsequent iterations beginning execution before previous iterations have completed. This compresses the schedule, allowing higher performance than can be achieved by scheduling only within the body of the loop.

At any given time a software pipeline can be executing instructions from several iterations. This potentially provides parallelism not otherwise available. The amount of parallelism available is still dependent on the particular program being compiled. Control and data dependencies must still be honored. However, the artificial constraint imposed by block scheduling that each block/iteration must complete before the next block/iteration is entered, has been relaxed.

A software pipeline is constructed by dividing each iteration into a series of equal size blocks of instructions. These blocks of instructions are the schedule stages.

Instructions are scheduled within each stage such that stages from different iterations can be executed simultaneously. New iterations are initiated as each stage completes.

There are three distinct phases to pipelined loop execution: *Prolog, Kernel, and Epilog*. On the first few iterations of the loop, the pipeline is not full and not all the pipeline stages are executing. This is the prolog phase. Once enough iterations have been initiated, the pipeline will be full and all stages will be executing. This is the kernel phase. Once the final iteration has begun execution the pipeline will begin to empty and again not all stages will be executing. This phase is the epilog. The execution phases of a three stage loop is shown in Figure 14.

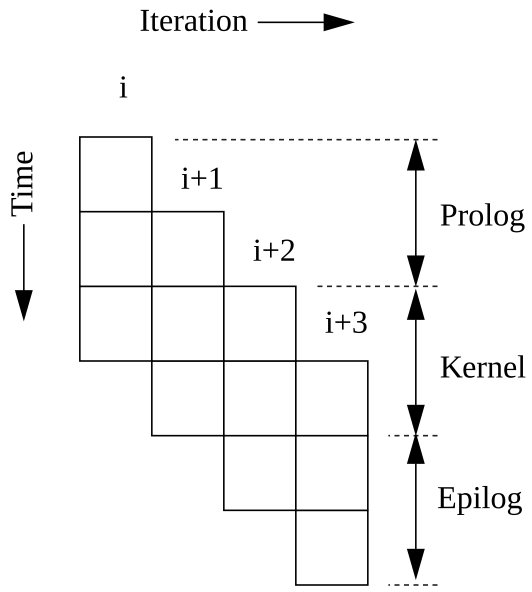


FIGURE 14. Phases of pipelined loop execution

Some mechanism is necessary to insure correct behavior in these different phases of software pipeline execution. Software pipelining can be implemented via either: 1) code segments constructed to execute each of the stages, or 2) conditional instructions with hardware support to execute only instructions applicable to the current execution phase, as in the Cydra 5 [144].

The key part of constructing a software pipeline is finding a steady state kernel, i.e. a schedule for the kernel must be found which can execute the stages from different iterations of the pipeline simultaneously. The same code must be used to execute successive iterations of the loop, thus the steady state requirement. The kernel should be as short as possible as this will generally reduce the time to execute the loop, even though the number of stages will tend to increase. This is analogous to constructing deeper hardware pipelines.

3.1 An Example of Software Pipelining

A schedule for a short loop on a VLIW architecture will be used to demonstrate the construction of a software pipeline. We will reuse the vector loop from the loop unrolling discussion for our example (Section 1.1 on page 44). The source code for this loop is shown again in Figure 15.

```
Do I = 1, N
    X[i] = A * (Y[i] + Z[i])
End
```

FIGURE 15. Source for a vector loop.

We will develop a schedule for a VLIW architecture with pipelined addition and multiplication function units, each with a 3 cycle latency. The loads, stores and loop control will be ignored and only the addition and multiplication will be scheduled. NOPs are not shown. A simple schedule for the loop body is shown in Figure 16.

```
1: t1 = Y[i] + Z[i]
2:
3:
4: X[i] = A * t1
```

FIGURE 16. The loop body without unrolling.

Execution of this schedule, starting with i = 1 and executing for two iterations would execute the stream of instructions shown in Figure 17.

```
1: t1 = Y[1] + Z[1]
2:
3:
4: X[1] = A * t1
5: t1 = Y[2] + Z[2]
6:
7:
8: X[2] = A * t1
9:  ...
```

FIGURE 17. Execution of a few iterations of a loop without unrolling.

This loop is a DOALL type loop - there are no dependencies between loop iterations. Because there are no inter-loop dependencies, the execution of this stream of instructions can be compressed. If we first look at the execution stream, compressed as much as possible and executed for six iterations, the execution will be much more efficient, as is shown in Figure 18.

```
1: t1 = Y[1] + Z[1]
2: t2 = Y[2] + Z[2]
3: t3 = Y[3] + Z[3]
4:   t4 = Y[4] + Z[4]; X[1] = A * t1
5:   t5 = Y[5] + Z[5]; X[2] = A * t2
6: t6 = Y[6] + Z[6]; X[3] = A * t3
7: X[4] = A * t4
8: X[5] = A * t5
9: X[6] = A * t6
```

FIGURE 18. Compressed execution of a few iterations of the loop.

There are a number of interesting features in the execution of this loop. It is executing in a pipelined fashion. There is a period of time, cycles 1 to 3 is the prolog phase, where the pipeline is filling up. The kernel phase is cycles 4 to 6, where the pipeline is full and running at maximum efficiency. Cycles 7 to 9 are the epilog. There are no more additions to do and the pipeline is draining.

With one small caveat, the instruction shown in cycle 4 of Figure 18 forms the kernel of this loop, which can be coded in one instruction in this case. A software pipeline schedule for this loop is shown in Figure 19.

Prolog 1: $t_1 = Y[1] + Z[1]$
2: $t_2 = Y[2] + Z[2]$
3: $t_3 = Y[3] + Z[3]$

Kernel 4: $t_{i\%3} = Y[i] + Z[i]; X[i-3] = A * t_{(i-3)\%3}$

Epilog 5: $X[N-2] = A * t_1$
6: $X[N-1] = A * t_2$
7: $X[N] = A * t_3$

FIGURE 19. A Software Pipeline version of the loop body.

The caveat to this schedule is that the temporary values produced by the addition must all have separate locations which are accessible by the multiplication, 3 cycles in the future. In other words, there are 3 simultaneously live values of t and this must be accounted for in the schedule. As is discussed in more detail later, there are several ways to handle this, by either providing hardware support, or unrolling the kernel and renaming each of the instances.

The details of how this schedule is coded also depend heavily on the hardware support available in the machine. If the machine has explicit hardware support for software pipeline execution, the schedule would be coded by just giving the kernel. The kernel is shown in Figure 20.

$$t_{i\%3} = Y[i] + Z[i]; X[i-3] = A * t_{(i-3)\%3}$$

FIGURE 20. The Kernel of the loop body.

The hardware would execute the prolog and epilog by executing the appropriate operations and squashing the remaining operations.

If pipelined loop control is not available in hardware, the prolog, epilog, and kernel must be expanded into separate sections of code and the control code must be added which determines when to enter and exit these sections. In addition, some compensation code may be necessary on exits to put the program in a known state. This is true, for instance, where the kernel has been unrolled to map temporary values to different registers. Depending on when the loop exits, the register mapping may not match the code following the loop, or the epilog. Compensation code must be added at these exits to align the registers.

Because of the unrolling, a pipelined loop may also require that some iterations be executed outside of the pipelined code. This is the same as in standard loop unrolling, when the iteration count is not a multiple of the unroll count. A pipelined loop, implemented without hardware support, might have the structure shown in Figure 21.

```
Prolog               1: t1 = Y[1] + Z[1]
                     2: t2 = Y[2] + Z[2]
                     3: t3 = Y[3] + Z[3]
                     ---------------------
Kernel               4: t1 = Y[i] + Z[i-3]; X[i] = A * t1
                     5: t2 = Y[i+1] + Z[i-2]; X[i+1] = A * t2
                     6: t3 = Y[i+2] + Z[i-1]; X[i+2] = A * t3
                     ---------------------
Epilog               7: X[i] = A * t1
                     8: X[i+1] = A * t2
                     9: X[i+2] = A * t3
                     ---------------------
Compensation Code
                     ---------------------
Exit
```

FIGURE 21. A Software Pipelined loop body with register expansion.

In this example, the kernel has been unrolled to map the 3 live temporary values into separate registers. The control instructions and compensation code have not been shown.

In contrast to a pipeline schedule, a schedule produced by standard loop unrolling looks much the same except that the code is packed together into a single block. In standard loop unrolling, the pipeline would have to be filled and drained each iteration of the unrolled loop, so the efficiency is not as high as with the pipelined loop, where the kernel executes at maximum efficiency for most of the iterations.

3.2 Software Pipelining Scheduling Methods

There are several methods for constructing a software pipeline. One method is to construct the software pipeline directly in the scheduler. This is the method used by Lam and Rau [99][144] and is also used by Tortoise (for more detail see Chapter IV, Section 4 on page 94). As each operation is scheduled, it is subject to constraints that the operation complete before its result is required by subsequent iterations and that resources are available to execute the operation at its relative position in all stages. These constraints are in addition to the normal constraints that an operation can only execute once it operands have been computed and that there are enough resources to execute the operation with respect to one iteration.

A problem with this type of scheduling is that the scheduling algorithm may fail to find a schedule. On attempting to schedule an operation, it may not be possible to have the operation complete before its result is required in subsequent iterations. This is because scheduling each operation in the kernel implies that the operation will execute at the same relative time in all iterations, including future iterations. Another previously scheduled operation may have been scheduled too early to allow the current operation to complete. If there were only resource constraints or only dependency constraints this could not happen, but trying to satisfy both types of constraints at once causes this problem.

It is possible that the schedule could be repaired by increasing the size or number of the stages, thus moving future operations later in time, and repairing the data structures to reflect the new operation times. However, this may require as much work as rebuilding the schedule. This method also implies that dependencies can be arbitrarily delayed. This is generally true only if each dependency is associated with a register as in most GPs. In microcode, where a data path must be an exact length, this reordering may not be possible.

Some of the more difficult scheduling issues were avoided in the early systems by only using software pipelining in restricted situations. In the Floating Point Systems compilers [181][26], software pipelining was restricted to a single fortran statement which contained no recurrences. Later work by Rau et al., on what became the Cydra 5, tried to minimize the problem of scheduling by removing as many resource constraints as possible [142][144]. This was provided in the form a large crossbar register file. This provided a register file with a large number of registers and a large number of ports, thus eliminating contention for registers and access.

While sufficiently complex loops could still require rescheduling, rescheduling is generally not a major problem. The stage size necessary to find a valid schedule can usually be estimated fairly accurately. Lam, when working on the compiler for the Warp project, reported that a schedule was usually produced after only one or two tries [99]. Lam also found a method for dealing with conditional code within a software pipeline. This was not possible with earlier software pipeline schedulers, which restricted the body of the loop to be a basic block [26][142][181].

Some scheduling algorithms which attempt to reduce the scheduling failure rate have been developed for microcode schedulers. Microcode tends to have more constraints than higher level instructions, so reducing the failure rate becomes more important. One method, discussed by Allen et al. in [7], is to use a two step scheduling process. The operations are first scheduled using forward (inter-iteration) dependencies. The sorted operations are then scheduled for software pipelining. This tends to reduce the failure rate because critical operations can be found, i.e. operations are already close to their final order and critical operations will have a higher probability of being frontier nodes. Also, operations later in the schedule can be moved because they are only tentatively scheduled.

Another method for constructing a software pipeline is to compress an already complete schedule. In this method, scheduling starts with a standard basic block schedule for the loop. The loop is unrolled and then examined to find a steady-state segment which becomes the kernel. The remaining portion of the unrolled schedule becomes the prolog and epilog. This process is repeated until the kernel can no longer be compressed.

This iterative method has the advantage that a schedule will be found in a well bounded amount of time [6]. Unfortunately, the time actually required to produce a schedule may be larger than that required by the construction method. This method also tends to produce less compact schedules than the construction method [7].

Another advantage of the compaction method is that, since the method starts with a complete schedule, resource constraints are more readily incorporated into the algorithm, and the algorithm can be more easily applied as a post-pass assembler optimization [77].

An interesting aside is the work on digital signal processors by Schwartz in [152]. In this work the goal was to find an optimal schedule and then construct the hardware to execute the schedule. Optimal schedules could be found because there were effectively no hardware constraints. This work also used a slightly more general type of execution than is usually in most systems. Generally, in a VLIW or horizontal architecture, the function unit executing a particular operation at a given point in the schedule is fixed, because the schedule is static. In the processor used by Schwartz, the function unit executing the operation could be shifted each iteration. This allows certain types of communication patterns to be optimized, producing more compact schedules than are otherwise possible.

3.3 The Performance of Software Pipelining

Software pipelining has been shown to work quite well in on a number of diverse architectures. On the Warp systolic array, Lam found loops scheduled using software pipelining to have an average 3 times performance increase over block scheduling [98]. Optimal performance was achieved for the majority of the loops in these benchmarks. Software pipelining has demonstrated to provide significant performance improvement on the Cray, over the Cray Fortran compiler [173][45]. These studies showed the performance of software pipelining on the Cray architecture was sometimes limited by the small number of registers available on the Cray. In a later study, Mangione-Smith et al. found that performance could be improved on the Cray architecture if the vector registers were reformatted to complement software pipelining by providing more and shorter vector registers [110].

A discouraging note on performance of software pipelining is found in [102]. Lee et al. study scheduling on a superscalar SPARC and find loop unrolling superior to loop pipelining:

> “This result is from the advantage loop unrolling has in reducing
> the loop control overhead and indices increment operations.”

However, the authors do note that loop unrolling and software pipelining are complementary techniques and their best results are from a combined use of unrolling and pipelining. A similar result is found by Weiss and Smith [191]. In this study, loop unrolling and a simple software pipelining algorithm is used to schedule code for a Cray-1S. Loop unrolling provides a speedup of 1.8, while software pipelining provides only a 1.3 times speedup. However, a very simple software pipelining algorithm is used in the study and, as noted in other studies [173][45][110] the Cray has too few vector registers to allow the most effective use of software pipelining.

Since some studies have reported good results with software pipelining and others have reported better results with loop unrolling, it is worth wondering if one of these techniques is better than the other and if so, which technique. The performance of software pipelining is very dependent on the target architecture and the structure of the particular loop being scheduled. In the limit, a scalar architecture with one cycle instructions will not benefit from software pipelining. An architecture with intermediate latencies and parallelism may find loop unrolling to be more beneficial because of its ability to remove loop dependent computations. On the other hand, a loop with a dependency structure which does not allow instructions to be moved or removed will not benefit from loop unrolling either.

Software pipelining will have maximum benefit on a architecture with long latencies and a large amount of parallelism. As discussed in Section 1.2 on page 45, it is more difficult to increase the efficiency on such an architecture with loop unrolling alone. Another factor which can favor software pipelining is difficult to schedule resource constraints, which makes it difficult to compact unrolled loops. For instance, suppose that our example loop in Figure 6 is to be scheduled for a target architecture which allows an addition and multiplication to be started each cycle, but not two additions or two multiplications. A schedule for this loop created using loop unrolling cannot be made 100% efficient because multiple additions cannot be issued together at the beginning of the loop body and multiple multiplications cannot be issued together and the end of the loop body (see Figure 8). Software pipelining allows these parts of the schedule to be migrated out of the loop body, providing better efficiency.

Perhaps the best characterization of a good architecture for software pipelining is provided by the microarchitectures on which software pipelining was developed. These architectures tend to have wide instructions with constrained resources and operations with long latencies. Software pipelining fits these architectures quite well, although hardware support for software pipelining does help [87]. This idea is expanded and generalized by Rau et al. in [141][142][144][149].

Software pipelining appears to be a successful scheduling technique, one which needs to be at least be considered in an optimizing compiler. However, its promise is tempered by that fact that it places heavy demands on machine resources, particularly instruction cache and registers.

CHAPTER IV
THE STRUCTURE OF THE OPTIMIZING COMPILER TORTOISE

There are two important reasons for using a tailored optimizing compiler when studying scheduling techniques. First, aggressive scheduling techniques require good data flow information, which is normally only found in optimizing compilers. In particular, the performance of aggressive scheduling techniques is highly dependent on flow analysis to drive program transformations such as induction variable strength reduction and promoting inter-loop operands into registers. In addition, the instruction mix produced by an optimizing compiler can be different from the mix produced by a non-optimizing compiler. For instance, in [80], Jouppi and Wall find that various types of optimizations and register allocation strategies can vary the measured amount of parallelism by a factor of almost 2. The same study found a significant difference in parallelism in the Livermore Loops when CSE detection was added for array reference computations. The difference instruction mix seen by the architecture can bias the results of performance related experiments. As we shall see in Chapter V, in addition to just being an optimizing compiler, the compiler must also be tailored to the architecture, to avoid bias from the scheduling techniques employed within the compiler.

An alternate approach to using a tailored optimizing compiler would be to test the performance characteristics of various architectures using a standard, widely available compiler such as the Gnu C compiler. This approach has the appeal that a compiler is more easily obtained and much less compiler work is necessary. Many architectural studies take just this approach (see Chapter II). This approach does have some limitations, however. In particular, it ignores the issue of how different compilation techniques interact with the architecture being studied. The assumptions inherent in the particular compilation algorithms and techniques used in the compiler will benefit computer architectures which match those assumptions and will be detrimental to performance on architectures where those assumptions do not hold. In particular, keeping the compiler invariant in our study would not allow us to explore trade-offs at the compiler/hardware boundary, e.g. “Is it better to have good register allocation algorithms in the compiler or register renaming in the architecture?”

Another widely used technique is to use a standard compiler, but apply a post pass after code generation to adapt the code produced to a particular architecture, or to apply optimization techniques. While more flexible than just changing the code generator, this limits the types of compilation techniques which can be studied because of the limited information available after code generation.

We would like to have the option of using aggressive scheduling techniques for a range of architectures. This requires that we have available an optimizing compiler which we can modify as necessary to implement our algorithms. Having decided that it is important to have such a compiler, the questions remain of where to obtain a compiler and what techniques to employ within the compiler.

Compilers are valuable commodities and are not readily available in source form, especially high quality optimizing compilers. The one exception is the Gnu C compiler, which is widely available. The problem with using Gnu C, at least in the version available at the time (1.37), was that it did not do much in the way of collecting flow analytic information and thus was weak in the area of loop optimizations. Since this was exactly the area which we wanted to study, this deficiency needed to be corrected if we were going go use the Gnu compiler. This left us with the options of extensively modifying the Gnu compiler or writing our own compiler.

We decided to write our own compiler for a number of reasons. The internals of the Gnu compiler seemed arcane and we did not want to spend our time becoming fluent in them. In addition, writing our own compiler would give us full control over the structure of the compiler and allow us to build a flexible basis for compiler and computer architecture studies. However, we are not particularly interested in the front end of the compiler, so we use the Gnu C front end to parse the source and build an intermediate representation (IR). Our compiler starts from this IR, runs flow analytic routines and produces assembler for the target architecture.

We are still left with some major issues to resolve: What should be the basic structure of the compiler and what language should we write it in? The basic structure of Tortoise derives from the desire to drive much of our transformations and scheduling from flow analytic information. The compilation process was developed on a theme of repeating passes of: 1) produce and gather some flow information followed by; 2) transform the IR based on the flow information. To support this process, the program being compiled is represented as directed graph where the nodes represent operations decorated with flow information, and the edges represent dependencies between the nodes.

For the implementation language we wanted to use something more powerful than C, which would aid us in exploring algorithms, perhaps at the cost of some runtime efficiency. Lisp has been used in research projects for this reason and was considered for this project. However, at the time this project began, Mathematica had just appeared and seemed to provide some interesting capabilities, so we decided to use it to implement our compiler. In particular, Mathematica provides extensive pattern matching on expressions which allowed us to quickly implement and test some of our algorithms.

We did run into two problems with using Mathematica. The first was that our data structures quickly deviated from the domain on which the Mathematica pattern matching worked. The Mathematica data structure is an expression tree. Our data structures quickly became general directed graphs with cycles, which cannot be directly represented as Mathematica expressions, and thus the pattern matching was not available to us. We could still use pattern matching where we either maintained or reconstructed a Mathematica expression, but we could not use it to match portions of our graph, which would have been a good way to drive code generation. We had to implement the pattern matching on the graph by hand, just as we would have to have to do if we had written in C or Pascal.

The other problem we encountered using Mathematica will be familiar to users of Lisp and other interpreted systems. The execution speed of an interpreted system is acceptable when implementing small programs and test cases. However, once we began to run our compiler with full data flow analysis and code generation on complete programs, the execution speed became unbearably slow. By the time it became obvious how slow the compilation speed was going to be, we were committed. We derived the name for our compiler, Tortoise, from it execution speed.

In Chapter III, we examined a number of aggressive scheduling techniques. We choose to implement loop unrolling and software pipelining. We are particularly interested in: 1) “scientific code”, which means loop optimizations are important and, 2) static architectures. The literature indicated that both loop unrolling and software pipelining are effective techniques for scheduling code for static architectures, and so it seemed that Tortoise should employ loop unrolling and software pipelining.

We choose not to implement trace scheduling. First of all, it seemed that implementing trace scheduling would require an additional large amount of effort, and from the literature, trace scheduling seems to require some types of speculative execution capabilities to be effective. This was outside the realm we wished to explore in this study, so we choose not to implement this technique.

In addition to the scheduling techniques, we needed to implement the data flow analysis necessary to allow us to support the transforms required by the techniques. And, to fulfill the intent to be a highly optimizing compiler, we needed to implement the transformations which would normally be employed in such an optimizing compiler. This includes induction variable detection and strength reduction, common sub-expression detection and reuse, load/store and extraneous assignment removal, etc. The remainder of this chapter will examine the particular techniques employed in Tortoise and its organization.

1 The Organization of Tortoise

The overall organization of Tortoise will be familiar to any student of compilation. There are three major sections: 1) the front end; 2) the data flow analysis and optimization section; and 3) the code generator and scheduling section. The Gnu C compiler (gcc) is used as the front end. It produces an intermediate representation (IR) consisting of lists of *RTL expressions* [170]. The RTL expressions are composed of simple unary or binary operations which are close to machine level, i.e. they generally have a simple translation into machine instructions. The IR is dumped from gcc as early as possible, before register allocation or optimizations such as loop unrolling, which tend obscure the structure of the program. Symbol table information and the initial block structure graph are also saved from gcc. The IR from gcc is parsed and used to form a Program Dependence Graph (PDG), which is analyzed and optimized in the data flow analysis and optimizations section of Tortoise. The PDG is used by the code generation and scheduling section to produce assembly code. Assembly source code is produced by Tortoise to avoid having to produce machine code directly. The overall structure of Tortoise is shown in Figure 22. More detailed diagrams of the analysis and code generation sections are shown in Figure 23, on page 73 and Figure 35, on page 94.

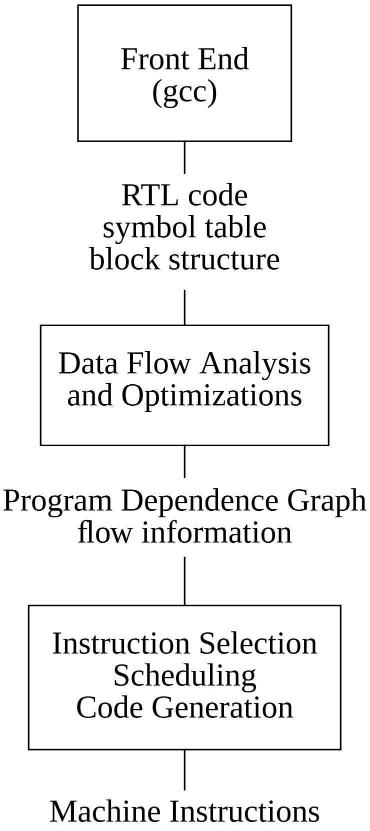


FIGURE 22. Organization of Tortoise

The front end of gcc performs some transformations normally associated with loop optimizations. Gcc rewrites array reference expressions as address expressions and also performs some other optimizations, such as strength reducing integer multiplications to shifts. These optimizations tend to be detrimental to the operation of Tortoise: The transformed expressions are often removed by later transformations and the extra operations add cases to routines which search for patterns in induction expressions. Having to deal with these types of optimizations is an unfortunate consequence of using a preexisting front-end.

2 Data Flow Analysis and Transformations

As stated previously, data flow analysis is necessary to allow us to support the program transformations we wish to make with Tortoise. The goal of data flow analysis is to discover the data flow dependencies between the memory locations referenced in the source program. We need to know which operations will share data through these locations so that we can reorder the operations properly. For instance, if two operations are linked via flow dependency, the dependent operation must execute second because it needs the data produced by the independent operation.

An anti-flow dependency, also called a write-after-read dependency, is where the second operation writes to a location following a previous reference. An output dependency is where two operations write to the same location. Anti-flow and output dependencies also impose an ordering on operators, because they indicate the reuse of a location and thus destruction of data, rather than sharing of data. An input dependency indicates multiple references to a location and does not impose an ordering. It does however, indicate sharing of data, and this knowledge can be used for some optimizations.

Traditional data flow only collects this much information, i.e. for every pair of dependent locations in a program, the dependency is categorized as a flow, an anti-flow, or an output dependency. However, in both software pipelining and loop unrolling, we can and do make use of some additional information: the *iteration distance*. Since software pipelining, and to some extent loop unrolling, schedules multiple iterations of a loop to execute simultaneously, we can produce more compact schedules if we know exactly how far apart, in terms of loop iterations, the operations in the dependency are. This information is not traditionally useful because traditional schedulers would never simultaneously consider the operations from multiple iterations. In our data flow analysis routines, we will collect the dependency type (flow, anti-flow, output, and input), and the iteration distance, where it can be determined to be a constant, integral number.

As the dependency information is gathered, it must be recorded and maintained in a form which will be usable by the compiler. There are two commons forms for representing dependency information. Dependency information can be maintained as tables encoding dependency relations between lists of pseudo instructions (*quads*). In this format the lists of quads are the primary data structure and the dependency tables are decoration. Or, the program operations can be linked together via the graph formed by dependency relations between the operations. The later structure is call a *program dependence graph* (PDG) and is the representation used throughout Tortoise [51].

While straightforward to describe, the dependency information for a program is not simple to discover and collect. Tortoise goes through a number of intermediate steps to discover the flow information in the source program. To describe this process at a gross level, there are three phases in collecting the data flow information: 1) The related definitions and references in each block are linked together; 2) A set of equations on the linked references are solved iteratively to find the dependencies; 3) The dependency information is used to construct the PDG; 4) The PDG is refined using knowledge of loop variables to find the full data flow graph.

The general approach taken in the data flow analysis and loop optimization section of Tortoise is to proceed in cycles where some property of the program graph is discovered and recorded, and then a transformation is made based on the property just recorded. A number of transformations on the program are intermixed with the data flow analysis. The transformations tend to simplify the graph, which provides more information for data flow analysis. Intermixing the data flow analysis and transformation yields a better result than if these passes were run sequentially. The analysis and transformation phases used in Tortoise are shown in Figure 23.

The speed of Tortoise was not considered an important issue at the beginning of this project, so no attempt is made to optimize this section by either combining phases or making incremental changes. If, for instance, a graph transformation invalidates some flow information which is needed later, the flow information is reconstructed by rerunning the flow analysis routines. This slows compiling speed but greatly simplifies the organization and function of Tortoise.

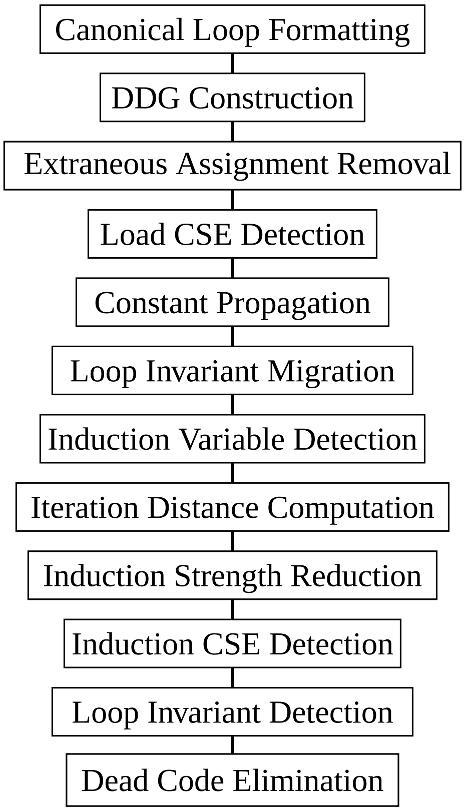


FIGURE 23. Tortoise Analysis and Transformation Phases

2.1 Canonical Loop Formatting

A few transformations are made directly on gcc’s IR, which is a list structure. The structure of loops is more easily recognized and modified in gcc’s RTL list structure form, so the detection of loop nesting and transformation to a canonical loop form is done with gcc’s IR. The canonical loop format used has a tail comparison and branch and is shown in Figure 24. This eliminates an unconditional branch from the bottom to the top of the loop, but requires an additional check before entering the loop to insure that the loop body will be executed at least once.

```
if (index < 1) goto exit
loop:
    loop body
    if (index++ <= N) goto loop
exit:
```

FIGURE 24. Canonical Loop Format

A loop header and tail block is also added to each loop to allow for later code migration out of the loop. All paths into the loops pass through the loop header block and all exits from the loop pass through the loop tail block.[1]

2.2 Block Flow Graph Reconstruction

After the loop header and tail blocks have been added, the block flow graph is modified to include the new blocks (an early version was constructed by gcc). The *block dominators* are then found. The definition of block dominators from [65] is:

> “If x and y are two (not necessarily distinct) nodes in a flow graph G, then x *dominates* y iff every path in G from its initial node to y contains x.”

The dominators are used when moving code out of loops. Certain code motions, e.g. loop invariant code motion out of a loop, requires a dominating block to move the code into. The addition of loop header and tail blocks assure that dominating blocks will be available when hoisting code.

2.3 Initial Program Dependence Graph Construction

The program dependence graph is constructed from the gcc RTL list structure and the block flow graph. The PDG encodes the known data and control dependencies between operations in the program. At this point, the operations in the graph are the same operations as those defined in gcc’s IR, e.g. add, multiply, load, store, etc. The data

[1] These are well formed loops, without branches into the middle of the loop, or exits from inside the loop to arbitrary locations, so this transformation is always possible.

dependencies encoded in the PDG consist of the dependencies found within expressions in gcc’s IR. Data dependencies between expressions and control dependencies will be discovered and added to the graph in following analysis phases. The data structures at this point is still a graph of basic blocks, where each block contains an ordered list of trees. The analysis and transformations which follow will gradually transform this structure into a PDG, which is the structure passed to the code generator and scheduler.

2.4 Initial Data Flow Analysis

The first step in the analysis phase is to perform standard data flow analysis, such as that described by Aho and Ullman in [2]. This determines the dependency relationship between every pair of memory locations in the program. The standard four types of dependencies are discovered at this point: flow, anti-flow, input and output. These are first recorded in def-use, use-def, use-use and def-def chains. An additional structure -- “forward assignment use-use chains” is also constructed for use in removing extraneous assignments (see “Data Dependency Graph Optimization” on page 76)

During the first data flow analysis there is not enough information to reliably disambiguate individual array element references. This is because of a circularity in the analysis algorithms. Distinguishing array elements requires induction variable analysis. Induction variable analysis requires complete flow information on the induction variables, which has not been discovered yet. For these reasons, arrays are treated as composite entities and individual array elements are not identified. If an array element is modified, the entire array is considered to be modified. Another way to view this is that the first data flow analysis is a scalar analysis and all variables are treated as scalar variables. Another data flow analysis will be performed to discover the dependency relationship between array elements.

One of the major functions of the first data flow analysis is to discover and remove temporary “registers” created by gcc. The temporaries are generally not useful to Tortoise as the form of the graph will change substantially before we perform register allocation. Gcc’s operations do not exactly match the operations on the target architectures and the graph transformations performed will remove the need for some registers and create others. The registers from gcc generally indicate flow dependencies. These will be encoded as a dependency link between two (other) nodes. In this case, the register node is discarded. If the register node is a merging point for two or more flow paths, the register node cannot be discarded. The node is retained in this case and may become an executable register copy operation. This is discovered later. In either case, the register assigned by gcc is used only as a label, not as an actual register.

2.5 Data Dependency Graph Optimization

The initial data flow analysis just performed allows a second data dependency graph to be constructed, which is more accurate than the first. A first set of optimizations is performed during the reconstruction: redundant nodes from flow and input dependencies are removed. Although flow and input dependencies are slightly different, both transformations remove extraneous nodes and produce new flow dependency links. For instance, in Figure 25, there is a dependency between statements 3 and 2, and between 2 and 1, caused by the assignment, d = a. If neither a or d is used elsewhere, this program fragment is equivalent to the single statement e = (b+c) * g.

```
1) a = b + c
2) d = a
3) e = d * g
```

FIGURE 25. An extraneous flow dependency

We effect this transformation in Tortoise by replacing the dependency between statements 3 and 2 with a dependency between statements 3 and 1. This transformation is illustrated in Figure 26. Later, statement 2 will be found to be unused and will be discarded during dead code elimination.

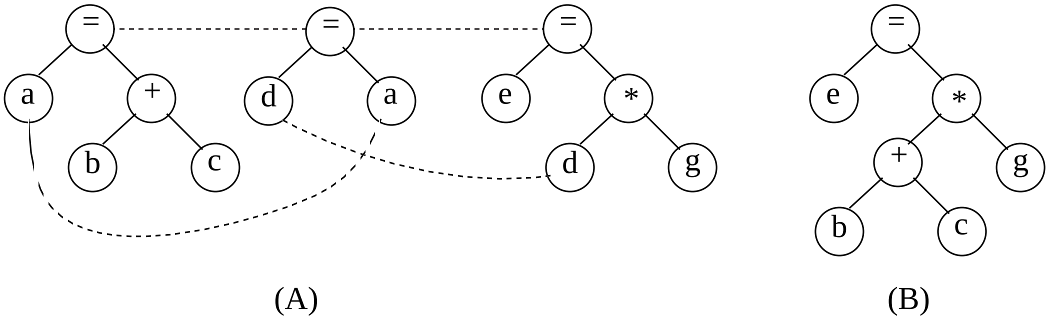


FIGURE 26. Dependency Graph Reconstruction - Flow Dependency

This transformation simplifies the graph and removes nodes which might otherwise have to be computed at run time. There are restrictions on when this transformation can be applied. Flow dependencies though either a register or scalar memory node can be removed and the nodes directly linked whenever there is a single *reaching definition*, i.e. whenever there is a single definition which will arrive at the reference during execution of the program. However, there are some further restrictions on when assignment nodes can be removed.

The transformations resulting from flow dependencies collapse multiple flow dependency links into a single link, removing intervening nodes in the process. The node removed can be register nodes, memory nodes and assignment nodes. In effect this transformation can promote memory nodes into registers (since a flow dependency may become a register during code generation) and remove redundant assignments. For instance, the graph encoding a = b+c; d = a; e = a*g will be transformed into the graph for e = (b+c)*g. This transformation is shown in Figure 26.

2.5.1 Extraneous Assignment Removal

Before an assignment can be removed the graph structure must be checked to insure that removal of the assignment does not change the semantics. In the simple flow dependency case shown in Figure 26, the assignment can be removed without disturbing the program semantics. However, because of the way the graph is organized, there is a potential problem when the assignment has an anti-flow dependency on a previous definition. The links made in the graph to the earlier definition will effectively merge the two definitions, which is not correct in all cases. The typical case where this occurs is an assignment at the top of a loop. The definitions and dependencies formed are shown in Figure 27.

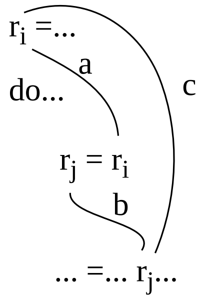


FIGURE 27. Dependencies Involved in Removing Assignment

A check must be made that the use on the right hand side of the assignment reaches the node where the definition created by the assignment is used. It is possible that there is an intervening write to one of the locations used in the expression which changes the value of the expression. In this case the expression would not read the use and the dependency cannot be rewritten. In Figure 27, links (a) and (b) will be replaced by a link (c) and a check must be made to insure that the definition r_i is valid to replace the use of r_j in the last statement. This is a “forward reaching assignment use,”[1] which is an additional flow analytic function which must be computed during data flow analysis.[2]

[1] The standard “reaching use” definition flows backward in execution time, i.e. from the use to the definition.

2.5.2 Load CSEs

The other optimization made during the data dependency graph construction is the sharing of load common subexpressions (CSEs). The input dependencies found during data flow analysis indicate multiple uses of a single variable. In the case where this variable is loaded from memory, all but one of the loads is replaced by a single load. All uses of the variable are replaced by flow links to the single load. This transformation is shown in Figure 28. In this example, the disjoint graphs for expressions (1) and (2) are replaced by a graph in which the result of loading the variable b is shared.

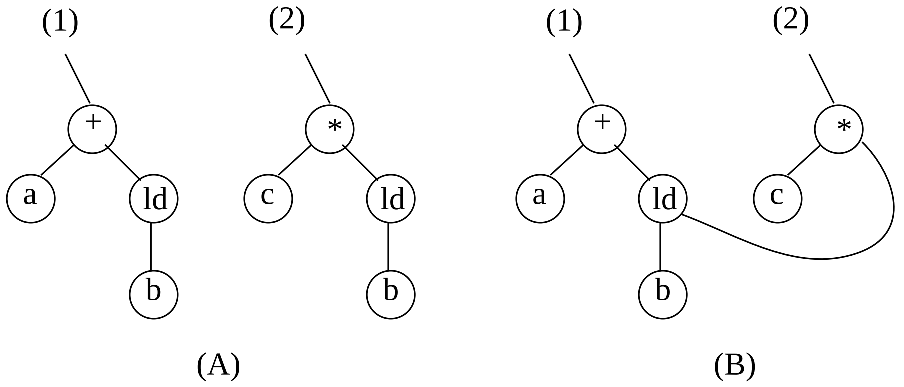


FIGURE 28. Input CSE Dependency Transformation

This “load CSE” detection and replacement also works for other scalar CSEs, i.e. gcc register nodes. These other CSEs are found and linked at this point also.

2.6 Constant Propagation

Once direct data dependency links have been established, constant detection, propagation, and folding are straightforward. *Constant detection* is the process of dis-

[2] This information is actually present in the other data flow information but not in a convenient form. The reaching definitions (coming into the assignments) are available at the block level but this information must be gathered together at the use of the assignment definition. The easiest way to do this is to add the extra flow analytic function “reaching use” and compute this during flow analysis.

covering which operations produce invariant results over the entire execution. This is trivial to determine for leaf nodes. Constant literals and variable addresses are constants. For interior nodes in the graph, the determination of whether an operation is constant must be derived from the structure of the graph. Starting at leaves containing constant values, constant information is *synthesized*, i.e. propagated “up” the graph, until a node is encountered which is not computable at compile time. This is *constant propagation*. Once an interior node has been determined to be a constant, its value can be found by applying its operation to the values of it children. This is *constant folding*.

For an subgraph of constant nodes in a program, there will be a set of “top” nodes which are referenced by non-constant nodes. Since their values have been computed during constant folding, the top nodes contain all the information necessary for further compilation. Only the top nodes will be used in remainder of compilation process, specifically in induction variable detection and code generation. The other constant nodes in the program are ignored and thus effectively discarded.

The decision to compute the values for constant nodes with separate machine instructions or to encode them in immediate fields within instructions is determined during code generation. Most of the constants found, even the “top” constants, will never appear as code because they will be removed during some later program transformation, such as induction variable reduction.

2.7 Loop Invariant Detection

A *loop invariant* is a variable whose value is constant for the duration of the loop. Loop invariant detection requires knowledge of which variables and expressions are constants, so loop invariant detection follows constant propagation. Loop invariants are only marked at this point. Graph transformations to move loop invariants out of loops are made at a later pass, which also makes the transformations associated with other optimizations.

A node is *loop invariant* iff it is:

1) a constant,
2) a use with no reaching definitions within the loop,
3) an operator with operands which are loop invariant.

2.8 Induction Variable Detection

An *induction variable* is a loop based variable which takes a linear sequence of values. Because the expressions for induction variable computations can be built up from other induction variables, induction variable information is not immediately obvious to the compiler and must be discovered. *Induction variable detection* is the process of finding the induction variables in the program. This is an iterative process, where each iteration may find more induction variables based on the current known set. Induction variable detection is complete when no additional induction variables are found.

Induction variable detection relies on the information from constant propagation and loop invariant detection, and is also synthesized information. The algorithm used here was initially taken from [4]. A variable is an *induction variable* iff it is computed by one of the following expression patterns:

1) i = {i+c, i-c}, where c is a constant or loop invariant,
2) j = {i*b, b*i, i /b, i+b, b+i, i-b, b-i, or i+i}, where i is an induction variable and b is a constant or loop invariant.

Induction variable detection searches for linear recurrences, i.e. computations of the form j = c*i+d, which can be rewritten as a simple additions within a loop. Induction variables can defined in terms of *basic induction variables*, which are those variables whose calculation is a linear expression involving only themselves and a constant. The other derived induction variables can be grouped in *families*, i.e. sets of induction variables whose values will be linearly related to each other. Knowledge of the basic induction variable and the family of each induction variable allows strength reduction to be performed on the induction variable computations.

The original loop induction detection algorithm described in [4] only computes the *family*, i.e. the constant offset added each iteration, and does not directly deal with either: 1) the initial value of the induction variable or, 2) recurrences formed from loop invariants which are also induction variables in an outer loop. These omissions lead to retaining induction expressions which are more complicated than necessary, particularly in outer loops.

The algorithm described here has been modified to incorporate patterns with loop invariants and to record the initial value of the induction variable when it is a constant or when it is a loop invariant expression formed from induction variable in an outer loop. This modification leads to a nested definition of induction variable information. For instance, in the program fragment shown in Figure 29, the expression for j in the inner loop involves both the induction on j in the inner loop and the induction on i in the outer loop. Collecting this information together as {j, 1, {i, 3, 1}+2}, where the first term of each tuple is the basic variable, the second term is the loop increment and the third term is the initial value expression, allows induction variable information at different loop levels to be manipulated together. The discovery and use of nested induction variables has also been reported by Padua and Wolfe in [134].

```
Do
    i = i+1
    j=i*3
    Do
            j = j+2
            y = x[i,j]
    End
End
```

FIGURE 29. Program fragment with nested induction variables.

To see how the use of nested induction variables differs from the traditional flat induction variables requires a look at the intended use of the induction variable information. Induction variable information is used to make induction variable strength reductions. A *strength reduction* is the replacement of an operator or computation by a less expensive operation which computes the same function. The goal of strength reduction on induction expressions is to transform the induction expressions into the simplest, or least costly, expressions possible. The ideal form of an induction expression is usually an initial assignment to a constant value in the loop header with an increment by a constant value on each loop iteration. If initial value and nested information are not gathered, the only information available to the compiler is that a node supplies the initial value for the induction. The original initial value node will often be more complex than a simple assignment or an increment by a constant. This complex initial value node will be retained in the induction expression, although it will often be moved out of the loop.

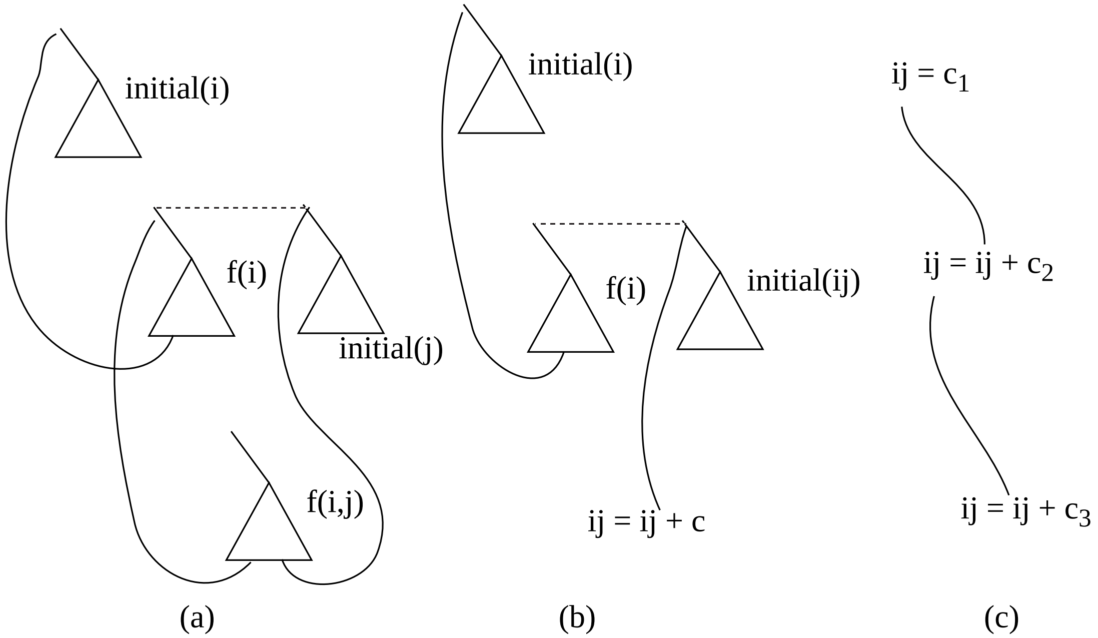


FIGURE 30. A Nested Induction Transformation.

The difference between multi-level and single-level transformations is illustrated in Figure 30. Figure 30(a) shows the original graph of an expression f(i,j) constructed from induction variables in nested loops. This program graph could, for instance, be produced by the x[i,j] reference in the inner loop in Figure 29. The expression to compute the address required to reference x[i,j] would be:

((j-lower_bound2)*dim2+(i-lower_bound1)*dimension1). (4)

The triangles in Figure 30 are meant to represent subgraphs of complex expressions such as (4). A strength reduction on the induction variable j, which does not use information about its initial value or dependence on i will result in the transformation shown in Figure 30(b). The original complex expressions involving i, which are needed to compute the initial value for the induction on f(i,j) will be retained. The equivalent program fragment is shown in Figure 31.

```
Do
    i = i+1
    j=i*3
    j’ = @x + ((j-lb2)*dim2 + (i-lb1)*dim1)
    Do
            j’ = j’ + dim2
            y = *j’
    End
End
```

FIGURE 31. Program fragment with rewritten inner induction.

Nested induction information allows the chain of expressions show in Figure 30(c) to be constructed directly. The original expressions will be discarded if they are not used elsewhere. The final version of the loop is shown in Figure 32. The complex expression for x[i,j] has been moved out of both loop and is left as the initial value of i’. However, as the initial values of i and j (i_init and j_init), and the array dimensions (dim1 and dim2) are likely to be constants, the initial value of i’ is likely to be a constant, and the entire expression could be replaced by i’ = c.

```
i’ = @x + ((j_init-lb2)*dim2 + (i_init-lb1)*dim1
Do
        i’ = i’ + dim1
        j’ = i’
        Do
                j’ = j’ + dim2
                y = *j’
        End
End
```

FIGURE 32. Program fragment with rewritten nested inductions.

Collecting nested induction information is straightforward and only requires a few simple modifications to the original algorithm. In the original algorithm, the initial value of the inductions were not collected and only constants were considered for the family information. The algorithm used here also collects initial value information, where possible. For each induction variable detected, a tuple of information is recorded: the *induction triple* is <*base_induction_node, family, initial_value*>. Each of these pieces of information is computed from a function which is dependent on the pattern of the expression forming the induction.

The *base induction node* is the original node with pattern i = i {+-} b from which the current node is derived. A base induction node is:

1) node(i), for i = i {+-} b,
2) base_node(i) for the patterns j = i {+-*} b.

The *family* of the induction node is the value which will ultimately be used to construct the induction increment once the induction is rewritten. The computation of the family is the same as in the original algorithm. The increment value, from which the family is derived, must be either a constant or the original node must be retained and used. The patterns for determining the family of an expression are:

1) c, for i = i {+-} c,
2) c, for i = j + c,

3) -c, for i = j - c,
4) c {*/} family(j), for i = j {*/} c, where c is constant,
5) the original node must be used, otherwise.

The *initial value* of the induction node is used, as its name implies, to give the induction expression its initial value. If there is a single constant reaching definition, the value of the constant is the initial value. If there is a single reaching definition, which is an induction expression from an outer loop, a nested induction expression will be formed. Otherwise the reaching definition nodes must be retained as the initial value.

The restrictions of a single reaching definition, which is a constant or outer loop induction, seems to limit the utility of this modification to a few special cases. To some extent this is true. However, this is a fairly common special case and it is often seen with multiple dimension array references. For languages like Fortran or C, where the array is one of the primary data structures, this case is quite common.

The initial value information is derived from the value of the reaching definition for the pattern i = i {+-} c, and the value of c and the initial value of the independent induction variable in the pattern j = i {op} c. The function for initial values is:

1) value(r), for i = i{+-}c, with a constant reaching definition r,
2) reaching(i), for i = i {+-} c, otherwise,
3) initial_value(j) {op} value(b), for i = j {op} b.

The result of these rules is a definition of induction variables which is potentially nested. The initial value of an induction triple can be another triple, e.g. {85,8,{45,1,2}}. As discussed earlier, and illustrated in Figure 30, a nested triple in the initial value position will indicate that an outer loop induction expression should be constructed when induction strength reduction is performed.

Given that this is a fairly simple modification to the original algorithm, it would be nice to be able to report a large performance benefit from using this modification. However, the primary performance benefit seems to be in Tortoise itself. Without numbers to support this claim, the graph is greatly simplified in a single pass because complex outer induction expressions are discarded. The performance results on the compiled code are mixed. Of the first 14 Livermore Loops, only Loop 12 and Loop 14 show noticeable effects of 10% and 5% performance improvement, respectively. The other 12 Loops show a performance improvements of less than 1%. Again, without quantification, the additional cost of finding and exploiting nested induction nodes seems to be low (or negative given the graph simplification), so this modification to the induction algorithm is probably worth using.

2.9 Iteration Distance Computation

The dependency information typically gathered for array references on high performance machines, i.e. vector and parallel architectures, is inadequate for software pipelining. Typically, only the direction (flow, anti-flow) and the special case of loop carried (flow in the current iteration) are considered [134]. Because of the overlapping of iterations which occurs in software pipelining, it is useful to have precise values for the number of iterations crossed on a flow or anti-flow dependency, when this can be determined. A loop with a recurrence is shown in Figure 33.

```
Do i =...
    X[i+1] = X[i] + X[i-1]
End
```

FIGURE 33. A loop containing a recurrence.

If it is known that X[i+1] forms a flow dependency with X[i] in the next iteration and X[i-1] in the following iteration, the value can be retained in a register for use in these iterations. This information is call the *iteration distance*. Like the induction variable initial value, the iteration distance can only be determined precisely in special, but common cases. The iteration distance can be found between array reference expressions in a loop where the expressions share the same family (see Section 2.8 on page 81). When the expressions share a family, it is possible to determine a precise, constant distance between the references. The method used in Tortoise to derive the iteration distance is to perform a symbolic arithmetic substraction between the array reference expressions using Mathematica's built in algebraic rules. This is more powerful than a commercial implementation, which would handle only a few predefined patterns, put not tremendously. If the result of the substraction is a constant, this is encoded as the iteration distance. If the subtracted expressions cannot be reduced to a constant, the fact that the references are dependent is retained. Symbolic data dependence testing is explored in [134].

The iteration distance is determined for every pair of array references. It can be thought of as a decoration on the dependency graph, but is encoded in a separate function -- iteration_distance: node × node → distance. The distance (d) is one of {constant, unknown, not_related}. Constants indicate that the references hit in a fixed number of iterations, in either direction: previous or subsequent iterations. A value of zero indicates that the references hit in the same iteration. Unknown indicates that the references are dependent, but the number of iterations at which they hit is not a constant. Not_related is the default case for references which are unrelated.

The iteration distance is used in two slightly different contexts and its meaning changes to match the context. As with the dependency graph, the iteration distance is used to determine and represent both data movement and operation scheduling. For operation scheduling, the distance between operations, in terms of cycles or slots in a schedule, is important and this distance is computed with respect to the top of the loop. When the iteration distance is used to determine a distance for data movement, the number of locations required becomes the important metric, and the distance is recomputed to be with respect to the definition points. These two functions are distinguished in Tortoise as the *node* iteration distance and the *operand* iteration distance, respectively.

2.10 Array Reference Refinement

When the data dependency graph is first constructed (see Section 2.3 on page 74), the information to distinguish between array references is not yet available. Once the induction variable information has been found and encoded in the iteration distance between array references, the dependency graph for array references can be refined. This requires examining all the dependencies between array references and modifying the dependency functions to reflect the new information for array references. This will tend to reduce the number of references which are dependent because the new information allows a finer discrimination.

Once the kill functions have been modified to reflect new array reference information, the data flow equations are re-solved using the new functions. Since it is much easier to rerun the same algorithms, no attempt is made to retain and incrementally modify previous data flow information. The definition chains are then reconstructed using the new data flow information. Once the definition chains have be reconstructed, the data dependency graph can be refined to reflect the new dependency information.

The graph transformations performed at this point are slightly different from those performed earlier when only the scalar information was available (see Section 2.5 on page 76). Array references typically depend on loop induction variables, so transformations to array references must retain information describing the relationship between the references with respect to the loop behavior. This information is the iteration distance found earlier (see Section 2.9 on page 87). When the dependency graph is modified to promote a loop carried dependency to a register, the iteration distance between the original nodes is retained and added to the new nodes. The iteration distance information is used during scheduling to determine the distance between the new nodes and during register allocation to determine the number of registers required by the node.

Suppose for instance, that loads for X[i] and X[i-1] are found and determined to be sharable as a CSE. The graph transformation, including the iteration distance decoration is shown in Figure 34.

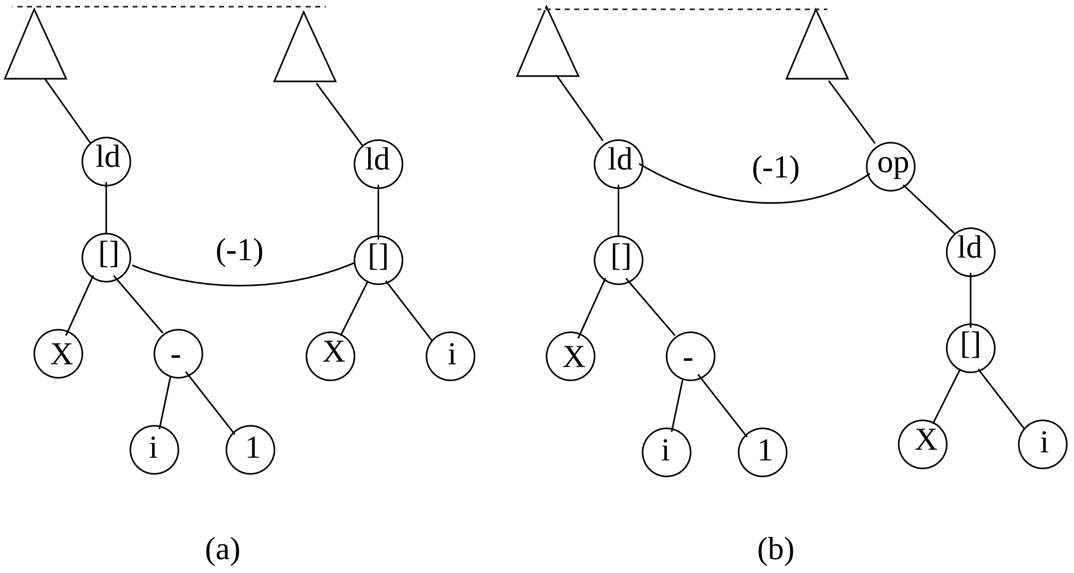


FIGURE 34. Array Reference Load CSE Transformation

On this transformation (array reference load CSEs), the original array reference subgraph is retained and a new intermediate operator is added between the use and the original load operator. The iteration distance decoration is added between the new operand marker and the load which is “preferred”. Later, during code generation, the iteration distance is copied to the parent node to mark the distance between the parent of the new operand marker and the preferred load. If it is determined to be advantageous to put the preferred load into a register and use it as a CSE, the iteration distance between the parent and the preferred load is used for scheduling and register allocation. If it is not advantageous to use a CSE, the original load is still present in the graph and can be used for code generation.

Whether it is advantageous to promote a load CSE into a register depends on the iteration distance and the number of registers available. Each iteration crossed produces another live value and requires an additional location to store. Because each iteration crossing consumes a register, on a machine with limited registers it is only beneficial to make this transformation for a small iteration distance. The limit is currently arbitrarily set to 2 iterations.

3 Machine Independent Optimizations

Following the reconstruction of the data flow graph, all the machine independent loop optimizations are performed en masse. The machine independent loop optimizations consist of induction variable strength reduction, induction variable CSE detection, and loop invariant code migration. The nodes involved are marked and processed (either moved or rewritten) in a single pass so that the original order within blocks can be maintained. This avoids re-sorting the operations to maintain the correct semantics between passes.

Even though all transformations are performed in a single pass, the transformations are independent of each other. The routine finds all the nodes which need to be processed and applies the appropriate action to each node.

3.1 Loop Invariant Hoisting

The loop invariant code migration algorithms are also standard [4]. The only modification is a restriction on the mobility of loop invariant nodes which contain definitions. Loop invariants can always be moved out of a loop, into the loop preheader block previously defined, if they do not contain a definition. If the loop invariant does contain a definition, e.g. an assignment, then care must be taken to insure that moving the definition does not change the semantics of the program. There are a number of cases where definitions can be safely moved, e.g. if the block containing the node dominates all exit nodes of the loop. None of these cases are currently exploited by Tortoise. Loop invariants are only moved if they do not contain a definition. The number of invariant nodes containing definitions are few enough in the benchmarks used that the special cases with definitions did not seem to be worth implementing.

3.2 Induction Variable Strength Reduction

Induction variable strength reduction was discussed earlier (see Section 2.8 on page 81). For each induction node, the induction triple is used to construct a pair of expressions to compute the initial value and the induction increment. These expressions are constructed directly from the induction triple and the nodes previously encoding the values are discarded, where possible. The algorithm proceeds from inner loops outward, placing nodes for the nested induction information in the appropriate loop preheader blocks (see Figure 30). If not enough information was available to allow recreation of the induction values, e.g. a constant value could not be determined for the initial value, the original nodes are moved out of the loop where possible and retained otherwise.

Induction variable CSE detection is also performed. Induction variable CSEs are found by grouping all inductions with the same family together. This is a specialized, but beneficial case of CSE detection. The reason that this is interesting, is that many machines provide an “indexed” load operation, where a constant offset can be provided with the index register. All the inductions with the same family will be able to share a single index register by providing a different constant offset. This transformation can save registers when compiling array expressions within loops. This could be considered the first machine dependent optimization performed. However, the capability of adding a constant offset to an index register on a load is quite common. It is common enough to be lumped together with the other machine independent transformations. The one part of this that is machine dependent is the size allowed for the offset. The size of the offset is typically much smaller than the address size of the machine and induction CSEs must be further divided into groups which are within the distance that can be encoded in the constant offset allowed in a machine instruction.

3.3 Type Propagation

Type information indicating which integers are addresses is synthesized at this point. The type information is only partially available in the original graph: at the leaves and at nodes containing operators which expect addresses, e.g. loads. The leaf nodes which define addresses are found and marked. Address information is then synthesized until a node which consumes and does not generate an address is encountered. Intermediate operators such as addition and multiplication are marked as being addresses. This information is used during instruction selection to determine which machine operator to use (see Section 4.1.3 on page 96).

3.4 Dead Code Elimination

A dead code elimination pass is performed which removes sections of the program graph made obsolete by previous passes. In particular, induction variable strength reduction tends to replace the original induction expressions with new ones. The original inductions expressions where are no longer used are discovered and marked during dead code elimination.

All memory stores and conditional operations are marked initially as live. Liveness information is then synthesized up the graph until no addition live nodes are found. Any nodes not found to contribute to a live node are marked as dead. The dead nodes are not removed from the graph, i.e. the graph is not reconstructed, but the dead nodes are ignored for the remainder of the compilation.

3.5 Summary of Machine Independent Transformations

Type propagation completes the analysis and machine independent transformation phase of the compiler. The methods and algorithms described are standard, with the exception of the detection of nested induction information, promotion of intermediates to registers across multiple loop iterations, and determination of the iteration distance between array references. Constructing nested induction variable information is unrelated to a particular computer architecture, or to software pipelining, but is interesting in its own right. Multiple iteration register promotion and the iteration distance determination are not required for software pipelining. However, the techniques required for software pipelining directly support multiple iteration register lifetimes, so this optimization is natural to use in conjunction with software pipelining.

4 Code Generation

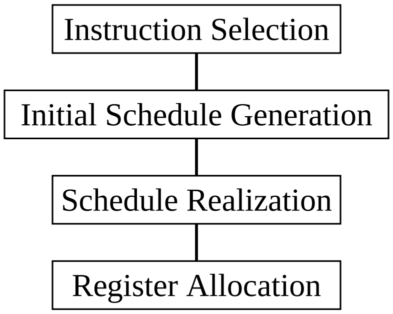


FIGURE 35. Tortoise Code Generation Phases

Although complex in its implementation, the code generation process is conceptually simple. Starting with the program dependency graph, the set of instructions to be generated is selected, the selected instructions are scheduled, and the registers to be used in each instruction are allocated. In Tortoise, instruction scheduling occurs in two phases. In the first phase a schedule is constructed for an abstract machine model which supports software pipelining. Then a second phase (*realization*) unrolls the pipelined schedule and implements the pipelined schedule in software on target architectures which do not support software pipelining. The phases in Tortoise’s code generator are shown in Figure 35.

4.1 Instruction Selection

A series of transformations are made to the program graph to change the IR from Gnu C operators and structure to machine operators and structure. The definition of the target machine is partially encoded in tables, e.g. the machine operations and instruction formats, and partially encoded in procedures, e.g. the procedures provided to determine which values will fit into constant or immediate fields. No attempt to formalize the machine description has been made, as that would be another study by itself. However, it is assumed that the operations and format of the target architectures will be very similar, and the differences between the target architectures will be in the format of the schedule. This is reasonable since we are attempting to focus on only the scheduling aspect of the target machines and have some freedom in the definitions of the architectures.

4.1.1 Initial Register Allocation

An initial register allocation, from an infinite set, provides a unique location for every potential register definition. Not every node is assigned a unique register. Nodes which share registers, e.g. during initialization, or when updating a register containing a loop induction variable, are found by merging shared uses in the program graph. Each shared use is indicated in the graph by multiple definitions reaching a single operand. These shared definitions are the individual links in a chain joining the nodes which share a register. A merging operation joins the nodes into a single group which is allocated a register.

4.1.2 Graph Structure Matching

For instruction selection to occur successfully, there must be a machine operation which matches the operator and number of type of operands for each node. A transformation is made to store nodes to copy the target node into one of the operands. This removes a special case check for store nodes during instruction selection. Call nodes are also modified to remove an extra node between the call node and the target. Both of these changes are local transformations and they are only performed to simplify the instruction selection routine.

4.1.3 Instruction Selection

Instruction selection determines an appropriate machine instruction for each node in the graph. This involves checking that there is a instruction format containing the correct type and number of operands, as well as the correct operator. At this point, each node should generate either 0 or 1 machine operations. A node can generate no machine operations because: 1) it provides a value which fits into a immediate field in another instruction or, 2) it is a node with sequencing or dependency semantics and does not indicate a machine operation (see Section 2.10 on page 89). Reorganization of the graph to insure that no nodes require more than one machine operation occurred in the previous pass (see Section 4.1.2 on page 96) and finding any nodes which require multiple machine operations at this point indicates a compiler error.

The instruction selection algorithm works from the top down. The machine instruction for a node is selected, followed by the instructions for the nodes operands. A simple heuristic is used to choose between instructions with multiple formats. Instructions with immediate fields are checked first to determine if the operands contain constants which fit the immediate fields. If the operands fit, the instruction is chosen. Otherwise, the instruction selection algorithm proceeds to check the next instruction in a list which is indexed by node operator and type. Because the target architectures have simple instruction sets with few instruction formats, this heuristic suffices.

When constants are found which do not fit immediate fields of other instructions, they are marked as executable and an appropriate instruction is selected to generate the constant.

4.1.4 Control Dependencies

Block termination nodes are now added to the graph. A control dependency between each node in the block and the block termination node is added to all the nodes in the graph. The block termination node is added to facilitate filling branch delay slots. The delay of the control dependency between the branch and the block terminator will be the number of delay slots. The delay for other operators will be zero. This allows any nodes which are not ancestors of the branch to be scheduled in the branch delay slots.

The addition of the control dependencies on all the nodes finishes the construction of the program dependency graph. This is the graph which is given to the scheduler. Every executable node has a machine operation and corresponding instruction format associated with it. The number of operands on the node matches the number of operands allowed in the instruction format. The scheduler will use this graph to construct machine schedules. This graph is not modified by the scheduler and is fixed for the remainder of the compilation. Some additional items will be added to decorate the graph, e.g. execution times, but the structure of the graph does not change.

For some types of scheduling, such as loop unrolling, creating a new version of the graph is enticing. The advantage would be that flow analysis, etc. could be performed again on the graph to obtain additional improvements in the schedule. However, the difficulties in making modifications to a graph structure argue for working with a list of machine instructions, rather than a graph. The scheduler currently implements a limited form of loop unrolling in conjunction with software pipelining. An optimization pass following this loop unrolling is not performed and the data structure used is a list of machine instructions. This simplifies the scheduler at the cost of some additional performance gains.

4.2 Instruction Scheduling

The scheduler accepts the Program Dependence Graph (PDG) along with accompanying tables and produces schedules for the target architecture. The selection of machine operations and instruction formats was performed previously. Register allocation and encoding the instructions in assembly language format is done later by other routines. The schedulers only function is to find a correct schedule, i.e. a correct ordering and timing of the operations. The output of the scheduler is an ordered list of instructions, organized in blocks, where each “instruction” is a record containing the necessary information to construct an assembly instruction and a pointer back into the PDG. This list of instructions is passed to the register allocation routines, and then to the assembly language code generator.

The scheduler constructs both software pipeline schedules for inner loops and basic block schedules for blocks which are not part of an inner loop.

4.2.1 Basic Block Scheduling

The basic block scheduler uses a list scheduling algorithm:

1) A topological sort is applied to a set of operations and dependencies to produce an ordered list such that if there is a dependency between two operations, the independent operation is placed first in the list:

$$\{V,E\} \rightarrow \langle o_1, o_2, ..., o_n \rangle, \text{ where } \forall \langle o_i, o_j \rangle \in E \Rightarrow i<j. \quad (5)$$

2) The ordered list of nodes is assigned an execution time such that the dependency ordering between operations (5) is still satisfied and, in addition, the minimum time dependency between dependent operations and machine resource constraints are also satisfied:

$$<o_1,o_2,...,o_n> \rightarrow <o'_1,o'_2,...,o'_n>, \forall <o'_i,o'_j> \in E => T(<o'_i,o'_j>)\ T(j)-T(i) \qquad (6)$$

In (6), $T(<o_i,o_j>)$ is the minimum time distance which must be maintained between o'_i and o'_j for correct execution and T(i) is the execution time of schedule position i.

The ordering of the two lists need not be the same. The routine assigning execution times may change the order of the operations to fill times left vacant due to instruction latencies, provided the original dependencies are honored. The topological sort in the first step simplifies the work required in the second. Once the sort has be performed, the scheduler is assured that the parents of each operation encountered will have been scheduled and that there are not circularities in the dependence graph. The scheduler only checks the execution time and operator on each parent to determine the earliest time at which the operation can be scheduled, and then searches for a slot in the schedule with enough resources to execute the operation.

The minimum time and resource constraints are not met is all cases, nor is this necessary. In the R3000, the resource constraints are complex because out-of-order completion is allowed, and there is only one result bus. The resource use for a typical multicycle instruction only allows execution of one instruction at the beginning and end cycles of the instruction. Finding a minimum schedule with this set of constraints without resource conflicts is a NP-complete tiling problem.

Scheduling this processor would be much easier if multiple results could be delivered each cycle, i.e. if there were only function unit resource constraints at the beginning of the instruction. This is the model which is used by the scheduler. Assuming that the processor busses are only taken at the beginning of instruction execution is not as accurate, but it is much easier to schedule and does not seem to be detrimental to performance.

Because the R3000 has pipeline hazard interlocks, the minimum time dependency does not have to be met in the schedule either. As with most current processors, the dependency relationship between instructions is encoded in the register usage. As an instruction enters the function units, the target register is marked busy and any subsequent instructions using the target register will be stalled until execution of the defining instruction has completed. This allows a correct schedule to be generated without padding dependent instructions with NOPs. This tends to decrease the size of the schedule and makes the schedule easier to read, but will increase the number of registers required in a compact schedule. This is because the target register is reserved during the time the instruction is proceeding through the function unit. This would not be necessary if registers were not being used to maintain dependency information.

The block scheduler produces a schedule which contains NOPs and then removes them as a separate pass. The NOPs are left in until the schedule is complete so that they are available to be filled by other instructions. This reduces the amount of instruction rearranging performed by the scheduler.

4.2.2 Inner Loop Scheduling

Inner loops are scheduled using software pipelining (see Chapter III, Section 3 on page 52). The software pipelining method is the same as used by Lam in [98]. First, an estimate for the length of the schedule is found. Then, instructions are placed within a schedule of the estimated length while checking for timing violations and resource constraints. If a schedule of the given length is found, the process terminates, otherwise the process is restarted with a longer schedule.

The initial schedule is constructed in a compressed form which could be directly executed on a machine with hardware support for software pipelining, i.e. the compressed schedule assumes the machine supports conditional execution and an indexed register file [143]. Before this schedule can be executed on an architecture without hardware support, it must be rewritten so that its execution will correctly implement the intended software pipeline. This involves the construction of separate sections of code for the prolog, kernel and epilog phases of the pipeline, along with checking to insure that there are enough iterations to enter the pipeline and cleanup sections to execute those iterations which do not fit the pipeline. Also, the loop must be unrolled so that any required register indices can be hard-coded directly into the schedule.

Once the schedule has been reconstructed, register allocation and spill code generation are performed using an interference graph coloring algorithm [24][33][21][15]. The completed schedule is then converted into the assembly language format of the target architecture and written to a file. This file is then assembled and executed using the target hosts software.

4.2.3 Terminology

Some terminology unique to software pipelining needs to be explained before delving into the scheduling algorithm. Figure 36 shows a software pipeline in uncompressed form (Figure 36(a)) and compressed form (Figure 36(b)). The uncompressed form shows the order and timing which would occur if a single iteration were executed. The compressed schedule shows the execution of the schedule when all the blocks of the schedule are executing simultaneously, i.e. during the kernel phase of the pipeline.

A single iteration of the loop is divided into a number of equal length pipe stages or *blocks*, as shown in Figure (a). The length of the block is the time between beginning successive iterations of the loop. The time between the start of successive iterations of the loop called the initiation interval. It will be referred to here as the *block length*. The *block count* is the uncompressed schedule length divided by the block length.

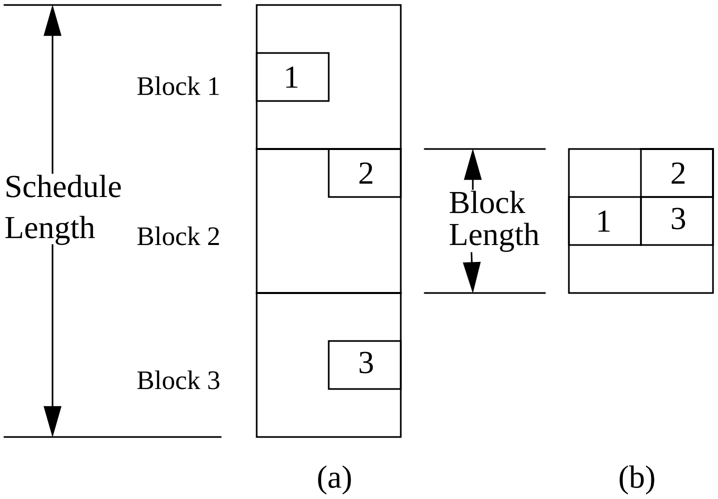


FIGURE 36. A Three Stage Pipeline Schedule

The block length and the number of blocks is determined by the latencies between operations and the resources required by those operations in the schedule. The intent is to make the block length as short as possible, trading off the number of blocks. The block length determines the execution efficiency when the full execution or kernel phase of the pipeline is reached. Assuming that the loop executes a reasonable number of iterations, this will tend to make the loop execute more efficiently.

As shown earlier (see Figure 14), there are three phases to execution of the software pipeline: *prolog*, *kernel*, and *epilog*. Correct execution during each of these phases must be insured. If the compressed schedule is executed on an architecture with hardware support, each operation in the schedule is tagged with a *block number* or other identification so that the operation is only executed when its associated block is executing. If the software pipeline is to be executed without hardware support, the block number is used to unroll the schedule to produce a purely software implementation of the pipeline.

As was mentioned during the discussion of the analysis phase of Tortoise, the *iteration distance* is the number of iterations between references with loop carried dependencies (see Section 2.9 on page 87). The iteration distance, where known, is used to determine the time between operations and the number of live values which are formed by overlapping iterations.

4.2.4 Initial Block Length Estimate

The task of finding a minimum block length for a schedule in the presence of resource constraints and a pipelined architecture is an NP-complete problem. So is finding a minium block length. However, because the scheduling algorithm repeats until a schedule of the given block length is found, we wish to estimate the block length as accurately as possible.

There are two fundamental limits on the block length of the schedule: the machine resources available to execute the operations and the latency between operations. Lower limits on the block length due to these factors can be independently computed and combined to give a good estimate on the block length of the schedule.

The block length for resource use is the number of resources required by the operations in the loop body, divided by the number of resources available each machine cycle. If the resources were all independent, the block length determined by resources would be:

$$\text{Block Length} = \text{Max}(\forall r \in R, \lceil (\sum r[n]) / r \rceil) \qquad (7)$$

Where R is the set of machine resources and r[n] is the machine resources required by a given node in the program graph.

Machine resources are not always independent. For instance, it may be possible to perform an integer and a floating point addition at the same time, but not two floating point operations. Or it may be possible to start another floating point addition in the cycle immediately following a floating point addition, but not following another operation. The resource equation can be elaborated as necessary to achieve a good estimate. However, if the function becomes too elaborate, the cost of estimating the block length can become expensive.

The other limitation on the block length is the amount of time necessary to compute recurrences, i.e. loop carried dependencies. The set of recurrences can be found by finding the cycles in the graph using the all-pairs shortest-path algorithm [54]. If we assume that all recurrences are independent, the required block length is the length of the time required to compute the longest recurrence. The block length required to compute any individual recurrence is the sum of the latencies for each operation in the recurrence, divided by the number of iterations which the recurrence crosses. If computing the recurrences where independent, the block determined by recurrence lengths would be:

$$\text{Block Length} = \text{Max}(\forall c \in C, \lceil t[c] / d[c] \rceil) \tag{8}$$

Where C is the set of recurrences in the loop, t[c] is the execution time or latency to compute the recurrence, and d[c] is the iteration distance of the recurrence. Assuming independence between resource use and recurrence computation, the estimated block length is the maximum of the two estimates:

$$\text{Block Length} = \text{Max}(\forall r \in R, \lceil (\sum r[n]) / r \rceil, \forall c \in C, \lceil t[c] / d[c] \rceil) \tag{9}$$

4.2.5 The Scheduling Algorithm

Once an estimate has been found for the block length, an empty schedule with the estimated block length is constructed and an attempt is made to fit the operations in the PDG into the schedule. Because software pipelining allows multiple iterations to execute concurrently, dependencies between iterations (recurrences) must be honored, as well as the usual inter-block dependencies. The intra-iteration dependencies produce cycles in the program graph and these must be handled correctly.

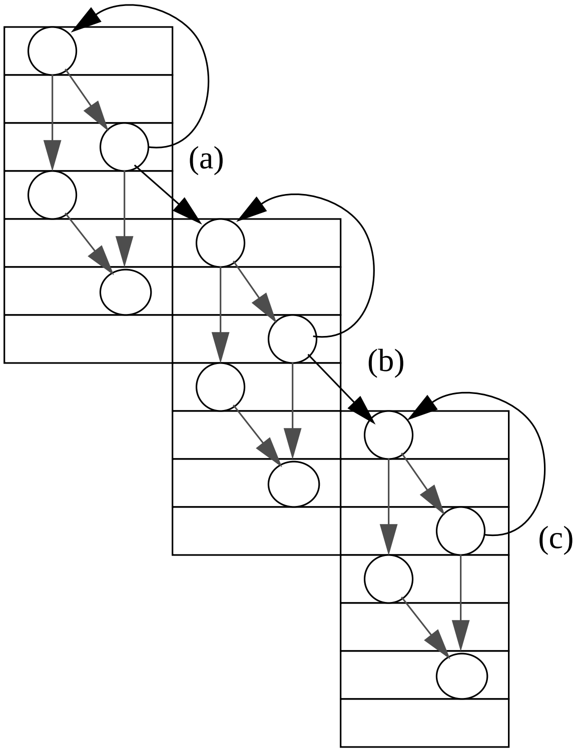


FIGURE 37. Formation of Strongly Connected Components

Cycles in the program graph will result in one or more *strongly connected components* in the graph. A strongly connected component is a subgraph in which there is a path between every pair of nodes. Figure 37 illustrates the formation of strongly connected components within the program graph. The cycles in the graph are formed by intra-iteration dependencies, labeled in the Figure as (a), (b) and (c). *Every* cycle in the graph contains an intra-iteration dependency. The other, inter-iteration dependencies, form an acyclic graph. This suggests a method for scheduling SCCs, which is to schedule each SCC using only the inter-iteration dependencies.

However, the execution constraints on the inter-iteration dependencies must still be honored. As each operation is scheduled it must be scheduled both late enough so that all of its parents have been scheduled (the normal direction) and also early enough so that it executes before its descendents execute in subsequent iterations. If it is not possible to find a position in the schedule meeting these criteria with enough machine resources to execute the operation, scheduling is restarted with a longer block length. The moves the beginning of the next iteration, giving more freedom in placing the operation. This algorithm is guaranteed to terminate, because in the limit, the iterations will not overlap and the algorithm will in effect be scheduling a DAG, which is always possible.

A strongly connected component in a graph does not necessary cover all the nodes in the graph. There can be nodes not included in the SCC with dependencies into or out of the SCC. The graph may contain one or more strongly connected components. If each strongly connected component in the graph is replaced by a single node, an *acyclic condensation* of the graph is formed. Because there are no cycles between strongly connected components, they can be scheduled as far apart a necessary. Scheduling SCCs further apart lengthens the total uncompressed length of the schedule, but does not change the block length.

Also, because there are no cycles between strongly connected components, a topological ordering can be found for the SCCs. This is the final step required to complete the scheduling algorithm. The scheduling algorithm is:

1) Estimate the block length of the schedule and initialize an empty schedule of that length.
2) Find the strongly connected components.
3) Produce the directed acyclic graph of strongly connected components.
4) Sort the DAG.
5) Schedule each strongly connected component.
6) If a schedule of the given block length cannot be found, increase the block length and start over.

The algorithm for scheduling a strongly connected component is:

1) Form the DAG of only inter-iteration dependencies for the component.
2) Sort the DAG.
3) Schedule each operation checking to insure that dependencies on both ancestors and descendents are honored.
4) Signal failure if the operation cannot honor all dependencies.

4.2.6 Schedule Realization

Once a software pipeline schedule has been found it must be implemented or *realized* on the target architecture. This involves the construction of separate sections of code for the prolog, kernel and epilog phases of the pipeline, along with code sections to insure that there are enough iterations to enter the pipeline and cleanup sections to execution the iterations which do not fit the pipeline. Also, the loop must be unrolled and any required register indices must be hard-coded into the schedule.

Ignoring for the moment the issue of unrolling the register indices, a pictorial view of the sections of code to be generated is shown in Figure 38. The prolog and epilog are generated by unrolling the compressed schedule the correct number of times, while selecting the instructions from the appropriate set of blocks. The prolog is constructed by selecting instructions from the first block, followed instructions from both the first and second blocks, etc. This unrolling continues until all the instructions are selected, at which point the prolog section is finished and the kernel section begins. The epilog is constructed is a similar manner, except that it starts with all but the instructions executing and continues until only instructions from the last block are executing. The cleanup loop executes in sequential fashion and is created from the uncompressed schedule. The entire schedule is joined together by conditional code to direct execution into the appropriate sections of code.

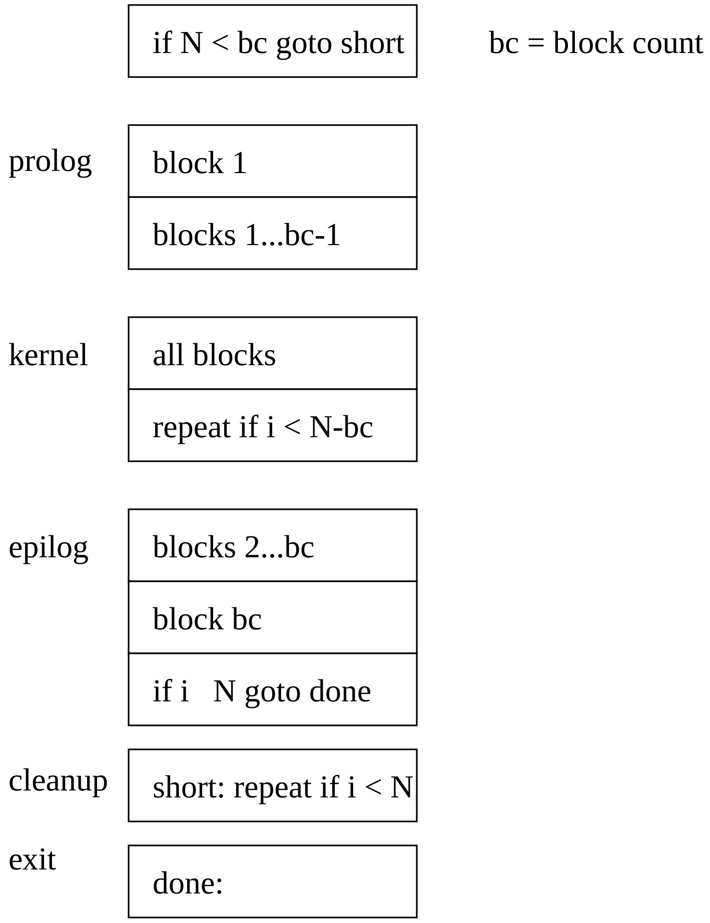


FIGURE 38. Software Pipeline Realization

The schedule is not completely realized yet as the register indices have to be rewritten. The scheduling algorithm allows multiple live instances of temporary, i.e. register, values to occur. One way this can happen is shown in Figure 39. Allowing multiple iterations to execute concurrently allows the lifetimes of a variable from different iterations to overlap, creating multiple live values. Also, the optimization promoting array references into registers allows these dependencies to span multiple iterations (see Section 2.10 on page 89). This also allows multiple live instances to be created.

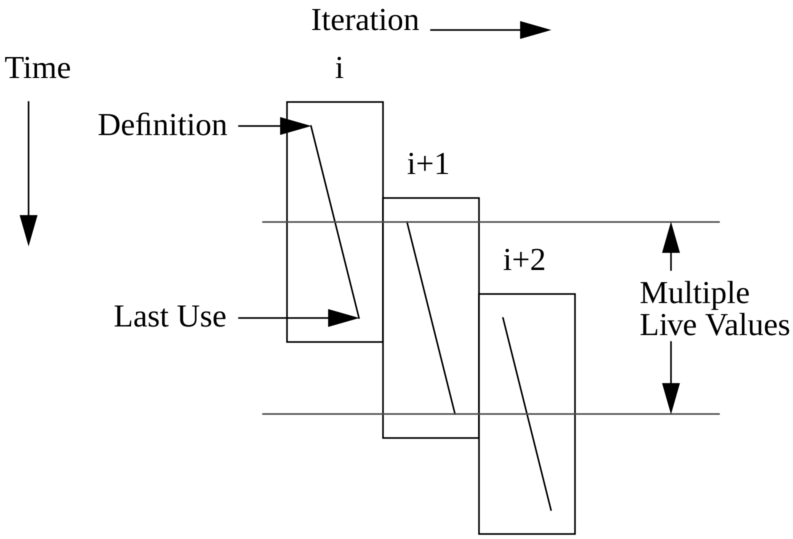


FIGURE 39. Multiple Live Register Values in a Software Pipeline

A queue-like mechanism must be implemented to handle multiple live instances of register values. Conceptually, a new value is pushed onto the top of the register queue at the definition. Each use retrieves a value offset from the top of the queue by the number of new definitions which have been pushed. The last use removes the oldest value from the queue.

If the target architecture supports indexed references into the register file, the register queues can be created by incrementing the register index at the beginning of each block, which is also the initiation of a new iteration. If hardware support is not available, queues can be implemented either by copying the registers in the queue to save the old values, or unrolling the schedule and hardcoding the register indices directly in the schedule.

Copying registers introduces some overhead, which may or may not be detrimental to performance depending on the size of the schedule and the hardware resources available to perform the operations. Unrolling the schedule can greatly increase the code size. The algorithm used in Tortoise employes a combination of unrolling and copying to implement register queues without excessive overhead in terms of either execution time or code size.

The schedule produced by register unrolling is the same as shown in Figure 38, with some minor modifications. The kernel of the schedule is unrolled and the register indices of register with multiple live values are hardcoded in the schedule. The number of times the kernel must be unrolled is the least common multiple of the iteration distances on all uses of registers with multiple live values. The least common multiple is required because the set of iteration distances forms a group which must be cycled through completely for the register indices to become invariant. If the least common multiple is so large that unrolling would produce an excessive amount of code, the kernel is not unrolled and the registers are instead copied to implement the register queue behavior.

The register queues are implemented via copying in the epilog and the clean-up code. These sections will be executed only a few times, so the execution overhead of copying is not as large as in the kernel. The combination of unrolling the kernel and copying registers in the other sections produces code with low execution overhead for the majority of iterations, i.e. in the kernel, and low code size overhead in the other portions of the loop.

4.3 Register Allocation

Once the schedule has been realized for the target architecture, register allocation is performed using a priority based graph coloring algorithm [24][33]. During realization, software pipeline specific constructs, such as register queues, have been re-implemented in standard machine operations, so special register allocation techniques are not required [148][149]. Unfortunately, the process of realizing the schedule alters the program structure, requiring the flow analytic information to be reconstructed so that an interference graph of register uses can be constructed.

The first step in the process of register allocation is to perform flow analysis on the registers (an infinite set of pseudo-registers at this point) to determine liveness. An interference graph is constructed from the liveness information. The liveness information is constructed on the instruction level, rather than the block level. This is more expensive than using block level liveness, but gives more precise information.

Once the interference graph is constructed, register allocation and spilling is performed. The spilling algorithm uses the “spill everywhere” heuristic of [24]. This is a somewhat weak heuristic and the allocator runs into the same problem with global variables encountered by Briggs et al. in [21]. This problem is exacerbated by the scheduler which is optimizing for a short block length, potentially at the expense of a longer uncompressed schedule length. No weight is given to register resources in the scheduler, which tends to spread the last operations to be scheduled widely apart in an attempt to fit those operations into a few remaining schedule slots. In [20], Bradlee et al. report that a better schedule results when the code scheduler knows about register constraints. On an architecture with a small number of registers, such as the MIPS architecture, a compression type scheduler, which is careful with registers will probably produce better code than the unconstrained scheduler used here [77]. At the very least, the register allocator should give some weighting to register use when scheduling operations.

4.3.1 Finishing Up

To finish the compilation, the completed schedule is reformatted in the assembly language of the target architecture. Subroutine entry and exit sequences are added, along with associated register save and restore sequences. The assembly code is written to a file, which is then assembled and executed using the target architecture’s software. Execution analysis is done using a combination of pixie and a modified version of a runtime analysis program, xsim, developed by Mike Smith [162][164][125].

CHAPTER V
EXPERIMENTS AND RESULTS

In our experiments we apply a set of static scheduling techniques to a set of benchmarks and then simulate the running of the benchmarks on a set of machines. The intention is to determine the effectiveness of the various scheduling techniques and also to examine interactions between compile time scheduling techniques and architectural features.

The cycles executed, and other performance characteristics, are found using *trace analysis*. In trace analysis, the executable is modified to produce a history or trace of the program's execution. The trace consists of a list the basic blocks executed and memory references. The trace is analyzed by another program to produce execution statistics. We use the MIPS utility *pixie* to instrument the executable for our R3000 experiments, and a modified version of pixie for the Aurora III experiments.

We use several programs to analyze the execution traces and produce execution statistics. All of the analysis tools employ the same methodology: using the basic block and memory references recorded in the trace, examine the executable to determine the sequence of instructions executed in the block. With this more complete record of the program's execution history, emulate enough of the architecture's behavior to estimate the execution time. The emulation can range from a simple version which just increments a cycle count for each instruction, to an elaborate simulation of the internal state of the processor, including updating instruction queues and cache lines.

The analysis tools used in these experiments are derivatives of two programs: *pixstats* and *xsim*. Pixstats is the MIPS utility supplied for program analysis [163]. Pixstats gives detailed processor execution statistics, but assumes a perfect memory system and does not give information on cache effects. We use pixstats in our runs on the R3000 comparing compilers and a modified version of pixstats for the experiments involving floating point latencies [125].

Pixstats does not collect or report cache performance, so for the experiments where we report cache behavior, we augment pixstats with the cache analysis tool *CacheUM* [125]. Using the address references, CacheUM emulates the behavior of a two level cache memory system and can be configured in a number of ways, including setting the total cache size and the line size.

The analysis tool used for the Aurora III based experiments is based on xsim, developed by Mike Smith [162][164]. Xsim has been substantially modified by Tom Huff and Mike Upton to model the behavior of Aurora III [119]. We further modified the Aurora III integer and floating point models to combine them into one model. The combined Aurora III model was then used as the basis for the DAE and VLIW processors models.

The scheduling techniques used are loop scheduling techniques, including basic block scheduling, loop unrolling, and software pipelining. The machine configurations cover a range of instruction issue methods, including scalar, VLIW, superscalar, and DAE. A scalar architecture with in-order issue with out-of-order completion is used as the base architecture for comparison. In addition, some of the experiments also vary other features of the machine model such as cache sizes, memory latencies, and instruction queue sizes. The full set of machine configurations investigated in shown in Table 2.

We are exploring the interactions of the components of highly complex systems with a large number of parameters. In order to minimize this complexity we use the standard experimental approach of fixing all put one or two parameters, which are allowed to vary during the course of the experiment. In this way we explore along one dimension in the space defined by our system, and then move to the next dimension.

TABLE 1. Machine Configurations

Machine Name	Description	Issue Rate	Memory System
R3000	R3000.	Single	Perfect - no miss penalty.
R3000 fp	R3000 with floating point latencies varied from 5 to 10 cycles, with and without pipelining.	Single	Perfect - no miss penalty.
R3000 pipe	R3000 with hardware support for software pipelining (indexed register operations and conditional prolog and epilog instruction execution).	Single	Perfect - no miss penalty, 8-64k word direct-mapped primary I and D-cache, 256k word secondary cache. 2/20, 3/20 and 5/141 memory penalties (2 cycle penalty for first level cache miss, 20 cycle penalty for second level cache miss, etc.).
R3000 a3	R3000 with Aurora III cache configuration.	Single	2k byte I-cache, 32k D-cache, 64k byte secondary I and D-caches. 2/20 memory penalty.
Aurora III	Aurora III superscalar architecture.	Dual	2k byte I-cache, 32k D-cache, 64k byte secondary I and D-caches. 2/20, 5/50, and 10/100 memory systems.
Aurora III scalar	Aurora III scalar architecture.	Single	2k byte I-cache, 32k D-cache, 64k byte secondary I and D-caches. 2/20 memory penalty.
Aurora III vliw	Aurora III VLIW architecture.	Dual	2k byte I-cache, 32k D-cache, 64k byte secondary I and D-caches. 2/20 memory penalty.
Aurora III dae	Aurora III DAE architecture.	Dual	2k byte I-cache, 32k D-cache, 64k byte secondary I and D-caches. 2/20 memory penalty.

First we investigate the performance of our scheduling algorithms on a scalar architecture. This gives us a baseline with which to compare the performance of our scheduling algorithms on other machine configurations. Next, the latencies in the floating point unit are varied while the other parameters of the processor are fixed. The goal of this experiment is to investigate the efficacy of our scheduling algorithms in dealing with medium and long latency operations. Then we turn to the Aurora III and attempt to determine what factors contribute to its increased performance over the baseline scalar processor. We then compare the superscalar Aurora III to similar machines with VLIW and DAE architectures. Finally, we examine the interaction of cache effects with our scheduling policies.

1 Scheduling a Scalar Architecture

Our first set of experiments compares the performance of several compilers and scheduling techniques on a current scalar architecture, the MIPS R3000, whose associated compiler is a commercial leader. These compiler experiments have several goals: We establish that Tortoise is capable of state-of-the-art code generation, which ensures that the instruction mixes are representative of those which would be produced by a good optimizing compiler. We establish that our compiler is not handicapping any of the scheduling techniques by, for instance, not providing information such as operation dependencies, which would be available in a good optimizing compiler. In addition, examining the behavior of the scheduling techniques in the simpler scalar environment yields a baseline for comparison, before proceeding into more complex architectures.

The commercial compiler we use for comparison is the MIPS CC Version 2.10 with -O2 level of optimization. The MIPS compiler is used for comparison because it is the vendor’s compiler for the architecture chosen and, as noted above, is one of the industry leaders. Gnu C Version 2.2.2 is also in these experiments with options -O2 (referred to as “gcc”) and -f unroll-all-loops (referred to as “gcc unroll”). Gcc normally unrolls loops to a fixed body size. A modified version of gcc is constructed to unroll loops a specified number of iterations, in this case 4 iterations (referred to as “gcc unroll

4”). Our compiler is run with block scheduling (referred to as “block”), software pipelining (referred to as “software pipeline”), loop unrolling to match standard gcc unroll counts (unroll), and loop unrolling 4 iterations (referred to as “unroll 4”).

TABLE 2. Compiler/Technique Performance on a Scalar Architecture

	Gnu C Variants				**Tortoise**			
Loop No.	**gcc**	**gcc unroll**	**unroll count**	**gcc unroll 4**	**block**	**software pipeline**	**unroll**	**unroll 4**
1	0.828	0.873	4	0.889	0.728	1.12	1.16	1.1
2	0.884	0.902	1	0.844	0.724	0.897	0.774	0.893
3	0.751	1.07	4	1.09	0.6	0.922	1.12	1.09
4	0.953	0.871	1	0.93	0.801	0.996	0.85	1.08
5	0.889	0.969	4	0.955	0.728	0.999	1.1	1.08
6	0.764	0.843	2	0.824	0.76	0.926	0.847	0.835
7	1.09	1.09	1	0.869	0.967	1.15	0.983	0.674
8	0.833	0.833	1	0.691	0.872	0.884	0.872	0.408
9	1.17	1.17	1	0.84	0.971	1.09	0.971	0.921
10	0.98	0.98	1	0.88	0.947	1.01	0.947	0.391
11	0.847	1.16	4	1.07	0.734	1.	1.19	1.15
12	0.77	1.05	4	0.976	0.715	1.22	1.18	1.08
13	0.917	0.917	1	0.849	1.34	1.43	1.34	0.563
14	0.888	0.911	2	0.826	1.03	1.19	1.09	0.949
High	1.17	1.17		1.09	1.34	1.43	1.34	1.15
Mean	0.884	0.962		0.884	0.819	1.04	1.01	0.77
Low	0.751	0.833		0.691	0.6	0.884	0.774	0.391

The cycles executed, and other performance characteristics, are found by running pixie and analyzing the trace using pixstats, a MIPS utility supplied for program analysis [163]. Pixstats assumes a perfect memory system and does not give information on cache effects. Initially, we will ignore cache effects to simplify the number of parameters in our machine model. Later, we will explore cache effects in some of the architectural experiments.

The results of these various compilers and techniques, applied to the first 14 Livermore Loops are shown in Table 2, on page 117. For each benchmark/compiler optimization, the speedup, i.e. ration of number of cycles executed over the MIPS compiler, is shown. The high and low values, and the harmonic mean for the set of 14 Loops is also shown for each compiler/technique in Table 2 and in Figure 40.

Running these techniques on a scalar processor shows some interesting results. First, gcc generally produces slower loop code than the MIPS compiler, although when loop unrolling is turned on, the performance is within 4% of MIPS. Loop unrolling to 4 iterations has worse performance than unrolling to a fixed size. Although gcc is not adequately instrumented to show the cause of the decreased performance, in our compiler this type of unrolling causes excessive register spilling. Gnu C does not do interval analysis and has a relatively poor register priority function, so very large basic blocks tend to cause the compiler to generate excessive register spills. This quickly negates any advantage of unrolling a large loop.

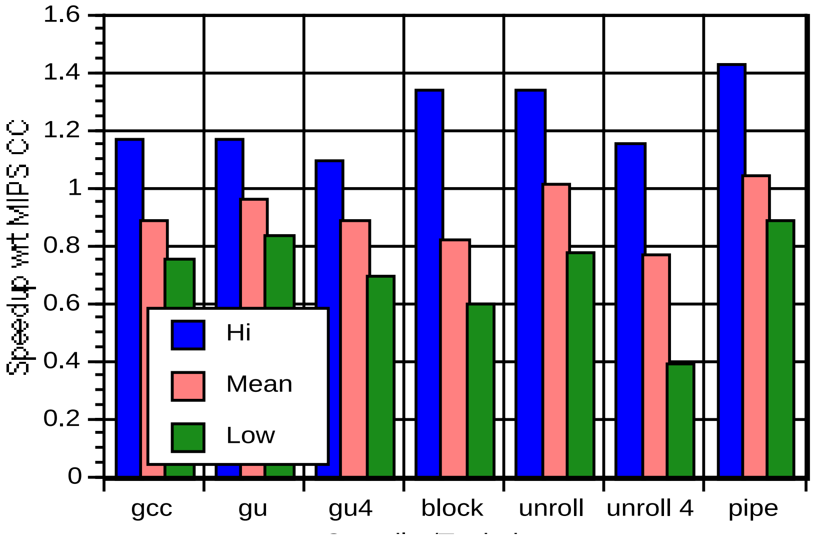


FIGURE 40. Compiler/Technique Speedup on Scalar Processor

1.1 Register Use

The registers consumed by each of the scheduling techniques for the first 14 Livermore Loops is shown in Table 3 and in Figure 41. The number of registers shown in the table is the number of registers which would be allocated if an infinite set of registers was available.

TABLE 3. Registers Use vs. Scheduling Technique

	Gnu C Variants				**Tortoise**			
Loop No.	**block**	**integer pipeline**	**unroll**	**unroll 4**	**block**	**float pipeline**	**unroll**	**unroll 4**
1	8.	11.	13.	13.	5.	8.	11.	11.
2	12.	13.	19.	24.	4.	4.	4.	4.
3	7.	10.	11.	11.	3.	4.	7.	7.
4	12.	15.	15.	16.	4.	4.	4.	4.
5	8.	11.	13.	13.	3.	3.	3.	3.
6	13.	16.	20.	22.	2.	2.	2.	2.
7	9.	12.	15.	19.	7.	9.	7.	20.
8	22.	26.	22.	37.	24.	24.	24.	35.
9	6.	9.	7.	19.	14.	16.	14.	46.
10	7.	10.	8.	22.	9.	10.	9.	42.
11	7.	10.	12.	12.	3.	3.	3.	3.
12	7.	10.	12.	12.	2.	4.	7.	7.
13	20.	22.	21.	67.	6.	8.	6.	29.
14	19.	22.	33.	51.	6.	9.	12.	22.
High	22.	26.	33.	67.	24.	24.	24.	46.
Avg.	11.2	14.1	15.8	24.1	6.57	7.71	8.07	16.8
Low	6.	9.	7.	11.	2.	2.	2.	2.

Figure 41 shows a marked difference in the number of integer registers consumed by different techniques. There is almost no increase in the number of floating point registers used, except for unroll 4, where the number of number of floating point registers used doubles from the other methods. There are a number of causes for the differences in register use between the techniques. The integer registers are used in these loops primarily for address calculations and induction variables. The induction variable analysis and optimization passes of the compiler will assign multiple registers to an induction variable when it is in an unrolled loop if the stride is not constant or is too large to fit in a memory immediate offset field. The stride may also be assigned to a register. This will tend to allocate integer registers in proportion to the amount of unrolling. The floating point register consumption is probably due to operations migrating over wider ranges as the body becomes larger under the various scheduling techniques.

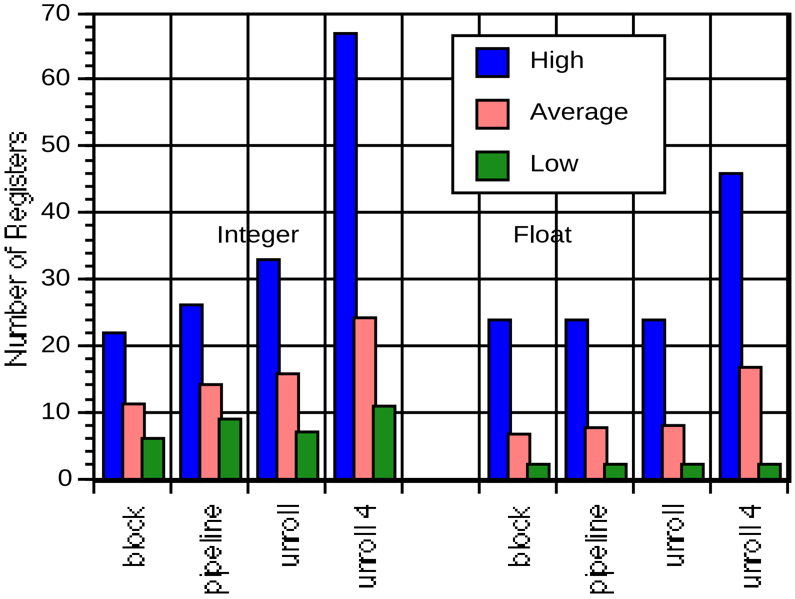


FIGURE 41. Registers Use vs. Scheduling Technique

1.2 Code Size

Code size is another parameter affected by these scheduling techniques. Software pipelining and loop unrolling trade larger code sizes for optimization opportunities and hopefully better performance. Code size can become an important performance factor due to its effect on cache behavior. The number of instructions generated for the first 14 Livermore Loops is shown in Figure 42 for each of the compilers and scheduling techniques used. While the overall increase in program size is not large (about 20% in the largest case), the increase in the size of the working set can be much more dramatic and will put pressure on the cache. We will examine this question more thoroughly in Section 4 on page 147.

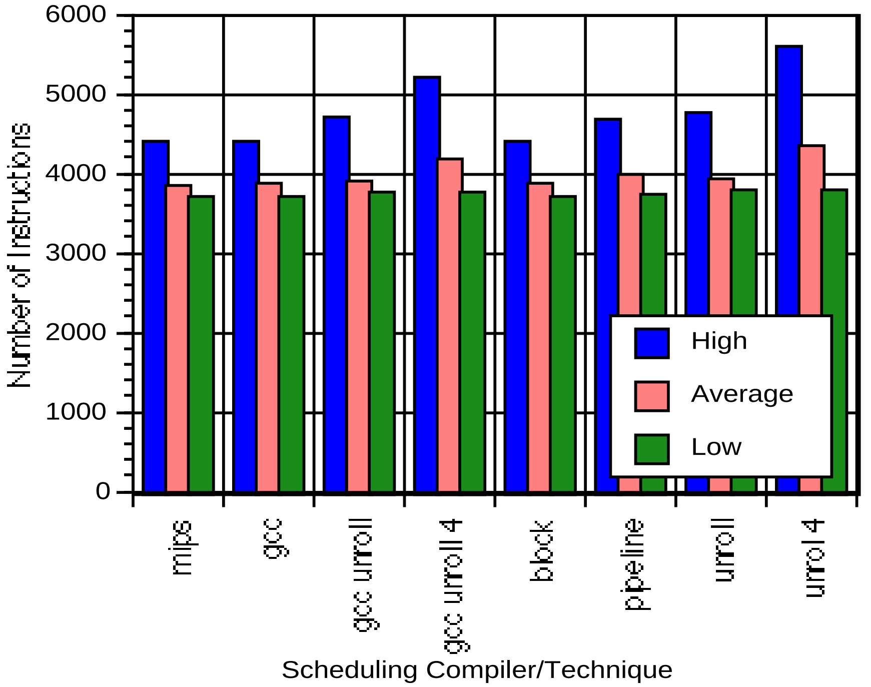


FIGURE 42. Code Size vs. Scheduling Technique

Table 4 shows a comparison between the performances of the code produced by each of the scheduling techniques. As expected, loop unrolling and software pipelining produce code which performs better than the code produced by block scheduling. Unroll 4 produces code with inferior performance to the code produced by block scheduling, probably due to increased register consumption. The comparison of software pipelining and loop unrolling shows an interesting correlation with the unroll count. Generally, the code produced by software pipelining performs better than the code produced by loop unrolling where the unroll count is 1 or 2 and worse than the code produced by loop unrolling where the loop count is four. This leads to the possibilities that either: 1) the criteria for determining the loop count is incorrect or; 2) software pipelining and loop unrolling are complementary techniques that could be used together, by selecting the best technique for each circumstance.

TABLE 4. Scheduling Techniques Performance Ratios

	Gnu C Variants			Tortoise		
Loop No.	**pipelined vs. block**	**unroll vs. block**	**unroll 4 vs. block**	**unroll count**	**pipelined vs. unrolled**	**pipelined vs. unroll 4**
1	1.53	1.59	1.52	4	0.965	1.01
2	1.24	1.07	1.23	1	1.16	1.
3	1.54	1.86	1.82	4	0.826	0.846
4	1.24	1.06	1.34	1	1.17	0.925
5	1.37	1.51	1.49	4	0.908	0.924
6	1.22	1.11	1.1	2	1.09	1.11
7	1.19	1.02	0.698	1	1.17	1.7
8	1.01	1.	0.468	1	1.01	2.16
9	1.12	1.	0.949	1	1.12	1.18
10	1.07	1.	0.413	1	1.07	2.58
11	1.36	1.62	1.57	4	0.843	0.866
12	1.71	1.65	1.51	4	1.04	1.13
13	1.06	1.	0.42	1	1.06	2.53
14	1.15	1.06	0.917	2	1.09	1.26
High	1.71	1.86	1.82		1.17	2.58
Mean	1.24	1.19	0.866		1.02	1.19
Low	1.01	1.	0.413		0.826	0.846

The unrolling algorithm used by gcc unrolls loops to fixed maximum size. This criteria has a number of desirable features: Smaller loops will benefit more from loop unrolling since instructions and overhead removed form a larger percentage the execution time of the loop (see Section 1.2 on page 45). In addition, there are potential detrimental cache effects from the amount of code produced by unrolling larger loops. In addition, as mentioned earlier and shown in Figure 41, on page 120, unrolling large loops can consume a large number of registers.

The possibility that software pipelining and loop unrolling are complementary techniques has been mentioned in previous studies [77][102]. The best approach would probably be to develop a hybrid algorithm which applies both unrolling and software pipelining.

The most surprising result of this set of experiments is that even on an architecture with relatively little amounts of parallelism, software pipelining outperforms loop unrolling. The R3000 has short operation latencies, even in the floating point unit and the units are which are not pipelined, so the execution of operations intended for the same unit cannot be overlapped. Overall, there is a small amount of parallelism to exploit and it is surprising that removing operations with loop unrolling is not more beneficial.

In addition, software pipelining uses fewer registers while requiring a comparable amount of code space. It would seem that software pipelining should consume more resources, because software pipelining is supposed to be exploiting parallelism by overlapping the execution of more operations, while using more instructions and registers to do so. This seems not to be the case for this architecture, where software pipelining uses the same amount of code space and fewer registers.

An advantage of software pipelining is that it “unrolls” only enough to fill open latencies and stops when there are no idle operation slots to fill or improvements in the schedule are not possible because of operation dependencies. This will tend to use less resources than an algorithm that is less sensitive to the code and architecture being scheduled.

2 Scheduling for Long Operation Latencies

Software pipelining should schedule operations with long latencies, especially pipelined operations, better than other scheduling methods. Since the function units on the R3000 are not pipelined and have relatively short latencies, the experiments on the R3000 did not address this issue. In this section we will describe and give results of an experiment designed to explore the effectiveness of the scheduling techniques at exploiting parallelism in the form of pipelined function units.

In this experiment, we will vary the latency of the floating point add and multiply units, rescheduling the benchmarks each time to match the latency of the target machine model. We will use the output of the MIPS C compiler as a base against which to compare the other scheduling techniques. Block scheduling, loop unrolling, and software pipelining are tested under (double precision) floating point add latencies which range from 2 to 7 cycles and multiply latencies which range from 5 to 10 cycles. This range of latencies is chosen because it matches current architectures on the low end (2 cycles), and what is thought to be probable for new processors on the high end (5 to 7 cycles).

A new analysis tool is necessary, as pixstats is configured to match the R3000 parameters. For that reason fpaUM, which was developed by David Nagle at the University of Michigan for the Aurora project, is used with pixie to provide performance data [125]. FpaUM allows the latencies of floating point operations to be set in a machine configuration file. FpaUM also allows function units to be pipelined, if desired.

Figure 43 shows the results of scheduling the first 14 Livermore Loops while varying the scheduling technique, floating point latencies and whether the floating point operations are pipelined. The performance metric is the ratio of harmonic means of the number of cycles executed for code produced by each of the scheduling technique and machine model with respect to the code produced by the MIPS C compiler an R3000 configuration. The number shown is the inverse of speedup, i.e. higher numbers represent longer execution times.

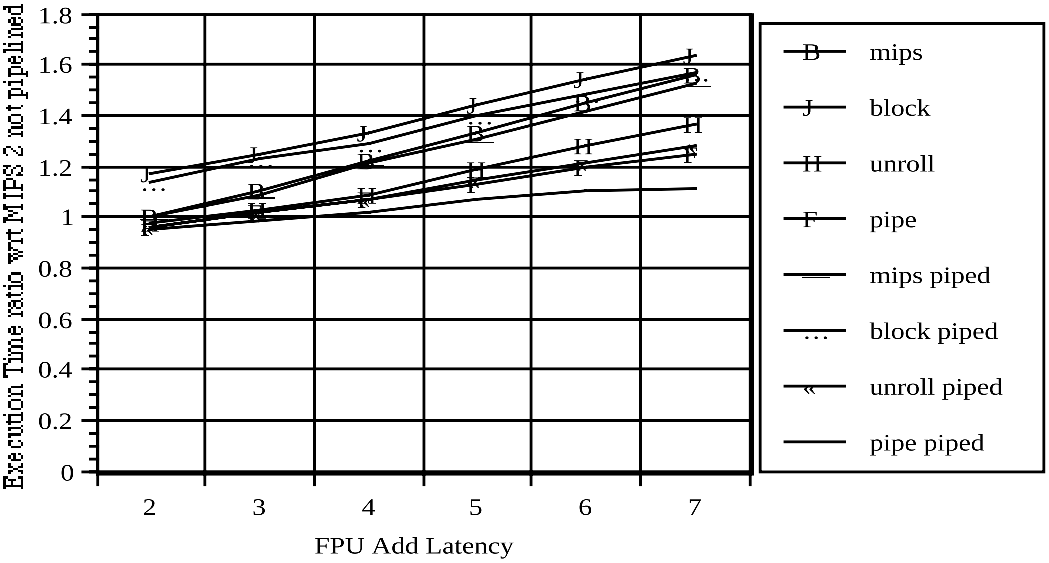


FIGURE 43. Execution Time vs. Increasing FPU Latency

The performance of the code produced by loop unrolling, software pipelining and the MIPS compiler are all fairly close on the R3000 configuration (FPU Add Latency 2). Pipelining the floating point operations on this configuration does not greatly improve performance. Block scheduling produces code which has almost 20% lower performance on this configuration. As the floating point latencies are increased, the execution times of all runs increases. MIPS (the MIPS compiler scheduling for the R3000) increases the fastest, followed by loop unrolling and last by software pipelining. We conclude that software pipelining is better at handling this type of parallelism, although the improvement over loop unrolling is only slight until the pipelines become moderately deep.

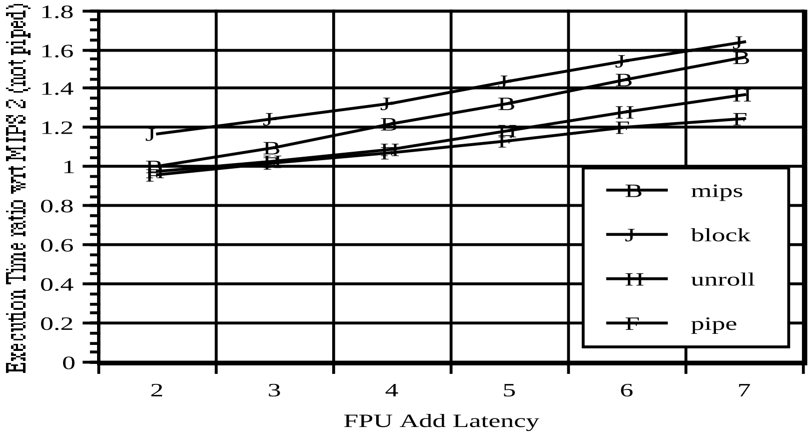


FIGURE 44. Execution Time vs. Increasing FPU Latency (FPU not pipelined)

Figure 44 and Figure 45 show the same information as Figure 43, split to show separate pipelined and non-pipelined runs.

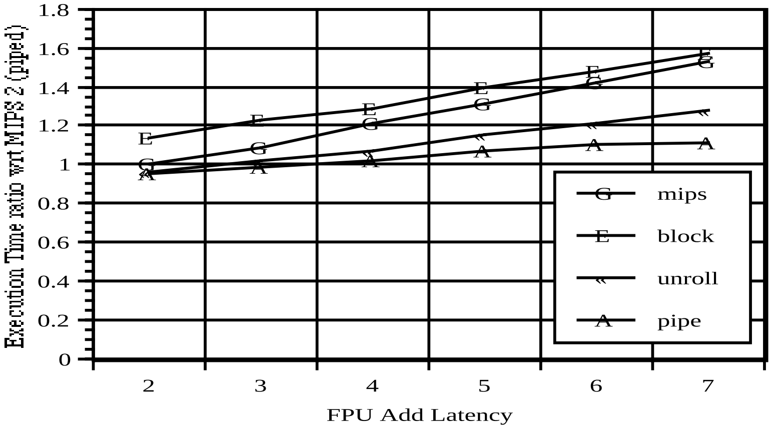
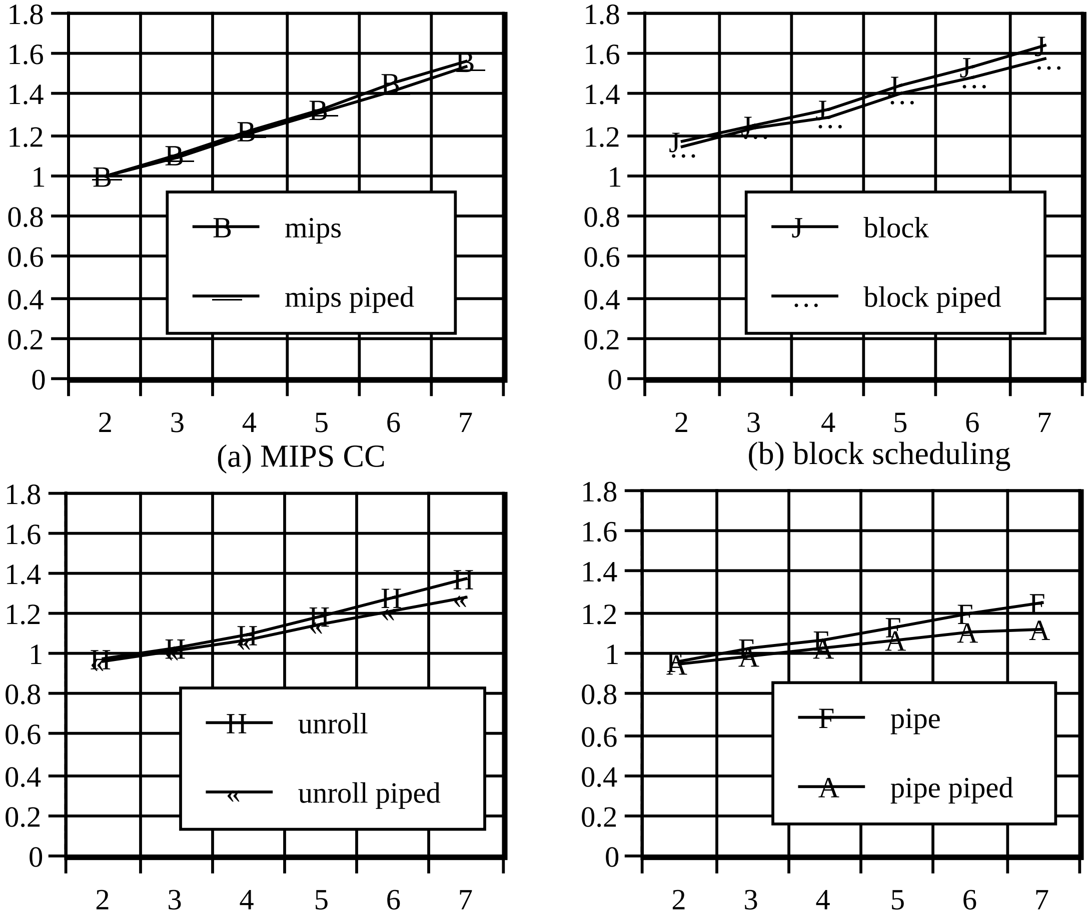


FIGURE 45. Execution Time vs. Increasing FPU Latency (FPU pipelined)

Two sets of runs for pipelined and non-pipelined function units show similar behaviors. Figure 46 divides the runs by technique, showing the difference in execution times between pipelined and non-pipelined floating point units. The non-pipelined execution times increase a little faster and the spread between the slowest and fastest runs is a little wider. But the difference is relatively small (less than 10%), even on the configurations with deep pipelines.



(a) MIPS CC

(b) block scheduling

(c) loop unrolling

(d) software pipelining

FIGURE 46. Execution Time Pipelined vs. not Pipelined FPU

The graphs presented so far show the results of increasing the latency of floating point operations given a fixed cycle time. This may be the case when designing a processor, but a more likely scenario is that dividing the floating point units into more pipe stages will allow the cycle time to decrease. Figure 47 assumes a fixed latency in the floating point unit and a decreasing cycle time. In this case we would derive a substantial benefit from increasing the number of pipe stages, even without better scheduling technology. Better scheduling provides additional performance.

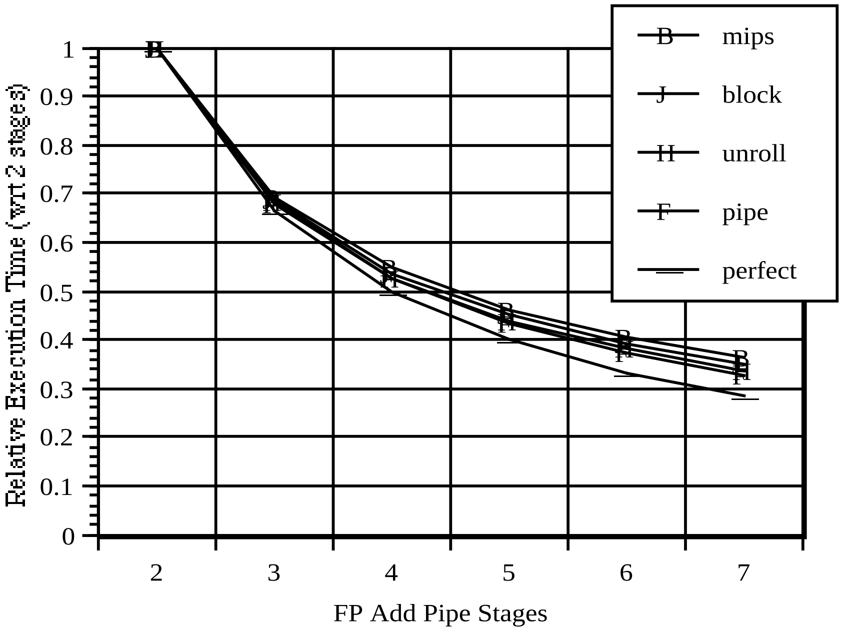


FIGURE 47. Execution Time vs. Increasing FPU Pipe Stages (Constant Latency)

Of course, this scenario of a fixed floating point latency and a decreasing cycle time is wildly optimistic because it assumes that the latency of *everything else* goes down proportionally, including memory latencies. This is also unlikely to be the case. The truth lies somewhere between these two extremes: Some benefit is derived from increasing pipe stages, along with cost. The point at which two effect balance with determine what the optimal number of pipe stages. Aggressive scheduling such as software pipelining will push the balance toward more pipe stages, i.e. more benefit will be derived from each additional pipe stage.

3 Scheduling and Issue Policies

In this section we describe a set of experiments designed to explore the interaction between an architecture's instruction issue policy and the scheduling techniques used by the compiler. Our goal is to compare block scheduling, loop unrolling and software pipelining on scalar, VLIW, DAE and superscalar architectures. We have already examined the compiler and scheduling techniques in some detail. However, for this experiment we need a more general analysis tool than either pixstats or fpaUM. We will still use trace based simulation and analysis to derive our performance figures. The analysis tool used is these experiments is based on xsim, developed by Mike Smith [162][164].

Xsim has been modified by Tom Huff and Mike Upton for the Aurora project at the University of Michigan [119]. The current version of the Aurora processor, the Aurora III, is a superscalar processor. We will use the Aurora III as the superscalar architecture in these experiments and modified versions of the Aurora III for the VLIW and a DAE architectures.

3.1 Aurora III

The Aurora III is a superscalar architecture which can issue two instructions per cycle if there are no data dependencies between instructions in the instruction window. Execution of the integer, floating point and memory units are decoupled. Coordination between the sub-systems is by a set of instruction, load and store queues. Ordering within each sub-system is supported by register score boards. In-order issue with out-of-order completion is supported by result reorder buffers.

The Aurora III has 32 integer and 32 double precision floating point registers, although most of our experiments use 16 double precision floating point registers to simplify comparison with the R3000.

The Aurora III has a two level cache: In the current simulations, the first level, on-chip caches are a 2k byte direct mapped I-cache and a 32k byte direct mapped D-cache. The second level caches is a 64k byte direct mapped I-cache and a 64k byte direct mapped D-cache. The final system will probably be 64k byte secondary caches, but the memory latency will probably be 150 to 200 cycles rather than the 20 cycles generally used for these simulations. In addition there will be branch prediction with instruction prefetching, which is not currently implemented. The memory latencies used is these simulations, except where we indicate otherwise, is a 2 cycle penalty for a first level cache miss and 20 cycles for a second level cache miss. There is also a 4 word store write buffer, which is used to collect and optimize writes.

The floating point unit has separate add, multiply and divide units. Both the add and multiply latencies are three cycles. The processor also has 64 bit wide data paths and supports double precision floating point load and store instructions. There are two result busses, a result reorder buffer and a store reorder buffer.

3.1.1 Aurora III Scalar Performance

The Aurora III incorporates a number of improvements over the R3000, including multiple result busses and 64 bit data paths[1]. The performance benefits of these features are worth investigating before looking at dual issue. For this set of experiments we define a scalar Aurora III, where the processor has been constrained to only issue a single instruction each cycle. Figure 48 shows the speedup of the code produced by each compiler/technique running on a scalar Aurora III, compared to the performance of the code produced by MIPS compiler running on the on the R3000. Generally, we will compare our code to the performance of code produced by the MIPS compiler running on the R3000 to have a standard for comparison[2].

[1] Later MIPS processors, such as the R4000, also implement 64 bit instructions.

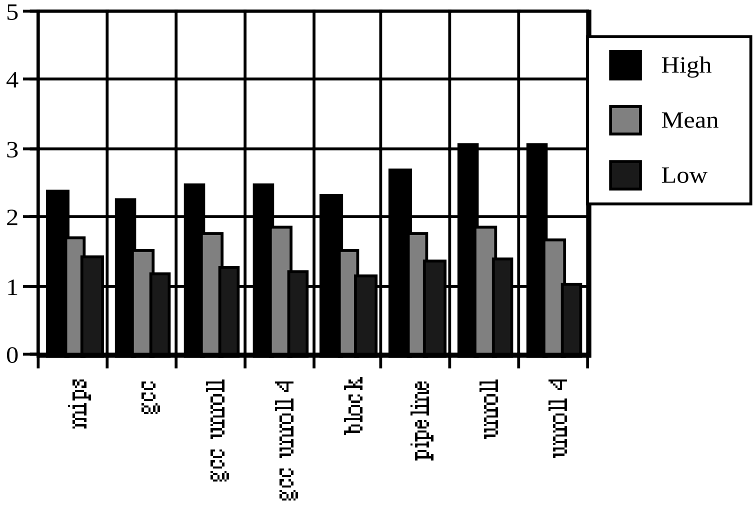


FIGURE 48. Scalar Aurora III vs. R3000 w. MIPS CC

Most of the compilers produce code with performance speedups in the 1.7 range, even without dual issue. There is a wider variation in the performance of our compiler using software pipelining and loop unrolling and either gcc or the MIPS compiler. This is due to one or two vector loops hitting the processor just right, but this does not raise the harmonic mean by much.

3.1.2 Double Precision Floating Point Loads and Stores

There are a number of features in the Aurora III which contribute to the 1.7 speedup over the R3000. Some of these features affect scheduling and need to be discussed before we proceed to examining dual issue.

[2] The base processor is an R3000 with a cache configuration matching the Aurora III cache configuration.

The Aurora III has 64 bit wide data paths and provides double precision floating point load and store instructions. This capability gives performance benefits in excess of just saving a cycle per each load or store, because rewriting two loads with a single double precision load removes scheduling constraints caused by the way these memory operations are handled.

The MIPS assembler expands double precision floating point load and store opcodes into two instructions. Our compiler uses this capability and treats these opcodes as one long instruction. This has scheduling implications. First, these compound instructions will not fit in a branch delay slot. This is one scheduling constraint. In addition, the Aurora III can only issue one memory operation per cycle, or only one memory operation per dual issue pair. We are still scheduling for a scalar architecture to this point, so the second constraint does not effect us yet, but this constraint will have an effect when we enable dual issue. Providing double precision load and store instructions removes these scheduling constraints and allows a more compact schedule to be generated.

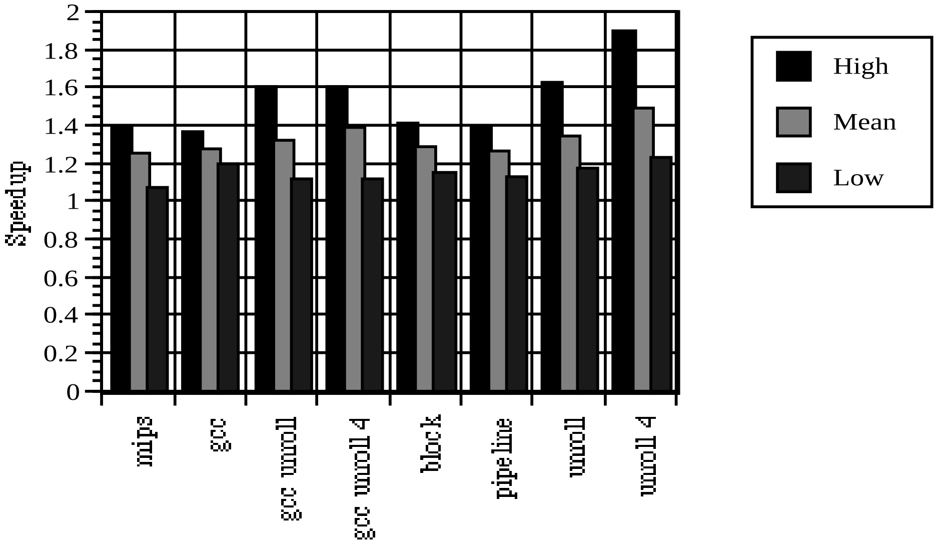


FIGURE 49. Scalar Aurora III Double vs. Single Load/Stores

Figure 49 shows a mean performance increase of about 1.25 due to providing double precision load and store instructions. In this figure, each compiler/technique is compared to itself with and without double precision load and store instructions. So, for instance, unroll 4 does not suddenly have better performance than software pipelining. However, unroll 4 does benefit more from the new instructions than the other compilers/ techniques. Because of the large number of registers consumed, unroll 4 tends to generate a larger number of load and store instructions to spill register contents. So unroll 4 derives more benefit from the new instructions than do the other techniques.

3.2 Decoupled Execution

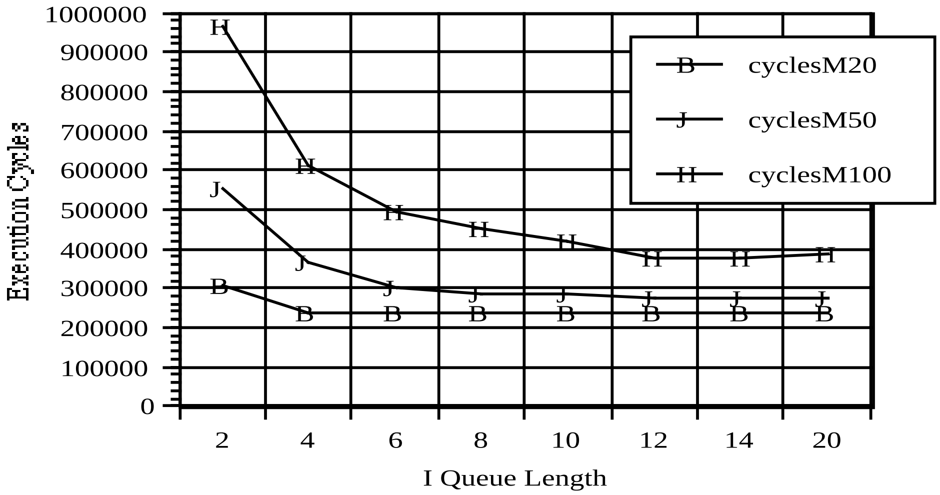


FIGURE 50. Aurora III Cycles vs. I-queue Length

Another feature of the Aurora III which effects scheduling is that the integer and floating point processors are decoupled. The instructions in the current instruction buffer in the instruction fetch unit are examined and either: 1) block because one of the registers is marked busy. 2) are sent directly to the integer unit. 3) are placed in the I-queue (instruction queue) to be delivered to the floating point unit. 4) are placed in the I-queue and the L-queue (load queue) to be send to both the floating point and memory units.

The queues allow decoupled execution of the sub-processors. In code without a recurrence between the floating point and integer processors, the integer processor tends to execute several cycles ahead of the floating point processor. The execution delay between the two processors hides much of the memory delay as in other decoupled processors, e.g. the DAE architecture in [154].

Figure 50 and Figure 51 show the execution time and I-queue stalls for Livermore Loop 2 under increasing I-queue length for memory systems with latencies of 20, 50 and 100 cycles. This execution behavior is similar to the decoupled behavior reported in [157].

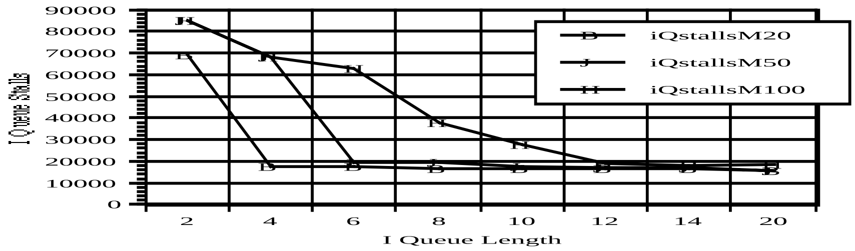


FIGURE 51. Aurora III Stalls vs. I-queue Length

Decoupled execution has scheduling implications because it tends to hide memory delays. This tends to decrease the relative performance of a VLIW execution model. In the VLIW model, NOPS are inserted in the schedule to remove pipeline hazards. In code such as Livermore Loop 4 (see Figure 52), an occasional data dependency, which cannot be resolved at compile time, will require the scheduler to treat the data dependence as a recurrence. The schedule will be padded with NOPS to satisfy the data dependency. In code such as this, where the data dependency is only occasionally a true dependency, decoupled execution would allow the next loop iteration to begin once the dependency is resolved. However, the VLIW version, the loop has been padded with NOPS. which must always be executed, even when there is no dependency. The next iteration must execute the NOPs before the next iteration can begin execution, which lowers the overall performance of the VLIW schedule. The ability to dynamically resolve pipeline hazards in the superscalar execution model is a decided advantage over VLIW in this case.

```
for (j=5; j <= N; j+=5) {
    temp = temp - X[lw] * Y[j];
    lw = lw+1;
    X[k-1] = Y[5] * temp;
}
```

FIGURE 52. Livermore Loop 4 - Occasional Data Dependency

3.2.1 Dual Issue Performance

When dual issue is enabled (see Figure 53), the mean performance speedup goes up to 2.3 time the R3000 and the performance variation of all the schedulers increases. The performance of code which cannot dual issue does not increase, while the performance of code which 100% dual issues can double. The performances of all three versions of gcc along with the MIPS compiler improve relative to our compiler with both software pipelining and loop unrolling. Only block scheduling and unroll 4 lag in performance.

At this point we are still running code which has been scheduled for a scalar processor. A scheduler which has a more accurate model of the architecture should have better performance. We will still use software pipelining and loop unrolling, but change Tortoise's model of the machine to match the dual issue nature of the Aurora III.

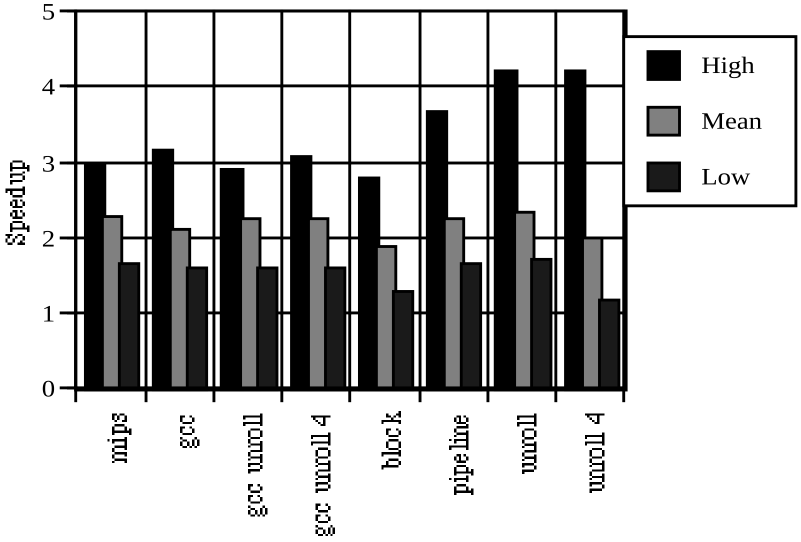


FIGURE 53. Dual Issue Aurora III vs. R3000 w. MIPS CC

3.2.2 Scheduling Models

Since we can gain a factor of two by dual issuing on the Aurora III, the percentage of dual issued operations will be our primary concern. Table 5 shows the percentage of dual issues which occur randomly when scheduling using a scalar machine model. As can be seen in the table, the percentage of dual issues is fairly low at 44%. This represents a speedup of dual versus scalar issue of about 1.3. We will introduce two machine models to attempt to improve on the percent of dual issues over using a naive scalar model.

The first model will treat the Aurora III as a VLIW architecture when scheduling operations. Because the Aurora III uses a fixed instruction window, a VLIW scheduling model is a good approximation for scheduling operations. However, the lifetimes of registers are different in a static superscalar architecture than in a VLIW architecture.

TABLE 5. Percent Dual Issue under Different Scheduling Models

Loop No.	scalar schedule	VLIW schedule	Double Latency
1	39.8	95.6	88.4
2	62.8	91.1	63.5
3	50.	98.2	69.3
4	14.4	72.5	64.5
5	12.5	55.1	32.6
6	58.8	63.8	81.8
7	72.5	96.	85.
8	48.3	84.4	34.
9	53.	75.5	51.6
10	49.8	84.2	31.8
11	15.7	66.1	97.1
12	42.4	97.2	91.6
13	46.6	76.4	56.1
14	51.6	80.6	72.6
High	72.5	98.2	97.1
Average	44.1	81.2	65.7
Low	12.5	55.1	31.8

In a static superscalar architecture, the register is live at the end of the operation even if this is in the middle of the instruction window. So register allocation must use a scalar paradigm. We first construct a VLIW schedule and then treat the operations as a linear scalar schedule in the register allocation routines.

Because registers are marked live in the scoreboard at the end of each operation, the machine can resolve pipeline hazards and stall when a data dependency exists. We can use this to overcome the problem of padding a schedule when there are unresolvable data dependencies (see Section 3.2 on page 133). Our scheduling algorithm first pads instructions to the width of the instruction window to avoid losing synchronism of the window and operation pair. Then, any NOPS pairs filling an entire cycle/instruction window are removed.

Table 5 shows that we have almost doubled the percentage of dual issues. However, some of the operations measured as dual issue are actually NOPS, and the performance only increases about 5% over the naive scalar model. The performance of the VLIW model compared to the MIPS compiler on the R3000 is shown in Figure 54. Another problem is that even though we are getting more dual issues, other bottlenecks are limiting performance, e.g. data dependencies and cache performance.

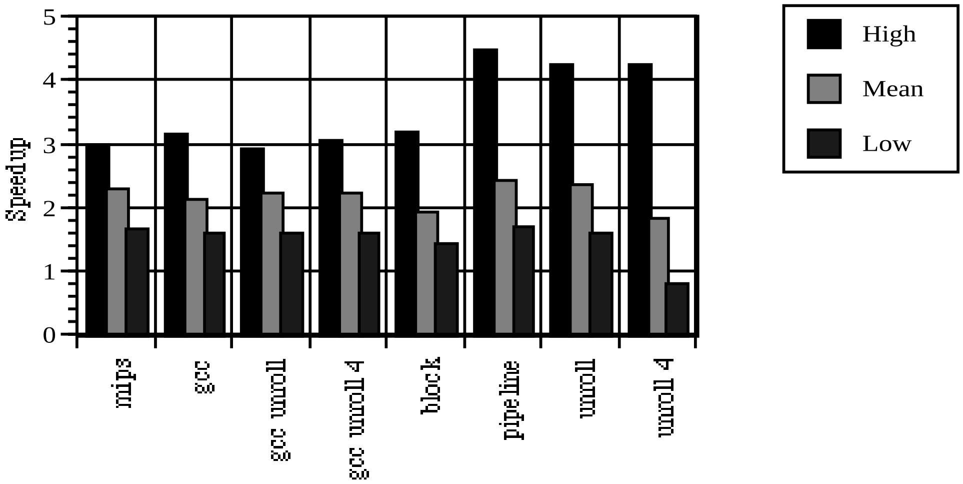


FIGURE 54. Dual Issue Scheduling (VLIW Model) vs. R3000 w. MIPS CC

We will introduce another scheduling model, to see if we can do better than our VLIW model. A superscalar architecture can also be modeled as a scalar architecture with the operation latencies doubled. The results of this scheduling model are shown in column 3 of Table 5 and Figure 55. The percentage of dual issues drops off some, because we are no longer inserting NOPS to pad the instruction windows. However, the performance is close to the performance of the VLIW model.

Figure 56 shows the speedup of dual issue (VLIW model scheduling) over scalar execution (scalar model scheduling) on the Aurora III. The performances of each compiler/technique are compared individually and are not scaled to each other. The mean improvements are in the range of 1.2 to 1.4. The loops with the worst performance in unroll 4 actually lose performance due to the padded instruction windows and additional registers allocated.

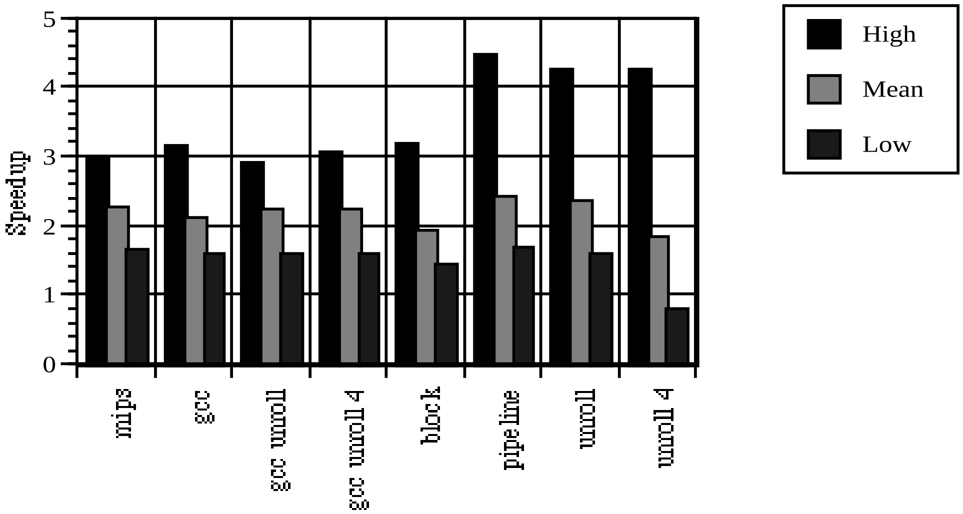


FIGURE 55. Dual Issue Scheduling (Latency Doubling Model) vs. MIPS CC

Figure 57 shows the registers used by software pipelining under the different machine models. Both the VLIW and double latency models use slightly more registers than the scalar model. The average number of floating point registers is consistently lower than the number of integer registers, by a margin on the order of 50%. The maximum numbers of floating point and integer registers is roughly the same. This indicates that a machine with only one half as many floating point registers as integer registers, like the R3000, probably has too few floating point registers for the type of applications represented by the Livermore Loops.

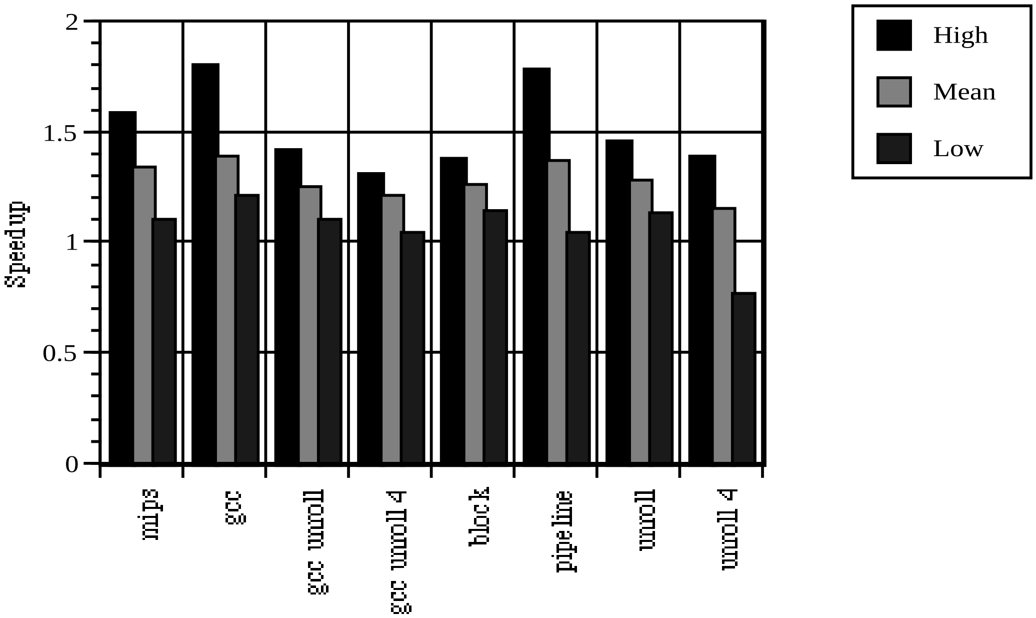


FIGURE 56. Dual v.s Scalar Issue (VLIW Scheduling Model)

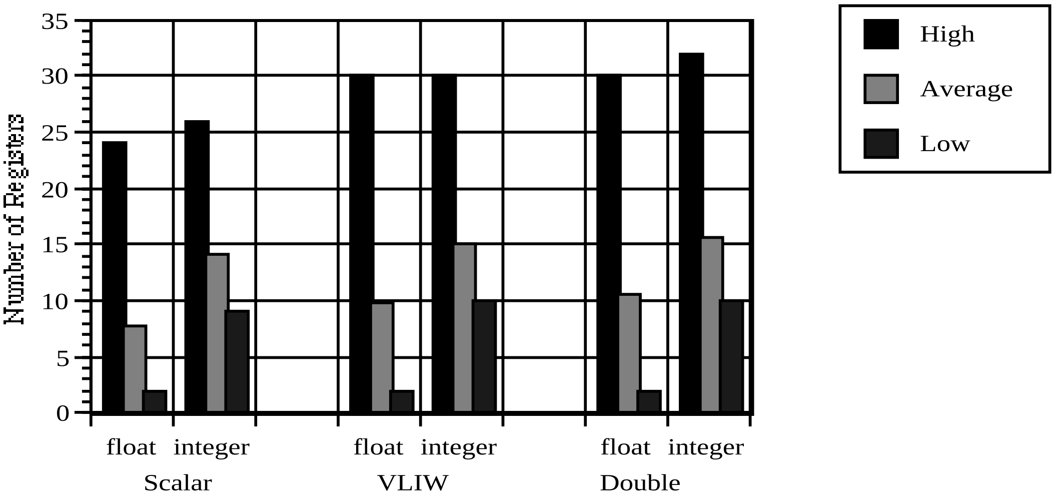


FIGURE 57. Register Use vs. Issue Models with Software Pipelining

3.3 Comparisons with VLIW and DAE

Having put all the machinery in place to compile using different scheduling algorithms, and simulate the programs and analyze the performance, we want to compare architectures with different issue models to our scalar and superscalar architectures.

One difference between a VLIW architecture and a static superscalar architecture is its use of registers. In a VLIW, the target registers become live at the end of the cycle when the operation exits the function unit, rather than at the end of the instruction which defined the target register. In addition, source registers are live until the end of the cycle in which they are redefined, rather than at the end of the instruction in which they are redefined.

Figure 58 shows an example of a superscalar register lifetime using a two cycle add operation. In this example, r1 is used and the destination register in the first instruction. In a superscalar machine, r1 would be marked live immediately following the first instruction. If the operation has a two cycle latency, any operations using r1 in the next two cycles would be delayed until the add operation has completed execution. If the result of the add operation is only used once, by the multiply in cycle 4 here, r1 is available for reuse. In a superscalar machine, r1 is marked busy in cycles 2 and 3, and available at cycle 4.

```
1:  r1 = add r2, r3    -
2:  ...                r1 live
3:  ...                r1 live
4:  r4 = mult r1, r5   r1 available
```

FIGURE 58. Superscalar Register Definitions

Figure 59 shows the register lifetime on a VLIW architecture two cycle operation. R1 is available in the two cycles following the add instruction, but would have a previous value, not the result of the add operation. It is the compiler's job to insert NOPs to insure that any pipeline hazards are removed.

```
1:  r1 = add r2, r3       -
2:  ...                   -
3:  ...                   -
4:  r4 = mult r1, r5      r1 live and available
```

FIGURE 59. VLIW Register Definitions

This difference in the definition of register liveness has performance implications. The first is that the VLIW architecture must insert NOPs between instructions to remove pipeline hazards. As mentioned in Section 3.2.2 on page 136, this can lower performance in certain situations because the schedule is less compact. Figure 60 shows a performance drop of about 20% for VLIW vs. superscalar using software pipelining, and a larger drop in performance for the other scheduling techniques.

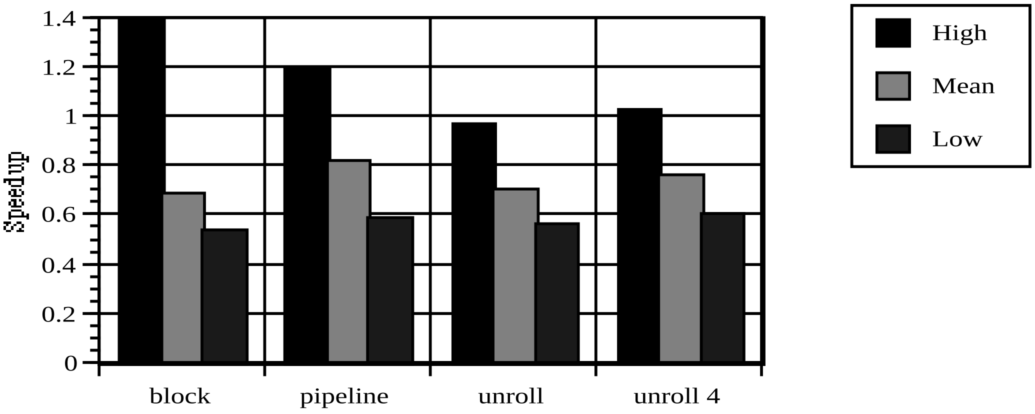


FIGURE 60. VLIW vs. Static Superscalar vs. Scheduling Technique

The register savings of VLIW are small, on the average, and are not large enough to make up for the additional padding required. Figure 61 shows the register use on each of the architectures.

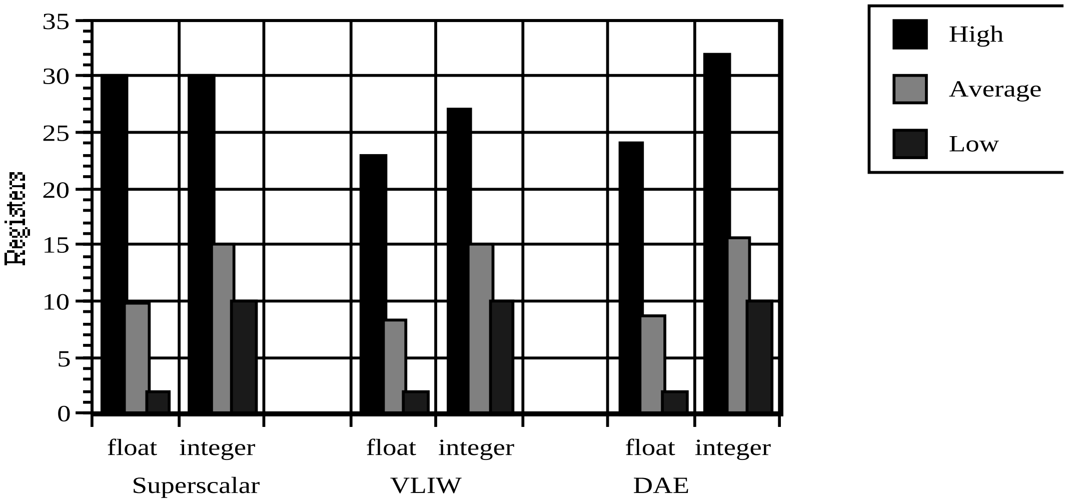


FIGURE 61. Register Use vs. Issue Policy with Software Pipelining

DAE is another interesting architecture on which to try our scheduling techniques. As mentioned in Section 3.2 on page 133, the Aurora III already has load and instruction queues and exhibits some of the characteristics of decoupled execution. However, in a DAE architecture, the communication with the memory unit is strictly via a set of queues. One register is removed from each of the integer and floating point register sets and the register ids are used to represent queues. The register ids are used in conjunction with load and store operators to select one of four data queues, i.e. integer data to memory, integer data to memory, floating point data to memory and floating point data from memory. There is also an address queue to send addresses to the memory system.

For each memory operation, either a load or store, the Access processor pushes an address onto the address queue. With each access processor memory address push, there is a corresponding reference in the Execute processor to a load or store data queue. A push to the store queue is indicated by using the store queue id as the target of an operation. A pop from the load queue is indicted by the use of the load queue id as a source operand.

The two DAE subprocessors usually implement different instruction sets, with the Access processor only being able to execute integer/address operations and the Execute processor implementing a more complete set of operations. The proper instructions must be routed to each subprocessor. This routing can be facilitated by defining a fixed instruction format where each half of the instruction is restricted to hold instructions for one fixed subprocessor. For simplicity, we will not split our processors; the operations will be allowed to execute from either side of the instruction window. This will make our results slightly optimistic over a more restricted format.

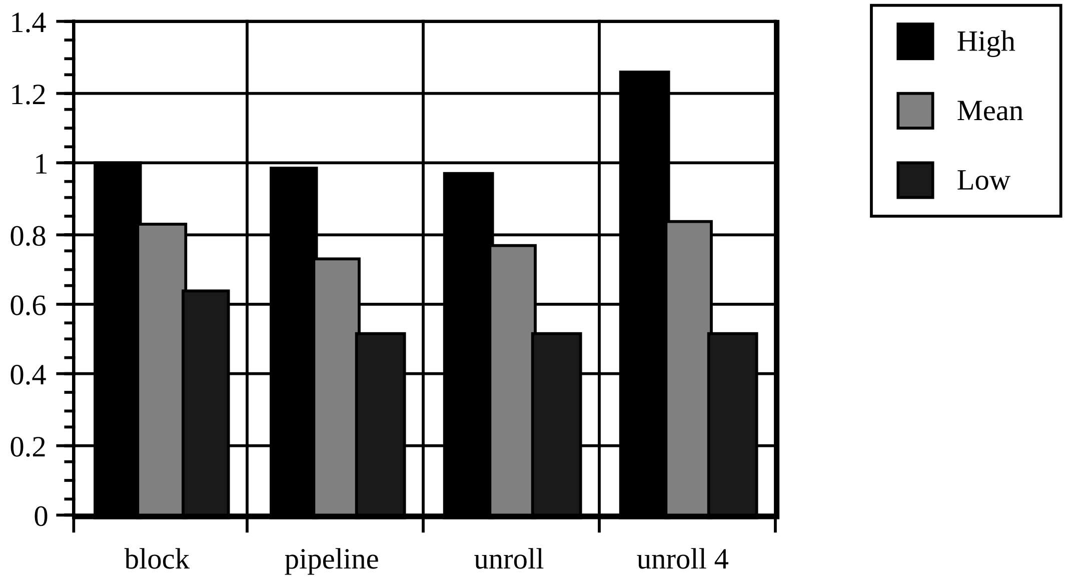


FIGURE 62. DAE vs. Static Superscalar vs. Scheduling Technique

However, we will adhere to the restrictions imposed by the DAE memory model: Each store data must go into the store queue and each load data from come from a load queue. Each reference to a load queue pops the queue, so the operand must be copied to another register if it is used as a CSE. Only one store queue reference and one load queue reference are allowed per instruction. If an instruction would use more than one load queue, one of the operands must be copied into a register.

The restrictions on queue use require additional move instructions to copy data to/from the queues and registers. This causes some performance degradation. Figure 62 shows performance of the DAE architecture as compared to the static superscalar for each of the scheduling techniques.

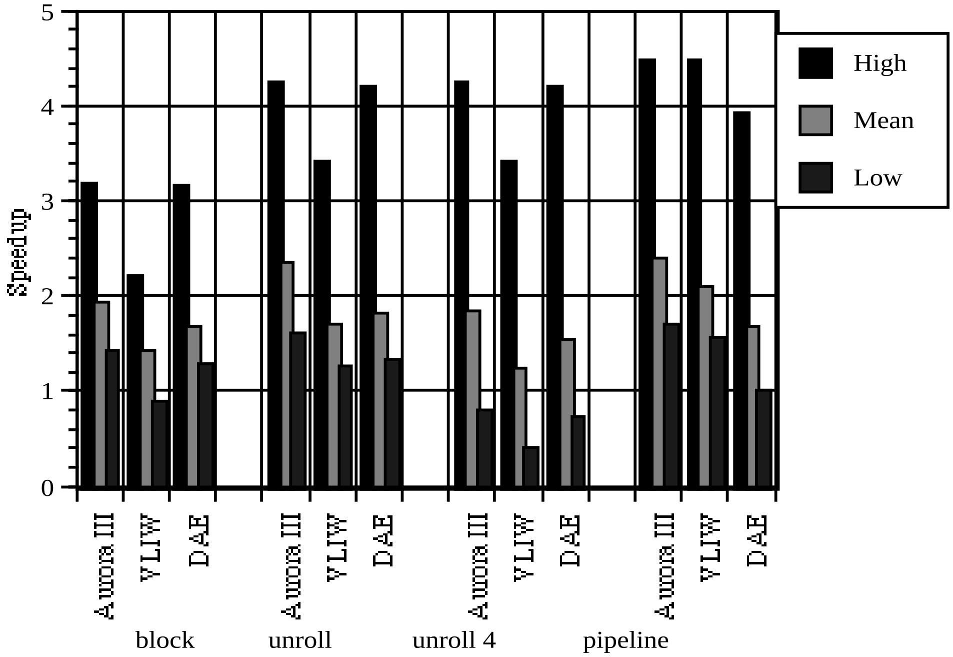


FIGURE 63. Aurora III, VLIW, and DAE vs. Scheduling Technique

Figure 63 shows the performances for all three multi-issue machines together with the scheduling technique employed, compared to the R3000. As expected, the superscalar architecture comes out on top. The relative performance of the VLIW and DAE architectures changes, depending on the scheduling technique used.

3.4 Aurora III Cache Behavior

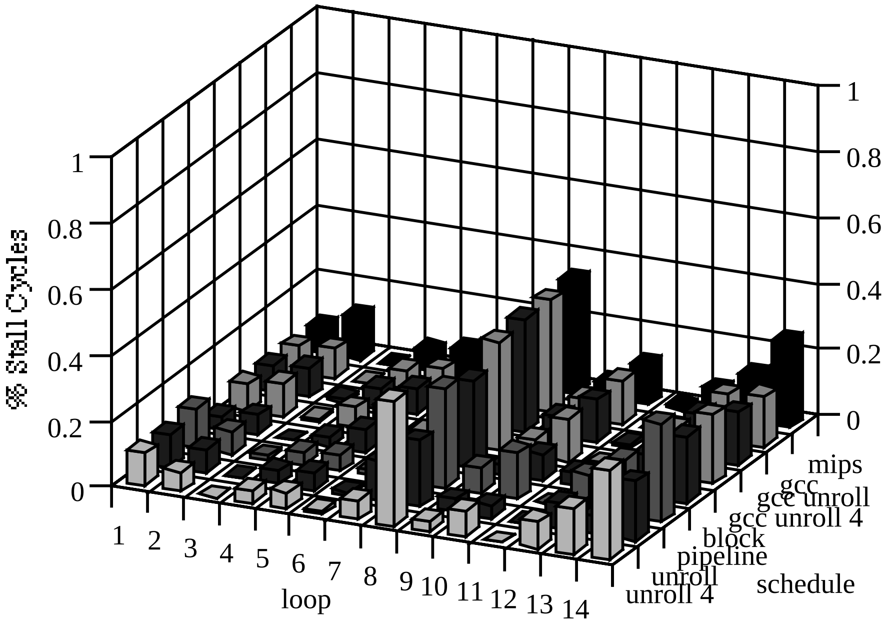


FIGURE 64. Percent Time Spent in D-Cache Stalls

We will consider cache behavior more thoroughly in the next section (see Section 4 on page 147). Here we will take a quick look at the cache behavior of the Aurora III with the current parameters. Figure 64 shows the percent of execution time spent in D-cache stalls. The D-cache performance looks fairly normal. However, the I-cache stalls, shown in Figure 65 show degenerate performance behavior for a few of the technique/benchmarks. This is because the 8k byte (2k instructions) I-cache is not large enough to hold the body of some of the larger schedules. Also, since the Aurora III does not have instruction prefetching and the memory system cannot stream instructions fast enough to keep up with the processor, the processor spends much of its time waiting on I-cache stalls.

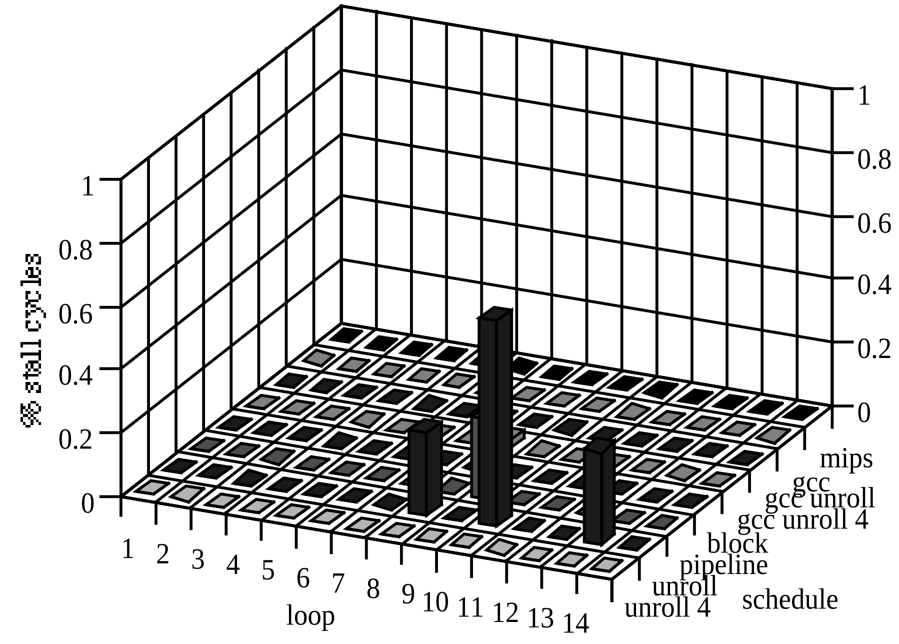


FIGURE 65. Percent Time Spent in I-Cache Stalls

4 Cache Effects

Both loop unrolling and software pipelining improve performance at the expense of increased code size. To this point we have not really investigated the effects of this increased code size, except to incorporate a realistic cache model into our simulations. We will now take a closer look at the effect these techniques have on code size and what this does with respect to cache behavior.

4.1 Previous Work

Previous studies [171][40][57][115] have found that differences in code density can affect performance. The size of the effect is found to decrease as the size of the cache is increased. Our experiment is similar to the experiments in earlier studies except for the uniformity condition. Uniformity assumes that code density changes equally for all parts of the program. Uniformity is intentionally avoided in our experiments. We wish to increase the code density in heavily used portions of the program, i.e. inner loops, and generate less compact code in less executed portions.

As the results here will show, our deviation from the uniformity condition is not enough to change nature of the effect. An increase in code size causes a decrease in performance and this effect disappears with larger caches.

First we will explore the effects of increased code sizes due to software pipelining in the context of the scalar R3000 architecture, with varying memory system parameters and software pipelining techniques. Then we will return to the Aurora III cache model and look at the cache behavior for our scheduling techniques there. Finally, we will look at the cache behavior on some larger programs, the Spec benchmarks, running on the R3000 again.

4.2 Cache Performance Effects from Software Pipelining

The increase in code size will depend on the implementation of software pipelining, as well as whether hardware support is available. If hardware support is available, the code size increase may be much less than if software pipelining is implemented entirely in the code. We examine the behavior of three types of hardware support for software pipelining:

1. Hardware support for both conditional instruction execution and register indexing (full support).
2. Hardware support for just register indexing (indexed support).
3. No hardware support (no support).

In these experiments, the R3000 is used as the base architecture. Two performance analysis tools, fpa_UM with CacheUM [131][125], are used to model the cache configurations and collect performance statistics.

In the first experiment, a code generator which employs software pipelining will be used to generate pipelined loops for the first fourteen Livermore loops. A high latency memory system with a two level cache organization is chosen for this experiment. The organization and latencies are taken from [131]. A first level cache miss has a latency of 5 cycles. A secondary miss has a latency of 141 cycles. These latencies are realistic for the current generation of processor and memory speeds.

In the architecture with full support, only the kernel stage of the loop is generated. This code is executed N+pipestages-1 times, where pipestages is the number of pipe stages in the loop. The extra pipestages-1 executions of the kernel are required to execute the epilogue portion of the loop.

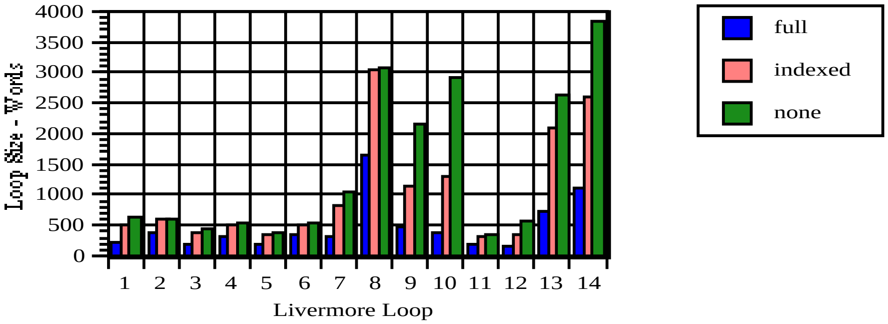


FIGURE 66. Code Sizes for the First Fourteen Livermore Loops

In the architecture with indexed register file support, separate code is produced to implement the various phases of pipelined loop execution: prologue, kernel, epilogue, etc. However, only one copy of each phase is produced and there is no unrolling to rename registers. This implementation produces an intermediate amount of code between architectures with full software pipeline support and no support.

In the architecture with no software pipeline support, additional unrolling is required to statically rename registers. This implementation requires the largest amount of code. As shown in Figure 66, the code size can increase dramatically on an architecture with no pipeline support. In this case, the code for loop 10 with no pipeline support is approximately ten times the size the code with full support. The other loops range from two to three times the size, between the largest and smallest implementations.

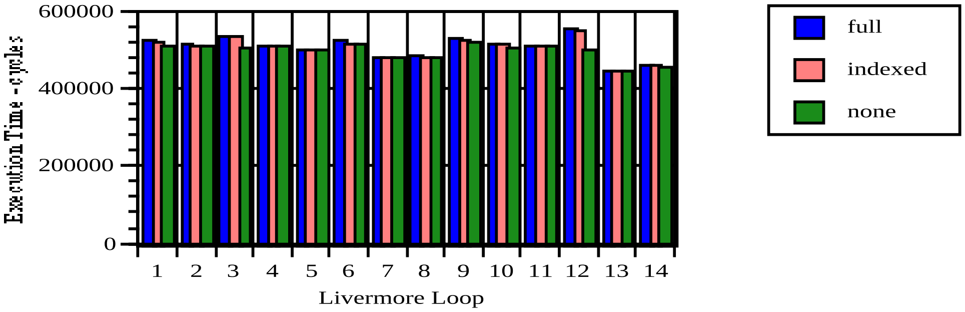


FIGURE 67. Execution Times for the First Fourteen Livermore Loops

If the effect of the increase in code size is not considered, there is little difference in execution time between the various implementations. The total cycles executed for each of the first 14 Livermore Loops is shown in Figure 67 with full, partial, no hardware support for software pipelining. As can be seen in the figure, there is almost no difference in the number of cycles executed between the implementation. Slightly fewer instructions are executed for implementations with indexed and no pipeline support. This is primarily due to fewer instructions being executed in the prologue and epilogue. Loop 12 shows the largest effect, of about 10%, mainly due the branches removed by unrolling a very small loop.

However, when cache effects are added, the performance picture changes. Figure 68 shows the total cycles executed for three versions of the first Livermore loop. The primary cache size is varied from 8 words to 1k words. As shown in Figure 68, a small cache can greatly degrade performance. Depending on the size of the cache and the code size, there can be a large performance penalty for the larger code size of the implementations without full hardware support. At a cache size of 16 words, Livermore loop 1 has almost a 40% performance penalty for implementing software pipelining via software. However, this effect diminishes very quickly and the effect is almost unnoticeable for caches of 32 words and larger.

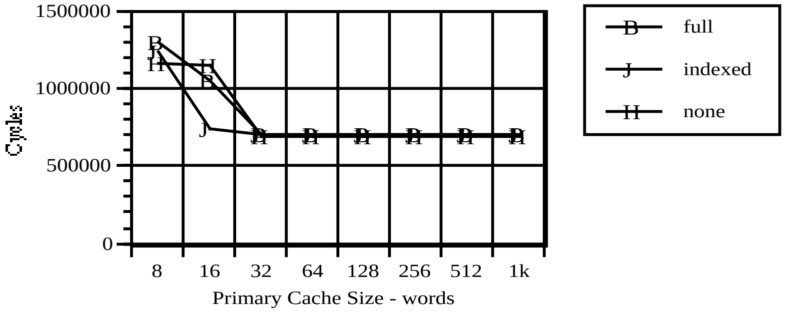


FIGURE 68. Execution Times for LL 1 vs. Primary Cache Size

4.3 Cache Behavior with Loop Unrolling

Because the Livermore Loops are so small, it is questionable how representative they are of “real” programs. It would be interesting to run the same set of experiments, generating code for various types of hardware support, on larger programs. Unfortunately, the current version of the compiler is unable to perform this task. However, we can study this problem by using another technique.

The structure of the code produced to implement software pipelining is similar to the code produced to implement standard loop unrolling. The effects of standard loop unrolling with respect to program size and I-cache effects should be very similar. To test this hypothesis, Version 2.2.2 of the Gnu-C compiler has been modified to accept a parameter controlling the amount of loop unrolling and the first fourteen Livermore Loops have been compiled and with various amounts of unrolling. Figure 69 shows the total number of cycles executed by the first Livermore Loop when the size of the loop body is allowed to increase to 10, 50 and 100 instructions. The execution behavior under these conditions looks very similar to the behavior shown in Figure 67, where the size of the loop varied in response to changes in the hardware model. This similarity seems to indicate that a significant portion of the performance gain from hardware support for software pipelining is due to reduced code size.

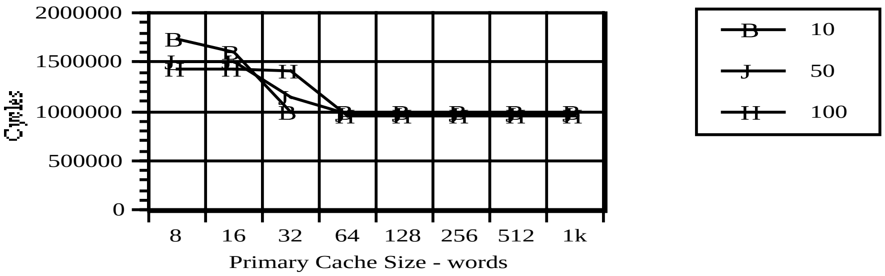


FIGURE 69. Execution Times for the First Livermore Loop using Gnu-C

Given these encouraging results, we will go one step further, and apply the modified Gnu-C compiler to larger and more realistic programs than the Livermore Loops. The SPEC suite is a collection of “real” programs currently in use as benchmarks. Four of the programs in the SPEC suite are provided in C and are thus available for this experiment (eqntott, espresso, gcc, and xlisp).

We compile eqntott, espresso, and xlisp with an increasing amount of unrolling. Figure 70 shows the code sizes produced when compiling xlisp. The results for the other benchmarks, which are not shown, are similar. The range of values executed for the unrolling parameter varies the size of the code produced by approximately 25% from the least to the most unrolling.

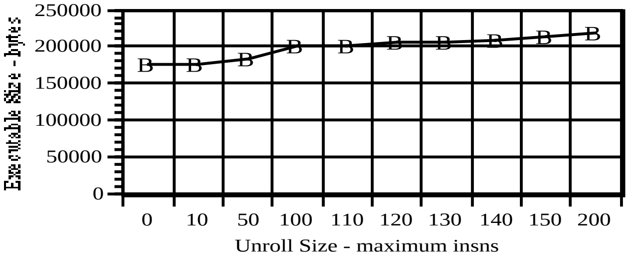


FIGURE 70. Code Size for xlisp

The move to a larger, more realistic program has two effects on the performance curves: 1) The of performance for different amounts of unrolling is not as pronounced, and 2) An effect on performance is present up to a 32k word primary I-cache.

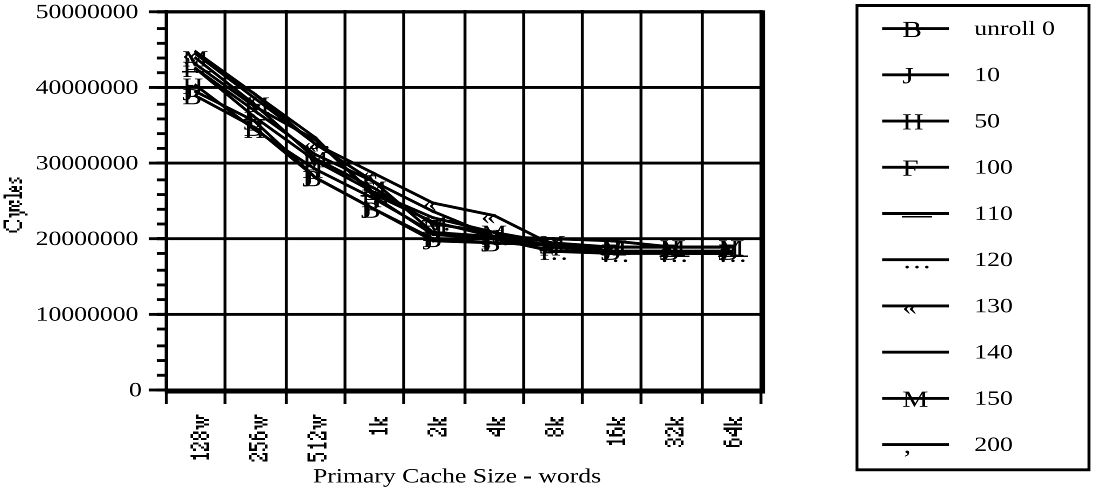


FIGURE 71. xlisp: Cycles vs. Primary Cache Size (Long Latency Mem.)

Performance statistics are shown in Figure 71 and Figure 72 for xlisp with executed with low and high latency memory configurations. The low latency memory has a 3 cycle penalty for a primary cache miss and a 20 cycle penalty for a secondary cache miss. The high latency memory has 5 cycle penalty for a primary cache miss and a 141 cycle penalty for a secondary cache miss.

Figure 71 and Figure 72 show the total cycles executed as a function of the size of the primary cache. In the range of a realistic primary cache on today's implementations, i.e. 1k-64k words, the effect of loop unrolling is small but noticeable. The range in performance is approximately 5% on the high end of this region for the high latency memory system. The performance range is smaller for the low latency memory system, but still noticeable.

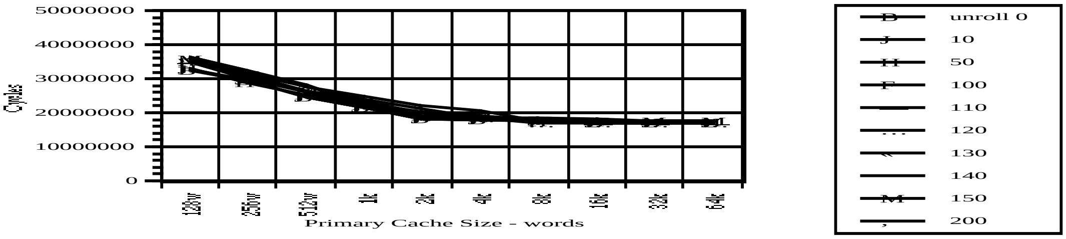


FIGURE 72. xlisp: Cycles vs. Primary Cache Size (Short Latency Mem.)

Figure 73 and Figure 74 show the number of cycles executed as a function of the amount of unrolling. A slight rising trend can be seen as the amount of unrolling is increased. This rising trend almost, but not quite disappears for the larger primary cache sizes. The trend does not disappear entirely because there are still some secondary cache misses which result in an approximately 1% performance loss over the set of programs shown here.

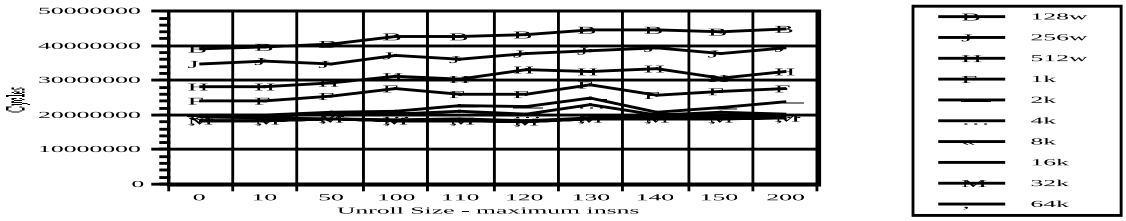


FIGURE 73. Cycles Executed for xlisp vs. Unroll Size (Long Latency Mem.)

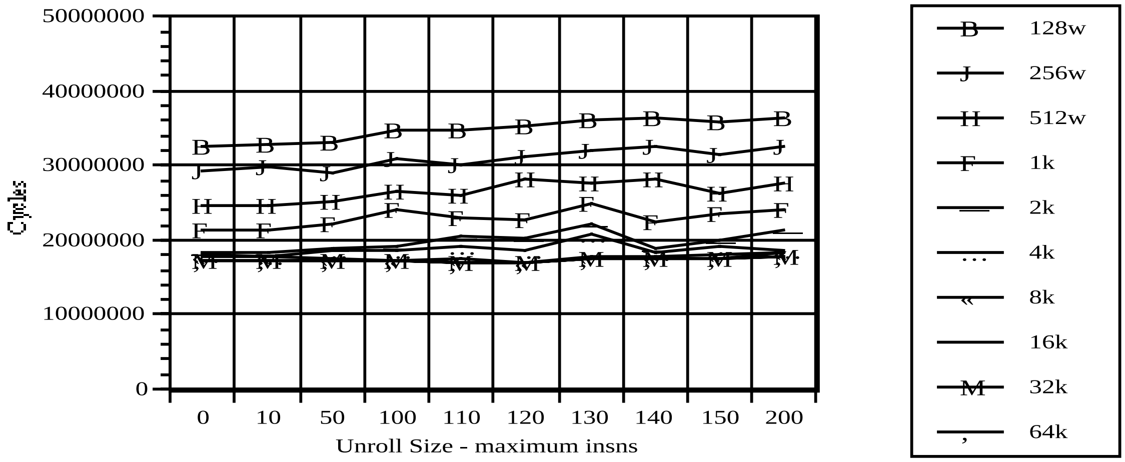


FIGURE 74. Cycles Executed for xlisp vs. Unroll Size (Short Latency Mem.)

4.4 Context Switch Effects

In [116] Mogul and Borg find a performance degradation due to context switching of 1% to 7% depending on the program mix and cache design. This study, and another by Steenkiste [171], show that the additional effect of having a larger code size when context switching might be 10% of cost of context switching, or 0.1% to 0.7% overall. This effect is small enough that it would be difficult to discern on most systems.

4.5 Summary of Cache Effects

The results shown here do support a small, but noticeable decrease in program performance when unrolling loops using standard loop unrolling techniques or software pipelining. While generally the effect is not large, it is large enough to be noticeable and under certain conditions can become quite large, e.g. as shown on the Aurora III in Section 3.4 on page 146. Because of the sharp decrease in performance which can result when the worst case arise, cache effects must be considered when implementing scheduling algorithms which can increase code size.

CHAPTER VI
CONCLUSIONS

1 Research Contributions

The data produced in this study shows a relatively small difference in performance between the architectures investigated. In Figure 63, the mean speedup for each of the three architectures investigated and four scheduling techniques implemented are shown together with the best and worst speedups for each of the benchmarks. The speedups shown are with respect to a scalar processor. Superscalar architectures are represented by the Aurora III. The VLIW and DAE architectures are versions of the Aurora III, modified to match the salient characteristics of these architectures.

Taking the best mean result for each the architectures, the VLIW architecture has performance within 15% of the superscalar architecture and the DAE architecture is within 10% of the VLIW. These relatively minor performance differences are reasonable considering the differences between these architectures. The fundamental difference between VLIW architecture and superscalar architecture is that a more rigid static schedule is used in the VLIW architecture. There is no dependence checking between instructions with an instruction window in VLIW and the use of a result must be scheduled after the result exits the function unit pipeline.

The DAE architecture has an issue policy, intermediate between that of superscalar and VLIW. Each sub-processor in a DAE architecture, the Address and Execute units, can issue independently of the other, but in-order issue is enforced within each sub-processor. This allows some dynamic behavior between the Address and Execute units. However, this additional flexibility is offset by the necessity of scheduling communication between the two sub-processors.[1] This communication scheduling may consume additional instructions which can reduce the total performance. The dynamic scheduling allowed by the DAE paradigm is often not enough to recover this performance loss. Another potential performance loss is due to the limitation that the DAE architecture cannot issue to multiple instructions to a single sub-processor, i.e. each sub-processor is restricted to single issue. There is no performance penalty when the schedule is balanced between access and execute type operations. However, if the schedule is unbalanced with one sub-processor having more operations to performance than the other, one sub-processor must stall to maintain synchronization between the two schedules.

The DAE architecture does fare better than the VLIW architecture under block scheduling or loop unrolling, probably because the decoupled and dynamic behavior of DAE is more of an advantage under these scheduling algorithms. Under software pipelining, the performance ranking of DAE versus VLIW is reversed.

Of course, caveat emptor applies to the generality of this study as it does to any other study replying on a set of benchmarks which are not actual applications. The set of benchmarks chosen, the Livermore Loops, is reported to provide good correlation with the performance of actual application loads consisting of scientific programs. However, the Livermore Loops tend to have largely static behavior. One of the criticisms of the original set of 14 Livermore Loops was that they overestimated the performance of general scientific applications because they contained too high a proportion of easily vectorizable loops. In our study, this may tend to bias the results in favor of VLIW

[1] If the production/consumption behavior of the code being scheduled matches the DAE queue model, no additional instructions are required. However, when there is a mismatch between the code production/consumption behavior and the DAE queue model, additional instructions are required to implement the code semantics.

architectures and software pipelining techniques, both of which reply on the compiler to provide good schedules and performance.

A corollary to the relatively small difference in performance between the architectures is that other features of the processor are more indicative of performance than the issue policy. The Aurora III incorporates a number of features designed to improve performance over the baseline MIPS R3000 processor. Figure 49 shows the high, mean and low speedups for the Aurora III with double precision floating point load and store operations, and without these operations. The mean speedup for these benchmarks for this single feature ranges from 1.2 to 1.5. This difference is larger than the performance for any of the architecture pairs.

While we have no direct data to support a claim that it is less costly to implement 64 bit wide data paths, than to implement a multi-issue superscalar or VLIW architecture, we can speculate that it should be, just because the problem is conceptually simpler. Implementing double precision load and store operations would still require changes to the compiler to generate the correct instructions and the processor model would need to be modified and verified, but these changes seem minor compared with changing the issue policy of a processor.

Of the other processor features investigated, the performance gain for pipelining the function units seems anomalous. Pipelining the function units did not have a large impact on performance, even for medium latency operations. Apparently, for latencies in the range we examine, the mix of instructions is such that delays can be filled with operations on different function units and the ability to pipeline a single operation is not critical. Of course, particular cases can benefit from pipelining one or more function units and the expense of adding pipelining to a function unit may be justified for just such special cases.

The second conclusion to draw form this work is the importance of incorporating compiler scheduling techniques when studying the performance of computer architectures. In the performance data shown in Figure 63, we find substantial performance improvements for loop unrolling and software pipelining over block scheduling, which is to be expected. However, a more interesting result is that the relative performance ranking of VLIW versus DAE changes with software pipelining versus the other loop scheduling techniques. This relative performance difference is too large to be dismissed easily. If the difference was 1% or 2% this could be considered to be a random accident of instruction scheduling. However, a relative performance difference of 10% or more is indicative of a real interaction between these architectures and the scheduling algorithm.

Software pipelining was developed with VLIW architectures and static scheduling in mind, so it is reasonable to expect a performance improvement on this architecture with this scheduling technique. We also get a performance improvement, although a very small one, on the superscalar architecture. This seems to be due to a slightly decreased register consumption by the software pipelining algorithm. This performance difference is small enough to be in the noise level of scheduling, so perhaps we should not read too much into this.

There are a number of techniques and algorithms employed in the compiler, other than the loop scheduling algorithms. Code optimization techniques such as constant propagation, induction variable detection and strength reduction, dead code elimination, register promotion and others all contribute to improved performance. Generally, no single technique used alone will give much performance improvement. However, the set of techniques collectively called “loop optimizations” will yield a substantial performance improvement for the type of codes used in this study. These techniques are standard in optimizing compilers and give performance improvements in the range of 1 to 2x. This outweighs the possible performance gain from switching loop scheduling algorithms, which explains why these techniques are “standard” and software pipelining is not.

One good result of this study is that we have a fairly “low noise” environment in which to compare hardware features and compiler algorithms. By incorporating our scheduling techniques and machines models into a single base optimizing compiler, we can compare the results without having to allow for differences caused by using different compilers and methodologies.

In looking at the standard loop optimization techniques, we have developed some new induction variable manipulations. These manipulations allow induction expressions formed by nested loops to be treated and manipulated as a single expression. This simplifies the work in the compiler to manipulate nested induction variables and in some cases also yields performance improvements. However, while this type of manipulation is intellectually interesting, it only rarely yields a significant performance improvement. Also, the pattern detection necessary to identify induction expressions which can be manipulated tends to be brittle and it is easy to write control flow structures which produce induction expressions which are not recognized as manipulatable.

While software pipelining is not a standard technique in current compilers, it is demonstrated in this study to work well, even on an architecture not particularly suited to its particular strengths. The experiments run on the scalar architecture, show software pipelining slightly outperforming other scheduling techniques, even with function units with no pipelining and relatively low latency. This advantage is magnified when the latency of the operations being scheduling is increased or static instruction scheduling is used, as in the VLIW architecture.

The primary reason for software pipelining’s performance improvement is its ability to merge loop iterations and thus hide instruction latencies. However, the software pipelining algorithm used in this study also seems to be more frugal in its resource use than other scheduling techniques. In particular, software pipelining tends to consume fewer registers than loop unrolling. This is an unexpected result. Software pipelining is supposed to be good at intermixing instructions to hide operation latencies. The assumption is that this would be at the expense of consuming additional resources, both instruction space and registers. The improved resource use may be because the software pipelining algorithm used in this work is self limiting. The schedules expand not by a fixed amount, but enough to fill the unused delay slots. This feedback seems to yield a balance between resource use and parallelism exploitation.

This does not negate the necessity of exploring other techniques. Scheduling techniques such as loop unrolling and trace scheduling each have their own particular benefits and a good compiler will have a repertoire of such techniques available. And, while software pipelining works well on loops, it is not extendable to other control structures one might to optimize, such as long runs of branching code.

In this study we briefly explore the interaction with the compiler scheduling techniques and the instruction cache. Degenerative cache behavior is possible when using optimization techniques which affect code size and placement. At current cache sizes, i.e. small to moderate size caches, the increasing disparity between memory and processor speeds can magnify this effect to where it can overwhelm any other performance improvements.

The effect of the scheduling techniques on cache behavior was generally found to be small. However, by employing a technique in a careless way, it is possible to cause performance to decrease. This can happen, for example, by unrolling a loop with a large body to the point where it overflows the instruction cache. This can give much lower performance than no unrolling at all. These optimization techniques do need to be carefully employed to yield good results.

2 Future Directions

There are a number of directions in which this research can continue. One metric which needs to be explored is the cost function for different architectures/machine models. The question: “Given a choice between architecture A and architecture B, what is the cost of these two architectures in terms of cycle time, die area or complexity,” is not easily answered. It is the assumption of this research that if a superscalar architecture and a VLIW architecture have the same performance in term of cycles, then it would be better to build the VLIW architecture because the reduced complexity of the architecture will lead to a smaller die area and a faster cycle time. It would be interesting to test this hypothesis by designing a set of architectures using the same process technology and then measuring the complexity of the design and determining the least cycle time.

There continues to be a need for accurate comparisons between compiler algorithms. There are several competing versions of software pipelining which should be examined and compared. The software pipelining algorithm used in this dissertation has several deficiencies. The algorithm used here is expensive because it iteratively rebuilds the entire schedule until all the scheduling constraints are satisfied. The unrolling type algorithms avoid rebuilding the entire schedule, although they may be expensive in other ways.

The algorithm used here is trying to simultaneously satisfy conflicting sets of constraints. Both resource and timing constraints are checked during scheduling. Trying to simultaneously satisfy conflicting constraints is always difficult and should be avoided. It would be much better if scheduling for timing and resources were separated. Also, timing constraints are not well modeled by the rigid schedule used here. The first pass of the software pipelining algorithm should just reorder the loop, trying to spread operations and fill delays. This is the software pipelining part. After that, a more traditional scheduler could place the instructions in a schedule according to available resources. This is the approach taken by Jain in [77] and it has the major advantage that it keeps most of parts of a standard compiler intact. It would be interesting to see how these different versions of software pipelining would compare when used in a common system.

Another area that this research touches on briefly is providing explicit cache control. There are two ways in which this relates to the work done here. First, any kind of cache control will inherently be a long latency operation. This means that techniques to handle operations with long latencies, such as software pipelining, will be needed to effectively schedule cache control.

Multi-issue architectures will both produce the necessity for, and provide the opportunity for explicit cache control. Multi-issue architectures produce the necessity for cache control because they put higher demands on the memory system. Everything else being equal, multi-issue multiplies the bandwidth at which instructions must be delivered to the processor. Also, the effect of any delay, including cache stalls, is magnified in a multi-issue architecture because there are that many more instructions that are not being executed during the delay.

At the same time, multi-issue architectures will provide an opportunity for explicitly controlling the cache because there will be more instruction slots which cannot be filled with other types of operations. Making a wider instruction window or adding more branch delay slots means that a lower fraction of these slots can be filled with useful operations. These unusable slots could be used to execute some cache control operations. This would avoid slowing the execution rate, because these would be slots which could not be otherwise filled and the explicit cache control instructions would potentially speed execution by reducing cache stalls.

Part of the idea of building this compiler system is that it would form a test bed in which optimization algorithms could be quickly implemented and tested. Unfortunately this did not turn out to be the case. Coding the compiler algorithms proved to be the most time consuming and difficult part of this research. Implementing the compiler algorithms proved to be much harder than designing and debugging the hardware features and analysis tools. We need to construct tools which allow us to describe and test compiler algorithms, particularly in the intermediate and late compilation phases, where the program objects being manipulated tend to be represented as graphs. We lack good tools for describing graph manipulations and transformations.

APPENDIX

The maximum number of blocks which might be required by a software pipeline schedule on an architecture is an important parameter of the architecture. The maximum number of blocks affects the width of the block count field (or conditional bits) in an architecture with hardware supported software pipelining, and indirectly, the number of registers used during scheduling.

The maximum number of blocks required to construct a software pipeline schedule is a function of the amount of parallelism available in the architecture. We show here that the maximum number of blocks for multi-issue, pipelined architectures is a function of the number of pipe stages in the architecture.

The maximum number of blocks for an architecture with a single function unit:

$\text{mii} =$ The Minimum Initiation Interval - minimum time interval before next concurrent loop iteration can be started.

$|\ \text{stages}\ | =$ The number of pipe stages to do an operation.

$|\ \text{opers}\ | =$ The number of operations to be executed on the function unit.

$\text{length} =$ The length of the loop body in machine cycles.

$|\ \text{blocks}\ | =$ The length of the loop body in units of MII machine cycles.

$$\text{mii} = |\ \text{opers}\ |$$

$$|\ \text{blocks}\ | = \text{length} / \text{MII}$$

$$\text{length} \leq |\ \text{opers}\ | * |\ \text{stages}\ |$$

$$\begin{aligned} |\ \text{blocks}\ | &\leq [|\ \text{opers}\ | * |\ \text{stages}\ |] / \text{mii} \\ &\leq [|\ \text{opers}\ | * |\ \text{stages}\ |] / |\ \text{opers}\ | \\ &\leq |\ \text{stages}\ | \end{aligned}$$

The maximum number of blocks for an architecture with multiple function units:

Assume a VLIW type architecture with a set of N function units: fu_1, fu_2,..., fu_n, where one operation can be issued to each function unit per cycle.

$| fu |$ = The number of function units.

$| opers_i |$ = The number of operations in the schedule for function unit i.

$| stages_i |$ = The number of pipe stages in function unit i.

length = The length of the loop body in machine cycles.

$| blocks |$ = The length of the loop body in units of MII machine cycles.

$| opers |$ = The total number of operations to be executed in the loop body.

$$N = | fu |$$

$$| stages | = \max | stages_i |, \forall i \in N$$

$$mii = \max | opers_i |, \forall i \in N$$

$$mii \quad 1/N$$

$$\begin{aligned} | blocks | &\leq [| opers | * | stages |] / mii \\ &\leq [| opers | * | stages |] / \max | opers_i | \\ &\leq [| opers | * | stages |] / 1/N \, | opers | \\ &\leq | fu | * | stages | \end{aligned}$$

Conclusion:

The maximum number of blocks which can concurrently execute on this architecture is | fu | * | stages |, where | fu | = number of function units and | stages | = the maximum number of pipe stages required to perform an operation on any of the function units.

BIBLIOGRAPHY

[1] R. D. Acosta, J. Kjelstrup, H. C. Torng, An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors, *IEEE Transactions on Computers* C-35(9), 1986, pp. 815-828.

[2] A. V. Aho, J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.

[3] A. V. Aho, S. C. Johnson, J. D. Ullman, Code Generation for Expressions with Common Subexpressions, *JACM* 24(1), 1977, pp. 146-160.

[4] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[5] A. Aiken, A. Nicolau, A Development Environment for Horizontal Microcode, *IEEE Transactions on Software Engineering* 14(5), 1988, pp. 584-594.

[6] A. Aiken, A. Nicolau, Optimal Loop Parallelization, *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988, pp. 308-317.

[7] V. H. Allan, B. Su, P. Wijaya, J. Wang, Foresighted Instruction Scheduling Under Timing Constraints, *IEEE Transactions on Computers* 41(9), 1992, pp. 1169-1172.

[8] R. Allen, K. Kennedy, Automatic Translation of FORTRAN Programs to Vector Form, *ACM Transactions on Programming Languages and Systems* 9(4), 1987, pp. 491-542.

[9] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, IBM System/360 Model 91: Machine Philosophy and Instruction Handling, *IBM Journal of Research and Development* , 1967, pp. 8-24.

[10] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, D. M. Powers, IBM System/360 Model 91: Floating-Point Execution Unit, *IBM Journal of Research and Development* , 1967, pp. 34-53.

[11] M. Annaratone, et al., The Warp Computer: Architecture, Implementation and Performance, *IEEE Transactions on Computers* C-36, 1987, pp. 1523-1537.

[12] T. M. Austin, G. S. Sohi, Dynamic Dependency Analysis of Ordinary Programs, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, vol. 20, pp. 342-351.

[13] M. E. Benitez, J. W. Davidson, Code Generation for Streaming: an Access/Exe-

cute Mechanism, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 132-141.

[14] D. Bernstein, I. Gartner, Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle, *JACM* 11(1), 1989, pp. 57-66.

[15] D. Bernstein, et al., Spill code minimization techniques for optimizing compilers, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989, pp. 258-263.

[16] F. Boeri, M. Auguin, OPSILA: A Vector and Parallel Processor, *IEEE Transactions on Computers* 42(1), 1993, pp. 76-82.

[17] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, J. W. Smith, IBM System/360 Model 91: Storage System, *IBM Journal of Research and Development* , 1967, pp. 54-68.

[18] A. Borg, R. E. Kessler, D. W. Wall, Generation and analysis of very long address traces, *Proceedings of the 17th Annual International Symposium on Computer Architecture,* 1990, pp. 270-279.

[19] D. G. Bradlee, S. J. Eggers, R. R. Henry, The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 330-339.

[20] D. G. Bradlee, S. J. Eggers, R. R. Henry, Integrating Register Allocation and Instruction Scheduling for RISCs, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 122-131.

[21] P. Briggs, K. D. Cooper, K. Kennedy, L. Torczon, Coloring Heuristics for Register Allocation, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989, pp. 275-284.

[22] M. Butler, et al., Single Instructions Stream Parallelism Is Greater than Two, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 276-286.

[23] D. Callahan, K. Kennedy, A. Porterfield, Software Prefetching, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 40-52.

[24] G. J. Chaitin, Register Allocation and Spilling Via Graph Coloring, *ACM SIGPLAN Notice* 17(6), 1982, pp. .

[25] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, W.-M. W. Hwu, IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 266-275.

[26] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family,"*IEEE Computer,* September 1981, pp. 18-27.

[27] W. Y. Chen, S. A. Mahlke, W.-m. W. Hwu, Tolerating First Level Memory Access Latency in High-Performance Systems, *Proceedings of the 1992 International Conference on Parallel Processing,* 1992, vol. II, pp. I-37-II-43.

[28] T.-F. Chen, J.-L. Baer, Reducing Memory Latency via Non-blocking and Prefetching Caches, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, vol. 20, pp. 51-61.

[29] C.-H. Chi, H. Dietz, Unified Management of Registers and Cache Using Liveness and Cache Bypass, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989, vol. 21, pp. 344-355.

[30] T.-c. Chiueh, Multi-Threaded Vectorization, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 352-361.

[31] P. Chow, M. Horowitz, Architectural Tradeoffs in the Design of MIPS-X, *Proceedings of the 14th Annual International Symposium on Computer Architecture,* 1987, pp. 300-308.

[32] F. Chow, S. Correll, M. Himelstein, E. Killian, L. Weber, How Many Addressing Modes are Enough?, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* 1987, pp. 117-121.

[33] F. C. Chow, J. L. Hennessy, The Priority-Based Coloring Approach to Register Allocation, *ACM Transactions on Programming Languages and Systems* 12(4), 1990, pp. 501-536.

[34] E. U. Cohler, J. E. Storer, "Functionally Parallel Architecture for Array Processors,"*IEEE Computer,* September 1981, pp. 28-36.

[35] R. Cohn, T. Gross, M. Lam, P. S. Tseng, Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* 1989, vol. 17, pp. 2-14.

[36] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, P. K. Rodman, A VLIW Architecture for a Trace Scheduling Compiler, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* 1987, pp. 180-192.

[37] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, P. K. Rodman, A VLIW Architecture for a Trace Scheduling Compiler, *IEEE Transactions on Computers* 37(8), 1988, pp. 967-979.

[38] R. P. Cook, M. Donde, An Experiment to Improve Operand Addressing, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 87-91.

[39] G. Cybenko, L. Kipp, L. Pointer, D. Kuck, *Supercomputer Performance Evaluation and the Perfect Benchmarks,* University of Illinois, CSRD Report No. 965, March 1990.

[40] J. Davidson, R. Vaughan, The Effect of Instruction Set Complexity on Program Size and Memory Performance., *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 60-64.

[41] J. W. Davidson, J. R. Rabung, D. B. Whalley, Relating Static and Dynamic Machine Code Measurements, *IEEE Transactions on Computers* 41(4), 1992, pp. 444-454.

[42] J. C. Dehnert, P. Y.-T. Hsu, J. P. Bratt, Overlapped Loop Support in the Cydra 5, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* 1989, vol. 17, pp. 26-38.

[43] D. R. Ditzel, H. R. McLellan, Register allocation for free: The C machine stack cache, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 48-56.

[44] R. J. Eickemeyer, J. H. Patel, Performance Evaluation of Multiple Register Sets, *Proceedings of the 14th Annual International Symposium on Computer Architecture,* 1987, pp. 264-271.

[45] C. Eisenbeis, W. Jalby, A. Lichnewsky, Squeezing More Cpu Performance Out of a Cray-2 by Vector Block Scheduling, *Proceedings of Supercomputing '88,* 1988, pp. 237-246.

[46] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* ACM Doctoral Dissertation Award, The MIT Press, 1985.

[47] M. K. Farrens, A. R. Pleszkun, Implementation of the PIPE processor, *Proceed-*

ings of the 16th Annual International Symposium on Computer Architecture, 1989, vol. 17, pp. 65-70.

[48] M. K. Farrens, *The Design and Analysis of a High-Performance Single-Chip Processor,* PhD Thesis, University of Wisconsin at Madison, 1989.

[49] M. K. Farrens, A. R. Pleszkun, "Implementation of the PIPE processor,"*IEEE Computer,* January 1991, pp. 65-70.

[50] E. S. T. Fernandes, F. M. B. Barbosa, Effects of Building Blocks on the Performance of Super-Scalar Architectures, *Proceedings of the 19th Annual International Symposium on Computer Architecture,* 1992, vol. 20, pp. 36-45.

[51] J. Ferrante, K. J. Ottenstein, J. D. Warren, The Program Dependence Graph and Its Use in Optimization, *ACM Transactions on Programming Languages and Systems* 9(3), 1987, pp. 319-349.

[52] J. A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Transactions on Computers* C-30(7), 1981, pp. 478-490.

[53] J. A. Fisher, The VLIW Machine: A Multiprocessor for Compiling Scientific Code, *IEEE Computer* 17(7), 1984, pp. 45-53.

[54] R. W. Floyd, Algorithm 97: Shortest Path, *Communications of the ACM* 5(6), 1962, pp. 345.

[55] M. J. Flynn, P. R. Low, IBM System/360 Model 91: Some Remarks on System Development, *IBM Journal of Research and Development* , 1967, pp. 2-7.

[56] M. J. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers* C-21(9), 1972, pp. 948-960.

[57] M. J. Flynn, C. L. Mitchell, J. M. Mulder, "And now a Case for More Complex Instruction Sets,"*IEEE Computer,* September 1987, pp. 71-83.

[58] C. C. Foster, E. M. Riseman, Percolation of code to enhance parallel dispatching and execution, *IEEE Transactions on Computers* C-21(12), 1972, pp. 1411-1415.

[59] M. Franklin, G. S. Sohi, The Expandable Split Window Paradigm for Exploiting Find-Grain Parallelism, *Proceedings of the 19th Annual International Symposium on Computer Architecture,* 1992, vol. 20, pp. 58-67.

[60] P. B. Gibbons, S. S. Muchnick, Efficient Instruction Scheduling for a pipelined architecture, *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* 1986, pp. 11-16.

[61] J. R. Goodman, W.-C. Hsu, On the Use of Registers vs. Cache to Minimize

Memory Traffic, *Proceedings of the 13th Annual International Symposium on Computer Architecture,* 1986, pp. 375-383.

[62] G. L. Graig, et al., *PIPE: A High Performance VLSI Processor Implementation,* Computer Sciences Department, Electrical and Computer Engineering Department, University of Wisconsin-Madison, Technical Report 1984.

[63] T. Gross, M. S. Lam, Compilation for a High-Performance Systolic Array, *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* 1986, pp. 27-38.

[64] R. Gupta, M. L. Soffa, T. Steele, Register Allocation Via Clique Separators, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989, pp. 264-274.

[65] M. S. Hecht, *Flow Analysis of Computer Programs*, T. E. Cheatham (Ed.), Programming Languages Series, Elsevier North-Holland, New York, New York, 1977.

[66] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill, Hardware/Software Tradeoffs for Increased Performance, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 2-11.

[67] J. L. Hennessy, D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.

[68] J. L. Hennessy, N. P. Jouppi, "Computer Technology and Architecture: An Evolving Interaction,"*IEEE Computer,* September 1991, pp. 18-29.

[69] M. D. Hill, A Case for Direct-Mapped Caches, *IEEE Computer* 21(12), 1988, pp. 25-40.

[70] W.-m. Hwu, Y. N. Patt, HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality, *Proceedings of the 13th Annual International Symposium on Computer Architecture,* 1986, pp. 297-306.

[71] W.-m. W. Hwu, P. P. Chang, Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator, *Proceedings of the 15th Annual International Symposium on Computer Architecture,* 1988, pp. 45-53.

[72] W.-M. W. Hwu, T. M. Conte, P. P. Chang, Comparing Software and Hardware Schemes For Reducing the Cost of Branches, *Proceedings of the 16th Annual International Symposium on Computer Architecture,* 1989, vol. 17, pp. 224-233.

[73] W.-m. W. Hwu, P. P. Chang, Achieving High Instruction Cache Performance with an Optimizing Compiler, *Proceedings of the 16th Annual International*

Symposium on Computer Architecture, 1989, vol. 17, .

[74] W.-m. W. Hwu, P. P. Chang, Inline Function Expansion for Compiling Realistic C Programs, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989, pp. 246-257.

[75] W.-m. W. Hwu, P. P. Chang, Efficient Instruction Sequencing with Inline Target Insertion, 41(12), 1992, pp. 1537-1551.

[76] IBM, *IBM RISC System/6000 Technology,* IBM Corporation, Technical Report SA23-2619, 1990.

[77] S. Jain, Circular scheduling: A new technique to perform software pipelining., R. L. Wexelblats (Ed.), *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991, vol. 26, pp. 219-228.

[78] M. S. Johnson, T. C. Miller, Effectiveness of a Machine-Level, Global Optimizer, *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986, pp. 99-108.

[79] W. M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[80] N. P. Jouppi, D. W. Wall, Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, vol. 17, pp. 272-282.

[81] N. P. Jouppi, The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance, *IEEE Transactions on Computers* 38(12), 1989, pp. 1645-1658.

[82] N. P. Jouppi, Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch BUffers, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, vol. 18, pp. 364-373.

[83] G. Kane, *MIPS RISC Architecture,* Prentice-Hall, Inc., Englewood Cliffs, 1988.

[84] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, ACM Doctoral Dissertation Award, The MIT Press, 1984.

[85] A. C. Klaiber, H. M. Levy, An Architecture for Software-Controlled Data Prefetching, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, vol. 19, pp. 43-53.

[86] D. E. Knuth, An Empirical Study of Fortran Programs, *Software Practice and Exterience* 1, 1971, pp. 105-133.

[87] P. M. Kogge, The Microprogramming of Pipelined Processors, *Proceedings of the 4th Annual Symposium on Computer Architecture,* 1977, pp. 63-69.

[88] R. F. Krick, A. Dollas, "The Evolution of Instruction Sequencing,"*IEEE Computer*, April 1991, pp. 5-15.

[89] D. Kroft, Lockup-free instruction fetch/prefetch cache organization, *Proceedings of the 8th Annual Symposium on Computer Architecture,* 1981, vol. 9, pp. 81-97.

[90] D. J. Kuck, Y. Muraoka, S.-C. Chen, On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup, *IEEE Transactions on Computers* C-21(12), 1972, pp. 1293-1310.

[91] D. J. Kuck, et al., "Measurements of Parallelism in Ordinary FORTRAN Programs,"*IEEE Computer*, January 1974, pp. 37-46.

[92] M. Kumar, Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications, *IEEE Transactions on Computers* C-37(9), 1988, pp. 1088-1098.

[93] S. R. Kunkel, J. E. Smith, Optimal Pipelining in Supercomputers, *Proceedings of the 13th Annual International Symposium on Computer Architecture,* 1986, pp. 404-411.

[94] L. Kurian, P. T. Hulina, L. D. Coraor, D. N. Mannai, Classification and Performance Evaluation of Instruction Buffering Techniques, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 150-159.

[95] L. Kurian, P. T. Hulina, L. D. Coraor, Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module, *Proceedings of the 19th Annual International Symposium on Computer Architecture,* 1992, vol. 20, pp. 236-245.

[96] J. Lah, D. E. Atkins, Tree compaction of microprograms, *Proceedings of the 16th Annual Workshop on Microprogramming,* 1983, pp. 22-33.

[97] M. Laird, A Comparison of Three Current Superscalar Designs, *Computer Architecture News* 20(3), 1992, pp. 14-21.

[98] M. S.-L. Lam, *A Systolic Array Optimizing Compiler,* Ph. D., Carnegie Mellon University, 1987.

[99] M. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, R. L. Wexelblats (Ed.), *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* 1988, vol. 23, pp. 318-328.

[100] M. S. Lam, R. P. Wilson, Limits of Control Flow on Parallelism, *Proceedings of the 19th Annual International Symposium on Computer Architecture,* 1992, vol. 20, pp. 46-57.

[101] B. W. Lampson, Fast Procedure Calls, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 66-76.

[102] R. L. Lee, A. Y. Kwok, F. A. Briggs, The Floating Point Performance of a Superscalar SPARC Processor, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 28-37.

[103] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors,"*IEEE Computer*, July 1988, pp. 47-55.

[104] J. L. Linn, SRDAG compaction: A generalization of trace scheduling to increase the use of global context information, *Proceedings of the 16th Annual Workshop on Microprogramming,* 1983, pp. 11-22.

[105] C. E. Love, *An Investigation of Static Versus Dynamic Scheduling,* Master's Thesis, University of Colorado at Boulder, 1989.

[106] C. E. Love, *The Decoupled And VLIW Architecture Simulator Code,* Department of Electrical Engineering, University of Colorado, Internal Report CSDG 89-4, May 1989.

[107] C. E. Love, H. F. Jordan, An Investigation of Static Versus Dynamic Scheduling, *Proceedings of the 17th Annual International Symposium on Computer Architecture,* 1990, vol. 18, pp. 192-201.

[108] S. A. Mahlke, W. Y. Chen, W.-M. W. Hwu, B. R. Rau, M. S. Schlansker, Sentinel Scheduling for VLIW and Superscalar Processors, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, vol. 20, pp. 238-247.

[109] W. Mangione-Smith, S. G. Abraham, E. S. Davidson, The Effects of Memory Latency and Fine-Grain Parallelism on Astronautics ZS-1 Performance, *Proceedings of the 23rd Hawaii International Conference on System Sciences,* 1990, pp. 288-296.

[110] W. Mangione-Smith, S. G. Abraham, E. S. Davidson, Vector Register Design for Polycyclic Vector Scheduling, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 154-163.

[111] W. M. Mangione-Smith, S. G. Abraham, E. S. Davidson, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1,"*IEEE Computer*, January 1991, pp. 39-46.

[112] S. McFarling, J. Hennessy, Reducing the Cost of Branches, *Proceedings of the 13th Annual International Symposium on Computer Architecture,* 1986, pp. 396-403.

[113] F. H. McMahon, *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range,* Lawrence Livermore National Laboratory, Technical UCRL-53745, December 1986.

[114] S. Melvin, Y. Patt, Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques, *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 287-296.

[115] C. L. Mitchell, *Processor Architecture and Cache Performance,* PhD, Stanford University, 1986.

[116] J. C. Mogul, A. Borg, The Effect of Context Switches on Cache Performance, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991, vol. 19, pp. 75-84.

[117] Motorola, *MC 88100 RISC Microprocessor User's Manual,* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[118] T. C. Mowry, M. S. Lam, A. Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, vol. 20, pp. 62-75.

[119] T. N. Mudge, et al., "The Design of a Microsupercomputer,"*IEEE Computer*, January 1991, pp. 57-64.

[120] T. N. Mudge, et al., The Design of a GaAs Micro-Supercomputer, *Proceedings of the Hawaii International Conference on System Sciences,* 1991, vol. 1, pp. 421-432.

[121] H. Mulder, Data Buffering: Run-time versus Compile-time Support, *SIGPLAN Notices* 24(5), 1989, pp. 144-151.

[122] H. Mulder, M. J. Flynn, Processor Architecture and Data Buffering, *IEEE Transactions on Computers* 41(10), 1992, pp. 1211-1222.

[123] MultiFlow, *Technical Summary,* MULTIFLOW Computer, Inc., Technical Report June 1987.

[124] K. Murakami, N. Irie, M. Kuga, S. Tomita, SIMP (Single Instruction Stream/

Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture, *Proceedings of the 16th Annual International Symposium on Computer Architecture,* 1989, vol. 17, pp. 78-85.

[125] D. Nagle, *Floating Point Simulation for the GaAs Micro-Supercomputer,* The University of Michigan, Internal Research Report September 1990.

[126] A. Nicolau, J. A. Fisher, Measuring the Parallelism Available for Very Long Instruction Word Architectures, *IEEE Transactions on Computers* C-33(11), 1984, pp. 968-976.

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

[128] R. S. Nikhil, Arvind, Can Dataflow Subsume von Neumann Computing, *Proceedings of the 16th Annual International Symposium on Computer Architecture,* 1989, vol. 17, pp. 262-272.

[129] S. Novack, A. Nicolau, An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism, *Proceedings of the 1992 International Conference on Parallel Processing,* 1992, vol. II, pp. II-297-II-301.

[130] O. A. Olukotun, R. B. Brown, R. J. Lomax, T. N. Mudge, K. A. Sakallah, Multilevel Optimization in the Design of a High-Performance GaAs Microcomputer, *IEEE Journal of Solid-State Circuits* 26(5), 1990, pp. 763-767.

[131] O. A. Olukotun, *Technology-Organization Tradeoffs in the Architecture of a High Performance Processor,* PhD, The University of Michigan, 1991.

[132] O. A. Olukotun, T. N. Mudge, R. B. Brown, Implementing a Cache for a High-Performance GaAs Microprocessor, Z. Vranesics (Ed.), *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 138-147.

[133] K. Olukotun, T. Mudge, Performance Optimization of Pipelined Primary Caches, A. Gottliebs (Ed.), *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, vol. 20, pp. 181-190.

[134] D. A. Padua, M. J. Wolfe, Advanced compiler optimizations for supercomputers, *Communications of the ACM* 29(12), 1986, pp. 1184-1201.

[135] G. M. Papadopoulos, K. R. Traub, Multithreading: A Revisionist View of Dataflow Architectures, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, vol. 19, pp. 342-351.

[136] J. H. Patel, E. S. Davidson, Improving the Throughput of a Pipeline by Insertion of Delays, *Proceedings of the 3rd Annual Symposium on Computer Architecture,*

1976, pp. 159-164.

[137] A. Pleszkun, et al., WISQ: A restartable architecture using queues, *Proceedings of the 14th Annual International Symposium on Computer Architecture,* 1987, pp. 290-299.

[138] A. R. Pleszkun, G. S. Sohi, The Performance Potential of Multiple Functional Unit Processors, *Proceedings of the 15th Annual International Symposium on Computer Architecture,* 1988, pp. 37-44.

[139] D. J. Quammen, D. R. Miller, Flexible Register Management for Sequential Programs, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, vol. 19, pp. 320-329.

[140] G. Radin, The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 39-47.

[141] B. R. Rau, C. D. Glaeser, Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing, *Proceedings of the 14th Annual Workshop on Microprogramming,* 1981, pp. 183-198.

[142] B. R. Rau, C. D. Glaeser, R. L. Picard, Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support, *Proceedings of the 9th Annual Symposium on Computer Architecture,* 1982, vol. 10, pp. 131-139.

[143] B. Rau, D. Glaeser, E. Greenwalt, Architectural Support for the Efficient Generation of Code for Horizontal Architectures, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 96-99.

[144] B. R. Rau, D. W. L. Yen, W. Yen, R. A. Towle, "The Cydra 5 Departmental Supercomputer,"*IEEE Computer,* 1989, pp. 12-35.

[145] B. R. Rau, Data flow and Dependence analysis for instruction level parallelism, *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing,* 1991, .

[146] R. B. Rau, Pseudo-Randomly Interleaved Memory, Z. Vranesics (Ed.), *Proceedings of the 18th Annual International Symposium on Computer Architecture,* 1991, vol. 19, pp. 74-83.

[147] B. R. Rau, e. al., *Code Generation Schema for Modulo Scheduled DO-Loops and WHILE-Loops,* Hewlett-Packard Laboratories, Technical Report HPL-92-47,

1992.

[148] B. R. Rau, M. Lee, P. P. Tirumalai, M. S. Schlansker, *Register Allocation for Modulo Scheduled Loops: Strategies, Algorithms and Heuristics,* Hewlett-Packard Laboratories, Technical Report HPL-92-48, April 1992.

[149] B. R. Rau, M. Lee, P. P. Tirumalai, M. S. Schlansker, Register Allocation for Software Pipelined Loops, R. L. Wexelblats (Ed.), *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation,* 1992, vol. 27, pp. 283-299.

[150] E. M. Riseman, C. C. Foster, The Inhibition of Potential Parallelism by Condition Jumps, *IEEE Transactions on Computers* C-21(12), 1972, pp. 1405-1411.

[151] A. Rogers, K. Li, Software Support for Speculative Loads, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, vol. 20, pp. 38-50.

[152] D. A. Schwartz, *Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs,* PhD, Georgia Institute of Technology, 1985.

[153] A. J. Smith, Cache Memories, *ACM Computing Surverys* 4(3), 1982, pp. 473-530.

[154] J. E. Smith, Decoupled Access/Execute Computer Architectures, *Proceedings of the 9th Annual Symposium on Computer Architecture*, 1982, pp. 112-119.

[155] J. E. Smith, T. J. Kaminski, Varieties of decoupled access/execute computer architectures, *Proceedings of the 20th Allerton Conference*, 1982, pp. 577-586.

[156] J. E. Smith, Decoupled Access/Execute Computer Architectures, *ACM Transactions on Computer Systems* 2(4), 1984, pp. 289-308.

[157] J. E. Smith, S. Weiss, H. Y. Pang, A Simulation Study of Decoupled Architecture Computers, *IEEE Transactions on Computers* C-35(8), 1986, pp. 692-702.

[158] J. E. Smith, et al., The ZS-1 Central Processor, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* 1987, pp. 199-204.

[159] J. E. Smith, S. D. Klinger, *Performance of the Astronautics ZS-1 Central Processor,* Astronautics Corporation of America, Internal Report March 1988.

[160] J. E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1,"*IEEE Computer,* July 1989, pp. 21-35.

[161] M. D. Smith, M. Johnson, M. A. Horowitz, Limits on Multiple Instruction Issue, *Proceedings of the Third International Conference on Architectural Support for*

Programming Languages and Operating Systems, 1989, vol. 17, pp. 290-302.

[162] M. D. Smith, M. S. Lam, M. A. Horowitz, Boosting Beyond Static Scheduling in a Superscalar Processor, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, vol. 18, pp. 345-353.

[163] M. D. Smith, *Tracing with pixie,* Stanford University, Technical April 4 1991.

[164] M. D. Smith, M. Horowitz, M. S. Lam, Efficient Superscalar Performance Through Boosting, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, vol. 20, pp. 248-261.

[165] K. So, V. Zecca, Cache Performance of Vector Processors, *Proceedings of the 15th Annual International Symposium on Computer Architecture,* 1988, pp. 261-268.

[166] G. S. Sohi, S. Vajapeyam, Instruction Issue Logic for High-Performance Interruptible Pipelined Processors, *Proceedings of the 14th Annual International Symposium on Computer Architecture,* 1987, pp. 27-34.

[167] G. S. Sohi, S. Vajapeyam, Tradeoffs in Instruction Format Design for Horizontal Architectures, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* 1989, vol. 17, pp. 15-25.

[168] G. S. Sohi, M. Franklin, High-Bandwidth Data Memory Systems for Superscalar Processors, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, vol. 19, pp. 53-62.

[169] G. S. Sohi, High-Bandwidth Interleaved Memories for Vector Processors--A Simulation Study, *IEEE Transactions on Computers* 42(1), 1993, pp. 76-82.

[170] R. Stallman, *Using and Porting GNU CC,* Free Software Foundation, Inc., 1990.

[171] P. Steenkiste, The Impact of Code Density on Instruction Cache Performance, *Proceedings of the 16th Annual International Symposium on Computer Architecture,* 1989, vol. 17, pp. 252-259.

[172] H. S. Stone, J. Cocke, "Computer Architecture in the 1990s,"*IEEE Computer*, September 1991, pp. 30-38.

[173] J.-h. Tang, E. Davidson, J. Tong, Polycyclic Vector Scheduling vs. Chaining on 1-Port Vector Supercomputers, *Proceedings of the 1988 International Conference on Supercomputing,* 1988, pp. 122-129.

[174] Thornton, Parallel Operation in the Control Data 6600, *AFIPS Proceedings*

FJCC, part 2, 1964, vol. 26, pp. 33-40.

[175] J. E. Thornton, *Design of a Computer -- The Control Data 6600*, Scott, Foresmann and Co., Glenview, Ill., 1970.

[176] P. Tirumalai, M. Lee, M. S. Schlansker, Parallelization of loops with exits on pipelined architectures, *Proceedings of the 1990 International Conference on Supercomputing*, 1990, pp. 200-212.

[177] G. S. Tjaden, M. J. Flynn, Detection and Parallel Execution of Independent Instructions, *IEEE Transactions on Computers* C-19(10), 1970, pp. 889-895.

[178] G. S. Tjaden, M. J. Flynn, Representation of Concurrency with Ordering Matrices, *IEEE Transactions on Computers* C-22(8), 1973, pp. 752-761.

[179] R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development* 11(1), 1967, pp. 25-33.

[180] H. C. Torng, M. Day, Interrupt Handling for Out-or-Order Execution Processors, *IEEE Transactions on Computers* 42(1), 1993, pp. 122-127.

[181] R. F. Touzeau, A Fortran Compiler for the FPS-164 Scientific Computer, *Proceedings of the ACM SIGPLAN '84 Conference on Programming Language Design and Implementation*, 1984, pp. 48-57.

[182] A. K. Uht, Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream, *IEEE Transactions on Computers* 41(7), 1992, pp. 826-841.

[183] A. K. Uht, Requirements for Optimal Execution of Loops with Tests, *IEEE Transactions on Parallel and Distributed Systems* 3(3), 1992, pp. 573-581.

[184] J. Uniejewski, SPEC Benchmark Suite: Designed for Today's Advanced Systems, *SPEC Newsletter* (Fall), 1989

[185] M. Upton, T. Huff, T. Mudge, R. Brown, Resource Allocation in a High Clock Rate Microprocessor, preprint.

[186] S. Vajapeyam, G. S. Sohi, W.-C. Hsu, An Empirical Study of the CRAY Y-MP Processor using the PERFECT Club Benchmarks, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, vol. 19, pp. 170-179.

[187] S. Vassiliadis, B. Blaner, R. J. Eickemeyer, On the Attributes of the SCISM Organization, *Computer Architecture News* 20(4), 1992, pp. 44-53.

[188] D. W. Wall, M. L. Powell, The Mahler Experience: Using an Intermediate Language as the Machine Description, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating*

Systems, 1987, pp. 100-104.

[189] D. W. Wall, Limits of Instructional-Level Parallelism, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, vol. 19, pp. 176-188.

[190] S. Weiss, J. E. Smith, Instruction Issue Logic in Pipelined Supercomputers, *IEEE Transactions on Computers* C-33, 1984, pp. 1013-1022.

[191] S. Weiss, J. E. Smith, A Study of Scalar Compilation Techniques for Pipelined Supercomputers, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* 1987, pp. 105-111.

[192] S. Weiss, Optimizing a Superscalar Machine to Run Vector Code, *IEEE Parallel and Distributed Technology* 1(2), 1993, pp. 73-83.

[193] C. A. Wiecek, A Case Study of VAX-11 Instruction Set Usage for Compiler Execution, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, vol. 10, pp. 177-184.

[194] M. Wolfe, Beyond Induction Variables, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* 1992, vol. 27, pp. 162-174.

[195] W. A. Wulf, "Compilers and Computer Architecture,"*IEEE Computer,* July 1981, pp. 41-47.

[196] Q. Yang, L. W. Yang, A Novel Cache Design for Vector Processing, *Proceedings of the 19th Annual International Symposium on Computer Architecture,* 1992, vol. 20, pp. 362-371.

ABSTRACT

LOOP OPTIMIZATION TECHNIQUES ON MULTI-ISSUE ARCHITECTURES

by
Dan Richard Kaiser

Chair: Trevor Mudge

This work examines the interaction of compiler scheduling techniques with processor features such as the instruction issue policy. Scheduling techniques designed to exploit instruction level parallelism are employed to schedule instructions for a set of multi-issue architectures. A compiler is developed which supports block scheduling, loop unrolling, and software pipelining for a range of target architectures. The compiler supports aggressive loop optimizations such as induction variable detection and strength reduction, and code hoisting. A set of machine configurations based on the MIPS R3000 ISA are simulated, allowing the performance of the combined compiler-processor to be studied. The Aurora III, a prototype superscalar processor, is used as a case study for the interaction of compiler scheduling techniques with processor architecture.

Our results show that the scheduling technique chosen for the compiler has a significant impact on the overall system performance and can even change the rank ordering when comparing the performance of VLIW, DAE and superscalar architectures. Our results further show that, while significant, the performance effects of the instruction issue policy may not be as large as the effects of other processor features, which may be less costly to implement, such as 64 bit wide data paths or store buffers.