RICE UNIVERSITY

Stochastic Instruction Scheduling

by

Philip J. Schielke

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Keith Cooper, Associate Professor
Department of Computer Science
Rice University

Devika Subramanian, Associate Professor
Department of Computer Science
Rice University

Linda Torczon, Research Scientist
Department of Computer Science
Rice University

John Bennett, Associate Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas

May, 2000

Stochastic Instruction Scheduling

Philip J. Schielke

Abstract

Instruction scheduling is a code reordering transformation used to hide latencies present in modern day microprocessors. Scheduling is often critical in achieving peak performance from these processors. The designer of a compiler's instruction scheduler has many choices to make, including the scope of scheduling, the underlying scheduling algorithm, and handling interactions between scheduling and other transformations.

List scheduling algorithms, and variants thereof, have been the dominate algorithms used by instruction schedulers for years. In this work we explore the strengths and weaknesses of this algorithm aided by the use of stochastic scheduling techniques. These new techniques we call RBF (randomized backward and forward scheduling) and iterative repair or IR. We examine how the algorithms perform in a variety of contexts, including different scheduling scope, different scheduling problem instances, different architectural features, and scheduling in the presence of register allocation.

IR is a search framework enjoying a lot of attention in the artificial intelligence community. IR scheduling techniques have shown promise on other scheduling problems such as shuttle mission scheduling. In this work we describe how to target the framework for compiler instruction scheduling. We describe the evolution of our algorithm, how to integrate register pressure concerns, and the technique's performance.

We evaluate our alternative algorithms based on a set of real applications and random instruction scheduling problems. Not surprisingly, list scheduling performs very well when scheduling basic blocks of machine instructions. However, there is some opportunity for alternative techniques when scheduling over larger scopes and targeting more complicated architectures. We describe an interesting link between list scheduling efficacy and the amount of parallelism available in a random problem instance.

Increasingly we see complex microprocessors being used in embedded systems applications. Often the price of such systems is affected by the amount of on-board memory needed to store executable code for the microprocessor. Much work on instruction scheduling has improved the running time of scheduled code while sacrificing code size. Thus, finding scheduling techniques that do not increase code size is another focus of this work. We also develop techniques to decrease code size without increasing running time using genetic algorithms–another stochastic search method.

Acknowledgments

No doubt there are many people that I have come in contact with during my years at Rice who deserve my heartfelt thanks. No doubt I will forget some of you. For this I apologize.

First, I give thanks to God, without whom the opportunity to study at such a fine institution and with such fine scholars would not have been possible. I also thank Him for bringing my wife Kerri and I together during my time at Rice. *Soli Deo Gloria*

I would also like to thank the following:

- my wife Kerri for supporting me in so many ways.
- my parents and sister who have always encouraged me and always loved me.
- my advisor Keith Cooper and co-advisor Devika Subramanian. Not only did they offer me fine examples of scholarship, but they are wonderful people as well.
- the other members of my thesis committee, Linda Torczon, and John Bennett.
- all those that have funded my research over the years including, DARPA, The state of Texas, and Texas Instruments.
- other members of the Massively Scalar Compiler Project over the years, Preston Briggs, Taylor Simpson, Nat McIntosh, John Lu, Lisa Thomas, Li Xu, Jingsong He, Alex Grosul, Steve Reeves, and especially Tim Harvey for his continued support of the compiler infrastructure.
- the other members of the entering PhD class of 1993, Eric Virro, Dylan Cooper, Gerald Kanapathy, Linlong Jiang, my roomie of two years Jon Hall, Cristiana Amza, Dejan Mircevski, and Shriram Krishnamurthi.
- Zoran Budimlic for helping me through the C exams.

- Johnny Chen for the many hours spent working with ASIMOV.

- Sanjeeb Dash for helping me with CPLEX.

- the departmental staff who always did their best to keep things running as smoothly as possible, Iva Jean Jorgensen, Rhonda Guajardo, Debra Purtee, Ann Lugg, Darnell Price, and Donna Jares.

Contents

Illustrations

Chapter 1

Introduction

Modern microprocessors are growing exceedingly complex. Many have multiple pipelined functional units, operations requiring varying numbers of cycles to execute, and complex timing requirements. Many of these processors are designed to exploit the fine grained or *instruction level parallelism* (ILP) available in code. In light of this, compiler optimizations have been developed that attempt to find even more parallelism in programs to improve program performance on these new architectures. For many of these microprocessors it is the compiler's job to manage this ILP and produce an instruction ordering, or *schedule*, that results in the fastest execution time on the target architecture. This job is typically performed by the compiler phase called the *instruction scheduler*.

One algorithm has dominated both the literature on instruction scheduling and the implementations of commercial instruction schedulers. I will refer to this technique as list scheduling. List scheduling is a heuristic-driven, greedy technique that constructs a schedule one time-slot at a time. There is little recent literature describing the strengths and weaknesses of the algorithm or alternatives to it. This is due in part to a limited understanding of the instruction scheduling solution space.

The main thrust of this work is to explore alternative instruction scheduling frameworks based on adding an element of randomness to the scheduling framework. Toward that end we describe two techniques: randomized backward and forward scheduling, or RBF, and iterative repair, or IR. We describe the evolution of our iterative repair algorithm and compare its performance to list scheduling under a variety of different conditions. We have learned many things about the instruction scheduling problem through our design of and experimentation with stochastic scheduling frameworks. This thesis explores how different scheduling scopes, target hardware features, and individual problem instances affect the scheduling problem and the performance of the various scheduling techniques. Finally, we explore how the algorithms can take into account other compiler transformations, specifically register allocation, in order to improve the final optimized code.

One of the motivations behind the scheduling work described in this dissertation is concern over code size. For some years concerns about code size have been out of fashion as memory systems and disk storage became cheaper and cheaper. Now however, we see renewed interest in code size optimization for several reasons. Certain architectural features, such as instruction caches, can benefit from smaller code sizes. Compact code is important when object code is being transmitted over slow communications channels (*e.g.* the Internet). Of primary concern is the growing use of general purpose microprocessors and special purpose digital signal processors (DSP's) in embedded systems applications. More and more consumer products (*e.g.* cars, microwaves, cellular phones) are controlled by microprocessors. The control code for these devices is typically burned into a read only memory (ROM) contained in the device. Thus, small code size keeps ROM sizes small, lowering the overall cost of the device.

Many general purpose modern microprocessors support out-of-order execution. That is, the hardware assumes some of the scheduling duties by having the ability to schedule the start of operations in a different order than they appear in the object code. At run time, the hardware can eliminate many dependences between operations that the compiler had to respect because the values of all variables are known. While the research reported in this thesis pertains less to these machines, code ordering is still important because the number of operations that such hardware can consider for out-of-order execution at a given time is fixed. On these machines, bad scheduling can still produce poor performance. Additionally, the work in chapter 4 on non-local scheduling is of interest. While out-of-order execution machines are prevalent today in the general purpose microprocessor world, VLIW type architectures are making a comeback in this arena and in the embedded systems arena. (Intel's IA-64 and Texas Instruments' TMS320C62xx being examples.) The bulk of this work is targeted towards these types of machines, for which the compiler controls most of the scheduling decisions.

This work is not concerned with software pipelining or modulo scheduling as described by Lam [55], and Allan, *et. al.* [3]. While this technique is important, it is beyond the scope of this work. However, since software pipelining often uses list scheduling as one of its' components, improvements to list scheduling may carry over to some of these more specialized techniques.

The work described in this thesis centers around three questions. The first question is, "How much performance improvement is possible by increasing the scope of scheduling without increasing code size?" That is, we will look at two techniques that increase the scope of the program available to the scheduler, but do not increase the size of the compiled code. The second questions is, "Does the use of alternative algorithms that use randomization improve the quality of the generated schedule?" We develop several novel scheduling techniques that use randomization to generate better schedules. The last question is, "How can we design scheduling algorithms that take the constraints of register allocation into account?" We will look at the possibilities of improving allocation through scheduling, and develop schedulers sensitive to the needs of the register allocator.

The rest of this work is organized as follows. The next three sections discuss the three questions that the rest of the dissertation expands upon. In chapter 2 we look at instruction scheduling from a historical perspective, our view of the instruction scheduling problem, and the list scheduling algorithm. Chapter 3 provides the necessary details on the experimental setup used throughout the rest of the work. In Chapter 4 we examine the problem of expanding the scope of instruction scheduling without increasing the size of the object code, the first question. The next three chapters look at the second question. In Chapter 5 we describe the first of our stochastic scheduling techniques, RBF. Chapter 6 describes the iterative repair framework and the evolution of our applying it to instruction scheduling. In Chapter 7 we use these alternative scheduling techniques to examine the conditions under which we can expect list scheduling to perform well. Chapter 8 explores the conflicts between scheduling and register allocation and ways of managing those conflicts in the context of iterative repair, the third question. Finally, chapter 9 gives some suggestions on directions for future work in these areas.

1.1 The Three Questions

This work can be broken up into three broad categories corresponding to the three questions we have studied. These three considerations are important when reasoning about any compiler optimization technique. They relate to the scope of the optimization, the underlying algorithm performing the optimization, and interactions with other optimizations. Of course these are not completely orthogonal considerations; decisions and assumptions made about one aspect affect the other two.

1.1.1 The scope of scheduling

We are interested in applying our new scheduling algorithms to a variety of instruction scheduling *scopes*. That is, the range of operations in the code that constitute a single instance of an instruction scheduling problem. At the same time we are concerned with limiting code growth for the reasons discussed earlier. We endeavor to answer the question, "How much performance improvement is possible by increasing the scope of scheduling without increasing code size?" On some architectures it may even be possible to reduce code size by using non-local techniques.

Often, schedulers are designed as local transformations. That is, they schedule one *basic block* at a time. A basic block is a maximal length sequence of straight line code. If one instruction in a basic block executes, all of them do. However, basic blocks tend to be small and offer a limited amount of instruction level parallelism. To address this problem, various approaches for regional or global scheduling have been proposed over the years. It is the goal of these techniques to find more parallelism by scheduling several basic blocks together. As an example, consider the assembly-like code in Figure 1.1. We assume a processor with two identical functional units. Assume that `LOAD` and `MULT` require two cycles to complete, and all other operations require only one cycle. The unscheduled code for two basic blocks is show in part (a). The next two parts show scheduled code. Two operations separated by an or-bar (`|`), begin at the same time on different functional units. That is, they execute in parallel. In part (b) each basic block has been individually list scheduled. In part (c) we have moved some operations between blocks, to simulate what a non-local technique may be able to do. Of course we must ensure that these motions are safe, but the example shows the potential benefit of a non-local scheduling technique. We have reduced the total execution time for these two blocks from 11 cycles in part (b) to nine cycles in part (c).

Unfortunately many global scheduling techniques increase the size of the object code to gain runtime improvements. When operations are moved between blocks, some techniques require that copies of these operations be inserted into other blocks to maintain program correctness. Other global techniques may replicate entire basic blocks in an attempt to uncover more instruction level parallelism. In chapter 4 we will look briefly at some of these techniques. Very little work has been done on non-local scheduling with the intent of improving the runtime performance of the object code *and* keeping the object code small. In chapter 4 we formulate extended basic block

```
foo: LOAD @a    => r0
     LOAD @b    => r1
     ADD  r0 r1 => r2
     LOAD @c    => r3
     MULT r3 r0 => r4
     SUB  r4 r2 => r5
     STOR r5 @d
     COMP r4 0  => r6
     BREQ r6 bar baz
bar: LDI  0 => r7
     STOR r7 @e
     EXIT
```

(a) Unscheduled code

```
foo: LOAD @a    => r0 | LOAD @b   => r1
     LOAD @c    => r3 | NOP
     ADD  r0 r1 => r2 | NOP
     MULT r3 r0 => r4 | NOP
     NOP              | NOP
     SUB  r4 r2 => r5 | COMP r4 0 => r6
     STOR r5 @d       | NOP
     BREQ r6 bar baz
bar: LDI  0 => r7     | NOP
     STOR r7 @e       | NOP
     EXIT
```

(b) Basic Block Scheduling

```
foo: LOAD @a    => r0 | LOAD @b   => r1
     LOAD @c    => r3 | LDI  0    => r7
     ADD  r0 r1 => r2 | STOR r7 @e
     MULT r3 r0 => r4 | NOP
     NOP              | NOP
     SUB  r4 r2 => r5 | COMP r4 0 => r6
     BREQ r6 bar baz
bar: STOR r5 @d       | NOP
     EXIT
```

(c) Global Scheduling

Figure 1.1 Example showing benefits of Non-local Scheduling

scheduling. This is a relatively simple approach to regional instruction scheduling that requires no code replication, and only a small amount of data-flow analysis. We contrast this approach with another non-local technique called dominator-path scheduling [80].

1.1.2 The choice of scheduling algorithm

The second question we consider focuses on the choice of the underlying scheduling algorithm. Does the use of alternative algorithms that use randomization improve the quality of the generated schedule?

Joseph Fisher's PhD thesis is a seminal work on scheduling micro-code (see section 2.2.1) [34]. He writes:

> Microcode optimization within basic blocks is not a critical issue. [...] The differences among the best strategies we could find are very small. [...] The best strategies were very close to optimal.

Further he quotes Graham Wood [87]:

> On reading the referenced papers, one unavoidably gains the impression that the automatic packing of micro-operations into micro-instruction words is a critical area of research into which a great deal of effort must be invested before user-micro-programmable systems become feasible.
>
> Experience with the above program, however, has led the author to conclude that, in practical situations, very little scope exists in which to practise [sic] the art of optimal packing. Straight line segments of vertical micro-operations typically are not very long – sequences of more than about ten statements without a jump or a label are uncommon, and the data inter-dependency in such cases is liable to be great. Scope for optimisation increases with the degree of parallelism within the micro-instruction word, but, in that case, resource contention is reduced and packing them becomes easy anyway.

Instruction scheduling is well known to be NP-complete under any set of realistic conditions. Thus the above statements seem quite strong. However, they correspond to much of the reported experience with instruction scheduling over the last 20 years. How do such simple algorithms perform so well on such a potentially difficult problem? In this work we explore the answer to that question aided by the use of alternate scheduling techniques including our iterative repair framework.

As presented, the list scheduling algorithm greedily constructs a single schedule from the solution space of all possible schedules based on a given priority heuristic. There is no guarantee that this solution is optimal. At the same time, performing an exhaustive search on the space of all schedules may not be feasible for a given problem instance. In addition, other schedule qualities (such as reduced register pressure, or improved dynamic execution properties) are not easily incorporated into a list scheduling heuristic. In Chapters 5 and 6 we look at some examples where list scheduling fails to find the optimal solution and we explore some alternatives to list scheduling, including randomized backward and forward list scheduling and iterative repair. We also attempt to find metrics that can be used to more easily identify code that may benefit from alternative scheduling algorithms.

As we will see, iterative repair offers several advantages over list scheduling. First, it uses a stochastic method (guided by heuristics) to generate a schedule in the solution space. Thus, many schedules can be generated and the shortest schedule (or "best" schedule by some other criteria) is kept. The iterative repair algorithm works by making small modifications to a complete (but not always correct) schedule. Therefore, it is able to evaluate the outcome of a scheduling choice before committing to it. This added flexibility comes at the cost of longer scheduling times.

In Chapter 7 we look at scheduling problems where simple list scheduling may not be the best choice. We explore how the size of the problem and the structure of the dependency graph affects the efficacy of list scheduling. We show how to estimate the optimal schedule for a given problem instance, and how to make intelligent choices about which scheduling technique to use.

1.1.3 Interaction between scheduling and register allocation

The machine dependent compiler transformations that occur near the end of the compilation process are often referred to as the *back end.* Instruction scheduling and register allocation typically fall into this category. The ordering of these two phases can vary from compiler to compiler. The performance of one can affect the efficacy of the other when it is run later. The third question we address is, "How can we design scheduling algorithms that take the constraints of register allocation into account?"

Unfortunately, scheduling tends to increase the live ranges of registers, thus making register allocation more difficult. We could perform register allocation before scheduling, but that would introduce artificial dependences into the scheduling graph making scheduling more difficult. Since it is not always clear in what order to perform these two code transformations, some sort of combined scheduling/allocation phase looks appealing. In chapter 8 we explore the tension between scheduling and register allocation in more depth, and give a detailed example of the problem. We examine how much potential there may be for scheduling solutions that take register pressure into account. We also explore ways of combining the two problems in the context of our iterative repair scheduling framework.

Chapter 2

Background and Prior Work

In this chapter we present some historical background on the instruction scheduling problem, our formulation of the problem, and the list scheduling algorithm. First we describe a formalization of the general scheduling problem. Next we look at the history of techniques used in the local *micro-code compaction* problem and the *parallel processor task scheduling* problem, two close relatives of the instruction scheduling problem. Then we compare these problems to the instruction scheduling problem and look at approaches that have been taken in the past. We then describe our model of instruction scheduling, explore scheduling as a constructive search problem, and present the list scheduling algorithm.

2.1 A Formal definition of Scheduling

Scheduling problems appear in many fields including compilation, operating systems, operations research, *etc.* To simplify the following discussion we formally define the scheduling problem here. We will refer to this framework throughout our discussion.

1. a set $\mathcal{R}$ of resources $\mathcal{R} = \{r_1, \ldots, r_{|\mathcal{R}|}\}$, the cardinality of $\mathcal{R}$ being denoted $|\mathcal{R}|$.
2. a length $|\mathcal{R}|$ resource configuration vector $\vec{\mathcal{R}}$ of $\mathbf{N}^{|\mathcal{R}|}$, where the kth entry of $\vec{\mathcal{R}}$ (denoted $\vec{\mathcal{R}}(k)$) gives the number of units of r_k available,
3. a set of $\mathcal{T}$ tasks $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$, the cardinality of $\mathcal{T}$ being denoted $|\mathcal{T}|$,
4. a latency function $l : \mathcal{T} \to \mathbf{N}$, where $l(t_j)$ is the number of time units t_j requires to complete,
5. a resource usage function $ru : \mathcal{T} \to \mathbf{N}^{|\mathcal{R}|}$ which maps a task to resources of each type that a task needs. All these resources have to be available all at once for the task to run.

6. a precedence graph PG $= (\mathcal{T}, \mathcal{E})$ imposing a partial order on $\mathcal{T}$. An edge $(u, v) \in \mathcal{E}$ implies that u must complete before v begins. (*i.e.*, v is dependent on u.)

Of course this model changes and is applied in different ways depending on the particular scheduling problem.

2.2 Prior work on various scheduling problems

In this section we look at previous work on instruction scheduling and problems related to instruction scheduling. We discuss micro-code compaction, parallel processor task scheduling, and finally instruction scheduling.

2.2.1 Micro-code compaction

The first thread we explore is that of micro-code compaction. Micro-code can be viewed as a level of abstraction below assembly code. It controls the specific low level details of how the hardware computes the various operations at the machine code level. The micro-code compaction problem is a close cousin of the instruction scheduling problem. There are several key differences. Typically, there is no notion of pipelining at the microcode level, as there is at the machine code level. A single micro-operation may control more than one hardware resource, whereas a machine code operation typically executes on a single functional unit. A micro-code scheduler often has to be concerned with transitory data resources (such as latches) whose contents may become undefined over time. (Beaty surveys some approaches to dealing with this problem in his thesis [4].) The instruction scheduler schedules operations that receive input from registers or memory which are not transitory.

Input to the compaction algorithm is a *microprogram* consisting of a sequence of *micro-operations* or MO's. The compaction algorithm constructs *micro-instructions* (or MI's) from the MO's on a basic block by basic block basis. An MI is made up of one or more MO's that begin execution at the same time controlling different parts of the hardware. The compaction problem is to compress the MO's into as few MI's as possible, while respecting both their resource needs and their data dependences.

Consider scheduling each basic block of MO's separately. In relation to our formal model, the set $\mathcal{R}$ is the set of hardware resources in the machine that can be controlled by the microprogram (ALU's, latches, *etc.*), and $\vec{\mathcal{R}}$ simply defines how many of each of these elements are in the machine. The set of tasks $\mathcal{T}$ is the set of MO's in the basic block. The latency function l defines how many machine cycles each MO requires to complete execution. The resource usage function ru defines how many of each resource is needed by the MO.

Microcode compaction techniques have a rich history in the literature. Fisher summarizes much of the early work in his thesis [34]. The surveyed techniques include several exponential time algorithms that return an optimal solution, and several techniques that approximate the optimal solution. The work by Ramamoorthy and Tsuchiya places critical path MO's in MI's first, then schedules other micro-operations around them [72]. Tsuchiya and Gonzalez describe, at a high-level, a repair based approach [84]. We look at this a little more in section 2.2.3. Yau, Schowe, and Tsuchiya describe a technique that constructs micro-instructions one at a time using a technique similar to list scheduling as well as a branch-and-bound algorithm that tries all micro-instruction orderings [88].

Fisher himself did extensive testing of local compaction algorithms for his dissertation. Specifically, he looked at 13 different priority heuristics for list scheduling. One of the best techniques, called *Highest Levels First* or HLF, is used in by our list scheduling algorithm. (See section 2.5 for more information.) As an example of a production compiler using these techniques, the Fortran compiler for the Floating Point Systems FPS-164 used list scheduling with the HLF scheme [83].

In the early 80's, Davidson, Landskov, Mallett, and Shriver published two papers on local micro-code compaction [56, 28]. In these papers, the authors compared several techniques for compacting micro-code. They concluded that *linear* algorithms (those that operate on micro-code only in original sequence order) were often beaten by other techniques including branch-and-bound, critical path scheduling, and list scheduling. List scheduling competed very favorably against the other good algorithms, was easier to code, and ran faster. The experiments described in [28] consist of three small compiler generated micro-programs.

2.2.2 Processor Task Scheduling

Another problem related to instruction scheduling is task scheduling on a multiprocessor computer. A program can be viewed as a set of tasks ($\mathcal{T}$). There is a partial order imposed on these tasks that must be preserved to insure program correctness. In reference to our formal model, the set $\mathcal{R}$ contains the different types of processors available. However, the typical case is that the machine has multiple instances of the same type of processor. Thus the cardinality of $\mathcal{R}$ will be one. This is the situation we assume in the rest of the discussion. The single entry in $\vec{\mathcal{R}}$ denotes the number of identical processors in the machine. The latency function l returns the amount of time each task requires. Typically each task simply requires one processor making the function ru trivial. The precedence graph is derived through careful program analysis (or programmer directives) to determine which portions of the program can be run in parallel.

This problem differs from instruction scheduling in several ways. One difference is that processor task scheduling is often done *on-line*. That is, the entire set of tasks may not be known at any given time. Additionally, the scheduler may only know a maximum time for a given task, not the actual time the task will require. These factors imply that scheduling cannot be done off-line. In processor task scheduling, the resources (processors) are typically equivalent. Any task can run on any processor. The job of the instruction scheduler, is to schedule the subparts of a task on a particular processor. The processor may have different functional units capable of executing different operations. In task scheduling it may be possible to preempt long tasks. Such preemption is not possible at the instruction level. Despite these differences, work on processor task scheduling has influenced research in both instruction scheduling and micro-code compaction.

A very early paper by Heller takes up the problem of scheduling a program and a set of programs on a machine with multiple processing elements [45]. Heller introduces the notion of using a partial ordering on tasks to drive scheduling. He suggests sampling schedules randomly and choosing the best one after a number of trials; a theme that will re-appear when we describe iterative repair.

Codd and Denning survey several techniques for scheduling tasks from an operating systems point of view, including list scheduling [20]. They suggest that when a processor becomes free to execute another job, that a *priority list* of tasks be scanned for the first task whose predecessors have all completed. Graham mentions such a technique a few years earlier, and presents some preliminary lower bounds on how well such a technique can perform [43]. Adam, *et. al.* evaluate several list scheduling priority techniques for task scheduling [1]. A priority heuristic very similar to the one we develop later was found to be within 4.4 percent from optimal in all but one of the random test cases that was generated. No experiments on real problems were performed.

2.2.3 Instruction Scheduling

In this section, we formalize the instruction scheduling problem, and review past approaches to the problem.

Modern processors employ various techniques to exploit the instruction level parallelism (or ILP) available in programs. For example, a single processor may have multiple functional units, each capable of executing different operations in parallel. Pipelined processors and functional units are also widely used. Not only are individual operations broken up into pipeline stages, but the *execute* stage of the pipeline may itself be pipelined, allowing for even more parallelism. On some architectures, instruction scheduling is merely an optimization technique used to hide the latency of some operations. We say these machines have hardware interlocks or protected pipelines. If one operation depends upon the result of another operation, the hardware ensures that the operation doesn't start until all of its operands are ready. On VLIW machines, the job of ensuring that one operation doesn't start before its operands are ready falls to the compiler.

We adapt the formal model to instruction scheduling in the following way. First, we refer to the partial ordering between operations as a data precedence graph or DPG. The set $\mathcal{R}$, we now denote $\mathcal{F}$ which is the set of functional units available in a given uniprocessor machine $\mathcal{M}$. The set of tasks $\mathcal{T}$, we now denote $\mathcal{O}$ which is the set of operations to be scheduled on $\mathcal{M}$. We modify the notion of the resource usage function ru. In our model, a machine operation only requires one functional unit. Thus, we define a new function $fu : \mathcal{O} \rightarrow \{\mathcal{F}\}$ which returns the set of functional units that can execute a particular operation. We add a functional unit parameter to the latency function l, so l now returns the latency of a particular operation running on a particular functional unit.

Because of pipelined execution units, it is often the case that a new operation can begin on a particular functional unit each cycle. In essence, an operation requires the resource (or functional unit) for only one cycle. Consider the pipelined floating point unit of the Power PC 750 (PPC750.) The pipeline has three stages and most floating point operations complete in three cycles. When an operation is started on the unit, another one can start during the next clock cycle instead of waiting three cycles for the previous operation to complete. Thus, we say the operation only requires the unit for one cycle. We refer to this as *issue time*. The issue time of an operation on a particular functional unit, $(it(op, fu))$ is the number of cycles after op begins that another operation can start on fu. The floating point multiply operation on the PPC750 requires four cycles, spending two cycles in the first stage of the pipeline. Thus we say it has a latency of four cycles and an issue time of two cycles. Integer divide requires 19 cycles to complete on the single stage integer pipeline. Thus, it has a latency of 19 cycles and an issue time of 19 cycles, since no other operation can enter the integer unit until the divide completes.

The following is a specialized version of our formal scheduling model for instruction scheduling. The model is adapted from Beaty's thesis [4] which was taken from Gasperoni [38]. It summarizes the discussion above.

1. a machine $\mathcal{M}$, a set of functional units $\mathcal{F} = \{f_1, \ldots, f_m\}$ that the machine possesses,

2. a functional unit configuration vector $\vec{\mathcal{F}}_{\mathcal{M}}$ of $\mathbf{N^m}$, where the kth entry of $\vec{\mathcal{F}}_{\mathcal{M}}$ (denoted $\vec{\mathcal{F}}_{\mathcal{M}}(k)$) gives the number of functional units f_k available in the machine configuration,

3. a set of n operations $\mathcal{O} = \{op_1, op_2, \ldots, op_n\}$,

4. a latency function $l : \mathcal{O} \times \mathcal{F} \to \mathbf{N}$, where $l(op_i, f_j)$ is the number of machine cycles op_i takes to execute on functional unit f_j,

5. an issue time $it : \mathcal{O} \times \mathcal{F} \to \mathbf{N}$, where $it(op_i, f_j)$ is the number of machine cycles between when op_i can begin on f_j and another operation can begin on f_j.

6. a functional unit usage function $fu : \mathcal{O} \to \{\mathcal{F}\}$. $fu(op_j)$ returns the set of functional units that can execute op_j,

7. a data precedence graph DPG $= (\mathcal{O}, \mathcal{E})$ imposing a partial ordering on $\mathcal{O}$. An edge $(u, v) \in \mathcal{E}$ implies that u must precede v in the final schedule. (*i.e.,* v is dependent on some value produced by u.)

In relation to our previous model, the set of resources is now the set of functional units in the machine. The set of tasks is now the set of machine operations to be scheduled. The notion of issue time is needed for pipelined execution. Issue time is similar to Beaty's notion of delay on the edges of the DPG.

Survey papers by Krishnamurthy, and Smotherman *et. al.* describe several broad categories of work on instruction scheduling for pipelined processors [54, 77]. Most of the techniques surveyed are variations on the theme of list scheduling. (See section 2.5). Some of the better known works include work by Bernstein *et. al.* [5, 7], Gibbons and Muchnik [39], Shieh and Papachristou [75], and Ellis [33]. Many of these techniques vary the properties of the DPG in various ways. Our DPG formulation is described in the next section. A good broad overview of the instruction scheduling problem can be found in chapter 17 of Muchnick's compiler textbook [65].

In this work we compare a simple (but powerful) list scheduling algorithm to several algorithms that incorporate a certain amount of randomness in the scheduling process. Randomness has been used in a few other approaches to the problem. Beaty's dissertation describes using genetic algorithms to perform instruction scheduling [4]. He compares various genetic algorithm formulations, some being based on a list scheduler and some not. He shows improvements over list scheduling in four of the six Livermore loops reported. Our work differs in several ways. First, the iterative repair approach is a fundamentally different framework than list scheduling. Second, this work focuses on trying to outperform list scheduling at any cost, and is only secondarily concerned with comparing the convergence properties of various formulations of the iterative repair algorithm. Finally, we describe more exhaustive experimental testing in this work.

Another approach has been taken by Moss, *et. al.* [62]. Their approach is to use various machine learning techniques in an attempt to "learn" the best list scheduling heuristic. They evaluate different learning methods and identify important features. While their technique is interesting, they have difficulty outperforming the best production instruction schedulers on the SPEC95 benchmarks.

For a slight digression, we mention again the work of Tsuchiya and Gonzales on microcode compaction [84]. This work is of interest because it appears to be the first case of a repair based approach to a problem so closely tied to instruction scheduling. They schedule dags of micro-operations that all have unit latency. [1] The first step is to "partition" the graph in two ways: the first being to schedule all operations as early as possible with respect to precedence constraints, and the second, as late as possible. Of course, either schedule may be invalid due to resource contention during a given cycle. To fix this contention problem they select an operation or set of operations to "delay" (*i.e.,* move later in the schedule). Instead of exhaustively trying

[1] This problem is NP-complete when the number of processors is greater than two and the DPG is not a tree. [19]

all combinations of which operations to delay, they develop a complex set of heuristics to determine which operations are selected. They present little experimental analysis on the efficacy of their algorithm.

Figure 2.1 shows some major works in the development of instruction scheduling techniques over the years, and how they fit together. An edge from work A to work B means that A cites B.

There has been some work done on the complexity of scheduling problems as well as what types of problems are more difficult. Palem and Simons have shown that any greedy algorithm produces a schedule within a factor of two of the optimal schedule for a target machine with multiple identical pipelines [68]. They also show that their "rank" algorithm is able to optimally schedule a certain class of DPG's (the monotone interval order graphs) in linear time, again on machines with multiple identical pipelines. Our experience is that these graphs occur infrequently in real codes (at least in problems that are large enough to be interesting.) Also, many modern architectures do not have identical pipelines.

2.3 Formulation of the Instruction Scheduling Problem

Here we describe the necessary details on the formulation of the instruction scheduling problem as used in the rest of this work.

2.3.1 Operations, Instructions, and the DPG

We make a distinction between machine operations and machine instructions. An *operation* is a single indivisible command sent to the microprocessor. Examples include an add operation, a load from memory operation, or a branch operation. We define an *instruction* to be a set of operations that begin at the same time on different functional units. Thus, the input to the scheduler is a sequence of operations, and the output is a sequence of instructions each containing an operation for each functional unit of the target architecture. We assume that the scheduler must insert `NOP` operations to enforce dependences between operations (a VLIW model.) An example of this process is shown in figure 2.2. In this example we show an unscheduled piece of code on the left, and the same fragment after scheduling on the right. We assume that the `ADD` (integer addition) operation requires one cycle, `LDI` (load immediate value) one cycle, the `MUL` (integer multiplication) operation two cycles, and the `LD` (integer load

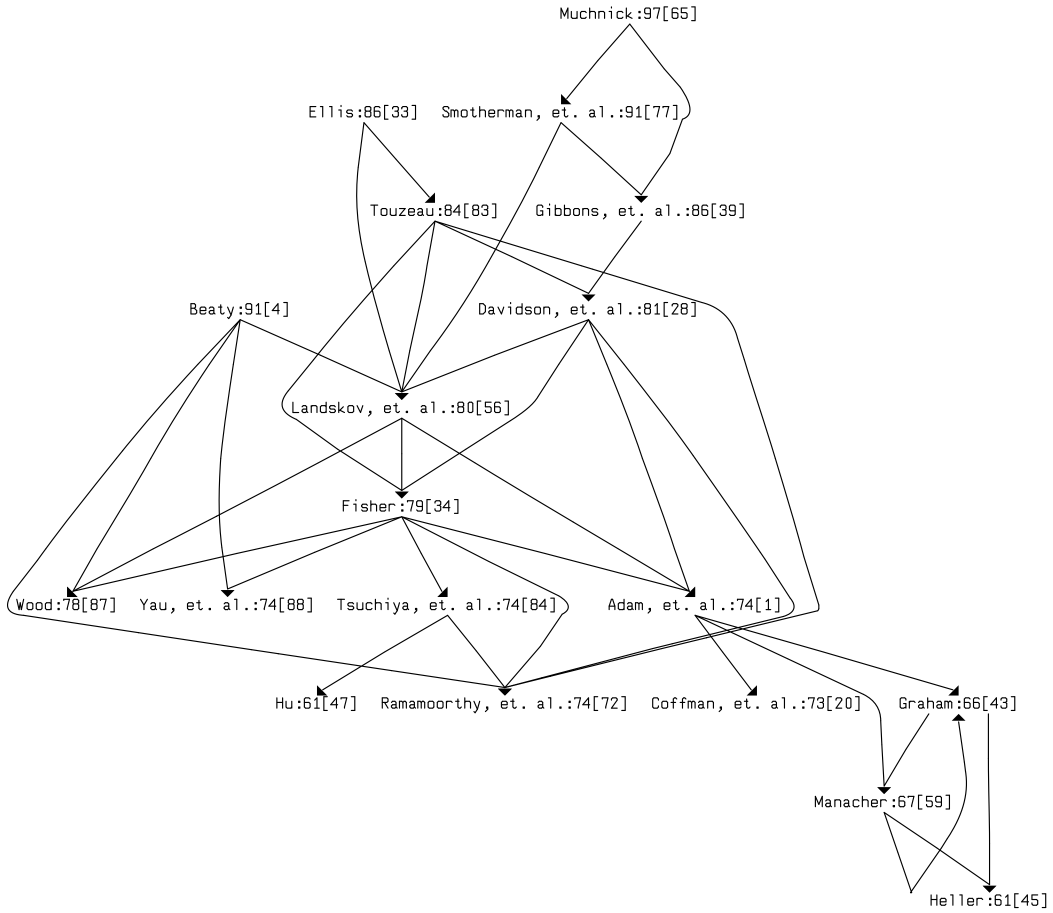


Figure 2.1 Work on instruction scheduling and related problems. An edge from A to B means that A cites B.

with offset) operation three cycles. We are scheduling this code on a machine with two identical functional units.

The first phase of the instruction scheduler is to build the DPG for the operations being scheduled. (This is also referred to as the data dependence graph or DDG in the literature.) Nodes in the graph represent operations, and edges between nodes represent dependences between operations. (Throughout this thesis we will be using the terms node and operations interchangeably as the context warrants.) Edges in our graphs follow the direction of the flow of values through the code. For example, if operation v must start after operation u completes (presumably because v requires a value that u computes) an edge is added from node u to node v in the graph. (*i.e.,* $(u, v) \in \mathcal{E}$.) An edge from u to v means that v somehow depends on a value computed by u. Note that this graph is a directed acyclic graph or DAG. The roots of the graph are all operations which depend on no other operations (*i.e.,* they have no parents in the graph), and the leaves are those operations on which no other operation depends (*i.e.,* they have no children in the graph.)

In an attempt to provide more flexibility to the scheduler, our DPG's have two different kinds of edges: *regular* edges, and *anti-edges*. Regular edges are used to enforce true and output dependences between operations, and anti-edges are used to enforce anti dependences between operations. A *true dependence* exists from operation u to operation v, if operation u computes a value that v uses (*i.e.,* a read after a write). An *output dependence* from u to v means that u and v compute a value that is being stored into the same physical register or memory location and u appears before v in the original program (*i.e.,* a write after a write). An *anti-dependence* from u to v means that v is storing a value into a location from which u is reading a value earlier in the operation sequence (*i.e.,* a write after a read). [2] In our model we assume that a regular edge from operation u to operation v implies that u must complete its execution (that is must finish storing all values it computes) before operation v can begin. An anti-edge from u to v implies that operation v must start at the same time as operation u or any amount of time later. Thus, we define our DPG's as triples: DPG$\equiv (\mathcal{O}, \mathcal{E}, \mathcal{E}')$ where $\mathcal{O}$ is the set of operations or nodes, $\mathcal{E}$ the set of regular edges, and $\mathcal{E}'$ the set of anti-edges. Output dependences are included in $\mathcal{E}$.

---

[2] We could use a renaming technique such as SSA [26] to eliminate anti and output dependences, but this could lead to insertion of copy operations, and could not be used to schedule code that has been register allocated.

```
LDI 2 => r1
LD @a 0 r0 => r2
LD @b 4 r0 => r3
MUL r1 r2 => r4
ADD r1 r3 => r5
MUL r4 r5 => r6
ADD r1 r7 => r7
```

(a) Before Scheduling

```
LD @a 0 r0 => r2  | LD @b 4 r0 => r3
LDI 2 => r1       |   NOP
ADD r1 r7 => r7   |   NOP
MUL r1 r2 => r4   | ADD r1 r3 => r5
  NOP             |   NOP
MUL r4 r5 => r6   |   NOP
  NOP             |   NOP
```

(b) After Scheduling

Figure 2.2 Scheduling example

A code fragment and its corresponding DPG is shown in figure 2.3. Solid arrows are used to indicate the regular edges, and dashed arrows are used to indicate the anti-edges. Notice the anti-edge between operation 5 and 6. Operation 5 reads from `r2` and and operation 6 writes into it.

2.3.2 Hardware Issues

It will be important to distinguish between individual functional units, and what we call functional unit types. The set $\mathcal{F}$ from our formal description is a set of functional unit types, with $\vec{\mathcal{F}}_{\mathcal{M}}(k)$ denoting the number of functional units of type k in the architecture. Each functional unit (*fu*) can execute a fixed set of operations. We denote this set as $ops(fu)$. Each operation on a given functional unit has a certain latency, $l(op, fu)$, and issue time, $it(op, fu)$. A functional unit type (or *fu_type*) is a set of identical functional units (one element of $\mathcal{F}$.) That is all functional units of the same type execute the same operations with the same latency and issue time. The

```
0: LDI 4 => r1
1: ADD r1 r0 => r2
2: MUL r3 r4 => r5
3: LD @a 0 r2 => r6
4: ADD r5 r6 => r7
5: ST @a 0 r2 r7
6: LDI 0 => r2
```

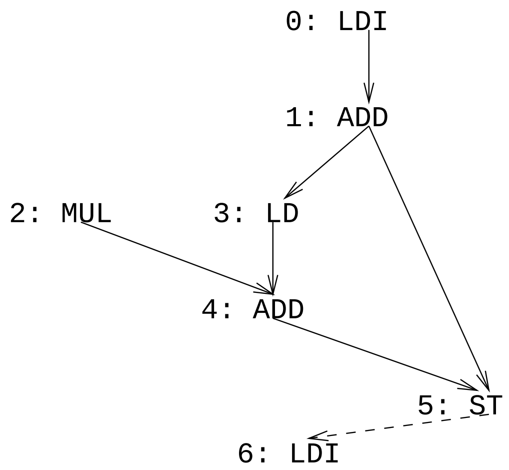


Figure 2.3 DPG example

set of *fu_types* for a given machine is a partitioning of the functional units on that machine. We extend the domain of $ops()$ to include functional unit types. That is:

$$ops(fu\_type) \equiv \{ops(fu) \mid fu \in fu\_type\}$$

We formally describe an fu_type as follows:

$$\forall fu \in fu\_type \forall fu' \in fu\_type (ops(fu) = ops(fu'))$$

That is, each fu in the fu_type must be able to execute the same set of operations. Additionally:

$$\forall fu \in fu\_type \forall fu' \in fu\_type \forall op \in ops(fu)(latency(op, fu) = latency(op, fu'))$$

and,

$$\forall fu \in fu\_type \forall fu' \in fu\_type \forall op \in ops(fu)(issuetime(op, fu) = issuetime(op, fu'))$$

All of the architectural models used in this work assume a VLIW type architecture. That is, the scheduler emits a sequence of operations that begin execution at the same time (an instruction.) To simulate scheduling code on a super-scalar architecture, we limit the number of operations that can begin in parallel during a given cycle. For example, there exist architectures with n functional units that can only initiate $m < n$ operations each cycle. We refer to m as the number of *issue slots* in the architecture, and add this to our machine model.

2.4 Scheduling as a search problem

Scheduling can be viewed as a search in the space of possible or feasible schedules. That is, one can design an algorithm that searches this space, and attempts to find the best schedule in that space. The solution space of such a technique would be the set of all possible schedules that satisfy the precedence constraints in the DPG and the resource constraints of the target architecture, or some subset thereof (*e.g.* the greedy schedules. [3])

The *constructive* search approach builds up a schedule one operation or cycle at a time. "Constructive methods incrementally extend valid, partial schedules until a complete schedule is synthesized or until backtracking is required" [90]. For even the simplest problems constructive spaces can become quite large.

Consider the following constructive approach. Begin with the empty schedule. At each level X of the search tree, select which operations will begin in cycle X, based on hardware and dependence constraints. Further we will consider only greedy schedules. Figure 2.4 shows part of such a search space for the code from Figure 2.2 scheduled on a single processor. At the first level in the tree, after the empty schedule, one of three operations can be scheduled next. The construction continues until all operations have been scheduled. We have shown only one thread through the search space. Other threads will have `NOP`'s in the schedule. The thread shown, happens to be the order in which the list scheduling algorithm, described in the next section, would construct the schedule. As we will see, list scheduling is really a heuristically directed walk through this greedy constructive search space. In Chapter 5 we make additional probes into this search space. In Chapter 6 we describe a completely different search space, the iterative repair space, and describe how it is probed.

2.5 The List Scheduling Algorithm

Here we describe list scheduling, or LS. As described, the algorithm schedules each basic block separately, but it can easily be adapted to schedule over larger program regions (see chapter 4.)

The first step is to construct the DPG as described earlier. Since we are scheduling over basic blocks, we must ensure that all operations within the block complete their

[3]We adopt the notion of greedy as described by Palem and Simons. "Any scheduling algorithm that never inserts a no-op if some instruction is available for scheduling is called a *greedy* scheduling algorithm" [68].

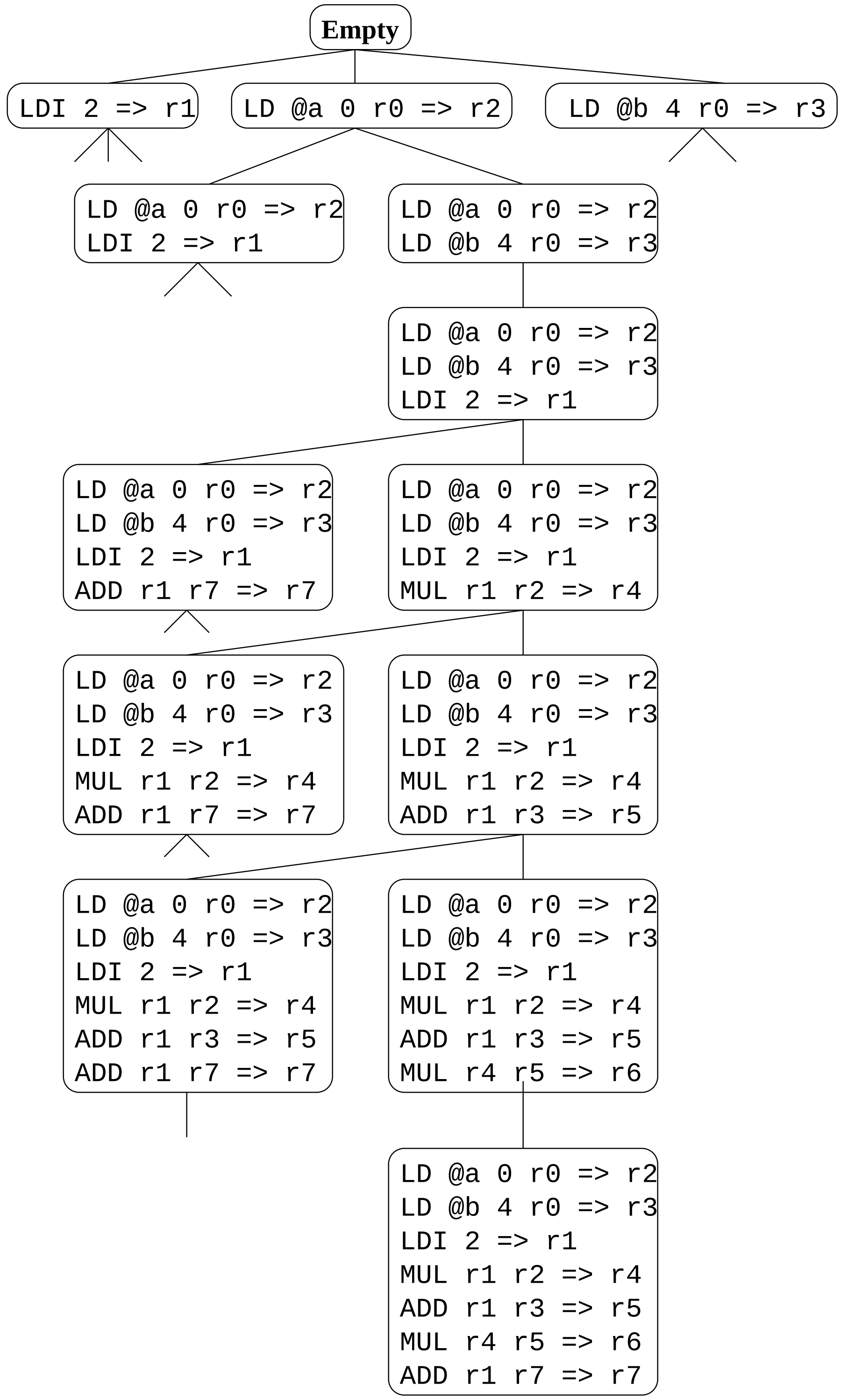


Figure 2.4 Constructive search space for code from Figure 2.2

computations by the end of the schedule. To enforce this we add regular edges from all nodes without children (leaf nodes) to the control-flow operation that ends the block. Thus there is a single leaf node, namely the control-flow operation.

In the next step, priorities are assigned to each node in the graph. There are several different heuristics that can be used to assign priorities. A common and effective strategy is to use the latency weighted depth of the node [39, 56]. (This is essentially the HLF from Fisher's thesis [34].) The *depth* of a node n is the length (number of nodes), of the longest path in the DPG from n to some leaf (including n and the leaf.) [4] The *latency weighted depth*, or LWD, is computed the same way, but the nodes along the path are weighted using the latency of the operation the node represents. As stated earlier there is a single leaf node in our graphs, namely the block ending control-flow operation. The following formula summarizes the priority computation for a node n.

$$priority(n) = \max\left(\forall_{p \in paths(n,\ldots,leaf)} \sum_{p_i=n}^{leaf} latency(p_i)\right)$$

Dynamic programming can be used to compute the priorities efficiently:

$$priority(n) = \begin{cases} latency(n) & \text{if } n \text{ is a leaf.} \\ latency(n) + max_{(m,n)\in E}(priority(m)) & \text{otherwise.} \end{cases}$$

Our DPG's also have anti-dependence edges as described earlier. These edges change the LWD calculation. First we must consider what the LWD actually represents. If we only consider a DPG with regular edges, then the LWD of a node n implies that any schedule requires at least LWD cycles to complete after n begins execution. Using this we can extend the notion of LWD to anti edges as well:

$$priority(n) = \begin{cases} latency(n) & \text{if } n \text{ is a leaf.} \\ max(latency(n) + max_{(m,n)\in E}(priority(m)), & \\ \qquad max_{(m,n)\in E'}(priority(m))) & \text{otherwise.} \end{cases}$$

We define two more functions on operations given a particular schedule or partial schedule. First, $s\_time(op) : \mathcal{O} \rightarrow \mathbf{N}$ returns the time (cycle number) that op is

[4]Using depth as a priority function to drive scheduling is not a new idea or even an idea unique to instruction scheduling or its ancestors. A paper by Hu in 1961 describes using the approach for assembly line problems [47].

scheduled to begin execution. $s\_fu(op) : \mathcal{O} \rightarrow \mathcal{F}$ returns the functional unit type to which op is assigned in the schedule.

The final phase is the actual list scheduling algorithm that constructs the schedule for the block. Starting at cycle 0, the list scheduler places operations into the schedule cycle by cycle. Any operation op that is *ready* at cycle X is a candidate to be scheduled at cycle X. An operation op is ready at cycle X if and only if

$$(\forall_{(op',op)\in\mathcal{E}} X \geq s\_time(op') + l(op', s\_fu(op'))) \wedge (\forall_{(op',op)\in\mathcal{E}'} X \geq s\_time(op'))$$

Intuitively, op is ready if all its regular parents in the DPG have completed their execution, and all its anti-parents have started execution.

The priorities computed previously are used to determine which of the ready operations to schedule, by selecting the highest priority operation first. Any tie in the priority of two operations is broken arbitrarily. The algorithm is detailed in Figure 2.5.

Instruction scheduling under any set of realistic assumptions has been shown to be NP-complete [85]. List scheduling is a polynomial time algorithm that selects one schedule out of the space of possible solutions. Assuming the DPG is built, (an $O(N^2)$ computation), we examine the complexity of the list scheduler as presented in Figure 2.5. Consider line (1) of the algorithm. Each operation is removed from the ready list exactly once. If a heap based priority queue is used as the ready list, this part of the code contributes $O(N \log N)$ to the running time. If we assume a constant bound on the latency of an operation, line (2) contributes $O(N)$. In line (3) we traverse each outgoing edge of the node. This is done for all nodes, giving a complexity of $O(E)$. Thus the worst case is when $E = N^2$, giving a worst case complexity of $O(N^2)$. For a typical scheduling problem $E = O(N)$ and the number of ready operations can be bound by a constant. In practice, the size of the ready list can usually be bound by a small constant. Thus, list scheduling is often linear in practice once the DPG has been built.

Notice that list scheduling is similar to the constructive search technique we described earlier. To optimally solve the problem with constructive search would be quite expensive, since all paths of the search space would need to be explored (modulo any branch-and-bound or other pruning techniques.) List scheduling, however, uses the priority heuristic to direct the walk through the constructive search space. At each branch in the search tree, the priority heuristic is used to determine which path to take.

Input: Data Precedence Graph (N,E) with priorities assigned to each node. Parameters of machine (instruction latencies, issue times, pipelining, number of functional units, etc.)

Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine.

Algorithm:
cycle = 0
ready-list = leaf nodes in DPG
inflight-list = empty list
while (*ready-list* or *inflight-list* not empty, and an issue slot is available)
(1) **for** *op* = (all nodes in *ready-list* in descending priority order)
    if (a functional unit exists for *op* to start at *cycle*)
        remove *op* from *ready-list* and add to *inflight-list*
        add *op* to schedule at time *cycle*
        check all op' where $(op, op') \in \mathcal{E}'$ and add to *ready-list* if ready
    endif
  endfor
  cycle = *cycle* + 1
(2) **for** *op* = (all nodes in *inflight-list*)
    if (*op* finishes at time *cycle*)
        remove *op* from *inflight-list*
(3)   check nodes waiting for *op* in DPG and add to *ready-list* if ready
    endif
  endfor
endwhile

Figure 2.5 List Scheduling algorithm

Another approach to list scheduling is to schedule operation by operation instead of cycle by cycle. That is, instead of picking an operation to schedule in a given cycle, a ready operation is selected (based on some heuristic) and inserted into the partial schedule as early (or late) as possible. For more on this method see the dissertation by Ellis [33].

Chapter 3

Experimental Background

In this chapter we provide the necessary details of our research compiler and our instruction scheduler. We present our experimental framework, the processor models used, and details on the benchmark programs used.

3.1 The Research Compiler and Scheduler

Our instruction scheduler is built upon the infrastructure of our uniprocessor research compiler [11]. The front end of the compiler consumes C or Fortran and compiles it into a low-level intermediate language called ILOC. ILOC is an assembly-like RISC language. All non-memory operations load their operands from virtual registers and store their results to virtual registers. Several memory operations are available to load and store values in the registers to and from memory. The ILOC can be passed to various ILOC-to-ILOC optimization phases that attempt to improve the code in various ways. All the benchmarks used in our experiments have passed through several optimization phases prior to scheduling. These include pointer analysis (for C codes), global value numbering, constant propagation, dead code elimination, lazy code motion, and operator strength reduction. We do not produce target code for a specific architecture. Rather our "back-end" translates each ILOC operation to C code. This C code can be instrumented to gather various statistics including, dynamic instruction counts, and cache behavior properties. This C code is them compiled and executed. In this way we are able to simulate the execution of code on abstract machines having the properties we are interested in. Figure 3.1 shows how the major components of the compiler fit together.

The instruction scheduler has two inputs. The first is a control flow graph or CFG that is produced from the ILOC code. The CFG consists of basic blocks of ILOC operations, and control flow edges between those blocks. The second input to the scheduler is an architecture description file (or ADF) for the target architecture. All our experiments are simulations of hypothetical ILOC machines. The ADF defines how

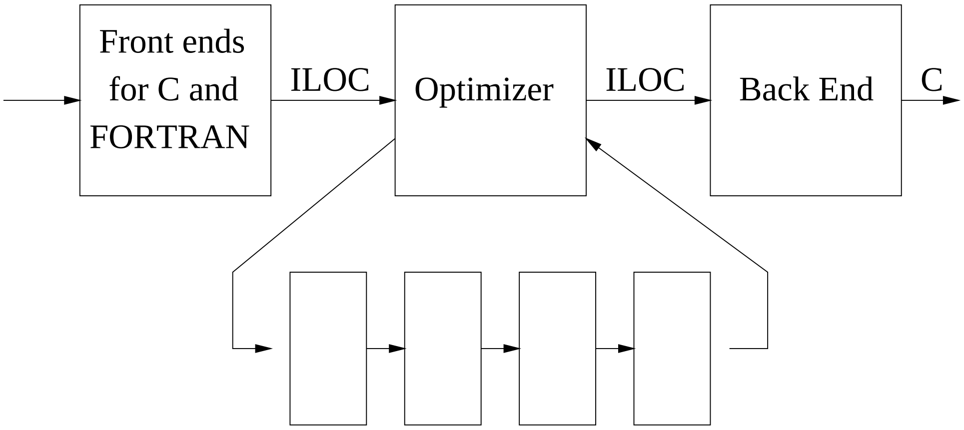


Figure 3.1 The MSCP research compiler

many functional units the machine has, which ILOC operations can execute on each functional unit, the latency (in cycles) of each ILOC operation, and the issue time of each operation. The latency of an operation is the number of cycles required for the operation to write all of its output (to memory or registers) after the operation begins. The ADF also specifies for each operation the number of cycles after which another operation can begin on the same functional unit (*i.e.,* the issue time). A value of unity for all operations means a new operation can begin on the functional unit during any cycle. The three main architectural models we use are described in section 3.2.

We will be interested in both *static* and *dynamic* instruction counts. The static instruction count is simply the number of (potentially parallel) ILOC instructions in the scheduled program. On VLIW architectures `NOP` operations may need to be inserted into the code to insure correctness. We define a `NOP` instruction to be an instruction which contains only `NOP` operations (an idle cycle.) These `NOP` instructions are also included in the static instruction count. In most of our experiments we do not add any additional operations to the code (other than `NOP`s), so the code size on a super-scalar type architecture would remain unchanged after scheduling. The dynamic instruction count is the number of instructions (including `NOP`s) that execute during a run of the scheduled code. This count can also be thought of as the number of cycles required to execute the code.

3.2 Architectural Models

Instruction scheduling is a machine dependent transformation. Any instruction scheduler is targeting a specific piece of hardware. We use several processor architectural models in our experiments. Each model has ILOC as its instruction set, but the latencies of operations, the functional units available and the pipeline characteristics vary from model to model. We assume all operation latencies are known at compile time. (For example, memory operation latencies may vary based on whether the access hits in the cache. We assume, for example, a cache hit for scheduling purposes.) We summarize the models here. Operation latencies, issue time, and the assignment of operations to various functional units are detailed in Appendix A for each of the models described below.

3.2.1 SVLIW

Our first model we call SVLIW which stands for simple VLIW. In this architecture we have two integer units, one floating point unit, one branch unit, and a memory unit. Each ILOC operation is assigned to exactly one type of functional unit. For example the integer add operation can only execute on one of the two integer units. We assume perfect branch prediction so the pipeline length is not important. Full details on the architecture can be found in Figure A.1.

3.2.2 C60

The next model (which we will refer to as the C60) is based on the Texas Instruments TMS320C62xx chip, the newest family of TI's digital signal processors [82]. From a scheduling perspective it has several interesting properties. The TMS320C62xx is a VLIW that allows up to eight operations to be initiated in parallel. All eight functional units are pipelined, and most operations have no delay slots. The exceptions are multiplies (latency of two cycles), branches (latency of six cycles), and loads from memory (latency of five cycles). `NOP`'s are inserted into the schedule for cycles where no operations are scheduled to begin. The `NOP` operation takes one argument specifying the number of idle cycles.

This architecture has a unique way of "packing" operations into an instruction. Operations are always fetched eight at a time. This is called a *fetch packet*. Bit zero of each operation, called the p-bit, specifies the execution grouping of the operation. If the p-bit of an operation o is 1, then operation $o+1$ is executed in parallel with operation o (*i.e.,* they are started in the same cycle). If the p-bit is 0, then operation $o+1$ begins the cycle after operation o. The operations that execute in parallel are called an *execute packet*. All operations in an execute packet must run on different functional units, and up to eight operations are allowed in a single execute packet. Each fetch packet starts a new execute packet, and execute packets cannot cross fetch packet boundaries. This scheme and the multiple-cycle `NOP` operation described above, allow the code for this VLIW to be very compact.

Of course, there is not a one-to-one mapping of ILOC operations to TMS320C62xx operations, but we feel our model highlights most of the salient features of this architecture from a scheduling perspective. Our model has eight fully pipelined functional units. The integer operations have latencies corresponding to the latencies of the TMS320C62xx. Since ILOC has floating point operations and the TMS320C62xx does not, these operations are added to our model. Each floating point operation is executed on a functional unit that executes the corresponding integer operation. Latencies for floating point operations are double those for integer operations. All ILOC intrinsics (cosine, power, square root, *etc.*) have a latency of 20 cycles. The previous models assumed perfect branch prediction. However, the C60 has no mechanism for predicting branches. Thus, every control-flow operation (including an unconditional jump) incurs a five cycle delay to refill the pipeline. We simulate this by adding five cycles to the dynamic instruction count each time a branch, jump, subroutine call, or subroutine return is executed.

Static instruction counts for the model reflect the fetch packet/execute packet scheme. We place as many execute packets as possible in each fetch packet. `NOP`'s in consecutive cycles are treated as a single operation for consistency with the multiple-cycle `NOP` on the C60. Each basic block begins a new fetch packet.

Full details of the architectural model are available in Figure A.2.

3.2.3 PPC750

Our final model is based on Motorola's Power PC 750 architecture [63]. The PPC is a super-scalar architecture, not a VLIW. However, for purposes of scheduling we treat it as a VLIW. We do not report any static operation counts for this architecture since `NOP`'s do not need to be inserted into the instruction stream to enforce dependences between operations. As with the SVLIW model we assume perfect branch prediction.

Our model contains 6 functional units: system register unit, integer unit 1, integer unit 2, floating point unit, memory unit, and branch processing unit. The 750 can only begin two operations each cycle, so we set the number of issue slots for the model to two. We assume cache hits on all memory operations. Thus, a load or a store of thirty-two bits always requires two cycles, with an issue time of one cycle. Like the C60 the functional unit types of the PPC750 are not disjoint. For example, both integer units can perform an integer add, but only one of them can perform an integer multiply or divide.

Another interesting feature, from a scheduling perspective, are the pipelined floating point and integer units. The integer unit only has one stage, but several integer operations require more than one cycle. Therefore, the integer unit is tied up for the entire latency of such operations, and no other operation can start until the previous one completes. The floating point unit is structured as a three stage pipeline. Several floating point operations, however, require more than three cycles, spending as many as 2 cycles in the first stage of the pipeline. A floating point divide of a double requires 31 cycles and no operation can enter the pipeline during that time. These features are modeled using the notion of *issue time* described in the last chapter.

Full details of the architecture are available in Figure A.3.

3.3 Benchmark programs

In order to adequately test our schedulers we have selected a variety of Fortran and C benchmark codes. The Fortran codes used are, the FMM suite [36] (consisting of `fmin`, `rkf45`, `seval`, `solve`, `svd`, `urand`, and `zeroin`), several programs from SPEC '89 (`tomcatv`, `fpppp`, `doduc`, and `matrix300`), `applu` from SPEC '95, and two programs from the NAS benchmark suite (`cg` and `mg`). The other programs are written in C. `adpcm` performs adaptive differential pulse code modulation. `clean` is the empty basic block removal pass of our research compiler. `dfa` is an implementation of the Knuth Morris Pratt string matching algorithm and makes heavy use of pointers. `water` comes from the SPLASH benchmark suite. `compress`, `jpeg`, and `go` are from SPEC. `dhrystone` is a famous hardware benchmarking program. `shorten` is a compressor/coder for waveform files. Not surprisingly, `fft` is an implementation of a 3-D fast Fourier transform algorithm. `gzip` is the GNU compression program. `nsieve` computes prime numbers based on the Sieve of Eratosthenes.

These codes were not hand picked for our particular problem, rather we attempted to collect a diverse selection of codes. The set of benchmarks is split almost evenly between Fortran and C. Several of the Fortran codes have some large basic blocks, including `applu`, `fpppp`, and `doduc`. The benchmark `dfa` is pointer intensive and uses lots of small functions. We use several codes representative of embedded systems applications (`adpcm`, `shorten`, `gzip`, and `jpeg`). Many of the other codes (especially the Fortran codes) are more scientific in nature. The `clean` benchmark comes from our own compiler. It includes a complete table driven parser for ILOC and is representative of a stand-alone compiler optimization pass.

To get an idea of the size of each benchmark, Table 3.1 shows the number of ILOC operations and the number of basic blocks in the optimized version of each code. These numbers reflect the code after optimization and prior to scheduling. C codes are listed first, followed by the Fortran codes.

Benchmark	Num. blocks	Num. ops
`adpcm`	82	339
`clean`	1438	7902
`compress`	227	1279
`dfa`	325	958
`dhrystone`	62	556
`fft`	241	1544
`go`	10059	50401
`gzip`	2093	10264
`jpeg`	1961	13029
`nsieve`	30	223
`shorten`	1379	5746
`water`	303	3509
`fmin`	46	197
`rkf45`	107	783
`seval`	34	354
`solve`	82	479
`svd`	159	979
`urand`	13	101
`zeroin`	28	155
`applu`	590	9798
`cg`	89	589
`doduc`	1572	19439
`fpppp`	314	8627
`matrix300`	50	319
`mg`	161	1789
`tomcatv`	51	684

Table 3.1 Number of basic blocks and ILOC operations in benchmark programs

Chapter 4

Non-local Instruction Scheduling

Because basic blocks typically have a limited amount of parallelism [73], scheduling methods that schedule more than a single basic block at one time have been developed in the hopes of improving program performance. In the next section, we look at some of the methods used in the past. While these methods do improve program performance, they also tend to increase the size of the code. In section 4.2, we look at some "non-local" scheduling techniques designed not to increase the size of the code. We refer to these techniques as non-local instead of global, because they "look" at small regions of basic blocks and not the entire procedure.

4.1 Previous work

There are several approaches that have been used to perform global instruction scheduling. At first, *ad hoc* methods were used to move operations between basic blocks that had already been scheduled in the hopes of improving the overall schedule [35]. *Trace Scheduling* was one of the first automated global scheduling techniques. In trace scheduling, the most frequently executed (loop free) path through the function is determined using profile information. This "trace" is treated like a large basic block. A DPG is created for the trace, and the trace is scheduled using a list scheduler. All the restrictions on moving operations between basic blocks are encoded in the DPG. After the first trace is scheduled, the next most frequently executed path is scheduled, and so on. A certain amount of "bookkeeping" must be done when scheduling the trace. Any operation that moves above a join point (a block with more than one incoming edge) in the trace must be copied into all other traces that enter the current trace at that point. Likewise, any operation that moves below a branch point (a block with more than one outgoing edge) must be copied into the other traces that exit the branch point, if the operation computes any values that are live in that trace. Fisher describes trace scheduling in some detail [35]. The technique has been used successfully in several research and industrial compilers [33, 58].

One criticism of trace scheduling is its potential for code explosion due to the bookkeeping code. Freudenberger, *et al.,* argue that this is not a major problem in practice [37]. They show an average code growth of six percent for the SPEC89 benchmark suite and detail ways to avoid bookkeeping (or compensation) code altogether. Restricting the trace scheduler to produce no compensation code only marginally degrades the performance of the scheduled code. However, most of the results in this work perform some form of loop unrolling prior to scheduling, which increases the code size.

Hwu, *et. al.* present another global scheduling technique know as *superblock* scheduling [49]. Superblocks are similar to traces except that during the construction certain blocks are duplicated to remove all join points in the traces. First, traces are built in like fashion to trace scheduling. Second, *tail duplication* is performed to remove side entrances into the trace. During tail duplication, all the blocks between the first side entrance and the end of the trace are replicated. Then, all other side entrances into the trace are redirected into the copy. This duplication can then be repeated on the duplicate. Therefore, a superblock can have several branch points, but only one entry point. This method reduces the amount of book keeping required, and making several copies of certain basic blocks can lead to greater opportunities for finding instruction level parallelism. However, the duplication of whole basic blocks can lead to substantial code growth.

Several other techniques that benefit from code replication or code growth have been used. These include Bernstein and Rodeh's "Global Instruction Scheduling" [8, 6], Ebcioglu and Nakatani's "Enhanced Percolation Scheduling" [32], Gupta and Soffa's "Region Scheduling"[44], and the work by Golumbic and Ranish for scheduling on the RS6000 [41].

These methods present a major change in the thinking behind instruction scheduling; code can be replicated in order to improve the overall execution time of the program. Unfortunately, the tradeoff for improved runtime efficiency has been growth in the static size of the executable code. In many (most) computing environments this tradeoff has been acceptable. Memory and disk space are fairly cheap, and most users are willing to sacrifice some of those resources for improved runtime performance.

There are, however, some cases where static code growth is undesirable. The work in instruction scheduling first stemmed from a desire to make microcodes as small as possible since these microcodes would be burned into the ROM's of the finished computer. As instruction scheduling found application on modern day microprocessors, the issue of small executables became less important. But today we see complex microprocessors and DSP's being used in commodity applications such as microwaves, cellular phones, and other embedded systems. Since the code for these applications is typically burned into a ROM, small code size results in a smaller ROM, saving both money and power. Additionally, small code is important for good instruction cache behavior and when code is being transmitted over a network. New processor designs and applications have once again made the focus on code size issues fashionable.

4.2 Two Non-Local techniques

In this section we look at two non-local scheduling techniques specifically designed to avoid increasing code size, namely dominator-path scheduling (DPS) [80], and extended basic block scheduling (EBBS).

4.2.1 Extended basic block scheduling

An extended basic block (or EBB) is a sequence of basic blocks, $B_1, \ldots, B_k$, such that, for $1 \leq i < k$, B_i is the only predecessor of B_{i+1} in the CFG, and B_1 does not have a unique predecessor [2]. We alter this slightly by allowing B_1 to have one or more predecessors. For scheduling purposes, we view extended basic blocks as a partitioning of the CFG; a basic block is a member of only one EBB.

The first step in EBB construction is to partition the CFG into extended basic blocks. We would like to group blocks together into extended basic blocks in such a way that the maximum amount of instruction level parallelism can be uncovered. Since we don't build dependence graphs until after we have selected which basic blocks will be scheduled together, we use a heuristic to drive the partitioning. We try to group large basic blocks (*i.e.,* blocks with large numbers of operations weighted by their latency) together into extended basic blocks. The hope is that by grouping large blocks together, we raise the probability that at least some operations can execute in parallel.

We define the set of *header* blocks to be all those blocks that will be the first block in some EBB. Initially, our set of headers consists of the start block and all blocks with more than one predecessor in the CFG. Once this initial set of headers is computed, we compute a weighted *size* for each basic block by performing a postorder walk over the CFG. The size for a header is set to zero. The size for all other blocks equals the total number of operations in the block weighted by their latencies plus the maximum size of all the block's successors in the CFG.

To construct the EBB's, we maintain a work-list of header blocks. When a block B is pulled off the work-list, other blocks are added to its EBB based on the sizes computed earlier. The successor of B in the CFG with the largest size is added to B's EBB. The other successors of B are added to to the work-list to become headers for some other EBB. This process continues for the block just added, until no more eligible blocks are found for the current EBB. For each EBB, a DPG is constructed, and the EBB is scheduled with LS.

We use the control flow graph of Figure 4.1 to demonstrate the construction. Initially the two hashed blocks (B_1 and B_5) are the header blocks. The field *ops* in each block shows the number of operations (weighted by latency) in the block. The *size* field is as described above. Notice the *size* of the header blocks is 0. The size of B_3 is $ops(B_3)$ plus the size of B_4. When B_1 is pulled off the header worklist, the size of B_2 and B_3 are compared. Since the size of B_3 is greater, it is added to the EBB headed by B_1. B_2 is added to the header worklist, to be the start block of some other EBB. Thus, in the final partitioning, there are three EBB's: (B_1, B_3, B_4), (B_2), and (B_5).

We must prohibit some operations from moving between the blocks of an EBB. This is done using *liveness* information. The set of values $livein(B)$ are those values that are live on entry into block B. That is, for each value $v \in livein(B)$ there exists some path in the CFG starting at the entry of B such that v is used before being defined. Likewise the set $liveout(B)$ are those values that are live after all operations in B have completed.

Assume a block B_1 has successors $B_2, B_3, \ldots, B_n$ in the CFG. Further assume that B_j is placed in the same EBB as B_1. We prohibit moving an operation from B_j into B_1, and vice versa, if that operation defines a value that is live along some path from B_1 to B_i where $i \neq j$. We call this set of values *path-live* with respect to B_j, or PL_{B_j}. The set is a subset of the set $liveout(B_1)$ as computed by the following equation:

$$PL_{B_j} = \bigcup_{B_i \in succ(B_1) \wedge B_i \neq B_j} livein(B_i)$$

Intuitively, we can't move the operation between B_1 and B_j if any value it defines is used in some block other than B_1 or B_j and that block is reachable from B_1 via some path not containing B_j. The operations that can be moved are called *partially dead* if they are in B_1 [53].

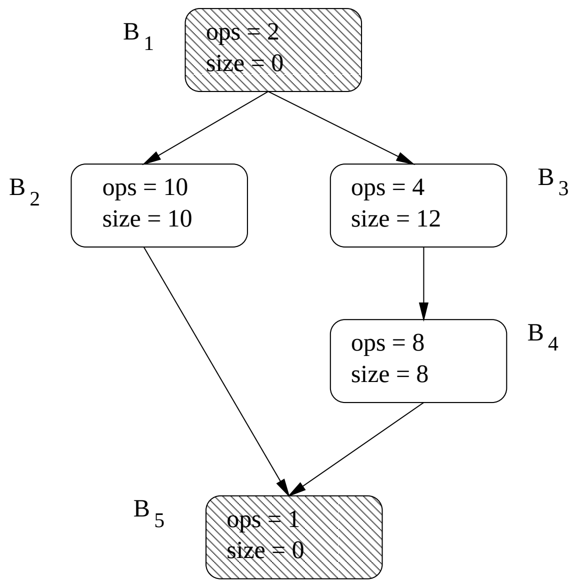


Figure 4.1 EBB Construction Example

Little work has been published on scheduling over extended basic blocks. Freudenberger, *et. al.* show some results of scheduling over extended basic blocks, but only after doing some amount of loop unrolling [37]. Since we are interested in what improvements are possible with no code growth, such loop unrolling is out of the question.

4.2.2 Dominator-path scheduling

Now we explore another non-local scheduling technique, dominator-path scheduling. DPS was originally described in Sweany's thesis [78]. Other work on DPS was done by Sweany and Beaty [80], and Huber [48].

We say a basic block B_1 *dominates* block B_2 if all paths from the start block of the CFG to B_2 must pass through B_1 [71]. If B_1 dominates B_2, and block B_2 executes on a given program run, then B_1 must also execute. We define the *immediate dominator* of a block B (or $idom(B)$) to be the dominator closest to B in the CFG. Each block must have a unique immediate dominator, except the start block which has no dominator. Let $G = (N, E)$ be a directed graph, where the set N is the set of basic blocks in the program, and define $E = \{(u, v) \mid u = idom(v)\}$. Since each block has a unique immediate dominator, this graph is a tree, called the *dominator-tree*. A dominator-

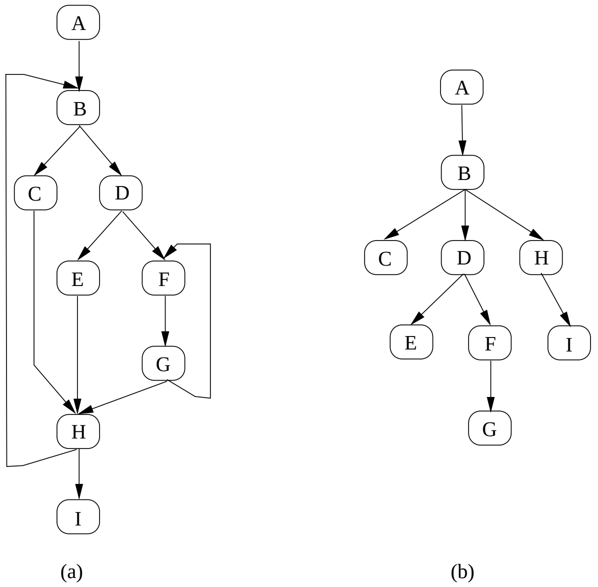


Figure 4.2 A CFG and its corresponding dominator tree

path is any path between two nodes of the dominator-tree. The control flow graph in part (a) of Figure 4.2 shows a CFG, and part (b) shows the corresponding dominator tree.

We now define two sets, $idef(B)$ and $iuse(B)$. For a basic block B, $idef(B)$ is the set of all values that may be defined on some path from $idom(B)$ to B (not including B or $idom(B)$.) Likewise $iuse(B)$, is the set of all values that may be used on some path from $idom(B)$ to B. The algorithm for efficiently computing these sets is given by Reif and Tarjan [81].

DPS schedules a dominator-path as if it were a single basic block. First, the blocks in the CFG must be partitioned into different dominator-paths. Huber describes several heuristics for doing path selection and reports on their relative success. We use a size heuristic similar to the one described above for EBBS. This is done via a bottom-up walk over the dominator-tree. The size of a leaf equals the latency-weighted number of operations in the block. For all other blocks, size equals the latency-weighted number of operations in the block plus the maximum size of all the block's children in the dominator-tree. This allows us to find the dominator-path with the largest number of latency weighted operations. When building the dominator-paths, we select the next block in the path by choosing the child in the dominator-tree with the largest size. All other children become the first block in some other dominator-path. Once the dominator-paths are selected, a DPG is created for each path, and the path is scheduled using LS. After each path is scheduled, liveness information and the $idef$ and $iuse$ sets must be recomputed to insure correctness. (See next section for an explanation.)

When the compiler builds the DPG for the dominator-path, it adds edges to prevent motion of certain operations between basic blocks. Assume B_1 is the immediate dominator of B_2. Sweany's original formulation prohibited moving an operation from B_2 up into B_1 if that operation defined a value in $idef(B_2) \cup iuse(B_2)$, or if it referenced a value in $idef(B_2)$. That is, we don't want to move an operation that defines a value V above a use or definition of V in the CFG. Likewise, an operation that references V is not allowed to move above a definition of V. This strategy is safe when B_1 dominates B_2 *and* B_2 post-dominates B_1 [5].

However, Huber showed that in the general case this strategy is unsafe. Figure 4.3-(a) illustrates the problem. In this simple CFG we show only the operations that use or define `r1`. Assume blocks B_1 and B_2 will be scheduled together. Clearly `r1` is not a member of $iuse(B_2)$ or $idef(B_2)$ since there are no blocks between B_1 and B_2. Thus, the definition of `r1` in B_2 could be unsafely hoisted into block B_1 causing the use of `r1` in B_4 to use the wrong value. Thus, Huber adds the restriction that any operation that defines a value in

$$idef(B_2) \cup iuse(B_2) \cup (liveout(B_1) - livein(B_2))$$

cannot be moved up from B_2 into B_1.

However, we have found that this formulation too is unsafe. Figure 4.3(b) demonstrates the problem. Again, assume that blocks B_1 and B_2 will be scheduled together. Note that $r1 \in liveout(B_1)$ and $r1 \in livein(B_2)$ since it is referenced before it is defined in B_2. Therefore, $r1$ is not in the set $liveout(B_1) - livein(B_2)$. Assuming operation 2 does not define anything that causes movement to be unsafe, we can move it up into block B_1. It would then be legal to move operation 3 into B_1. Thus both operations in B_2 could be moved into block B_1, which would cause operation 4 in block B_4 to potentially get the wrong value. Once a dominator-path is selected for scheduling, no updating of the liveness information is done during the scheduling of that dominator-path. Some sort of incremental update would be one way to solve

[5] A block B_2 post-dominates B_1 if every path from B_1 to the end of the CFG must pass through B_2 (simply the dominance relation in reverse.)

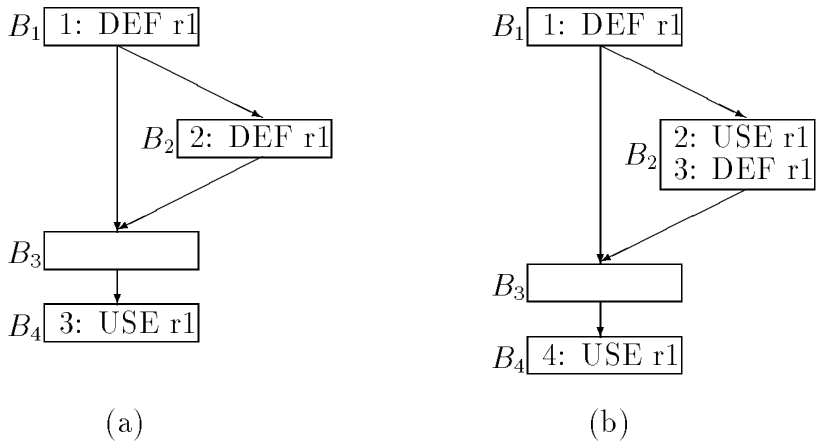


Figure 4.3 DPS example

this problem, since moving operation 2 into B_1 would cause $r1$'s removal from the set $livein(B_2)$.

We use an approach that doesn't require incremental updates. What we really want to capture are those values that are live along paths other than paths from B_1 to B_2. This is fairly straightforward if B_1 is the only parent of B_2 in the CFG; we simply use the path-live notion discussed in the previous section. In other cases, we take the conservative approach and disallow any operation that defines a value in $liveout(B_1)$ to move up.

Now we consider motion of an operation in the downward direction. Sweany does not allow an operation to move down the CFG, that is into block B_2 from its dominator B_1, but he does mention that this could be done if B_2 *post-dominates* B_1. Huber allows operations to move down into the post-dominator if they don't define anything in $idef(B_2) \cup iuse(B_2)$ or use anything in $idef(B_2)$. No downward motion is allowed if B_2 does not post-dominate B_1. We take this one step further by allowing motion of an operation from B_1 into B_2 if B_1 is the predecessor of B_2 in the CFG, and the operation computes values that are only live along the edge (B_1, B_2) (again the path-live notion from section 4.2.1.) In any other case where B_2 does not post-dominate B_1, we take the conservative approach and disallow any motion of operations that compute a value in $liveout(B_1)$.

Loops pose additional concerns. We must be careful not to allow any code that defines memory to move outside of its current loop or to a different loop nesting

$dontdef = idef(B_2) \cup iuse(B_2)$
if B_2 does not post-dominate B_1
    then if B_1 is the predecessor of B_2 in CFG
        $dontdef = dontdef \cup PL_{B_2}$
    else
        $dontdef = dontdef \cup liveout(B_1)$
if B_2 and B_1 are in different loops
    then $dontdef = dontdef \cup liveout(B_1)$
        $dontdef = dontdef \cup memory\_values$

Figure 4.4 Summary of prohibited moves between B_1 and B_2

depth.[6] In addition to the restrictions described above, we disallow any operation that defines memory from moving between two blocks if they are in different loops or at different loop nesting levels. Finally, we don't allow an operation that defines registers in $liveout(B_1)$ to move between the two blocks.

To summarize, we disallow motion of an operation between block B_2 and its immediate dominator B_1 (forward or backward) if that operation defines a value in the set $dontdef$. This set is defined in figure 4.4. Additionally any operations that use a value in $idef(B_2)$ are not allowed to move.

4.2.3 An additional caveat

One should wonder whether the data-flow analysis used by the two schedulers needs to be updated during scheduling. In the case of DPS the answer is yes.

Figure 4.5 illustrates the problem. The original code is shown on the left. Assume that blocks B_1 and B_3 will be scheduled together and blocks B_2 and B_4 will be scheduled together. There are three paths to the use of `r1`: (B_1, B_3, B_5), (B_1, B_2, B_3, B_5), and (B_1, B_2, B_4, B_5). With no update of the liveness information or $idef$ set, the incorrect code shown on the right side of the figure could result. In this case the path (B_1, B_2, B_3, B_5) results in the use of `r1` coming from assignment 2 instead of assignment 1 as in the original program.

How can we arrive at this incorrect solution? There are two ways. Assume that the dominator-path (B_1, B_3) is scheduled first, and operation 1 moves from B_3 into B_1.

[6] Recall that scheduling follows optimization. The optimization should include some careful code motion [51].

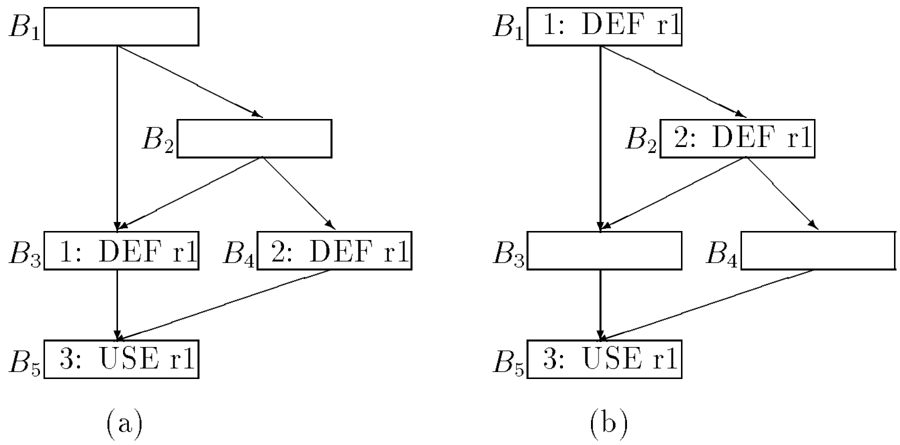


Figure 4.5 DPS example showing need to recompute liveness information and *idef* and *iuse* sets.

Now the value `r1` is in liveout of B_2 since there is a path where `r1` is used before being defined. However, without updating the liveness information, operation 2 could the be moved from B_4 into B_2. Now consider the situation if the dominator path (B_2, B_4) is scheduled first, and operation 2 moves from B_4 into B_2. Now `r1` is defined along some path between B_3 and its immediate dominator B_1, and thus should be added to the set $idef(B_3)$. Not doing so allows operation 1 to move up into B_1, and we get the same situation as shown in Figure 4.5 (b). Because of this problem, liveness information and the *idef* and *iuse* sets are recomputed after scheduling a dominator path of length greater than one.

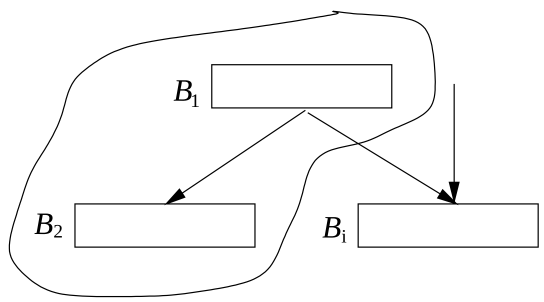


Figure 4.6 Example EBB

Notice that for EBBS the above situation can never occur. Updates of liveness information are not needed when scheduling over EBB's. When operations are moved inside an EBB the liveness sets inside other EBB's are not affected. The CFG fragment in Figure 4.6 will be used in the following discussion. B_1 and B_2 are basic blocks in the same EBB and B_i represents a block in some other EBB. It suffices to show that legal movement of operations between B_1 and B_2 doesn't affect the set $liveout(B_i)$.

There are several cases to consider. First consider a definition of `r1` moving from B_2 up into B_1. For this to be legal it must hold that $r1 \notin livein(B_i)$. Assuming no definitions or uses of `r1` in B_i, $r1 \notin liveout(B_i)$. (If $r1 \in liveout(B_i)$ the point is moot, since `r1` must be redefined in B_i.) Since there can be no path from the EBB which contains B_i into B_2, it is impossible for `r1` to be in $liveout(B_i)$. The same argument holds for a definition moving down the CFG from B_1 into B_2. For this to be legal it must hold that $r1 \notin livein(B_i)$, and moving the definition down can't affect $livein(B_i)$ by definition of live. Uses of values moving between B_1 and B_2 can not affect $livein(B_i)$ or $liveout(B_i)$ since there is no path from B_i into B_2 (again by definition of EBB).

Clearly, there is a computational advantage to using EBBS over DPS as the data-flow information doesn't need to be constantly recomputed. This advantage is borne out by the results in the next section.

4.3 Experimental Results

In this section we present some experimental results. EBBS and DPS are compared to scheduling over basic blocks on our various architectural models. For these experiments we allow pointer based loads (ILOC's `PLDor` operation) to move freely between basic blocks. This can of course be dangerous because a load through a pointer to an uninitialized memory location may move above a null pointer test causing a segmentation violation. This problem has been alleviated by changing the ILOC to force certain problematic `PLDor` operations to remain in their original basic block. This transformation was done by hand and would require additional program analysis to be done automatically. Results of these techniques (for some benchmarks) when `PLDor`'s are not allowed to move to higher basic blocks are presented by Cooper and Schielke [21].

After scheduling we run a restricted form of the `clean` optimization that removes empty basic blocks from the code. During scheduling it is possible for all operations but the control flow operation to be moved out of a basic block. These blocks can be removed if doing so doesn't violate any timing constraints between other operations.

4.3.1 SVLIW architecture

Table 4.1 shows the dynamic instruction counts for our benchmark codes on the SVLIW architecture. The smallest dynamic instruction count is underlined indicating which technique produced the largest improvement for that benchmark. Both EBBS and DPS resulted in faster code than basic block scheduling. Slightly better than fifty percent of the time DPS outperformed EBBS, and a few of these wins were substantial. On average, EBBS produced a 6.5 percent reduction in the number of dynamic instructions executed, and DPS produced a 7.9 percent reduction.

Table 4.2 shows the static instruction counts for the same experiments. Note that all the object codes have the same number of operations; only the number of instructions changes. DPS did better by this metric in roughly the same number of experiments. However, the static and dynamic improvements did not necessarily occur on the same codes. This demonstrates that smaller more compact code does not always result in enhanced runtime performance. On average EBBS reduced static code size by 11.8 percent and DPS by 12.7 percent.

Another interesting issue is borne out by the `tomcatv` benchmark. EBBS gained virtually no improvement in dynamic instruction count, but a 4.3% reduction in static code size. There are two possible explanations for this. Either the regions of the program for which shorter static schedules were found execute infrequently, or the shorter static schedules resulted in shortening less frequently executed blocks while lengthening more frequently executed blocks. The next chapter goes into this caveat in more detail.

When performing basic block scheduling, we found each block had an average of 6.5 operations (over all benchmarks). On average, an EBB consisted of 1.8 basic blocks and 11.6 operations. Dominator paths averaged 2.1 basic blocks and 14.0 operations, each. On average dominator paths are larger but the tradeoff is that more edges must be added to the DPG to prevent illegal code motions. Sometimes the increased size of the scheduling scope of DPS pays off dramatically (*e.g.* `adpcm`, `fmin`, `doduc`, `tomcatv`). However, this is not always the case (*e.g.* `dfa`).

We also measured the amount of time required to schedule. The scheduling times for each benchmark are shown in Table 4.3. The scheduling time for all benchmarks

	Basic Block	EBBS		DPS	
Benchmark	Insts	Insts	% dec.	Insts	% dec.
`adpcm`	13625959	12604412	7.5	<u>10874359</u>	20.2
`clean`	4515619	3769537	16.5	<u>3635091</u>	19.5
`compress`	10641307	9625986	9.5	<u>9602440</u>	9.8
`dfa`	696450	<u>594859</u>	14.6	634611	8.9
`dhrystone`	3660102	3350091	8.5	<u>3230091</u>	11.8
`fft`	22469970	<u>22150570</u>	1.4	22199415	1.2
`go`	589209782	501244191	14.9	<u>490480041</u>	16.8
`gzip`	1941439	<u>1752377</u>	9.7	1807075	6.9
`jpeg`	45900780	<u>42529297</u>	7.4	43051578	6.2
`nsieve`	2288889385	<u>2254242028</u>	1.5	2254242034	1.5
`shorten`	2166038	2076582	4.1	<u>1996296</u>	7.8
`water`	36111497	33651055	6.8	<u>33058154</u>	8.5
`fmin`	5370	4495	16.3	<u>4114</u>	23.4
`rkf45`	818884	761549	7.0	<u>749877</u>	8.4
`seval`	3340	3283	1.1	<u>3281</u>	1.8
`solve`	2813	2671	5.1	<u>2642</u>	6.1
`svd`	14649	<u>13963</u>	4.7	14053	4.1
`urand`	1117	<u>1081</u>	3.2	1093	2.2
`zeroin`	4603	4095	11.0	<u>4043</u>	12.2
`applu`	884028559	<u>866954001</u>	1.9	867750207	1.8
`cg`	1432716153	<u>1431526586</u>	0.1	1431532626	0.1
`doduc`	16953587	16141047	4.8	<u>15264246</u>	10.0
`fpppp`	95701038	90646630	5.3	<u>89892306</u>	6.1
`matrix300`	43073238	<u>42883523</u>	0.4	42885923	0.4
`mg`	87661635	84067412	4.1	<u>84066684</u>	4.1
`tomcatv`	436717483	436707058	0.0	<u>408095938</u>	6.6

Table 4.1 Dynamic Instruction Counts for SVLIW, EBBS vs. DPS

	Basic Block	EBBS		DPS	
Benchmark	Insts	Insts	% dec.	Insts	% dec.
`adpcm`	343	311	9.3	294	14.3
`clean`	11429	9973	12.7	9965	12.8
`compress`	1595	1374	13.9	1379	13.5
`dfa`	1356	1000	26.3	1040	23.3
`dhrystone`	525	471	10.3	459	12.6
`fft`	2708	2514	7.2	2492	8.0
`go`	73456	54010	26.5	52240	28.9
`gzip`	12217	10008	18.1	10090	17.4
`jpeg`	19785	17484	11.6	17843	9.8
`nsieve`	271	249	8.1	247	8.9
`shorten`	7819	6823	12.7	6800	13.0
`water`	6469	6039	6.6	5822	10.0
`fmin`	712	503	29.4	447	37.2
`rkf45`	2387	2055	13.9	2030	15.0
`seval`	1055	993	5.9	1012	4.1
`solve`	1007	935	7.1	928	7.8
`svd`	2481	2234	10.0	2268	8.6
`urand`	192	172	10.4	168	12.5
`zeroin`	545	446	18.2	443	18.7
`applu`	13334	12933	3.0	12851	3.6
`cg`	946	882	6.8	893	5.6
`doduc`	42127	38536	8.5	37394	11.2
`fpppp`	10521	9796	6.9	9662	8.2
`matrix300`	429	361	15.9	367	14.5
`mg`	2108	2040	3.2	2041	3.2
`tomcatv`	953	912	4.3	887	6.9

Table 4.2 Static Instruction Counts for SVLIW, EBBS vs. DPS

Benchmark	BB	EBBS	DPS
adpcm	0.03	0.04	0.04
clean	0.91	1.30	6.82
compress	0.06	0.08	0.38
dfa	0.04	0.07	1.15
dhrystone	0.05	0.03	0.07
fft	0.09	0.14	1.05
go	4.83	7.96	431.51
gzip	0.68	1.11	8.8
jpeg	1.93	2.68	7.08
nsieve	0.02	0.01	0.02
shorten	0.46	0.61	8.63
water	0.23	0.28	0.48
fmin	0.02	0.03	0.03
rkf45	0.06	0.06	0.17
seval	0.01	0.02	0.02
solve	0.04	0.04	0.07
svd	0.04	0.08	0.42
urand	0.01	0.02	0.01
zeroin	0.01	0.01	0.02
applu	0.55	0.70	2.46
cg	0.02	0.03	0.08
doduc	1.28	1.66	11.64
fpppp	35.67	32.75	38.76
matrix300	0.02	0.03	0.03
mg	0.08	0.09	0.23
tomcatv	0.02	0.06	0.08

Table 4.3 Scheduling times in seconds

was 47 seconds for basic block scheduling, 50 seconds for EBBS, and 520 seconds for DPS. This comparison is a bit unfair. Several of our C codes have many functions in each ILOC module. Thus DPS is performing the dominator analysis for the whole file every time a dominator-path containing more than one basic block is scheduled. The go benchmark contributed over 400 seconds alone, due to large modules containing many ILOC functions. Some sort of incremental updating of *idom* and *iuse* sets would help make DPS more competitive from a compile-time perspective.

4.3.2 C60 Architecture

Table 4.4 shows the dynamic instruction counts for our C60 architecture model. Static instruction counts (*i.e.,* fetch packet counts) are reported in table 4.5. In dynamic instruction counts, we see improvements over basic block scheduling similar to those seen for the other architectures. On average, EBBS showed a 4.6 percent improvement over basic block scheduling, and DPS a 5.8 percent improvement. DPS outperformed EBBS on fifteen of the benchmarks. Since roughly the same number of control-flow operations are executed regardless of the scheduling technique used, these percentages would improve if the architecture could take advantage of branch prediction information.

However, static code sizes didn't show the improvements seen when targeting the SVLIW architecture. In fact, the code size increased on all benchmarks (indicated by the negative percentages) an average of 3.4 percent for EBBS and 2.9 percent for DPS.

These code increases are due to the compaction method employed by the architecture, as described earlier. Consider a basic block that has eight operations all packed into one instruction. If six of these operations are moved into another block, and the number of instructions in that block is increased by one, the overall length of the code will increase by one instruction. While we have not added any operations to the compiled code, the number of instructions has increased due to the code motion. This shows how effective the C60 design is at keeping object code compact. It also highlights the need for improved scheduling techniques to keep the static code size for these architectures small, while still improving runtime performance.

In terms of dynamic instruction count, on most of the benchmarks the same technique that performed the best on SVLIW, also performed the best on C60. This suggests that most of the time the EBB and DP building algorithms are selecting the same basic blocks regardless of architecture. It also suggests that the selection of

	Basic Block	EBBS		DPS	
Benchmark	Insts	Insts	% dec.	Insts	% dec.
adpcm	25869274	25027266	3.3	23717774	8.3
clean	7121253	6484949	8.9	6374311	10.5
compress	6451355	5967052	7.5	5969723	7.5
dfa	1234290	1165659	5.6	1196470	3.1
dhrystone	6310417	6040417	4.3	5920418	6.2
fft	19080610	18872854	1.1	18866424	1.1
go	873554813	800500402	8.4	794506738	9.0
gzip	3630649	3463878	4.6	3468937	4.5
jpeg	59715530	56913698	4.7	57755148	3.3
nsieve	3810982127	3793661381	0.5	3793661381	0.5
shorten	4014951	3939604	1.9	3874954	3.5
water	37219738	34985459	6.0	34400785	7.6
fmin	4363	3852	11.7	3604	17.4
rkf45	622271	578933	7.0	565012	9.2
seval	2508	2454	2.2	2454	2.2
solve	2820	2724	3.4	2697	4.4
svd	12713	12151	4.4	12268	3.5
urand	1504	1487	1.1	1494	0.7
zeroin	3710	3388	8.7	3300	11.1
applu	676646005	660381369	2.4	660639674	2.4
cg	1115328725	1114112782	0.1	1114120191	0.1
doduc	13506556	12718250	5.8	12143306	10.1
fpppp	57176443	49874088	12.8	49776879	12.9
matrix300	33652694	33403762	0.7	33404562	0.7
mg	56027886	54673420	2.4	54410177	2.9
tomcatv	332355068	332344654	0.0	306334614	7.8

Table 4.4 Dynamic Instruction Counts for C60, EBBS vs DPS

	Basic Block	EBBS		DPS	
Benchmark	Insts	Insts	% dec.	Insts	% dec.
adpcm	98	100	-1.9	99	-0.9
clean	1909	1977	-3.5	1966	-2.9
compress	302	309	-2.2	307	-1.6
dfa	344	356	-3.4	352	-2.2
dhrystone	108	111	-2.7	113	-4.5
fft	360	374	-3.8	368	-2.1
go	12669	13462	-6.2	13440	-6.0
gzip	2516	2682	-6.5	2644	-5.0
jpeg	3003	3059	-1.8	3059	-1.8
nsieve	49	51	-4.0	52	-6.0
shorten	1615	1650	-2.1	1649	-2.0
water	731	743	-1.5	741	-1.3
fmin	61	65	-6.5	64	-4.8
rkf45	186	189	-1.5	187	-0.4
seval	80	83	-3.7	83	-3.7
solve	118	121	-2.4	122	-3.3
svd	239	245	-2.4	243	-1.6
urand	23	24	-4.2	23	0
zeroin	40	42	-4.9	43	-7.4
applu	1868	1898	-1.5	1889	-1.0
cg	150	154	-2.6	152	-1.2
doduc	3764	3992	-6.0	3894	-3.4
fpppp	1551	1561	-0.5	1564	-0.7
matrix300	77	81	-5.1	81	-5.1
mg	373	380	-1.8	380	-1.8
tomcatv	125	131	-4.7	132	-5.5

Table 4.5 Static Instruction Counts for C60, EBBS vs DPS

which technique to use is fairly insensitive to architecture, although differences in operation latencies and in functional units will change the results slightly. Also see the results in the next section for the PPC750.

4.3.3 PPC750 Architecture

Table 4.6 reports dynamic instruction counts when scheduling on the PPC750 using EBBS and DPS. Static results are not reported since the PPC750 does not require implicit NOP's to be inserted. The numbers look comparable to other architectures except for a substantial increase in running time for `fpppp`. Once again the careless movement of code into more frequently executed blocks is to blame.

4.4 EBB Construction Heuristics

Various heuristics can be used select which blocks belong to which extended basic block or dominator path. Huber surveys and compares several strategies for construction of dominator paths. We do not duplicate his work here, but rather retarget some of his techniques toward building extended basic blocks for scheduling purposes. Some of Huber's other heuristics involving dominators and post-dominators are not applicable to extended basic blocks.

4.4.1 Operation Density

Huber defines *operation density* (which he calls instruction density) as the number of operations in the block prior to scheduling divided by the length of the blocks critical path. The *critical path* is the length of the shortest possible schedule for the block given the precedence constraints imposed by the DPG. Thus the unit of operation density is operations per cycle. Intuitively, the higher the density of a block the fewer "gaps" there are for operations in other blocks to occupy.

The notion of operation density underlies several of Huber's heuristics. For example we could attempt to group together blocks with low density, or try to alternate high and low density blocks within an EBB. We describe three heuristics for building EBB's based on density.

First we need to determine a threshold to define high and low density. This is done by computing the density for all basic blocks and finding the average. We classify any block having a density value below the average as low density and any block with density above the average as high density.

	Basic Block	EBBS		DPS	
Benchmark	Insts	Insts	% dec.	Insts	% dec.
adpcm	10099568	9458460	6.3	8498968	15.8
clean	3135572	2704522	13.7	2579861	17.7
compress	6330836	5387476	14.9	5379336	15.0
dfa	456358	395923	13.2	412917	9.5
dhrystone	2730101	2560101	6.2	2510101	8.1
fft	10061265	9818240	2.4	9810010	2.5
go	363430164	320719566	11.8	313583874	13.7
gzip	1504428	1400111	6.9	1386398	7.8
jpeg	32992742	31223308	5.4	31392039	4.9
nsieve	1510981283	1493660543	1.1	1493660551	1.1
shorten	1533947	1491475	2.8	1426897	7.0
water	16911814	16468732	2.6	15582108	7.9
fmin	3839	3684	4.0	3519	8.3
rkf45	438503	408322	6.9	406624	7.3
seval	2655	2621	1.3	2607	1.8
solve	1570	1477	5.9	1461	6.9
svd	7998	7590	5.1	7648	4.4
urand	1031	1024	0.7	986	4.4
zeroin	3772	3516	6.8	3457	8.4
applu	455904964	447032669	1.9	447464221	1.9
cg	522124378	520954400	0.2	520959810	0.2
doduc	15254873	14832434	2.8	14382112	5.7
fpppp	94083373	102496332	-8.8	99003037	-5.1
matrix300	14675862	14550178	0.9	14549778	0.9
mg	47926868	44332727	7.5	44070213	8.0
tomcatv	266181287	266170802	0.0	250564822	5.9

Table 4.6 Dynamic Instruction Counts for PPC750, EBBS vs DPS

The first heuristic we call *same density.* It works by trying to group blocks with low density together, the intuition being that in a series of blocks with low densities there is the most freedom of movement for operations between blocks because of the many "holes" in the schedule of any single basic block. To perform the heuristic, each header block is determined to be of high or low density. If it is of high density, the current EBB is constructed by always selecting successor blocks with the highest density. Likewise if the header was of low density, we always select the successor with the lowest density.

Our second heuristic we call *alternate density.* In this heuristic we try to alternate blocks of different densities into the same EBB. If the previous block in the EBB is of high density then we select the next block in the EBB to be of low density (if one is available) and vice versa.

As described, the density metric doesn't really reflect the size of a block. That is a block could be very dense but have few operations. Our third and fourth metrics mirror the first two, except that the density for each block is multiplied by the number of operations in the block. This value has units operations squared per cycle. The two heuristics are called *same density 2* and *alternate density 2*.

4.4.2 Random selection

In order to get a feel for whether the EBB building heuristics are at all effective, we tried building EBB's randomly. That is, the selection of which predecessor of a block to add to the EBB is done randomly.

4.4.3 Size

Our original EBB building heuristic, which we refer to as the *size* heuristic, tried to group blocks together that had large numbers of operations weighted by latency. The heuristic *size2* works the same as *size*, except that it does not weight operations by their latency. That is, it tries to group together blocks with large numbers of operations irrespective of the latency of those operations.

4.4.4 Experimental Results

In this section we present results for the various EBB construction heuristics described above when targeting the SVLIW architecture. The five techniques are compared to the *size* heuristic used in the previous experiments. The tables present the percentage reduction in dynamic instruction count (Table 4.7) and static instruction count (Table 4.8) over the *size* heuristic. As before, a negative percentage indicates an increase in dynamic or static instruction count. The heuristics are abbreviated *sd* for *same density*, *ad* for *alternate density*, *sd2* for *same density 2*, *ad2* for *alternate density 2*, and *ran* for *random*. For each benchmark we underline the best result.

Over all the benchmarks the original *size* heuristic performed best. However, there were several individual benchmarks for which a different heuristic outperformed *size*. The biggest win in terms of dynamic instruction count was on `shorten` where *same density* reduced the running time by 3.6 percent. In terms of static instruction count, the biggest win came on `dhrystone` with a reduction of .64 percent using *same density* and *random*.

While the heuristic used to build EBB's clearly affects individual benchmarks, the data do not clearly indicate which heuristic is best overall. Interestingly the *random* heuristic performed better on average than three of the heuristics tested. We have no explanation for this result. The data suggests that a compiler would do well to try two or three heuristics at compile time, and select the best result based on some metric (*e.g.* static instruction count). This conclusion is similar to what Huber found for dominator path selection heuristics.

At an intuitive level scheduling, low density and high density blocks together makes sense, but as the data shows it doesn't always help in practice. The only way that scheduling more than one block at the same time can be beneficial, is if there are few dependences between operations in those blocks. There is little to be gained from scheduling two blocks together, regardless of density or any other metric, if there exist a large number of dependences between them. Dependences limit the possibility of filling the gaps in one block's schedule with operations from another. In our system, we build the dependence graph long after we group blocks together into EBB's. Since we have no dependence information available when building EBB's, the *size* heuristic makes sense. By putting as many operations into a single EBB as possible, we increase the probability that at least some operations won't interfere with others. That is, given a block B, the more operations we attempt to schedule with B, the higher the probability that some operation will fill the gaps in B's schedule. This allows compaction when the blocks are scheduled together. One could conceive of more sophisticated block selection heuristics that took dependence information into account. However, it is not clear that doing so would be more compile time efficient than a simpler best of two or best of three approach.

4.5 Summary of work on EBBS and DPS

In this chapter we have explored EBBS and DPS, two nonlocal scheduling techniques. We believe our formulation of EBBS to be novel. Additionally, we corrected a slight flaw in DPS, a previously described technique. In general, EBBS is the more straightforward of the two approaches, and cheaper in terms of compile time, while the large scheduling scopes created by DPS may lead to better overall improvements. This last point, however, is unclear. Various EBB construction heuristics were explored, with no clear winner. The *size* heuristic we describe appears to work quite well in practice.

Benchmark	*sd*	*ad*	*sd2*	*ad2*	*size2*	*ran*
`adpcm`	0.27	-4.85	-3.98	-0.59	0	<u>0.39</u>
`clean`	-3.5	-0.2	-2.41	-1.5	<u>-0.00</u>	-0.98
`compress`	-8.95	-0.72	-0.51	-9.33	<u>0</u>	-9.79
`dfa`	-4.23	-0.26	-0.06	-4.52	<u>0</u>	-0.00
`dhrystone`	-2.97	<u>1.19</u>	-0.89	-2.08	0	-2.97
`fft`	-0.00	0.03	-0.08	<u>0.12</u>	0	<u>0.12</u>
`go`	-3.48	-3.59	-3.48	-2.31	<u>0.02</u>	-1.35
`gzip`	-0.35	-2.69	-2.29	-0.77	<u>0.00</u>	-1.07
`jpeg`	-3.3	-1.8	-3.19	-1.86	<u>-0.00</u>	-2.19
`nsieve`	<u>0</u>	-0.00	-0.00	<u>0</u>	<u>0</u>	-0.00
`shorten`	<u>3.62</u>	-0.19	3.55	-0.11	0	3.4
`water`	-0.01	-0.78	<u>-0.00</u>	-0.78	-1.16	-0.01
`fmin`	<u>0</u>	-16.27	<u>0</u>	-16.27	<u>0</u>	<u>0</u>
`rkf45`	<u>0.48</u>	-6.16	-0.3	-5.53	0	0.44
`seval`	0	<u>0.03</u>	-0.02	-0.51	0	-0.6
`solve`	-0.21	-1.04	-0.51	-0.81	<u>0</u>	-1.08
`svd`	-0.61	-0.61	-1.22	-0.77	<u>0</u>	-1.32
`urand`	<u>0</u>	-0.73	-0.73	<u>0</u>	<u>0</u>	<u>0</u>
`zeroin`	<u>0.22</u>	-0.94	<u>0.22</u>	-0.94	-0.94	<u>0.22</u>
`applu`	-0.96	-0.73	-1.73	<u>0.04</u>	0	-1.67
`cg`	-0.00	-0.00	-0.00	-0.00	<u>0</u>	-0.07
`doduc`	-1.97	-0.46	-1.43	-1.06	-0.00	-1.21
`fpppp`	-3.45	-0.49	-4.83	<u>1.35</u>	-0.00	0.98
`matrix300`	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0	-0.00
`mg`	-4.26	-0.00	-0.00	-4.25	<u>0</u>	-4.26
`tomcatv`	-0.00	<u>1.19</u>	1.18	0.00	0	1.18
Average	-1.29	-1.54	-0.87	-2.01	<u>-0.08</u>	-0.84

Table 4.7 EBB construction heuristics vs. *size* heuristic. Dynamic instruction reduction on SVLIW.

Benchmark	*sd*	*ad*	*sd2*	*ad2*	*size2*	*ran*
`adpcm`	0.32	-6.42	-2.24	-3.85	0	-2.56
`clean`	-0.81	-2.13	-1.67	-1.22	0.03	-1.06
`compress`	-2.03	-2.03	-2.46	-2.03	0.07	-1.74
`dfa`	-0.89	-2.59	-0.29	-3.89	0	-0.39
`dhrystone`	0.64	-1.48	-1.05	-2.54	0	0.64
`fft`	0	-0.31	-0.43	0.12	0	-0.11
`go`	-1.53	-5.03	-2.93	-2.97	-0.00	-2.25
`gzip`	-1.72	-3.58	-2.38	-2.74	0.14	-2.69
`jpeg`	-0.7	-1.56	-1.24	-1.03	-0.05	-0.99
`nsieve`	0	-0.39	-0.79	0	0	-0.39
`shorten`	-0.71	-2.17	-2.16	-1.19	0.07	-1.59
`water`	-0.29	-1.22	-0.55	-0.98	-0.11	-0.3
`fmin`	0	-21.46	0	-21.46	0	0.2
`rkf45`	-0.91	-4.86	-2.76	-5.25	0	-0.77
`seval`	0	-1.6	0	-1.6	0	-1.6
`solve`	-0.63	-0.74	-0.85	-0.63	0	-1.27
`svd`	-0.57	-0.57	-1.69	-1.29	0	-2.14
`urand`	0	-4.64	-4.64	0	0	0
`zeroin`	0	-1.56	0	-1.56	-1.56	0
`applu`	-0.53	-0.64	-0.94	-0.12	-0.01	-0.87
`cg`	-0.1	0	0.11	-0.22	0	-0.1
`doduc`	-0.93	-1.18	-1.32	-1.09	0.02	-1.26
`fpppp`	-0.6	-0.34	-1.38	-0.3	-0.04	0.14
`matrix300`	-0.27	-0.27	-0.27	0	0	-0.27
`mg`	-1.12	0	-0.33	-0.77	0	-1.12
`tomcatv`	0	0	0.11	-0.21	0	0.11
Average	-0.51	-2.57	-1.24	-2.19	-0.06	-0.86

Table 4.8 EBB construction heuristics vs. *size* heuristic. Static instruction reduction on SVLIW.

Chapter 5

Adding Randomness to List Scheduling

The basic list scheduling algorithm uses a heuristic to direct the construction of the schedule. The heuristic determines which operation the algorithm will schedule next. If the heuristic makes a poor decision, the final schedule may not be optimal. Simple heuristics often assign an integer priority to each node as described in Section 2.5. If several operations that are ready for scheduling have the same priority value, the list scheduling algorithm typically has some arbitrary basis for breaking the tie. (For example, the last node of highest priority in the ready list is selected.) This arbitrary decision could lead to a non-optimal schedule, when picking another operation of highest priority would lead to an optimal schedule.

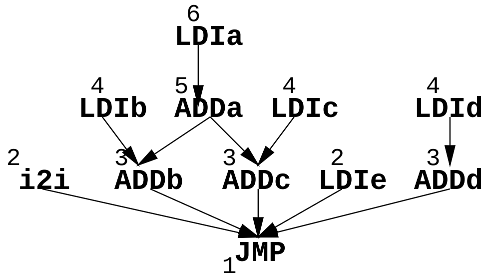
In this chapter we explore adding a degree of randomness to the list scheduling algorithm. We call this approach randomized backward and forward scheduling or RBF. This is essentially a Monte Carlo list scheduling algorithm that breaks ties in the priority assignment randomly. The next section motivates the need for random tie breaking and backward scheduling and gives some example applications. Section 5.2 describes our randomized backward and forward list scheduling algorithm, and the last section details the experimental results of applying the technique.

5.1 Motivation

In this section we motivate the need for RBF scheduling by looking at examples where random tie breaking and backward scheduling result in shorter schedules. Finally we discuss the solution space available to the RBF algorithm.

5.1.1 Random Tie Breaking

A traditional list scheduler returns a single solution by breaking any ties in the priority of two or more operations arbitrarily. (For example, when our list scheduler finds more than one operation from the ready list with highest priority, it picks the last one in the list.) By running the list scheduler several times and breaking ties randomly,



Two possible list schedules

LDIa	LDIb	LDIa	LDIb
ADDa	LDIc	ADDa	LDId
LDId	i2i	LDIc	ADDd
ADDb	ADDc	ADDb	ADDc
ADDd	LDIe	i2i	LDIe
---		JMP	
JMP			

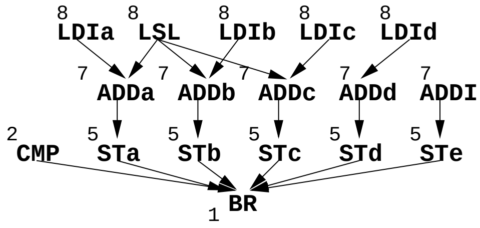
Figure 5.1 Example block from `tomcatv`

we generate more and potentially better solutions. [7] Figure 5.1 is an example from the `tomcatv` benchmark. Assume all load immediates (`LDI`) take one cycle, all add operations (`ADD`) take two cycles, and the copy (`i2i`) takes one cycle. Assume we are scheduling on a machine with two identical functional units. The numbers next to the operations are the priority values used by LS. In this figure we see two different list schedules that could be generated from the DPG. The second one requires one less cycle. The critical decision comes in the second cycle, where the tie between the `LDId` and `LDIc` must be broken. Scheduling `LDId` early enough results in a shorter schedule.

5.1.2 Backward List Scheduling

There are some blocks for which a *backward* list scheduler can generate a better solution. A backward list scheduler works by reversing the direction of all edges in DPG, and scheduling the *finish times* of each operation. (Note that we must also

[7]Some compilers may schedule several times using *different* priority heuristics. We believe our method of random tie breaking on a *single* heuristic to be novel.



Forward List			Backward List		
LDIa	LSL		LDId		
LDIb	LDIc		ADDI	LSL	
LDId	ADDa		ADDd	LDIc	
ADDb	ADDc		ADDc	LDIb	STe
ADDd	ADDI	STa	ADDb	LDIa	STd
CMP		STb	ADDa		STc
		STc			STb
		STd			STa
		STe		---	
	---			---	
	---		CMP		
	---		BR		
BR					

Figure 5.2 Example block from `go`, showing the benefits of backward list scheduling.

check the start time of each scheduled operation to ensure enough functional units are available for a given cycle.) This technique tends to cluster operations toward the end of the schedule instead of the beginning like a forward list scheduler. For an example of a block that benefits from backward list scheduling see Figure 5.2, which shows a block from the SPEC `go` benchmark. Assume there are two integer units that can execute the `LDI` operations (one cycle), the `LSL` operation (one cycle), the `ADD` operations (two cycles), `ADDI` operation (one cycle), and the `CMP` operation (one cycle). A separate memory unit executes the `ST` operations (four cycles). All functional units are completely pipelined. A forward list scheduler will schedule the four `LDI` operations and the the `LSL` before scheduling any of the `ADD` operations. This delays the start of the higher latency store operations (`ST`). A better schedule can be found by a backward list scheduler as shown in the example.

5.1.3 The List Scheduling Solution Space

As discussed in Section 2.4, instruction scheduling can be viewed as searching a solution space of possible schedules. List scheduling is a constructive approach that produces exactly one solution from the search space. Even though list scheduling produces only one schedule, there is typically a small subset of schedules from which the solution comes. This subset is imposed by the priority heuristic driving the algorithm. The schedule returned by list scheduling is one schedule out of the set of schedules imposed by the priority function, the particular schedule constructed being determined by the way the scheduler breaks ties in that function.

List scheduling does not select a schedule from the space of all possible solutions. Rather, regardless of priority function, it restricts its attention to the subset of *greedy* schedules as described by Palem and Simons [68]. The particular priority function restricts the space further to a (hopefully good) subset of the greedy schedules. The list scheduler then constructs a single solution from this restricted solution space by arbitrarily (deterministically) breaking ties in the priority function.

Since the LWD priority function has been shown to be effective, we would like to more thoroughly explore the solution space imposed by that function. It's entirely possible that the single solution produced by a list scheduler for a particular problem, may not be the best one in the restrictive solution space imposed by the priority function. The goal of RBF scheduling as described in the next section is to more thoroughly explore the solution space imposed by the priority function.

5.2 RBF scheduling

In this section we discuss our random tie breaking list scheduling technique.

5.2.1 General RBF scheduling

RBF scheduling is a modification of the traditional list scheduling algorithm. Instead of arbitrarily breaking ties in the list scheduling priority arbitrarily, RBF breaks them randomly. Thus, different runs of the scheduler on the same problem can lead to different solutions being constructed from the restricted solution space described in the previous section. RBF works by performing some number of forward list scheduling passes (breaking ties in the priority function randomly) and performing some number of backward list scheduling passes (again breaking ties randomly). The best (shortest) schedule constructed over all the runs is kept as the final schedule.

First, we must decide how many passes of list scheduling to perform on each problem. To do this we compute an estimate of the size of the restricted solution space. This is done during a preliminary pass of list scheduling. Our initial estimate of this size is unity. That is, we know there is at least one schedule in the solution space. This estimate is updated during list scheduling.

The list scheduler selects the highest priority node from the ready list (max) and assigns it to a particular functional unit (fu). If this node's priority value is different than the priority value of the last node scheduled and the current cycle being scheduled is greater than that of the last operation scheduled, we update the solution space size estimate. To update our estimate, we count the number of operations in the ready list that have the same priority as max and can execute on fu. This value represents the number of choices available to the list scheduler at this point in the scheduling process. The update computation is summarized in Figure 5.3. The notation C_r^n should be read n choose r.

Once we have the estimate of the solution space size, we compute how many passes of the random-tie-breaking list scheduler are needed to provide some coverage guarantee of the solution space. That is, we want to compute how many passes of the random-tie-breaking list scheduler are needed to find the best solution (with some high probability) in the reduced solution space. Let N be the solution space size estimate as computed above. Let d be the optimal solution density. That is, the ratio of the number of "optimal" solutions in the space, divided by the total number of solutions in the space. Optimal in this case means locally optimal (*i.e.,* the best solution in the restricted search space.) The probability of selecting an optimal solution from the space for a single pass of list scheduling is simply d. Since each list scheduling pass samples the space independently and with no replacement, the probability is the same for each pass or trial. Thus, the probability of *not* finding the optimal solution after T trials of the randomized list scheduling algorithm is $(1-d)^T$. We want to bound this probability to some small value, say ϵ. That is, $(1-d)^T = \epsilon$. Solving for T we get, $T = \frac{log(\epsilon)}{log(1-d)}$.

The situation is complicated by the fact that there is no way to know the value of d a priori. The results in the next section present some evidence for typical values of d. In the worst case $d = \frac{1}{N}$. Substituting into the above equation for T we get, $T = \frac{log(\epsilon)}{log(1-\frac{1}{N})}$. Unfortunately, N can be very large for some scheduling problems, and

Input: *max* the operation selected for scheduling
fu the functional unit selected for *max*
The *ready_list* of operations
Output: update the global variable *estimate* based on current choices

type = **fu_type**(*fu*)
ties = number of operations with priority **priority**(*max*) in *ready_list*
that can run on *type*
avail = Number of fu's of *type* available (before scheduling *max*)
if (**priority**(*max*) $\neq$ *old_priority*)
then $estimate = estimate * C_{avail}^{ties}$
old_priority = **priority**(*max*)
endif

Figure 5.3 Estimating the solution space size for RBF

keeping ϵ small would require a large number of passes. Thus we put a cap on the number of trials that can be run to bound the scheduling (and compilation) time.

Notice that as d approaches one (a solution space where every solution is optimal) T approaches zero. As long as ϵ is approximately equal to $1 - d$, T is approximately equal to 1. In general, if we know that the typical case is that d is closer to 1 than it is to $\frac{1}{N}$, fewer trials will be needed to stumble upon an optimal solution.

The RBF scheduling framework is presented in Figure 5.4. In the algorithm we assume that calls to **list_schedule**(DPG,N) perform N passes of list scheduling with random tie breaking, returning the shortest schedule found.

5.2.2 Weighting techniques for EBB and DPS

Scheduling for extended basic blocks and dominator paths is complicated by the fact that the shortest schedule for a particular problem may not result in the shortest running time. This is because blocks being scheduled together do not all have the same execution frequency.

We would like to account for this when selecting the "best" schedule in RBF. In order to better evaluate the runtime performance of a particular schedule we weight instructions in the schedule based on some estimate of the number of times they will execute. The best way to do this is to use block level profile information. Each

Input: DPG
Output: a valid schedule for DPG

compute *estimate* of forward list scheduling solution space
$T = \frac{log(\epsilon)}{log(1-\frac{1}{estimate})}$
if $(T > cap)$ **then** $T = cap$ **endif**
for_sch = **forward_list_schedule**(DPG,T)
compute *estimate* of backward list scheduling solution space
$T = \frac{log(\epsilon)}{log(1-\frac{1}{estimate})}$
if $(T > cap)$ **then** $T = cap$ **endif**
back_sch = **backward_list_schedule**(DPG,T)
return the shorter of *back_sch* and *for_sch*

Figure 5.4 RBF scheduling framework

instruction in a particular block is simply weighted by the number of times that block executes in a run of the program. If e_{max} is the maximum number of times any block in an EBB or DP is executed, then the *weight* of an instruction in block B is the number of times it executes, e_B, divided by e_{max}. The weights are summed for the entire schedule to give a weighted length.

If profile information is not available, we can use a fixed weighting scheme. The instructions in the initial block B_1 of the DP or EBB get a weight of 1. The weight for an instruction in block B_i equals $W * weight(B_{i-1})$ where $0 < W < 1$ is the weighting constant, and $weight(B_{i-1})$ is the weight of instructions in block B_{i-1}. For EBB's this weighting makes a lot of sense since block B_{i-1} will be executed no more than block B_i. For dominator paths, this formulation is problematic since some operations can move into and out of loops at different nesting levels. We expect code at higher nesting levels to be executed more frequently.

5.3 Experimental Results

In this section we present experimental results of applying the RBF technique described above. We would like to see whether RBF is able to find better solutions than the simple LS scheduling technique as described in Chapter 2. In all cases except fpppp the RBF scheduler is capped at 250 trials. For the fpppp benchmark it is capped at 50 trials due to several extremely large basic blocks in this program.

5.3.1 Basic Blocks – Performance results

In this section we present data for applying LS and RBF to basic blocks. Here we present data for the SVLIW architecture. Data for other architectural models is presented in Appendix B. Table 5.1 shows dynamic and static instruction counts for RBF and LS when targeting the SVLIW architecture. The number of RBF trials T was capped at 250 (50 for fpppp), and ϵ was set to 0.01. The columns labeled "%" show the percentage reduction found by RBF. An entry of "0" in this column means the results were identical.

Notice that the improvements are small. There are several possible reasons for this. It's possible that LS is typically finding the globally optimal solution. This possibility is explored in the next chapter. It may also be that the restricted solution space explored by RBF has a high (locally) optimal solution density, in which case LS typically finds the best solution in the restricted space. The next section explores that possibility.

5.3.2 Basic Blocks – Solution Space Sizes and Optimal Solution Densities

Table 5.2 shows solution space estimates and optimal solution densities for basic blocks for the forward random-tie-breaking list scheduling solution space. The second column in the table reports the number of scheduling problems (blocks) for the benchmark. The third column reports the average solution space size and standard deviation for the benchmark. The fourth column indicates the percentage of blocks whose solution space estimate was unity. Finally, the fifth column reports the percentage of problems whose solution space size was greater than 100. The average size of the solution space varies widely across benchmarks. Notice that the percentage of problems with only one solution in the restricted space is very large. Over all the benchmarks about 85% of the problems contained only one solution in the restricted space.

We also estimated the optimal solution density of the scheduling problem instances for each benchmark. That is, the ratio of the number of solutions found with lowest length divided by the number of trials run (in this case 500.) This result is reported in the fifth column of the table. The randomized forward list scheduler was run 500

	Dynamic Instructions			Static Instructions		
Benchmark	LS	RBF	%	LS	RBF	%
adpcm	13625959	13625959	0	343	343	0
clean	4515619	4515223	0.01	11429	11421	0.07
compress	10641307	10638494	0.03	1595	1594	0.06
dfa	696450	696450	0	1356	1356	0
dhrystone	3660102	3660102	0	525	525	0
fft	22469970	22408520	0.27	2708	2705	0.11
go	589209782	589029192	0.03	73456	73391	0.09
gzip	1941439	1941227	0.01	12217	12206	0.09
jpeg	45900780	45851813	0.11	19785	19771	0.07
nsieve	2288889385	2288889380	0.00	271	270	0.37
shorten	2166038	2166035	0.00	7819	7813	0.08
water	36111497	36110982	0.00	6469	6465	0.06
fmin	5370	5370	0	712	712	0
rkf45	818884	818884	0	2387	2387	0
seval	3340	3340	0	1055	1055	0
solve	2813	2813	0	1007	1007	0
svd	14649	14649	0	2481	2481	0
urand	1117	1117	0	192	192	0
zeroin	4603	4603	0	545	545	0
applu	884028559	883987259	0.00	13334	13319	0.11
cg	1432716153	1432716153	0	946	946	0
doduc	16953587	16941419	0.07	42127	42092	0.08
fpppp	95701038	95671829	0.03	10521	10518	0.03
matrix300	43073238	43073238	0	429	429	0
mg	87661635	87661635	0	2108	2108	0
tomcatv	436717483	434116483	0.60	953	951	0.21

Table 5.1 RBF vs. LS targeting SVLIW, on basic blocks

Benchmark	# blocks	avg. size	% unity	% > 100	avg. density
adpcm	82	178.6 ± 1193.7	87.8	2.4	1.0 ± 0.0
clean	1438	52.9 ± 671.4	89.5	1.2	0.985 ± 0.089
compress	227	222.9 ± 1467.3	82.4	2.6	0.969 ± 0.106
dfa	325	32.3 ± 553.8	96.3	0.6	1.0 ± 0.0
dhrystone	62	337.8 ± 1767.4	80.6	4.8	1.0 ± 0.0
fft	241	352.3 ± 1809.1	68.9	3.7	0.992 ± 0.064
go	10059	49.9 ± 654.5	89.5	1.0	0.986 ± 0.076
gzip	2093	99.8 ± 930.5	80.7	1.9	0.989 ± 0.066
jpeg	1961	96.0 ± 940.7	86.9	1.3	0.995 ± 0.049
nsieve	30	9.7 ± 34.9	86.7	3.3	0.835 ± 0.234
shorten	1379	48.5 ± 611.2	91.1	0.9	0.980 ± 0.097
water	303	463.4 ± 2025.9	79.9	6.3	0.992 ± 0.052
fmin	46	5.3 ± 21.2	87.0	2.2	1.0 ± 0.0
rkf45	107	9.2 ± 44.8	77.6	2.8	1.0 ± 0.0
seval	34	4.1 ± 7.6	70.6	0.0	1.0 ± 0.0
solve	82	3.1 ± 10.2	69.5	0.0	1.0 ± 0.0
svd	159	68.3 ± 790.7	81.1	1.3	1.0 ± 0.0
urand	13	825.5 ± 2655.3	84.6	15.4	N/A
zeroin	28	3.1 ± 5.9	64.3	0.0	1.0 ± 0.0
applu	590	938.2 ± 2776.2	66.6	14.4	1.0 ± 0.0
cg	89	1.5 ± 2.1	85.4	0.0	1.0 ± 0.0
doduc	1572	570.2 ± 2227.0	68.3	8.5	0.984 ± 0.085
fpppp	314	934.0 ± 2836.7	67.2	12.4	1.0 ± 0.0
matrix300	50	1.6 ± 2.2	80.0	0.0	1.0 ± 0.0
mg	161	699.4 ± 2523.3	67.7	8.1	1.0 ± 0.0
tomcatv	51	1399.6 ± 3365.9	47.1	19.6	0.92 ± 0.144

Table 5.2 Solution Space Size Estimates for Basic Blocks on SVLIW

times for each problem. We only looked at problems whose solution space estimate was between 2 and 100 inclusive. For problems with only one solution in the solution space the density must be 1. For problems with greater than 100 solutions we didn't feel 500 trials gave adequate coverage of the space for a reliable estimate as ϵ grew larger than 0.01. Numbers reported are averages and standard deviations for all problems in the benchmark meeting the above criteria. An entry of N/A means no problem instances in the benchmark met the solution space size criteria.

While the average solution space size is large for some problems, the optimal solution density is also very high. This suggests little opportunity for the random tie breaking approach to find better solutions. However, it also suggests that the number of trials required by RBF to find better solutions when they *do* exist is much lower than what we've been using. In fact, we detected very little difference in our results when using a cap of 50 trials instead of cap of 250 trials.

5.3.3 EBB's – Performance Results

In this section we compare the performance of RBF to LS when scheduling over extended basic blocks. We look at performance results when the algorithm simply picks the shortest schedule, and when the algorithm uses the weighting scheme described in Section 5.2.2. Table 5.3 reports dynamic and static operation counts for scheduling over EBB's. For this data only the schedule length was used in selecting the final schedule. Note the many cases where the code got slower (indicated by a negative percentage in the table.) This is because some operations were getting moved to a block where they executed more frequently. The biggest win for RBF came in the `fpppp` benchmark which ran 2.13% faster. As with basic block scheduling, the code size improvements found by RBF are small when scheduling over EBB's. As we would expect there are no increases in code size, but the reductions we do see are small. The largest reduction is .22% in `tomcatv`.

Next we evaluate the weighting techniques described in Section 5.2.2. In the first set of experiments we used a fixed weighting scheme. Instructions in each block of the EBB are weighted at .75 times the instructions in the predecessor block of the EBB. In the second set of experiments we used block level profile information to determine what the exact weights of the blocks should be.

Table 5.4 reports dynamic instruction counts and Table 5.5 gives the static results for the two sets of experiments. There are no increases in running time using the

	Dynamic Instructions			Static Instructions		
Benchmark	LS	RBF	%	LS	RBF	%
adpcm	12604412	12604412	0	311	311	0
clean	3769537	3772665	-0.07	9973	9967	0.06
compress	9625986	9626377	0.00	1374	1374	0
dfa	594859	594859	0	1000	1000	0
dhrystone	3350091	3350091	0	471	471	0
fft	22150570	22096800	0.24	2514	2511	0.12
go	501244191	501603369	-0.06	54010	53947	0.12
gzip	1752377	1757653	-0.29	10008	9996	0.12
jpeg	42529297	42480608	0.11	17484	17470	0.08
nsieve	2254242028	2254242023	0.00	249	248	0.40
shorten	2076582	2076584	0.00	6823	6817	0.09
water	33651055	34299695	-1.92	6039	6037	0.03
fmin	4495	4495	0	503	503	0
rkf45	761549	761482	0.01	2055	2052	0.15
seval	3283	3283	0	993	993	0
solve	2671	2672	-0.03	935	935	0
svd	13963	13963	0	2234	2234	0
urand	1081	1081	0	172	172	0
zeroin	4095	4095	0	446	446	0
applu	866954001	866912703	0.00	12933	12918	0.12
cg	1431526586	1431526586	0	882	882	0
doduc	16141047	16152702	-0.06	38536	38506	0.08
fpppp	90646630	88718146	2.13	9796	9737	0.60
matrix300	42883523	42883523	0	361	361	0
mg	84067412	84067412	0	2040	2040	0
tomcatv	436707058	434106078	0.60	912	910	0.22

Table 5.3 RBF vs. LS targeting SVLIW, on EBB's

weighted RBF scheme over LS and we even see a few small reductions in running time. The biggest improvement is found in `fpppp` where the running time is reduced by as much as 2.25 percent.

Comparing the static instruction sizes between Tables 5.3 and 5.5 we see a slight increase in static code size by adding the weighting scheme to RBF. This is because the algorithm no longer returns the shortest schedule it finds but rather the shortest weighted schedule. However the size of the code decreased slightly over LS as indicated in the table. Still, all improvements are quite small.

5.3.4 EBB's – Solution Space Size

In Table 5.6 we report solution space size estimates for extended basic blocks. Optimal solution densities are not reported because it is not clear what constitutes "optimal" when scheduling for extended basic blocks.

Not surprisingly the solution space sizes are larger than for basic blocks. The percentage of problems with only one solution in the restricted space has declined and the percentage of problems with greater than 100 solutions has increased over all benchmarks. These numbers demonstrate increased opportunity for RBF when scheduling over EBB's. This correlates to our slightly better performance findings described in the previous section.

5.3.5 DPS – Performance Results

In this section we look at the performance benefits of RBF over LS when scheduling over dominator-paths. The first set of data shows the results of unweighted RBF scheduling versus LS scheduling on dominator-paths. Table 5.7 gives dynamic and static instruction counts. There are some slight dynamic instruction count improvements but also some increases. The `water` benchmark showed an almost 2% increase in dynamic instruction count. Again, this is because operations may be getting moved from blocks with lower execution frequency to blocks with higher frequency. There were a few reductions in code size, the largest being .42% in `fpppp`.

Again we use the weighting techniques to make RBF competitive, this time applying the techniques to dominator-paths. Table 5.8 shows the dynamic results, and Table 5.9 the static results. Not surprisingly the profile based weighting did better overall, but the improvements were still small. The largest speedup over LS was 1.19% in `dfa`. In terms of code size, scheduling the weighting techniques showed little dif-

		0.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%
adpcm	12604412	12604412	0	12604412	0
clean	3769537	3768780	0.02	3768780	0.02
compress	9625986	9625986	0	9625986	0
dfa	594859	594859	0	594859	0
dhrystone	3350091	3350091	0	3350091	0
fft	22150570	22089120	0.28	22089120	0.28
go	501244191	500748080	0.10	500707496	0.11
gzip	1752377	1752161	0.01	1752161	0.01
jpeg	42529297	42478289	0.12	42478289	0.12
nsieve	2254242028	2254242023	0.00	2254242023	0.00
shorten	2076582	2076578	0.00	2076578	0.00
water	33651055	33649583	0.00	33649583	0.00
fmin	4495	4495	0	4495	0
rkf45	761549	761549	0	761482	0.01
seval	3283	3283	0	3283	0
solve	2671	2671	0	2671	0
svd	13963	13960	0.02	13960	0.02
urand	1081	1081	0	1081	0
zeroin	4095	4095	0	4095	0
applu	866954001	866812701	0.02	866812701	0.02
cg	1431526586	1431526586	0	1431526586	0
doduc	16141047	16133244	0.05	16131044	0.06
fpppp	90646630	88610400	2.25	88610400	2.25
matrix300	42883523	42883523	0	42883523	0
mg	84067412	84067412	0	84067412	0
tomcatv	436707058	434106038	0.60	434106078	0.60

Table 5.4 Dynamic instruction counts for SVLIW on EBB's, RBF with weighting vs. LS

		0.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%
`adpcm`	311	311	0	311	0
`clean`	9973	9967	0.06	9967	0.06
`compress`	1374	1374	0	1374	0
`dfa`	1000	1000	0	1000	0
`dhrystone`	471	471	0	471	0
`fft`	2514	2511	0.12	2511	0.12
`go`	54010	53949	0.11	53953	0.11
`gzip`	10008	9996	0.12	9998	0.10
`jpeg`	17484	17470	0.08	17470	0.08
`nsieve`	249	248	0.40	248	0.40
`shorten`	6823	6817	0.09	6817	0.09
`water`	6039	6037	0.03	6037	0.03
`fmin`	503	503	0	503	0
`rkf45`	2055	2055	0	2052	0.15
`seval`	993	993	0	993	0
`solve`	935	935	0	935	0
`svd`	2234	2234	0	2234	0
`urand`	172	172	0	172	0
`zeroin`	446	446	0	446	0
`applu`	12933	12918	0.12	12918	0.12
`cg`	882	882	0	882	0
`doduc`	38536	38512	0.06	38509	0.07
`fpppp`	9796	9737	0.60	9737	0.60
`matrix300`	361	361	0	361	0
`mg`	2040	2040	0	2040	0
`tomcatv`	912	910	0.22	910	0.22

Table 5.5 Static instruction counts for SVLIW on EBB's, RBF with weighting vs. LS

Benchmark	# EBB's	avg. size	% unity	% > 100
`adpcm`	51	334.2 ± 1534.6	80.4	5.9
`clean`	940	89.0 ± 888.7	82.1	1.9
`compress`	141	290.3 ± 1659.4	70.9	5.0
`dfa`	158	65.7 ± 793.0	77.2	1.3
`dhrystone`	42	478.0 ± 2129.2	78.6	4.8
`fft`	155	574.2 ± 2226.8	60.6	7.1
`go`	5095	119.0 ± 1003.5	75.6	2.8
`gzip`	1228	202.5 ± 1336.3	67.7	4.2
`jpeg`	1202	158.5 ± 1200.2	78.1	2.2
`nsieve`	19	24.2 ± 62.3	84.2	10.5
`shorten`	855	90.2 ± 876.2	83.4	1.6
`water`	192	796.5 ± 2583.9	71.4	11.5
`fmin`	23	9.5 ± 29.4	73.9	4.3
`rkf45`	62	198.0 ± 1273.8	66.1	6.5
`seval`	22	461.2 ± 2081.6	59.1	4.5
`solve`	53	15.5 ± 88.1	62.3	1.9
`svd`	101	24.5 ± 119.4	68.3	4.0
`urand`	8	450.8 ± 947.9	75.0	25.0
`zeroin`	15	5.2 ± 7.7	53.3	0.0
`applu`	402	1474.8 ± 3375.6	55.2	22.4
`cg`	62	3.7 ± 18.0	80.6	1.6
`doduc`	906	1085.9 ± 3017.6	38.3	16.3
`fpppp`	185	1441.4 ± 3446.0	50.3	18.9
`matrix300`	29	2.6 ± 3.1	55.2	0.0
`mg`	124	746.1 ± 2594.7	59.7	8.1
`tomcatv`	30	2396.7 ± 4105.0	23.3	40.0

Table 5.6 Solution Space Size Estimates for EBB's on SVLIW

	Dynamic Instructions			Static Instructions		
Benchmark	LS	RBF	%	LS	RBF	%
`adpcm`	10874359	10874359	0	294	294	0
`clean`	3635091	3635473	0.00	9965	9961	0.04
`compress`	9602440	9619037	-0.16	1379	1377	0.15
`dfa`	634611	634623	0.00	1040	1040	0
`dhrystone`	3230091	3230091	0	459	459	0
`fft`	22199415	22157931	0.19	2492	2491	0.04
`go`	490480041	492224216	-0.35	52240	52196	0.08
`gzip`	1807075	1807551	-0.02	10090	10083	0.07
`jpeg`	43051578	43003440	0.11	17843	17830	0.07
`nsieve`	2254242034	2254242029	0.00	247	246	0.40
`shorten`	1996296	1996295	0.00	6800	6794	0.09
`water`	33058154	33706986	-1.95	5822	5821	0.02
`fmin`	4114	4114	0	447	447	0
`rkf45`	749877	749877	0	2030	2030	0
`seval`	3281	3281	0	1012	1012	0
`solve`	2642	2643	-0.03	928	928	0
`svd`	14053	14155	-0.72	2268	2268	0
`urand`	1093	1093	0	168	168	0
`zeroin`	4043	4043	0	443	443	0
`applu`	867750207	867713909	0.00	12851	12837	0.11
`cg`	1431532626	1431532626	0	893	893	0
`doduc`	15264246	15293599	-0.18	37394	37366	0.07
`fpppp`	89892306	89311801	0.65	9662	9621	0.42
`matrix300`	42885923	42885923	0	367	367	0
`mg`	84066684	84066684	0	2041	2041	0
`tomcatv`	408095938	406795438	0.32	887	886	0.11

Table 5.7 RBF vs. LS targeting SVLIW, on DP's

ference over scheduling without weighting. The largest reduction was again in `dfa` at 1.54%.

5.3.6 DPS – Solution Space Size

Table 5.10 reports solution space size estimated for dominator-path scheduling problems. In most cases the solution space sizes are larger than those for EBB's. This is not surprising since dominator-path problems typically contain more operations to be scheduled. This data would seem to indicate more opportunity for RBF to find better solutions. However, the observed improvements of RBF over LS on dominator paths are actually slightly less than those of EBB's. We believe there may be less opportunity for improvements on DPS because of the large number of edges that must be inserted into the DPG to prevent illegal motion of operations between blocks.

5.3.7 Other Architectures

As can be seen in Appendix A, other architectures may derive more benefit from the RBF techniques described in this chapter. The appendix shows data for the PPC750 and C60 architectures. Both of these architectures have functional unit types that execute non-disjoint sets of operations. For example, one functional unit may be able to execute operations A and B and another can execute only A. Clearly if A and B are both ready at the same time, we want to be sure that A doesn't get scheduled on the functional unit required by B. The list scheduling priority heuristic does not take this fact into account. For example, we may want to increase the priority of all B operations in the previous example. However, the RBF framework can handle this situation without changes to the priority heuristic that may or may not improve the results.

The largest win when targeting the C60 came on the `fpppp` benchmark where we observe a 2.3% decrease in the running time of the EBB scheduled code when using RBF with the weighting techniques. Most of the other dynamic improvements were fairly small. Notice how the static instruction count changes for this architecture vary wildly. We see improvements up to 4.9% (`solve` DPS with profile weighting) but also see code size increases of as much as 4.75% (`zeroin` profile weighting). Also notice that in the VLIW case, we see some increase in code size when using the weighting techniques which buys us some decreases in running time. This pattern does not

		0.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%
`adpcm`	10874359	10874359	0	10874359	0
`clean`	3635091	3634389	0.02	3634389	0.02
`compress`	9602440	9602390	0.00	9602390	0.00
`dfa`	634611	627088	1.19	627088	1.19
`dhrystone`	3230091	3230091	0	3230091	0
`fft`	22199415	22199415	0	22137975	0.28
`go`	490480041	490275532	0.04	490252941	0.05
`gzip`	1807075	1807072	0.00	1807072	0.00
`jpeg`	43051578	43000567	0.12	43000567	0.12
`nsieve`	2254242034	2254242029	0.00	2254242029	0.00
`shorten`	1996296	1996293	0.00	1996293	0.00
`water`	33058154	33058026	0.00	33058026	0.00
`fmin`	4114	4114	0	4114	0
`rkf45`	749877	749877	0	749877	0
`seval`	3281	3281	0	3281	0
`solve`	2642	2642	0	2642	0
`svd`	14053	14050	0.02	14050	0.02
`urand`	1093	1093	0	1093	0
`zeroin`	4043	4043	0	4043	0
`applu`	867750207	867668907	0.01	867663907	0.01
`cg`	1431532626	1431532626	0	1431532626	0
`doduc`	15264246	15253976	0.07	15251831	0.08
`fpppp`	89892306	89890998	0.00	89293120	0.67
`matrix300`	42885923	42885923	0	42885923	0
`mg`	84066684	84066684	0	84066684	0
`tomcatv`	408095938	406795438	0.32	406795438	0.32

Table 5.8 Dynamic instruction counts for SVLIW on DP's, RBF with weighting vs. LS

		0.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%
adpcm	294	294	0	294	0
clean	9965	9961	0.04	9961	0.04
compress	1379	1377	0.15	1377	0.15
dfa	1040	1024	1.54	1024	1.54
dhrystone	459	459	0	459	0
fft	2492	2492	0	2491	0.04
go	52240	52195	0.09	52192	0.09
gzip	10090	10081	0.09	10082	0.08
jpeg	17843	17829	0.08	17828	0.08
nsieve	247	246	0.40	246	0.40
shorten	6800	6794	0.09	6796	0.06
water	5822	5821	0.02	5821	0.02
fmin	447	447	0	447	0
rkf45	2030	2030	0	2030	0
seval	1012	1012	0	1012	0
solve	928	928	0	928	0
svd	2268	2268	0	2268	0
urand	168	168	0	168	0
zeroin	443	443	0	443	0
applu	12851	12838	0.10	12837	0.11
cg	893	893	0	893	0
doduc	37394	37373	0.06	37375	0.05
fpppp	9662	9660	0.02	9621	0.42
matrix300	367	367	0	367	0
mg	2041	2041	0	2041	0
tomcatv	887	886	0.11	886	0.11

Table 5.9 Static instruction counts for SVLIW on DP's, RBF with weighting vs. LS

Benchmark	blocks	avg. size	% unity	% > 100
adpcm	45	544.9 ± 2140.0	82.2	6.7
clean	824	90.9 ± 884.3	81.7	2.1
compress	118	356.0 ± 1809.5	72.9	5.9
dfa	147	69.4 ± 821.9	87.8	0.7
dhrystone	38	527.7 ± 2232.6	78.9	5.3
fft	102	708.8 ± 2458.1	67.6	8.8
go	4418	172.5 ± 1221.2	79.8	3.8
gzip	1054	266.5 ± 1549.0	72.8	4.3
jpeg	999	168.7 ± 1252.3	82.3	2.5
nsieve	13	549.2 ± 1837.7	84.6	15.4
shorten	724	116.7 ± 1015.8	83.3	2.1
water	127	802.3 ± 2691.1	79.5	11.0
fmin	21	8.0 ± 20.8	81.0	0.0
rkf45	50	418.9 ± 1623.4	68.0	10.0
seval	18	32.6 ± 116.5	66.7	5.6
solve	37	275.3 ± 1620.8	70.3	2.7
svd	64	332.5 ± 1739.2	71.9	6.3
urand	7	617.9 ± 1242.5	71.4	28.6
zeroin	12	26.5 ± 69.7	66.7	8.3
applu	243	1744.7 ± 3778.1	68.7	18.5
cg	39	2.5 ± 5.6	71.8	0.0
doduc	640	1231.5 ± 3200.4	54.7	17.0
fpppp	149	1295.7 ± 3261.8	64.4	16.1
matrix300	17	12.4 ± 29.0	58.8	5.9
mg	84	742.4 ± 2572.3	61.9	10.7
tomcatv	11	3723.8 ± 4747.9	36.4	54.5

Table 5.10 Solution Space Size Estimates for DPs on SVLIW

hold true on the C60. These variances are due to the instruction compaction scheme described earlier.

RBF techniques showed more promise on the PPC750 architectural model. While most of the basic block improvements were small, three benchmarks showed improvement of over two and a half percent. Scheduling on EBB's brought improvements of up to 4.9% (`urand` with weighting). Improvements on dominator paths were slightly less. Notice the importance of the weighting schemes when scheduling over extended basic blocks and dominator paths. Without it we see some noticeable increases in running time.

5.4 Conclusions

In this chapter we explored the effects of scheduling "backwards" and randomly breaking ties in the list scheduling priority heuristic. A simple priority heuristic was used. It is possible that a more complicated heuristic could outperform the latency weighted depth heuristic, especially when scheduling over program regions larger than a single basic block. However, designing good heuristics is difficult and doesn't always pay off. By breaking ties in a simple heuristic randomly we are able to explore a larger space of solutions, and achieve some modest performance improvements.

It appears from the data that there is little opportunity to explore other solutions in the restricted solution space imposed by the list scheduling heuristic for traditional architectures and over the standard benchmarks. However on more complex architectures (complex from a scheduling perspective) some improvements are possible as shown in Section 5.3.7. The size of the solution spaces available to RBF are typically small, and the solution densities typically quite high. This suggests, that LS is often able to find the best solution in that restricted space. We saw some modest improvements in running time and code size on a few benchmarks.

Chapter 6

Iterative Repair Scheduling

Of late, the primary emphasis in instruction scheduling research has been in giving more "fodder" to the scheduler by increasing scheduling scope (as per Chapter 4), performing loop unrolling, *etc.* There has also been work on modifications to the list scheduling priority heuristic. The lack of interest in scheduling *algorithms* is due in large part to the belief that list scheduling finds an optimal solution most of the time. There is some evidence to support this, but there are very few recent experimental studies that explore alternative scheduling techniques on modern architectures.

One other reason to explore alternative scheduling techniques is the possibility of additional computing resources being available to the compiler. As computing resources increase (parallel processors, higher clock speeds), the compiler could conceivably spend more time trying to solve difficult problems. Iterative repair scheduling may be one way to generate better schedules by throwing more computational power at the problem. Using IR we can *search* the scheduling space to try to find better schedules. What is not clear is whether there are significantly better schedules in the solution space than those found by LS.

In this chapter we develop an alternative approach to one pass list scheduling and the RBF technique described previously. The technique is based on a scheduling technique developed in the AI community called iterative repair. Once we have this alternative algorithm at our disposal, we explore the efficacy of one pass list scheduling in the next chapter.

The rest of this chapter is organized as follows. In section 6.1 we introduce the iterative repair technique and generalize it to instruction scheduling. The evolution of our iterative repair algorithm is discussed in detail in section 6.2. Experimental results are given in the final section.

6.1 Introduction to Iterative Repair Scheduling

Here we introduce the application of a repair based scheduling technique called "iterative repair" (or IR) to the problem of instruction scheduling in a compiler. The general framework comes from the AI community and is described by Lin and Kernighan [57], and Zweben, *et. al.* [91, 90]. The technique has shown promise for several scheduling venues including space shuttle mission scheduling [89]. The technique was originally devised as a replacement for constructive search techniques. (See section 2.4.) It is especially useful in situations where resources, or deadlines can change during the course of the schedule (*e.g.* shuttle mission scheduling.)

Here we present IR as a technique to perform instruction scheduling. The generalized algorithm is presented in figure 6.2. The idea is straightforward. First, we create an instruction schedule that begins each operation as early as possible with respect to the precedence constraints of the DPG, ignoring the resource constraints imposed by the limited number of functional units. We say an operation is *in conflict* if it is scheduled at a time and on a functional unit that is over utilized with respect to the constraints imposed by the hardware model. The earliest cycle with a conflict is found, and one of the conflicting operations is selected (line (1) in the algorithm). This operation and all operations that depend on it are removed from the schedule (called *unscheduling*). The selected operation and its dependent operations are then inserted back into the schedule (called *rescheduling*) at a later point (line (2) in the algorithm). The process of unscheduling and rescheduling is called a *repair*. We continue repairing the schedule until there are no more resource conflicts. The algorithm is run a user specified number of times (or trials), and the shortest schedule over all trials is selected as the final schedule.

The iterative repair scheduling algorithm runs in polynomial time. Several variables affect the running time. (Here we equate running time with number of operations moved farther down the schedule.) Let T be the number of trials, R the number of repairs per trial, and C the complexity of a single repair. Then the the worst case complexity of the algorithm is $O(TRC)$. The parameters R and C will be discussed for each version of iterative repair that we describe.

It is important to note that not all possible schedules exist in the iterative repair solution space as we have described it. Rather, the set of schedules available to a particular flavor of IR is a subset of the greedy schedules. The solution space for the more powerful IR techniques is exactly the set of greedy schedules. For less powerful IR

```
cycle 1:A B C D | A B C | A B
cycle 2:        | D     | C D
```

(a) (b) (c)

Figure 6.1 Example showing incompleteness of iterative repair

techniques, the solutions spaces available to IR and LS (for a given priority heuristic) on a given problem could be disjoint, intersecting, or identical.

Consider the example in figure 6.1. Let `A`, `B`, `C`, and `D`, be operations with no dependences, and unit latency. Assume a machine with three identical functional units, each capable of executing all the operations. The schedule in part (a) of the figure is the as-early-as-possible (or ASAP for short) schedule used as the iterative repair starting point. Part (b) shows a schedule that iterative repair might generate, and part (c) shows a schedule that can *never* be generated by our formulation of iterative repair scheduling. This schedule won't be generated because the repair process stops when there are no more resource conflicts in the schedule.

Iterative repair provides an important improvement over list scheduling; the trade-off between the performance of the scheduled code and compilation time is made explicit. Performing more iterations of the algorithm results in a longer compile time, but usually produces a better final schedule. As an example of this see Figure 6.3. In this experiment we used an iterative repair scheduler to schedule a block from `twldrv` in the `fpppp` benchmark. The graph shows the number of iterative repair trials run versus the average length of the schedule found. We ran the scheduler ten times at each trial level with different random number seeds. Notice how the average schedule gets better as we increase the number of trials.

Another advantage is that the iterative repair algorithm *always* has a complete schedule to look at. Whereas a list scheduler builds up the schedule cycle by cycle, a repair-based algorithm starts with a complete schedule and modifies it. As we will describe in a moment, this added information can be used to help make better scheduling decisions.

6.2 Evolution of Our Technique

One major difference in using iterative repair for instruction scheduling is the use of resources. Unlike the complex resource interaction of job-shop problems, each

Input: Data Precedence Graph. Parameters of machine (instruction latencies, pipelining, number of functional units, etc.). The number of trials to perform *trials*.

Output: A schedule containing all operations in the graph that satisfies the precedence constraints of the DPG and the resource constraints of the machine.

Algorithm:
min = largest integer
shortest is a schedule initially empty
for x = (1 to *trials*)
    Create an initial *schedule* by scheduling all operations as early as possible
        subject to precedence constraints.
    while (there exist resource conflicts in *schedule*)
        conflict_time = the cycle of the first resource conflict in the schedule
(1)   select an *operation* that has a resource conflict at *conflict_time*
        unschedule *operation* and all its successors in DPG
(2)   reschedule *operation* and its successors later in *schedule*.
    endwhile
    if length of *schedule* is less than *min*
        then *min* = length of *schedule*
        shortest = *schedule*
    endif
endfor

Figure 6.2 Basic Iterative Repair Scheduling algorithm

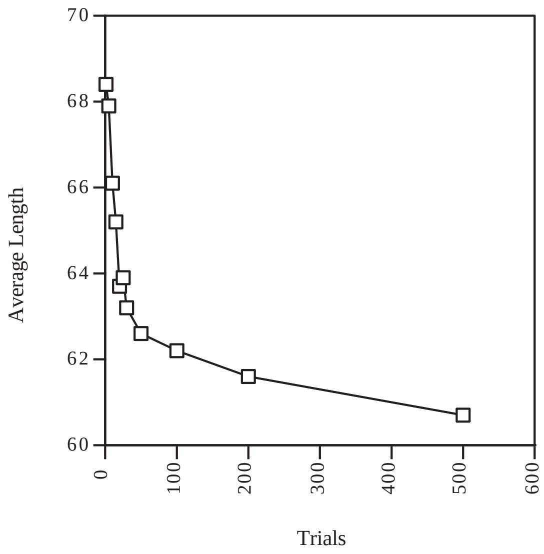


Figure 6.3 Example showing the time/schedule quality tradeoff of iterative repair.

task (operation) in an instruction scheduling problem requires only one resource: the functional unit on which it runs. (Registers can also be modeled as resources, as can other hardware constraints. See chapter 8.) Additionally, in order to model the hardware pipeline, a resource (functional unit) may not be required for the entire duration of the operation.

On architectures where the number of issue of slots is less than the number of functional units (*e.g.* our PPC750 model), we must also make a repair if there are too many operations scheduled to start at the same time. Thus, there exists a conflict if too many operations are scheduled on a given functional unit type, or if more operations are scheduled (over all functional unit types) during a given cycle than the number of issue slots available in the architecture.

During our research on using iterative repair for instruction scheduling we have made gradual changes to our move operator. This section presents the sequence of major refinements to the algorithm. The strengths and weaknesses of each refinement are described. Section 6.3 presents experimental results on each variation of the algorithm and demonstrates the improvements attainable with each refinement. In the following discussion, the numbers in parentheses refer to lines in the general algorithm as presented in figure 6.2.

6.2.1 IR0 scheduling

This version of iterative repair is designed to run quickly and is functionally equivalent to the algorithm used for NASA shuttle mission scheduling. The candidate (1) is selected randomly from those operations involved in the earliest resource conflict. Rescheduling is done by placing the operations back in the schedule at the earliest point where both precedence constraints *and* resource constraints are satisfied.

This method guarantees that the number of conflicts in the schedule will decrease by at least one after each repair. Thus we can perform at most N repairs per trial, where N is the number of operations, and each repair will cause at most N operations to move later in the schedule. At first glance, it would appear that this algorithm is $O(N^2)$ per trial. Closer inspection reveals that this is not the case. Let p be the maximum number of operations that can execute in parallel. If $p = N$ then all N operations in the graph must be repaired. But notice that if $p = N$ only one operation will be moved per repair, and the cost of completely repairing the schedule in $O(N)$. (We assume search time to find a free cycle is constant. For example, we could simply keep track of the next free slot in the schedule instead of scanning the schedule each repair.) Assume c is the length of the longest chain in the DPG. If $c = N - 1$ and the first operation in the chain is in conflict, then one repair is performed that moves $N - 1$ operations. Again the cost of repairing the schedule is $O(N)$. Notice that c and p can *not* both equal N. That is N operations can't execute in parallel if the longest chain is also N. In fact $p \times c = O(N)$. Thus, the number of operations that must be moved to repair the schedule is $O(N)$. Thus the worst case complexity of the whole repair process is $O(TN)$, where T is the number of trials to be performed.

6.2.2 IR-ran

The first problem we noticed with IR0 is illustrated in figure 6.4. This figure shows a portion of a DPG taken from the `fmin` benchmark. Assume that the double subtract operations (`dSUB`) take two cycles, the double multiplies (`dMUL`) take four cycles, and the absolute value (`dABS`) takes one cycle. Further we assume that there is only one completely pipelined functional unit on which to execute all these operations.

Figure 6.5 (a) shows the ASAP schedule. The first four subtract operations are initially scheduled during cycle zero. Whenever one of these operations gets moved `dSUB-e` and one of `dMUL-c` or `dMUL-d` get rescheduled. Scheduling the `dMUL` first results in a non-optimal schedule as shown in figure 6.5 (b). To alleviate this problem we modify the rescheduling phase of the algorithm to randomly select which successor operation to reschedule next. This version of the algorithm we refer to as IR-ran. When `dSUB-e` is scheduled first the schedule in figure 6.5 (c) results.

This modification changes the complexity of the algorithm. Every time we move a node we randomly iterate over the set of its' successors. This random iteration requires at most $O(s^2)$ time where s is the number of successors. We let S be the maximum number of successors of any node in the DPG. The running time is then $O(TNS^2)$. In practice S can often be bound by a small constant, and the impact of the random iteration is negligible. Also, notice that we have expanded the solution space available to the algorithm. This expansion occurs because a different choice of which successor to reschedule can result in different final schedules (as seen in Figure 6.5.)

6.2.3 IR-rdf

As mentioned earlier, the solution space for the iterative repair algorithm does not contain all possible schedules. Unfortunately, even IR-ran does not necessarily contain a schedule of optimal length. An example of this problem is shown in a block from the `zeroin` benchmark as pictured in Figure 6.6. Again we assume only one fully pipelined functional unit for all of the operations. We use the same latencies from the previous example and assign a latency of six to `dDIV`, a latency of one to the register-register copy (`d2d`), and a latency of one to the `JMPl` operation. The optimal schedule is shown in figure 6.7 (a). Any optimal schedule must have the `dDIV` scheduled at cycle 0. However in order for IR0 or IR-ran to keep `dDIV` in cycle 0 it must move `dMUL-a`. This forces `dMUL-b` to move after the `dSUB`, resulting in the non-optimal schedule in figure 6.7 (b).

The major constraint in IR0 and IR-ran scheduling is the rescheduling phase. Instead of rescheduling an operation at a point where its resource needs are met, IR-rdf (the next generation of our algorithm) ignores resource constraints when rescheduling. In essence, each operation that is rescheduled is moved at most one cycle later in the schedule. A later repair will remove any new conflicts introduced by this rescheduling.

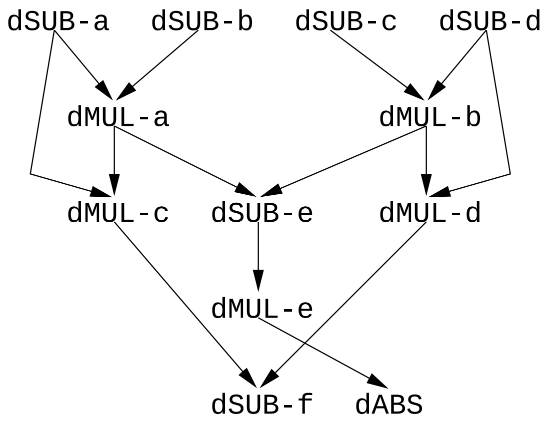


Figure 6.4 DPG from `fmin`

0: dSUB-a,dSUB-b,dSUB-c,dSUB-d	0: dSUB-a	0: dSUB-a
1: NOP	1: dSUB-c	1: dSUB-c
2: dMUL-a,dMUL-b	2: dSUB-b	2: dSUB-b
3: NOP	3: dSUB-d	3: dSUB-d
4: NOP	4: dMUL-a	4: dMUL-a
5: NOP	5: dMUL-b	5: dMUL-b
6: dMUL-c,dSUB-e,dMUL-f	6: NOP	6: NOP
7: NOP	7: NOP	7: NOP
8: dMUL-e	8: dMUL-c	8: dMUL-c
9: NOP	9: dMUL-d	9: dSUB-e
10:dSUB-f	10:dSUB-e	10:dMUL-d
11:NOP	11:NOP	11:dMUL-e
12:dABS	12:dMUL-e	12:NOP
	13:dSUB-f	13:NOP
	14:NOP	14:dSUB-f
	15:NOP	15:dABS
	16:dABS	
(a)	(b)	(c)

Figure 6.5 Scheduling figure 6.4 with iterative repair

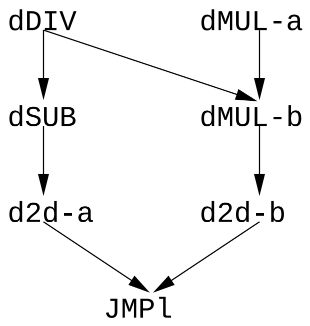


Figure 6.6 DPG from `zeroin`

(a)	(b)
`0: dDIV`	`0: dDIV`
`1: dMUL-a`	`1: dMUL-a`
`2: NOP`	`2: NOP`
`3: NOP`	`3: NOP`
`4: NOP`	`4: NOP`
`5: NOP`	`5: NOP`
`6: dMUL-b`	`6: dSUB`
`7: dSUB`	`7: dMUL-b`
`8: NOP`	`8: d2d-a`
`9: d2d-a`	`9: NOP`
`10:d2d-b`	`10:NOP`
`11:JMPl`	`11:d2d-b`
	`12:JMPl`

Figure 6.7 Two schedules for figure 6.6

Notice that we no longer need to reschedule successor operations randomly as with IR-ran, since all successors are simply moved at most one cycle later in the schedule.

This technique makes the search space and the solution space much larger, so we must explore it intelligently. The solution space contains all ASAP greedy schedules. An optimal length schedule for the problem is guaranteed to be in the space. In order to facilitate an intelligent search, we define an *evaluation function* that is used to determine the "goodness" of a schedule, with or without resource conflicts. The evaluation function maps schedules to integers. After each repair, we can evaluate the new schedule and see if it is "better" (using the evaluation function) than the schedule prior to the repair.

Our evaluation function computes the *resource dilation factor* (or RDF) of the schedule after each repair operation [89]. The RDF is computed by looking at each cycle c in the schedule. Let t be a functional unit type. Then, $n(t)$ is the number of functional units of type t available in the given architecture. Let $ops(c,t)$ be the number of operations scheduled to begin at cycle c on functional unit type t. We then compute $r_{c,t}$, which is the contribution to the RDF of a particular functional unit type t at cycle c, and sum over all cycles and functional unit types. The RDF is computed as follows:

$$\sum_c \sum_t r_{c,t} = \begin{cases} n(t) & \text{if } n(t) \leq ops(c,t). \\ ops(c,t) & \text{otherwise.} \end{cases}$$

Alternatively, RDF equals the number of cycles in the schedule multiplied by the number of functional units plus the number of operations that need to be moved due to a resource conflict. The RDF can also be computed as the sum of the operations in the schedule and the number of "holes" (unused functional units) in the schedule. We compute the RDF incrementally, to keep the computation as efficient as possible. After each repair, the new RDF is compared to the previous RDF. If the new RDF is lower, the new schedule is accepted. If the new RDF is higher, the new schedule is rejected stochastically. Currently we retain a schedule with a higher RDF five percent of the time. This is done to avoid local minima. Other hill-climbing techniques such as simulated annealing could also be used, but we have found little profit in using more complicated methods on this problem.

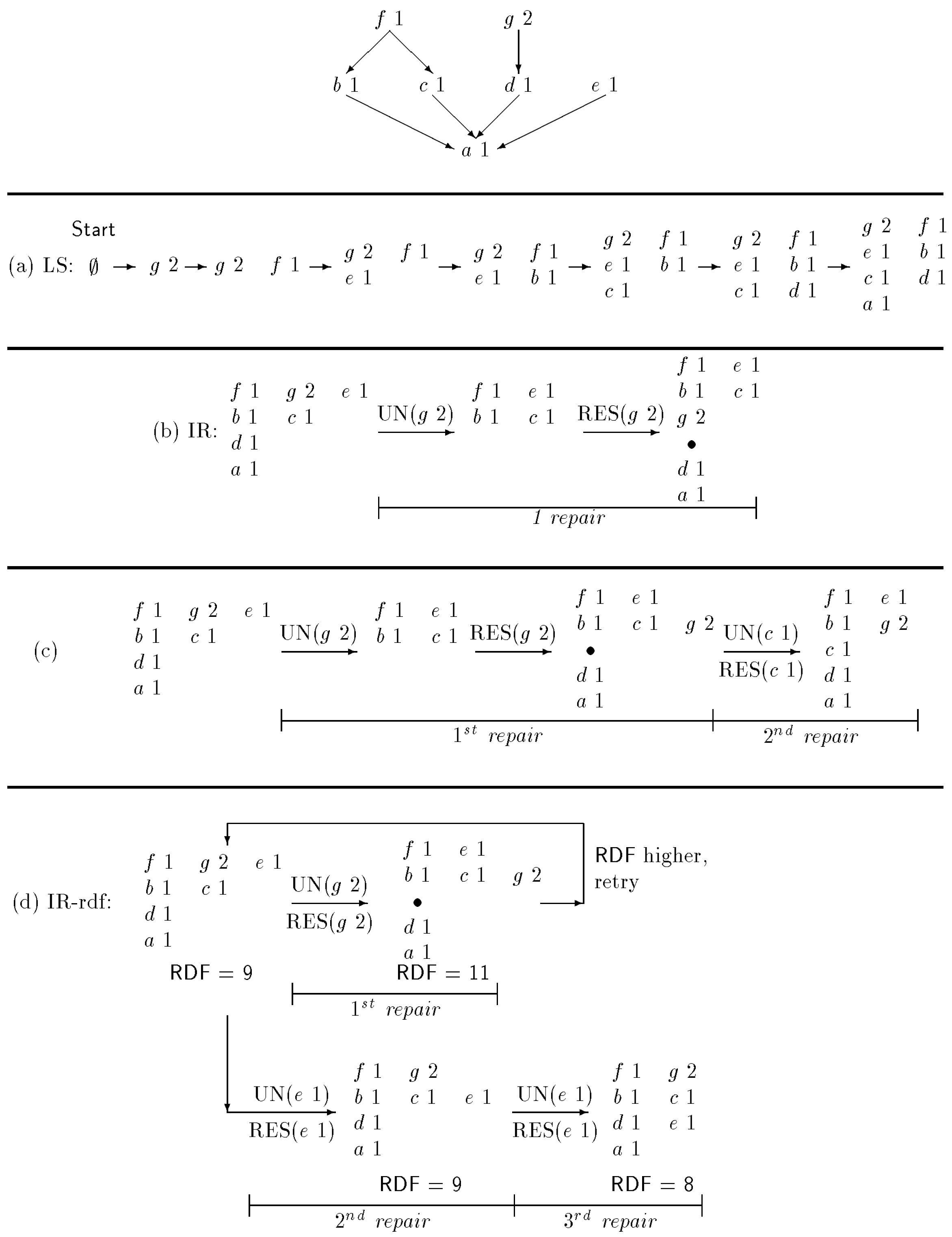


Figure 6.8 Comparing the Various Scheduling Techniques

Figure 6.8 illustrates our discussion thus far. At the top of the figure is a DPG. The operations are labeled with letters and the number next to each letter indicates the latency of the operation. Assume that all issue times are one cycle and the code is being scheduled on a machine with two identical functional units. Part (a) in the figure shows the progression of the schedule as constructed by LS. We start with an empty schedule, and build up the schedule cycle by cycle, until all operations have been scheduled. In part (b) we see how the IR0 algorithm generated a schedule. We start with all operations scheduled as early as possible. There is a conflict in cycle 0, so operation g is randomly selected for repair. It and its successors are unscheduled (UN($g2$) in the figure), and g is rescheduled (RES($g2$) in the figure) at the first cycle where all its resource constraints are met (cycle 2.) It's successors (d and a) are also re-scheduled. A bullet in the schedule indicates a `NOP`. In part (c), the schedule is constructed with the current technique (IR-rdf) but no backtracking based on the RDF is performed. Notice how g is rescheduled in cycle 1 which effectively moves the resource conflict from cycle 0 to cycle 1. A later repair on operation c yields the final schedule. Finally in part (d) we see how the full power of the IR-rdf algorithm is used to construct the final schedule. The schedule after the repair of g is found to have a higher RDF than the original schedule. Thus it is discarded, and the algorithm selects to repair operation e instead. The new RDF is the same as the original schedule so the schedule is kept. Finally, e is repaired again and a solution of optimal length results.

Let's look at the running time of the new formulation. The analysis is similar to that of IR0. Each repair still moves at most $O(N)$ operations, but we may need to perform $O(N^2)$ repairs. Consider a set of N nodes with no dependences being scheduled on a single functional unit. $N-1$ nodes must be moved to repair cycle 0, $N-2$ nodes for cycle 1 and so on. However, each of *these* moves takes constant time since only one operation is moved per repair. The maximum number of nodes that can be moved in a single repair is the length of the longest chain c in the DPG. Each node may be moved a maximum of p times where p is the maximum number of nodes that can execute in parallel (*i.e.,* the maximum number of nodes in conflict at any point during scheduling.) Notice that p and c can not both equal N, since N operations can not execute in parallel if the longest chain in the DPG is length N. Again, $p \times c = O(N)$. Thus, our worst case running time is $O(Ncp) = O(N^2)$. This does not take into account the backtracking done based on the RDF. This is difficult to quantify, since it depends on properties of the problem that can't be known until the problem is actually scheduled.

When scheduling over EBB's and dominator paths, we would like to encourage the scheduler to push operations down in the CFG to potentially less executed areas of the code as we did with RBF scheduling. To do this we define the *weighted* RDF. Simply put, operations in blocks that are farther down in the control flow graph contribute less to the RDF. Formally, let an EBB be an ordered list of blocks $B_1, \ldots, B_k$ such that B_i is the only predecessor of B_{i+1} in the CFG and B_1 has any number of predecessors. Then, each $r_{c,t}$ for a given block B_i in the above equation is multiplied by W^i, where $W^i < 1$ is the *weighting factor*. Ideally, the weighting factor would vary for each block in the EBB, and would be equal to the percentage of times the block executes with respect to the header of the EBB or DP. As in the last chapter with RBF scheduling, we tried a fixed weighting factor of .75 and weighting based on block level profile information.

6.2.4 IR-rdf-bias scheduling

IR-rdf-bias is built on IR-rdf. We use the same rescheduling technique proposed in the previous section, but we bias the selection of which conflicting operation to move. Before scheduling is performed each node in the DPG is assigned a priority using the latency weighted depth heuristic as used in the LS algorithm. Operations with a higher priority are less likely to be selected for moving. Thus, high priority nodes have a better chance of appearing earlier in the schedule.

To perform the biased selection we compute the *inverse priority* for each operation at a given cycle. Let *high* be the priority of the highest priority operation for a given cycle. Then the inverse priority of an operation equals *high*, minus the priority of the operation, plus one. These inverse priorities are then summed for the cycle. The probability that an operation gets chosen to be moved is equal to its inverse priority divided by the sum of the inverse priorities for the cycle. For example, assume three operations with priorities 1, 5, and 4. The highest priority value is 5. To compute an operation's inverse priority, we take 5 minus the operations priority. Thus, the inverse priorities are 5, 1, and 2, respectively. The probability that the first operation is selected to be moved is 5/8, the second 1/8, and the third 1/4. Biasing does not alter the worst case complexity of the algorithm as described in the previous section, but will increase the constant.

6.2.5 IR-length-bias scheduling

We discovered that simply using schedule length as the evaluation function sometimes works better than using the RDF. IR-length scheduling is the same as IR-rdf scheduling except that we keep or discard a new schedule solely based on the length of the new schedule and not on the RDF. Note that any repair can increase the length of the schedule by at most one cycle. A repair that does not increase schedule length is always accepted, while one that increases the length of the schedule is accepted 10 per cent of the time. We refer to the combination of the length heuristic and biased selection (see previous section) as IR-length-bias.

6.2.6 A move operator for intersecting functional unit types

As described in Chapter 3, some of our architectural models contain intersecting functional unit types. That is, given two different functional unit types f_i and f_j, $ops(f_i) \cap ops(f_j) \neq \emptyset$. When confronted with an operation that can run on more than one functional unit type, the LS algorithm arbitrarily selects on which type to schedule the operation. In our IR algorithm all of the operations in the intersection of the two types are initially scheduled on the same type. We modify the move operator for IR algorithms to employ a new kind of repair. This repair attempts to reschedule a conflicting operation to begin during the same cycle but on another functional unit type, sort of a horizontal instead of a vertical move. [8]

The current implementation of this additional move operator works as follows. After a conflicting node is chosen, the algorithm scans the functional unit types in the operation's current cycle to see if another functional unit type is available to execute the operation. If there is one available (*i.e.,* no additional conflicts are created by moving the operation to the other functional unit type) the operation is rescheduled to start in the same cycle on the new functional unit type and the repair is complete. If there is no other functional unit type available during the current cycle, but there does exist a different functional unit type on which the operation can execute, it is moved there with a probability of 50 percent. Functional unit types are canonically ordered, and an operation can only move to a functional unit type higher in this order. This ensures termination by preventing oscillations of operations between functional unit types.

[8]This is similar to the REASSIGN-POOL operator for shuttle mission scheduling as described on page 60 of Zhang's thesis [89].

6.3 Results of Iterative Repair Scheduling

6.3.1 Local Scheduling

In this section we compare the results of scheduling using the various iterative repair techniques described above on the SVLIW architectural model. The first set of tables report the results when applying the various scheduling techniques to single basic blocks. Each technique was run for 10 and 100 iterations on each block as indicated by the table heading. Any table entry that shows a result better than that found by LS is underlined. Each technique was run five times on each benchmark with a different random number seed. This could potentially give five different results for each benchmark. The reported results for the various iterative repair techniques are *averages* over the five runs. An entry without a decimal point indicates that the same result was produced on each of the five runs. Tables 6.1 through 6.3 report the dynamic and static instruction counts on the SVLIW architecture.

We draw some conclusions from this data:

1. As expected, increasing from 10 trials to 100 trials improved the performance of each particular IR technique.

2. The improvement going from IR0 to IR-ran was inconsistent. Increasing the size of the solution space leads to better solutions being available to the algorithm. However, poor solutions can also be added to the space which means that more trials may be needed in IR-ran to produce better solutions on average. On average IR-ran found a better solution than LS only once (for the `gzip` benchmark.)

3. Moving to IR-rdf dramatically increases the size of the solution space. Notice the slight degradation in `adpcm` when moving from IR-ran to IR-rdf at 10 trials due to this space increase. However, the increase in the size of the space is more than made up for by the superior solutions available in that space.

4. It is not clear whether IR-length-bias or IR-rdf-bias performs better on average.

5. Only very slight improvements over LS were observed, including some degradations. In terms of static code size there was only one benchmark (`mg`) for which some IR algorithm didn't find a better solution on average than LS. In terms of dynamic instruction count, there were several such benchmarks: `jpeg`,

Table 6.1 Dynamic Instruction Counts IR vs. LS targeting SVLIW, on basic blocks (1 of 2)

Benchmark	LS	IR		IR-ran	
		10	100	10	100
`adpcm`	13625959	13625959	13625959	13625959	13625959
`clean`	4515619	4524672.2	4520098.8	4525044.8	4520099.4
`compress`	10641307	10719391.8	10650015.0	10674493.2	10649447.0
`dfa`	696450	696450	696450	696450	696450
`dhrystone`	3660102	3662110.0	3660106.6	3662110.2	3660105.4
`fft`	22469970	23118520.2	22710807.8	23242787.8	22887756.8
`go`	589209782	590312971.6	589222214.6	590396556.8	589239340.0
`gzip`	1941439	1944670.6	1941230.2	1945563.4	1941229.8
`jpeg`	45900780	46711699.2	46521245.4	47094313.2	46674989.4
`nsieve`	2288889385	2288889394.2	2288889386.6	2288889394.2	2288889387.4
`shorten`	2166038	2180467.6	2166049.8	2182561.0	2166050.0
`water`	36111497	36306221.0	36250163.0	36349779.0	36264194.0
`fmin`	5370	5381.8	5377	5370	5370
`rkf45`	818884	821530.0	818885	820417.6	818884
`seval`	3340	3377.0	3356	3368.0	3349.8
`solve`	2813	2813	2813	2813	2813
`svd`	14649	14667.2	14659.2	14668.2	14657.0
`urand`	1117	1129.0	1117	1127.0	1117
`zeroin`	4603	4609	4609	4609.4	4609
`applu`	884028559	903604586.8	897515388.8	907473658.8	900000618.6
`cg`	1432716153	1433344753.8	1433276650	1433291933.0	1433262153
`doduc`	16953587	17328168.2	17197867.2	17299084.4	17166809.0
`fpppp`	95701038	107825759.2	106285331.4	108862435.8	106286228.4
`matrix300`	43073238	43073288.0	43073238	43073278.0	43073238
`mg`	87661635	95108953.8	91347742.2	95592833.8	91405671.8
`tomcatv`	436717483	455442646.0	448165964.0	456217896.2	450501764.0

Table 6.2 Dynamic Instruction Counts IR vs. LS targeting SVLIW, on basic blocks (2 of 2)

Benchmark	LS	IR-rdf 10	IR-rdf 100	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	13625959	13630060.6	13625959	13625959	13625959	13625959	13625959
clean	4515619	4515224.2	4515223	4516197.2	4515223	4516198.0	4515223
compress	10641307	10638494	10638494	10638521.0	10638494	10639088.6	10638494
dfa	696450	696450	696450	696450	696450	696450	696450
dhrystone	3660102	3660102	3660102	3660102	3660102	3660102	3660102
fft	22469970	22408521	22408520.8	22408520.8	22408520.6	22445385.4	22408521
go	589209782	589142608.4	589039436.8	589083262.0	589030623.0	589123281.2	589090440.2
gzip	1941439	1941228.6	1941227	1941227	1941227	1941227	1941227
jpeg	45900780	46243352.6	46054488.2	46049336.4	45906027.4	46042696.8	45925612.8
nsieve	2288889385	2288889380.2	2288889380	2288889380	2288889380	2288889380	2288889380
shorten	2166038	2166055.8	2166035	2166051.2	2166035	2166038.8	2166035.8
water	36111497	36162157.0	36145378.8	36113555.0	36110982	36111964.8	36111135.6
fmin	5370	5370	5370	5370	5370	5370	5370
rkf45	818884	818884	818884	818884	818884	818884	818884
seval	3340	3341.8	3340	3340	3340	3340	3340
solve	2813	2813	2813	2813	2813	2813	2813
svd	14649	14649	14649	14649	14649	14649	14649
urand	1117	1117	1117	1117	1117	1117	1117
zeroin	4603	4604.2	4603	4603	4603	4603	4603
applu	884028559	887560024.4	886477655.0	885409485.4	884692054.4	884490523.8	884312255.4
cg	1432716153	1432716153	1432716153	1432716153	1432716153	1432716153	1432716153
doduc	16953587	16994411.2	16966268.0	16959309.6	16949791.2	16956841.4	16949674.8
fpppp	95701038	100836137.4	98844254.6	96623891.0	96333376.0	96116283.6	95905666.2
matrix300	43073238	43073238	43073238	43073238	43073238	43073238	43073238
mg	87661635	88499932.6	88067404.6	87666739.0	87662451.0	87662995	87662723.0
tomcatv	436717483	436722588.6	434896783.0	435160967.6	434116483	435678103.2	434376583.0

Benchmark	LS	IR 10	IR 100	IR-ran 10	IR-ran 100	IR-rdf 10	IR-rdf 100	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	343	343	343	343	343	343.2	343	343	343	343	343
clean	11429	11446.2	11429.4	11448.6	11429.4	11423.6	11421	11421.4	11421	11422.0	11421
compress	1595	1607.8	1600.8	1606.8	1600.2	1594	1594	1594.2	1594	1594.4	1594
dfa	1356	1356	1356	1356	1356	1356	1356	1356	1356	1356	1356
dhrystone	525	533.2	529.6	533.4	528.4	525	525	525	525	525	525
fft	2708	2725.0	2714.0	2725.8	2716.0	2706	2705.8	2705.8	2705.6	2707.0	2706
go	73456	73643.2	73462.0	73646.2	73461.0	73431.6	73392.2	73403.4	73392.8	73417.0	73404.2
gzip	12217	12251.0	12219.2	12250.6	12215.4	12208.0	12205	12206.4	12205	12207.6	12205.6
jpeg	19785	19917.4	19856.8	19934.2	19859.2	19799.6	19787.6	19788.0	19779.0	19792.0	19782.2
nsieve	271	281.8	276.6	281.0	277.4	270.2	270	270	270	270	270
shorten	7819	7833.6	7816.0	7829.8	7816.4	7816.6	7813	7814.4	7813	7815.2	7814.4
water	6469	6608.0	6566.4	6625.8	6570.0	6484.6	6476.6	6466.6	6465	6467.6	6465.8
fmin	712	713.8	713	712	712	712	712	712	712	712	712
rkf45	2387	2391.0	2388	2389.0	2387	2387	2387	2387	2387	2387	2387
seval	1055	1061.8	1058	1060.4	1057.2	1056.0	1055.6	1055	1055	1055	1055
solve	1007	1007	1007	1007	1007	1007	1007	1007	1007	1007	1007
svd	2481	2484.4	2482.8	2485.4	2482.6	2481	2481	2481	2481	2481	2481
urand	192	193.2	192	193.0	192	192	192	192	192	192	192
zeroin	545	547	547	547.2	547	545.2	545	545	545	545	545
applu	13334	13803.8	13629.4	13877.2	13687.2	13403.6	13369.0	13351.6	13330.6	13338.6	13325.8
cg	946	950.6	948	948.4	947	946	946	946	946	946	946
doduc	42127	42927.8	42633.4	42908.6	42595.6	42185.4	42140.8	42126.4	42107.0	42131.8	42114.6
fpppp	10521	11440.8	11343.2	11480.0	11323.2	10828.2	10616.0	10560.0	10530.4	10527.8	10521.2
matrix300	429	430.0	429	429.8	429	429	429	429	429	429	429
mg	2108	2176.2	2142.8	2180.4	2144.0	2117.2	2112.4	2109.2	2108.6	2109	2108.8
tomcatv	953	972.4	964.8	972.4	966.6	956.2	951.6	953.4	951	952.6	951.2

Table 6.3 Static Instruction Counts IR vs. LS targeting SVLIW, on basic blocks

applu, and mg. The biggest improvement was observed on tomcatv for which IR-rdf-bias at 100 trials found a solution that was about .5 percent better in terms of dynamic instruction count than LS.

6. Comparing the IR results to the RBF results from the previous chapter we see even fewer wins for IR. In terms of static instruction count, IR found a better solution than RBF only for gzip. There were no wins in terms of dynamic instruction count and several losses.

The results for IR on other architectural models are mixed. Table C.2 shows dynamic instruction counts when targeting the C60. (Numbers for the less successful IR0 and IR-RAN are not shown.) In all but three cases some IR algorithm found a better schedule than LS. IR did better than RBF in five cases with the biggest win coming on tomcatv at about four tenths of a percent improvement over RBF. Static instruction count results can be found in Table C.1. As we saw in the last chapter, static instruction counts on this architecture are somewhat unpredictable.

Finding good schedules on the PPC750 was difficult for IR. Results are shown in Table C.9. Some benchmarks did show improvement, however many did not. Comparing the results to the RBF results of the last chapter, IR was only able to find improvements in three cases (svd, zeroin, and cg.) We know that good schedules exist in the space, but the more complex move operator has increased the size of the space to such an extent that we either need to run more trials, or add additional intelligence to the search technique to find them.

6.3.2 Extended Basic Block Scheduling

In this section we report the results of the various scheduling techniques when scheduling over EBB's. In light of the local scheduling results, numbers for the poorer techniques are not reported. Thus the following tables report results for the IR-rdf-bias, IR-length-bias, IR-rdf-wbias (0.75 weighting factor), and IR-rdf-profbias (profile based weighting) scheduling techniques. Tables 6.4 and 6.5 report dynamic instruction counts when targeting SVLIW, and Table 6.6 reports static instruction counts.

Note that the IR algorithms performed better against LS than in the basic block experiments. Only on applu did some IR algorithm not find at least as good a solution as LS in terms of dynamic instruction count. The biggest win came on fpppp with the IR-rdf-wbias techniques posting nearly a 4 and a half percent reduction in dynamic instruction count over LS, and a 2.2 percent reduction over the best solution returned

Benchmark	LS	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	12604412	12604532.0	12604572.0	12604572.0	12604592.0
clean	3769537	3774961.0	3776595.6	3777351.0	3778066.2
compress	9625986	9627286.2	9627836.4	9627140.8	9626598.4
dfa	594859	594859	594859	594859	594859
dhrystone	3350091	3368091.0	3350091	3368091.0	3350091
fft	22150570	22060449.0	22060449.0	22109601.6	22082465.0
go	501244191	502031477.4	501733484.4	502025907.4	501760076.6
gzip	1752377	1756579.2	1757614.0	1756727.2	1755564.0
jpeg	42529297	42778819.8	42570809.0	42705351.8	42609559.2
nsieve	2254242028	2254242023	2254242023	2254242023	2254242023
shorten	2076582	2086488.2	2076580.0	2089812.2	2079917.8
water	33651055	33766418.6	33711637.4	33727045.8	33697314.2
fmin	4495	4495	4495	4495	4495
rkf45	761549	761502.6	761533.6	761556.2	761472.2
seval	3283	3284	3283.8	3283.0	3284
solve	2671	2671.2	2671	2671.2	2671
svd	13963	13960.8	13961.0	13961.2	13960.4
urand	1081	1105.0	1093.0	1099.0	1099.0
zeroin	4095	4095	4095	4095	4095
applu	866954001	868408719.4	867863617.6	867558786.8	867273780.0
cg	1431526586	1431529387.6	1431526586	1431529387.6	1431527519.8
doduc	16141047	16162300.6	16150247.4	16157659.4	16148811.0
fpppp	90646630	89730795.4	88646000.0	90596556.6	90221188.8
matrix300	42883523	42883523	42883523	42883523	42883523
mg	84067412	84072244.4	84067412.8	84067956.4	84067412.6
tomcatv	436707058	434629306.0	434106038	436188886.0	434886346.0

Table 6.4 Dynamic Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over EBB's (1 of 2)

Benchmark	LS	IR-rdf-wbias 10	IR-rdf-wbias 100	IR-rdf-profbias 10	IR-rdf-profbias 100
`adpcm`	12604412	12604412	12604412	12604412	12604412
`clean`	3769537	3767152.2	3759516	3754460.0	3749446.4
`compress`	9625986	9626064.2	9625961	9626011.0	9622181.0
`dfa`	594859	594859	594859	594859	594859
`dhrystone`	3350091	3362091.0	3350091	3368091.0	3350091
`fft`	22150570	22007201.8	22007200.6	22007194	22007194
`go`	501244191	498313347.4	498155642.2	497792482.2	497541808.8
`gzip`	1752377	1752222.6	1752159	1752221.6	1752159
`jpeg`	42529297	42711983.2	42532756.4	42625453.0	42472095.8
`nsieve`	2254242028	2254242023	2254242023	2254242023	2254242023
`shorten`	2076582	2080086.4	2076578	2076622.4	2076513
`water`	33651055	33628855.6	33612868.0	33513067.4	33495643.0
`fmin`	4495	4495	4495	4495	4495
`rkf45`	761549	761473.2	761467.2	761218.6	761188.6
`seval`	3283	3279	3279	3279	3279
`solve`	2671	2671	2671	2671	2671
`svd`	13963	13958	13958	13958	13958
`urand`	1081	1081	1081	1081	1081
`zeroin`	4095	4095	4095	4095	4095
`applu`	866954001	868124590.6	867584611.8	868129912.4	867613612.4
`cg`	1431526586	1431526586	1431526586	1431526586.2	1431526586
`doduc`	16141047	16139114.6	16132933.4	16106471.0	16098210.8
`fpppp`	90646630	87351841.6	86697420.6	87524163.0	86833000.6
`matrix300`	42883523	42883523	42883523	42883523	42883523
`mg`	84067412	84068772	84067412	84072244.0	84067412
`tomcatv`	436707058	434629302.0	434106038	434629302.0	434106042.0

Table 6.5 Dynamic Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over EBB's (2 of 2)

Table 6.6 Static Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over EBB's

Benchmark	LS	IR-rdf-bias		IR-length-bias		IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100	10	100	10	100
adpcm	311	311	311	311	311	311	311	311	311
clean	9973	9969.2	9967.2	9969.0	9967	9969.4	9970	9978.8	9980.0
compress	1374	1375.4	1374	1374	1374	1375.0	1374	1374.8	1375.8
dfa	1000	1000	1000	1000	1000	1000	1000	1000	1000
dhrystone	471	471.6	471	471.6	471	471.4	471	472.6	471.6
fft	2514	2512	2512	2513.4	2512	2512.2	2511.6	2513	2513
go	54010	53982.8	53954.0	53991.8	53963.8	54056.4	54025.6	54247.8	54222.6
gzip	10008	10000.6	9996	10000.8	9995.8	10001.6	9998.4	10004.4	10001.4
jpeg	17484	17490.4	17477.6	17492.2	17480.6	17492.0	17479.0	17499.2	17492.6
nsieve	249	248	248	248	248	248	248	248	248
shorten	6823	6819.0	6817	6820.4	6818.4	6819.8	6817.4	6823.6	6822.8
water	6039	6042.0	6038.4	6039.8	6037	6046.8	6043.4	6048.8	6046.4
fmin	503	503	503	503	503	503	503	503.6	503.8
rkf45	2055	2050.8	2050	2051.0	2049.6	2051.8	2051.6	2052.6	2051.4
seval	993	993	993	993.2	993	994	994	994	994
solve	935	935	935	935	935	935	935	935	935
svd	2234	2234	2234	2234	2234	2234	2234	2234	2234
urand	172	172	172	172	172	172	172	172	172
zeroin	446	446	446	446	446	446	446	446	446
applu	12933	12951.2	12934.4	12939.4	12924.8	12951.6	12932.8	12956.2	12938.6
cg	882	882.4	882	882.4	882	882	882	882.4	882
doduc	38536	38542.8	38524.0	38550.6	38531.2	38551.4	38535.6	38637.0	38627.6
fpppp	9796	9782.8	9735.8	9793.0	9782.8	9768.8	9734.0	9773.6	9743.4
matrix300	361	361	361	361	361	361	361	361	361.2
mg	2040	2041.0	2040	2040.4	2040	2041	2040	2041.0	2040
tomcatv	912	911.0	910	912.0	910.6	911.2	910	911.0	910

by RBF. Looking at the static instruction counts, some IR technique performed better than LS in almost half of the benchmarks, though improvements were still small.

Results for other architectures are again found in Appendix C. When scheduling on the C60 (Tables C.3 and C.4), some IR technique found a better solution than the best RBF solution in a majority of the benchmarks, but again the improvements were modest. The biggest win for IR over LS came on the FPPPP benchmark, but RBF found an even better solution.

Results for the PPC750 are shown in Tables C.10 and C.11. On nine benchmarks some IR technique found a better solution than the best RBF technique. The largest win over RBF came on `seval` with a three percent improvement. However, as with basic blocks, there were several benchmarks for which IR didn't even beat LS. Clearly more work is needed to target IR for this architecture.

6.3.3 Dominator-Path Scheduling

In this section we look at some results of applying iterative repair scheduling techniques to dominator-paths. As we did for EBB scheduling we report on only the most powerful IR techniques: IR-rdf-bias, IR-length-bias, IR-rdf-wbias, and IR-rdf-profbias. This section shows results for the SVLIW architecture. Other results can be found in Appendix C.

Again, to see any real improvements in dynamic instruction counts (Tables 6.7 and 6.8) one needs to look at the results for the weighted techniques. The largest dynamic instruction count reduction was on `fpppp` using the profile based weighting (around 3 percent). Static count reductions were very minimal indeed. In general IR seems to have more difficulty finding better solutions for dominator-paths than it did for EBB's.

6.3.4 Best Results

In this section we look at the *best* (instead of the average) solution found by some IR technique on a variety of problems to get a feel for how much opportunity there is to improve over list scheduling using IR. That is, we looked at each individual solution for each IR technique, and recorded the best single solution over all runs and all techniques. For basic blocks, EBB's, and DP's we report the best solution found by any version of an iterative repair algorithm. These results are normalized to LS. A value less than one indicates that some IR technique found a better schedule than

Benchmark	LS	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	10874359	10874359	10874359	10874359	10874359
clean	3635091	3637987.6	3635771.4	3636400.8	3635945.2
compress	9602440	9605363.0	9603107.6	9604177.8	9605499.4
dfa	634611	634795.2	634824.2	634798.4	634749.8
dhrystone	3230091	3230091	3230091	3236091.0	3230091
fft	22199415	22133879.0	22146167.0	22256759.4	22166647.0
go	490480041	491166048.8	490843208.4	491182437.2	490678820.2
gzip	1807075	1807100.8	1807100.4	1807082.8	1807102.0
jpeg	43051578	43322223.4	43095556.2	43286497.6	43176816.2
nsieve	2254242034	2254242029	2254242029	2254242029	2254242029
shorten	1996296	1996336.0	1996371.4	1996337.0	1996372.2
water	33058154	33073588.8	33064727.2	33063104.4	33059083.0
fmin	4114	4125.6	4119.4	4119.8	4116.8
rkf45	749877	749877	749877.8	749878.6	749877.4
seval	3281	3281	3281	3281	3281
solve	2642	2642	2642	2642	2642
svd	14053	14058.0	14052.0	14068.4	14051.0
urand	1093	1103.8	1107.4	1109.2	1107.4
zeroin	4043	4043	4043	4043	4043
applu	867750207	870034122.2	869178241.4	869295117.2	868655872.2
cg	1431532626	1431534493.6	1431534493.6	1431534494.8	1431535431.0
doduc	15264246	15292160.6	15276755.2	15286888.4	15274299.6
fpppp	89892306	89583664.2	88066267.6	89806193.0	89509012.6
matrix300	42885923	42885923	42885923	42885923	42885923
mg	84066684	84437944.8	84131604.0	84236941.6	84193212.0
tomcatv	408095938	408628386.0	406556605.8	410965245.4	409918674.0

Table 6.7 Dynamic Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over DP's (1 of 2)

Benchmark	LS	IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100
adpcm	10874359	10874359	10874359	10874359	10874359
clean	3635091	3635436.6	3633670	3633602.0	3633599
compress	9602440	9607488.4	9606348.0	9606950.8	9605223.0
dfa	634611	634851.0	634824.6	634890.2	634749.0
dhrystone	3230091	3244091.0	3230091	3230091	3230091
fft	22199415	22084727.2	22056055	22068343.2	22056055
go	490480041	488622742.6	488293897.6	488039967.8	487670829.4
gzip	1807075	1807092.6	1807069.2	1807087.6	1807069
jpeg	43051578	43321140.0	43197987.8	43320055.4	43094090.0
nsieve	2254242034	2254242029	2254242029	2254242036.0	2254242036.0
shorten	1996296	2009499.4	1996280.2	1996181.2	1996165
water	33058154	33075098.6	33065936.0	33010833.6	32999587.0
fmin	4114	4120.8	4116.4	4120.4	4114
rkf45	749877	750731.4	749877	749877	749877
seval	3281	3282	3282	3281	3281
solve	2642	2642	2642	2642	2642
svd	14053	14059.6	14048	14055.0	14048
urand	1093	1093	1093	1093.6	1093
zeroin	4043	4043	4043	4043	4043
applu	867750207	870417272.8	869045571.0	869699611.0	869025904.2
cg	1431532626	1431532626	1431532626	1431532626	1431532626
doduc	15264246	15275890.8	15258725.0	15257222.6	15242886.0
fpppp	89892306	90516180.2		87934837.6	87173630.4
matrix300	42885923	42885923	42885923	42885923	42885923
mg	84066684	84470356.0	84084860.0	84437944.8	84070996.0
tomcatv	408095938	410708158.0	407064710.0	408628386.0	407593847.2

Table 6.8 Dynamic Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over DP's (2 of 2)

Table 6.9 Static Instruction Counts various IR techniques vs. LS targeting SVLIW scheduling over DP's

Benchmark	LS	IR-rdf-bias		IR-length-bias		IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100	10	100	10	100
adpcm	294	294	294	294	294	294	294	294	294
clean	9965	9962.8	9961.2	9962.2	9961	9962.6	9962	9978.4	9977.2
compress	1379	1379.0	1377.6	1379.8	1378.2	1379.8	1378.4	1379.8	1378.6
dfa	1040	1046.4	1046.4	1047.4	1045.6	1047.8	1046.4	1048.6	1045.6
dhrystone	459	459	459	459.2	459	459.6	459	459.6	459.6
fft	2492	2491.2	2491	2493.2	2491.4	2491.6	2491	2493.2	2491.8
go	52240	52263.6	52221.6	52261.0	52224.4	52407.8	52337.4	52545.8	52489.6
gzip	10090	10086.8	10084.4	10085.8	10082.8	10092.6	10086.2	10095.8	10093.8
jpeg	17843	17856.6	17845.2	17858.0	17848.2	17859.2	17847.6	17872.6	17861.6
nsieve	247	246	246	246	246	246	246	247.4	247.4
shorten	6800	6796.4	6794	6797.0	6795.4	6800.6	6794.8	6804.8	6803.0
water	5822	5833.6	5825.8	5829.8	5823.6	5837.6	5828.8	5838.6	5831.6
fmin	447	448.2	447.2	447.6	447	448	447.4	448	447
rkf45	2030	2030	2030	2030	2030	2031.2	2030	2030.2	2030.4
seval	1012	1012	1012	1012	1012	1013	1013	1012	1012
solve	928	928	928	928	928	928	928	928	928
svd	2268	2268.6	2268	2268.8	2268	2269.0	2268	2269.2	2268
urand	168	168	168	168	168	168	168	168.6	168
zeroin	443	443	443	443	443	443	443	443	443
applu	12851	12889.2	12868.8	12871.8	12857.0	12891.6	12867.2	12903.0	12877.6
cg	893	893	893	893	893	893	893	893	893
doduc	37394	37428.6	37395.8	37435.4	37407.0	37451.6	37421.4	37615.0	37596.2
fpppp	9662	9682.8	9644.6	9670.4	9659.6	9732.6		9718.4	9674.8
matrix300	367	367	367	367	367	367	367	367.6	367.4
mg	2041	2046.2	2043	2044.6	2042.2	2046.8	2042.8	2046.2	2043
tomcatv	887	890.4	888.2	891.4	889.2	891.8	888.0	890.4	888.2

LS. We present separate results for static and dynamic instruction counts for basic block scheduling, EBBS, and DPS on the SVLIW architecture. We don't report which technique generated the best solution. Most of the best solutions for basic blocks came from the IR-rdf-bias technique. Most of the best dynamic solutions for EBB and DPS were from IR-rdf-bias using profile based weighting.

Most numbers in the table are unity, and the others are close. The closeness of these numbers to what we saw when looking at averages, indicates that some IR technique is converging at around 100 trials most of the time. This is not always the case however. Notice that for `fpppp` on basic blocks, and `jpeg` on dominator paths, the best result was above one. We know that schedules as least as good as those found by LS exist in the IR solution space. However, 100 trials was insufficient to find those solutions in these two cases.

6.4 Summary of IR

In this chapter we applied the iterative repair scheduling technique to the problem of instruction scheduling in a compiler. To our knowledge, this has not been attempted before. IR offers several advantages over a constructive technique like LS. Since IR has a complete schedule to look at, it can make more intelligent scheduling decisions. Also, the randomized methodology of IR allows many solutions to be generated, and the best one selected. One drawback of IR, is that it is slower than LS for many problems.

When applying IR to real scheduling problems, LS proved to be quite competitive. In general, IR faired better on more complex architectures (*i.e.,* the PPC750 and the C60), and in more complex problem spaces (*i.e.* EBB's and DP's.) The next chapter explores why it is so difficult to produce schedules that are superior to those produced by LS.

Benchmark	Dynamic BB	Dynamic EBB	Dynamic DPS	Static BB	Static EBB	Static DPS
adpcm	1.000	1.000	1.000	1.000	1.000	1.000
clean	1.000	.994	1.000	.999	.999	1.000
compress	1.000	.999	1.000	.999	1.000	.999
dfa	1.000	1.000	1.000	1.000	1.000	1.003
dhrystone	1.000	1.000	1.000	1.000	1.000	1.000
fft	.997	.994	.994	.999	.999	1.000
go	1.000	.993	.994	.999	.999	1.000
gzip	1.000	1.000	1.000	.999	.999	.999
jpeg	1.000	.999	1.001	1.000	1.000	1.000
nsieve	1.000	1.000	1.000	.996	.996	.996
shorten	1.000	1.000	1.000	.999	.999	.999
water	1.000	.995	.998	.999	1.000	1.000
fmin	1.000	1.000	1.000	1.000	1.000	1.000
rkf45	1.000	1.000	1.000	1.000	.997	1.000
seval	1.000	.999	1.000	1.000	1.000	1.000
solve	1.000	1.000	1.000	1.000	1.000	1.000
svd	1.000	1.000	1.000	1.000	1.000	1.000
urand	1.000	1.000	1.000	1.000	1.000	1.000
zeroin	1.000	1.000	1.000	1.000	1.000	1.000
applu	1.000	1.001	1.001	.999	.999	1.000
cg	1.000	1.000	1.000	1.000	1.000	1.000
doduc	1.000	.997	.999	.999	1.000	1.000
fpppp	1.002	.955	.965	1.000	.993	.998
matrix300	1.000	1.000	1.000	1.000	1.000	1.000
mg	1.000	1.000	1.000	1.000	1.000	1.000
tomcatv	.994	.994	.994	.998	.998	1.000

Table 6.10 Best IR solution targeting SVLIW

Chapter 7

The Near Optimality of List Scheduling

As the experimental evidence in the last chapter showed, when scheduling over basic blocks it is very difficult to generate better schedules than LS. There appear to be more opportunities to generate shorter schedules for extended basic blocks and dominator paths, and finding better *runtime* schedules for these scopes appears even more promising.

The work described in this chapter has two goals. First, to try to understand why the LS algorithm performs so well. Second, to develop some insight into the types of instruction scheduling problems that are "difficult" and thus warrant the use of a more powerful scheduling technique like RBF or IR.

In his thesis, Fisher evaluated many list scheduling techniques on random graphs. He used a lower bound metric similar to what we describe in Section 7.1.1 to establish the minimum schedule length for a particular problem. He then charted the effectiveness of the various techniques against that lower bound. He demonstrated that as the number of operations in a scheduling problem increases list scheduling tends to find more suboptimal schedules. (We verify this result later on.) Our work extends his in several key ways. First, we evaluate the performance of LS on real codes as well as random ones. Second, we use more powerful scheduling techniques to try to beat LS including RBF, IR, and integer programming. Finally we explore the relationship between scheduling difficulty and the amount of parallelism present in the scheduling problem.

It has been shown that a certain class of graphs known as the monotone interval order graphs (MIOG's) can be optimally scheduled in linear time by a simple greedy algorithm similar to list scheduling [68]. However, our experience is that on problems large enough to be interesting, these graphs appear infrequently in real code. For our benchmarks, about half the basic blocks with more than five operations could be represented with MIOG's. In blocks with more than ten operations, only about 18 percent of DPG's were MIOG's. We conclude that there must be some other reason that LS is doing so well scheduling real code.

There are two possible reasons for our limited success in finding shorter schedules. The first possibility is that the alternative algorithms are not very good and therefore seldom find schedules better than those found by LS. The second possibility is that LS is already finding many solutions that are close to optimal, leaving little room for alternative techniques to achieve any improvement. The next two sections give evidence for the latter argument. Section 7.2 investigates situations under which an alternative scheduling algorithm may prove profitable.

7.1 How many problems is LS scheduling optimally?

In this section we establish lower and upper bounds for the number of instruction scheduling problem instances from real code that LS is scheduling optimally. The next subsection describes a method to estimate the minimum schedule length given a DPG and an architecture. Section 7.1.2 describes how we encode scheduling problems as integer programming problems. These problems are solved with Ilog's CPLEX (version 6.0.2) to determine an optimal solution. Section 7.1.3 presents experimental results on the number of problems LS is scheduling optimally.

7.1.1 A lower bound on minimum schedule length

One way to see if LS is finding optimal schedules would be to determine the optimal solution for each basic block by enumerating all schedules. Since this may be prohibitively expensive for many problem instances, we first develop a metric based on the DPG that represents a lower bound on the schedule for that DPG on a given architecture. The metric takes into account several ideas.

The first component of our estimate is the critical path length of the DPG G denoted by $cpl(G)$. This is the length (in cycles) of the longest (latency weighted) path from a leaf in G to a root. Since any schedule must ensure all dependences in G are satisfied, there exists no valid schedule whose length is less than the critical path length. Thus, if a scheduling algorithm finds a schedule whose length equals the critical path length for a particular problem, no more work on that problem will result in a better schedule.

The second component of the metric takes into account hardware constraints of the target architecture. This is a variation of Fisher's technique described in Chapter 6 of his thesis [34] and is less costly to compute. Since critical path length does not take hardware constraints into account, we refine this minimum schedule length estimate in the following way. First, we find all operations along some critical path in the DPG. These operations are important because if any of them is scheduled later than their ASAP start time, the final schedule will be longer than the critical path length. These operations are scheduled as early as possible and a *resource* array $R[f, c]$ is created. This two-dimensional array is indexed by functional unit type and cycle. Each entry in the table represents the over-utilization of a particular functional unit type in a given cycle when the critical path operations are scheduled as early as possible. For example, a value of zero for functional unit type f_i at cycle c means that exactly $\mid f_i \mid$ critical operations are using f_i at time c. (Recall that a functional unit type is a *set* of identical functional units, thus $\mid f_i \mid$ is the number of units of a given type available.) In general, a value $v \neq 0$ indicates that there are $v + \mid f_i \mid$ operations scheduled to use the functional unit type. For each functional unit type we compute i and j such that $\sum_{i \leq x \leq j} R[fu, x]$ is maximized. That is, we find the largest subsequence sum in the resource array. This value, max, is the *minimum* number of cycles that operations on some critical path must be delayed when scheduling for the target architecture. Dividing max by $\mid f_i \mid$ gives the minimum number of cycles that the critical path will need to be delayed due to hardware constraints. Thus, the value $\frac{max}{|f_i|} + cpl(G)$ gives a lower bound on the schedule length of the DPG scheduled on a particular processor. We refer to this as the augmented critical path length.

The above computation is correct on architectures with non-intersecting functional unit types, but breaks down when such is not the case. For example, assume two operations can run on f_i, one of them can run on f_j, and both are critical path operations that start at the same time in the ASAP schedule. If the target architecture has one unit of each type, and both are assumed to execute on f_i we would erroneously increment our minimum schedule length by one. Thus we must be more careful with this type of architecture. We define a *class* of functional types as a set of functional types. Functional units f_a and f_b are in the same class if and only if there exists a sequence $(f_a, \ldots, f_b)$ of functional unit types such that $ops(f_i) \cap ops(f_{i+1}) \neq \emptyset$ for all f_i in the sequence. Intuitively, if there exists more than one functional unit type that can execute a given operation, all those types must be in the same functional unit class. We conservatively assume that if an operation can execute on one functional unit type in a given class, it can execute on any type in the class. To compute the augmented critical path length, we proceed as in the preceding paragraph but index the resource array R by functional unit class instead of functional unit type. In some situations this can grossly underestimate the minimum schedule length. For example, on the PPC750 only one functional unit type can execute the `iDIV` operation, but there are three functional unit types in the class due to other operations. Thus if two critical `iDIV` operations requiring the functional unit for 19 cycles are scheduled to start at the same time, we would add $\lceil \frac{19}{3} \rceil = 7$ cycles to the augmented critical path length instead of 19. To correct for this, we make another pass over the schedule and compute the augmented critical path length on critical operations that can only execute on a single functional unit type.

The final measure we use for the minimum schedule length is based on the total number of operations in the DPG that can execute on a given functional unit class. Dividing the number of operations that can execute on a class by the number of functional units in that class, gives a lower bound on the length of the schedule. On processors where the number of issue slots is less than the number of functional units, we divide the total number of operations in the DPG by the number of issue slots and find the maximum of this value and the previous value. We call this value the *saturation* length. Our lower bound on schedule length is the maximum of the augmented critical path length and the saturation length. No valid can schedule can be shorter than this estimate.

7.1.2 Instruction Scheduling as an Integer Programming Problem

To solve instances of instruction scheduling problems optimally, we encode the instance as an integer programming problem and feed it to the CPLEX solver. [9] Encoding the scheduling problem as an integer programming problem proceeds as follows. For every regular edge (op_1, op_2) in the DPG we add the inequality $Xop_1 - Xop_2 \geq lat(op_2)$ to the problem. This ensures that the starting time of op_1 (represented by the variable Xop_1) is at least $lat(op_2)$ (the latency of op_2) cycles greater than the starting time of op_2 (represented by the variable Xop_2). For anti-edges the right-hand side, $lat(op_2)$, is replaced by 0. This takes care of the precedence constraints.

Encoding the resource constraints is a bit trickier. For every functional unit f_i that can execute op we add a binary variable Yf_iop indicating whether or not op executes on f_i. Since only one functional unit can execute a given operation, we add the following constraint equation: $Yfu_1op + \ldots + Yfu_nop = 1$. This forces the

[9] This approach was chosen because of the ready availability of CPLEX and the ease of using CPLEX from within our own code. Other optimal techniques can be used, such as formulating the scheduling problem as a satisfiability problem [25].

operation onto one and only one functional unit. If two operations op_1 and op_2 are not related via some chain of dependences (or are only related via anti-dependences), then they could be scheduled to run at the same time. If this happens we must be certain that they don't execute on the same functional unit. Assume that both operations can execute on f_1 and f_2. We must ensure that if the operations run on the same functional unit that they are scheduled at different times, or if they are scheduled at the same time, are running on different functional units. The following five inequalities enforce this constraint.

$$t_i + t_j \leq 1$$

$$Xop_1 - Xop_2 - 1001t_i \geq -1000$$

$$Xop_2 - Xop_1 - 1001t_j \geq -1000$$

$$Yf_1op_1 + Yf_1op_2 - t_i - t_j \leq 1$$

$$Yf_2op_1 + Yf_2op_2 - t_i - t_j \leq 1$$

Let's look at how these equations work. We introduce two binary variables t_i and t_j. The first equation forces at most one of these to be one (both could be zero.) Intuitively these variables indicate whether op_1 or op_2 is scheduled first. If t_i is one (t_j must be zero) then op_1 must start before op_2 by the second equation. The third equation is trivially satisfied. (We assume no two operations are separated by more than 1000 cycles in any schedule.) The fourth and fifth equations are also trivially satisfied. Since the two operations start at different times, we don't care what functional units they execute on. To handle operations whose issue time is greater than 1, we replace the coefficient 1001 in the second and third equations with 1000 plus the issue time of the operation on the given functional unit.

Let's look at what happens if t_i and t_j are both zero. This situation forces the operations to be scheduled on different functional units. The fourth equation ensures that only one operation can execute on f_1 and the fifth equation does the same for f_2. Since the operations are on different functional units, we don't care about their start times and the second and third equations are trivially satisfied.

We have CPLEX minimize the starting time of the last operation in the basic block (or extended basic block.) We help CPLEX by setting an upper bound on the optimal schedule length equal to the length of the schedule returned by LS. A lower bound is established by using the minimum schedule length estimate. We limit CPLEX to three minutes per scheduling problem. If we still don't know the optimal schedule length, we assume that the LS solution is non-optimal to establish a lower-bound on the number of optimal schedules found by LS, and optimal to establish an upper bound.

7.1.3 Experimental results

In this section we experimentally determine the number of scheduling problems from real codes that LS is scheduling optimally. In the following experiments we establish lower and upper bounds on the performance of LS on basic blocks and extended basic blocks. The first step is to schedule the problem with LS. If the schedule length returned by LS equals the minimum schedule length estimate we count that as an optimal schedule. If the length of the schedule returned by LS is greater than the minimum length estimate then we don't yet know the optimal length schedule. We schedule the problem with RBF and IR-rdf-bias. If one of these returns a better schedule, we know the LS result is non-optimal. If not, we still don't know whether the LS solution is optimal. In these cases we formulate an integer programming problem as described above and solve it using CPLEX to find the optimal solution. If CPLEX finds an optimal solution within three minutes, we can classify the LS result as optimal or non-optimal. If CPLEX fails to find an optimal solution in three minutes, but was able to find some solution lower than LS, then we know the LS solution was non-optimal. In all other cases, we assume that the LS solution was non-optimal to compute the lower bound, and optimal to compute the upper bound.

We computed the percentage of problems that LS was able to schedule optimally on the SVLIW architecture. The data in Table 7.1 shows our experimental results. Each entry in the table can be a single value or a range. A single value indicates that we were able to prove every solution returned by LS to be either optimal or non-optimal. A range of values shows the lower bound on the percentage of problems for which LS found an optimal solution and the upper bound. By "optimal" in this case, we mean the shortest static schedule. The numbers are quite high. Clearly LS is performing very well.

7.2 Amount of Parallelism

Recently there has been work that attempts to classify the difficulty of particular instances of NP-complete problems. For example Cheeseman, *et. al.* examine the

Benchmark	Basic blocks	EBB's
`adpcm`	100	98.0 - 100
`clean`	99.4 - 99.5	98.5 - 99.4
`compress`	98.7 - 99.6	97.2 - 100
`dfa`	100	100
`dhrystone`	100	97.6 - 100
`fft`	98.3 - 98.8	97.4 - 98.1
`go`	99.3 - 99.4	97.9 - 98.9
`gzip`	99.1 - 99.6	96.6 - 99.1
`jpeg`	99.0 - 99.8	97.8 - 99.7
`nsieve`	96.7	94.7
`shorten`	99.3 - 99.6	98.5 - 99.3
`water`	98.0 - 99.7	94.3 - 100
`fmin`	100	100
`rkf45`	100	96.8 - 98.4
`seval`	100	100
`solve`	100	100
`svd`	100	100
`urand`	92.3 - 100	87.5 - 100
`zeroin`	100	100
`applu`	95.4 - 98.8	91.8 - 98.3
`cg`	100	98.4 - 100
`doduc`	96.8 - 98.4	93.1 - 97.5
`fpppp`	95.9 - 99.0	91.9 - 97.8
`matrix300`	100	96.6 - 100
`mg`	99.4 - 100	99.2 - 100
`tomcatv`	88.2 - 98.0	73.3 - 96.7

Table 7.1 Percentage of problems solved optimally by LS

difficulty of the Hamilton Circuit problem, graph coloring, and K-satisfiability [18]. One of the most interesting results is the sharp correlation between average connectivity in graph coloring problems and difficulty of solving the problem. On many problems there seems to be a pattern of "easy-hard-easy". A problem instance with very few constraints tends to be easy, as is an over constrained problem. The problems somewhere in between tend to be harder. By harder we mean that finding a feasible solution typically requires exponential time.

Most of this work focuses on problems where there may or may not be a solution (*i.e.,* a decision problem.) The instruction scheduling problem is formulated as an optimization problem, as any algorithm attempts to find the shortest schedule (or best schedule by some other evaluation metric.) We would like to determine properties of scheduling problem instances that serve as a guide for more advanced scheduling algorithms. We saw an example of this in the last chapter where we used the solution space size estimate (computed during an initial pass of LS) to determine the number of trials for RBF.

In this section we explore what effect the parallelism available in a scheduling problem instance has on the difficulty of scheduling that instance. We define the available parallelism as the worst case scheduling time for the problem (complete serialization of all operations) divided by the critical path length of the problem (the shortest possible schedule). [10]

In some sense the available parallelism reflects how constrained the problem is. If the available parallelism is unity, then the DPG is a single thread, and there is only one possible solution. When the available parallelism is very high, virtually any choice made by a scheduler will yield an optimal solution. Although we would expect an incomplete algorithm to perform quite well when there is a lot of parallelism (though the solution space is large, it is replete with optimal solutions), it may be difficult to *prove* what the optimal schedule length is. The CPLEX solver uses a branch and bound technique to solve the scheduling problems we feed it. If the parallelism is high and the shortest schedule length estimate is actually lower than the optimal solution length, CPLEX will have to search virtually the entire (and very large) solution space to guarantee that it has found an optimal solution.

To further study the affect available parallelism has on scheduling algorithms we performed several experiments. In these experiments we generate scheduling problems

[10] This measure is not original. [73]

randomly. These problems are then scheduled with LS, RBF, and IR. If LS finds a solution whose length equals the minimum schedule length estimate, then we know LS has found an optimal solution. If RBF or IR find a better solution than LS then we know the LS solution is non-optimal. In the ambiguous cases, we feed the scheduling problem to CPLEX, and have it compute the optimal solution.

7.2.1 Scheduling random three address code on identical functional units

Scheduling problems are generated in the following way. Each problem has a fixed number of operations. In this first set of experiments we randomly assign a latency of 1-4 to each operation uniformly. Each operation is 3 address, and each register is also chosen randomly. The first operand of the first operation is always register number 0. Each successive register number is chosen randomly (uniform distribution) from the range 0 to the highest register number used so far plus 1. If an operand register is the result register of a previous operation, a true dependence exists between the operations. This process creates operations with zero, one, or, two incoming true dependence edges. We can prevent anti and output dependences by forcing the result register of each operation to have a register number one greater than the largest register used so far. We schedule problems of 10, 20, and 50 operations on a machine with one functional unit, and a machine with two identical functional units. All issue times are one cycle. For each block scheduled we record its parallelism and the schedule length for each of the techniques described above. (CPLEX is only run in the 10 operation case, because finding an optimal solution for larger problems requires too much time.) We then compute the percentage of basic blocks at a given level of available parallelism for which LS failed to find an optimal schedule. The stunning results are shown in Figures 7.1 and 7.2. These show the available parallelism on the x-axis and the percentage of blocks for which LS failed to obtain an optimal solution on the y-axis. Figure 7.1 shows the results for 10, 20, and 50 operations on a single functional unit. At each level of available parallelism, at least 1000 experiments were performed. In most cases over 2000 were performed. Figure 7.2 shows the results for 20 and 50 operations on two functional units. Note that for problems with only 10 operations, it is much more difficult to find problems with high enough parallelism to appear on the two functional unit figure.

The graphs clearly show that if there is a small amount of available parallelism in a problem LS can easily find an optimal schedule. Likewise, if the amount of available

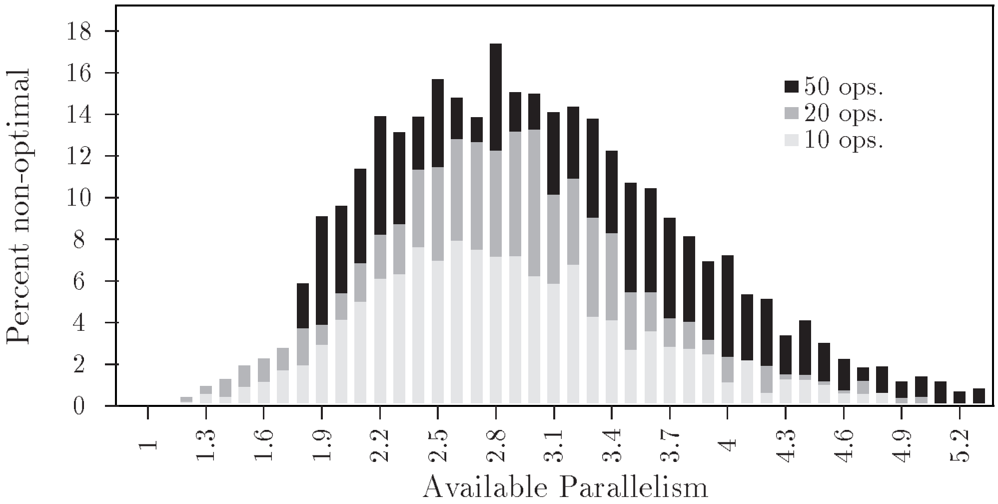


Figure 7.1 Percentage of problems for which LS failed to find an optimal solution as a function of available parallelism on one functional unit.

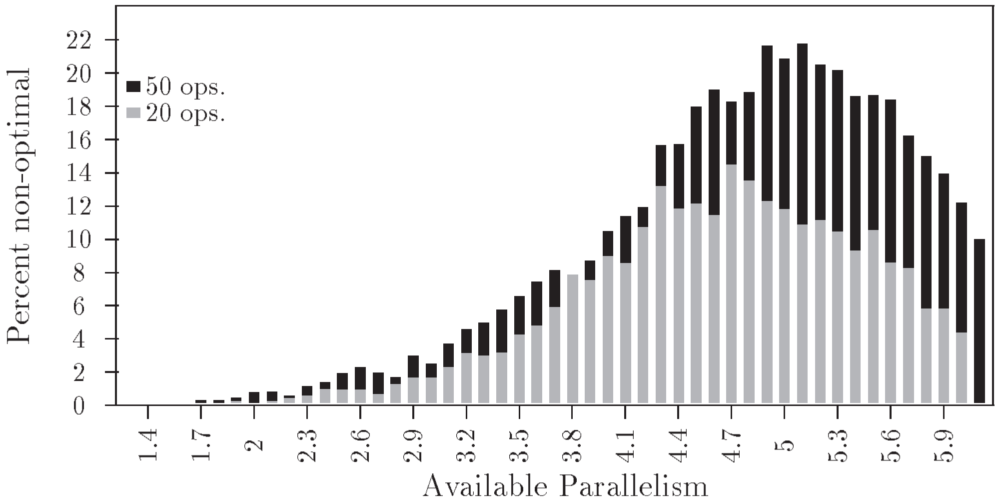


Figure 7.2 Percentage of problems for which LS failed to find an optimal solution as a function of available parallelism on two functional units.

parallelism is very high, an optimal schedule is often computed by LS. Somewhere in the middle of these two extremes LS struggles. The worst area for LS for a single functional unit seems to be somewhere between available parallelism of 2.8 and 3.0, and for two functional units somewhere between 4.7 and 5.1. We currently have no explanation for the precise shape of the curves or for the location of the maxima.

It should be noted that over the 85,000 plus problems scheduled on one functional unit, CPLEX *never* returned a schedule shorter than all of the other techniques applied to that problem. This shouldn't be too surprising since we do run IR for 2000 trials, and we know that a solution of optimal length exists in the solution space. It is interesting to look at the difficulty of finding an optimal solution in CPLEX as a function of available parallelism. Not surprisingly as the available parallelism increases, CPLEX must work harder to find the optimal solution. This is due in part to the formulation of the integer programming problem as described above. More parallelism implies fewer long chains of dependences which means we add more constraints to the problem to prevent operations from running at the same time on the same functional unit. A graph of available parallelism vs. the average number of iterations required by CPLEX to solve the problem optimally is shown in Figure 7.3

An interesting aside is to look at how often IR was responsible for finding the shortest schedule. The graphs in Figure 7.4 and Figure 7.5 show the percentage of problems for which iterative repair found a schedule shorter than all other techniques. (Not including the optimal solutions found by CPLEX of course.) The graphs show that IR is able to find some superior schedules over RBF when certain amounts of available parallelism are present. Recall that IR-rdf-bias was run for 2000 trials in these experiments.

The picture changes a bit when we exclude output and anti-dependences from the mix of problems. In the next experiment we scheduled problems with 20 operations on a single functional unit. There were no output or anti-dependences in the problem. The results are shown in Figure 7.6. The dark bars show the percentage of problems for which some technique found a solution better than LS. The lighter bars show the percentage of problems for which IR found a schedule shorter than all other techniques. While the shape of the curves is similar, it appears that problems without output or anti dependences are easier for LS to solve optimally.

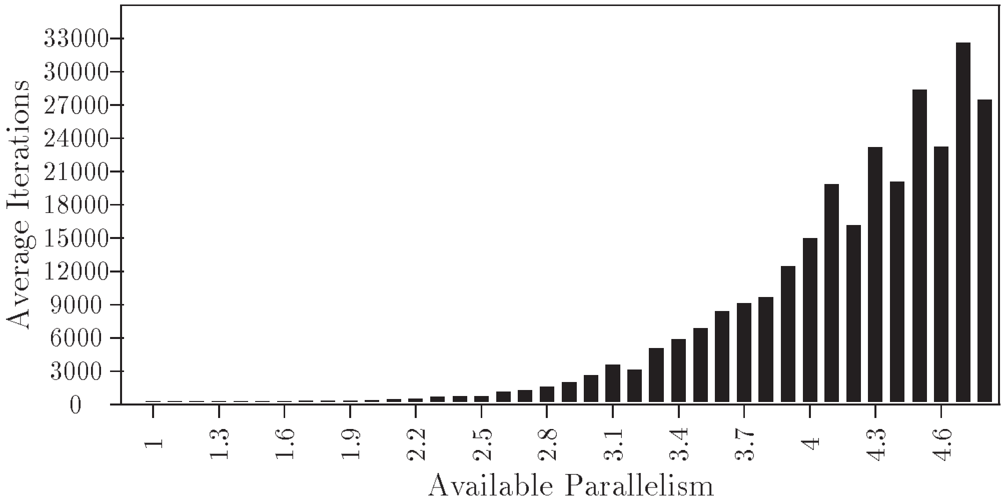


Figure 7.3 Average number of iterations CPLEX requires to find the optimal solution as a function of available parallelism. 10 operations per problem scheduled on one functional unit.

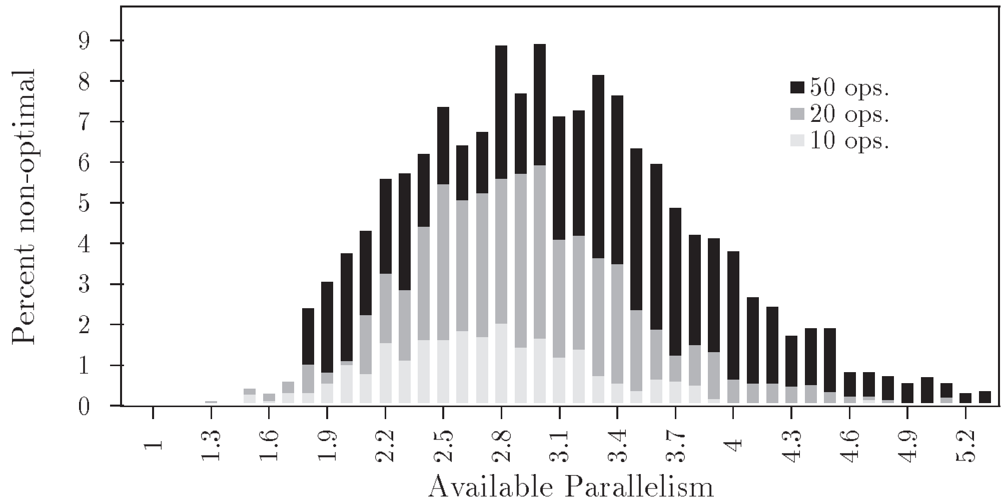


Figure 7.4 Percentage of problems for which IR-rdf-bias found a better solution than all other techniques as a function of available parallelism on one functional unit

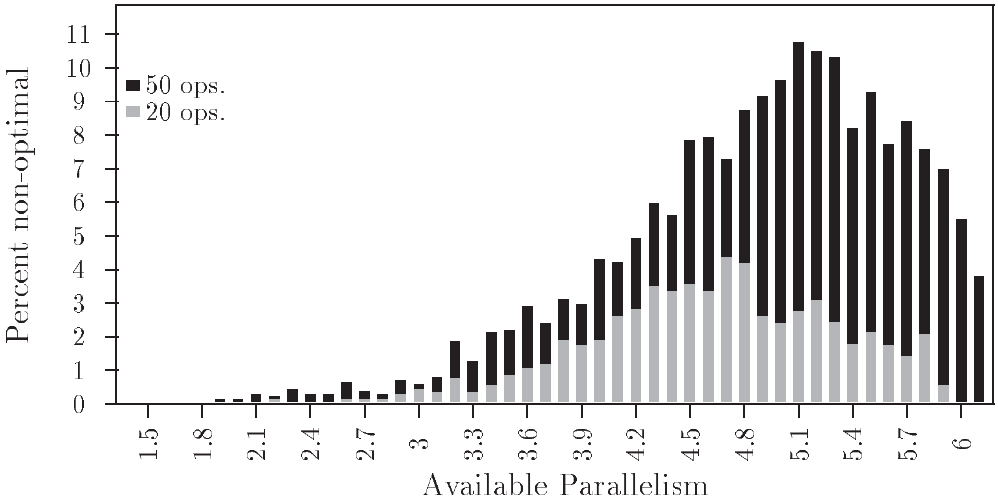


Figure 7.5 Percentage of problems for which IR-rdf-bias found a better solution than all other techniques as a function of available parallelism on two functional units

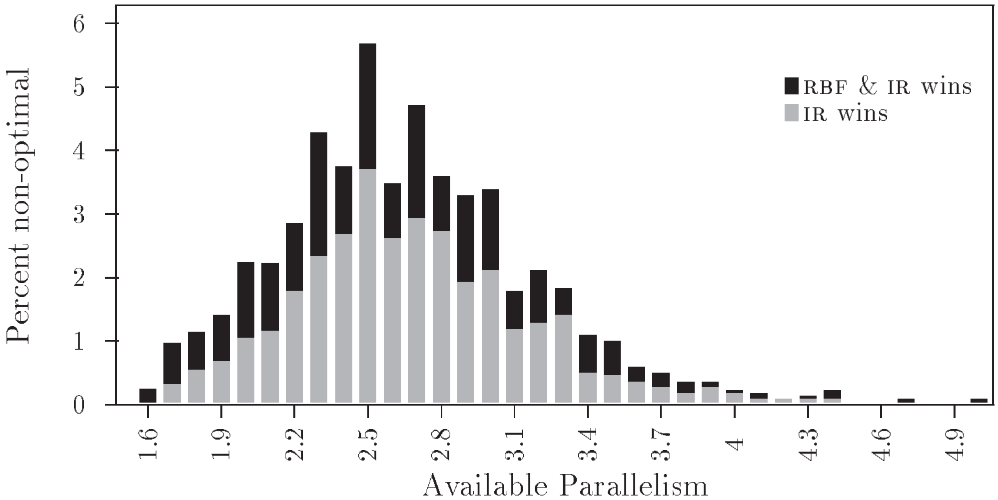


Figure 7.6 Percentage of problems with no anti or output dependences for which LS failed to find an optimal solution as a function of available parallelism on one functional unit

7.2.2 Scheduling random three address code on SVLIW

In these experiments we scheduled randomly generated problems on the SVLIW architecture. The operand and target registers were selected in the same manner as before. Operand selection was done differently. We looked at all our benchmarks and counted the operation types they contained. That is, we recorded the number of operations that could execute on each functional unit and their latencies. An operation type is simple a tuple containing functional unit type and latency. From this data we created a distribution of operation types. When we generated random problems, we selected the operation based on the distribution. In this way, we are able to generated random problems that better reflect real code.

Figures 7.7 and 7.8 show the results for scheduling problems with 20 and 50 operations on the SVLIW architecture. Note that it was difficult to generate problems with sufficient parallelism to find the peaks of the two curves. But clearly, quite a bit of parallelism is required in a problem instance before LS has difficulty finding the best solution. Also notice that the number of operations being scheduled didn't have as big an impact here. In fact at some parallelism levels, LS did worse on 20 operation problems than on 50 operation problems. There are several possible reasons for this. One possibility, is that we may not be running RBF or IR for enough iterations at the 50 op level to find all the optimal solutions. Another possibility is that the architecture simply lessens the impact of problem size on list scheduling.

7.2.3 Available Parallelism in real codes

Given the results of the previous sections, we can logically conclude that in order for LS to perform poorly a scheduling problem must contain a certain amount of available parallelism. In this section we look at the amount of available parallelism in our benchmark codes. Recall that earlier we mentioned that previous research has shown little ILP in basic blocks. The following data confirms that finding.

We computed the available parallelism on all 21,496 basic blocks from our benchmarks, when targeting the SVLIW architecture. Of those, 12,294 (roughly 57 percent) had available parallelism equal to 1. 108 had available parallelism over 6, with the largest recorded value at 69.3. Parallelism for all other blocks is shown in Figure 7.9. Available parallelism is again on the x-axis. The y-axis shows the percentage of blocks that had that level of available parallelism.

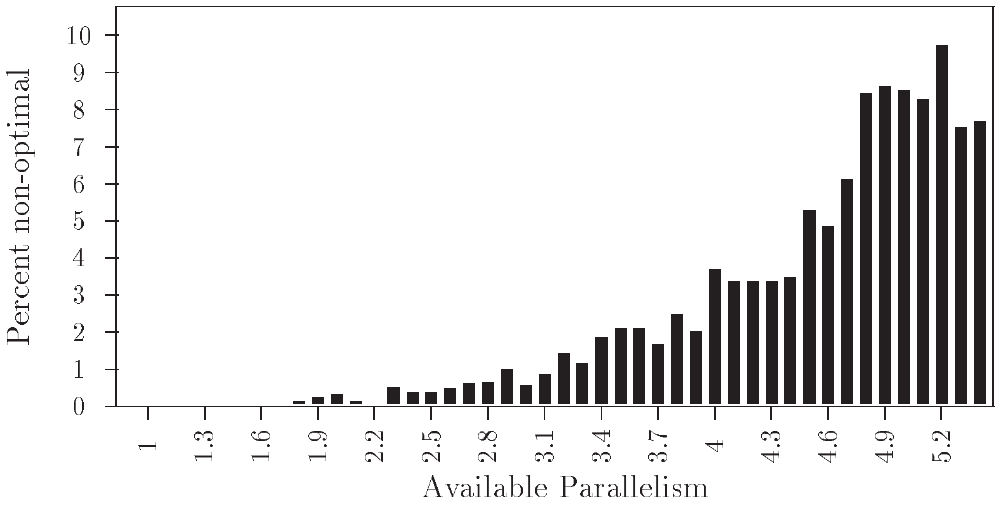


Figure 7.7 Percentage of problems for which LS failed to find an optimal solution as a function of available parallelism on SVLIW. Each problem with 20 operations.

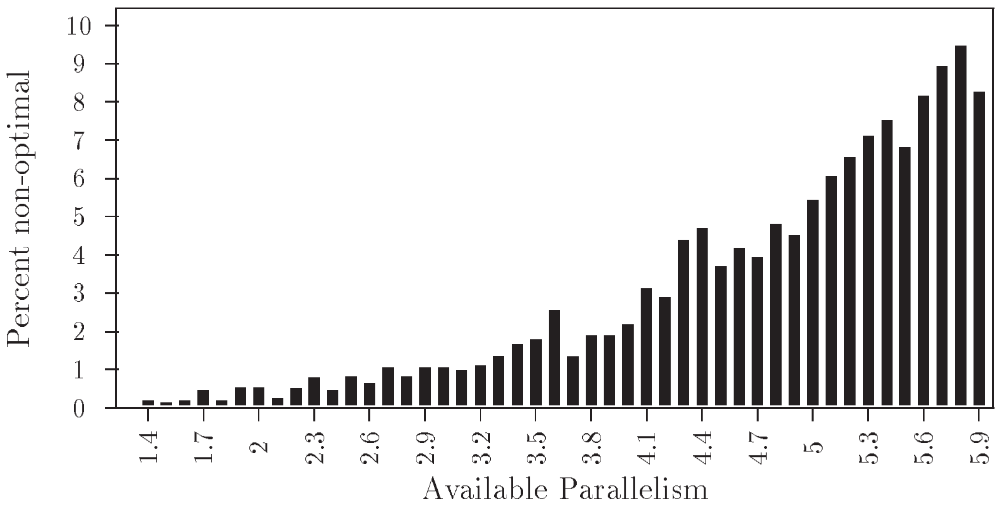


Figure 7.8 Percentage of problems for which LS failed to find an optimal solution as a function of available parallelism on SVLIW. Each problem with 50 operations.

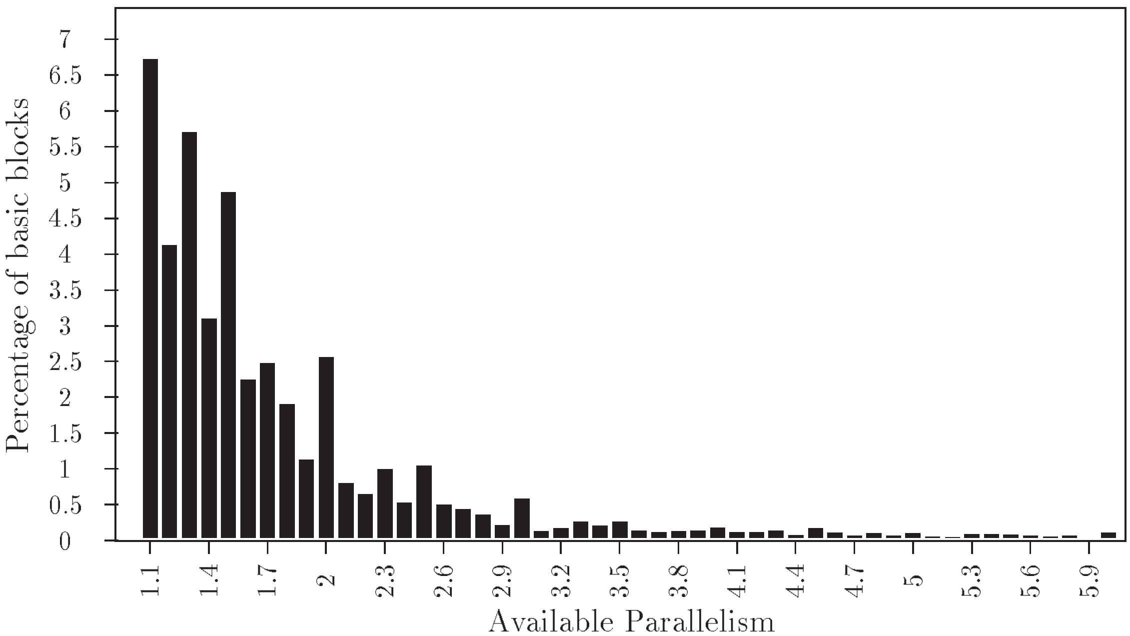


Figure 7.9 Available Parallelism in Basic Blocks on SVLIW

We also computed the available parallelism for all 12,100 EBB's from our benchmarks. 4,399 of the problems (about 36 percent) had available parallelism equal to 1. 80 had available parallelism values over 6, the largest being 65.6. Available parallelism values for all other blocks are shown in Figure 7.10. Notice that the EBB curve falls more slowly than the basic block curve. In general EBB's have higher amounts of available parallelism, although not as high as we might hope.

Finally, we computed the available parallelism present in all 10,001 dominator-paths from our benchmarks. Surprisingly, 5,115 of these (about 51 percent) had parallelism values equal to 1. Only 35, had parallelism values greater than 6 with the largest value being 47.2 Values for the rest of the dominator paths are shown in Figure 7.11. The results in Chapter 4 indicated that DP's contained (on average) more operations than EBB's. However, when we compare the amount of available parallelism present in the two scopes, EBB's generally have more parallelism. This is due to the greater number of restrictions on movement between operations within an EBB. This tends to lead to more edges in the DPG and reduced parallelism. The parallelism data also help to explain why the alternative scheduling techniques were generally more profitable on EBB's than on dominator-paths.

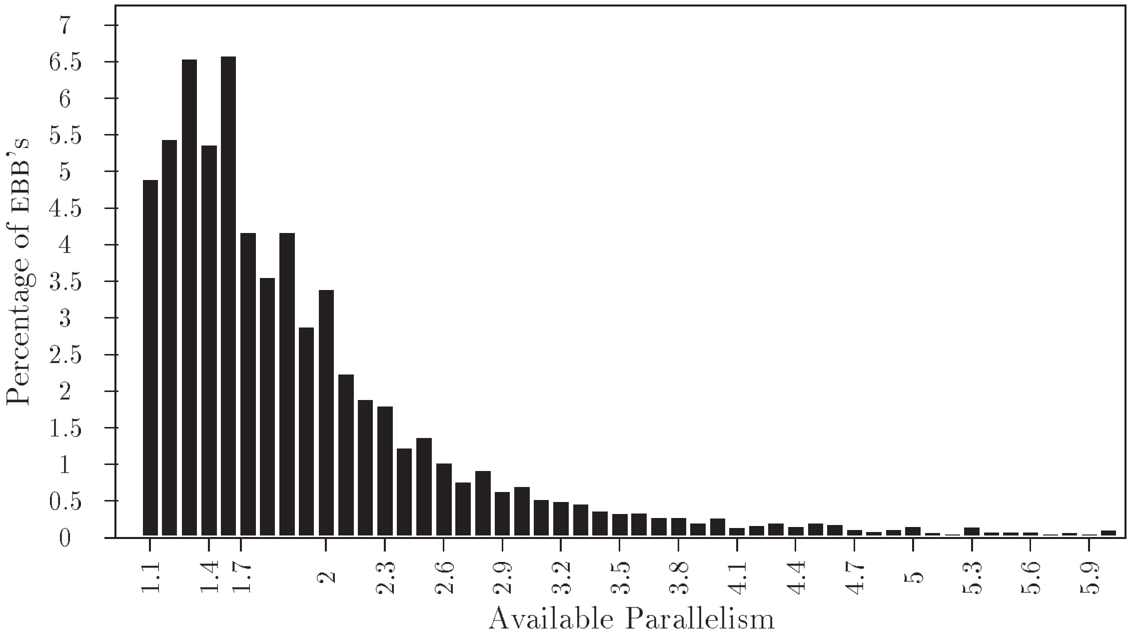


Figure 7.10 Available Parallelism in Extended Basic Blocks on SVLIW

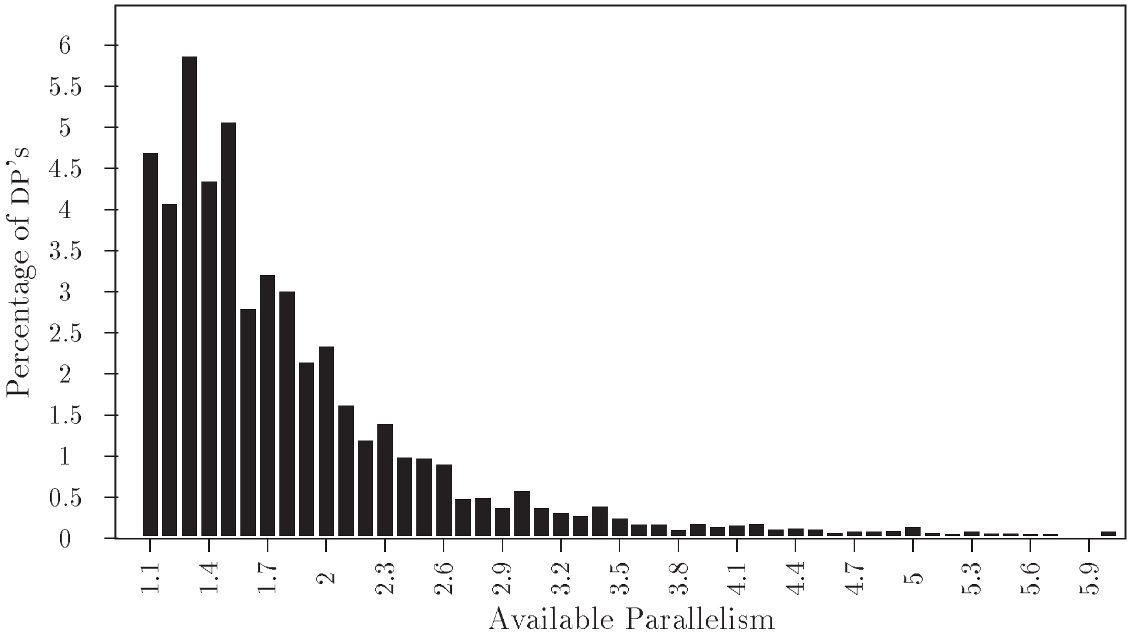


Figure 7.11 Available Parallelism in Dominator Paths on SVLIW

7.2.4 Other factors affecting scheduling

Not surprisingly other features of a particular scheduling instance can affect how well LS performs on that instance. For example simply looking at the number of edges in the DPG tells us a little. The graph in Figure 7.12 plots the number of edges (problems with ten operations) on the x-axis and the percentage of problems for which LS failed to find the optimal solution on the y-axis. Again we see a bit of the easy-hard-easy pattern emerging. The graph for 20 operations is similar with the peak around 29 to 31 edges.

We have tried to establish correlations based on other measures as well but with less success.

7.3 Some Conclusions

List scheduling is a powerful scheduling technique, able to produce near optimal schedules on a variety of problems and architectures. We have shown this by computing a lower bound estimate on the length of a particular schedule, computing alternate schedules with other algorithms, and computing an optimal solution using an integer programming approach. With these techniques at our disposal we were able to establish upper and lower bounds on list scheduling's optimality when scheduling real codes.

We have also discovered an interesting link between the available parallelism present in a scheduling problem, and the ability of LS to schedule the problem optimally. On simple toy architectures this correlation is quite regular. On more complicated architectures it is less regular. Until a scheduling instance has a sufficient amount of available parallelism we would expect LS to perform quite well.

That being said, an alternate scheduling approach bears considering. First we could schedule using LS and compare the result to a minimum schedule length estimate. If the schedule is optimal, no further work remains. If not, we can employ a more expensive technique like RBF or IR. The use of IR can be directed by constructing a parallelism vs. LS failure curve as done above, and comparing instances of scheduling problems to that curve.

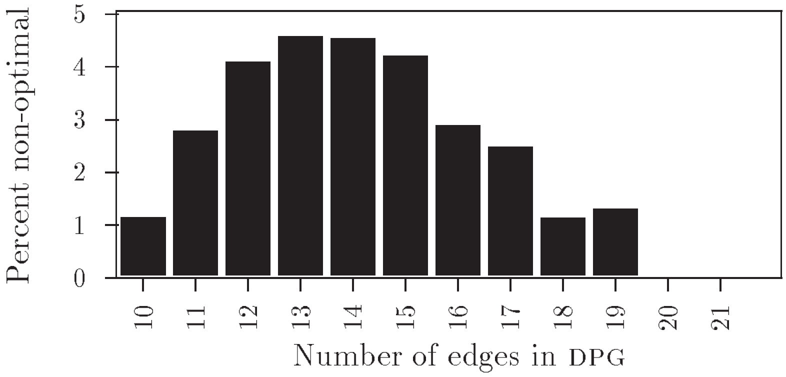


Figure 7.12 Percentage of problems for which LS failed to find an optimal solution as a function of the number of edges in the DPG. 10 operations on one functional unit.

Chapter 8

Scheduling and Register Allocation

A good body of work has developed on the interplay between register allocation and instruction scheduling. These two compiler phases are often antagonistic; an improved schedule may induce more spill code from the allocator, but a better allocation may unnecessarily inhibit instruction level parallelism. In this chapter we explore this tension. First we look at an example showing the tension between allocation and scheduling. Next we survey the prior work on integrating the two compiler phases. In section 8.3 we look at how to integrate register pressure concerns into an iterative repair scheduler, and finally some experimental results.

8.1 An example

Figures 8.2 through 8.4 show an example taken from the CRAIG paper that illustrates the allocation-scheduling tension [10]. Assume a machine with two identical functional units, where loads (LD) and stores (ST) take two cycles, and all other operations take one cycle. The machine has four general purpose registers. Figure 8.2 shows intermediate code that could be generated by the source code in Figure 8.1.

```
x_pos = x_pos + (x_vel * t);
y_pos = y_pos + (y_vel * t);
```

Figure 8.1 Source Code

We see that operations 7 through 11 are basically independent of operations 1 through 6, thus these two sets of operations can be run in parallel. This is reflected in Figure 8.3 which shows the result of first scheduling the code, and then performing register assignment. The resultant code executes in seven cycles and requires five registers (`r0` - `r4`). Since the machine has only four registers, spill code would need to be inserted.

```
 1: LD [x_pos] => r0
 2: LD [x_vel] => r1
 3: LD [t]     => r2
 4: MULT r1 r2 => r3
 5: ADD r0 r3  => r4
 6: ST [x_pos] r4
 7: LD [y_pos] => r5
 8: LD [y_vel] => r6
 9: MULT r6 r2 => r7
10: ADD r5 r7  => r8
11: ST [y_pos] r8
```

Figure 8.2 Original Intermediate Code

Figure 8.4 shows the result of performing allocation first and then scheduling. Notice the many new dependences created by the early allocation. Some important ones to note are the anti-dependences from operation 8 to 5, and from 7 to 6. This code takes nine cycles to execute but uses only three registers (`r0` - `r2`.) Note that different allocations can ultimately result in different schedules. (To see this, look at the presentation of the example in the paper [10].)

A technique that attempts to solve the two problems at the same time, may find a solution like that in figure 8.5. This schedule uses four registers and executes in eight cycles.

Typically the instruction scheduling and register allocation phases are completely distinct; neither phase pays any concern to the requirements or restrictions of the other. Thus, we have the phase ordering problem illustrated in the above example. Determining which phase to run first is largely dependent on the target architecture and the code being compiled. In the above example, if the target machine has sufficient registers we would be happy with the code in Figure 8.3. However if that schedule causes too much spill code to be generated, the code in Figure 8.4 may be preferable.

8.2 Previous work

There are several different approaches for looking at this problem in the literature. Several authors have looked at the phase ordering problem [9, 17, 42, 79].

```
LD [x_vel] => r1  |  LD [t]     => r2
LD [x_pos] => r0  |  LD [y_vel] => r6
MULT r1 r2 => r3  |  LD [y_pos] => r5
ADD r0 r3  => r4  |  MULT r6 r2 => r7
ST [x_pos] r4     |  ADD r5 r7  => r8
ST [y_pos] r8
NOP
```

(a) After scheduling

```
LD [x_vel] => r0  |  LD [t]     => r1
LD [x_pos] => r2  |  LD [y_vel] => r3
MULT r0 r1 => r0  |  LD [y_pos] => r4
ADD r0 r2  => r0  |  MULT r3 r1 => r1
ST [x_pos] r0     |  ADD r4 r1  => r1
ST [y_pos] r1
NOP
```

(b) After allocation

Figure 8.3 Code scheduled then allocated

There have been several methods developed for integrating the register allocation and instruction scheduling phases. The amount of integration varies from making one phase sensitive to the needs of the other to completely combining the two phases. We will not try to categorize the level of integration here, but rather give a brief overview of the different ideas.

Goodman and Hsu use a scheduler that is sensitive to register pressure [42]. If the number of available registers falls below a particular threshold while scheduling, a different scheduling algorithm is invoked. This algorithm schedules with the goal of minimizing register usage. Once the number of available registers is sufficiently high, the original scheduling algorithm is used. Another method they use, modifies the "width" of the DPG used by the scheduler, during register allocation. By using the DPG, the register allocator has better information about the dependences it will insert. After allocation, an "unmodified" scheduler is used.

In the work of Bradlee, *et.al.*, the method of Goodman and Hsu is compared to a complicated three pass system called RASE [9]. The first pass of RASE is a prescheduler that computes the length of the schedule for each basic block, for a range of register limits. This information is used by global register allocator (the

```
 1: LD [x_pos] => r0
 2: LD [x_vel] => r1
 3: LD [t]     => r2
 4: MULT r1 r2 => r1
 5: ADD r0 r1  => r0
 6: ST [x_pos] r0
 7: LD [y_pos] => r0
 8: LD [y_vel] => r1
 9: MULT r1 r2 => r1
10: ADD r0 r1  => r0
11: ST [y_pos] r0
```

(a) After allocation

```
LD [x_vel] => r1  |  LD [t]     => r2
LD [x_pos] => r0  |  NOP
MULT r1 r2 => r1  |  NOP
ADD r0 r1  => r0  |  LD [y_vel] => r1
ST [x_pos] r0     |  LD [y_pos] => r0
MULT r1 r2 => r1  |  NOP
ADD r0 r1  => r1  |  NOP
ST [y_pos] r1     |  NOP
NOP               |  NOP
```

(b) After scheduling

Figure 8.4 Code allocated then scheduled

second phase), when deciding what register limit to impose on each basic block. The final phase schedules each block, and makes *local* register assignments.

Norris and Pollock modify a Briggs style allocator [14] to use information from the DPG to limit the number of dependences introduced by the allocator [67] . They add edges to the interference graph to reflect all possible code reorderings, that satisfy the DPG for a basic block. They discuss several heuristics for adding these edges, and look at when to add them during the allocation process. Later on some of the interference graph edges may be removed to aid in coloring and reducing spilling, at the cost of adding additional edges to the DPG.

Pinter's work combines information from a modified form of DPG, and the register interference graph, to create the *parallel interference graph* [70]. First the transi-

```
LD [x_vel] => r0  |  LD [t]     => r1
LD [x_pos] => r2  |  LD [y_vel] => r3
MULT r0 r1 => r0  |  NOP
ADD r0 r2  => r0  |  LD [y_pos] => r2
ST [x_pos] r0     |  MULT r3 r1 => r1
ADD r2 r1  => r1  |  NOP
ST [y_pos] r1     |  NOP
NOP               |  NOP
```

Figure 8.5 A combined solution

tive closure of the DPG is taken, and all edges become undirected. Then, edges are added between operations that have a resource constraint imposed by the architecture. Edges in the complement of this graph, represent operations that could be run in parallel. This new interference graph takes into account all the possible parallelism in the program.

Brasier, *et. al.* have done work on CRAIG (Combining Register Assignment and Instruction Scheduling) [10]. First, CRAIG attempts to perform scheduling before register allocation (**late** register assignment). If spill code is required, CRAIG performs a round of register allocation followed by scheduling (**early** register assignment). If no spilling was required, CRAIG combines the two interference graphs by adding edges from the **late** interference graph to the **early** interference graph. Adding the appropriate edges makes that graph harder to color, but allows more parallelism to be exploited. This new graph is used for register assignment, and the resultant code is then scheduled.

Motwani, *et.al.* provide NP-completeness results for a simple instance of CRISP (Combined Register Allocation and Instruction Scheduling Problem) [64]. They also describe an algorithm for combining the problems when scheduling/allocating for a basic block, called the (α,β)-Combined Heuristic. They define a value called the *register rank* for each operation. This value is analogous to the list scheduling priority described above, except that it is used to minimize register pressure instead of schedule length. The register rank and a traditional list scheduling priority are summed together (weighted by the coefficients α and β), to create a new list scheduling priority value. The best α and β are determined experimentally. List scheduling is then performed in the usual way, using the new priority values. Register allocation is then performed on the block.

8.3 A Pressure Sensitive Iterative Repair Scheduler

The most effective register allocators are *global* in scope. That is, they allocate registers for an entire function at once. By contrast, scheduling is a local or at best a "regional" transformation. A scheduler typically schedules only a portion of the function at a time. We have looked at some other techniques that combine global allocation and scheduling. Now we look at how to make our iterative repair scheduler generate schedules that are more amenable to global register allocation.

We have explored several techniques for making our iterative repair scheduler sensitive to register pressure. Registers can be treated as an additional resource by the IR scheduler. However, accounting for the number of registers required at a given point in the schedule is more complicated than accounting for functional unit usage, especially in the context of a global allocator.

The first step in making the scheduler sensitive to register pressure is to estimate the register pressure within a given schedule. We say estimate, because there are several factors that make an exact calculation impossible. The value we will be computing is the maximum number of registers required to keep all currently live values in registers at any point in the schedule. Since we are scheduling over fixed program regions (basic blocks, EBB's, *etc*) we have to make some simplifying assumptions. For example it is possible that the global register allocator will spill some value around the the region we are scheduling over, lowering the register pressure in that region. We will have to assume that all values that are live on entry into the region are in registers, and likewise that all values live out of the region can be placed in registers. Additionally, the allocator has other tricks to reduce register pressure, including rematerialization [14] and register coalescing. (We obviate the second hurdle by performing register coalescing prior to scheduling.)

These issues aside, it is still possible to get a reasonable estimate of local register pressure. There is some preliminary work that must be done prior to scheduling. The next two sections describe some computations needed to estimate register pressure, and repair register pressure violations. Section 8.3.3 describes several techniques that the scheduler could use to reduce register pressure in a schedule. Finally, we look at some modifications that were made to the register allocator to facilitate allocating the scheduled code.

8.3.1 Finding last uses in the DPG

In order to accurately estimate register pressure we must be able to detect where in the schedule the *last use* of a particular register occurs. Toward that end, for each operation in the DPG and for each register used by the operation we compute whether that operation could be the last operation to use that register. For each operation we attach two bit vector sets containing register numbers. One we will call $lu(op)$ for last uses, and the other $plu(op)$ for *potential* last uses. If a register $r \in lu(op)$ for some operation, then no matter where op is placed in the final schedule that operation is the last to use r. This means that the register allocator is free to use that register for a subsequent definition. If a register $r \in plu(op)$ then that operation may or may not be the last use of r dependent on its placement in the schedule.

Figure 8.6 gives a DPG and the associated lu and plu sets for each operation. Actual operations are not shown. A register on the right side of the arrow indicates a definition, and a register on the left side indicates a use. Assume that `r1` is live into the block and `r4` and `r5` are live out of the block. Empty lu and plu sets are not shown. Notice operation 0 has no elements in either set. This is because it can never be the last use of `r1` since an operation that must be executed later (due to precedence constraints) also uses the register. Since there is no explicit ordering constraints between operations 1 and 2, either one of them could be the last to use `r2`. Thus, `r2` is placed in the plu sets for both. Since `r4` is live out of the block and `r3` isn't, only `r3` is placed in the lu set of operation 3.

We annotate operations with an additional set called the *live-here* set. Registers in this set are used on some path from the operation to a sink operation with no intervening definitions. All registers in the live out set of the block are in the *live-here* set of sink operations. We maintain two global sets called `seen_registers` and `potential_last_uses`. The algorithm in figure 8.7 computes the lu and plu sets for all operations. It calls the algorithm in figure 8.8 which recursively computes the *live-here* sets for the operations.

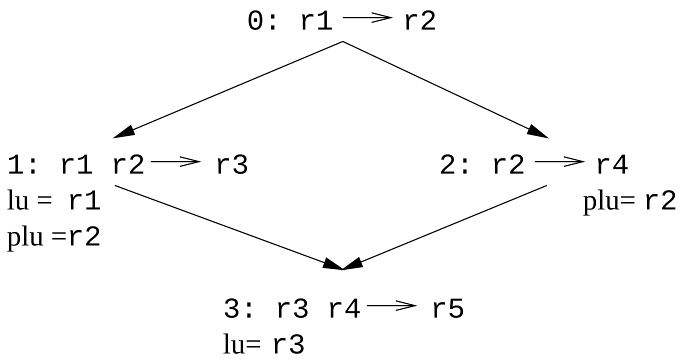


Figure 8.6 Last uses in the DPG

```
compute_last_use_sets

forall operations (op) in DPG
    if (op has no children in DPG)
        then add all registers in live out of block to op.live-here
        compute_live-here for op
    endif
endforall
forall operations (op) in DPG
    forall registers (r) used by op
        if (r ∉ op.live − here)
            then if (r ∈ potential_last_uses)
                then Insert r into op.plu
                else Insert r into op.lu
            endif
        endif
    endforall
endforall
```

Figure 8.7 Computing the last use sets

compute_live-here(op)

set $live - here = \emptyset$
forall children (*op'*) of *op* in DPG
    live-here = live-here $\cup op'.live - here$
    live-here = live-here $\cup$ all registers used by *op'*
    live-here = live-here $-$ all registers defined by *op'*
endforall
remove all registers defined by *op* from *seen_registers*
forall registers (r) used by *op*
    if ($r \notin live - here$)
        then if ($r \notin seen\_registers$)
            then insert r into *seen_registers*
            else insert r into *potential_last_uses*
        endif
    endif
endforall
op.live-here $= live - here \cup$ *op.live* $- here$
compute_live-here for parents *op'* of *op* if all children of *op'* finished

Figure 8.8 Computing the live-here sets

8.3.2 Computing register pressure in the schedule

Computing local register pressure in the schedule is easy. We maintain an array of livesets, one for each cycle in the schedule. Computing these, requires a simple backwards scan over the block. We start with the set of liveout registers. At each cycle in the schedule, we remove all registers defined by some operation in that cycle from the liveset, and add all registers used to the liveset. The register pressure at any point in the schedule is simply the size of the liveset at that point in the schedule.

Computing the liveset is straightforward, but we don't want to scan the entire schedule after every repair to recompute the livesets and register pressures. Thus we update the livesets as we move operations around the schedule. When we move an operation from time X in the schedule to time $Y > X$ we remove all registers defined by the operation from the corresponding livesets (including the liveset at time X, $X + 1$, *etc*, but not from the liveset for time Y.) Likewise we add all registers used by the operation to the corresponding livesets between time X and time Y.

8.3.3 Reducing register pressure with IR

Now that we are able to track register pressure changes during the iterative repair process, we would like to use this information to generate better schedules. There are several techniques one could use to direct the IR scheduler to find schedules with lower register pressure. The first three reduce pressure purely via code motion. The other suggests a spilling technique that the *scheduler* could use.

8.3.3.1 Modifying the final schedule evaluation function

In IR, once a schedule is free of constraint violations, we compare it to the "best" schedule found so far. For regular basic block scheduling we simply keep the schedule with the shortest static length. When scheduling EBB's IR-rdf-wbias uses a weighting technique to pick which schedule to keep. (Refer to section 6.2.3.) To encourage IR to find schedules with lower register pressure we redefine our notion of "best". Specifically we penalize schedules that have a maximum register pressure greater than the number of available registers.

How this penalty should be computed is not always clear. If the pressure in a particular block is too great, some value somewhere in the program will have to be spilled (or re-materialized). But that spill may not take place in this block, because the allocator is global. Even if spill code is inserted into the current block, it may be in some sense "free" on some architectures. For example, when scheduling on a VLIW machine, if there is little use of the memory functional unit in a given block, spill instructions might get scheduled in these holes not lengthening the overall length of the schedule. Of course even this spilling may cause undesirable effects such as more cache misses at runtime, so in general we would like to avoid it as much as possible. In summary, even though we know that register pressure is too high in a given block, it is difficult to measure how adversely that violation will affect the whole program.

8.3.3.2 Modifying the repair evaluation function

Another technique to aid IR in finding schedules with low register pressure, is to change the criteria for accepting a particular repair. Previously we described how to use evaluation functions such as RDF or schedule length to determine whether or not to accept or reject a particular repair. Now we would like to add concerns over register pressure to that criteria.

One method that could be used is to refuse any repair that increases the highest pressure in the schedule. Another approach would be to reject a repair that raises the highest pressure in the schedule above the number of available registers. Of course to avoid local minima, some "worse" schedules would still need to be kept.

8.3.3.3 Repairing register pressure violations

An even more aggressive approach is to force the scheduler to specifically target and repair areas of high pressure in the schedule. Think of registers as just another machine resource. If too many registers are being used we have a constraint violation and we attempt to repair it. Of course registers are in some sense a "soft" resource. That is a schedule is still valid even if it has register pressure violations. The register allocator will take care of those.

Here we present one approach to repair register pressure violations through code motion. Assume that at time t in the schedule, the register pressure is greater than the available number of registers. We must find some operation to move that can reduce the register pressure at point t. We have computed the lu sets and the plu sets for each operation. Consider an operation whose lu set is smaller than the number of registers defined by the operation. (A load immediate is an example of such an operation. It uses no registers so the size of its lu set is zero, but it always defines one register.) Assume this operation is scheduled at time $t_0 < t$. If we move this operation *after* t then we are guaranteed to have reduced the register pressure at time t. The set of operations where the set $lu \cup plu$ is smaller than the number of registers defined by the operation, will cause register pressure to be lowered at t when they are moved past t. Operations whose lu set (now ignoring plu) is smaller than then number of registers defined by the operation *may* reduced register pressure when moved past t.

More details on one way to implement this repair operation are presented in the experimental section.

8.3.3.4 Local Spilling

If register pressure can not be sufficiently reduced via code motion techniques, the scheduler could do some amount of local spilling. For example, consider the set of registers live into a block, live out of the block, and not used or defined in the block. We call these pass-through registers. If we store n of these at the beginning of the block and reload them at the end of the block, we have reduced the intervening register pressure by n. It is possible that this spill code is free, if the scheduler can find holes in the schedule to fill the spill operations with.

8.3.4 Considerations for the Register Allocator

Several important modifications had to be made to our register allocator to facilitate scheduling after register allocation. The major issue is to assure that the allocator does not generate any *circular* dependences among operations in a single instruction. For example consider the following ILOC instruction prior to register allocation:

```
iADD r1 r2 => r3 | iADD r4 r5 => r6
```

Assume that `r1` and `r4` are no longer live after this instruction. It is possible that the register allocator could produce the following:

```
iADD r1 r2 => r4 | iADD r4 r5 => r1
```

If we try to schedule this allocated code, the scheduler will fail because of the circular anti-dependences between the two operations. This situation could be handled in the scheduler by forcing all operations involved in a circular dependence to be scheduled during the same instruction, but this would be difficult and limits the flexibility of the scheduler. A far more desirable solution would be to prevent the allocator from ever producing this circular dependence. We do this by modifying the way in which the allocator builds the interference graph. In the above example, the allocator ensures that `r1` and `r2` interfere with `r6`, and `r4` and `r5` interfere with r3. Thus the second piece of code will never be generated by the allocator, but the following code (using the same number of registers) could be:

```
iADD r1 r2 => r1 | iADD r4 r5 => r4
```

8.4 Experimental Results

We have performed some experiments to determine the potential profitability of reducing local register pressure via scheduling. The first question to answer is one of opportunity. That is, how many blocks have alternate schedules with lower register pressure. Next, we must see how profitable reducing local register pressure is in the global scheme of things. That is, given that we've reduced *local* register pressure throughout a program, does this reduce the size or speed of the program after *global* register allocation and scheduling of the spill code. The next section describes our pressure sensitive iterative repair scheduler. Then, we describe our various experiments and present our experimental results.

8.4.1 Using IR to reduce register pressure

This version of our iterative repair scheduler has been made sensitive to register pressure. To try to improve the code we use several techniques to reduce the register pressure in scheduled code. These have been discussed above at a high level. Now we present the details of the scheduler used in these experiments.

First we schedule with LS and record the length of the resultant schedule and its highest register pressure. Next, we do a pass of 200 iterations of IR-rdf-bias. However, instead of keeping the schedule with the shortest length, the final schedule is the one with the lowest register pressure. (Ties in the register pressure metric are broken by looking at schedule length.) Next, we perform another 200 iterations of IR-rdf-bias, but instead of using an ASAP schedule as the starting point for IR, we use an "as late as possible" initial schedule. We have found, that some schedules will have lower register pressure when operations are clustered around the end of the schedule instead of the beginning. The as late as possible starting point tends to encourage that effect. Again we use register pressure as the criteria for finding the "best" schedule.

The final technique is also based on IR-rdf-bias, but with several modifications. We call it IR-rp. Instead of keeping the shortest schedule as the final schedule, we keep the schedule with the lowest register pressure and break ties by looking at schedule length. We also modify the criteria for keeping or discarding the result of an individual repair. A repair that increases the maximum register pressure in the program is always rejected (modulo the five percent acceptance rate of any repair.) We also perform repairs that attempt to reduce register pressure. If a cycle is detected where the register pressure is above some threshold (*e.g.* the number of machine registers) we attempt to move an operation that will lower the register pressure at that point. Prior to scheduling we compute the set of operations that, if moved forward in the schedule, *could* lower register pressure. This is the set of operations that define more operations than the size of their *lu* set. When a "pressure repair" is performed, we randomly select one operation from this set that is currently scheduled before the region of high pressure, and move it to the cycle after the high pressure point. We then move all its successors in the DPG. This selection is weighted just like the biased selection for repairing functional unit conflicts. We try to move operations with low priority in the hopes that the schedule will not be lengthened to much. Currently we set the threshold value very low (lower than the number of registers in the machine), to try to reduce register pressure regardless of the hardware resources. This is because reducing register pressure can be beneficial even in blocks where the pressure is less than the number of machine registers, especially when values must be spilled due to other blocks. (Again, this is because a *global* allocator is used after the scheduler.)

Of course there may be no way to reduce register pressure below the threshold simply by moving operations. When such a situation occurs, the repair process is complete when all resource conflicts are removed from the schedule. It is also possible that so many operations get moved, that the length of the schedule becomes longer than the serialization of the operations. If such a schedule occurs, it is thrown out and the process restarts with the next trial.

To summarize, we run four versions of the scheduler: LS, 200 iterations of IR-rdf-bias using register pressure as the metric, 200 iterations of IR-rdf-bias starting with an ALAP schedule, and 200 iterations of IR-rp. It should be noted that the number of IR iterations performed is on a sliding scale; a fewer number of trials will be performed on extremely large blocks to make scheduling times more reasonable.

8.4.2 Pressure Reduction Possibilities

We ran the four schedulers as found above on our suite of benchmarks and kept track of how many blocks had alternate schedules with lower register pressure than the schedule produced by LS. For these experiments we simply keep track of the schedule with the lowest pressure returned by any of the four methods We are not concerned with the length of the final schedule. These experiments were done on the VLIW and C60 architectures scheduling over basic blocks and EBB's. The results are summarized in Table 8.1. Notice that there are quite a few opportunities to reduce register pressure with alternate schedules. The opportunities on EBB's are even greater.

8.4.3 Overall Improvements

Now we look at whether the opportunities described above can be translated into real code improvements. Toward that end, we perform additional experiments. Our baseline for comparison is LS followed by global register allocation followed by LS again to schedule the spill code. We have found that this technique virtually always results in better code than performing allocation first followed by scheduling. [11]

After scheduling with the four techniques described above, we look at the schedule lengths and the register pressures of the four returned schedules. If all the lengths are the same, we choose the schedule with the lowest register pressure. Otherwise, we choose the schedule with the lowest register pressure, provided the length of that schedule is no more than five percent greater than the schedule produced by LS. Ties are broken in favor of the LS schedule.

We compared using our pressure sensitive scheduler to simply using one pass LS. After scheduling, the code is passed to the register allocator, and then back to the scheduler for a final pass of LS that schedules any introduced spill code. Tables 8.2 and 8.3 present results when scheduling over basic blocks and EBB's on VLIW and Table 8.4 on the C60. We show reductions in dynamic and static instruction count when the allocator has 16 or 32 registers available on VLIW, and 32 registers on C60. (The *real* C60 has 32 general purpose registers.) The numbers shown represent the best of two runs of the pressure sensitive iterative repair scheduler using different random number seeds. A single register set is assumed. Note that several of the benchmarks require more than 16 registers, thus they can not be included in that particular experiment. These are noted in the table with an asterisk.

Notice some dramatic reductions in running time and code size for both architectures. You will also notice a few times when the pressure sensitive scheduler did not lead to overall program improvements. Note that in all the above cases local register pressure was reduced by the pressure sensitive scheduler (although not necessarily

[11] This corresponds with the findings of Sweany and Beaty [79].

	VLIW		C60	
Benchmark	BB	EBB	BB	EBB
`adpcm`	4.9	25.5	4.9	21.6
`clean`	13.6	30.4	13.1	29.5
`compress`	18.9	42.6	21.6	44.7
`dfa`	1.8	37.3	1.5	50.0
`dhrystone`	29.0	45.2	27.4	40.5
`fft`	17.0	40.0	16.2	39.4
`go`	7.3	34.6	7.5	35.5
`gzip`	11.8	33.5	12.4	30.7
`jpeg`	14.8	40.7	14.5	40.0
`nsieve`	13.3	31.6	10.0	26.3
`shorten`	9.1	31.5	11.0	32.7
`water`	22.8	41.7	23.1	43.2
`fmin`	2.2	17.4	2.2	13.0
`rkf45`	19.6	48.4	19.6	46.8
`seval`	23.5	45.5	20.6	45.5
`solve`	18.3	27.5	18.3	27.5
`svd`	18.9	40.6	20.8	41.6
`urand`	7.7	12.5	7.7	12.5
`zeroin`	3.6	33.3	10.7	33.3
`applu`	20.2	35.4	18.5	32.3
`cg`	24.7	46.8	18.0	38.7
`doduc`	30.7	56.1	29.4	55.1
`fpppp`	18.5	46.6	17.2	44.8
`matrix300`	22.0	70.0	22.0	70.0
`mg`	18.6	23.5	19.3	24.3
`tomcatv`	23.5	46.7	23.5	43.3

Table 8.1 Percentage of problems for which an alternative scheduling technique discovered a schedule with lower register pressure than LS

	Dynamic		Static	
Benchmark	16 reg	32 reg	16 reg	32 reg
adpcm	0	0	0	0
clean	.18	0.01	0.11	0.04
compress	40.6	7.12	7.97	0.9
dfa	0	0	0	0
dhrystone	1.05	-0.0	0.17	-1.25
fft	4.97	5.38	4.22	1.21
go	*	0.02	*	0.05
gzip	-0.31	0.21	0.26	0.02
jpeg	4.31	-0.06	0.83	0.23
nsieve	0	0	0	0
shorten	0.79	0.04	1.6	0.62
water	5.87	2.52	5.85	7.48
fmin	-2.32	0	-2.39	0
rkf45	*	-0.00	*	0.04
seval	5.01	0.36	4.98	1.1
solve	0.96	0	0.83	0
svd	0.72	0.34	1.21	1.24
urand	-2.31	0	0	0
zeroin	0.70	-0.03	2.83	-0.17
applu	*	-2.94	*	-0.45
cg	0.00	-0.00	0.7	-0.09
doduc	*	5.17	*	4.61
fpppp	*	2.5	*	0.8
matrix300	0.01	0	3.27	0
mg	43.73	25.10	13.98	6.54
tomcatv	0.19	13.08	2.46	9.35

Table 8.2 Overall improvement found by using register pressure sensitive IR scheduler on basic blocks targeting SVLIW

	Dynamic		Static	
Benchmark	16 reg	32 reg	16 reg	32 reg
adpcm	0.34	0.36	0.27	0.3
clean	4.91	0.08	0.90	-0.27
compress	6.07	-0.05	3.14	-0.33
dfa	-0.25	-3.59	-1.10	-2.88
dhrystone	1.1	0.28	0.53	-0.18
fft	-1.34	5.14	0.96	0.96
go	*	-0.30	*	-0.83
gzip	0.52	0.04	-0.05	-0.68
jpeg	3.42	-0.38	0.50	0.15
nsieve	-0.00	-0.00	0	-1.08
shorten	-0.56	-0.55	0.49	0.10
water	4.46	1.62	8.33	7.29
fmin	4.49	4.98	-3.18	0.19
rkf45	*	0.38	*	0.04
seval	2.64	0.18	3.63	1.25
solve	2.47	-0.06	3.14	-0.08
svd	3.03	-0.75	2.55	-1.34
urand	-1.54	0.53	-0.50	4.81
zeroin	19.92	-0.25	-1.20	-0.42
applu	*	-3.71	*	-1.99
cg	0.01	-0.00	1.46	0.10
doduc	*	5.76	*	4.18
fpppp	*	-1.8	*	-0.4
matrix300	0.00	-0.00	-0.39	-0.47
mg	41.34	30.68	15.33	8.66
tomcatv	4.12	11.96	1.36	4.38

Table 8.3 Overall improvement found by using register pressure sensitive IR scheduler on EBB's targeting SVLIW

	Basic Blocks		EBB's	
Benchmark	Dyn	Sta	Dyn	Sta
`adpcm`	0	0	0.00	0
`clean`	0.01	-0.03	0.14	0.37
`compress`	3.10	0.79	-0.16	0.26
`dfa`	0	0	-2.00	0.75
`dhrystone`	0	0.8	0.00	0
`fft`	8.65	0.34	0.37	0.84
`go`	0.00	0.03	0.11	0.65
`gzip`	-0.20	0.09	-0.01	0.53
`jpeg`	0.00	-0.09	-0.2	0.5
`nsieve`	0	1.39	0.00	2.74
`shorten`	0.04	0.16	-0.34	7.2
`water`	1.06	1.98	1.06	2.13
`fmin`	0	0	-0.13	1.2
`rkf45`	0	0.82	-0.08	0.39
`seval`	0.56	-1.00	1.09	0.97
`solve`	0	0	-0.05	0.53
`svd`	0.33	0.77	-0.17	2.51
`urand`	0	0	0	0
`zeroin`	0	0	-0.47	-1.91
`applu`	0.65	1.50	0.31	1.7
`cg`	0	0.45	0.00	1.35
`doduc`	0.97	2.01	1.98	2.35
`fpppp`	-2.49	0.43	-0.81	0.26
`matrix300`	0	-2.51	-0.04	-2.43
`mg`	23.34	1.65	26.55	0.83
`tomcatv`	18.53	1.65	10.39	-2.21

Table 8.4 Overall improvement found using register pressure sensitive IR scheduler targeting C60

in as many blocks as indicated in Table 8.1 due to the added constraints placed on schedule length growth.) However these reductions in local register pressure may not always yield overall reductions in running time or code size. One reason is that we may accept a longer schedule for a block if the register pressure is less. If this block did not contribute to increased spilling when pressure was higher, then the sensitive scheduler will lose due to increased schedule length. It should also be mentioned our register allocator is aggressive and *very* good. Rematerialization alone reduces a lot of the potential benefit of a pressure sensitive scheduler since some of the code motion done by the scheduler can be done by the allocator through rematerialization of values.

8.5 Summary

In this chapter we explored the tension between instruction scheduling and register allocation. We described how to make an iterative repair scheduler sensitive to register pressure. We designed an iterative repair scheduler that made dramatic efforts to reduce local register pressure. Using this tool we saw many opportunities to reduce local register purely via the code motion performed by the iterative repair scheduler. When placed in the context of the whole compiler with an aggressive global register allocator, some dramatic code improvements were observed when employing the pressure sensitive IR scheduler instead of the simple list scheduler.

Chapter 9

Contributions and Future Work

In this final chapter we present the contributions of this work and suggest directions for possible future work.

9.1 Non-local scheduling

There seems to be little reason why most production compilers couldn't *at least* perform EBB scheduling as described in Chapter 4. The technique requires relatively little data flow information, is easy to reason about, is easily designed to run quickly, and produces fair improvements over basic block scheduling. We hope this work will "raise the bar" for what defines the minimal scope of instruction scheduling. Constructing extended basic blocks by grouping large blocks together (the *size* heuristic) seems to work quite well in practice.

Dominator-path scheduling can be more profitable than extended basic block scheduling but requires longer compile times, and is more difficult to implement. Trying both extended basic block scheduling and dominator-path scheduling may prove profitable.

We have shown that non-local scheduling can indeed reduce the running time of programs *without* increasing the size of the program. On certain architectures we can achieve both reductions in running time and code size using extended basic block scheduling and dominator-path scheduling.

That being said there is still other work to be done in this area. It would be useful to compare the techniques described in Chapter 4 to a trace scheduler designed to not increase code size.

Clearly on the C60 style architecture, there is room for improvement in terms of code size. Due to the unique instruction packing technique used by this architecture, our non-local scheduling techniques, though adding no additional operations, resulted in larger code sizes in most cases. Scheduling for reduced code size on this type of architecture requires more study.

9.2 Rbf scheduling

We have studied scheduling problem instances in both a forward and backward way. Additionally, we have looked at adding an element of randomness to a list scheduler. That is, by breaking ties in a simple list scheduling priority heuristic randomly, scheduling the same problem several times, and keeping the best schedule, we were able to achieve small improvements in the final schedule. This technique appears even more promising on large problem instances, complicated architectures, and scheduling scopes larger than a basic block. Using profile directed and non-profile directed weighting schemes when scheduling over EBB's and DP's can further improve the running times of scheduled code.

Although the improvements shown were modest, the technique can be made to be very low cost in terms of compile time. The cost of scheduling a problem instance 2 or 3 times forward and backward should be very small in real compilers, especially when compared to the time required to build the DPG.

9.3 Ir scheduling

We have applied a novel scheduling technique from the AI community to the problem of instruction scheduling in a compiler. This technique is called iterative repair. We have described the evolution of our iterative repair algorithm, and believe we have developed a powerful alternative instruction scheduling technique. This technique has several benefits over a single pass constructive technique like LS. First, since IR always has a complete schedule available, it can make more intelligent scheduling decisions. Second, as more compile time is thrown at the problem, IR can generate better and better solutions.

Iterative repair had difficulty finding solutions to real instruction scheduling problems that were better than those generated by LS. At the basic block level, improvements were virtually unnoticeable. As we moved to more complicated architectural models, including architectures with "intersecting" functional unit types and multiple pipeline models, IR faired somewhat better. Ir showed the most promise when scheduling over extended basic blocks using weighting techniques based on block level profile information. Our largest improvement over list scheduling was 4.4 percent, scheduling over EBB's for the PPC750 on the `fpppp` benchmark.

While IR is expensive, it clearly needn't be applied to every instruction scheduling problem. Cheaper techniques such as LS and RBF can be applied first. If they fail to find a schedule of sufficient quality, then IR (or even a complete technique) can be applied. Measuring other features of the particular scheduling problem (see next section) may yield more insight as to whether a more expensive technique is justified.

9.4 The Near Optimality of LS

We have used scheduling problems from real codes and hundreds of thousands of randomly generated scheduling problems in order to gain understanding of the power of list scheduling. Testing the optimality of schedules produced by LS was done in several ways: comparing the length of the schedule to a lower bound, using the alternate scheduling methods described above, and finding the optimal schedule length by formulating the problem as an integer programming problem and feeding that problem to the CPLEX solver. As evidence of a widely held belief, we have found that list scheduling routinely finds an optimal length schedule. We have confirmed Fisher's results, that as scheduling problems get larger, LS is less likely to find an optimal solution.

We have uncovered an interesting link between the amount of parallelism available in a scheduling problem and the probability that the problem will be scheduled optimally by LS. Typically, when there is very little parallelism available in a problem and when the available parallelism is very high, LS has little difficulty finding an optimal solution. When scheduling three address code on a single processor a fairly sharp "peak" was observed when graphing the amount of available parallelism in a problem versus the probability that LS would find a suboptimal solution. This peak, and the shape of the curve changes for different architectures. We have also observed that the available parallelism in real codes is typically low even when looking at extended basic blocks and dominator-paths. We believe this is one reason that LS is able to perform so well on real scheduling problems.

Directions for future work in this area are numerous. To better understand the "difficulty" of instruction scheduling problems, one idea may be to try other complete algorithms on the problem. We used CPLEX to solve an integer programming version of scheduling problems. The only pattern we detected with this method was that as the parallelism increased the difficulty of solving the integer programming problem also increased. This is not surprising since we must add many equations when a problem is very parallel to prevent tasks from executing on the same functional unit at the same time. Another approach, (*e.g.* constraint satisfaction) may prove more fruitful.

Ultimately, when given a scheduling problem, one would like some reasonable idea about what sort of algorithm to apply to the problem. One approach would be to construct a neural net for instruction scheduling problems. Once could train the net with various features of instruction scheduling problems (or possibly whole problem instances) and results of a given algorithm on each instance. An interesting experiment given such a neural net would be to determine how well the net can predict whether list scheduling can optimally solve a particular instruction scheduling problem instance.

9.5 Integration with Register Allocation

We have used our alternative scheduling techniques to explore the opportunities for a scheduler to reduce local register pressure. On average, we found that we could reduce local register on 16 percent of basic blocks and 38 percent of EBB's over list scheduling. This shows that there is plenty of room for an intelligent scheduler to reduce local register pressure.

However many of these opportunities do not always lead to superior overall code. One problem is that the scheduler is reducing local register pressure, while most register allocators work globally. Thus reducing local pressure in a block may have little effect if the global allocator spills values around that block. A good graph coloring register allocator that makes use of rematerialization can also find some of the same opportunities for reducing register pressure that can be found by an intelligent scheduler. These issues aside, we did see some improvements (some dramatic) in overall code quality by using our pressure sensitive IR scheduler. As the data shows, trading off reduced register pressure for longer schedules can be tricky. While the technique often wins, some slower code may be generated.

There are even more parameters to play with when using the IR scheduler to reduce register pressure. One could change the way reduced register pressure is traded off for longer schedules. The scheduler may also be able to introduce some spill code itself. This could be especially useful when the scheduler determines that the spill code it inserts will not lengthen the overall schedule. In essence the scheduler would be doing *local allocation* and the the global register allocator would merely perform register assignment. It is difficult to predict how well this would work in practice.

Appendix A

Hardware Models

Number of functional units: 5
Number of issue slots: 5
All issue times for all operations are 1
Memory unit: All byte (`b`) and word (`w`) loads and stores to memory require two cycles. All integer (`i`) and floating point loads and stores require four cycles. All double (`d`) loads and stores require eight cycles.
Branch unit: All branch, jump subroutine, and return operations take one cycle.

Integer and Floating Point units											
2 Integer Units						Floating Point Unit					
op	lat	op	lat	op	lat	op	lat	op	lat	op	lat
FRAME	1	iPOW	5	lXOR	1	fADD	4	f2f	1	fCMPle	3
iABS	2	i2c	1	lNOT	1	fSUB	4	f2d	1	fCMPge	3
iLDI	1	i2q	1	iMOD	8	fNEG	4	f2d	1	fCMPlt	3
iSIGN	2	iSL	1	iMIN	8	fMUL	7	d2i	1	fCMPgt	3
iADD	2	iSLI	1	iMAX	8	fDIV	14	d2f	1	dCMPeq	6
iSUB	2	iSR	1	iCMP	1	dADD	8	d2d	1	dCMPne	6
iNEG	1	iSRI	1	iCMPeq	2	dSUB	8	f2i	1	dCMPle	6
iMUL	5	lSHIFT	1	iCMPne	2	dSIGN	2	fLDI	1	dCMPge	6
iDIV	8	lSL	1	iCMPle	2	dLOG	20	dLDI	1	dCMPlt	6
iADDI	2	lSLI	1	iCMPge	2	dEXP	20	dPOWi	20	dCMPgt	6
iSUBI	2	lSR	1	iCMPlt	2	dNEG	8	dTRUNC	3		
GT	1	lSRI	1	iCMPgt	2	dMUL	14	dMAX	20		
EQ	1	lAND	1	i2i	1	dDIV	28	dMIN	20		
NE	1	lOR	1	i2f	1	fCMP	3	dPOW	20		
LE	1	lNAND	1	i2d	1	dCMP	6	dATAN	20		
GE	1	lNOR	1			dSQRT	20	fCMPeq	3		
LT	1	lEQV	1			dABS	3	fCMPne	3		

Figure A.1 The SVLIW architecture

Number of functional units: 8; 2 each of S unit, M unit, L unit, and D unit.
Number of issue slots: 8
All issue times for all operations are 1
D unit executes all mem ops. Loads of integers, words, bytes, and floats take 5 cycles; doubles 10 cycles. Stores of integers, words, bytes, and floats take 1 cycles; doubles 2 cycles. In addition the D unit can execute integer copies (`i2X`) and the ILOC operations `iADD`, `iNEG`, and `iSUB` with the latencies shown below.
All other units:

S unit						M unit		L unit			
`dMAX`	20	`lSHIFT`	1	`BRgt`	1	`iMUL`	2	`dABS`	2	`fCMPgt`	2
`dTRUNC`	20	`lSL`	1	`JSRl`	1	`iDIV`	4	`iABS`	2	`dCMPeq`	4
`dSQRT`	20	`lSLI`	1	`dJSRl`	1	`iMOD`	4	`fADD`	2	`dCMPne`	4
`dATAN`	20	`lSR`	1	`iJSRl`	1	`fMUL`	4	`dADD`	4	`dCMPle`	4
`dEXP`	20	`lSRI`	1	`fJSRl`	1	`fDIV`	8	`iADD`	1	`dCMPge`	4
`dPOW`	20	`lAND`	1	`JSRr`	1	`dMUL`	8	`lAND`	1	`dCMPlt`	4
`dPOWi`	20	`lOR`	1	`dJSRr`	1	`dDIV`	16	`iCMPeq`	1	`dCMPgt`	4
`dLOG`	20	`lNAND`	1	`iJSRr`	1			`iCMPne`	1	`f2i`	1
`iPOW`	20	`lNOR`	1	`fJSRr`	1			`iCMPle`	1	`f2f`	1
`FRAME`	1	`lEQV`	1	`RTN`	1			`iCMPge`	1	`f2d`	1
`iLDI`	1	`lXOR`	1	`iRTN`	1			`iCMPlt`	1	`d2i`	1
`iSIGN`	1	`lNOT`	1	`dTRN`	1			`iCMPgt`	1	`d2d`	1
`dSIGN`	4	`iMIN`	1	`fRTN`	1			`iCMP`	1	`d2f`	1
`iADD`	1	`dMIN`	4	`fADD`	2			`EQ`	1	`i2i`	1
`iSUB`	1	`iMAX`	1	`fSUB`	2			`NE`	1	`i2f`	1
`iNEG`	1	`HALT`	1	`fNEG`	2			`LE`	1	`i2d`	1
`iADDI`	1	`JMPl`	1	`dADD`	4			`GE`	1	`fNEG`	2
`iSUBI`	1	`JMPr`	1	`dSUB`	4			`LT`	1	`dNEG`	4
`i2i`	1	`JMPT`	1	`dNEG`	4			`GT`	1	`iNEG`	1
`i2f`	1	`BR`	1	`f2i`	1			`fCMP`	2	`lNOT`	1
`i2d`	1	`BReq`	1	`f2f`	1			`dCMP`	4	`lOR`	1
`iSL`	1	`BRne`	1	`f2d`	1			`fCMPeq`	2	`lNAND`	1
`iSLI`	1	`BRle`	1	`d2i`	1			`fCMPne`	2	`lNOR`	1
`iSR`	1	`BRge`	1	`d2f`	1			`fCMPle`	2	`fSUB`	2
`iSRI`	1	`BRlt`	1	`d2d`	1			`fCMPge`	2	`dSUB`	4
`dLDI`	1	`fLDI`	1					`fCMPlt`	2	`iSUB`	1
								`iSUBI`	1	`lXOR`	1

Figure A.2 The C60 architecture model

Number of functional units: 6; SRU, IU1, IU2, FPU, LSU, BPU
Number of issue slots: 2
All issue times for all operations are 1 except as noted in the table. A superscript next to a latency denotes the issue time for that operation.
BPU executes all ILOC control flow operations including subroutine calls and returns. All have 1 cycle latency.
IU2 is the same as IU1 (described below) except it does *not* execute `iDIV`, `iMOD`, or `iMUL`.
LSU executes all memory operations. All loads and stores have latency 2 cycles.
All other units:

IU1				FPU				SRU	
iABS	1	i2f	1	fADD	3	dCMPge	3	lAND	1
iLDI	1	i2d	1	fSUB	3	dCMPlt	3	lOR	1
iSIGN	1	iSL	1	fNEG	3	dCMPgt	3	lNAND	1
iADD	1	iSLI	1	fMUL	3	f2i	3	lNOR	1
iSUB	1	iSR	1	fDIV	17^{17}	f2f	3	lEQV	1
iNEG	1	iSRI	1	dADD	3	f2d	3	lXOR	1
iMUL	3^{3}	lSHIFT	1	dSUB	3	d2i	3	lNOT	1
iDIV	19^{19}	lSL	1	dSIGN	3	d2f	3		
iADDI	1	lSLI	1	dLOG	20	d2d	3		
iSUBI	1	lSR	1	dEXP	20	dLDI	3		
iCMP	1	lSRI	1	dNEG	3	dABS	3		
EQ	1	iMOD	19^{19}	dMUL	4^{2}	dPOWI	20		
NE	1	iMIN	8	dDIV	31^{31}	dTRUNC	3		
LE	1	iMAX	8	fCMP	3	dMAX	20		
GE	1	lAND	1	dCMP	3	dMIN	20		
LT	1	lOR	1	fCMPeq	3	dPOW	20		
GT	1	lNAND	1	fCMPne	3	dATAN	20		
iCMPeq	1	lNOR	1	fCMPle	3	fLDI	3		
iCMPne	1	lEQV	1	fCMPge	3	dSQRT	3		
iCMPle	1	lXOR	1	fCMPlt	3				
iCMPge	1	lNOT	1	fCMPgt	3				
iCMPlt	1	iPOW	5	dCMPeq	3				
iCMPgt	1			dCMPne	3				
i2i	1			dCMPle	3				

Figure A.3 The PPC750 architecture model

Appendix B

Rbf scheduling results

	Dynamic Instructions			Static Instructions		
Benchmark	LS	RBF	%	LS	RBF	%
adpcm	25869274	25853278	0.06	98	98	0
clean	7121253	7120446	0.01	1909	1906	0.16
compress	6451355	6451078	0.00	302	302	0
dfa	1234290	1234285	0.00	344	344	0
dhrystone	6310417	6310416	0.00	108	108	0
fft	19080610	19057570	0.12	360	359	0.28
go	873554813	873205538	0.04	12669	12662	0.06
gzip	3630649	3630635	0.00	2516	2514	0.08
jpeg	59715530	59650104	0.11	3003	3003	0
nsieve	3810982127	3810982122	0.00	49	49	0
shorten	4014951	4014948	0.00	1615	1615	0
water	37219738	37209116	0.03	731	731	0
fmin	4363	4363	0	61	61	0
rkf45	622271	621957	0.05	186	186	0
seval	2508	2506	0.08	80	80	0
solve	2820	2819	0.04	118	117	0.85
svd	12713	12712	0.01	239	239	0
urand	1504	1504	0	23	23	0
zeroin	3710	3710	0	40	40	0
applu	676646005	676524005	0.02	1868	1862	0.32
cg	1115328725	1115327325	0.00	150	150	0
doduc	13506556	13497869	0.06	3764	3759	0.13
fpppp	57176443	57149684	0.05	1551	1548	0.19
matrix300	33652694	33652694	0	77	77	0
mg	56027886	56027062	0.00	373	371	0.54
tomcatv	332355068	332339788	0.00	125	125	0

Table B.1 Rbf vs. ls targeting c60, on basic blocks

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
adpcm	25027266	25027266	0	25027266	0	25027266	0
clean	6484949	6484699	0.00	6484220	0.01	6484220	0.01
compress	5967052	5966583	0.01	5966583	0.01	5966583	0.01
dfa	1165659	1165658	0.00	1165658	0.00	1165658	0.00
dhrystone	6040417	6040417	0	6040417	0	6040417	0
fft	18872854	18889994	-0.08	18861333	0.06	18861333	0.06
go	800500402	800539246	0.00	800077136	0.05	800077136	0.05
gzip	3463878	3462656	0.04	3461930	0.06	3459100	0.14
jpeg	56913698	56854164	0.1	56854144	0.1	56854144	0.1
nsieve	3793661381	3793655488	0.00	3793655488	0.00	3793655488	0.00
shorten	3939604	3939601	0.00	3939601	0.00	3939601	0.00
water	34985459	35546959	-1.59	34960943	0.07	34960943	0.07
fmin	3852	3852	0	3852	0	3852	0
rkf45	578933	578789	0.02	578859	0.01	578789	0.02
seval	2454	2453	0.04	2453	0.04	2453	0.04
solve	2724	2721	0.11	2721	0.11	2721	0.11
svd	12151	12137	0.12	12137	0.12	12137	0.12
urand	1487	1487	0	1487	0	1487	0
zeroin	3388	3388	0	3388	0	3388	0
applu	660381369	660259321	0.02	660259317	0.02	660259317	0.02
cg	1114112782	1114112780	0.00	1114112780	0.00	1114112780	0.00
doduc	12718250	12716901	0.01	12714948	0.03	12714948	0.03
fpppp	49874088	49005677	1.74	48718716	2.32	48718716	2.32
matrix300	33403762	33403762	0	33403762	0	33403762	0
mg	54673420	54667724	0.01	54667724	0.01	54667724	0.01
tomcatv	332344654	332329374	0.00	332329374	0.00	332329374	0.00

Table B.2 Dynamic Instruction Counts RBF vs. LS targeting C60 scheduling over EBB's

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
`adpcm`	100	100	0	99	1.00	101	-0.99
`clean`	1977	1977	0	1955	1.11	1959	0.91
`compress`	309	308	0.32	306	0.97	309	0
`dfa`	356	356	0	352	1.12	353	0.84
`dhrystone`	111	111	0	112	-0.89	112	-0.89
`fft`	374	372	0.53	362	3.21	359	4.01
`go`	13462	13456	0.04	13338	0.92	13344	0.88
`gzip`	2682	2691	-0.33	2666	0.6	2672	0.37
`jpeg`	3059	3059	0	3048	0.36	3052	0.23
`nsieve`	51	50	1.96	49	3.92	49	3.92
`shorten`	1650	1651	-0.05	1635	0.91	1649	0.06
`water`	743	741	0.27	740	0.4	738	0.67
`fmin`	65	65	0	62	4.62	62	4.62
`rkf45`	189	191	-1.05	189	0	190	-0.52
`seval`	83	83	0	80	3.61	80	3.61
`solve`	121	121	0	117	3.31	118	2.48
`svd`	245	244	0.41	236	3.67	238	2.86
`urand`	24	24	0	24	0	24	0
`zeroin`	42	42	0	44	-4.75	44	-4.75
`applu`	1898	1895	0.16	1839	3.11	1846	2.74
`cg`	154	154	0	151	1.95	151	1.95
`doduc`	3992	3992	0	3967	0.63	3969	0.58
`fpppp`	1561	1544	1.09	1542	1.21	1540	1.34
`matrix300`	81	81	0	84	-3.69	84	-3.69
`mg`	380	380	0	365	3.95	368	3.16
`tomcatv`	131	130	0.76	129	1.53	129	1.53

Table B.3 Static Instruction Counts RBF vs. LS targeting C60 scheduling over EBB's

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
adpcm	23717774	23701878	0.07	23701878	0.07	23701878	0.07
clean	6374311	6402029	-0.42	6373635	0.01	6373635	0.01
compress	5969723	5966782	0.05	5966782	0.05	5966782	0.05
dfa	1196470	1196469	0.00	1188818	0.64	1188818	0.64
dhrystone	5920418	5920417	0.00	5920417	0.00	5920417	0.00
fft	18866424	18879719	-0.06	18871544	-0.02	18871544	-0.02
go	794506738	796330825	-0.23	794269084	0.03	794217007	0.04
gzip	3468937	3472317	-0.09	3468928	0.00	3468927	0.00
jpeg	57755148	57747082	0.01	57747077	0.01	57747077	0.01
nsieve	3793661381	3793655488	0.00	3793655488	0.00	3793655488	0.00
shorten	3874954	3874954	0	3874952	0.00	3874952	0.00
water	34400785	35113774	-2.06	34393294	0.02	34393294	0.02
fmin	3604	3604	0	3604	0	3604	0
rkf45	565012	564698	0.06	564698	0.06	564698	0.06
seval	2454	2453	0.04	2453	0.04	2453	0.04
solve	2697	2694	0.11	2695	0.07	2694	0.11
svd	12268	12306	-0.3	12258	0.08	12258	0.08
urand	1494	1494	0	1494	0	1494	0
zeroin	3300	3300	0	3300	0	3300	0
applu	660639674	660631327	0.00	660472973	0.03	660472923	0.03
cg	1114120191	1114120191	0	1114120191	0	1114120191	0
doduc	12143306	12148818	-0.04	12136331	0.06	12136331	0.06
fpppp	49776879	49021681	1.32	49982325	-0.41	48889387	1.78
matrix300	33404562	33404562	0	33404562	0	33404562	0
mg	54410177	54404497	0.01	54404497	0.01	54404497	0.01
tomcatv	306334614	306319334	0.00	306319334	0.00	306319334	0.00

Table B.4 Dynamic Instruction Counts RBF vs. LS targeting C60 scheduling over DP's

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
adpcm	99	99	0	98	1.01	98	1.01
clean	1966	1962	0.2	1945	1.07	1948	0.92
compress	307	309	-0.64	306	0.33	306	0.33
dfa	352	351	0.28	346	1.7	346	1.7
dhrystone	113	112	0.88	113	0	112	0.88
fft	368	367	0.27	361	1.9	358	2.72
go	13440	13427	0.10	13333	0.80	13321	0.89
gzip	2644	2644	0	2638	0.23	2643	0.04
jpeg	3059	3057	0.07	3057	0.07	3056	0.1
nsieve	52	51	1.92	51	1.92	51	1.92
shorten	1649	1650	-0.05	1634	0.91	1633	0.97
water	741	736	0.67	733	1.08	733	1.08
fmin	64	64	0	63	1.56	62	3.13
rkf45	187	187	0	184	1.6	187	0
seval	83	83	0	81	2.41	81	2.41
solve	122	119	2.46	119	2.46	116	4.92
svd	243	242	0.41	239	1.65	238	2.06
urand	23	23	0	23	0	23	0
zeroin	43	43	0	43	0	43	0
applu	1889	1870	1.01	1856	1.75	1849	2.12
cg	152	152	0	150	1.32	150	1.32
doduc	3894	3986	-2.35	3980	-2.20	3972	-1.99
fpppp	1564	1545	1.21	1560	0.26	1547	1.09
matrix300	81	81	0	84	-3.69	85	-4.93
mg	380	379	0.26	372	2.11	372	2.11
tomcatv	132	133	-0.75	133	-0.75	134	-1.51

Table B.5 Static Instruction Counts RBF vs. LS targeting C60 scheduling over DP's

Benchmark	LS	RBF	%
adpcm	10099568	10099568	0
clean	3135572	3123260	0.39
compress	6330836	6327798	0.05
dfa	456358	456358	0
dhrystone	2730101	2730101	0
fft	10061265	9803711	2.56
go	363430164	362804782	0.17
gzip	1504428	1504367	0.00
jpeg	32992742	32991687	0.00
nsieve	1510981283	1510981283	0
shorten	1533947	1533880	0.00
water	16911814	16893458	0.11
fmin	3839	3832	0.18
rkf45	438503	436366	0.49
seval	2655	2633	0.83
solve	1570	1568	0.13
svd	7998	7997	0.01
urand	1031	1001	2.91
zeroin	3772	3758	0.37
applu	455904964	455390455	0.11
cg	522124378	522093578	0.01
doduc	15254873	15088446	1.09
fpppp	94083373	93599473	0.51
matrix300	14675862	14675862	0
mg	47926868	46914392	2.11
tomcatv	266181287	258393585	2.93

Table B.6 Dynamic Instruction Counts RBF vs. LS targeting PPC750, on basic blocks

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
adpcm	9458460	9458560	0.00	9458460	0	9458460	0
clean	2704522	2759467	-2.02	2681612	0.85	2681143	0.86
compress	5387476	5383897	0.07	5381350	0.11	5381251	0.12
dfa	395923	395923	0	395923	0	395923	0
dhrystone	2560101	2560101	0	2560101	0	2550101	0.39
fft	9818240	9591672	2.31	9572470	2.50	9564784	2.58
go	320719566	322511748	-0.55	317890392	0.88	317717950	0.94
gzip	1400111	1411322	-0.79	1373679	1.89	1373668	1.89
jpeg	31223308	31284005	-0.18	31218301	0.02	31218345	0.02
nsieve	1493660543	1493660543	0	1493660543	0	1493660543	0
shorten	1491475	1532282	-2.73	1475506	1.07	1475505	1.07
water	16468732	17924506	-8.83	16376620	0.56	16366220	0.62
fmin	3684	3677	0.19	3677	0.19	3677	0.19
rkf45	408322	405167	0.77	405156	0.78	405153	0.78
seval	2621	2598	0.88	2598	0.88	2598	0.88
solve	1477	1476	0.07	1474	0.20	1474	0.2
svd	7590	7592	-0.02	7583	0.09	7581	0.12
urand	1024	1004	1.95	974	4.88	974	4.88
zeroin	3516	3516	0	3511	0.14	3511	0.14
applu	447032669	446768173	0.06	446433959	0.13	446418159	0.14
cg	520954400	520931526	0.00	520931526	0.00	520931526	0.00
doduc	14832434	14701091	0.89	14670341	1.09	14653506	1.21
fpppp	102496332	98738452	3.67	98389787	4.01	98377039	4.02
matrix300	14550178	14550178	0	14550178	0	14550178	0
mg	44332727	44097981	0.53	44075349	0.58	44071269	0.59
tomcatv	266170802	259683640	2.44	259683600	2.44	259683640	2.44

Table B.7 Dynamic Instruction Counts RBF vs. LS targeting PPC750 scheduling over EBB's

		No weighting		.75 weighting		Profile weighting	
Benchmark	LS	RBF	%	RBF	%	RBF	%
adpcm	8498968	8499068	0.00	8498968	0	8498968	0
clean	2579861	2589583	-0.37	2566738	0.51	2566751	0.51
compress	5379336	5385276	-0.1	5376124	0.06	5373436	0.11
dfa	412917	415738	-0.67	412880	0.01	412884	0.01
dhrystone	2510101	2510101	0	2510101	0	2510101	0
fft	9810010	9697640	1.15	9683286	1.29	9560406	2.54
go	313583874	318728709	-1.64	312013457	0.50	311721557	0.59
gzip	1386398	1377330	0.65	1370884	1.12	1370769	1.13
jpeg	31392039	31620688	-0.72	31388960	0.00	31388957	0.00
nsieve	1493660551	1493660551	0	1493660551	0	1493660551	0
shorten	1426897	1435651	-0.6	1426894	0.00	1426765	0.01
water	15582108	16100662	-3.32	15538949	0.28	15538459	0.28
fmin	3519	3513	0.17	3518	0.03	3512	0.2
rkf45	406624	404515	0.52	404511	0.52	404219	0.59
seval	2607	2589	0.69	2589	0.69	2589	0.69
solve	1461	1460	0.07	1461	0	1459	0.14
svd	7648	7652	-0.04	7645	0.04	7641	0.09
urand	986	986	0	986	0	986	0
zeroin	3457	3457	0	3457	0	3457	0
applu	447464221	447076759	0.09	447033757	0.1	446905157	0.12
cg	520959810	520936954	0.00	520936954	0.00	520936954	0.00
doduc	14382112	14281480	0.7	14286027	0.67	14265035	0.81
fpppp	99003037	100528449	-1.54	98844851	0.16	98636572	0.37
matrix300	14549778	14549778	0	14549778	0	14549778	0
mg	44070213	43835632	0.53	43858056	0.48	43835256	0.53
tomcatv	250564822	247979120	1.03	247979120	1.03	247979120	1.03

Table B.8 Dynamic Instruction Counts RBF vs. LS targeting PPC750 scheduling over DP's

Appendix C

IR scheduling results

Benchmark	LS	IR-rdf		IR-rdf-bias		IR-length-bias	
		10	100	10	100	10	100
adpcm	98	98	98	98	98	98	98
clean	1909	1907.0	1907.6	1907.4	1907.2	1907.8	1908.4
compress	302	301.0	299.6	302.0	301.2	301.0	301.0
dfa	344	344	344	344	344	344	344
dhrystone	108	108.6	108.2	108.4	108.2	108.2	108.6
fft	360	361.0	359.0	358.4	359.4	359.0	358.2
go	12669	12684.8	12684.0	12676.4	12676.8	12678.0	12676.8
gzip	2516	2514.4	2514.2	2513.0	2516.4	2513.4	2514.8
jpeg	3003	3005.4	3006.4	3002.8	3005.8	3004.2	3006.2
nsieve	49	49.0	49.2	49.4	49.2	49.4	49.4
shorten	1615	1614.4	1614.4	1614.6	1614.4	1613.2	1614.4
water	731	729.4	731.4	732.8	731.8	732.4	731.2
fmin	61	61	61	61	61	61	61
rkf45	186	185.4	185.8	185.4	185.2	185.4	185.2
seval	80	80	80	80	80	80	80
solve	118	117.4	117.4	117.6	117.4	117.6	117.4
svd	239	240.0	240.6	238.2	238.0	238.4	238.2
urand	23	23.6	23.2	23.2	23.4	23.2	23.2
zeroin	40	39.8	39.6	39.8	39.8	39.8	39.8
applu	1868	1877.6	1881.6	1869.8	1876.4	1871.6	1867.0
cg	150	148.4	148	148.2	148.6	148.4	148.4
doduc	3764	3762.0	3765.0	3759.6	3765.2	3763.4	3766.4
fpppp	1551	1481.4	1541.6	1457.2	1533.8	1516.4	1542.4
matrix300	77	77	77	77	77	77	77
mg	373	380.6	381.6	383.0	382.6	382.0	378.8
tomcatv	125	127.2	126.0	126.8	127.0	127.0	127.6

Table C.1 Static Instruction Counts IR vs. LS targeting C60, on basic blocks

Table C.2 Dynamic Instruction Counts IR vs. LS targeting C60, on basic blocks

Benchmark	LS	IR-rdf 10	IR-rdf 100	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	25869274	25853278	25853278	25853278	25853278	25853278	25853278
clean	7121253	7120513.6	7120446	7120446	7120446	7120463.0	7120446
compress	6451355	6452208.2	6451078	6451078	6451078	6451640.6	6451078
dfa	1234290	1234284.4	1234284	1234285.2	1234284	1234285.2	1234284
dhrystone	6310417	6310417	6310416.6	6310416.8	6310416.2	6310416.4	6310416
fft	19080610	19057570	19057570	19057570	19057570	19057570	19057570
go	873554813	873152676.6	873098935.2	873110332.0	873099024.6	873106237.0	873099430.4
gzip	3630649	3630635.8	3630635.4	3630636.0	3630635	3630635.6	3630635
jpeg	59715530	59686550.8	59600377	59634906.4	59615065.0	59654387.8	59610169.0
nsieve	3810982127	3810982122.2	3810982122	3810982122	3810982122	3810982122	3810982122
shorten	4014951	4014948.4	4014948	4014948.2	4014948	4014948.2	4014948
water	37219738	37228339.0	37223140.0	37212887.0	37209613.2	37210930.4	37209616.0
fmin	4363	4363	4363	4363	4363	4363	4363
rkf45	622271	622394.4	621957	621987.0	621962.0	621987.0	621962.0
seval	2508	2507.8	2506	2506	2506	2506	2506
solve	2820	2819	2819	2819	2819	2819	2819
svd	12713	12712	12712	12712.2	12712	12712.2	12712
urand	1504	1504	1504	1504	1504	1504	1504
zeroin	3710	3710	3710	3710	3710	3710	3710
applu	676646005	676962387.4	676445783.4	676474790.6	676234285.0	676312769.0	676142417.8
cg	1115328725	1115327325	1115327325	1115327325	1115327325	1115327325	1115327325
doduc	13506556	13527984.8	13515558.4	13509900.2	13506960.8	13505091.6	13503486.2
fpppp	57176443	59283643.2	58271909.4	57622640.0	57460542.2	57182372.2	57118625
matrix300	33652694	33652694	33652694	33652694	33652694	33652694	33652694
mg	56027886	56097270.0	56027062	56027062	56027062	56027226.8	56027062
tomcatv	332355068	333121104.0	331555408.0	331559488.0	331038268.0	331039288	331036228.0

Benchmark	LS	IR-rdf-bias		IR-length-bias	
		10	100	10	100
adpcm	25027266	25027406.0	25027346.0	25027406.0	25027346.0
clean	6484949	6506509.6	6519702.8	6506503.4	6515751.8
compress	5967052	5977480.6	5972375.6	5975701.2	5975206.8
dfa	1165659	1165657	1165657	1165657	1165657
dhrystone	6040417	6040417	6040417	6040417	6040417
fft	18872854	18836762.0	18865430.8	18869531.6	18861341.2
go	800500402	799961122.6	800015951.0	800089942.2	800039884.8
gzip	3463878	3460531.6	3457140.2	3456893.4	3456931.4
jpeg	56913698	56936717.0	56916880.4	56956326.0	56887792.2
nsieve	3793661381	3793655488	3793655488	3793655488	3793655488
shorten	3939604	3939595.4	3939594.6	3939595.8	3939594.0
water	34985459	35367287.0	35361913.2	35119621.8	35069228.0
fmin	3852	3852	3852	3852	3852
rkf45	578933	578850.0	578819.4	578849.8	578819.2
seval	2454	2453	2452.8	2453	2452.8
solve	2724	2721.6	2721.6	2721.6	2721.6
svd	12151	12139.0	12138.4	12139.0	12138.4
urand	1487	1487	1487	1487	1487
zeroin	3388	3388	3388	3388	3388
applu	660381369	660287168.0	660040862.8	660104587.0	659984755.4
cg	1114112782	1114112780	1114112780	1114112780	1114112780
doduc	12718250	12734430.2	12732585.4	12729798.8	12725487.2
fpppp	49874088	49462646.4	49131492.8	50038297.8	49859700.2
matrix300	33403762	33403762	33403762	33403762	33403762
mg	54673420	54667724.2	54667724	54667724	54667724
tomcatv	332344654	331028874	331026834.0	331809174.0	331026834.0

Table C.3 Dynamic Instruction Counts various IR techniques vs. LS targeting C60 scheduling over EBB's (1 of 2)

Benchmark	LS	IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100
`adpcm`	25027266	25027266	25027266	25027266	25027266
`clean`	6484949	6484935.8	6479831.2	6478173.4	6474841
`compress`	5967052	5966576.2	5966508	5966518.0	5966508
`dfa`	1165659	1165657	1165657	1165657	1165657
`dhrystone`	6040417	6040417	6040417	6040417	6040417
`fft`	18872854	18820365.2	18820365	18820365	18820365
`go`	800500402	797535296.6	797416611.8	797391333.0	797321989.0
`gzip`	3463878	3457614.8	3456624	3457861.6	3456624
`jpeg`	56913698	56864880.8	56819102.0	56838909.6	56819051.2
`nsieve`	3793661381	3793655488	3793655488	3793655488	3793655488
`shorten`	3939604	3939591.4	3939591	3939593.0	3939591
`water`	34985459	34935783.0	34932458.4	34911484.8	34873573.0
`fmin`	3852	3852	3852	3852	3852
`rkf45`	578933	578862.2	578812.8	578848.2	578812.8
`seval`	2454	2450	2450	2450	2450
`solve`	2724	2720.4	2720	2720.8	2720
`svd`	12151	12137.6	12137	12137.6	12137
`urand`	1487	1487	1487	1487	1487
`zeroin`	3388	3388	3388	3388	3388
`applu`	660381369	660168983.2	660013037.0	660177142.6	659980737.0
`cg`	1114112782	1114112780	1114112780	1114112780	1114112780
`doduc`	12718250	12724255.4	12720551.2	12720279.8	12717535.0
`fpppp`	49874088	49172909.6	48865521.4	49056489.8	48860003.4
`matrix300`	33403762	33403762	33403762	33403762	33403762
`mg`	54673420	54667724	54667724	54667724	54667724
`tomcatv`	332344654	331028874	331026834.0	331288974.0	331026834.0

Table C.4 Dynamic Instruction Counts various IR techniques vs. LS targeting C60 scheduling over EBB's (2 of 2)

Benchmark	LS	IR-rdf-bias		IR-length-bias		IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100	10	100	10	100
adpcm	100	100	99.8	100	100	99.2	99.4	100	100
clean	1977	1973.4	1973.2	1975.0	1975.0	1974.0	1974.0	1972.2	1972.6
compress	309	309.8	309.2	309	309	310.4	310.2	309.6	311.0
dfa	356	356	356	356	356	356	356	356	356
dhrystone	111	109.6	109.4	110.4	110.4	108.8	108.6	110.2	110.0
fft	374	368.6	369.0	369.2	369.2	368.2	367.8	369.0	368.8
go	13462	13515.8	13516.8	13506.0	13506.0	13471.2	13468.6	13470.4	13477.6
gzip	2682	2689.0	2688.6	2687.0	2687.0	2678.2	2678.6	2685.6	2684.0
jpeg	3059	3062.2	3063.8	3060.0	3060.0	3063.8	3066.6	3061.2	3062.4
nsieve	51	51.4	51	51	51	51.4	51	51.4	51
shorten	1650	1654.0	1653.4	1653.0	1653.0	1654.4	1653.2	1651.4	1653.4
water	743	742.6	743.4	741.2	741.2	742.0	739.6	743.0	744.0
fmin	65	65	64.6	64.8	64.8	65	64.6	65	64.6
rkf45	189	190.8	191.0	190.4	190.4	190.0	190	191.0	190.8
seval	83	83.8	83.2	83.6	83.6	83.2	83.0	83.2	83.4
solve	121	121.2	121.4	121.2	121.2	121.6	121.8	122	122
svd	245	245.4	246.0	245.2	245.2	244.8	245.0	245.0	245.2
urand	24	24	24	24.6	24.6	24	24	24	24
zeroin	42	42.2	42	42.2	42.2	42	42	42	42
applu	1898	1889.8	1894.6	1888.8	1888.8	1894.0	1891.6	1889.0	1899.2
cg	154	153.2	153.2	153	153	153.4	153.4	153.2	153.8
doduc	3992	3992.4	3987.8	3990.4	3989.0	3990.2	3989.2	3989.8	3985.0
fpppp	1561	1470.4	1471.4	1480.4	1480.4	1471.0	1470.6	1473.6	1472.6
matrix300	81	80.6	80.6	80.6	80.6	80.8	80.8	80.8	80.8
mg	380	384.8	386.0	385.4	385.4	383.4	385.6	384.4	385.6
tomcatv	131	133.2	133.2	132.8	132.8	133.8	133.4	133.6	133.4

Table C.5 Static Instruction Counts various IR techniques vs. LS targeting C60 scheduling over EBB's

Benchmark	LS	IR-rdf-bias		IR-length-bias	
		10	100	10	100
`adpcm`	23717774	23701878	23701918.0	23701878	23701918.0
`clean`	6374311	6388610.4	6393053.4	6389835.0	6391540.8
`compress`	5969723	5966737.0	5966782.0	5966752.0	5966757.0
`dfa`	1196470	1196463.8	1196462.6	1196464.2	1196461.4
`dhrystone`	5920418	5920417.4	5920417	5920417	5920417
`fft`	18866424	18887929.6	18846968.4	18865658.0	18887928.6
`go`	794506738	794192927.6	794287624.4	794342890.8	794387402.0
`gzip`	3468937	3468944.2	3468946.2	3468948.4	3468948.0
`jpeg`	57755148	57733671.6	57674655.0	57723880.6	57694289.6
`nsieve`	3793661381	3793655489.0	3793655488.4	3793655488.8	3793655488.2
`shorten`	3874954	3874999.2	3875009.2	3874999.0	3875010.0
`water`	34400785	34615488.4	34556669.6	34458402.2	34425553.6
`fmin`	3604	3604	3604	3604	3604
`rkf45`	565012	564729.0	564704.4	564729.8	564704.4
`seval`	2454	2453	2453	2453	2453
`solve`	2697	2694.4	2694	2694.4	2694
`svd`	12268	12260.2	12260.0	12260.2	12259.4
`urand`	1494	1512.0	1506.6	1513.8	1504.8
`zeroin`	3300	3300	3300	3300	3300
`applu`	660639674	660665162.0	660360006.4	661103430.6	660373139.0
`cg`	1114120191	1114120191	1114120191	1114120191	1114120191
`doduc`	12143306	12154094.8	12150589.4	12149900.0	12146720.8
`fpppp`	49776879	50237029.2	49844636.0	50430771.4	50181114.6
`matrix300`	33404562	33404562	33404562	33404562	33404562
`mg`	54410177	54441943.4	54404497	54404497	54404497
`tomcatv`	306334614	307363818.0	306319342.0	306582494.0	307100650.0

Table C.6 Dynamic Instruction Counts various IR techniques vs. LS targeting C60 scheduling over DP's (1 of 2)

Benchmark	LS	IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100
`adpcm`	23717774	23701878	23701878	23701878	23701878
`clean`	6374311	6374164.4	6373188	6362459.6	6362458.8
`compress`	5969723	5967294.4	5966707	5966751.8	5966712.0
`dfa`	1196470	1196457	1196457	1196457	1196457.6
`dhrystone`	5920418	5920417.4	5920417	5920417.4	5920417
`fft`	18866424	18810104	18810104	18810104	18810104
`go`	794506738	792650732.6	792497869.2	792372290.6	792321309.4
`gzip`	3468937	3468921.2	3468920.2	3468936.6	3468925.6
`jpeg`	57755148	46180944.2	57674653.4	57729379.0	57674653.6
`nsieve`	3793661381	3793655489.0	3793655488.4	3793655502.2	3793655501.6
`shorten`	3874954	3874944.6	3874944	3874945.6	3874944
`water`	34400785	34410619.0	34419237.0	34441490.4	34423131.8
`fmin`	3604	3604	3604	3604	3604
`rkf45`	565012	565577.6	564914.2	564728.0	564703.0
`seval`	2454	2454	2454	2453	2453
`solve`	2697	2694.8	2694	2694.4	2694
`svd`	12268	12258	12258	12258	12258
`urand`	1494	1485	1485	1485	1485
`zeroin`	3300	3300	3300	3300	3300
`applu`	660639674	660742529.2	660359263.0	660605014.0	660333103.0
`cg`	1114120191	1114120191	1114120191	1114120191	1114120191
`fpppp`	49776879	50958600.2	50773549.8	49913434.4	49759578.2
`doduc`	12143306	12147850.0	12141837.8	12143287.8	12141359
`matrix300`	33404562	33404562	33404562	33404562	33404562
`mg`	54410177	54434455.4	54404497	54404497	54404497
`tomcatv`	306334614	307365834.0	306840546.0	307623914.0	306321378.0

Table C.7 Dynamic Instruction Counts various IR techniques vs. LS targeting C60 scheduling over DP's (2 of 2)

Table C.8 Static Instruction Counts various IR techniques vs. LS targeting C60 scheduling over DP's

Benchmark	LS	IR-rdf-bias		IR-length-bias		IR-rdf-wbias		IR-rdf-profbias	
		10	100	10	100	10	100	10	100
`adpcm`	99	99	99.0	98.8	98.8	97.8	97.4	98.6	99.0
`clean`	1966	1966.6	1966.0	1966.8	1965.8	1968.6	1968.4	1967.4	1965.0
`compress`	307	308.0	308.4	307.6	307.4	307.2	307.6	307.8	307.2
`dfa`	352	350.6	350.4	350.8	350.4	350.8	351.6	350.6	351.0
`dhrystone`	113	112.4	111.0	111.6	112.0	112.2	112.0	111.8	112.2
`fft`	368	362.8	363.4	363.0	364.6	365.2	363.2	364.0	365.0
`go`	13440	13509.4	13511.6	13497.6	13488.2	13472.2	13468.0	13483.4	13475.6
`gzip`	2644	2651.2	2653.8	2652.2	2650.0	2646.2	2646.4	2650.2	2653.4
`jpeg`	3059	3060.6	3061.8	3064.6	3065.8	3061.8	3061.2	3061.6	3061.4
`nsieve`	52	50.8	50.6	50.6	50.6	50.6	50.2	50.0	50.2
`shorten`	1649	1652.4	1651.4	1651.8	1651.2	1657.0	1655.4	1651.6	1654.4
`water`	741	743.2	745.8	742.8	741.8	740.4	739.8	741.8	742.4
`fmin`	64	63.6	63.8	63.8	63.6	64	64	64	64
`rkf45`	187	187.6	187.6	187.2	186.6	187.4	187.4	188.0	188.2
`seval`	83	83.0	82.4	83.2	82.2	83.4	83.4	82.6	82.6
`solve`	122	121.8	121.2	121.8	121.0	121.6	121.8	121.2	121.2
`svd`	243	243.0	243.0	243.0	243.8	242.2	241.8	243.0	242.6
`urand`	23	23	23	23	23	23	23	23	23
`zeroin`	43	43	43	43	43	43	43	43	43
`applu`	1889	1902.0	1896.2	1897.0	1892.2	1892.4	1892.6	1896.6	1897.6
`cg`	152	151.2	151	151.2	151.6	151	151.4	151	151.8
`fpppp`	1564	1468.4	1465.6	1471.8	1473.8	1468.0	1466.2	1473.2	1472.6
`doduc`	3894	3879.8	3877.6	3884.8	3884.0	3872.4	3868.0	3869.4	3856
`matrix300`	81	81.4	81.4	81.4	81.4	81.2	81.0	80.8	80.8
`mg`	380	386.2	386.4	385.8	386.8	384.6	384.2	385.6	381.2
`tomcatv`	132	134.4	134.6	134.2	136.0	133.6	134.6	134.6	132.6

Table C.9 Dynamic Instruction Counts IR vs. LS targeting PPC750, on basic blocks

Benchmark	LS	IR-rdf		IR-rdf-bias		IR-length-bias	
		10	100	10	100	10	100
`adpcm`	10099568	10119568.0	10099568	10139588.0	10099568	10119568.0	10099568
`clean`	3135572	3137406.6	3125726.0	3136874.4	3123978.2	3131590.8	3124241.0
`compres`	6330836	6465012.2	6327818.0	6374761.4	6327808.0	6417833.2	6327848.0
`dfa`	456358	456358	456358	456386.2	456358	456386.2	456358
`dhrystone`	2730101	2742104.4	2730103	2740104.0	2730101.4	2730102.8	2730101.2
`fft`	10061265	10459588.8	10181363.8	10265950.8	10050088.0	10435934.6	10126785.0
`go`	363430164	364057227.4	363373934.8	363777707.0	363259891.0	364080668.4	363385970.2
`gzip`	1504428	1505496.2	1504384.8	1506776.8	1504385.2	1505596.4	1504374.8
`jpeg`	32992742	33795185.2	33390328.8	33478853.0	33207087.8	33612362.4	33356316.4
`nsieve`	1510981283	1510981291.6	1510981287.4	1527043706.2	1510981284.8	1510981287.8	1510981285.0
`shorten`	1533947	1534597.8	1533951.6	1537161.6	1533913.6	1533966.4	1533913.6
`water`	16911814	17327570.8	17222886.4	17227374.0	17125247.2	17159259.2	17059990.0
`fmin`	3839	3832	3832	3832	3832	3832	3832
`rkf45`	438503	440484.4	437210.8	440379.0	437422	441651.0	438055.6
`seval`	2655	2648.4	2614.8	2634.0	2620.8	2665.6	2633.2
`solve`	1570	1568	1568	1568	1568	1568	1568
`svd`	7998	8011.8	8002.6	8005.2	7996.6	8013.0	7999.2
`urand`	1031	1008.6	1001.2	1010.4	1001.0	1016.4	1011.2
`zeroin`	3772	3758.6	3755.2	3762.6	3758	3772.6	3756.4
`applu`	455904964	467595769.4	463271861.8	464461924.8	460562583.6	466201907.6	461380417.4
`cg`	522124378	522011354.0	522003234.0	522013314.0	522001274	522056118.0	522005194.0
`doduc`	15254873	15282099.0	15193744.0	15261609.4	15170739.2	15310112.6	15219840.4
`fpppp`	94083373	96433703.0	96086223.2	96084424.6	95843477.8	96591728.4	96074296.6
`matrix300`	14675862	14675942.0	14675862	14675882.0	14675862	14675862	14675862
`mg`	47926868	49910323.6	49011680.4	48528068.0	47915326.0	51275739.2	50344409.6
`tomcatv`	266181287	272686341.6	265412463.2	269312027.4	265934447.0	273201999.2	269047487.0

Benchmark	LS	IR-rdf-bias 10	IR-rdf-bias 100	IR-length-bias 10	IR-length-bias 100
adpcm	9458460	9488520.0	9488480.0	9478560.0	9478520.0
clean	2704522	2810226.6	2764541.4	2787038.8	2760302.0
compres	5387476	5403275.2	5389289.2	5421571.4	5384841.8
dfa	395923	396082.4	395923	396009.4	395923
dhrystone	2560101	2586103.0	2560101.6	2566102.2	2558101.0
fft	9818240	10160114.8	9871638.2	10235795.4	9925248.8
go	320719566	325497309.4	323388012.4	324736366.6	322338887.0
gzip	1400111	1412471.6	1408677.0	1403146.2	1403538.0
jpeg	31223308	32276764.2	31821867.6	32198488.6	31969834.6
nsieve	1493660543	1493660559.2	1493660553.4	1509722964.8	1493660544.8
shorten	1491475	1535256.8	1523519.0	1532021.4	1531297.8
water	16468732	17291778.0	16883903.8	17016836.8	16848474.4
fmin	3684	3691.0	3677.6	3678.0	3677.6
rkf45	408322	408565.6	406746.0	411460.8	407689.8
seval	2621	2626.0	2587.6	2638.2	2611.2
solve	1477	1477.4	1475.4	1477.6	1476.2
svd	7590	7607.6	7601.6	7618.6	7607.6
urand	1024	1068.2	1055.0	1086.4	1068.8
zeroin	3516	3523.6	3518.8	3516.4	3515.2
applu	447032669	457775944.8	450788864.4	459697798.8	453759616.6
cg	520954400	520844645.4	520836984.8	520883715.8	520832215.4
doduc	14832434	14943592.0	14846723.0	14997876.8	14882381.6
fpppp	102496332	102004976.4	101885486.0	105526404.8	105350516.0
matrix300	14550178	14558581.2	14550338.0	14562578.0	14550178
mg	44332727	45836834.4	44679982.2	49130754.6	48067260.8
tomcatv	266170802	273527578.6	268523337.2	279170975.8	272679477.4

Table C.10 Dynamic Instruction Counts various IR techniques vs. LS targeting PPC750 scheduling over EBB's (1 of 2)

Benchmark	LS	10	100	10	100
		IR-rdf-wbias		IR-rdf-profbias	
`adpcm`	9458460	9458380.0	9458360	9458440.0	9458360
`clean`	2704522	2807274.0	2768313.0	2771371.8	2709523.2
`compres`	5387476	5383513.2	5378977.4	5385471.2	5373368.0
`dfa`	395923	395966.6	395923	395966.6	395923
`dhrystone`	2560101	2572103.4	2558102.0	2564102.8	2540101.2
`fft`	9818240	10030728.6	9785461.6	10035378.8	9766104.0
`go`	320719566	320062520.6	316463839.6	317810062.2	313956812.8
`gzip`	1400111	1391748.0	1384407.4	1385079.4	1375645.8
`jpeg`	31223308	32099692.6	31606267.8	31744374.2	31351804.8
`nsieve`	1493660543	1493660568.4	1493660554.8	1493660574.4	1493660562.6
`shorten`	1491475	1505450.8	1491553.8	1509868.8	1483073.2
`water`	16468732	16722847.2	16512084.4	16570960.4	16358794.2
`fmin`	3684	3657.6	3669.6	3575.2	3567.6
`rkf45`	408322	409697.0	406656.4	410026.4	406409.6
`seval`	2621	2616.2	2587.8	2616.0	2589.0
`solve`	1477	1477.0	1474.4	1476.2	1473.8
`svd`	7590	7612.2	7593.6	7586.6	7578.2
`urand`	1024	1039.8	1034.8	1022.6	985.0
`zeroin`	3516	3520.4	3515.0	3520.6	3517.0
`applu`	447032669	456591090.4	451403999.6	457395184.6	451281762.0
`cg`	520954400	520839311.6	520831278	520841364.2	520832211.8
`doduc`	14832434	14884622.2	14787232.6	14827347.0	14718132.6
`fpppp`	102496332	98465253.0	98462501.8	98632559.8	97983385.8
`matrix300`	14550178	14558581.2	14550338.0	14566418.0	14550178
`mg`	44332727	44776902.6	44224763.0	44847700.8	44176377.0
`tomcatv`	266170802	273721403.6	269563523.8	272419903.6	268784263.8

Table C.11 Dynamic Instruction Counts various IR techniques vs. LS targeting PPC750 scheduling over EBB's (2 of 2)

Appendix D

Reduced Code Space using Genetic Algorithms

(This work originally published By Cooper, Schielke, and Subramanian [22].)

For years the compiler community has been developing "optimizations", or transformations, whose ultimate goal is to improve the quality of the code generated by the compiler. For example, constant propagation attempts to find all registers whose values at run time are provably constant. The computations of these values can be replaced by a less costly load immediate operation. Later, an optimization to eliminate useless code can remove all the load immediate operations whose result is never used.

With so many optimizations available to the compiler, it is virtually impossible to select the best set of optimizations to run on a particular piece of code. Historically, compiler writers have made one of two assumptions. Either a fixed optimization order is "good enough" for all programs, or giving the user a large set of flags that control optimization is sufficient, because it shifts the burden onto the user.

Interplay between optimizations occurs frequently. A transformation can create opportunities for other transformations. Similarly, a transformation can eliminate opportunities for other transformations. These interactions also depend on the program being compiled, and they are often difficult to predict. Multiple applications of the same transformation at different points in the optimization sequence may produce further improvements. With all these possibilities, it is unlikely that a single fixed sequence of optimizations will be able to produce optimal results for all programs.

The focus of this work is discovering optimization sequences that result in small code sizes. Small code sizes are increasingly important as the number of embedded applications increases. Since code for these systems is typically burned into a ROM, the embedded systems software designer may be willing to tolerate substantially longer compile times (on the "final" compile) in the hopes of reducing the size of the object code.

A genetic algorithm (or GA) is essentially a biased sampling search technique. Instead of merely choosing solutions at random, the GA evolves solutions by merging parts of different solutions and making small mutational changes to solutions, to produce new solutions. Genetic algorithms have a rich history in the literature on artificial intelligence. The idea is generally attributed to Holland [46]. Other excellent sources of information on GA's are books by Goldberg [40], Davis [29], and Mitchell [60].

Genetic algorithms have been applied to other compiler problems with some success. In his thesis, Beaty tried genetic algorithms for instruction scheduling [4]. Nisbet is currently investigating genetic algorithms that discover transformation sequences for parallelizing FORTRAN loops [66].

GA's appear to be a good fit to the problem of finding compiler optimization sequences to minimize code size. First, the space of possible sequences is quite large when several optimizations are at the compiler's disposal. This space is too large to search completely. Second, we have a very good evaluation function to assess the quality of a solution. We simply perform the optimizations and compute the size of the compiled code. Our objective function is discrete and nonlinear, making the problem difficult to address with more classical combinatorial optimization techniques. Finally, the amount of time spent by the GA is flexible. More computation time may result in better solutions, but the algorithm can be interrupted at any time to return the best solution found.

Our experiments show that GA's are well suited to the problem of finding good optimization sequences. For each program in our set of benchmarks, the GA discovered a sequence that produces smaller codes than the hand-designed sequence used in our compiler for many years. In many cases, the resulting code is also faster. These results showed us how to design a different fixed sequence that produces smaller code than our default sequence. Finally, we show that additional savings in space of up to five percent can be obtained by running the GA on a program-by-program basis versus a single fixed sequence for all programs. This suggests the potential of designing optimization sequences that are custom-fitted for a program, rather than constructing a universal, one-size-fits-all sequence.

In Section D.1, we give the necessary details of our research compiler and describe the optimizations used in our experiments. Section D.2 gives an overview of genetic algorithms and provides the details of the GA we used in our experiments. In Section D.3, we present some experimental results for several benchmark programs. We compare code sizes for compiling with no optimization, fixed optimization sequences, and optimizing using sequences found by the genetic algorithm. Finally, in Section D.4, we present some conclusions, observations, and directions for future work.

D.1 Our compiler framework

Our research compiler includes front ends for C and FORTRAN. The front ends translate the source code into our low-level intermediate language, called ILOC. ILOC is an assembly-level language and is RISC-like; operands for all computational operations must be in registers. Several flavors of load and store operations transfer values between registers and memory.

Once the code has been transformed into ILOC, we have various optimization passes that can be applied. Each optimization is an ILOC-to-ILOC transformation designed to improve the code in some way. In the experiments for this work, the following 10 program transformations were used:

1. Constant propagation (`cprop`): This pass is an implementation of Wegman and Zadeck's sparse conditional constant propagation technique [86]. We run cprop with an option (`-m`) that disables conversion of multiplies into shifts and adds. That conversion forestalls some subsequent optimizations and increases code size. We also enable the option (`-f`) that performs propagation on symbolic constants.

2. Dead Code Elimination (`dead`): Our dead code eliminator is based on Kennedy's work [50] and uses SSA form [27].

3. Empty Basic Block Removal (`clean`): This pass removes basic blocks that contain only a single control-flow operation.

4. Global Value Numbering (`valnum`): Value numbering is done using the SCC-based value numbering described by Cooper, Briggs, and Simpson [23, 13]. The `valnum` implementation has many options. The work in this chapter used the following settings: perform constant folding (`-f`), trace values through memory (`-m`), remove some redundant computations (`-a`), perform algebraic simplification (`-s`), and perform value driven code motion (`-c`).

5. Lazy Code Motion (`lazy`): This pass performs code motion using techniques described by Drechsler and Stadel [31], and Knoop, *et al.* [52].

6. Partial Redundancy Elimination (`partial`): This pass implements Morel and Renvoise's technique for partial redundancy elimination [61], following Drechsler and Stadel [30].

7. Peephole Optimization (`combine`): This pass is an SSA-based peephole optimizer that combines ILOC operations according to a Power-PC 601 architectural model.

8. Reassociation (`shape`): This pass reorders expressions using commutativity and associativity to expose opportunities for other optimizations [12]. It reassociates complex expressions to expose loop invariant subexpressions.

9. Register Coalescing (`coalesce`): This pass coalesces register-to-register copies using the interference graph of a Chaitin style register allocator [16].

10. Operator Strength Reduction (`strength`): This pass performs operator strength reduction using an SSA-based algorithm due to Cooper, Simpson, and Vick [24]. The algorithm differs from prior work in that it finds secondary effects in a single pass—it need not be repeated in an iterative fashion.

It should be noted that a simple global value-numbering pass that renumbers registers is run before `partial` and `lazy` to ensure that the input ILOC conforms to the naming requirements of these passes. Additionally, `shape` is run before `partial` to enforce other code shape requirements.

To gather execution statistics and test for correctness. the ILOC code is translated into C code. This C code is instrumented to tabulate statistics, including the number of operations executed during a run of the program. The C code is then compiled and run.

This work focuses on finding optimization sequences that result in reduced code size. Typically, this would refer to the number of bytes required to store the object code. The analog to this in our ILOC framework is the *static operation count*. That is, the number of ILOC operations in the optimized code. A reasonable approximation to the number of bytes the code requires can be made by assuming each ILOC operation occupies a four byte word.

D.2 The genetic algorithm

Genetic algorithms are basically search algorithms designed to mimic the process of natural selection and evolution in nature. To begin, we need to define a few terms. A *population* consists of a fixed number of members, or *chromosomes*. Each chromosome is a fixed-length string of *genes*. The *fitness* of a chromosome is some measure of how desirable it is to have that chromosome in the population. A *generation* is a time step in which several events occur. Some of the most "unfit" chromosomes die and are removed from the population. To replace these, some number of *crossover* operations are applied to the population. A crossover is an operation analogous to mating or gene splicing. It combines part of one chromosome with part of another chromosome to create a new chromosome. Finally, some amount of *mutation* occurs in the population, in which genes are changed randomly with some (typically low) probability. It is also possible to have *elitism* in the population in which some of the most fit genes are immune from mutation between generations.

Using these definitions, we define a GA to find customized compiler optimization sequences. As described in the previous section, we have ten optimizations at our disposal. An optimization is analogous to a gene. We have selected a chromosome length of twelve genes. This corresponds to performing a sequence of twelve optimizations. (The selection of twelve is arbitrary. The fixed sequence used as a default in our system has length 12. We describe it in Section D.3.) We have ten different genes and chromosomes of length twelve for a total solution-space size of 10^{12}. We set a population size of twenty chromosomes. Larger populations do not produce appreciably different results. The fitness value for a particular chromosome (optimization sequence) is the size of the object code produced when that optimization sequence is applied to the source code. Unlike typical genetic algorithms, we desire chromosomes with *low* fitness values since these correspond to smaller object codes.

Each generation in our experiments consists of the following steps:

1. Compute a fitness value for each chromosome. The code being compiled is passed through the front end of the compiler and then optimized according to the optimization sequence defined by the chromosome. After optimization, we remove empty basic blocks from the code (*i.e.*, a final pass of the `clean` optimization) and perform Briggs style register allocation with 32 integer and 32 floating-point registers [15]. The number of static ILOC operations in the final code is recorded as the fitness value for that chromosome.

2. The chromosomes are sorted by fitness values from lowest to highest. The population is split into a lower and upper half, based on fitness value, each half consisting of 10 chromosomes. The upper half consists of the 10 chromosomes with the lowest fitness values.

3. The chromosome with the highest fitness value (worst performance) is removed from the population. Three additional chromosomes are chosen at random from the lower half and removed from the population.

4. To fill the four vacancies in the population, new chromosomes are generated using the "crossover" operation. Two "parent" chromosomes are randomly chosen from the upper half of the population. The first six genes of one chromosome are concatenated with the last six genes of the other chromosome and vice versa, creating two new chromosomes.[12] This operation is performed twice, to produce four new chromosomes.

5. Next, fifteen chromosomes are subjected to mutation. The best performing chromosome is exempted from mutation, a form of elitism. The four chromosomes created by crossover in the previous step are also exempted. In the fifteen chromosomes subject to mutation, each gene is considered. For a chromosome in the lower half of the population, mutation occurs with a probability of 0.1 (or 10 percent). For a chromosome in the upper half of the population, the probability of mutation is reduced to 0.05 (or 5 percent). To mutate a gene, it is replaced with a randomly selected gene.

6. Duplicate chromosomes are removed from the population and replaced with random chromosomes.

This process is repeated for 1000 generations, and we keep track of the best chromosome found over the course of the run.

When computing fitness values, we also simulate a run of the optimized code to test for correctness. If any error was encountered during the optimization or the execution, a fitness value of infinity is assigned to the chromosome.

Since we run the code each time we optimize it, we also keep track of the number of operations executed during the run. Each time any ILOC operation is executed,

[12]The current version of our algorithm splits the chromosome in a random position. This change does not noticeably alter our results.

the *dynamic* operation counter is incremented. The number of dynamic operations is a crude estimate of the number of CPU cycles needed to execute the program. Operations that typically require more time to execute (like divide) are *not* weighted in this count. We use these dynamic operation counts as secondary fitness values. When sorting the chromosomes by fitness value (static operation count), we break any ties by looking at the dynamic operation count. Thus, the final optimization sequence returned by the algorithm has the lowest dynamic operation count of all optimization sequences of the given fitness.

Note that the runtime bottleneck of this genetic algorithm process is the computation of the fitness values. Also notice that at least one chromosome (the one with the best fitness value) remains unchanged from one generation to the next. Thus, there is no need to recompute the fitness value for that chromosome. It is likely that many of the other chromosomes may not be mutated. Over the course of 1000 generations many chromosomes get "regenerated". On average, over the whole run of 1000 generations, we found that about 45 per cent of chromosomes already had fitness values computed previously. To improve the runtime performance of the algorithm we keep a hash table of fitness values indexed by chromosome. When computing fitness values, we first check the hash table to see if the fitness value for that chromosome is already available. If so, there is no need to recompute it and we move on to the next chromosome.

D.3 Experimental Results

To test the efficacy of our GA, we used it to find optimization sequences for several benchmark programs. The FORTRAN programs used in the experiments were `fmin`, `rkf45`, `seval`, `solve`, `svd`, `urand`, and `zeroin` from the FMM benchmark suite [36] and `tomcatv` from SPEC. The C codes used were `adpcm`, which performs adaptive differential pulse code modulation, `compress`, which is the UNIX file-compression utility, `fft`, a fast Fourier transformation algorithm, `dfa`, an implementation of the Knuth-Morris-Pratt string-matching algorithm, `dhrystone`, a common hardware benchmark, and `nsieve`, code that implements the Sieve of Eratosthenes as written by Al Aburto.

The GA described in the previous section is implemented with a Perl script. The script determines the optimization sequence according to the algorithm and then calls the program `ctest`, which is the optimization-testing program for our research compiler. `Ctest` runs the corresponding optimization passes on the ILOC code, including

	Unoptimized		GA results				
Benchmark	static	dynamic	static	% red.	dynamic	% red.	gen. found
adpcm	438	17221981	351	19.9	12290460	28.6	6
compress	1753	8402188	1318	24.8	5545480	34.0	(77,79)
dfa	1744	842382	1107	36.5	496164	41.1	806
dhrystone	760	4920264	536	29.5	3200191	35.0	(22,920)
fft	2415	18339859	1757	27.2	14574279	20.5	2
fmin	374	2192	187	50.0	963	56.1	32
nsieve	353	761244374	202	42.8	539954218	29.1	(0, 189)
rkf45	1525	511251	745	51.1	201470	60.6	74
seval	1061	3594	288	72.9	842	76.6	39
solve	1023	2729	437	57.3	1029	62.3	(33,58)
svd	2087	13049	972	53.4	4760	63.5	26
tomcatv	2250	1379982621	551	75.5	232833969	83.1	90
urand	204	1563	93	54.4	613	60.1	(0,18)
zeroin	273	1815	150	45.1	809	55.4	(239,270)

Table D.1 Optimizing with the GA.

one additional pass of `clean` and the register allocator. Finally, `ctest` has the resultant ILOC code converted to C, compiled, and run. The number of static and dynamic operations are recorded and reported back to the Perl script for use as fitness values and secondary fitness values, respectively.

D.3.1 Solutions found by the GA

Table D.1 presents the results of running our GA on the benchmark codes. Column two shows the unoptimized code size (number of static operations), and column three the unoptimized dynamic operation count. (Unoptimized code has been run through one pass of `clean` and the register allocator.) Column four reports the smallest code size returned by the GA, and column six reports the lowest dynamic operation count found for codes having the size reported in column four. (For example, several solutions resulting in object code of size 187 were found for `fmin`. The best dynamic operation count of those solutions was 963.) Columns five and seven report the percentage reduction of static operations and dynamic operations, respectively, over the unoptimized case. Recall that the algorithm is tuned to find the solution with the lowest static size and is only secondarily concerned about execution time results. The last column labeled "gen. found " reports the generation in which the GA found the reported solution. A pair of numbers in this column indicates that the first occurrence of a solution producing the result in column four was found in a different generation

Benchmark	GA sequence	reduced sequence
adpcm	tonosdnzscno	osnzc
compress	ncsnzosvndvs	nczvnds
dfa	rztonvncodvs	rztcodvs
dhrystone	covolcnocdvc	covlcodvc
fft	ovocdvscnnos	ovcdvs
fmin	tdcotcscnnvo	dcsvo
nsieve	snzdvvcdsvss	nzdvcs
rkf45	ovndtdcvsndd	ovndtcvsd
seval	rnldoncvconv	ldovco
solve	dondvnsdsdcv	ovndcv
svd	odvdvdsnnnss	odvs
tomcatv	dntvvccocvdv	tvcodv
urand	cnottcdtvooc	nodvc
zeroin	rsdosvncnsss	rsosvcs

Table D.2 Optimization sequences found by the GA.

than the solution resulting in the best dynamic instruction count as reported in column six. Thus, the first number in the pair is the first generation where a solution resulting in minimal code size appeared. The second number is the generation where the final solution appeared.

The results in Table D.1 show the overall impact of optimization on code size and speed for our compiler. Improvements in code space range from 20 to 75 percent. Improvements in speed range from 20 to 83 percent. These numbers show the potential for improvement in our compiler. The size and speed of the unoptimized code depends heavily on the quality of code emitted by the compiler's front end. In our compiler, the code is carefully shaped to expose opportunities for optimization. This may produce slower unoptimized code than necessary. Other authors, however, have reported total improvements that are comparable [74].

Table D.2 reports the optimization sequences found by the GA which give the results reported in Table D.1. Each letter in the sequence corresponds to one ILOC optimization. The letters correspond to `ctest` options indicating `ctest` should perform that optimization. The mapping from these letters to their corresponding optimizations is given in Table D.3.

The last column of Table D.2 is a reduced sequence that produces the same results as the sequence found by the GA. These smaller sequences were found by an automatic tool, starting from the sequence returned by the GA and removing optimizations from the sequence one at a time. A simple work-list algorithm is used, the work-list being initialized with the sequence returned by the GA. When a sequence s of length l is removed from the work-list, l sequences of length $l-1$ are generated by removing a single optimization from s. Each shorter sequence is tested and if it produces the same results as the original sequence it is added to the work-list. A hash table of tested solutions is maintained to avoid testing a solution more than once. The process continues until the work-list is empty, and the shortest solution found is returned. Smaller sequences producing the same results may exist.

It is interesting to note which optimizations appear frequently in the sequences (especially the reduced sequences) and which ones do not. Peephole optimization (`combine` or `o`) appears in twelve of the fourteen reduced sequences. Register coalescing (`coalesce` or `s`) and dead code elimination (`dead` or `d`) appear in most sequences. This is not surprising, since these optimizations almost always remove operations from the code. Global value numbering (`valnum` or `v`) was also present in most sequences.

Other transformations were less prevalent. Not surprisingly, partial redundancy elimination (`partial` or `l`) is seen in very few sequences (only twice in `seval`, and `dhrystone`) This is due to the fact that `partial` frequently adds operations to the code in order to reduce the lengths of some paths through the code. The effects of `partial` can reduce run times but tend to increase code size. Another code motion technique, lazy code motion (`lazy` or `z`), appeared in only four cases. `Lazy` can increase code size but tends to produce fewer long register live ranges than `partial`. It should be noted that `valnum` was run with an option that performs value driven code motion as described in Simpson's thesis [76]. Operator strength reduction (`strength` or `t`) appeared in only three reduced sequences (`dfa`, `rkf45`, and `tomcatv`), but appeared more frequently in non-reduced sequences. This suggests that the optimization does not typically increase code size, but tends not to reduce it either. Reassociation (`shape` or `r`) appeared in only two cases (`dfa` and `zeroin`).

There was some repetition of optimizations, but it was not very consistent across benchmarks. Repeated optimizations occurred in the reduced sequences only a few times: `valnum` in `dhrystone`, `fft`, `rkf45`, `solve`, and `tomcatv`; `combine` in `dhrystone` and `seval`; `clean` in compress, `dead` in `rkf45`; `coalesce` three times in `zeroin`; `cprop` three times in `dhrystone`.

gene	optimization
c	`cprop`
d	`dead`
l	`partial`
n	`clean`
o	`combine`
r	`shape`
s	`coalesce`
t	`strength`
v	`valnum`
z	`lazy`

Table D.3 Mapping of `ctest` options (genes) to corresponding optimizations.

The GA required about 1 day (on a Sparc Ultra-10) to run 1000 generations for most benchmarks. This running time is dominated by the time required to compute fitness values, since that requires actually optimizing the code and running it. Some time could be saved by modifications to our testing program `ctest`. With the exception of `dfa` and `zeroin`, the best solution in terms of code size was found by the GA in less than 100 generations. This suggests that with some fine tuning of the GA parameters, the GA would need to be run for far fewer than 1000 generations to produce quality results. Also, we were able to find reduced sequences of nine or fewer optimizations in all cases. This suggests that a smaller chromosome size could be used by the algorithm, further improving total running time.

D.3.2 Performance of GA against a fixed sequence

Typical compilers present a variety of optimization levels to the user. Each of the optimization levels represents some fixed sequence of optimizations that the compiler applies to the code.

In this section, we investigate the performance of the GA against fixed optimization sequences. We compare the solutions found by the GA to two fixed sequences. The first is used as the default in our system for producing highly optimized code. This sequence was designed to produce code with low *dynamic* operation counts, not necessarily low static code size. This sequence is `rvzcodtvzcod`. The second sequence is based on our observations of optimization sequences returned by the GA. We tried

	rvzcodtvzcod		nodvcodvs	
Benchmark	operations	%	operations	%
adpcm	362	3.0	356	1.4
compress	1412	6.7	1325	0.5
dfa	1168	5.2	1145	3.3
dhrystone	574	6.6	544	1.5
fft	1973	10.9	1757	0.0
fmin	203	7.9	198	5.6
nsieve	227	11.0	202	0.0
rkf45	832	10.5	751	0.8
seval	313	8.0	297	3.0
solve	609	28.2	438	0.2
svd	1641	40.8	973	0.1
tomcatv	770	28.4	565	2.5
urand	93	0.0	93	0.0
zeroin	158	5.1	154	2.6

Table D.4 Comparison of GA solutions and fixed sequence solutions – static operation counts. Percentages indicate improvement of GA solution over the fixed sequence.

several sequences using optimizations that were prevalent in the reduced sequences of GA solutions and repeated some of them. The best fixed sequence we developed was `nodvcodvs`. `Clean` and the register allocator were run after each optimization sequence. Static code size numbers are presented in Table D.4 and dynamic operation counts in Table D.5. The percentages indicate the percent reduction given by the GA compared to the fixed string. A negative value indicates that the fixed string performed better. Notice that the GA performed very well against the first sequence in terms of static code size. The second sequence we developed performed quite well against the GA but still lost by as much as 5.6 percent in terms of static code size.

The first sequence (our default sequence) found several solutions resulting in code executing fewer operations at run time, but also had quite a few losses. The GA did slightly better than the second sequence in terms of dynamic operation counts with only a couple of losses, and won by as much as 14.5 percent. Remember that the GA is specifically tuned to find sequences that result in small code. Improved running times are only a secondary goal of the algorithm.

	rvzcodtvzcod		nodvcodvs	
Benchmark	operations	%	operations	%
adpcm	11965360	-2.7	12440360	1.2
compress	7494982	26.0	5548268	0.1
dfa	546396	9.2	511349	3.0
dhrystone	3390200	5.6	3270193	2.1
fft	15088485	3.4	14574279	0.0
fmin	955	-0.8	947	-1.7
nsieve	570608938	5.4	554450986	2.6
rkf45	214146	5.9	215620	6.6
seval	809	-4.1	985	14.5
solve	1099	6.4	1033	0.4
svd	4731	-0.6	4758	-0.0
tomcatv	186355888	-24.9	239463612	2.8
urand	631	2.9	613	0.0
zeroin	829	2.4	811	0.2

Table D.5 Comparison of GA solutions and fixed sequence solutions – dynamic operation counts. Percentages indicate improvement of GA solution over the fixed sequence.

Small changes in the fixed sequence can have dramatic impacts. For example, when the `shape` transformation was inserted into the sequence we developed above, the code size for `dfa` was reduced to 1109 operations from 1145. However, the same sequence *increased* the code size for `tomcatv` to 737 operations, from 565. This emphasizes the potential improvement of using customized optimization sequences for different programs.

D.3.3 The GA vs. random probing.

We would like some assurance that the GA is doing something useful. That is, can the same results be found quickly by simply trying random optimization sequences? To test this, we performed extensive testing on two benchmarks, `fmin` and `adpcm`. We ran 25 experiments using the GA with different random number seeds. We forced the GA to stop after it found a solution of the same quality as reported in Table D.1. The number of unique solutions tested was reported. We also ran 25 experiments where we randomly choose an optimization sequence consisting of 12 optimizations from the pool. Again we stopped after we found a solution of the quality reported in Table D.1. The number of unique solutions tested was reported. One would hope that average number of solutions tested by the GA would be less than the average number tested by random sampling to find the same quality solution.

The average number of solutions tested by the random algorithm was 351.8 for `adpcm` with a standard deviation of 311.8. The GA tried an average of 198.7 solutions with a standard deviation of 129.1. At the .05 level of significance the random algorithm tests on average more than 42 solutions more than the GA. On average 3330.9 random solutions were tested for `fmin` with a standard deviation of 2513.0. The GA tested on average 431.6 solutions on `fmin` with a standard deviation of 176.0. At the .05 level of significance we conclude that the random algorithm tests on average greater than or equal to 2070 more solutions than the GA.

Since the overhead required to perform the GA algorithm vs. random testing is insignificant compared to the time required to actually test solutions, we conclude that the GA converges on quality solutions faster than mere random probing of the solution space. That is, given the same fixed amount of run time, the GA on average would produce a better solution than random probing.

D.3.4 Module specific optimization sequences

Different programs have different optimization requirements. Extending that idea, we could conclude that different modules or functions of the same program may have different optimization needs. As a preliminary test of that hypothesis, we ran our GA on the separate modules of several of our benchmarks. That is, the GA was allowed to produce different optimization sequences for each module. This test was performed on `rkf45`, `adpcm`, `fft`, and `dhrystone`. `Rkf45` has three ILOC functions. Each one was optimized separately. The total code size for the three was 741 operations; a 1.5 percent reduction over the best fixed sequence and a 0.9 percent improvement over running the GA on the whole program. `adpcm` has six functions and we computed optimization sequences for each one. We saw no reduction in the static size of the code over running the GA on the whole program. However, separate optimization did lead to a 2.0 percent improvement in dynamic operation count. `fft` contains ten functions that we grouped into seven separate modules. Applying the GA to each module separately resulted in a total program size of 1743 operations; a 0.8 percent reduction from the whole program GA solution. A slight increase in dynamic operation count was observed. `Dhrystone` consists of two C modules. Optimizing them separately resulted in a total size of 533 operations or a reduction of 0.6 percent over running the GA on the entire program. No change in dynamic operation count was observed.

D.4 Conclusions and Observations

In this chapter, we described a genetic algorithm that is designed to find compiler-optimization sequences resulting in reduced static code sizes. The algorithm was tested on a variety of benchmarks, and the solutions generated by the algorithm were compared to no optimization and fixed optimization sequences. The results of these experiments are summarized below:

- The GA found optimization sequences that dramatically reduced the code size (static operation count) compared with no optimization. The GA produced optimization sequences that resulted in smaller code than the compiler's default sequence. In most cases, the smaller code executed fewer instructions, as well.
- By observing patterns in the GA solution sequences, we were able to construct a fixed optimization sequence that generated up to 40 percent smaller codes than the standard sequence used in our compiler. In many cases, the resulting code is also faster, up to 26 percent faster. This suggests that the GA can be used to give a "tune-up" to an existing compiler by finding a better optimization sequence that uses the existing passes.
- When applied to a single program, the GA generated a program-specific optimization sequence that was often better than either the compiler's old default sequence or the new sequence described in Section D.3.2. This shows that the "best" sequence differs from program to program. In particular, if a program is too large for the available space and the space cannot be enlarged, it may be worth running a GA for several hours to obtain a custom tailored optimization sequence.

We believe there are several avenues for continued research based on this work:

- The number and kind of optimizations available to the GA can be increased.
- Several of the optimizations allow options that force the optimization to use different algorithms or heuristics. These options could also be put under the

control of the GA. For example, one "gene" could correspond to the `valnum` optimization as described. A separate gene could correspond to running `valnum` without performing value driven code motion.

- There are many parameters of the algorithm that can be modified. The parameterization presented was one of several tried and produced the best results of those tested. Further changes may produce better results or help the algorithm to converge more quickly.

- The preliminary work into module specific optimization showed modest improvements. More study needs to be done to see if separate optimization for different modules in the same program is a viable technique for reducing code size.

- We described how to use the GA to reduce code size. However, by simply changing the fitness function, the algorithm can be tuned to find solutions resulting in other desirable properties such as reduced running time or power consumption.

We believe that using a GA to find optimization sequences could be a valuable technique for embedded systems compilers. When compile time is plentiful, an embedded system software designer may want to consider a GA approach to final optimization, especially when code size is of critical importance. A GA can be used to find a high quality optimization sequence for particular set of optimizations.

This work was supported by DARPA through USAFRL grant F30602-97-2-298. The motivation for this work came from a talk by Andy Nisbet on his GAPS genetic algorithm for parallelizing FORTRAN loops. Many thanks to past and present members of the Massively Scalar Compiler Group for the development and programming of the optimizations used in this work. These include Preston Briggs, Tim Harvey, John Lu, Rob Shillner, Taylor Simpson, and Linda Torczon.

Bibliography

[1] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, September 1995.

[4] Steven John Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Colorado State University, Fort Collins, Colorado, 1991.

[5] David Bernstein. An improved approximation algorithm for scheduling pipelined machines. In *Proceedings of the International Conference on Parallel Processing*, pages 430–433, August 1988.

[6] David Bernstein, Doron Cohen, and Hugo Krawczyk. Code duplication: An assist for global instruction scheduling. *SIGMICRO Newsletter*, 22(12):103–113, December 1991. *Proceedings of the 24th Annual International Symposium on Microarchitecture*.

[7] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[8] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Notices*, 26(6):241–255, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[9] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth*

International Conference on Architectural Support for Programming Languages and Systems, pages 122–131, April 1991.

[10] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. CRAIG: A practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[11] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994.

[12] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[13] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, June 1997.

[14] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[15] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[16] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[17] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

[18] Peter Cheeseman, Bob Kaneksky, and William M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, 1991.

[19] Edward G. Coffman, editor. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[20] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[21] Keith D. Cooper and Philip J. Schielke. Non-local instruction scheduling with limited code growth. In *Languages, Compilers, and Tools for Embedded Systems ACM SIGPLAN Workshop LCTES '98, Montreal, Canada, June 19-20, 1998, Proceedings*, pages 193–207. Springer Verlag, June 1998.

[22] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Languages, Compilers, and Tools for Embedded Systems ACM SIGPLAN Workshop LCTES '99, Atlanta, Georgia, May 23-25, 1999, Proceedings*, pages 193–207. ACM, July 1999.

[23] Keith D. Cooper and L. Taylor Simpson. Scc-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, 1995.

[24] Keith D. Cooper, L. Taylor Simpson, and Chris A Vick. Operator strength reduction. Technical Report CRPC-TR95635-S, Center for Research on Parallel Computation, Rice University, 1995.

[25] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of AAAI-94*, pages 1902–1097, 1994.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[27] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[28] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[29] Lawrence Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[30] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[31] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *SIGPLAN Notices*, 28(5):29–38, May 1993.

[32] Kemal Ebcioglu and Toshio Nakatani. A new compilation technique for parallelizing regions with unpredictable branches on a VLIW architecture. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[33] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, 1986.

[34] Joseph A. Fisher. *The optimization of horizontal microcode within and beyond basic blocks : an application of processor scheduling with resources.* PhD thesis, New York University, New York, 1979.

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

[36] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[37] Stefan Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and supression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, July 1994.

[38] F. Gasperoni. Compilation techniques for VLIW architectures. Technical report, Courant Institute of Mathematical Sciences, March 1989.

[39] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[40] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesly, 1989.

[41] M. C. Golumbic and V. Rainish. Instruction scheduling beyond basic blocks. *IBM Journal of Research and Development*, 34(1):93–97, January 1990.

[42] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, July 1988.

[43] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Jorunal*, pages 1563–1581, 1966.

[44] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[45] J. Heller. Sequencing aspects of multiprogramming. *Journal of the Association for Computing Machinery*, 8:426–439, 1961.

[46] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[47] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, November 1961.

[48] Brett L. Huber. Path-selection heuristics for dominator-path scheduling. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, Michigan, 1995.

[49] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscaler compilation. *Journal of Supercomputing – Special Issue*, 7:229–248, July 1993.

[50] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[51] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[52] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[53] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *SIGPLAN Notices*, 29(6):147–158, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[54] Sanjay M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[55] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[56] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. ACM Computing Surveys, pages 261–294, September 1980.

[57] S. Lin and B. Kernighan. An effective heuristic for the traveling salesman problem. *Operations Research*, 21, 1973.

[58] P. Geoffrey Lowney, Stephen M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, Robert P. Nix, J. S. O'Donnell, and J. C. Ruttenburg. The Multiflow trace scheduling compiler. *Journal of Supercomputing – Special Issue*, 7:51–142, July 1993.

[59] G. K. Manacher. Production and stabilization of real-time task schedules. *Journal of the Association for Computing Machinery*, 14(3):439–465, July 1967.

[60] Melanie Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

[61] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[62] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[63] Motorola. *MPC750 RISC Microprocessor User's Manual*, August 1997. MPC740UM/AD.

[64] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Report TR 698, Courant Institute, July 1995.

[65] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[66] Andy P Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *Poster Session at The International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '98*, 1998.

[67] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. *Proceedings of Supercompting '93*, pages 804–813, November 1993.

[68] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on risc machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, September 1993.

[69] C. H. Papadimitiou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3):405–409, August 1979.

[70] Schlomit S. Pinter. Register allocation with instruction scheduling: A new approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[71] R.T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, NY, USA, December 1959.

[72] C. V. Ramamoorthy and M. Tsuchiya. A high-level language for microprogramming. *IEEE Transactions on Computers*, 23(8):791–801, August 1974.

[73] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing – Special Issue*, 7:9–50, July 1993.

[74] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, pages 660–676, November 1980.

[75] J. J. Shieh and Christos A. Papachristou. On reordering instruction streams for pipelined computers. *SIGMICRO Newsletter*, pages 199–206, August 1989. *Proceedings of the 22nd Annual Workshop in Microprogramming and Microarchitecture (MICRO-22)*.

[76] L. Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, May 1996.

[77] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. *SIGMICRO Newsletter*, 22(12):93–102, December 1991. *Proceedings of the 24th Annual International Symposium on Microarchitecture*.

[78] Philip H. Sweany. *Inter-Block Code Motion without Copies*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.

[79] Philip H. Sweany and Steven J. Beaty. Post-compaction register assignment in a retargetable compiler. *SIGMICRO Newsletter*, 21(12):107–116, December 1990. *Proceedings of the 23rd Annual Workshop in Microprogramming and Microarchitecture (MICRO-23)*.

[80] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling – A global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, December 1992. *Proceedings of the 25th Annual International Symposium on Microarchitecture*.

[81] Robert E. Tarjan and John H. Reif. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1982.

[82] Texas Instruments. *TMS320C62xx CPU and Instruction Set Reference Guide*, January 1997. Literature Number: SPRU189A.

[83] Roy F. Touzeau. A Fortran compiler for the FPS-164 Scientific Computer. *SIGPLAN Notices*, 19(6):48–57, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[84] M. Tsuchiya and M. J. Gonzales. An approach to optimization of horizontal microprograms. *(SIGMICRO) 7th Annual Worksop on Microprogramming Preprints*, pages 85–90, September 1974.

[85] Jeffrey D. Ullman. Complexity of sequencing problems. In E. G. Coffman, editor, *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, 1976.

[86] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[87] Graham Wood. On the packing of micro-operations into micro-instruction words. *SIGMICRO Newsletter*, pages 51–55, December 1978. *Proceedings of the 11th Annual Microprogramming Workshop.*

[88] S. S. Yau, A. C. Schowe, and M. Tsuchiya. On storage optimization of horizontal microprograms. *(SIGMICRO) 7th Annual Worksop on Microprogramming Preprints*, pages 98–106, September 1974.

[89] W. Zhang and T. G. Dietterich. A reinforcement learning apporach to job-shop scheduling. In *Proceedings of IJCAI-95*, 1995.

[90] Monte Zweben, Eugene Davis, Brian Daun, and Michael J. Deale. Scheduling and rescheduling with iterative repair. IEEE Transactions on Systems, Man, and Cybernetics, 23(6):1588–1596, 1993.

[91] Monte Zweben, Eugene Davis, Brian Daun, Ellen Drascher, Michael Deale, and Megan Eskey. Learning to improve constraint-based scheduling. *Artificial Intelligence*, 58:271–296, 1992.