Three Architectural Models for Compiler-Controlled Speculative Execution

*Pohua P. Chang Nancy J. Warter Scott A. Mahlke William Y. Chen Wen-mei W. Hwu**

Abstract

To effectively exploit instruction level parallelism, the compiler must move instructions across branches. When an instruction is moved above a branch that it is control dependent on, it is considered to be speculatively executed since it is executed before it is known whether or not its result is needed. There are potential hazards when speculatively executing instructions. If these hazards can be eliminated, the compiler can more aggressively schedule the code. The hazards of speculative execution are outlined in this paper. Three architectural models: restricted, general and boosting, which have increasing amounts of support for removing these hazards are discussed. The performance gained by each level of additional hardware support is analyzed using the IMPACT C compiler which performs superblock scheduling for superscalar and superpipelined processors.

Index terms - Conditional branches, exception handling, speculative execution, static code scheduling, superblock, superpipelining, superscalar.

*The authors are with the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, Illinois, 61801.

1 Introduction

For non-numeric programs, there is insufficient instruction level parallelism available within a basic block to exploit superscalar and superpipelined processors [1][2][3]. To schedule instructions beyond the basic block boundary, instructions have to be moved across conditional branches. There are two problems that need to be addressed in order for a scheduler to move instructions above branches. First, to schedule the code efficiently, the scheduler must identify the likely executed paths and then move instructions along these paths. Second, when the branch is mispredicted, executing the instruction should not alter the behavior of the program.

Dynamically scheduled processors can use hardware branch prediction [4] to schedule instructions from the likely executed path or schedule instructions from both paths of a conditional branch such as in the IBM 360/91 [5]. Statically scheduled processors can either predict the branch direction using profiling or some other static branch prediction mechanism or use guarded instructions to schedule instructions along both paths [6]. For loop intensive code, static branch prediction is accurate and techniques such as loop unrolling and software pipelining are effective at scheduling code across iterations in a well-defined manner [7][8][9][10]. For control intensive code, profiling provides accurate branch prediction [11]. Once the direction of the branch is determined, blocks which tend to execute together can be grouped to form a trace [12][13]. To reduce some of the bookkeeping complexity, the side entrances to the trace can be removed to form a superblock [14].

In dynamically and statically scheduled processors in which the scheduling scope is enlarged by predicting the branch direction, there are possible hazards to moving instructions above branches if the instruction is speculatively executed. An instruction is speculatively executed if it is moved above a conditional branch that it is control dependent upon [15]. A speculatively executed in-

struction should neither cause an exception which terminates the program nor incorrectly overwrite a value when the branch is mispredicted. Various hardware techniques can be used to prevent such hazards. Buffers can be used to store the values of the moved instructions until the branch commits [16][2][17]. If the branch is taken, the values in the buffers are squashed. In this model, exception handling can be delayed until the branch commits. Alternatively, non-trapping instructions can be used to guarantee that a moved instruction does not cause an exception [18].

In this paper we focus on static scheduling using profiling information to predict the branch direction. We present a superblock scheduling algorithm that supports three code percolation models which require varying degrees of hardware support to enable code motion across branches. We present the architecture support required for each model. Our experimental results show the performance of the three models on superscalar and superpipelined processors.

2 Superblock Scheduling

Superblock scheduling is an extension to trace scheduling [12] which reduces some of the bookkeeping complexity. The superblock scheduling algorithm is a four-step process,

1. trace selection,
2. superblock formation and enlarging,
3. dependence graph generation, and
4. list scheduling.

Steps 3 and 4 are used for both prepass and postpass code scheduling. Prepass code scheduling is performed prior to register allocation to reduce the effect of artificial data dependences that are

```
avg = 0;
weight = 0;
count = 0;
while(ptr != NIL)  {
   count = count + 1;
   if(ptr->wt < 0)
      weight = weight - ptr->wt;
   else
      weight = weight + ptr->wt;
   ptr = ptr->next;
}
if(count != 0)
   avg = weight/count;
```

Figure 1: C code segment.

introduced by register assignment [19][20]. Postpass code scheduling is performed after register allocation.

The C code segment in Figure 1 will be used in this paper to illustrate the superblock scheduling algorithm. Compiling the C code segment for a load/store architecture produces the assembly language shown in Figure 2. The assembly code format is *opcode destination, source1, source2* where the number of source operands depends on the opcode. The weighted control flow graph of the assembly code segment is shown is Figure 3. The weights on the arcs of the graph correspond to the execution frequency of the control transfers. For example, basic block 2 (**BB2**) executed 100 times with the control going from **BB2** to **BB4** 90% of the time and from **BB2** to **BB3** the remaining 10% of the time. This information can be gathered using profiling.

The first step of the superblock scheduling algorithm is to use trace selection to form traces from the most frequently executed paths of the program [12]. Figure 4 shows the portion of the control flow graph corresponding to the `while` loop after trace selection. The dashed box outlines the most frequently executed path of the loop. In addition to a top entry and a bottom exit point, traces can have multiple side entry and exit points. A side entry point is a branch into the middle of a trace and a side exit is a branch out of the middle of a trace. For example, the arc from **BB2**

```
(i1)          load   r1,   _ptr
(i2)          mov    r7,   0          // avg
(i3)          mov    r2,   0          // count
(i4)          mov    r3,   0          // weight
(i5)          beq    L3,   r1,   0
(i6)    L0:   add    r2,   r2,   1
(i7)          load   r4,   0[r1]      // ptr->wt
(i8)          bge    L1,   r4,   0
(i9)          sub    r3,   r3,   r4
(i10)         br     L2
(i11)   L1:   add    r3,   r3,   r4
(i12)   L2:   load   r1,   4[r1]
(i13)         bne    L0,   r1,   0
(i14)   L3:   beq    L4,   r2,   0
(i15)         div    r7,   r3,   r2
(i16)         store  _avg  r7
(i17)   L4:
```

Figure 2: Assembly code segment.

to **BB3** in Figure 4 is a side exit and the arc from **BB3** to **BB5** is a side entrance.

To move code across a side entrance, complex bookkeeping is required to ensure correct program execution [12][21]. For example, to schedule the code within the trace efficiently, it may be desirable to move instruction **i12** from **BB5** to **BB4**. To ensure correct execution when the control flow is through **BB3**, **i12** must also be copied into **BB3** and the branch instruction **i10** must be modified to point to instruction **i13**. If there were another path out of **BB3** then a new basic block would need to be created between **BB3** and **BB5** to hold instruction **i12** and a branch to **BB5**. In this case, the branch instruction **i10** would branch to the new basic block.

The second step of the superblock scheduling algorithm is to form superblocks. Superblocks avoid the complex repairs associated with moving code across side entrances by removing all side entrances from a trace. Side entrances to a trace can be removed using a technique called *tail duplication* [14]. A copy of the tail portion of the trace from the side entrance to the end of the trace is appended to the end of the function. All side entrances into the trace are then moved to the corresponding duplicate basic blocks. The remaining trace with only a single entrance is a superblock. Figure 5 shows the loop portion of the control flow graph after superblock formation

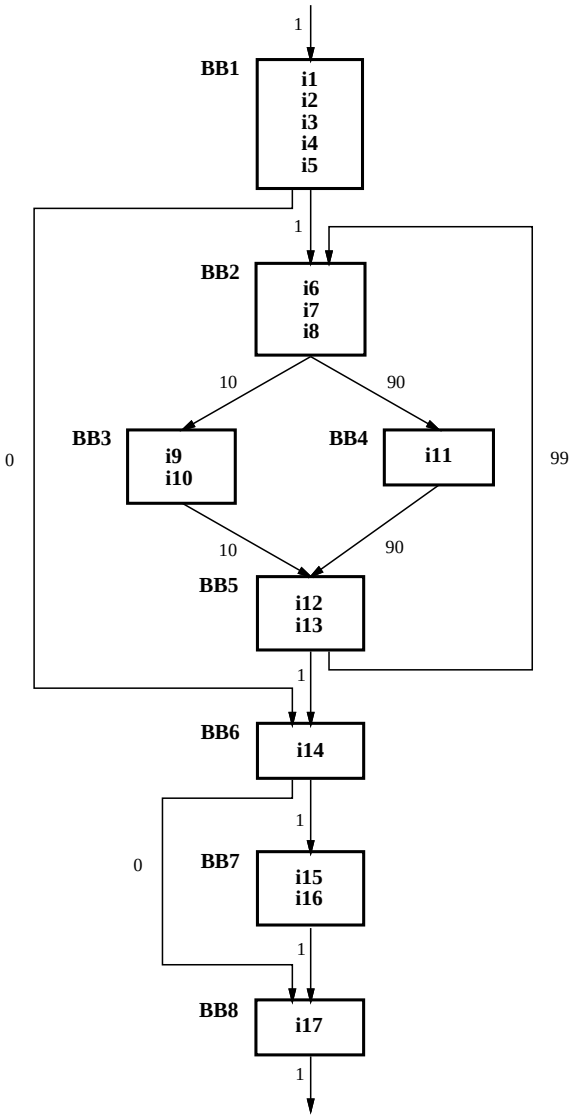

Figure 3: Weighted control flow graph.

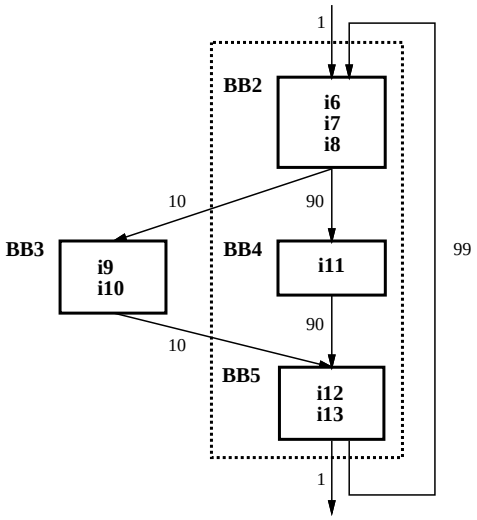

Figure 4: Loop portion of control flow graph after trace selection.

and branch expansion.[1] During tail duplication, BB5 is copied to form superblock 2, (**SB2**). Since **BB3** only branches to **BB5**, the branch instruction **i10** can be removed and the two basic blocks merged to form **BB3'**. Note that superblock 1, **SB1**, no longer has a side entrance.

Loop-based transformations such as loop peeling and loop unrolling [22] can be used to enlarge superblock loops, a superblock which ends with a control flow arc to itself. For superblock loops that usually iterate only a small number of times, a few iterations can be peeled off and added to the superblock. For most cases, the peeled iterations will suffice and the body of the loop will not need to be executed. For superblock loops that iterate a large number of times, the superblock loop is unrolled several times.

After superblock formation many classic code optimizations are performed to take advantage of the profile information encoded in the superblock structure and to clean up the code after the above transformations. These optimizations include the local and global versions of: constant propaga-

[1] Note that the profile information is scaled during tail duplication. This reduces the accuracy of the profile information.

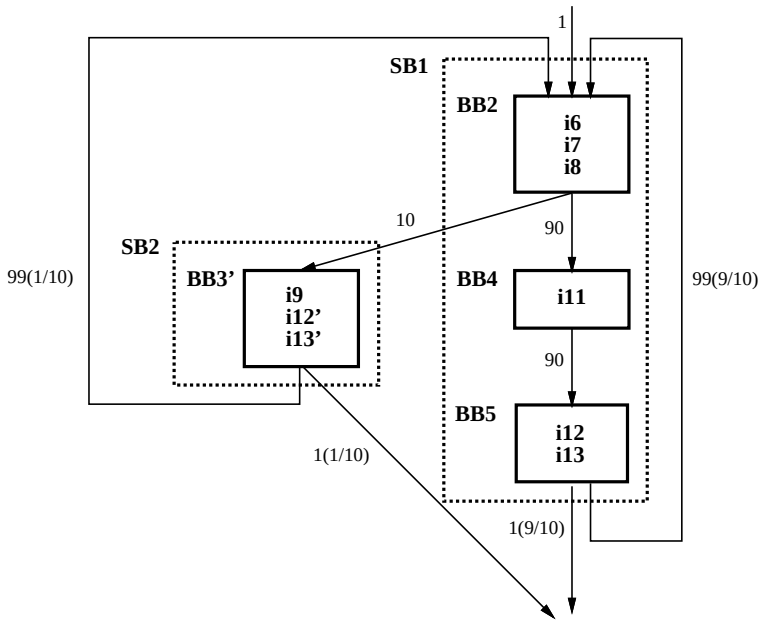

Figure 5: Loop portion of control flow graph after superblock formation and branch expansion.

tion, copy propagation, common subexpression elimination, redundant load and store elimination, dead code removal, branch expansion and constant folding [23][14]. Local strength reduction, local constant combining and global loop invariant code removal, loop induction strength reduction, and loop induction elimination are also performed. To improve the amount of parallelism in superblocks, register renaming, loop induction variable expansion, accumulator expansion, and tree height reduction are applied to each superblock [21].

The third step in the superblock scheduling algorithm is to build a dependence graph. The dependence graph represents the data and control dependences between instructions. There are three types of data dependences, flow, anti, and output. Control dependences represent the ordering between a branch instruction and the instructions following the branch. There is a control dependence between a branch and a subsequent instruction i if the branch instruction must execute before instruction i.

The last step in the scheduling algorithm is to perform list scheduling using the dependence graph and instruction latencies to indicate which instructions can be scheduled together. 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. The best combination of nodes is determined by using heuristics which assign priorities to the ready nodes [20]. A node is ready if all of its parents in the dependence graph have been scheduled and the result produced by each parent is available.

If the number of dependences are reduced, a more efficient code schedule can be found. Of the data dependences, only the flow dependences are *true* dependences. Output and anti dependences are formed when registers are reused. Hardware or software renaming can be used to remove these dependences. Control dependences can also be removed by adding hardware support. If a control dependence is removed, the corresponding instruction can be moved across the branch. Three superblock scheduling models with increasing degrees of hardware support to enable code motion across branches are presented in the next section.

3 Code Motion Across Branches

The instructions within a superblock are placed linearly in instruction memory. Thus, the side exits of the superblock correspond to conditional branches where the branch is likely not taken. To efficiently schedule code within a superblock, the compiler should be able to move instructions across branches. Let I and Br denote two instructions where I is the instruction to move and Br is a branch instruction. We define *live-out(Br)* as the set of variables which may be used before defined when Br is taken. Moving I from above to below Br (downward code motion) is relatively

straight forward. If Br does not depend on I then I can be moved below Br. If the destination register of I is in live-out(Br), then a copy of I must be inserted between Br and its target.

In order to reduce the critical path of a superblock, upward code motion is more common. For instance, moving a load instruction earlier to hide the load delay. When an instruction is moved upward across a branch, it is executed speculatively since the result of the instruction is only needed when the branch is not taken. For upward code motion, moving instruction I from below to above branch Br, there are two major restrictions.

Restriction 1: The destination of I is not in live-out(Br).

Restriction 2: I must not cause an exception that may terminate the program execution when Br is taken.

Three superblock scheduling models: *restricted code percolation*, *general code percolation*, and *boosting code percolation* require varying degrees of hardware support to remove part or all of the restrictions on upward code motion. The restricted code percolation model enforces both Restrictions 1 and 2. Only instructions that cannot cause exceptions and those that do not overwrite a value in the live-out set of the taken path of a conditional branch can be moved above the branch. The general code percolation model strictly enforces Restriction 1 but not Restriction 2. In the boosting code percolation model [17], code motion is unrestricted. In the Section 4 we discuss the architecture support required for each model.

Examples of code motion can be shown using the assembly code in Figure 6. This is the assembly code of the C code in Figure 1 after superblock formation. The loop has been unrolled once to allow more code motion and to illustrate the hazards of moving instructions across branches. Only the instructions within the superblock loop have been labeled. In the unrolled iteration, registers r1

```
                load    r1,   _ptr
                mov     r7,   0          // avg
                mov     r2,   0          // count
                mov     r3,   0          // weight
                beq     L3,   r1,   0
(I1)    L0:     add     r2,   r2,   1
(I2)            load    r4,   0[r1]      // ptr->wt
(I3)            blt     L1,   r4,   0
(I4)            add     r3,   r3,   r4
(I5)            load    r5,   4[r1]      // ptr->next
(I6)            beq     L3,   r5,   0
(I7)            add     r2,   r2,   1
(I8)            load    r6,   0[r5]      // ptr->wt
(I9)            blt     L1',  r6,   0
(I10)           add     r3,   r3,   r6
(I11)           load    r1,   4[r5]      // ptr->next
(I12)           bne     L0,   r1,   0
        L3:     beq     L4,   r2,   0
                div     r7,   r3,   r2
                store   _avg, r7
        L4:
                .
                .
                .

        L1':    mov     r1,   r5
                mov     r4,   r6
        L1:     sub     r3,   r3,   r4
                load    r1,   4[r1]      // ptr->next
                bne     L0,   r1,   0
```

Figure 6: Assembly code of C segment after superblock formation and loop unrolling.

live-out(I3) = {r1, r3, r4}

live-out(I6) = {r2, r3, r7}

live-out(I9) = {r3, r5, r6}

Figure 7: Live-out sets for superblock loop branch instructions.

and r4 have been renamed to r5 and r6 respectively. Note that once the loop has been unrolled and renamed, branch **I9** must branch to **L1'** to restore r1 and r4 before the code at **L1** is executed.[2] Also note that the code within the superblock corresponding to **L0** is placed sequentially in instruction memory. The live-out sets of the three branches within the superblock loop are shown in Figure 7.

Performing dependence analysis on **I1** through **I12** for each code percolation model produces the dependence graphs shown in Figure 8. The data dependences are represented by solid arcs and labeled with **f** for flow and **o** for output (there are no anti dependences). The control dependences

[2]Tail duplication can be recursively applied to form a superblock at label **L1'**.

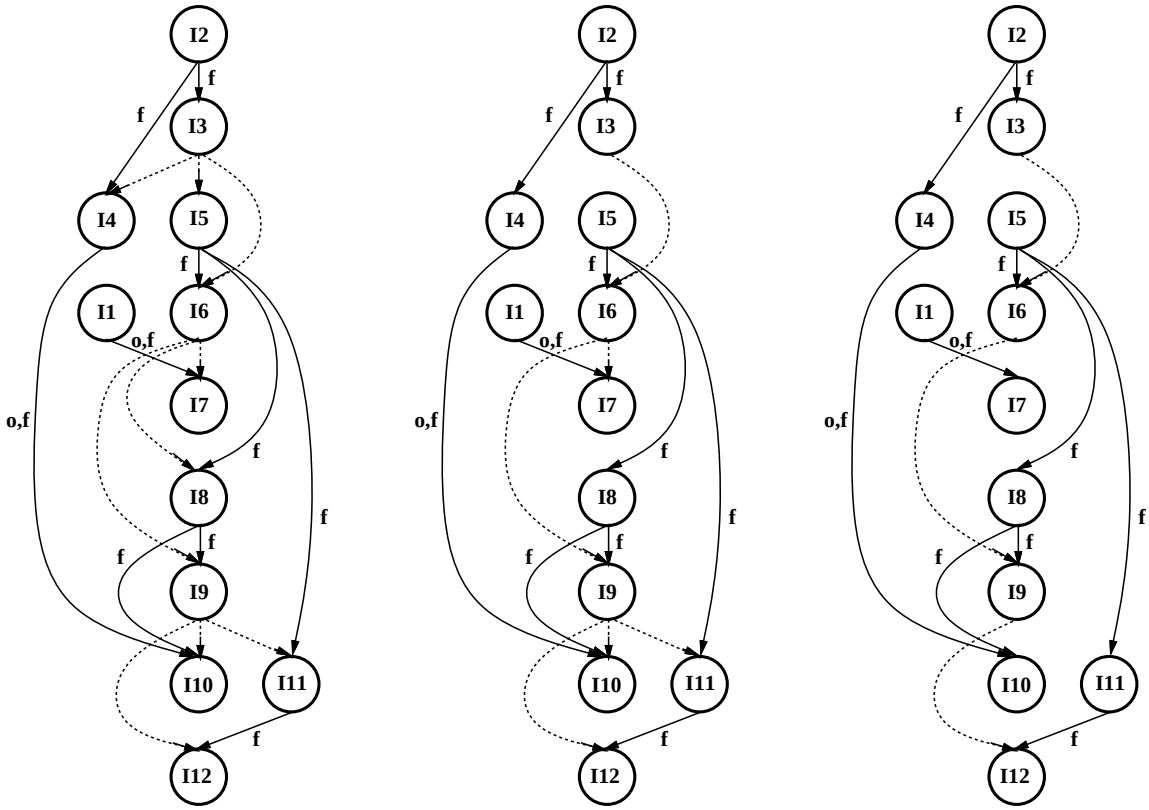

Figure 8: Dependence graphs for the three superblock scheduling models.

are represented by dashed arcs. It is clear from the corresponding number of control dependence arcs in the three graphs that code motion in the restricted code percolation model (9 arcs) is the most limited, then general (6 arcs) and then boosting (3 arcs). In the general code percolation model, control dependence arcs can be removed if the destination of the sink of the arc is not in *live-out(source of the arc)*. In all cases, control dependence arcs between two branch instructions cannot be removed unless the order of the branches does not matter (e.g., in a switch statement). Other than this constraint, all remaining control dependence arcs can be removed in the boosting code percolation model.

The code schedules determined from the graphs in Figure 8 are shown in Figure 9. The actions

Model:	**Restricted**	**General**	**Boosting**
Restrictions:	*1 and 2*	*1*	*none*
Schedule:			
t1:	{I1, I2}	{I1, I2, I5}	{I1, I2, I5}
t2:			{I7}
t3:	{I3, I4, I5}	{I3, I4, I8, I11}	{I3, I4, I8, I11}
t4:		{I6}	{I6}
t5:	{I6, I7, I8}	{I7, I9, I10, I12}	{I9, I10, I12}
t6:			
t7:	{I9, I10, I11}		
t8:			
t9:	{I12}		
Without hardware support:	*executes properly*	*segmentation violation*	*segmentation violation and live value lost*

Figure 9: Code schedules and execution results for the three superblock scheduling models.

that result when the code is executed on processors without additional hardware support are given. The code schedules assume uniform function unit resources with the exception that only one branch can be executed per cycle.[3] The integer ALU instructions have a one cycle latency and the load instructions have a two cycle latency.

For restricted code percolation (both restrictions), the loop takes 9 cycles to execute and the program executes properly without additional hardware support. When only Restriction 1 is observed, general code percolation, load instruction **I5** can be issued in cycle *t1*. This reduces the loop execution time to 5 cycles. Note that since only one branch can be executed per cycle, branch **I6** cannot be issued until cycle *t4*. While this does not affect the code schedule, if there is no additional hardware support, instruction **I8** will cause a segmentation violation by accessing memory through a nil pointer. In the boosting code schedule, there are no restrictions on code motion

[3]This assumption is here in order to illustrate the hazards of removing Restriction 1. In our simulations we do not impose this restriction unless specified.

across branches and thus instruction **I7** can be issued in cycle *t2*. Since r2 is in the live-out set of instruction **I6**, without additional hardware support, **count** will be incremented one too many times and if the program terminated normally, **avg** would be incorrect. Furthermore, as in the case of general code percolation, without hardware support there will be a segmentation violation which will terminate the program. In this example, the schedule using boosting code percolation does not improve upon the schedule achieved from general code percolation.

4 Architecture Support

In this section we discuss the details of the architecture support required by the three scheduling models. Architecture support is required to relax the restrictions on upward code motion. An instruction that is moved above a branch is referred to as a *speculative instruction*. When Restriction 1 is relaxed, a speculative instruction can overwrite a value used on the taken path. Therefore, some form of buffering is required to ensure that the value is not written until the branch direction is determined. To relax Restriction 2, a speculative instruction should not cause an exception if the branch is taken. In addition, when any instruction is moved above a branch and the branch is taken, the instruction may cause an extra page fault. While additional page faults do not alter the program's outcome, they will reduce the program's performance. To avoid extra page faults, an alternative approach is to handle page faults of speculative instructions when the branch commits. The next three sections describe the architecture support needed for each code percolation model. Table 1 provides a summary of the three models.

Table 1: Characteristics of the three scheduling models.

	Restricted	General	Boosting
Scheduling Restrictions	1 and 2	1	none
Hardware Suport	none	non-trapping instructions	shadow register file shadow store buffer support for reexecuting instructions
Exception Handling for Speculative Instructions	prohibited	ignored	supported

4.1 Restricted Code Percolation

The restricted code percolation model assumes that the underlying architecture supports a class of trapping instructions. These typically include floating point instructions, memory access instructions, and the integer divide instruction. These instructions cannot be moved across a branch unless the compiler can prove that their input values will not result in any exceptions that will terminate the program. A non-trapping instruction can be moved across a branch if it does not violate Restriction 1. The majority of existing commercial processors support this model with only minor variations.

The hardware support for handling page faults does not need to be modified to support restricted code percolation. Page faults are handled when they occur. Since memory accesses are not speculatively executed, the only source of additional page faults will be from instruction memory page faults. Since instructions are speculatively executed along the most likely executed path, they will likely be in the working set in memory and thus will not usually cause additional page faults.

4.2 General Code Percolation

The general code percolation model assumes that the trapping instructions in the restricted code percolation model have non-trapping counterparts [18][24]. Our implementation of general code percolation assumes that there are non-trapping versions for integer divide, memory loads, and floating point arithmetic. These instructions can also be moved across a branch if they do not violate Restriction 1. Memory stores are still not percolated above branches for two reasons. First, it is difficult to perform perfect memory disambiguation to ensure that Restriction 1 is not violated. Second, in a load/store architecture, stores are typically not on the critical path and thus will not impact the performance as much as a load or an arithmetic instruction.

There are two types of exceptions, arithmetic and access violation. To implement non-trapping instructions, the function unit in which the exception condition occurs must have hardware to detect whether the instruction is trapping or non-trapping and only raise the exception flag for a trapping instruction. For a non-trapping load instruction, if there is an access violation, the load is aborted. When an exception condition exists for a non-trapping instruction, the value written into the destination register will be garbage. The use of this value is unpredictable, it may eventually cause an exception or it may lead to an incorrect result. Thus, code compiled with general code percolation will not necessarily raise an exception when an exception condition exists.

When an exception condition exists for a speculative instruction and the branch is taken, this condition is ignored as it should be. However, it is also ignored when the branch is not taken. The garbage value returned may eventually cause an exception but there is no guarantee. If the program does not terminate due to an exception, the output will likely be incorrect. Since the program has an error (i.e., an exception condition exists in the original program), it is valid to

produce incorrect output. However, from a debugging point of view, a detectable error has become undetectable, which is undesirable. Therefore, code should first be compiled with restricted code percolation until the code is debugged. Then general code percolation can be turned on to improve the performance. This approach may not be suitable for critical applications such as transaction processing where unreported errors are not acceptable.

Some applications such as concurrent garbage collection rely on trapping instructions to execute properly. For such applications, a compiler flag can be used to prohibit certain instructions from being speculatively executed. Alternatively, additional hardware support can be used to handle exceptions for speculative instructions [25].

As with restricted code percolation, page faults are handled when they occur. No additional hardware beyond traditional hardware support is required to handle page faults. Since memory accesses can be percolated, the number of page faults for the general model may be larger than the number for the restricted model.

4.3 Boosting Code Percolation

Boosting code percolation is based on Smith et. al.'s speculative execution model [17]. Speculative (boosted) instructions which violate Restrictions 1 and 2 can be moved above a branch because no action is committed until the branch commits. The basic architecture support for boosting is shown in Figure 10. This architecture is similar to the TORCH architecture [17]. The shadow register file is required to hold the result of a non-store boosted instruction until the branch commits. The shadow store buffer is required to hold the value of a boosted store instruction until the branch commits. Instructions that are moved above conditional branches are marked as boosted. An instruction can be moved above more than one branch instruction. This would require additional bits to indicate the

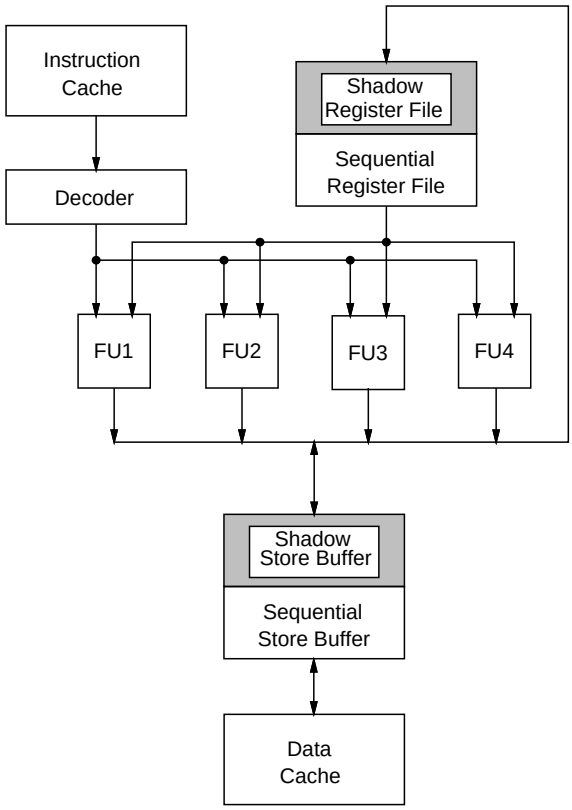

Figure 10: Architecture support for the boosting code percolation model.

number of branches that an instruction has moved across and also additional buffering. However, our experimental results corroborate Smith et. al.'s findings that the largest performance increase is seen for moving instructions across one branch instruction [17][24]. Therefore, this model assumes that instructions are only moved across one conditional branch.[4]

If the boosted instruction finishes before the branch commits, the result is stored in the shadow register file until the branch commits. Since code is scheduled within a superblock, instructions are moved across a branch from the not-taken path. Thus, if the branch is not taken, the values in the shadow register are copied to the sequential register file. However, if the branch is taken,

[4]If multiple branches can be issued in the same cycle, there must be an ordering of branches and hardware to support multiple squashing delay slots. Boosted instructions can be issued with multiple branches provided they are issued in the proper slot.

the values in the shadow register are cleared. Likewise, when the branch commits the values in the shadow write buffer are copied into the write buffer. If the branch is taken then the values in the shadow write buffer are squashed.

Since boosted instructions may still be executing when the branch commits, the execution pipeline must distinguish between boosted and regular instructions. When the branch commits and the branch is not taken, any boosted instructions in the execution pipeline are converted to normal instructions. If the branch is taken then any boosted instructions in the pipeline are squashed (except those in the branch delay slot).

All exception handling for boosted instructions, including page fault handling, is delayed until the branch commits. Page faults could also be handled immediately in this model but the hardware is available to delay page fault handling until the branch commits. When a boosted instruction causes a page fault or exception the condition is stored until the branch commits. If the branch is taken, the exception condition is ignored. Otherwise, the values in the shadow buffers are cleared and the boosted instructions and delay slot instructions (boosted or not) in the execution pipeline are squashed. At this point the processor is in a sequentially consistent state and the boosted instructions are reexecuted sequentially until the exception occurs. To reexecute the boosted instructions, the program counter of the first boosted instruction, *pc_boost*, must be saved.[5]

The instructions can either be reexecuted in software by the exception handling routine or in hardware. In the software scheme, the only additional hardware for exception handling is for the *pc_boost* register. In the hardware scheme, the instruction fetch mechanism must be altered to fetch from *pc_boost* when an exception condition exists when the branch commits. Only instructions that

[5] Alternatively, the program counter of the previous branch plus the delay slot offset can be saved. This avoids hardware required to detect the first boosted instruction after a branch.

are marked as boosted are reexecuted, all others are squashed at the instruction fetch unit. After an exception on a boosted instruction is handled (assuming it does not terminate the program), only boosted instructions are executed until the branch instruction. Then the exception condition is cleared and instruction fetch returns to normal operation.

5 Experiments

The purpose of this study is to analyze the cost-effectiveness of the three scheduling models. In the previous section we analyzed the cost with respect to the amount of hardware support required by each model. In this section we analyze the performance of each model for superscalar and superpipelined processors.

5.1 Methodology

To study the performance of the three scheduling models, each model has been implemented in the superblock scheduler of the IMPACT-I C compiler. The IMPACT-I C Compiler [24] is a retargetable, optimizing compiler designed to generate efficient code for superscalar and superpipelined processors. The performance of code generated by the IMPACT-I C compiler for the MIPS R2000 is slightly better than that of the commercial MIPS C compiler[6] [14]. Therefore, the scheduling results reported in this paper are based on highly optimized code.

The IMPACT-I C compiler uses profile information to form superblocks. The profiler measures the execution count of every basic block and collects branch statistics. A machine description file is used to characterize the target machine. The machine description includes the instruction set,

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

Table 2: Benchmarks.

name	*description*
cccp	GNU C preprocessor
cmp	compare files
compress	compress files
eqn	typeset mathematical formulas for troff
eqntott	boolean minimization
espresso	boolean minimization
grep	string search
lex	lexical analysis program generator
tbl	format tables for troff
wc	word count
yacc	parsing program generator

microarchitecture, and the code percolation model. The microarchitecture is defined by the number and type of instructions that can be issued in a cycle and the instruction latencies.

To evaluate the performance of a code percolation model on a specific target architecture, a benchmark was compiled using the composite profile of 20 different inputs. Using a different input than those used to compile the program, profiling information is used to calculate the best and worst case execution times of each benchmark. The execution time of a benchmark is calculated by multiplying the time to execute each superblock by its weight and adding a mis-predicted branch penalty. The worst case execution time is due to long instruction latencies that protrude from one superblock to another superblock. For the benchmark programs used in this study (Table 1), the difference between the best case and the worst case execution time is always negligible.

5.2 Processor Architecture

The base processor is a pipelined, single-instruction-issue processor that supports the restricted code percolation model with basic block scheduling. Its instruction set is a superset of the MIPS R2000 instruction set with additional branching modes [26]. Table 3 shows the instruction latencies. Instructions are issued in order. Read-after-write hazards are handled by stalling the instruction-

Table 3: 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

unit pipeline. The microarchitecture uses a squashing branch scheme [27] and profile-based branch prediction. Branch prediction is used to layout the superblocks such that the branches are likely not taken. If the branch is taken, the instruction(s) following the branch is squashed. If the branch is predicted taken, the base processor has one branch delay slot. The processor has 64 integer registers and 32 floating-point registers.[7]

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 read-after-write hazard between it and a previous instruction. All the subsequent instructions are also

[7] The code for these benchmarks contains very few floating point instructions.

blocked. All the instructions in the buffer are issued before the next instruction is fetched. The maximum number of instructions that can be decoded and dispatched simultaneously is called the *issue rate*. The superscalar processor also contains multiple function units. In this study, unless otherwise specified, we assume uniform function units where every instruction can be executed from every instruction slot. When the issue rate is greater than one, the number of branch slots increases [27].

The superpipelined version of this processor has deeper pipelining for each function unit. If the number of pipeline stages is increased by a factor **P**, the clock cycle is reduced by approximately the 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 **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 with the degree of pipelining [27].

5.3 Results

In this section we first motivate the need for superblock scheduling and then analyze the relative performance of each of the superblock scheduling models for superscalar and superpipelined architectures. In addition, we characterize the performance of the models for various hardware resource assumptions.

5.3.1 Basic Block vs. Superblock Scheduling

First, we want to verify the need for superblock scheduling. Figure 11 shows that the speedup that can be achieved using basic block scheduling for an 8-issue processor ranges from 1.153 for *eqntott*

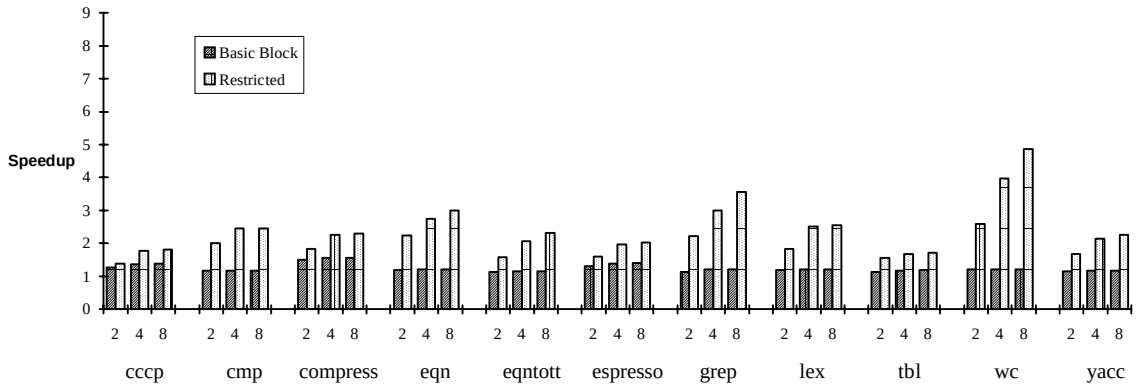

Figure 11: Comparison of Basic Block and Restricted Superblock Scheduling.

to 1.569 for *compress*. Whereas for superblock scheduling using the restricted model, the speedup ranges from 1.724 for *tbl* to 4.869 for *wc* for an issue-8 processor.

5.3.2 Scheduling Superscalar and Superpipelined Processors

Next we want to analyze the performance of the three scheduling models on superscalar and superpipelined processors with uniform function units. Figure 12 shows the speedup of the three scheduling models for a superscalar processor model. The speedup for the general and boosting code percolation models with uniform function units is approximately the same. The most significant speedup for boosting over general is for *grep* which performs 22% better on an issue-4 processor but the same for an issue-8 processor. For an issue-8 processor, the general (boosting) code percolation model performs from 13%(13%) to 144%(146%) better than the restricted code percolation model for *eqntott* and *tbl* respectively.

Figures 13 and 14 show the speedup of the three code percolation models when superpipelining is added. The degree of superpipelining in Figures 13 and 14 is 2 and 4 respectively. The issue rate of the combined superscalar/superpipelined processor ranges from 1 to 4. A pure superpipelined processor corresponds to issue rate 1. The relative performance among the three models remains

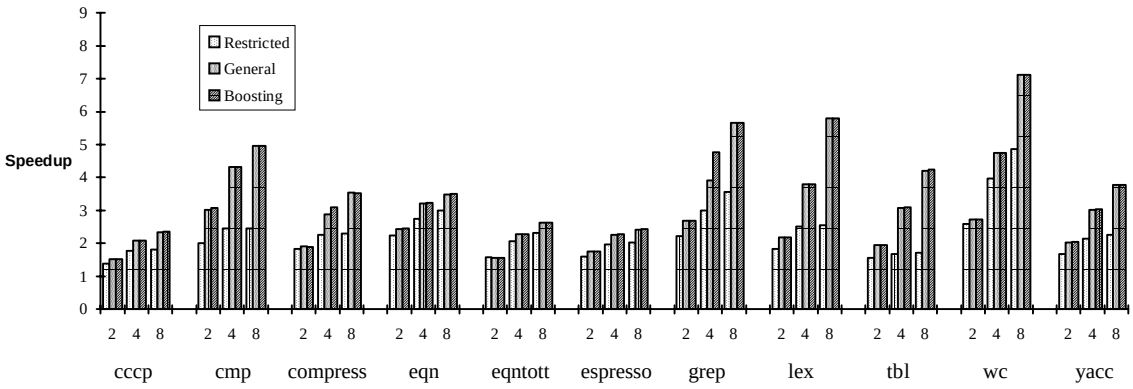

Figure 12: Comparison of scheduling models for a superscalar processor model.

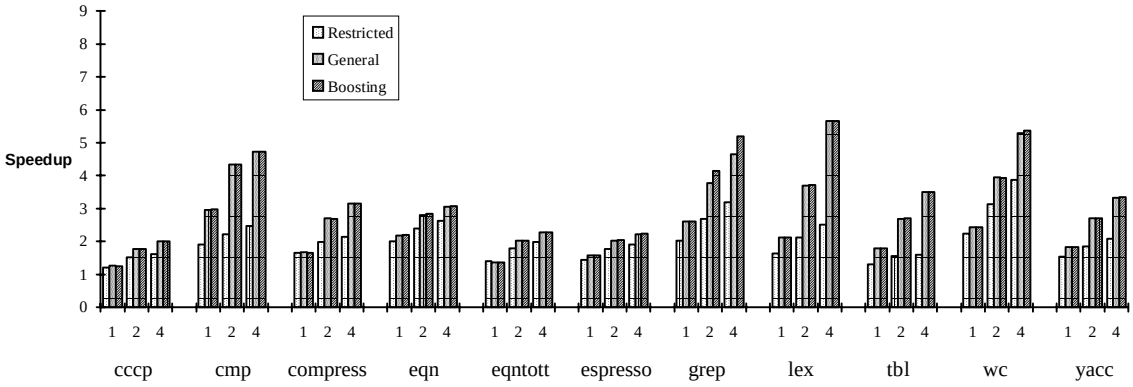

Figure 13: Comparison of scheduling models for a superpipelined processor model with ($\mathbf{P} = 2$).

the same for superpipelined as for superscalar. Comparing the performance of the three models on a superscalar processor for issue rates 2 and 4 (Figure 12) with the performance of the models for the pure superpipelined processors in Figures 13 and 14 it can be seen that all models perform slightly better on the superscalar processors. This is due to the higher branch penalty for superpipelined processors.

Figures 12 - 14 show that the general code percolation model performs almost as well as the boosting code percolation model even though code motion is more restricted. This implies that there are not many cases where code to be moved across a branch is constrained by the live-out set of the branch. This is expected after applying register renaming, loop induction variable

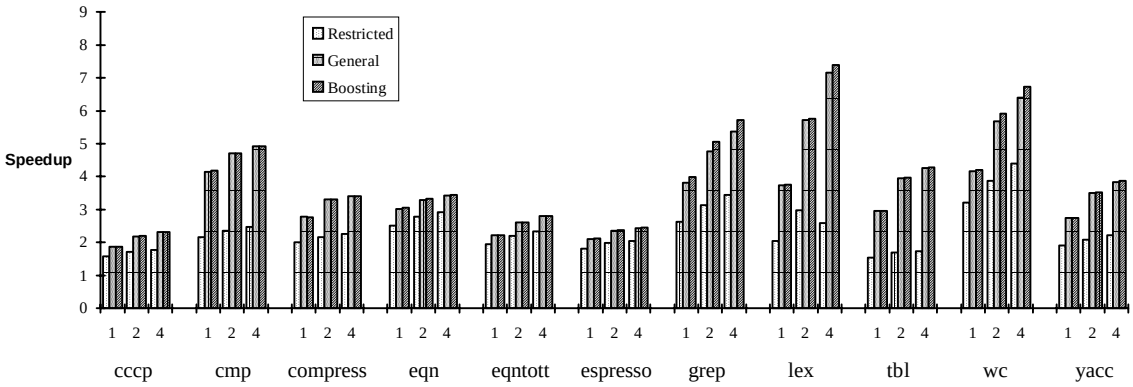

Figure 14: Comparison of scheduling models for a superpipelined processor model with ($\mathbf{P} = 4$).

expansion, and accumulator expansion. Furthermore, since the boosting code percolation model supports speculatively executed stores, these results show that the benefit of moving stores above branches is small. The fact that both the general and boosting models perform considerably better than the restricted code percolation model implies that moving any or all of the following types of instructions: memory loads, integer divide, and floating point arithmetic, greatly reduces the critical path. Since our benchmark set is not floating point intensive and there are usually many more loads than integer divide instructions, these results imply that scheduling loads early has a large impact on the performance. Since the latency of floating point arithmetic is relatively large, scheduling these instructions earlier will also benefit numerical applications.

5.3.3 Scheduling a Superscalar with Non-uniform Function Units

The cost to replicate all function units for each additional instruction slot can be very high. Therefore, we have evaluated the performance degradations due to non-uniform function unit resources. Since the relative behavior of the three scheduling models is the same for both the superscalar and the superpipelined processors, we only analyze the effect of limiting resources for the superscalar processor. Figure 15 shows the speedup of the three scheduling models for a superscalar processor

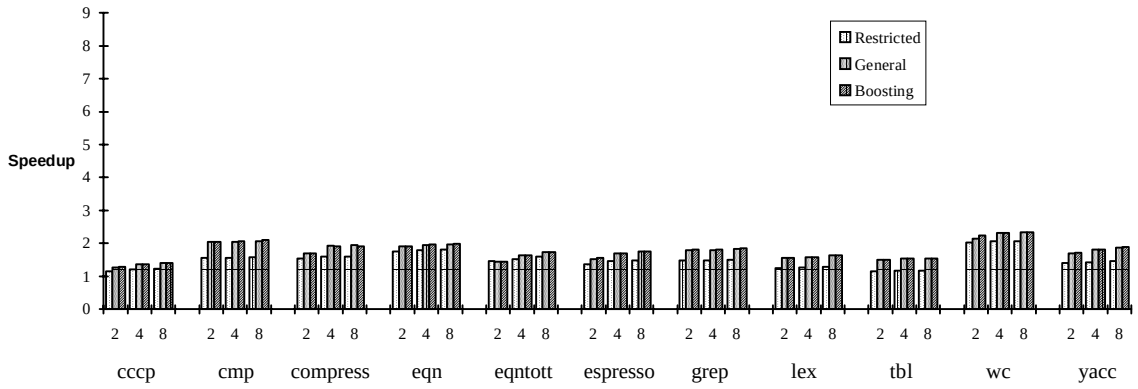

Figure 15: Comparison of scheduling models for a superscalar with limited resources.

where only one memory access, branch, or floating point instruction can be issued per cycle. These resource limitations are overly restrictive for an issue-8 processor. However, it is interesting to compare an issue-4 processor with unlimited resource against the issue-4 processor with limited resources. With unlimited resources, an issue-4 machine using the general or boosting code percolation model performs from 10% to 51% better than one using the restricted code percolation model (for *eqntott* and *lex* respectively). However, with limited resources, an issue-4 machine using the general or boosting code percolation model only performs from 1% to 4% better than one using the restricted code percolation model (for *eqntott* and *lex* respectively). The difference in relative performance indicates that the general and boosting models can take advantage of additional resources better than the restricted model.

5.3.4 Effect of Load Delay

Memory loads are usually on the critical execution path. For single-issue architectures there is a sufficient number of independent instructions available to the scheduler to hide moderate memory load latencies. However, the demand for independent instructions to schedule after a load grows as a multiple of the issue rate and load delay. As a result, for high-issue rate processors, the

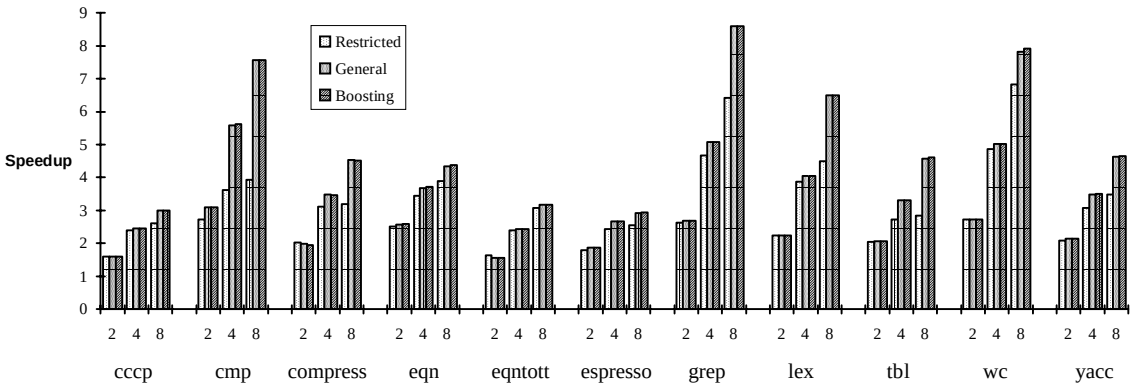

Figure 16: Comparison of scheduling models for a superscalar with load delay 1.

limited supply of independent instructions can no longer hide long memory load latencies. The benefit of reducing the load delay is clearly shown in Figures 16 and 17 which show the speedup for superscalar processors with load delays of 1 and 3 respectively.

Another interesting point is that the relative performance of the restricted code percolation compared to boosting and general code percolation increases when the load delay is decreased. When the load delay is decreased from 3 to 2 for an 8-issue processor, the speedup for general and boosting code percolation increases from 12% for *lex* to 37% for *grep* while the speedup for restricted code percolation increases from 20% for *espresso* to 44% for *grep*. Likewise, when the load delay is decreased from 2 to 1 for an 8-issue processor, the speedup for general and boosting code percolation increases from 8% for *tbl* to 53% for *cmp* while the speedup for restricted code percolation increases from 25% for *espresso* and 80% for *grep*. This is expected since loads cannot be moved across branches in the restricted model and thus are more likely to be on the critical path than in the general and boosting models. Therefore, restricted code percolation is more sensitive to increasing the memory access delay.

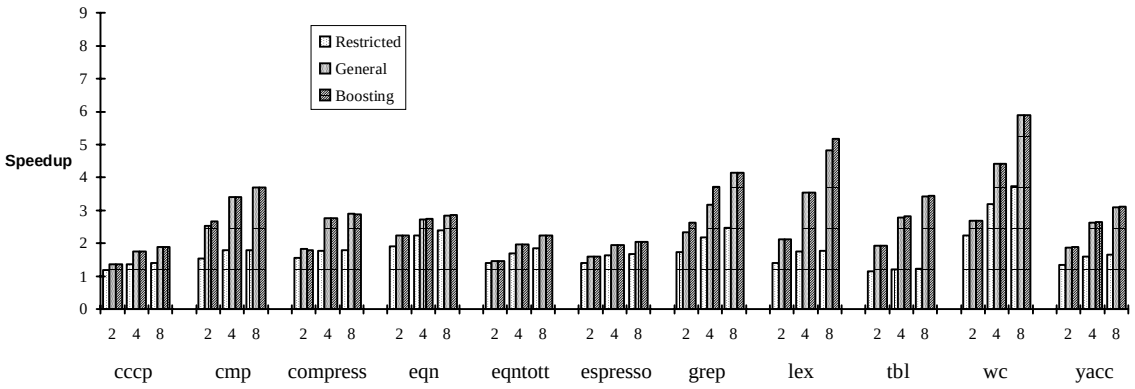

Figure 17: Comparison of scheduling models for a superscalar with load delay 3.

5.3.5 Scheduling a Superscalar with 8K Data Cache

In the previous experiments we have assumed an ideal instruction and data cache. To analyze the effect of the data cache, which typically has a higher miss ratio than the instruction cache, we replaced the ideal data cache with an 8K direct mapped data cache with 32 byte blocks. An 8K data cache was chosen to represent moderate sized on-chip caches in the near future. Therefore, for the range from moderate to large data cache sizes, the performance impact due to cache misses is bounded by the speedup shown in Figure 18 and those in Figure 12. We assume that the processor stalls on a cache miss. The initial delay to memory is 4 cycles and the transfer size is 32 bits. For an 8 issue processor, Figure 18 shows that the effect of the data cache misses effectively decreases the speedup of boosting and general from 50% for *compress* to approximately 0% for *eqntott* and of restricted code percolation from 34% for *compress* to approximately 0% for *eqntott*. As expected, the performance of the data cache has a greater impact on the more aggressive scheduling models.

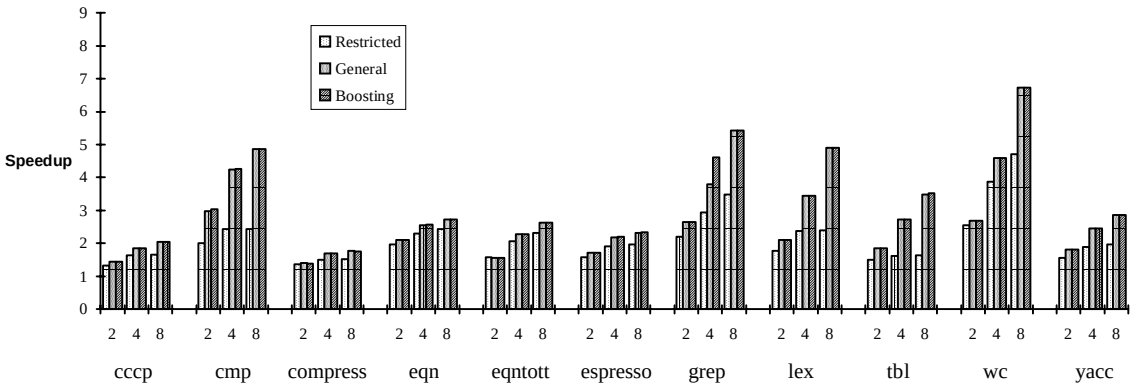

Figure 18: Comparison of scheduling models for a superscalar with 8K data cache.

6 Application to Other Global Code Scheduling Techniques

In this paper we have analyzed three scheduling models with varying degrees of architecture support for speculative execution. We have shown that general code percolation is an effective model for superblock scheduling. Other global instruction scheduling techniques such as those based on the Program Dependence Graph [15] and Aiken and Nicolau's percolation scheduling [28] can also benefit from general code percolation. However, when an instruction is speculatively executed, some form of branch prediction is needed [12][29][30]. Without branch prediction, a speculative instruction may be moved from a less frequently executed path to a more frequently executed path and possibly *increase* the program execution time.

While trace scheduling uses branch frequencies to determine the scheduling region, it allows side entrances into traces [12]. During scheduling, these side entrances require incremental bookkeeping to perform code duplication when an operation is moved upward across a side entrance. Similarly, percolation scheduling [29] and global instruction scheduling based on the Program Dependence Graph [15] apply incremental code duplication when an operation is moved across a program

merge point.[8] Superblocks eliminate the need for incremental bookkeeping by performing tail duplication to remove side entrances to the trace. In addition to simplifying scheduling, separating superblock formation from scheduling allows the compiler to apply superblock optimizations. These optimizations increase the size of the superblock and remove dependences in order to increase the instruction level parallelism [21].

7 Conclusion

In this paper we have analyzed three code percolation for superscalar and superpipelined processors. We have shown that increasing the scheduling scope from basic block to superblock increases the available parallelism. There is enough parallelism within a superblock to achieve up to 300% speedup over basic block scheduling for an issue-8 uniform function unit superscalar processor with a restricted code percolation model. Within a superblock there can be many conditional branch instructions. To efficiently schedule the code, instructions must be moved from below to above a conditional branch on the sequential path. However, there is the danger that these speculatively executed instructions may have adverse side-effects when the branch is taken. Thus, restrictions must be placed on code motion to ensure that the program executes properly. The three scheduling models for moving code across branches: restricted code percolation, general code percolation, and boosting code percolation, use varying degrees of hardware support to remove the restrictions on code motion.

Restricted code percolation assumes traditional trapping instructions (e.g., integer divide, memory access, floating point arithmetic). The non-trapping instructions can be moved across a branch

[8] A program merge point refers to a basic block with multiple predecessors.

if they do not write over any values along alternate execution path of the branch. General code percolation supports both trapping and non-trapping versions for memory loads, integer divide, and floating point arithmetic. The non-trapping versions are used when these instructions are speculatively executed, to guarantee that they do not cause an exception that terminates the program incorrectly. Thus, it requires a larger subset of non-trapping instructions and minimal support to detect a non-trapping instruction to prevent raising the exception condition flag when the instruction terminates. No extra hardware support is required to support page fault handling in either the restricted or general code percolation model. Boosting uses a shadow register file to hold the results of instructions that have been moved across a conditional branch until that branch commits. An extra bit is required per instruction to indicate that the instruction has been moved across a branch. In addition, extra hardware is required to control the execution pipeline, shadow register file and shadow store buffer when a branch commits. To handle precise exceptions and page faults the program counter of the first instruction to be move across a branch must be saved.

The boosting code percolation model is the least restrictive; however, it also requires the most hardware support. In this paper, we analyzed the speedup of all three models on superscalar and superpipelined processors. On average, the boosting code percolation model performs slightly better than general code percolation. Both the boosting and general code percolation models perform considerably better (between 13% and 145% for an issue-8 processor) than restricted code percolation. Similar trends have been shown for processors with varying resource assumptions.

We believe that future processor instruction sets should support some form of the general code percolation model in order to be competitive in the superscalar and superpipelining domain. Superblock scheduling and other global code scheduling techniques can exploit the general code percolation model. We hope to see future research and engineering work in the direction of making

general code percolation an extended part of existing architectures and an integral part of future processor architectures.

Acknowledgements

The authors would like to thank John Holm, Bob Horst at Tandem, Andy Glew at Intel, Roland Ouelette at DEC, James Smith at CRAY Research and all members of the IMPACT research group for their support, comments, and suggestions. This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Intel Corporation, the AMD 29K Advanced Processor Development Division, Hewlett-Packard, SUN Microsystems, NCR 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).

References

[1] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. c-21, pp. 1405–1411, December 1972.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[3] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[4] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, January 1984.

[5] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[6] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[7] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[8] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.

[9] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105–109, October 1987.

[10] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[11] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[12] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[13] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[14] 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, December 1991.

[15] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[16] W. W. Hwu and Y. N. Patt, "Hpsm, a high performance restricted data flow architecture having minimal functio nality," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 297–306, June 1986.

[17] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[18] 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, April 1987.

[19] 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.

[20] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[21] 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 structure for VLIW and superscalar compilation," *Journal of Supercomputing*, February 1993.

[22] K. Anantha and F. Long, "Code compaction for parallel architectures," *Software Practice and Experience*, vol. 20, pp. 537–554, June 1990.

[23] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[24] 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 International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[25] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. S. ansker, "Sentinel scheduling for VLIW and superscalar processors," in *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

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

[27] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," Tech. Rep. CSG-123, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1990.

[28] A. Aiken and A. Nicolau, "A development environment for horizontal microcode," *IEEE Transactions on Software Engineering*, vol. 14, pp. 584–594, May 1988.

[29] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[30] M. C. Golumbic and V. Rainish, "Instruction scheduling beyond basic blocks," *IBM Journal of Research and Development*, vol. 34, pp. 93–97, January 1990.