May 1991

UILU-ENG-91-2228
CRHC-91-18

Center for Reliable and High-Performance Computing

THE IMPORTANCE OF PREPASS CODE SCHEDULING FOR SUPERSCALAR AND SUPERPIPELINED PROCESSORS

Pohua P. Chang
Daniel M. Lavery
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2228 (CRHC-91-18)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION NSF, NCR, AMD, NASA, CSRD, IBM
6c. ADDRESS *(City, State, and ZIP Code)* 1101 W. Springfield Avenue Urbana, IL 61801		7b. ADDRESS *(City, State, and ZIP Code)* Washington DC 20250 Dayton Ohio 45420 Langley VA
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-91-J-1283 NASA NAG 1-613

8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS			
7b	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors

12. PERSONAL AUTHOR(S) Chang, Pohua P., Daniel M. Lavery, and Wen-Mei Hwu

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ________ TO ________	14. DATE OF REPORT *(Year, Month, Day)* 91-05-28	15. PAGE COUNT 47

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Code Scheduling, control-intensive programs, optimizing compiler, register allocation, superpipelined processors, and superscalar processors

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Superscalar and superpipelined processors provide hardware that has the ability to execute many instructions in parallel. In order for this potential to be translated into the speedup of real programs, the compiler must be able to schedule instructions so that they can be overlapped without interlock delays and resource conflicts. Previous work has shown that prepass code scheduling helps to produce a better schedule for scientific programs. But the importance of prescheduling has never been demonstrated for control-intensive non-numerical programs. These programs are significantly different from the scientific programs because they contain frequent branches and they require global scheduling in order to find enough independent instructions.

continued on back

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* 22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. continued

In this paper, the code scheduler of the IMPACT-I C compiler is described and used to study the importance of prepass code scheduling for a set of production C programs. It is shown that, in contrast to the results previously obtained for scientific programs, prescheduling is not important for compiling control-intensive programs to the current generation of superscalar and superpipelined processors. However, if the some of the current restrictions on upward code motion are removed in future architectures, prescheduling substantially improves the execution time of this class of programs on both superscalar and superpipelined processors.

The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors

Pohua P. Chang Daniel M. Lavery Wen-mei W. Hwu*

May 24, 1991

Abstract

Superscalar and superpipelined processors provide hardware that has the ability to execute many instructions in parallel. In order for this potential to be translated into the speedup of real programs, the compiler must be able to schedule instructions so that they can be overlapped without interlock delays and resource conflicts. Previous work has shown that prepass code scheduling helps to produce a better schedule for scientific programs. But the importance of prescheduling has never been demonstrated for control-intensive non-numerical programs. These programs are significantly different from the scientific programs because they contain frequent branches and they require global scheduling in order to find enough independent instructions.

In this paper, the code scheduler of the IMPACT-I C compiler is described and used to study the importance of prepass code scheduling for a set of production C programs. It is shown that, in contrast to the results previously obtained for scientific programs, prescheduling is not important for compiling control-intensive programs to the current generation of superscalar and superpipelined processors. However, if some of the current restrictions on upward code motion are removed in future architectures, prescheduling substantially improves the execution time of this class of programs on both superscalar and superpipelined processors.

Index terms - Code scheduling, control-intensive programs, optimizing compiler, register allocation, superpipelined processors, superscalar processors.

*The authors are with the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, Illinois, 61801. Daniel Lavery is also with the Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, Illinois, 61801.

1 Introduction

Current high-performance processors use hardware techniques to exploit instruction-level parallelism. Pipelining is common, and many designs are capable of executing nearly one instruction per cycle. Performance can be boosted further either by executing more than one instruction per cycle, or by reducing the length of the clock cycle. Superscalar processors fetch, decode, and execute more than one instruction per cycle by providing multiple functional units and datapaths. Superpipelined processors divide the pipeline into smaller segments that have less delay, allowing the clock cycle to be shortened. In order for the full performance to be extracted from these parallel microarchitectures, some method must be used to minimize the stalls caused by the control and data dependencies between instructions. As the pipelining depth or the instruction issue rate increases, these stalls become more costly.

Code scheduling is a technique that tries to rearrange the instruction sequence to minimize the execution time. Usually, code scheduling is performed after register allocation (postpass or postscheduling). However, the register allocator introduces extra dependencies whenever it reuses registers. These extra dependencies restrict the ability of the code scheduler to move instructions to their desired positions. On the other hand, if code scheduling is performed before register allocation (prepass or prescheduling), the register lifetimes may be lengthened, which may increase the amount of spill code added by the register allocator.

In previous work, Goodman and Hsu [1] showed that a prepass scheduler can keep track of the number of available registers to avoid introducing excessive spill code. Hwu and Chang [2] showed that a prescheduling, register allocation, postscheduling sequence extracts more performance from scientific benchmarks than postscheduling alone. Both of these results apply to scientific programs with code scheduling and register allocation performed within large basic blocks. The importance of prescheduling has never been demonstrated for control-intensive non-numerical programs.

For the study reported in this paper, code scheduling is performed before and after register allocation. As it reorganizes the instructions, the prescheduler tries to control the increase in the register lifetimes, helping the register allocator to minimize the number of registers used. We compile a set of production C programs using the IMPACT-I C compiler in order to examine the effectiveness of prescheduling for control-intensive non-numerical programs. It is important to evaluate prescheduling on this class of codes for two reasons. First, compared to the scientific applications studied earlier, these C programs have frequent branches, creating small basic blocks in which there is only a little parallelism. Code scheduling and register allocation are performed globally in order to find more parallelism and to reduce the register save and restore overhead. It is not clear that the results based on local scheduling and register allocation for scientific codes are directly applicable here. Second, even with global scheduling and register allocation, these control-intensive programs have less inherent parallelism than scientific applications. The advantage of prescheduling for programs with limited parallelism needs to be demonstrated.

This paper also empirically evaluates the advantages of prescheduling for the superscalar and superpipelined implementations of current and future architectures. We compile the set of C benchmarks to several different parallel implementations of a base architecture and calculate the execution time and the number of dynamic memory references from the schedule. For each case, we compile once with both prescheduling and postscheduling turned on and once with only postscheduling turned on in order to compare the two methods. In order for these parallel microarchitectures to speed up the execution of control-intensive programs, the compiler must be able to generate efficient code with sufficient parallelism to utilize them. The study done in this paper shows that for architectures that relax the current restrictions on upward code motion, prescheduling helps to achieve this goal.

In other related work, Hennessy and Gross [3] provided a good description of the code scheduling problem and a scheduling algorithm. Fisher [4] and Ellis [5] described a very effective global scheduling algorithm called trace scheduling. A paper by Chaitin [6] presented the graph-coloring-based register allocation algorithm on which our global register allocator is based.

This paper is organized as follows. Section 2 gives the necessary background on prescheduling and postscheduling, our C compiler, and its register allocator and scheduler. The experimental methodology and the results are discussed in Section 3. The conclusion is presented in Section 4.

2 Background

2.1 Prepass vs. Postpass Code Scheduling

The code scheduler has one primary goal: to rearrange the instructions so that the code sequence is executed in the smallest number of cycles. For example, to avoid stalls due to an instruction with a long latency (such as a load or a multiply), the scheduler will try to move it upward in the code so that its result is ready in time for use by a subsequent instruction. While reorganizing the code, it must preserve the correctness of the original program with respect to the data and control dependencies. In this work, it is assumed that there is no dynamic code scheduling, but the hardware does have interlocks and register renaming to handle pipeline hazards. All of the instruction latencies and the type and number of functional units are visible to the code scheduler.

The dependencies are expressed in the form of a dependence graph. Prior to register allocation, the only dependencies expressed in the graph result from the operations necessary to implement the computation specified by the source program [1]. Because temporary variables are written only once, the only dependencies related to them are flow (read-after-write) dependencies. For the user level variables, there may be flow, anti- (write-after-read) [7], or output (write-after-write) dependencies.

During register allocation, dependencies resulting from the reuse and spilling of registers are added to the dependence graph. When a register is reused, anti- and output dependencies are created because the last read or write of the variable currently occupying the register is followed by the write of the new variable. When a register is spilled, the same kinds of dependencies are created because of the register saves and restores. In addition, flow and anti- dependencies are created because of the memory read and write.

Code scheduling can be performed either before or after register allocation, or both. No matter when it is done, the dependencies in the initial code sequence constrain the code scheduler, in some cases preventing it from moving an instruction to its desired location. If code scheduling is performed after register allocation, it is additionally restricted by the extra dependencies resulting from the reuse and spilling of registers described earlier. As a consequence, the instructions may not be moved around as effectively as they could be.

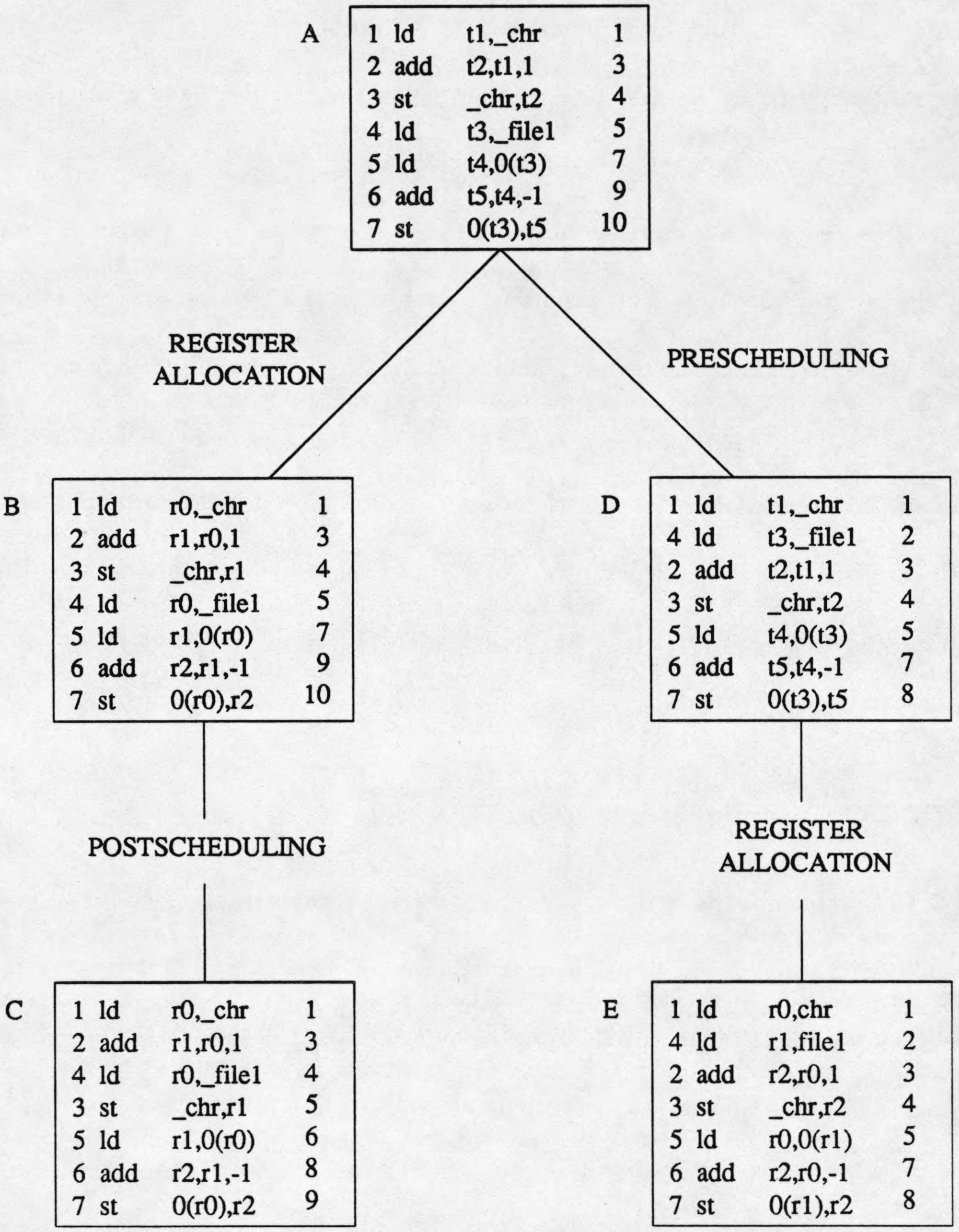
One way around this is to perform prepass code scheduling. Then the scheduler can move

[1] This assumes that the single assignment rule is used for compiler generated temporaries. Depending on the amount of optimization performed by the compiler before code scheduling, the number of instructions used and the dependency pattern created may vary. In any case, there is some given dependency pattern that the code scheduler must work with.

the instructions close to their desired positions without the hindrance of the register recycling dependencies. However, if the prepass code scheduler is not careful about moving instructions, it can greatly increase the register lifetimes. For example, in order to avoid delays due to a load instruction the code scheduler tries to insert useful operations between the load and the instruction which uses the value loaded. This increases the lifetime of the destination register of the load, increasing the chance that the register will have to be spilled. If the scheduler inserts too many instructions, then the value loaded will be available sooner than it needs to be and will take up space in the register for a longer time than is necessary. This is a disadvantage of prescheduling, but it can be minimized by an intelligent scheduler. The prepass scheduler should insert no more instructions than are necessary to avoid delays. Temporary values should be produced as late as possible and used as early as possible. It is shown later in this paper that if the scheduling is done intelligently, the benefits of the increased code movement flexibility outweigh the cost of the extra register spilling.

There is another disadvantage to prescheduling if postscheduling is not also done. During register allocation, the optimized sequence of instructions is perturbed by the spill code added, and there is no code motion opportunity to reduce the effects of this. If code scheduling is performed before and after register allocation, then the postpass scheduler can make the final adjustments to account for the extra code and dependencies added during register allocation. Because most of the code motion is already completed, the postpass scheduler is less hindered by the extra dependencies.

Figure 1 shows a code sequence (A) as it progresses through register allocation (B) and then postscheduling (C). For each instruction, the first operand is the destination, and the next one or two operands are the sources. *ld* is a load instruction and *st* is a store. The base-register-plus-



A

```
1 ld    t1,_chr      1
2 add   t2,t1,1      3
3 st    _chr,t2      4
4 ld    t3,_file1    5
5 ld    t4,0(t3)     7
6 add   t5,t4,-1     9
7 st    0(t3),t5    10
```

B

```
1 ld    r0,_chr      1
2 add   r1,r0,1      3
3 st    _chr,r1      4
4 ld    r0,_file1    5
5 ld    r1,0(r0)     7
6 add   r2,r1,-1     9
7 st    0(r0),r2    10
```

D

```
1 ld    t1,_chr      1
4 ld    t3,_file1    2
2 add   t2,t1,1      3
3 st    _chr,t2      4
5 ld    t4,0(t3)     5
6 add   t5,t4,-1     7
7 st    0(t3),t5     8
```

C

```
1 ld    r0,_chr      1
2 add   r1,r0,1      3
4 ld    r0,_file1    4
3 st    _chr,r1      5
5 ld    r1,0(r0)     6
6 add   r2,r1,-1     8
7 st    0(r0),r2     9
```

E

```
1 ld    r0,chr       1
4 ld    r1,file1     2
2 add   r2,r0,1      3
3 st    _chr,r2      4
5 ld    r0,0(r1)     5
6 add   r2,r0,-1     7
7 st    0(r1),r2     8
```

Figure 1: Examples of postpass and prepass code scheduling.

displacement addressing mode is similar to that of the MIPS R2000. For example, the memory address for the instruction *ld r1,x(r0)* is generated by adding *x* to the contents of *r0*. The number to the right of each instruction is the cycle in which the instruction is issued assuming that loads have a latency of 2 cycles and all the other instructions shown have a latency of 1 cycle.

In Figure 1 (C), instruction 4 cannot be moved ahead of instruction 2 because of the reuse of register 0 by the register allocator. This results in a stall when the operand for instruction 1 is not available in time because of the memory access delay. The corresponding sequence with prescheduling (D) and then register allocation (E) is also shown. The postscheduled version takes 1 cycle longer to execute than the prescheduled version. Both use the same number of registers, but the average register lifetime for the prescheduled sequence is slightly longer. Both of these examples are extracted from the most frequently executed block of code generated by our compiler for the Unix utility *cmp*.

2.2 IMPACT-I C Compiler

The IMPACT-I C Compiler [8] is a retargetable, optimizing compiler designed to generate very efficient code for pipelined and multiple-instruction-issue processors. High quality code generators have been built for the MIPS R2000 [9], the Sun SPARC [10], the AMD 29K [11], and the Intel i860 [12] processors. Code generators are under construction for the IBM RS/6000 [13] and the Intel i486 [14]. IMPACT-I is used to study the effectiveness of new code optimization techniques and to study alternative approaches in the design of processors that exploit fine-grain parallelism.

The IMPACT-I C compiler currently performs a wide variety of machine-independent and machine-dependent code optimizations. The machine-independent optimizations include the classic local and global code optimizations [15], inline function expansion [16], instruction placement optimization [17], loop unrolling, intelligent generation of switch statements [18], and jump optimization. Machine-dependent optimizations include profile-based branch prediction [19], constant preloading, graph-coloring-based register allocation [6], and code scheduling. IMPACT-I also contains a profiler to target the most frequently executed program sections for optimization.

2.3 Register Allocation

The IMPACT-I global register allocator is based on the graph-coloring algorithm described in [6]. The algorithm constructs an interference graph in which each node represents a variable (either a temporary or a user-defined variable). An arc is added between two nodes if they are ever simultaneously live. Two adjacent nodes cannot be allocated to the same register. The algorithm tries to color the graph using **r** colors, where **r** is the number of available registers. If the graph cannot be colored in **r** colors, then a register must be spilled, and the coloring attempted again.

A natural result of this algorithm is that two variables which do not have overlapping live ranges (i.e. are not adjacent in the interference graph) are often allocated the same register. This register reuse introduces dependencies that prevent the code scheduler from overlapping otherwise independent instructions which read or write the two variables. Because the algorithm does not take into account the cost of instructions that cannot be overlapped, it may allocate registers in a way that handicaps the code scheduler.

2.4 Superblock Scheduling

This section describes the IMPACT-I code scheduler, which is based on a new variation of trace scheduling [4, 5] that we call superblock scheduling. The idea is to select frequently executed paths through the code and optimize them, perhaps at the expense of the less frequently executed paths. Instead of inserting bookkeeping instructions where two traces join, we duplicate part of the trace and optimize the original copy. This method is especially useful for the control-intensive benchmarks studicd in this paper because it provides an easier way to find parallelism beyond the basic block boundaries. We describe the scheduler here because the results presented in this paper are based on this kind of global scheduling. Superblock scheduling is performed in the six-step process shown below:

1. Trace selection
2. Superblock formation and enlargement
3. Classic code optimization
4. Dependence graph construction
5. Dependence graph optimization
6. List scheduling

When a program is compiled with the prescheduling option turned off, steps 1 through 3 are completed, followed by register allocation. Then the dependence graph is constructed and optimized, and the code is scheduled. When the prescheduling option is turned on, steps 4 through 6 are also performed just before register allocation. The next subsection describes the program representation used and the modifications to the code prior to scheduling. Later subsections describe each step of the process.

2.4.1 Program Representation and Preparation

In our C compiler, a function is represented as a weighted control flow graph. A control flow graph is a directed graph where each node is a basic block, and each arc is a possible control transfer path between two basic blocks. A weighted control flow graph is derived from an ordinary control flow graph by annotating each node (arc) with the average execution count of the corresponding basic block (control transfer path). The average execution count is the arithmetic mean of the execution counts for all the profiled runs of the program [2].

Several steps are taken to prepare the program for optimization and code scheduling. First, the control flow graphs are generated for each function. Then the compiler performs a few optimizations such as constant folding, dead code removal, and jump optimization to form larger basic blocks. Probes are inserted into all the basic blocks to collect the execution counts, and the program is profiled several times with different inputs. The results from all the runs are averaged and used to assign weights to the nodes and arcs of the graphs. Frequently executed function calls are then expanded inline if possible [16].

2.4.2 Step 1: Trace Selection

The goal of trace selection is to divide the function into a set of traces such that for each block **X**, if there is a block **Y** immediately following (preceding) **X** in a trace, **Y** is the block most likely to be executed after (before) block **X** when the program is run with real data [3]. The block most likely to be executed after (before) block **X** is determined by examining the execution counts of all the arcs leaving (entering) block **X**. The trace becomes the unit in which instructions are

[2] Each run uses different inputs, and the profiling is performed by the built-in profiler in our compiler.

[3] The real data are represented by the inputs used during profiling.

rearranged. As a result, code movement across basic block boundaries is automatically done in such a way as to optimize the more frequently executed paths. When the schedule along one path can be improved at the expense of the schedule along another path, the decision is made in favor of the more frequently traveled path (i.e., the one in the trace).

The formation of a trace begins with the selection of a seed block. Then the trace is grown forward as far as possible by adding the most likely successor of the last block to the end of the trace. Similarly, the trace is also grown backward as far as possible. This heuristic was first proposed by Ellis [5] and improved by Chang and Hwu [20]. A node is not added to a trace unless its execution count is higher than a minimum count and the probability of entering it from its predecessor or leaving it for its successor in the trace is greater than a minimum probability [4].

Figure 2 shows a weighted control flow graph and four traces (surrounded by the dotted lines) produced by trace selection. Two features of the trace selection algorithm are visible in Figure 2 . First, each block appears in only one trace. Traces distinguish the frequently executed paths along which the schedule should be optimized. If a block were to appear in more than one trace, the optimizations along the various paths may have conflicting results. Second, if a block **X** is in one trace, and the block most likely to follow **X** is already in another trace, then block **X** is the last one in its trace, even if it has other possible successors. We do not want to optimize a path between block **X** and one of these other successors, because it may have an adverse effect on the schedule for the path between **X** and its most likely successor. In the example, block **F** is not added to the trace containing block **E** because the path from block **E** to block **D** is executed much more often than the path from block **E** to block **F**.

[4] In the experiments done for this paper, the minimum count is 50 and minimum probability is 70%.

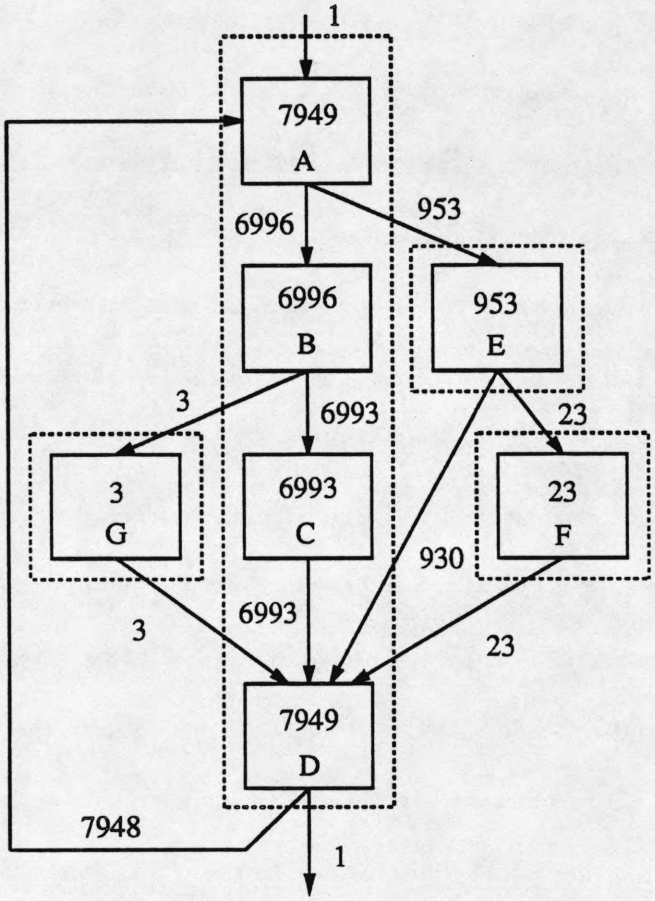


Figure 2: An example of trace selection.

Once the traces have been selected, the basic blocks of each trace are laid out sequentially in memory [17]. Then superblocks are formed and enlarged as described in the next subsection.

2.4.3 Step 2: Superblock Formation and Enlargement

We define a *side exit* as a branch from any block **X** in the trace except the last one to a block **Y** (**Y** can be in or out of the trace) where **Y** does not immediately follow **X** in the trace. A *side entrance* is defined as a branch from a block **X** (**X** can be in or out of the trace) to any block **Y** in the trace except the first one, where **X** does not immediately precede **Y** in the trace. We define a superblock as a trace that has no side entrances and zero or more side exits. The goal of superblock formation is to convert a trace that has side entrances and exits into a superblock. The motivation and method for doing this is explained in the following paragraph.

In the traces formed in step 1, there may be many side exits and entrances. In the trace in Figure 2, there are side exits from blocks **A** and **B** and several side entrances at block **D**. The side entrances especially increase the difficulty of code scheduling because complex bookkeeping must be done when code is moved above and below these entrances [4]. These complex repairs could be avoided if side entrances could be removed from the trace. One way to do this would be not to add a block to a trace if it produces a side entrance. However, for control-intensive programs, this would limit the size of the traces and the effectiveness of trace scheduling. Instead we chose to remove the side entrances using a technique called *tail duplication* (essentially a more aggressive but simple form of bookkeeping). A copy is made of the tail portion of the trace from the side entrance to the end and is appended to the end of the function. Each block copied forms a new superblock. All side entrances into the trace are then moved to the corresponding duplicate basic blocks. At this point, the trace, with only a single entrance remaining, becomes a superblock that can be optimized with special handling only for the side exits.

Trace selection, superblock formation, and tail duplication can be applied to the newly duplicated basic blocks to form larger superblocks. The existing traces can also be reexamined to see if one of the newly created blocks should be added to a trace. In some cases, a small superblock created by tail duplication can be copied and merged with one or more of its predecessors. For this to be profitable, the small superblock must be the most likely successor of its predecessor and must satisfy the minimum execution count and probability thresholds. Figure 3 shows the effect of the superblock formation step on the weighted control flow graph from Figure 2. In this example, block **D'** can be copied and merged with its predecessor **E**.

Tail duplication adds to the code size, which is one reason why a block is not added to a trace if its execution count is below the threshold value. If the block is not executed often enough, optimizing the path to that block is not worth the cost of the increased code size and compilation effort. Also note that the profile information is scaled during tail duplication. For example, in Figure 3, block **D** is now executed 6993 times instead of 7949. Therefore, the weights for each of its outgoing arcs are multiplied by the fraction 6993/7949 and rounded to the nearest integer value. This reduces the accuracy of the profile information. For optimizing the code, the approximate information is good enough. For accurate analysis of the final schedule however, the transformed program must be reprofiled.

An added benefit of tail duplication is that the classic code optimizations can be more easily applied to superblocks than to traces [21]. The IMPACT-I compiler uses the superblock as a common foundation for both classic optimizations and code scheduling.

At this point, several code transformations are performed that enlarge the size of the su-

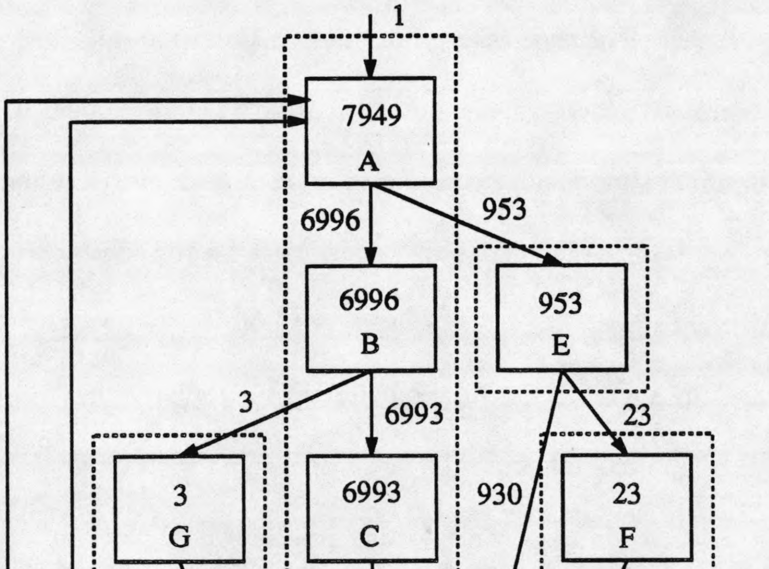


Figure 3: An example of superblock formation.

perblocks and reduce the depth of critical paths. Superblocks that are contained in loops can profit from some loop-based transformations. For loops that are usually executed only a few times, a few iterations can be *peeled* off and added to the superblock before the loop. For loops that iterate a larger number of times, the loop can be unrolled. In either case, only the part of the loop that is in the superblock needs to be replicated. Infrequently executed paths through the loop are not duplicated. The transformations performed to reduce the depth of critical paths include induction variable expansion, operation combining, and operation folding [8].

2.4.4 Step 3: Classic Code Optimization

Next, many classic code optimizations are performed that take advantage of the profile information encoded in the superblock structure and clean up the code after all of the transformations performed in step 2. In both local and global versions, these include: constant propagation, copy propagation, common subexpression elimination, redundant load and store elimination, dead code removal, and constant folding. Local strength reduction, local constant combining and global loop invariant code removal, loop induction strength reduction, and loop induction elimination are also performed [21]. At this point, the program is reprofiled if the final schedule is to be analyzed.

2.4.5 Step 4: Dependence Graph Construction

In this step, a conservative dependence graph is built for each superblock. Data dependence arcs are added as if the superblock were a basic block. However, unlike basic blocks, superblocks may contain branches. For each conditional branch instruction **I**, we define *live_out*(**I**) as the set of variables that may be used before they are defined when **I** is taken. A data dependence arc is added from an instruction to a conditional branch I below it if the instruction writes a variable that is in *live_out*(I) or if the instruction may cause an exception. A control dependence arc is added from a conditional branch I to an instruction below it in the superblock if the destination variable of the instruction is in *live_out*(I) or if the instruction may cause an exception.

Each flow dependence arc has a length associated with it that is equal to the latency of the instruction that is the source of the dependency. The anti- and output dependence arcs have length 1. It is assumed that these dependencies are handled by the hardware register renaming. The side exits in the superblock are predicted to not be taken, so there is no delay for a control dependence and the length of the arc is 0 [5]. No memory disambiguation is done before adding the dependence arcs. Some of these arcs can later be removed as discussed in the next subsection.

2.4.6 Step 5: Dependence Graph Optimization

In this step, the dependence graph is optimized by removing some of the dependence arcs. Memory disambiguation is performed and dependence arcs are removed for any memory accesses that can be resolved. During the list scheduling step (described in the next subsection), the instructions are reordered to improve the execution time within the constraints of the dependencies. Instructions are moved upward and downward across branches. There are two major restrictions on moving an instruction upward across a branch I:

1. The instruction must not write a variable that is in *live_out*(I).

2. The instruction must not cause an exception that terminates the program execution.

[5] For the superscalar processors, multiple branches can be issued in a cycle and the architecture uses a squashing branch scheme [19].

As an example of the second restriction, it is not safe to move a division or floating-point instruction above a branch because of the possibilities of a division by zero or a floating-point exception, respectively. It is also not safe to move a memory load instruction above a branch because of the possibility of a memory access violation. Page faults are not a problem, because they do not cause the execution to terminate. However, moving loads from below to above branches may increase the number of page faults.

We have implemented two different code scheduling models for the purpose of experimentation. The first model enforces both of the restrictions and is called *restricted percolation.* This model is necessary for the current generation of commercial architectures where a subset of the instructions can cause traps. When this model is used, no additional dependence arcs are removed after memory disambiguation. The second model allows the second restriction to be avoided. This model is called *general percolation.* In this model, the architecture provides non-trapping versions of the instructions that can cause exceptions [22]. Whenever an instruction is moved upward across a branch, the non-trapping version is used.

If an exception occurs during a non-trapping instruction, the exception is simply ignored (except for page faults, which are handled normally). An invalid value is placed in the destination register for loads and arithmetic operations. Instructions that use a (possibly invalid) value generated by a non-trapping instruction can also be percolated. This can make error conditions that normally cause traps harder to detect. Also, some programs require all traps to be detected and so do not allow general percolation. Moving a load from below to above a branch increases the total number of memory accesses made by the program, because the load is now always executed regardless of which path is taken. Because the load is moved up from the most frequently executed path, the number of extra references should be moderate. When the general percolation model is used, any control dependence arcs which result only from the second restriction can be removed.

An instruction can always be moved downward across a branch. However, if it may cause a trap, that exception is only detected when the branch is not taken. The ability to move such instructions from above to below a branch does not improve the schedule very much and we prefer not to lose the exception. Therefore, we do not move an instruction downward across branches if it may cause an exception.

If an instruction is moved from above to below a conditional branch I and it writes a variable that is in *live_out*(I), the instruction must also be inserted between I and its target. In our compiler, for ease of implementation, code motion of this type is done during the code optimization phases prior to list scheduling. Therefore, the scheduler does not move an instruction below a branch if it writes a variable that is in *live_out*(I).

2.4.7 Step 6: List Scheduling

In this step, the dependence graph is scheduled. Because the code is scheduled before register allocation as well as after, the scheduling algorithm is careful to keep the register lifetimes to a minimum while trying to optimize the code for the pipeline. Temporary values are produced as late as possible and used as soon as possible, shortening the register lifetimes and reducing the amount of spilling. The algorithm also tries to control the number of simultaneously live registers to reduce spilling.

The general idea of the list scheduling algorithm is to pick, from a set of nodes (instructions) that are *ready* to be scheduled, the best combination of nodes to issue in a cycle. A node is *ready* if all of its parents have been scheduled and the result produced by each parent is available (i.e. since the time that the parent node was scheduled, enough cycles have passed to cover its latency). When a node is ready, it is placed in a set of nodes called the *active set.* There are a set of instruction templates for the processor that specify the possible combinations of instructions that can be issued in a cycle. For each cycle, the scheduler finds the best set of nodes from the active set to fill each template. Then it issues the best instruction template and marks the nodes in the template as scheduled. The best node is determined by examining the priority of each node in the active set. The priorities of all the nodes in a template are added together to determine the best template. If there are no nodes in the active set, the scheduler does not have to issue no-ops. In this case, the flow dependencies are enforced by the hardware interlocks. The scheduler simply advances the cycle count and checks to see if nodes become ready to be scheduled. The list scheduling algorithm that we use is shown in Figure 4.

```
algorithm list_schedule(dependence graph D) begin
    for (each node N in graph) begin
        compute priority(N);
        add N to unscheduled_set;
    end_for
    while (unscheduled_set is not empty) begin
        active_set = the set of nodes in unscheduled_set
                     that are ready to be scheduled;
        sort active_set according to node priority;
        for (each instruction template) begin
            find the best set of nodes from active_set to fill it;
            priority of each instruction template = the sum of the
                 priorities of nodes in the instruction template;
        end_for
        issue the best instruction template;
        remove nodes in the issued instruction template from
            unscheduled_set;
    end_while
end_algorithm
```

Figure 4: The list scheduling algorithm.

The priority for each node is computed statically before scheduling begins. It is the weighted sum of the values returned by several heuristic functions. Each heuristic function **F***i*(N) (where N is a node) returns a priority value between 0 and 1. For a given node, one heuristic function may return a high value, and another a low value. Each function is assigned a weight **W***i* to resolve these kinds of conflicts. The function *priority*(N) returns (sum i=1 to n, **F***i*(N)*$*$***W***i*). Some of the heuristic functions used are described below beginning with the highly weighted ones:

slackness(N) This heuristic function assumes that resources are unlimited and that the best schedule length is equal to the depth of the dependence graph. It finds the latest time that node N can be issued without increasing the length of the best schedule and then assigns a priority between 0 and 1 based on that. Nodes that can be postponed without increasing the length of the schedule receive a lower priority. Issuing nodes as late as possible reduces the register lifetimes.

exec_count(N) Nodes above a branch (including the branch) are given higher priority than nodes below the branch. This is because the nodes above the branch are executed more times than the nodes below the branch. We do not want to move an operation with a lower execution count upward across a branch, if it will delay the issuing of the branch.

register_use(N) This function gives a high priority to nodes that are the last to use a variable, because they free registers. It gives a low priority to nodes that write a variable because they require a new register. This reduces the number of simultaneously live registers.

uncover(N) High priority is given to nodes that have many children. Once a node like this is issued, many nodes are added to the active set. Branches, loads, and stores are favored by this heuristic.

orig_order(N) If two nodes can be scheduled in any order, the node which appears first in the original code sequence receives a higher priority.

The weight given to each of these heuristic functions can be tailored to the target architecture. For example, if the architecture has a small number of registers, the **register_use(N)** and **slackness(N)** heuristics might be given more weight. The **uncover(N)** heuristic might be emphasized for a microarchitecture with lots of parallelism and a large register file. In this paper, we use the same set of weights for all of the experiments.

3 Experiments

3.1 Methodology

This section presents an empirical evaluation of the importance of prescheduling for the superscalar and superpipelined versions of existing and future architectures. Each experiment consists of compiling and optimizing a set of control-intensive, production C programs as described in Section 2.4. In each experiment, the benchmarks are compiled for several different implementations of a base architecture. For each case, we compile once with both prescheduling and postscheduling turned on and once with only postscheduling turned on. For each compilation, the program execution time, and the number of dynamic memory references are calculated (assuming a 100% cache hit rate) using the schedule for each superblock and the profile information. The number of dynamic references gives an indication of the amount of register spilling. It is also affected by the number of loads moved from below to above branches.

The time for each execution of a superblock depends upon whether or not a side exit is taken. The profile information indicates how many times each path is taken, and this quantity is multiplied by the execution time of that path to get the total time spent executing that path during the measured run of the program. The totals for all the paths are then added to get the total execution time for the superblock. The number of dynamic memory references is calculated in a similar manner.

The execution time result for each compilation is reported as a speedup relative to the compilation for the base microarchitecture. The speedup for benchmark **B** running on processor implementation **I** is equal to the execution time of the benchmark on the base implementation divided by the execution time of the benchmark on microarchitecture **I**. Numbers greater than one indicate that benchmark **B** ran faster on microarchitecture **I** than on the base implementation. For the register spilling results, we define a metric called the *memory reference ratio* (MRR). The memory reference ratio is the number of dynamic memory references issued for benchmark **B** running on implementation **I** divided by the number of dynamic memory accesses for the benchmark on the base microarchitecture. Numbers greater than one indicate that more memory accesses were made when the benchmark ran on implementation **I** than when it ran on the base microarchitecture. The memory reference ratio is an indication of the demands placed upon the memory system. In a real system where the cache hit rate is not 100%, extra memory accesses may cause the speedup reported here to be reduced. The speedup and memory reference ratio shown in Section 3.5 for each combination of microarchitecture and compilation options are both the arithmetic means of the speedups and memory reference ratios for all the benchmarks on that combination.

3.2 Processor Architecture

In addition to the benchmark, the scheduler takes as input a machine description file that characterizes the instruction set, the microarchitecture (including the number of instructions that can be issued in a cycle and the instruction latencies), and the code scheduling model and options (this is where prescheduling is turned on and off). The base microarchitecture is a pipelined, single-instruction-issue processor that supports the restricted percolation model. Its instruction set is a superset of the MIPS R2000 instruction set. Table 1 shows the instruction latencies. Instructions are issued in order and it is assumed that there is hardware register renaming to eliminate write-after-read and write-after-write hazards. Read-after-write hazards are handled by stalling the instruction-unit pipeline. The microarchitecture uses a squashing branch scheme [19] and profile-based branch prediction. One branch slot is allocated by the compiler for each predicted-taken branch. The processor has 32 integer registers and 32 floating-point registers [6]. Of the 32 integer registers, 8 are reserved as special registers (for the stack

Table 1: Instruction latencies.

FUNCTION	LATENCY
integer ALU	1
barrel shifter	1
integer multiply	3
integer divide	25
load	2
store	-
FP ALU	3
FP conversion	3
FP multiply	4
FP divide	25

[6] The code for these benchmarks contains very few floating point instructions. In the experiments, whenever we change the integer register file size, we also change the floating-pont register file size by the same amount.

pointer, frame pointer, parameter passing registers [7], etc.) and are not available for use by the register allocator. All the speedups and memory reference ratios reported in Section 3.5 are relative to this base microarchitecture.

The superscalar version of this processor fetches multiple instructions into an instruction buffer and decodes them in parallel. An instruction is blocked in the instruction unit if there is a flow dependency between it and a previous instruction. All the subsequent instructions are also blocked. All the instructions in the buffer are issued before more instructions are fetched. The maximum number of instructions that can be decoded and dispatched simultaneously is called the *issue rate*. The superscalar processor also contains multiple functional units. Each functional unit can be a single unit such as an ALU, or a group of different units such as a cache interface, an ALU, and branch logic. The capabilities of the functional units determine how many of a particular class of instructions can be executed in parallel. For example, if only one of the functional units contains a store unit, then only one store can be issued in a cycle. For the processors in this paper, all the functional units contain a load unit, an integer ALU, a floating-point ALU, and branch logic. Only one of the functional units contains a store unit because the ability to do multiple loads and branches [19] is more important than the ability to do multiple stores [8]. When the issue rate is greater than one, the number of branch slots increases [19].

The superpipelined version of this processor has deeper pipelining for each functional unit. If the number of pipestages is increased by a factor **P**, the clock cycle is reduced by that same factor. The latency in clock cycles is longer, but in real time it is the same as the base

From this point on, we will simply refer to *the register file size*, meaning the integer register file size.

[7] The parameter passing registers are used as temporary registers for leaf-level functions.

microarchitecture. The throughput increases by up to the factor **P**. We refer to the factor **P** as the *degree of superpipelining.* The instruction fetch and decode unit is also more heavily pipelined to keep the microarchitecture balanced. Because of this, the number of branch slots allocated for the predicted-taken branches increases [19].

For the superscalar processor, the additional datapaths, functional units, and instruction decoding logic may increase the cycle time. For the superpipelined processor, the cycle time is actually reduced by less than the factor **P** because of the latch delays. This paper reports speedups based on ideal cycle times and leaves the reader with the task of scaling the speedups to account for the above effects.

3.3 Benchmarks

The benchmarks used are shown in Table 2 along with the inputs with which each one is profiled prior to optimization. The SIZE column specifies the size of each program in number of lines of code. After superblock formation and the classic code optimizations, each benchmark is profiled again with one input that is not in the set shown in Table 2. Recall that after superblock formation, the profile information is only approximate. The benchmarks must be reprofiled in order to accurately measure the execution time and the number of dynamic memory references. In most cases, a compiled production program will not be run with exactly the same inputs that it is profiled with. By using an input which is not in the set that was used for optimization, we get a more realistic estimate of how well the benchmark was optimized for general inputs.

Table 2: The benchmarks.

BENCHMARK	SIZE	BENCHMARK DESCRIPTION	INPUT DESCRIPTION
cccp	4787	GNU C preprocessor	20 C source files (100 - 5000 lines)
cmp	141	compare files	20 similar/dissimilar files
compress	1514	compress files	20 C source files (100 - 5000 lines)
eqn	2569	typeset math formulas	20 ditroff files (100 - 4000 lines)
eqntott	3461	boolean minimization	5 files of boolean equations
espresso	6722	boolean minimization	20 original espresso benchmarks
grep	464	string search	20 C source files with search strings
lex	3316	lexical analyzer generator	5 lexers for C, lisp, pascal, awk, pic
li	7747	lisp interpreter	5 gabriel benchmarks
mpla	38970	pla generator	20 boolean functions
qsort	136	quick sort	Built-in input
tbl	2817	format tables for troff	20 ditroff files (100 - 4000 lines)
wc	120	word count	20 C source files (100 - 5000 lines)
yacc	2303	parser generator	10 grammars for C, pascal, pic, eqn

3.4 Compiler Calibration

It is important to measure the effectiveness of prescheduling using a compiler that produces highly optimized code prior to code scheduling. Code that is not well optimized can contain redundant instructions that change the dependency pattern and allow the prescheduler to produce deceptively parallel code. On the other hand, some dependencies may not be removed by a poor optimizer, restricting the ability of the prescheduler to move code. To calibrate the quality of the code generated by IMPACT-I, the execution time of its output code has been compared to that of the commercial MIPS C compiler[8], which is well known for its excellent code optimization capabilities. For the benchmarks described earlier, the performance of IMPACT-I is slightly better than that of the MIPS C compiler [21]. Thus, the evaluation of prescheduling reported in this paper is based on well optimized sequential code.

[8] MIPS Release 2.1 using the (-O4) option.

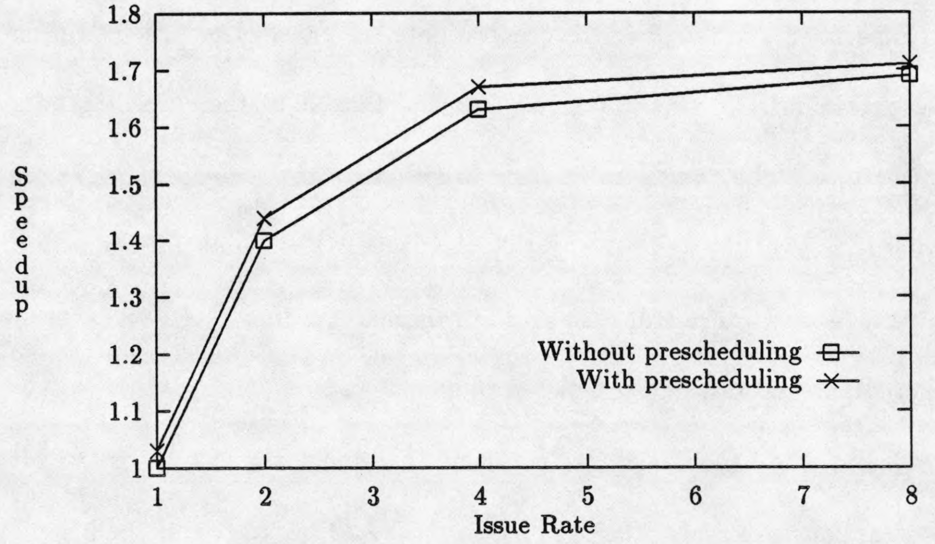
3.5 Results

3.5.1 The Importance of Prescheduling for Existing Architectures

In this section, two experiments are performed to investigate the effect of prescheduling on the performance of the superscalar and superpipelined implementations of the current generation of commercial architectures. The goal is to find out whether or not these processors require prescheduling in order to exploit the fine-grain parallelism in the C benchmarks. Some instructions in these architectures can cause traps, so all the compilations for these two experiments adhere to the restricted percolation code scheduling model.

In the first experiment, the benchmarks are compiled for superscalar processors with issue rates from 1 to 8 instructions per cycle. These processors have 32 registers and the instruction latencies given in Section 3.2. For each case, the benchmarks are compiled once with prescheduling and once without it. The speedups and memory reference ratios are calculated with respect to the single-instruction-issue base architecture described in Section 3.2. Prescheduling is turned off for this 32-register base architecture.

The results are shown in Figure 5. The speedup and memory reference ratio (MRR) numbers show the effect of the increase in issue rate over the base processor. The two curves show the performance with and without prescheduling. Notice that the speedup for the single-instruction-issue processor without prescheduling is 1. This is the base case. Prescheduling extracts a little more performance for every issue rate, but the performance increase is limited by the restricted percolation code scheduling model. We have observed that loads are often in the critical path. This was illustrated in the code segment that was shown in Figure 1. However, with the restricted percolation model, loads cannot be moved from below to above branches, limiting



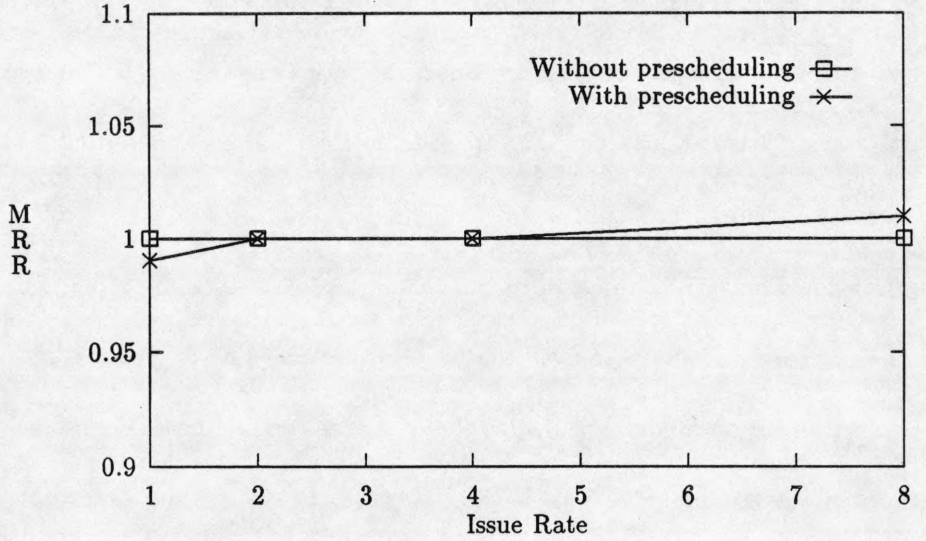


Figure 5: The performance of prescheduling for the superscalar versions of existing architectures. The base architecture is the single-instruction-issue processor with no prescheduling. All the processors have 32 registers.

the ability of the prescheduler to optimize the critical path.

The changes in the amount of memory references for this experiment are purely due to spilling because loads cannot be moved above branches. For issue rate 1, there are less memory references with prescheduling than without it. Before code scheduling, the instruction sequence is not optimized. Some temporaries are produced too early, resulting in register lifetimes that are longer than they have to be. Prescheduling has more freedom to rearrange the code to shorten the register lifetimes and reduce spilling. As the issue rate increases, the prescheduler tries to take advantage of the parallelism. More temporaries are simultaneously live, demanding more registers and increasing the amount of spilling.

In the second experiment, the benchmarks are compiled for superpipelined processors with the degree of superpipelining varied from 1 to 3. We refer to these as 1X-, 2X-, and 3X-superpipelined processors respectively. These processors have 32 registers. For each case, the benchmarks are compiled once with prescheduling and once without it. The speedups and memory reference ratios are calculated with respect to the same single-instruction-issue base architecture as for the first experiment.

The results are shown in Figure 6. Prescheduling again maintains a small advantage in speedup for all the processors and reduces register spilling for the single-instruction-issue processor. Prescheduling does not increase the register spilling until the parallelism in the microarchitecture is increased by a factor of six (for the two-instruction-issue 3X-superpipelined processor). Even then the increase is only 1%.

The results of this section show that prescheduling is not important for compiling control-intensive programs to today's architectures. The frequent branches in the C programs we used combined with the restrictions on code movement imposed by trapping instructions hinder

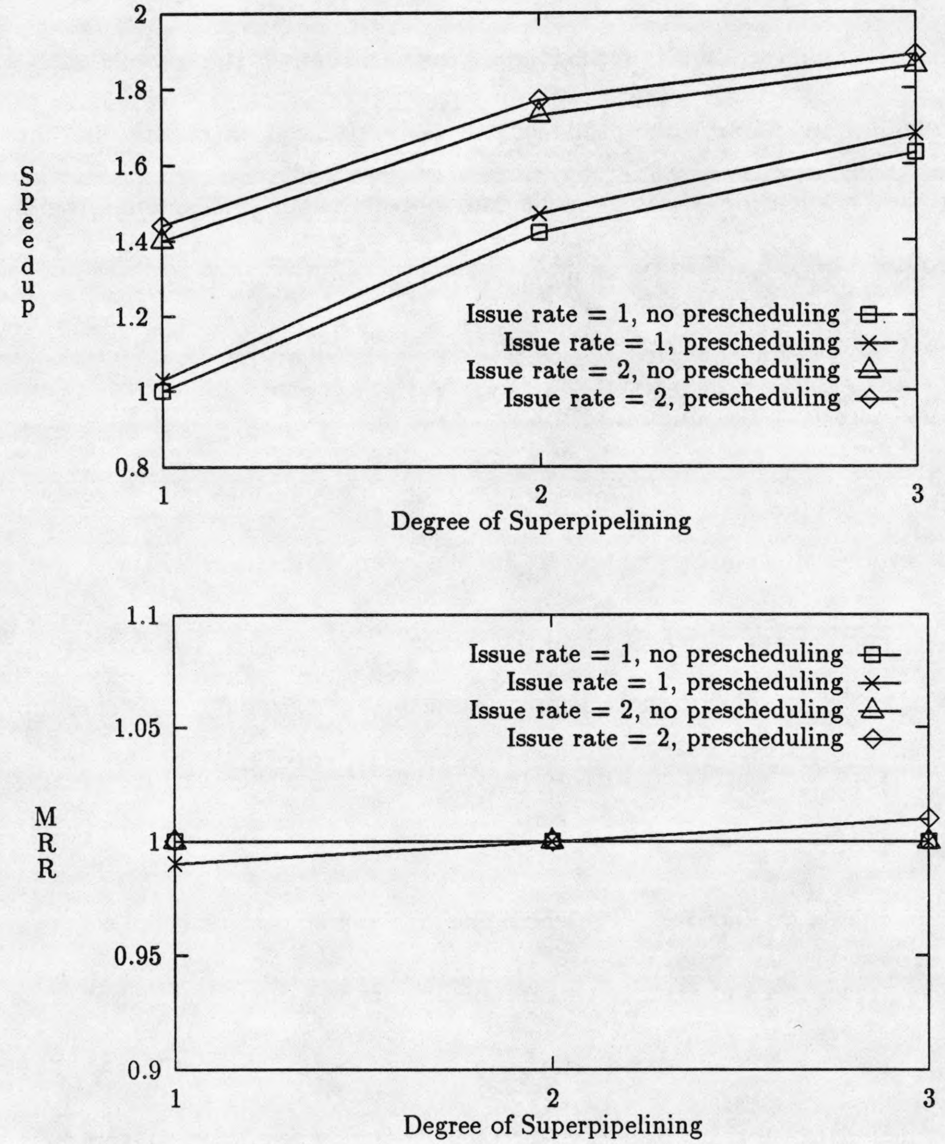


Figure 6: The performance of prescheduling for the superpipelined versions of existing architectures. The base architecture is the single-instruction-issue 1X-superpipelined processor with no prescheduling. All the processors have 32 registers.

the code scheduler so much that the extra dependencies added by register allocation don't have much additional effect. In order to obtain more speedup from these benchmarks using processors that exploit fine-grain parallelism, some way must be found to eliminate or work around the restrictions imposed by trapping instructions. The next subsection presents the results obtained by doing just that. It is shown that once this restriction is removed, prescheduling becomes critical to exploiting the newly obtained code movement opportunities.

3.5.2 The Importance of Prescheduling for Future Architectures

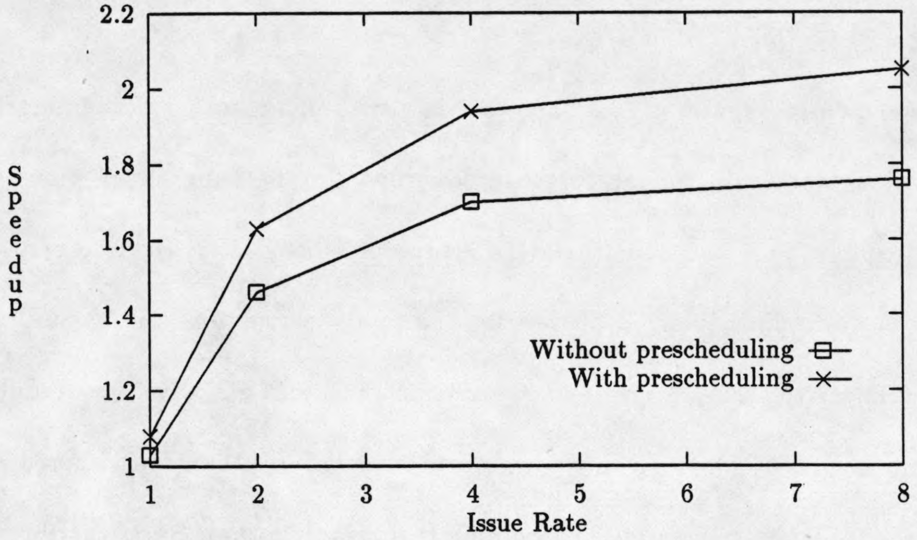
In this section, two experiments are performed to study the effect of prescheduling on the performance of the superscalar and superpipelined implementations of an architecture that supports the general percolation code scheduling model. The goal is to demonstrate that these processors require prescheduling in order to exploit the extra parallelism in the C benchmarks made available by general percolation.

In the first experiment, the benchmarks are again compiled for superscalar processors with issue rates from 1 to 8 instructions per cycle. These processors have 32 registers and the instruction latencies given in Section 3.2. For each case, the benchmarks are compiled once with prescheduling and once without it. This time the compiler makes use of the general percolation model. The speedups and memory reference ratios are calculated with respect to the single-instruction-issue base architecture described in Section 3.2. Prescheduling is turned off and restricted percolation is used for this 32-register base architecture. This way, the speedup and change in memory references due to both prescheduling and the general code percolation model is shown. The change in memory references is due to both spilling and to loads that are moved from below to above branches.

The results are shown in Figure 7. Notice that the speedup for the single-instruction-issue processor without prescheduling is greater than 1. This is the speedup over the base architecture due to general percolation. The performance advantage of prescheduling is now much more pronounced. For issue rate 8, the advantage is 17%. The hardware that supports the general percolation model provides richer opportunities for parallelism, but prescheduling is required to take advantage of them. Prescheduling now increases the amount of spilling even for the single-instruction-issue processor as it exploits the opportunities provided by the general percolation model. The speedup with prescheduling increases faster than without it in spite of the spilling, which also increases faster with prescheduling. At the high issue rates, there are more unused instruction slots to hide the spill code, and the extra parallelism exploited overcomes any loss due to spill code that cannot be hidden.

The memory reference ratio is constant without prescheduling. The slight increase over the base microarchitecture is due only to loads that are moved from below to above branches. Without prescheduling, the register allocation algorithm provides the same number of registers for all issue rates even though that may not be enough to support the parallelism available in the hardware. With prescheduling, the increase in memory references is small when the issue rate is low. The scheduler moves instructions only enough to satisfy the pipeline constraints and exploit the available parallelism. This keeps the register lifetimes to a minimum, reducing the spilling. As the issue rate increases, the scheduler takes advantage of the opportunities to issue instructions in parallel and as a result is forced to increase the number of registers used.

In second experiment, the benchmarks are compiled for superpipelined processors with the degree of superpipelining varied from 1 to 3. These processors have 32 registers. For each case, the benchmarks are compiled once with prescheduling and once without it. Again,



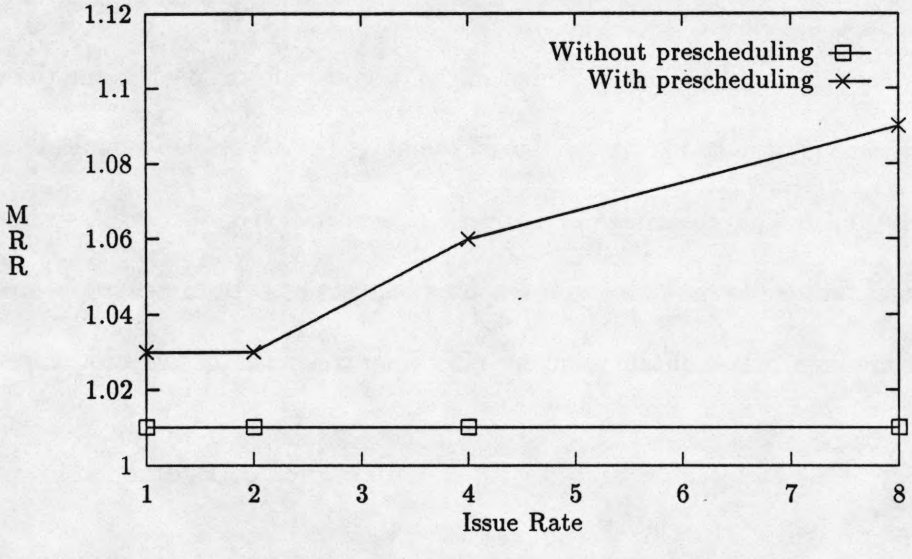


Figure 7: The performance of prescheduling for the superscalar versions of architectures that support general percolation. The base architecture is a single-instruction-issue processor with no prescheduling and restricted percolation. All the processors have 32 registers.

the compiler uses the general percolation model. The speedups and memory reference ratios are calculated with respect to the familiar single-instruction-issue base architecture with 32 registers. Prescheduling is turned off and the restricted percolation model is used for the base processor.

The results are shown in Figure 8. The increases in performance for prescheduling with the general percolation model are similar to those described for the superscalar processors. Prescheduling can exploit both the superscalar and superpipelined microarchitectures very well. Note that for the 3X-superpipelined processors, the single-instruction-issue version with prescheduling outperforms the two-instruction-issue version that does not have prescheduling.

This section demonstrated that for control-intensive benchmarks, the general percolation code scheduling model provides more code motion opportunities, but these opportunities have to be taken advantage of before register allocation. Once the restrictions imposed by trapping instructions are removed, the dependencies added during register allocation become the major impediment to reorganizing the code. Without prescheduling, the added dependencies prevent the code scheduler from taking advantage of the general percolation model to the point that there is little or no advantage to providing non-trapping instructions. Both general percolation and prescheduling are required to obtain good speedup from control-intensive programs.

3.5.3 The Effect of Register File Size on the Performance of Prescheduling

In this section, an experiment is performed to study how the advantage of prescheduling varies with the register file size. We also want to see the extent to which larger register file sizes decrease the extra memory referencing that results from prescheduling. For the experiment, we pick a middle-of-the-road superscalar processor with issue rate 4, and vary its register file size

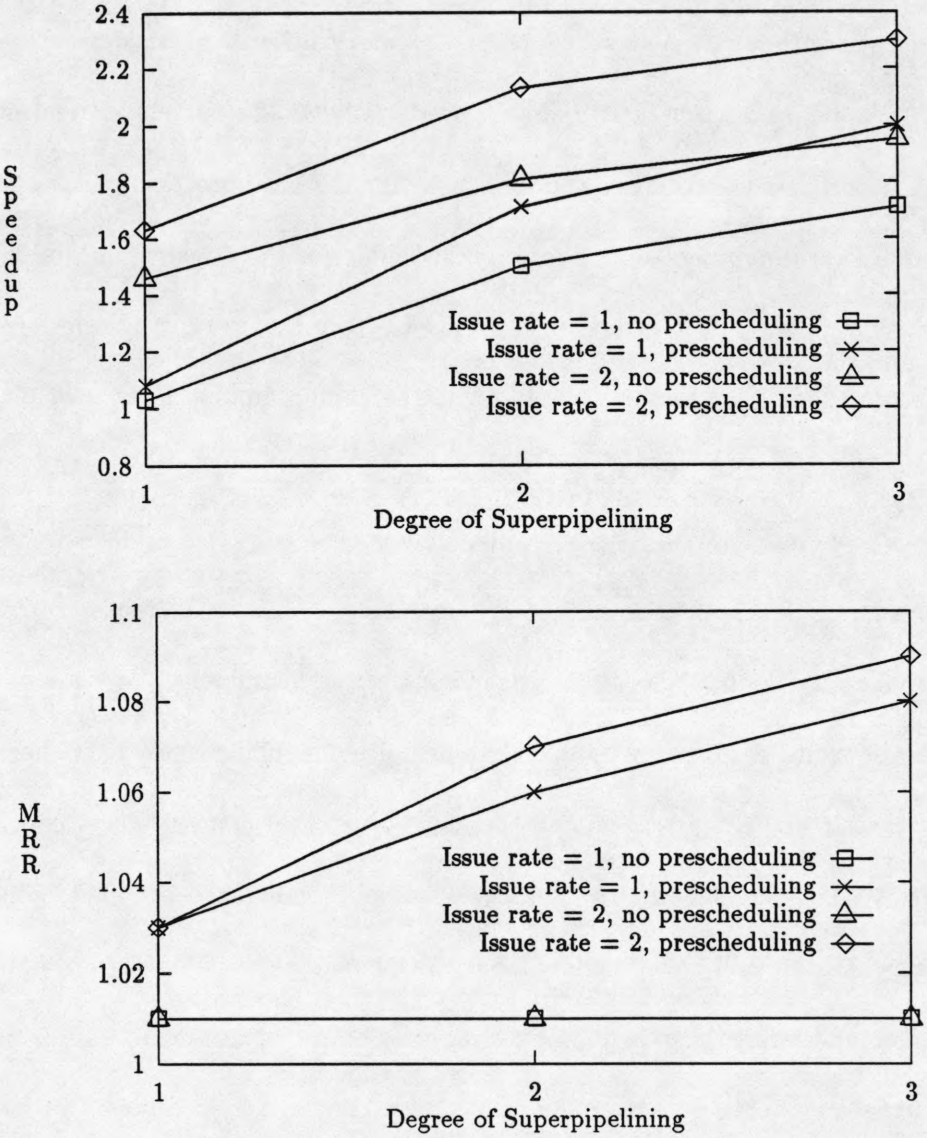
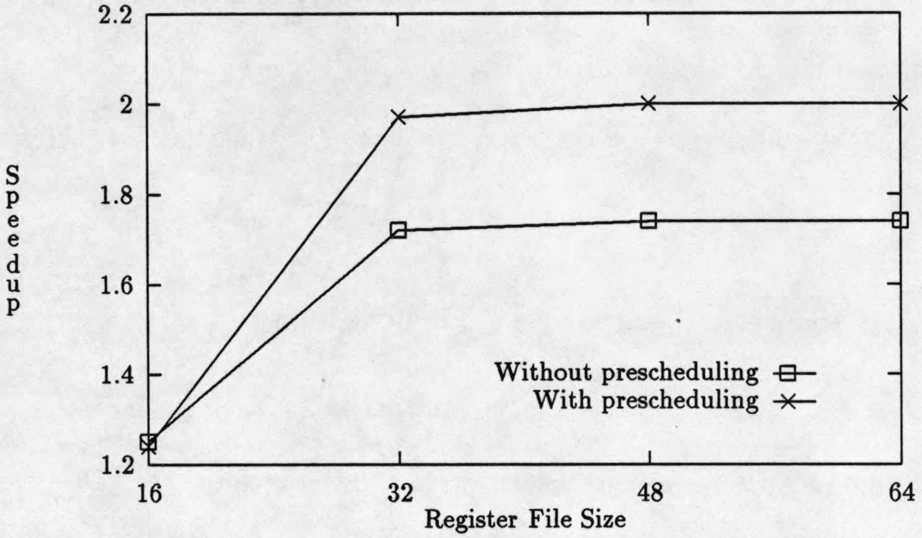


Figure 8: The performance of prescheduling for the superpipelined versions of architectures that support general percolation. The base architecture is a single-instruction-issue processor with no prescheduling and restricted percolation. All the processors have 32 registers.

from 16 to 64 registers (recall that 8 of these registers are reserved as special registers). We use the general percolation code scheduling model since it represents the class of architectures for which prescheduling is important. For each case, the benchmarks are compiled once with prescheduling and once without it. The speedups and memory reference ratios are calculated with respect to the single-instruction-issue base architecture with 32 registers. Prescheduling is turned off and the restricted percolation model is used for the base processor. The speedup and memory reference ratio numbers show the combined effect of the 4-instruction issue rate, the general percolation model, and the register file size. The two curves show the performance with and without prescheduling. For this experiment, the change in the number of memory references is due purely to register spilling.

The results are shown in Figure 9. The execution time decreases as the number of registers increases, because there is less spill code and fewer dependencies due to the reuse of registers. When the register file size is 16, there is no advantage to prescheduling. There are too few registers and there is quite a bit more spill code when prescheduling is used. (There is a lot of spilling in general; without prescheduling there are 80% more memory accesses with 16 registers than with 32.) Whatever improvement prescheduling can make by rearranging the instructions is lost when the spill code is added. For the current register file sizes of about 32 and future sizes of 48 and larger, prescheduling has a performance advantage of approximately 14%. Prescheduling's advantage does not diminish as the register file size increases because the register allocator reuses registers in a similar way (when there is no spilling) regardless of the number of available registers. As the number of registers is increased, the register allocator may still allocate the same register to two nodes that are not adjacent (adding a dependency), when it might be able to use a different register (since there are so more) to avoid adding a



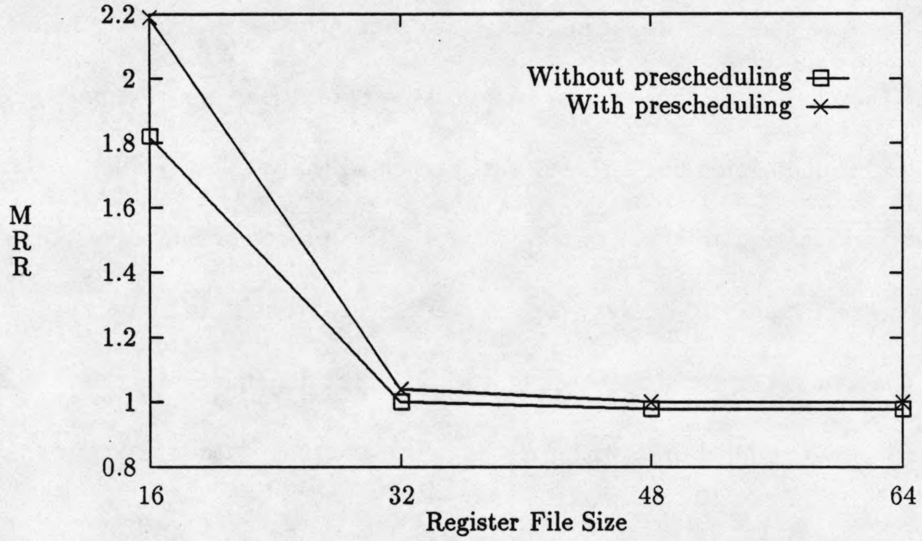


Figure 9: The performance of prescheduling for processors with various register file sizes. All the processors except the base architecture have issue rate 4 and support the general percolation model. The base architecture is a single-instruction-issue processor with no prescheduling, restricted percolation, and 32 registers.

dependency.

As the register file size increases, the difference in spilling with and without prescheduling diminishes, because the register set has more space to support the longer register lifetimes. For register file sizes 32 and larger, the difference is less than 4%. When the register file size is reduced from 64 to 32, there is almost no change in the amount of spilling, and therefore no change in speedup.

3.5.4 The Effect of Register Allocation on Code Scheduling

In this section, an experiment is performed to study how much the extra dependencies added during register allocation hinder the code scheduler given an ideal architecture. This gives an indication of how much register allocation changes the dependency graph for control-intensive programs. The effects of the hardware constraints are minimized as much as possible. We model a processor that has an unlimited instruction issue rate for all instructions, and unit instruction latencies. Unit instruction latencies were chosen so that each dependency produces the same delay and has a similar effect on the results. The processor supports the general percolation code scheduling model. We vary the register file size from 16 to 64 because this has a direct effect on the amount of register recycling and the extra dependencies added. For each case, the benchmarks are compiled once with prescheduling and once without it. The speedups and memory reference ratios are calculated with respect to the single-instruction-issue base architecture with 32 registers. Prescheduling is turned off and the restricted percolation model is used for the base processor. The speedup and memory reference ratio numbers show the combined effect of the unlimited issue rate, the general percolation model, and the register file size.

The results are shown in Figure 10 and are similar to the those for the previous experiment. The speedup over the base single-instruction-issue processor is higher due to the unlimited issue rate. Prescheduling's performance advantage for the larger register sizes increases to approximately 20% because the hardware can exploit more parallelism. The difference in the memory reference ratio is also larger because the scheduler moves instructions more to take advantage of the unlimited issue rate. For register file sizes of 32 and larger, the register allocator clearly handicaps the code scheduler by adding dependencies. The register allocator reuses registers without regard for its effect on the final schedule.

4 Conclusion

This paper discussed the interaction between register allocation and code scheduling and the importance of performing prepass as well as postpass code scheduling. The register allocator introduces extra dependencies between the instructions whenever it reuses registers and adds spill code. If code scheduling is performed only after register allocation, these extra dependencies restrict the ability of the code scheduler to move instructions to their desired positions. On the other hand, if code scheduling is done only before register allocation, the register lifetimes may be lengthened, increasing the amount of spill code added by the register allocator. There is also no opportunity to optimize the code added by the register allocator. If both prepass and postpass scheduling are performed, and the prescheduler is careful to minimize the use of registers by moving code only as much as necessary to minimize delays, better performance can be achieved and spilling can be controlled.

The IMPACT-I C compiler's code scheduler was described in detail. It is used for both

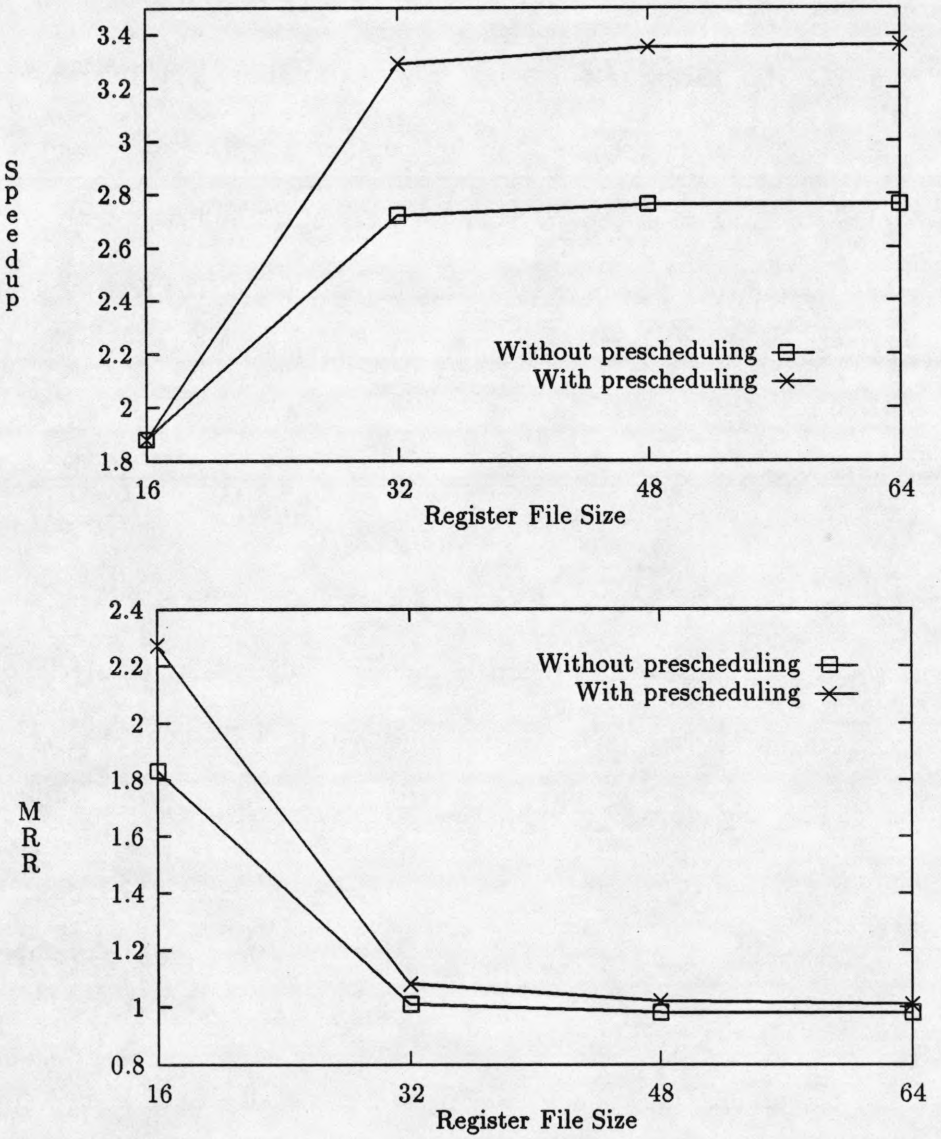


Figure 10: The performance of prescheduling for an ideal processor with various register file sizes. The ideal processor has an unlimited issue rate, unit instruction latencies and supports the general percolation model. The base architecture is a single-instruction-issue processor with no prescheduling, restricted percolation, and 32 registers.

prescheduling and postscheduling. It finds the most frequently executed paths in the functions and lays the basic blocks of the paths out sequentially in memory. Code movement and register allocation is done across basic block boundaries in order to find more fine-grain parallelism. This is especially useful for the non-numerical C programs studied in this paper because they have frequent branches.

Experimental results showed that prescheduling is not important for compiling control-intensive programs to today's architectures. Prescheduling extracts slightly more performance from each processor studied, but the frequent branches in the C programs we used combined with the inability to move loads above branches hinder the code scheduler so much that the extra dependencies added by register allocation do not create too many additional problems. This is in contrast to the results previously obtained for scientific codes. In those programs branches are less frequent, making the restrictions on code percolation less problematic and increasing the importance of prescheduling.

If the restrictions imposed by trapping instructions are removed, but prescheduling is not used, performance does not improve much for the benchmarks we looked at. The dependencies added during register allocation become the major hindrance when reorganizing the code. In order to obtain more speedup from these benchmarks using processors that exploit fine-grain parallelism, both general code percolation and prescheduling must be used. Using an intelligent scheduler, we have shown experimentally that prescheduling, combined with the general percolation code scheduling model, can substantially improve the execution time of control-intensive programs on both superscalar and superpipelined processors.

Acknowledgments

The authors would like to thank John Andrews, Dave Lilja, and Merle Levy for their comments on this paper. They would also like to acknowledge Scott Mahlke and all the members of the IMPACT research group for their support. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). Daniel Lavery is also supported by the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign under Grant DOE DE-FGO2-85ER25001 from the U.S. Department of Energy, and the IBM Corporation.

References

[1] J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[2] W. W. Hwu and P. P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 45–53, June 1988.

[3] J. L. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 422–448, July 1983.

[4] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[5] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures," Ph.D Thesis, MIT Press, Cambridge, MA, 1986.

[6] G. J. Chaitin, "Register Allocation and Spilling Via Graph Coloring," *ACM SIGPLAN Notices*, vol. 17, pp. 98–105, June 1982.

[7] D. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," Center for Reliable and High-Performance Computing Report, University of Illinois at Urbana-Champaign, Jan. 1991.

[9] G. Kane, *MIPS R2000 RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[10] Sun Microsystems, "The SPARC Architecture Manual," Part No. 800-1399-07, Revision 50, Mountain View, CA, Aug. 1987.

[11] Advanced Micro Devices, "Am29000 32-Bit Streamlined Instruction Processor," Users Manual, Sunnyvale, CA, 1988.

[12] Intel, "i860 64-bit Microprocessor," Order Number 240296-002, Santa Clara, CA, Apr. 1989.

[13] H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM Journal of Research and Development*, vol. 34, pp. 85–92, Jan. 1990.

[14] Intel, "i486 Microprocessor," Order Number 240440-001, Santa Clara, CA, Apr. 1989.

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

[16] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling Realistic C Programs," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[17] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 242–251, June 1989.

[18] P. P. Chang and W. W. Hwu, "Control Flow Optimization for Supercomputer Scalar Processing," in *Proceedings of the 1989 International Conference on Supercomputing*, July 1989.

[19] P. P. Chang and W. W. Hwu, "Forward Semantic: A Compiler-assisted Instruction Fetch Method for Heavily Pipelined Processors," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, Aug. 1989.

[20] P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," in *Proceedings of the 21st International Microprogramming Workshop*, pp. 21–29, Nov. 1988.

[21] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," Center for Reliable and High-Performance Computing Report, University of Illinois at Urbana-Champaign, Apr. 1991.

[22] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, Oct. 1987.