The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors

Pohua P. Chang Daniel M. Lavery Scott A. Mahlke

William Y. Chen Wen-mei W. Hwu*

March 8, 1994

Abstract

Superscalar and superpipelined processors utilize parallelism to achieve peak performance that can be several times higher than that of conventional scalar processors. In order for this potential to be translated into the speedup of real programs, the compiler must be able to schedule instructions so that the parallel hardware is effectively utilized. 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-numeric programs. These programs are significantly different from the scientific programs because they contain frequent branches. The compiler must do global scheduling in order to find enough independent instructions.

In this paper, the code optimizer and scheduler of the IMPACT-I C compiler is described. Within this framework, we 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 can be removed in future architectures, prescheduling would substantially improve 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 duplicating decode/issue units, 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, techniques must be used to minimize the stalls caused by the control and data dependences 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 dependences whenever it reuses registers. These extra dependences 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-numeric 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-numeric 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 limited 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. If only a small amount of parallelism can be extracted from these C programs, the restrictions imposed by the register allocator may not be significant.

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 dependences. In this work, it is assumed that the instructions are statically scheduled. All of the instruction latencies and the type and number of functional units are visible to the code scheduler.

The dependences are expressed in the form of a dependence graph. Prior to register allocation,

the only data dependences 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 dependences related to them are flow (read-after-write) dependences. For the user-level variables, there may be flow, anti- (write-after-read), and output (write-after-write) dependences for both registers and memory locations.

During register allocation, dependences resulting from the reuse and spilling of registers are added to the dependence graph. When a register is reused, anti- and output dependences 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, an anti-dependence is created because the register spilled to memory will soon be reused. A flow dependence is created because the value written to memory will eventually be read into a register again. When a register is refilled, an anti-dependence may created if the memory from which the value is read eventually gets written again.

Code scheduling can be performed either before or after register allocation, or both. No matter when scheduling is performed, the dependences in the initial code sequence constrain the reordering of instructions. If code scheduling is performed after register allocation, the scheduler is additionally restricted by the extra dependences 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 the

[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 dependence pattern created may vary. In any case, there is some given dependence pattern that the code scheduler must work with.

instructions close to their desired positions without the hindrance of the dependences caused by register recycling. 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. The scheduler can also attempt to exploit more parallelism than the register file is capable of supporting by producing too many simultaneously live values.

The above are some of the disadvantages of prescheduling. The first one 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. The second disadvantage can be reduced by increasing the register file size or more tightly integrating the code scheduler and register allocator as in [1]. We evaluate the performance of prescheduling for various register file sizes, but do not consider more integrated schemes in this paper. It is shown in Section 3 that if the prescheduling is done intelligently, the benefits of the increased code movement flexibility outweigh the cost of the extra register spilling for the control-intensive benchmarks that we studied. It is also shown that to take full advantage of the parallel microarchitectures, enough registers must be provided to hold all the simultaneously live values.

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 dependences added during register allocation. Because most of the code motion is already completed, the postpass scheduler is less hindered by the extra dependences.

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-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 [2]. 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. This example is extracted from the most frequently executed block of code generated by our compiler for the Unix utility **cmp**.

The importance of prescheduling becomes more pronounced as the intermediate code becomes more parallel. If the initial dependence graph has very few edges, then the majority of the con-

[2]For this example, we assume that the set of unused registers is managed as a stack. It is also assumed that the register allocator tries to minimize the number of registers in order to reduce the procedure call overhead associated with saving and restoring registers.

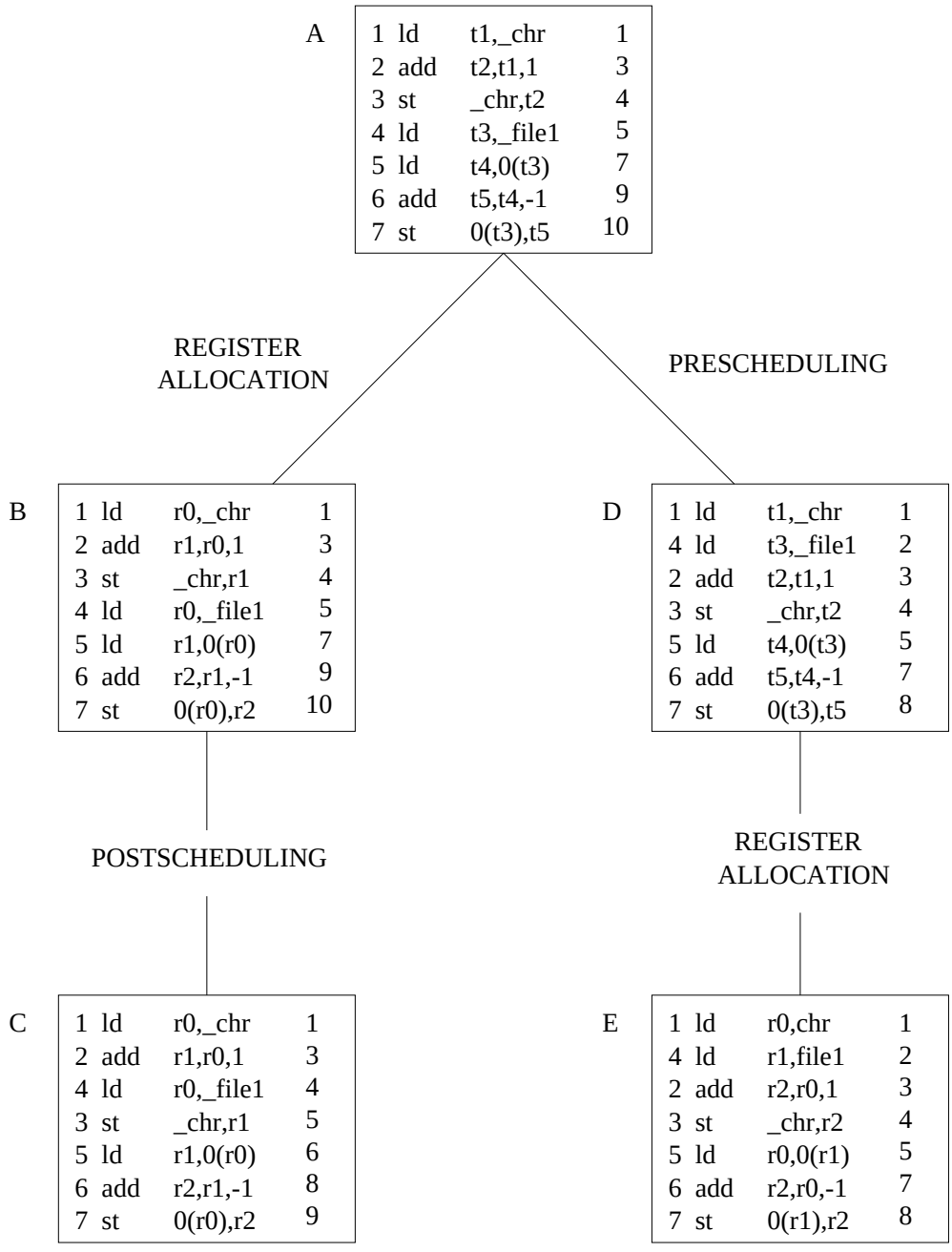

Figure 1: Examples of postpass and prepass code scheduling.

straints come from the edges added by the register allocator. This is why prescheduling is so important for scientific programs. We show that, using global optimization techniques combined with the proper architectural support, enough parallelism can be extracted from control-intensive programs to make prescheduling necessary.

2.2 IMPACT-I C Compiler

The IMPACT-I C Compiler [7] is a retargetable, optimizing compiler designed to generate very efficient code for pipelined and multiple-instruction-issue processors. Code generators have been built for the MIPS R2000 [8], the Sun SPARC [9], the AMD 29K [10], the Intel i860 [11], and the HP PA [12] processors. 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 instruction-level parallelism. The compiler contains a profiler to identify the most frequently executed program paths. This information is used to guide the global code optimization and scheduling.

IMPACT-I 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 [13], inline expansion of frequently executed functions [14], instruction placement optimization [15], profile-based classic code optimizations [16], and profile-based optimizations to increase the available instruction-level parallelism [17]. The machine-dependent optimizations include profile-based branch prediction [18], graph-coloring-based register allocation [6], and code scheduling. The results in this paper are based on the IMPACT compiler implementation. The task of evaluating the importance of the results for other compiler systems is left to the reader. The following sections describe the global register allocator and scheduler.

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 value. 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 values 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 dependences 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 studied in this paper because the parallelism within a basic block is very limited. Superblock scheduling provides an easier way to find parallelism beyond the basic block boundaries. We describe superblock scheduling here because the results presented in this paper are based on

this kind of global optimization. Superblock scheduling is performed in the six-step process shown below:

1. Trace selection
2. Superblock formation
3. Superblock 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. The final subsection comments on some concerns that have been expressed about the effects of superblock scheduling on the code size and compile time.

2.4.1 Program Representation, Profiling, and Preparation

In our C compiler, a function is represented as a weighted flow graph [16]. Several steps are taken to prepare the program for optimization and code scheduling. First, the flow graphs are generated for each function. Probes are then 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 inline expanded [14].

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**. 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 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 algorithm and heuristics we use for trace selection were first proposed by Ellis [5] and improved by Chang and Hwu [19]. An example of the result of trace selection on a weighted flow graph can be seen in [16]. 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 [3]. Once the traces have been selected, the basic blocks of each trace are laid out sequentially in memory [15]. Then superblocks are formed and optimized as described in the next subsections.

2.4.3 Step 2: Superblock Formation

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

[3] In the experiments done for this paper, the minimum count is 50 and the minimum probability is 60%.

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. 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*. 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. Only 1 copy of a block is ever made. 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.

An example of superblock formation can be seen in [16]. When a block is copied, its execution count in the original trace is reduced by the weight of the side entrances removed. If the block has multiple successors, the proportion of the weight that should be subtracted from each arc is not known. As an approximation, we multiply the weight of each outgoing arc by a fraction equal to the new weight of the block divided by the weight of the block before tail duplication. For the profile-based optimizations, this approximate information is good enough. For accurate analysis of the final schedule however, the transformed program must be reprofiled after superblock optimization.

An additional benefit of tail duplication is that code optimizations can be more easily applied to superblocks than to traces [16]. The IMPACT-I compiler uses the superblock as a common foundation for both optimizations and code scheduling.

2.4.4 Step 3: Superblock Optimization

After superblock formation, many classic code optimizations are performed that take advantage of the profile information encoded in the superblock structure[4]. These optimizations are designed to decrease the number of instructions on the frequently executed paths, perhaps at the expense of the infrequently executed ones [16]. They include: constant propagation, copy propagation, constant combining, common subexpression elimination, redundant load and store elimination, dead code removal, loop invariant code removal, loop induction variable elimination, and global variable migration.

Next, several profile-based code transformations are performed that increase the available instruction-level parallelism of the intermediate code [17] [20]. These optimizations increase the size of superblocks and eliminate data dependences between instructions. They are applied only to the most frequently executed superblocks to control code expansion and compile time. They include: branch target expansion, loop peeling, loop unrolling, register renaming, induction variable expansion, accumulator variable expansion, operation migration, operation combining, and tree height reduction.

[4]Traditional local and global optimizations that do not utilize profile information are also performed at this point [13].

2.4.5 Step 4: Dependence Graph Construction

In this step, a conservative dependence graph is built for each superblock. The dependence graph is a directed acyclic graph in which the nodes are instructions, and there is an arc from node **x** to node **y** if instruction **y** depends on instruction **x** (i.e. nodes in the graph depend on their parents). 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. Memory disambiguation is done and data-dependence arcs are added between pairs of memory references that cannot be disambiguated.

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 dependence. The anti- and output-dependence arcs have length 0. We assume that the hardware ensures that anti- and output-dependent instructions issued in the same clock cycle are executed correctly [5]. 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 [6].

[5] There are several techniques for doing this. For example, the hardware can do register renaming for register dependences and memory access sequence control for memory dependences.

[6] For the superscalar processors, multiple branches can be issued in a cycle and the architecture uses a squashing branch scheme [18].

2.4.6 Step 5: Dependence Graph Optimization

In this step, the dependence graph is optimized by removing some of the dependence arcs. During the list scheduling step (described in the next subsection), the instructions are reordered to improve the execution time within the constraints of the dependences. Instructions are moved upward and downward across branches. Moving instructions upward across branches is called speculative code motion. 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.

The first restriction can usually be eliminated with sufficient compiler variable renaming support. 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. Whenever an instruction is moved upward across a branch,

the non-trapping version is used. A similar approach has been implemented in the Multiflow Trace computer [21] [7].

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.

For programs which would never have trapped when scheduled using conventional techniques, this invalid value does not affect the correctness of the program because the results of the instructions moved above the branch are not used when the branch is taken (a result of restriction 1). However, for all other programs (e.g. undebugged code, or programs which rely on traps during normal operation), errors which would have caused a trap may now cause an exception at a later trapping instruction, or may cause an incorrect result.

Smith, Lam, and Horowitz described a method called *boosting* which uses extra hardware to remove both the first and second restriction without ignoring exceptions [22]. We have shown that boosting and general percolation have similar performance [23]. Currently, we are investigating *sentinel scheduling*, a very promising new technique which allows the code scheduling flexibility of general percolation without ignoring exceptions and without requiring much extra hardware [24]. The results achieved with general percolation in this paper confirm the importance of speculative code motion and show the potential of these new techniques.

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

[7] The Multiflow Trace eventually detects some floating-point exceptions by writing NaN to the destination register of the instruction that would have generated an exception.

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.

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 described above. Therefore, the scheduler does not move an instruction below a branch if it writes a variable that is in *live_out*(**I**). It also does not move an instruction downward across branches if it may cause an exception. In these cases, the 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.

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 various factors that the scheduler takes into account are summarized in a priority which is computed for each node in the graph before scheduling begins.

The general idea of the list scheduling algorithm is to pick, from the set of nodes in the dependence graph that are *ready* to be scheduled, the highest-priority 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 highest-priority set of nodes from the active set to fill each template. Then it issues the highest-priority instruction template and marks the nodes in the template as scheduled. The priorities of all the nodes in a template are added together to determine the highest-priority template. If there are no nodes in the active set, the scheduler does not have to issue no-ops. In this case, the flow dependences 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 priority computed for each node is the weighted sum of the values returned by several heuristic functions. Each heuristic function $\mathbf{F}_i(\mathbf{N})$ (where $\mathbf{N}$ 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 $\mathbf{W}_i$ to resolve these kinds of conflicts. The function *priority*($\mathbf{N}$) returns $\sum_{i=1}^{n} \mathbf{F}_i(\mathbf{N}) * \mathbf{W}_i$. 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.

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 have many source registers, because they may 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, **register_use(N)** might be given more weight. The **uncover(N)** heuristic might be emphasized for a architecture with lots of parallelism and a large register file. In this paper, we use the same set of weights for all of the experiments.

These heuristics were developed based on our experience with control-intensive programs and the results in this paper are based upon them. The importance of prescheduling may vary with different heuristics. The evaluation of the importance of prescheduling for different heuristics is beyond the scope of this paper.

2.4.8 The Effect of Superblock Scheduling on Compile Time and Code Size

In [20], we have measured the code expansion and compile time increase due to trace selection, superblock formation and optimization for the benchmarks used in this paper. The code size is increased by an average of 100%. Cache simulation results in [20] show that despite the code size increase, an instruction cache of 16K bytes or larger performs nearly as well as an ideal cache. Since most future processors will have an instruction cache at least this large, we do not expect code expansion to be a problem.

As with trace scheduling, superblock scheduling does increase the compilation time. The increase is about 140% on average (including profiling for one input) in our prototype compiler. However, this extra effort is worthwhile if it can significantly reduce the execution time of important frequently-executed programs such as the Unix programs that make up part of our benchmark set. Currently, most microprocessor manufacturers are already producing superscalar processors with issue rates between 2 and 5. In [20], it is shown that the superblock techniques do significantly improve the performance of important programs for these issue rates. The increased compile time can be viewed as part of the overall workload on a machine. If the time saved by the faster execution of important programs is greater than the increased compile time, then there is a net performance gain for the whole workload. During program development, when the compile time may be more critical than the run time, the compiler optimizations can be turned off.

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 again with only postscheduling turned on. For each compilation, the program execution time and the number of dynamic memory references are calculated 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. We assume a 100% cache hit rate for these experiments.

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 the 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.

As mentioned in Section 2.4.3, we reprofile the program prior to list scheduling and register allocation. During the profiling process, code is generated by IMPACT for the MIPS R2000 processor and the program is executed on a DECStation 3100. For every new compilation, we check the program output during this execution to verify the correctness of our compiler optimizations.

After list scheduling, we can again generate MIPS code, execute the scheduled code sequentially, and produce an instruction trace. Using these traces, we have simulated the superscalar execution of the benchmarks in a previous study and found that the simulated execution time matches the time calculated as described above [25]. We also verified that the program output during this execution and trace generation was correct [8].

The execution time result for each compilation is reported as a speedup relative to the compilation for the base microarchitecture. 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 that cause misses will cause the speedup reported here to be reduced. Even if the extra memory references do not cause cache misses, the delays due to the original cache misses will become relatively more significant as the execution time is decreased. Delays due to page faults will also become more significant as the the execution time is reduced.

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 issue rate and the instruction

[8] For the code scheduled using the general percolation model, we must mask the exceptions produced by the trapping instructions of the R2000. However, because we also scheduled and executed the code with the restricted percolation model, we know that the exceptions are due to general percolation rather than compiler optimization bugs.

Table 1: Instruction latencies.

FUNCTION	LATENCY	FUNCTION	LATENCY
integer ALU	1	FP ALU	3
integer multiply	3	FP multiply	3
integer divide	10	FP divide	10
integer branch	1 / 1 slot	FP branch	3 / 1 slot
load	2	FP conversion	3
store	1		

latencies), and the code scheduling model and options (this is where prescheduling is turned on and off). The base microarchitecture is a pipelined, single-issue processor that supports the restricted percolation model. Loads, stores, integer divides, and floating-point instructions can cause traps. Its instruction set is similar to the MIPS R2000 instruction set. Table 1 shows the instruction latencies. Instructions are issued in order (there is no dynamic code scheduling). The processor is assumed to have interlocks for structural and read-after-write hazards. The microarchitecture uses a squashing branch scheme [18] 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 [9]. Of the 32 integer registers, 8 are reserved as special registers (for the stack pointer, frame pointer, parameter passing registers [10], etc.). Four registers in each register file are reserved as spill registers [11]. These 12 reserved registers are not available for assignment 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

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

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

[11] The 4 spill registers are used in a round-robin fashion to reduce dependences.

and decodes them in parallel. The issue rate is the maximum number of instructions that can be fetched and issued per cycle. An instruction is held in the instruction unit if there is a flow dependence between it and a previous instruction. All the subsequent instructions are also held. All the instructions in the buffer do not have to be issued before more instructions are fetched. We assume that once the fetch address is known, the required number of instructions can be fetched in a single cache access. 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 the processors in this paper, all the functional units are capable of executing any instruction. Thus, there are no restrictions placed on the combinations of instructions that can be issued in the same cycle. When the issue rate is increased, the number of cycles of delay due to mispredicted branches remains the same, but the number of instructions squashed increases. Since the program execution time is decreased by superscalar execution, the branch penalty becomes relatively larger.

The superpipelined version of this processor has deeper pipelining for each functional unit. If the number of pipestages is increased by a factor $\mathbf{P}$, 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 microarchitecture. The throughput increases by up to the factor $\mathbf{P}$. We refer to the factor $\mathbf{P}$ as the *degree of superpipelining*. The instruction fetch and decode unit is also more heavily pipelined to keep the microarchitecture balanced. Because of this and the more deeply pipelined compare-and-branch units, the number of cycles of delay due to mispredicted branches and the number of instructions squashed increases [18].

For the superscalar processor, the additional datapaths, functional units, and instruction unit logic may increase the cycle time. For the superpipelined processor, the cycle time is actually

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
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

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. We have attempted to use diverse input data for profiling in order to optimize each program for a wide range of possible inputs. The Size column specifies the size of each program in number of lines of code. After superblock formation and optimization, each benchmark is profiled again with a single 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 3 shows the dynamic frequencies of different classes of control instructions before/after superblock formation and optimization. For example, the BRANCHES TAKEN column shows the percentage of taken branches among all instructions. A - means that the program had less than 0.1% instructions of that type. Before superblock optimization, 1 out of every 3 or 4 instructions is a control instruction. Superblock formation and optimization may decrease or increase the total percentage of control instructions. The percentage of taken branches decreases and the percentage not taken increases. The formation of superblocks places frequently executed blocks in a sequence and adjusts the appropriate branch target addresses. As a result, the branches in a superblock are usually not taken. One of the superblock optimizations reduces the number of indirect jumps. If there is a favored target for the jump, a branch is added to test specifically for that target, avoiding the indirect jump for most cases. Superblock optimizations do not change the number of function calls. However, the total number of instructions executed may increase or decrease, causing the percentage of function calls and returns to vary. As a whole, the programs are just about as control-intensive after superblock formation as before.

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 dependence pattern and allow the prescheduler to produce deceptively parallel code. On the other hand, some dependences may not be removed by a poor optimizer,

Table 3: Percentage of branches, jumps, and function calls+returns among all instructions. The statistics are shown for before/after superblock formation and optimization.

Benchmark	Branches Taken	Branches Not Taken	Uncond. Jumps	Indirect Jumps	function Calls	Total
cccp	9.2/5.2	18.3/24.2	3.1/1.7	4.2/0.2	0.3/0.3	35.1/31.5
cmp	4.0/1.3	15.1/22.5	0.1/-	0/0	-/-	19.3/23.7
compress	5.7/2.7	10.7/13.7	0.8/0.3	0/0	-/-	17.1/16.8
eqn	4.2/3.3	22.1/24.8	0.9/1.0	0.5/0.1	1.1/1.4	28.9/30.7
eqntott	14.6/9.3	23.1/19.5	0.2/0.2	-/-	0.8/0.7	38.7/29.8
espresso	9.7/4.1	9.2/14.2	1.1/0.6	0/0	0.7/0.6	20.6/19.6
grep	6.8/1.6	38.5/47.2	6.5/1.0	-/-	-/-	51.7/49.8
lex	18.1/3.4	18.7/32.8	0.5/-	-/-	-/-	37.4/36.3
li	5.3/4.4	16.7/17.6	1.7/1.0	0/0	2.3/2.3	25.9/25.3
qsort	9.1/2.7	5.7/10.4	2.0/0.7	0/0	1.2/1.1	18.0/14.9
tbl	17.6/3.7	10.3/25.9	0.5/0.6	-/-	2.2/2.4	30.7/32.5
wc	5.5/2.4	24.9/31.2	2.6/2.0	0/0	-/-	33.0/35.6
yacc	15.9/4.1	16.7/27.2	0.4/0.5	-/-	0.3/0.3	33.3/32.1

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[12], 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 [16]. Thus, the evaluation of prescheduling reported in this paper is based on well optimized sequential code.

[12]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 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 instruction-level 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 all 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-issue base architecture described in Section 3.2. Prescheduling is turned off for the 32-register base architecture. The changes in the amount of memory references for this experiment are purely due to spilling because loads cannot be moved above branches.

The individual results for each benchmark are shown in Figure 2. The gray bars in the charts show the increased performance and number of memory references, without prescheduling, compared to the base processor for issue rates 2, 4, and 8. The black bars in the speedup chart show the additional speedup due to prescheduling. In the MRR chart, the black bars show the total MRR with prescheduling. The separate bars are used because prescheduling does not always produce an increase in the MRR.

Prescheduling extracts little or no extra performance for most of the benchmarks. **compress** is

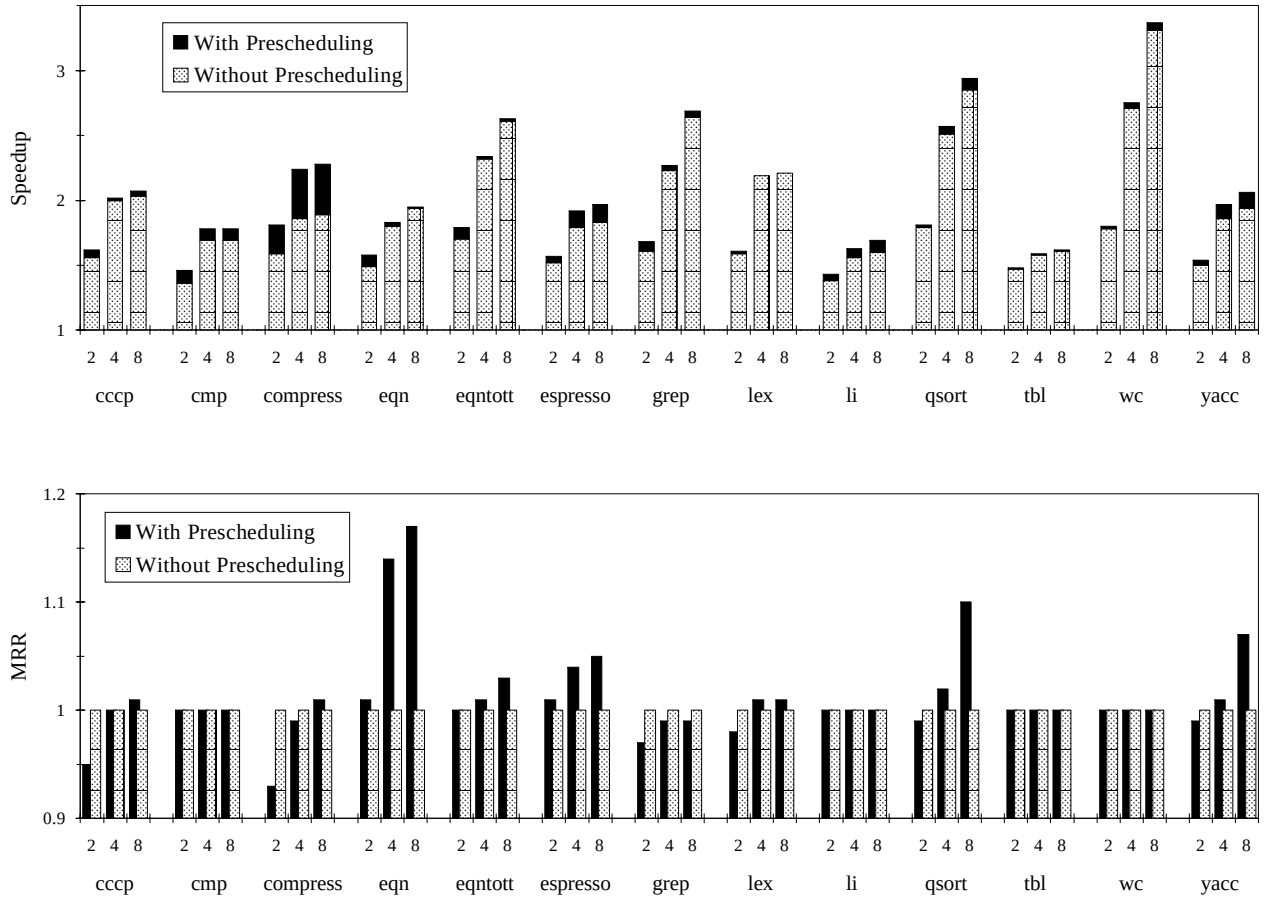

Figure 2: The performance of prescheduling for the superscalar versions of existing architectures. The base architecture is the single-issue processor with no prescheduling. All the processors have 32 registers.

the only benchmark that shows a really marked improvement. For issue rate 4, the average speedup with prescheduling is less than 4% higher than without it. Performance for a single-issue processor (not shown) is also improved very slightly for each benchmark. 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 the ability of the prescheduler to optimize the critical path.

The memory reference ratio is always 1 without prescheduling. This means that there are no more memory references than for the single-issue base architecture. The reason for this is that the register allocation algorithm assigns registers in the same way regardless of the issue rate. For issue rate 2, there are often 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 the chance 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 values are simultaneously live, demanding more registers and increasing the amount of spilling. For **eqn** and **qsort** there are 10 to 18% more memory references as a result of prescheduling and little additional performance. In these cases, the improvements in the schedule made before register allocation are are offset by extra spills. For **compress**, the number of memory references is reduced or only slightly increased even though there are large gains in performance. The average MRR is only 2% higher with prescheduling.

In the second experiment, the benchmarks are compiled for superpipelined processors with the degree of superpipelining varied from 2 to 3. We refer to these as 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-issue base architecture as for the first experiment.

The results are shown in Figure 3. Processors 2 and 3 are single-issue, 2X- and 3X-superpipelined

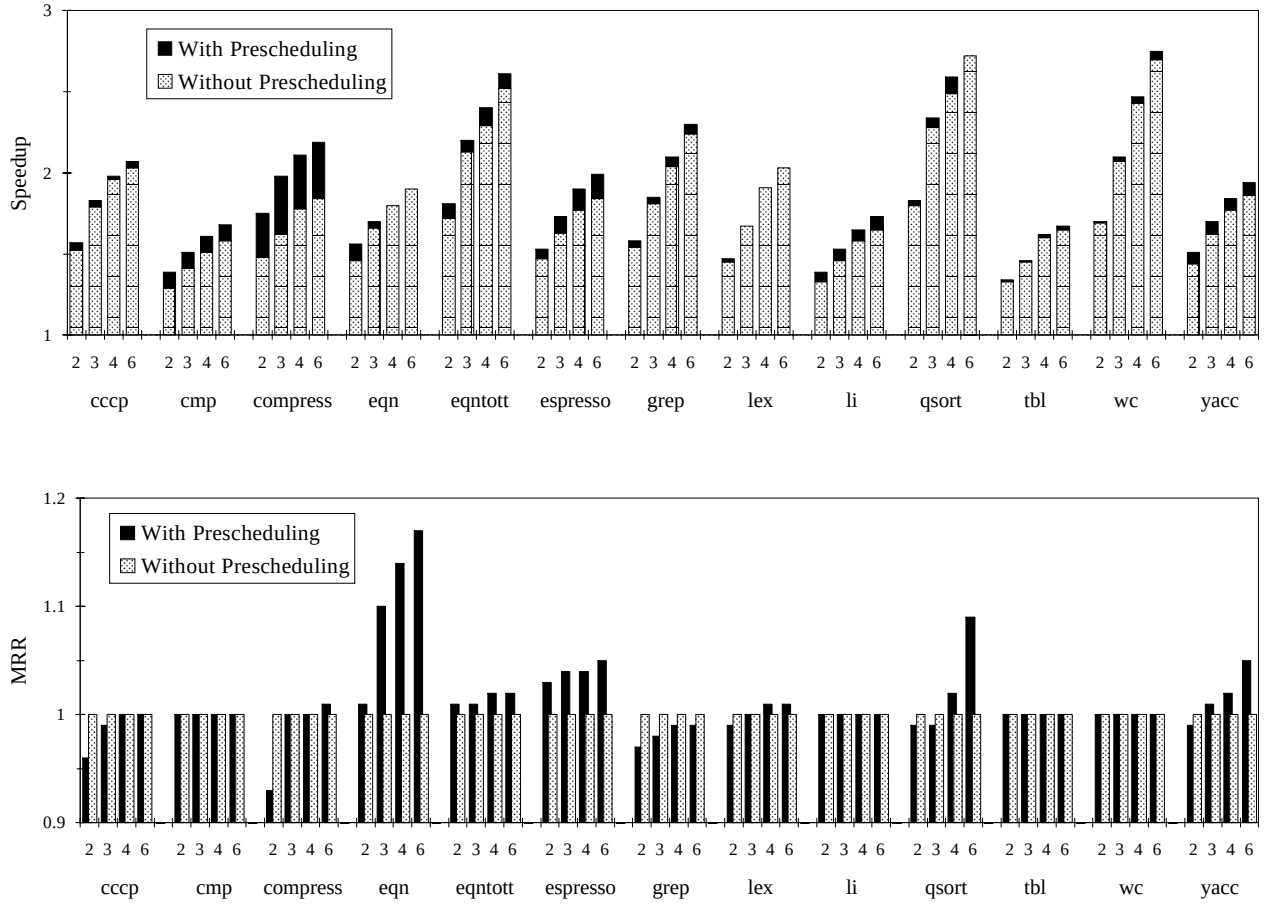

Figure 3: The performance of prescheduling for the superpipelined versions of existing architectures. The base architecture is the single-issue processor with no prescheduling. 2 and 3 are single-issue, 2X- and 3X-superpipelined processors respectively. 4 and 6 are dual-issue, 2X- and 3X-superpipelined processors respectively. All the processors have 32 registers.

microarchitectures respectively. Processors 4 and 6 are dual-issue, 2X- and 3X-superpipelined microarchitectures. In the previous experiment, there was often a large gap in performance between

the processors with issue rates 2 and 4. The single-issue 3X-superpipelined microarchitecture fills that gap. Again prescheduling has only a small advantage in speedup for most of the benchmarks and only **compress** is improved dramatically. The MRR results are also similar to the previous experiment.

The results of this section show that prescheduling is not important for compiling control-intensive programs to today's architectures. For the restricted percolation model, the frequent branches in the C benchmarks hinder the code scheduler so much that the extra dependences added by register allocation don't have much effect. In order to exploit more instruction-level parallelism in these benchmarks, some way must be found to eliminate 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-issue base architecture described in Section 3.2. Prescheduling is turned off and restricted percolation is used for this 32-register base architecture. Therefore, the speedup and change in memory references due to both prescheduling and the general code percolation model are 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 4. The performance advantage of prescheduling is now much more pronounced. For issue rate 4, prescheduling improves the speedup by more than 10% for every benchmark except **eqntott** and **qsort**. For issue rate 8, the speedups of **cmp**, **lex**, and **tbl** are improved by more than 95%. The average speedup is increased by 26% for issue rate 4. 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. Note that without prescheduling, the speedup with general percolation is not much better than for restricted percolation (in section 3.5.1). The hardware that supports the general percolation model provides richer opportunities for parallelism, but prescheduling is required to take advantage of them.

The memory reference ratio is no longer constant without prescheduling. The increase over the base microarchitecture is due only to loads that are moved from below to above branches. Prescheduling now increases the number of memory references in almost every case as it exploits the opportunities provided by the general percolation model. The increase in memory references is usually 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

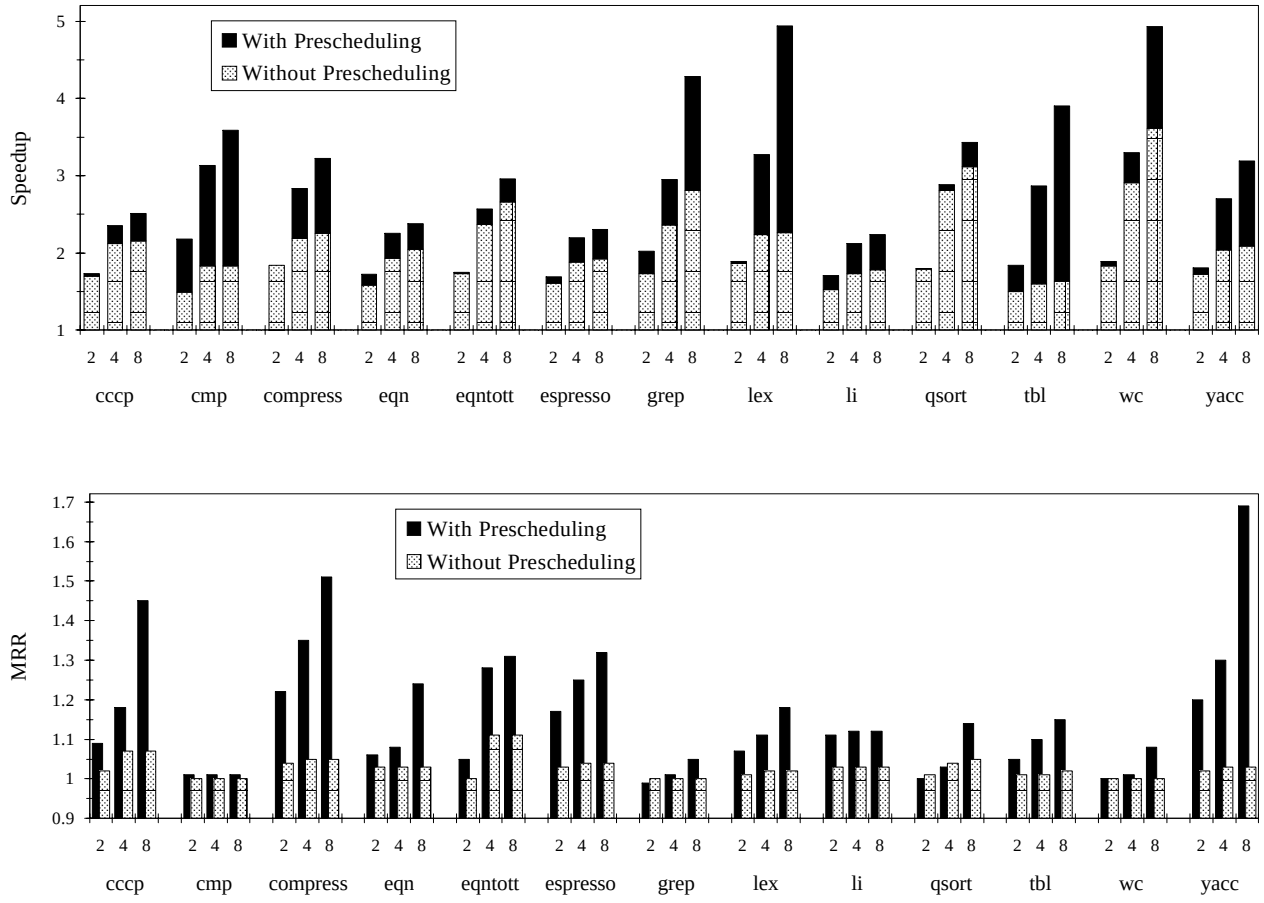

Figure 4: The performance of prescheduling for the superscalar versions of architectures that support general percolation. The base architecture is a single-issue processor with no prescheduling and restricted percolation. All the processors have 32 registers.

of registers used. The speedup with prescheduling increases faster than without it in spite of the MRR, which also increases faster with prescheduling. At the high issue rates, there are more unused instruction slots to hide spill code, and the extra parallelism exploited overcomes any loss due to spill code that cannot be hidden. Prescheduling increases the average MRR by 11%.

The speedups for **compress** and **qsort** are very slightly lower with prescheduling (which cannot be seen in the figure) for issue rate 2. There are more memory references for **compress** with prescheduling. When the issue rate is 2 it is more difficult to find empty instruction slots in which to hide spill code. For **qsort**, the MRR is very slightly improved, indicating that some temporaries may have been produced too early in the original optimized code. For issue rate 1 (not shown), the speedup is slightly lower with prescheduling than without for about half of the benchmarks. In these cases, the MRR is usually slightly higher. For some of the benchmarks, particularly **cccp** and **yacc**, at issue rate 8, the increase in the MRR with prescheduling is quite a bit higher than the increase in the speedup. The above results indicate that performance may be further improved by more tightly integrating the code scheduler and register allocator.

In second experiment, the benchmarks are compiled for superpipelined processors with the degree of superpipelining varied from 2 to 3. These processors have 32 registers. For each case, the benchmarks are compiled once with prescheduling and once without it. Again, the compiler uses the general percolation model. The speedups and memory reference ratios are calculated with respect to the familiar single-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 5. The increases in performance for prescheduling with the general percolation model are similar to those described for the superscalar processors. The MRR results are also similar. Prescheduling can exploit both the superscalar and superpipelined mi-

croarchitectures very well. Note that for the 3X-superpipelined processors, the single-issue version with prescheduling often outperforms the dual-issue version that does not have prescheduling. Also note that the speedups for the dual-issue superscalar machine (in the previous experiment) and the single-issue, 2X-superpipelined machine are similar. The same is true for the superscalar machine with issue rate 4 and the dual-issue, 2X-superpipelined machine. This matches results that have been reported in the literature before [26].

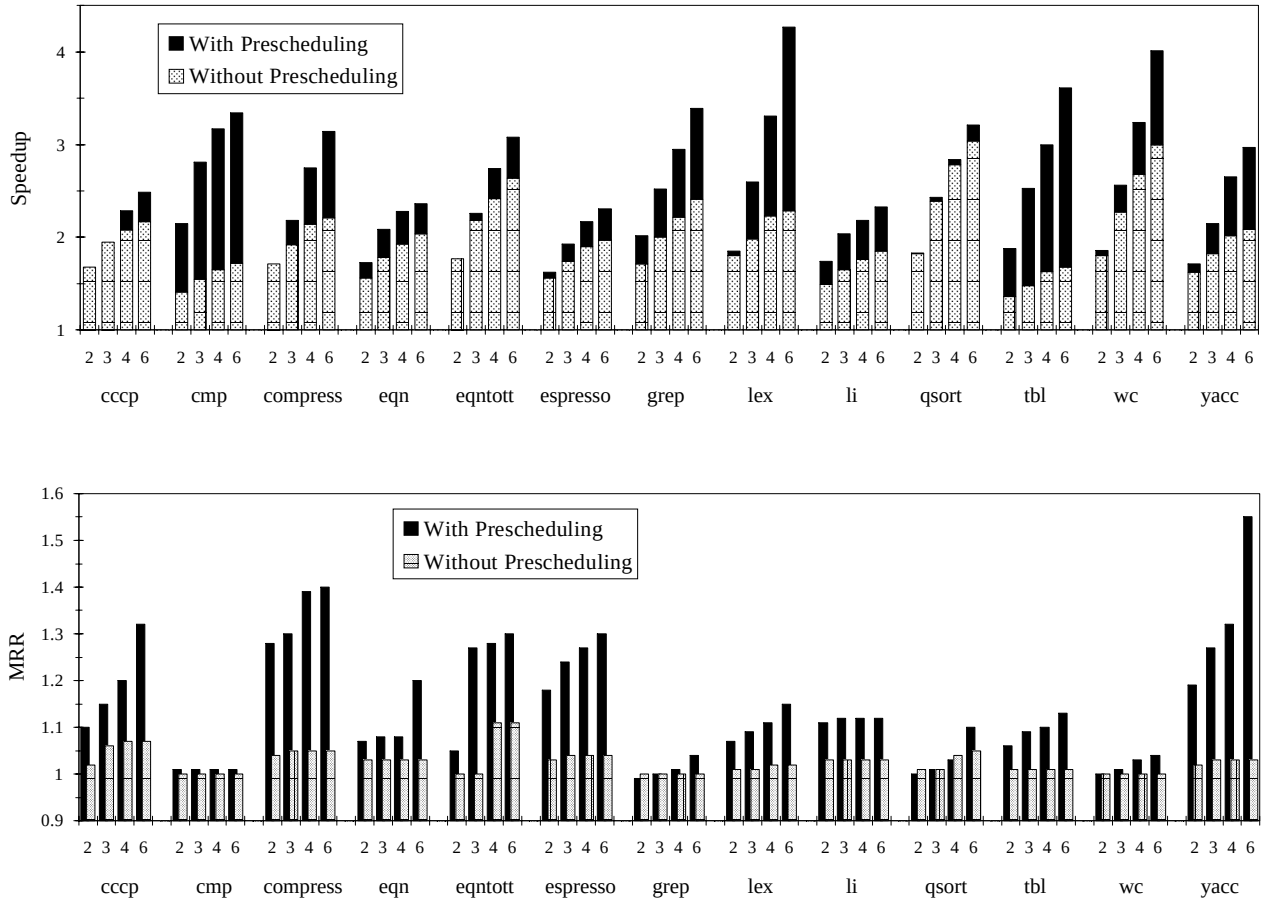

Figure 5: The performance of prescheduling for the superpipelined versions of architectures that support general percolation. The base architecture is a single-issue processor with no prescheduling and restricted percolation. 2 and 3 are single-issue, 2X- and 3X-superpipelined processors respectively. 4 and 6 are dual-issue, 2X- and 3X-superpipelined processors respectively. All the processors have 32 registers.

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 dependences added during register allocation become the major impediment to reorganizing the code. Without prescheduling, the added dependences 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.

Between the time that we submitted the first version of this paper for review and the time of the final draft, we continued to add optimizations to our compiler that increased the available instruction-level parallelism of the intermediate code. For this final draft, we repeated our experiments and found that the advantage of prescheduling was greater for the more parallel code. We predict the importance of prescheduling will continue to increase in the future as improved compiler optimization techniques find more parallelism.

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 from 24 to 64 registers (recall that 12 of these registers are reserved). 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-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. For this experiment, the change in the number of memory references between register file sizes is due purely to register spilling.

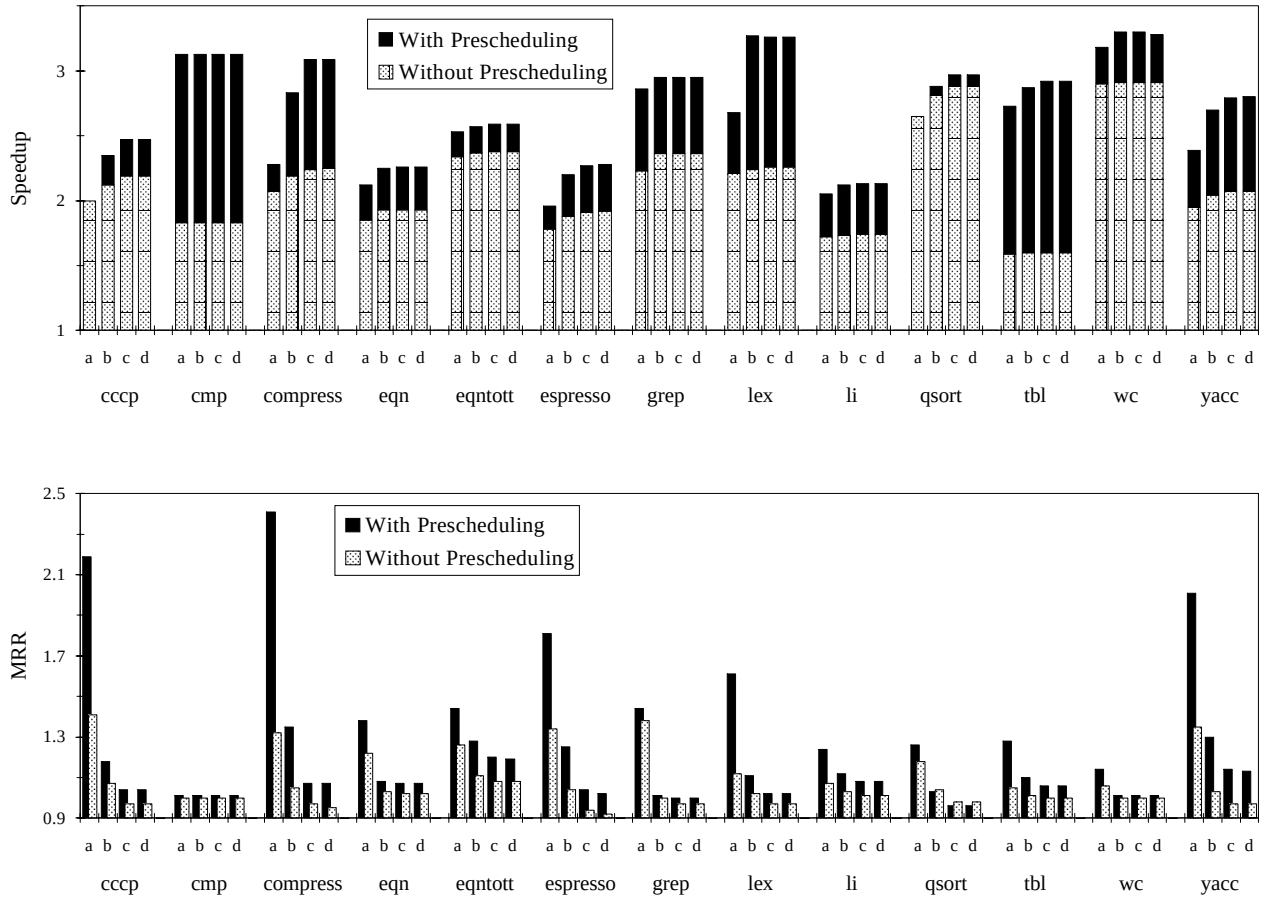

Figure 6: The performance of prescheduling for processors with various register file sizes. Processors a, b, c, and d have 24, 32, 48, and 64 registers respectively. All the processors except the base architecture have issue rate 4 and support the general percolation model. The base architecture is a single-issue processor with no prescheduling, restricted percolation, and 32 registers.
.

The results are shown in Figure 6. Machines a, b, c, and d have a register file size of 24, 32,

48, and 64 respectively. The execution time decreases as the number of registers increases, because there is less spill code and fewer dependences due to the reuse of registers. When the register file size is 24, there is still an advantage to prescheduling. However, there is quite a bit of extra spilling. **cccp** and **qsort** show very slight performance decreases with prescheduling because of this. More registers are probably needed to fully exploit the parallelism. We also obtained results for 16 registers (of which 12 are reserved). 4 allocatable registers is simply not enough. Many of the benchmarks ran slower with prescheduling than without, and prescheduling greatly increased the number of memory references. 32 registers is a reasonable number for these benchmarks. There is only a small increase in performance when moving to 48 or 64 registers and the MRR increase is fairly low. The processors with 48 and 64 registers perform equivalently.

Prescheduling's advantage does not diminish as the register file size increases because the register allocator reuses registers in a similar way regardless of the number of available registers. Note that without prescheduling, little advantage is taken of the increasing register file size. The small increase in performance is due to a decrease in spills. The code scheduler cannot take advantage of the larger register file size to increase performance further. As the number of registers is increased, the register allocator may still allocate the same register to two nodes that are not adjacent, when it might be able to use a different register to avoid adding a dependence. The average increase in performance due to prescheduling is at least 26% for register file sizes of 32 or greater. As the register file size increases, the difference in MRR 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 11%.

3.5.4 The Effect of Register Allocation on Code Scheduling

In this section, an experiment is performed to study how much the extra dependences added during register allocation hinder the code scheduler given an ideal architecture. This gives an indication of how much register allocation changes the dependence 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 issue rate for all instructions, and unit instruction latencies. Unit instruction latencies were chosen so that each dependence 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 24 to 64 because this has a direct effect on the amount of register recycling and the extra dependences 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-issue base architecture with 32 registers. The base architecture has the latencies shown in Table 1. 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 unit latencies, the general percolation model, and the register file size.

The results are shown in Figure 7 and are similar to the those for the previous experiment. The speedup over the base single-issue processor is higher due to the unlimited issue rate and unit latencies. Prescheduling's performance advantage for the larger register sizes increases to approximately 43% on average 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 dependences. The register allocator reuses registers

without regard for the effect on the final schedule. There is now a more pronounced difference in spilling between 32 and 48 registers. This is because exploiting more parallelism requires more registers. For 48 or more registers, there is less than 11% more spilling with prescheduling.

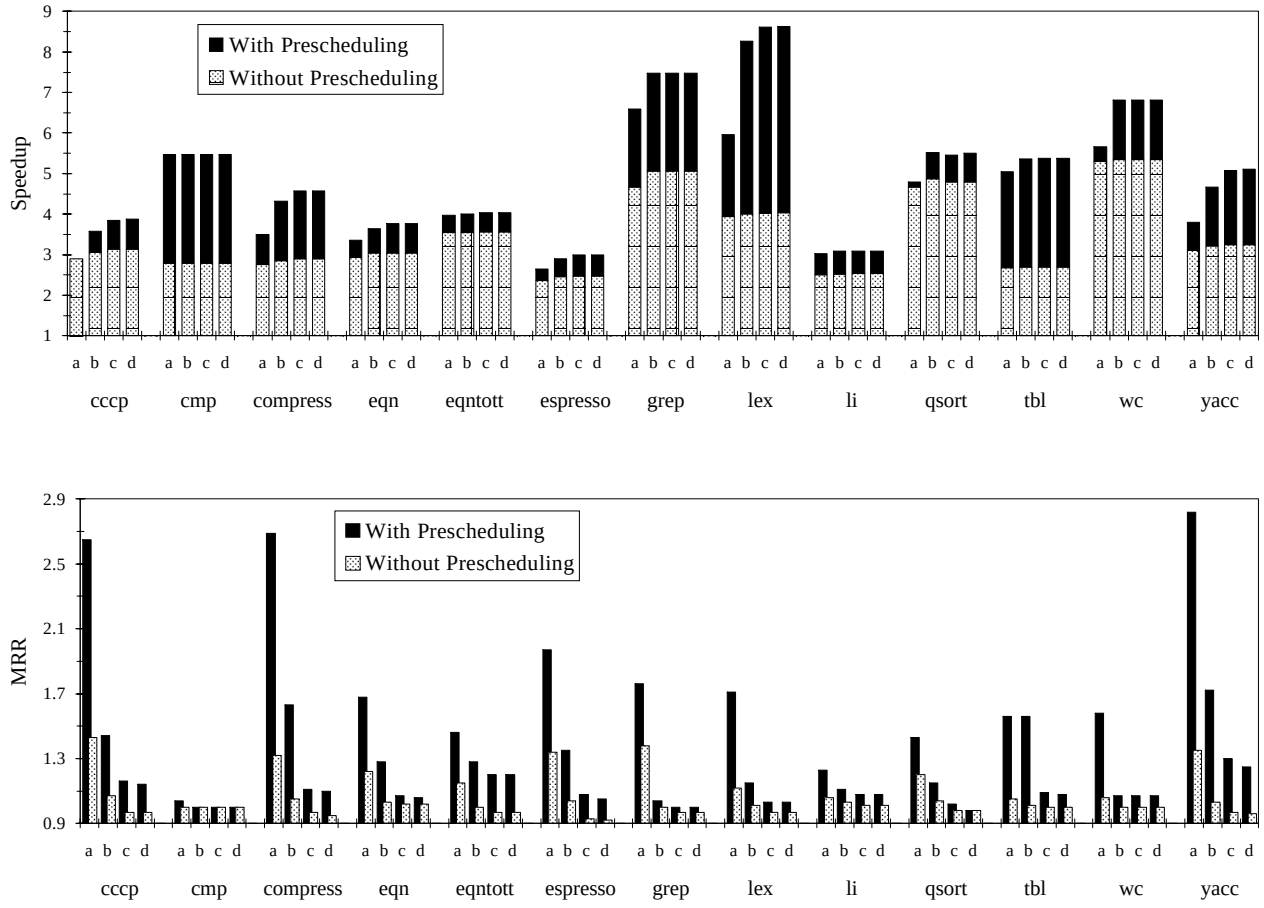

Figure 7: The performance of prescheduling for an ideal processor with various register file sizes. The ideal processor has an unlimited issue rate, unit instruction latencies and supports the general percolation model. Processors a, b, c, and d have 24, 32, 48, and 64 registers respectively. The base architecture is a single-issue processor with no prescheduling, restricted percolation, and 32 registers.

.

4 Conclusion

This paper studied 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 dependences between the instructions whenever it reuses registers and adds spill code. If code scheduling is performed only after register allocation, these extra dependences 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 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 instruction-level parallelism. This is especially useful for the non-numeric 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 dependences 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 dependences 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 instruction-level 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 the first version of this paper and Roger Bringmann for his assistance with the graphs. They would also like to acknowledge 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-9308013, 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] 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," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, May 1991.

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

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

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

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

[12] Hewlett-Packard Company, "Precision Architecture and Instruction Set Reference Manual, 3rd Ed.," Part Number 09740-90039, Apr. 1989.

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

[14] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-Guided Automatic Inline Expansion for C Programs," *Software Practice and Experience*, vol. 22, pp. 349–369, May 1992.

[15] 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.

[16] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[17] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler Code Transformations for Superscalar-Based High-Performance Systems," in *Proceedings of Supercomputing '92*, Nov. 1992.

[18] 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.

[19] 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.

[20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *to appear in The Journal of Supercomputng.*

[21] 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.

[22] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[23] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three Superblock Scheduling Models for Superscalar and Superpipelined Processors," Center for Reliable and High-Performance Computing Report CRHC-91-25, University of Illinois at Urbana-Champaign, Oct. 1991.

[24] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," in *Proceedings of the Fifth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[25] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors," in *Proceedings of the 24th International Symposium and Workshop on Microarchitecture*, Nov. 1991.

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