Three Superblock Scheduling Models for Superscalar and Superpipelined Processors

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

Center for Reliable and High-Performance Computing

University of Illinois

Urbana, IL 61801

`hwu@crhc.uiuc.edu`

Abstract

To efficiently schedule superscalar and superpipelined processors, it is necessary to move instructions across branches. This requires increasing the scheduling scope beyond the basic block. Superblock scheduling, a static scheduling method, is a variant of trace scheduling that removes the bookkeeping complexity associated with branches into a trace by removing these entrances using a method called *tail duplication.* Once the scheduling scope is enlarged, there are hazards to moving an instruction above a conditional branch because the instruction is normally only executed on one path of the conditional branch. To allow the compiler to schedule code more aggressively, hardware support can be provided to prevent such hazards. In this paper we analyze the architecture support and performance of three superblock scheduling models.

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

1 Introduction

For non-numeric programs, there is insufficient instruction level parallelism available within a basic block to exploit superscalar and superpipelined processors[20] [21] . 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 either use hardware branch prediction[17] to schedule instructions from the likely executed path or schedule instructions from both paths of a conditional branch[19]. 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[11]. 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 [18] [23] [16]. For control intensive code, profiling provides accurate branch prediction [13]. Once the direction of the branch is determined, blocks which tend to execute together can be grouped to form a trace[9] [3]. To reduce some of the bookkeeping complexity, the side entrances to the trace can be removed to form a superblock [5].

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 across branches. An instruction that is moved above a conditional branch should not cause an exception which terminates the program or incorrectly overwrites 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 [12] [21] [22]. 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 [8].

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 hardware 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[9] 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 dependencies that are introduced by register assignment [10] [6]. Postpass code scheduling is performed after register

```
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.

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 [9]. 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** to **BB3** in

```
(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.

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 bookkepping is required to ensure correct program execution [9]. 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**, then **i12** must also be copied into **BB3** and the branch instruction **i10** 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* [5]. 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

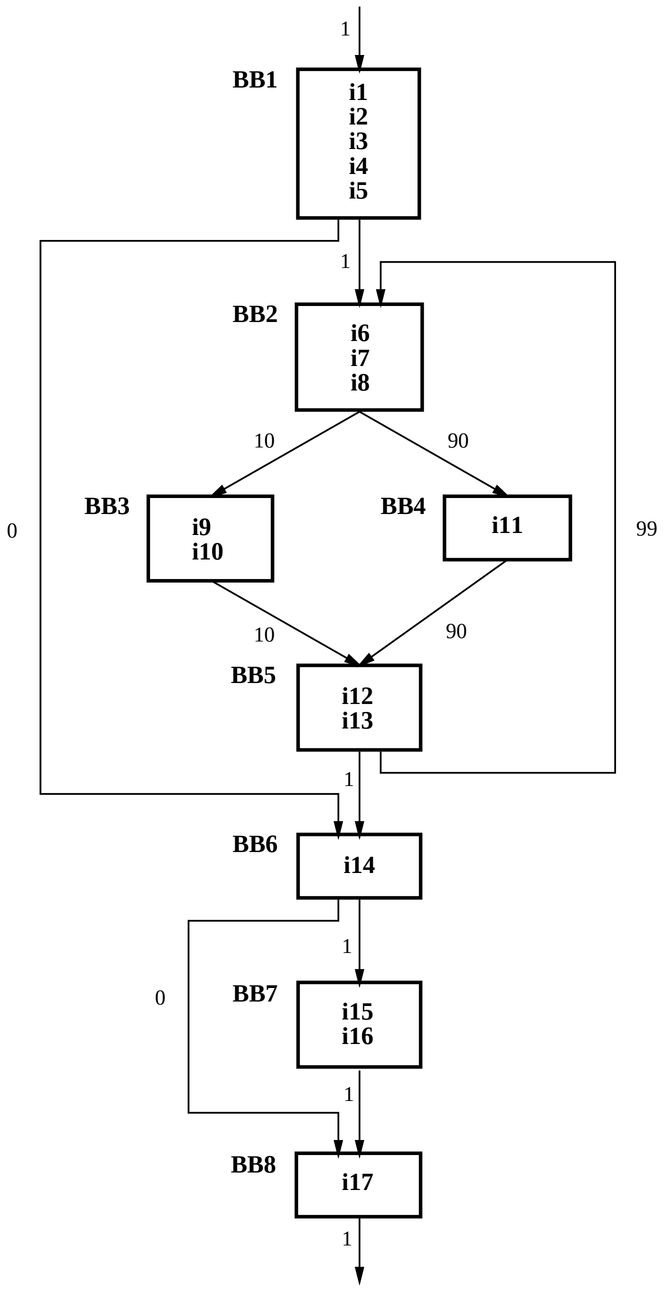

Figure 3: Weighted control flow graph.

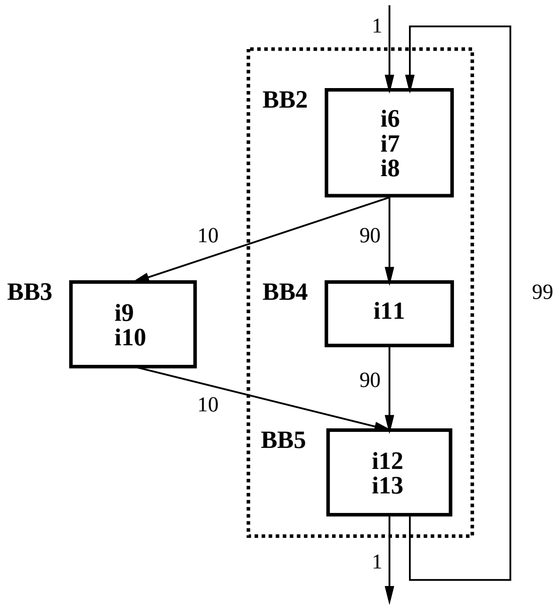

Figure 4: Loop portion of control flow graph after trace selection.

superblock. Figure 5 shows the loop portion of the control flow graph after superblock formation 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 [2] 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.

[1]Note that the profile information is scaled during tail duplication. This reduces the accuracy of the profile information. For an accurate analysis of the final schedule, the transformed program must be reprofiled.

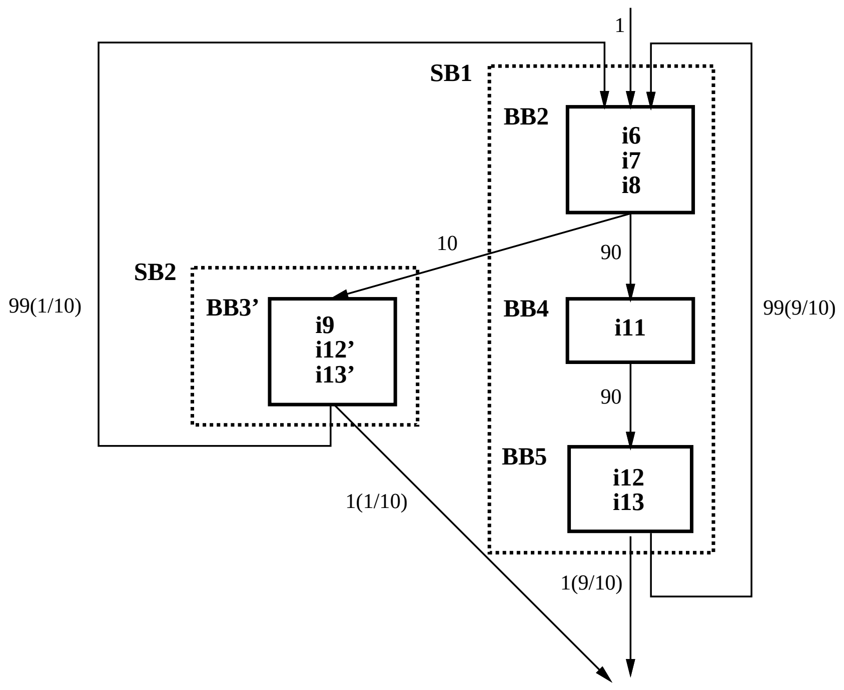

Figure 5: Loop portion of control flow graph after superblock formation and branch expansion.

After superblock formation many classic code optimizations are performed that take advantage of the profile information encoded in the superblock structure and clean up the code after the above transformations. These optimizations include the local and global versions of: constant propagation, copy propagation, common subexpression elimination, redundant load and store elimination, dead code removal, branch expansion and constant folding [1]. 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 [5].

The third step in the superblock scheduling algorithm is to build a dependence graph. The dependence graph represents the data and control dependencies between instructions. There are three types of data dependencies, flow, anti, and output. A flow dependence between two instructions i and j indicates that an instruction j reads a value that instruction i writes and instruction i executes before instruction j. This is also known as a read-after-write dependence. Likewise, anti and output dependencies refer to write-after-read and write-after-write dependencies respectively. Control dependencies 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[6]. 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 dependencies are reduced, a more efficient code schedule can be found. Of the data dependencies, only the flow dependencies are *true* dependencies. Output and anti dependencies are formed when registers are reused. Hardware or software renaming can be used to remove these dependencies. Control dependencies can also be removed by adding hardware support. If a control dependency is removed, the corresponding instruction can be moved across the branch. Three superblock scheduling models with varying 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(I), 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. For upward code motion, moving instruction I from below to above branch Br, there are two major restrictions.

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

Restriction 2: *I* must not cause an exception that may terminate the program execution.

Three superblock scheduling models: *restricted code percolation*, *general code percolation*, and *speculative 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 speculative code percolation model [22], 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 loop superblock have been labeled. In the unrolled iteration, registers r1 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 loop superblock are shown in Figure 7.

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

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

```
                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 loop superblock branch instructions.

dependencies 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 (6 arcs) is the most limited, then general (4 arcs) and then speculative (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 speculative code percolation model.

The code schedules determined from the graphs in Figure 8 are shown in Figure 9. The actions 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 speculative code schedule, there are no restrictions on code motion across branches and thus instruction **I7** can be issued in cycle *t2*. Since r2 is in the live-out set

[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.

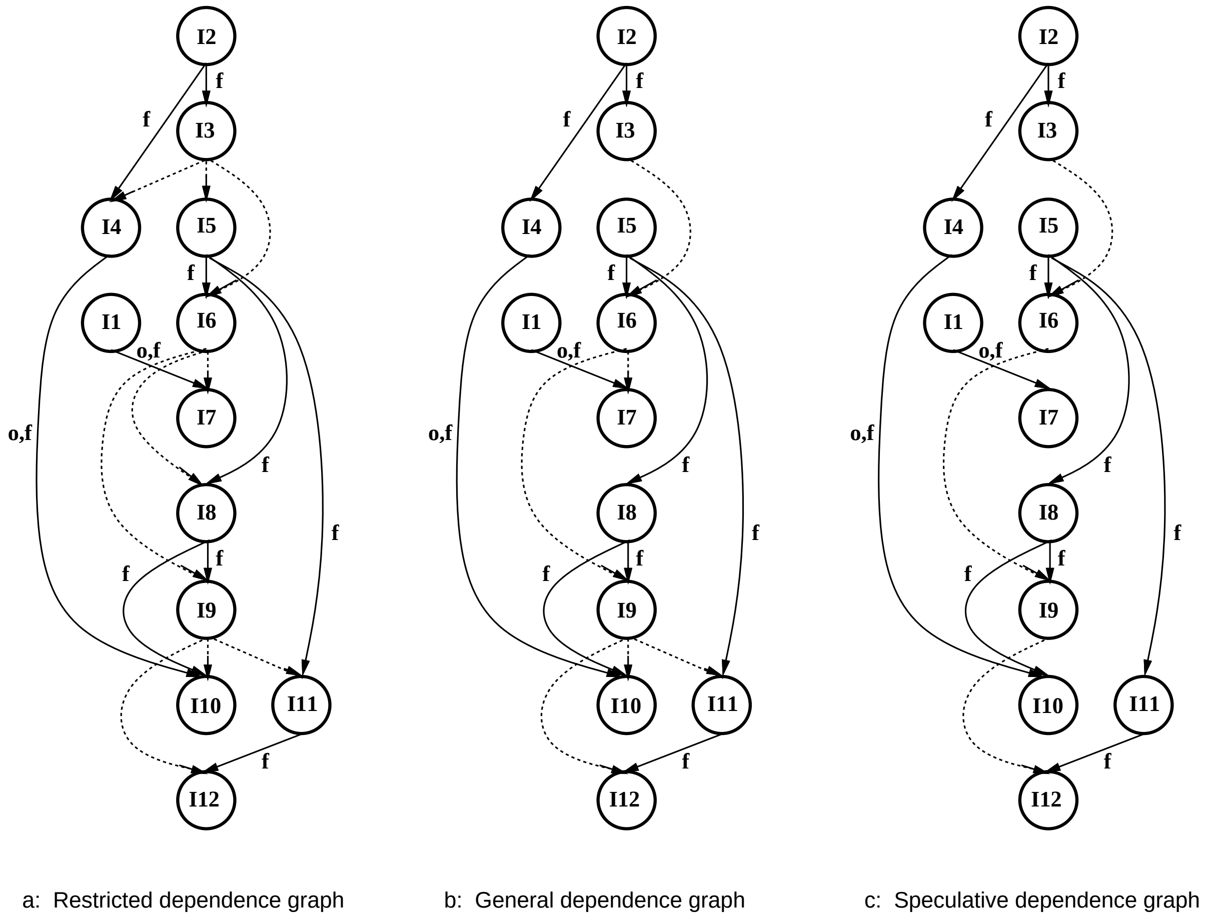
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I12
f
o,f
a: Restricted dependence graph
b: General dependence graph
c: Speculative dependence graph

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

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

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

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. However, 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 speculative 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 superblock 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 *boosted instruction* [22]. When Restriction 1 is relaxed, a boosted 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 boosted 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 boosted instructions when the branch commits. The next three sections describe the architectural support needed for each code scheduling model.

4.1 Restricted Code Percolation

The restricted code percolation model assumes that the underlying architecture supports the following types of trapping instructions: integer divide, memory access, and floating point arithmetic. Therefore, these instructions can never be moved across a branch since they may cause an exception 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 boosted, the only source of additional page faults will be from instruction memory page faults. Instructions are boosted from the most likely executed path, therefore these instructions will likely be in the working set in memory and thus will not usually cause 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 [8]. Specifically, general code percolation assumes that there are non-trapping versions for integer divide, memory loads, and floating point arithmetic. Thus, 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 not executed. 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.

As with restricted code percolation, page faults are handled when they occur. If there is an access violation, the page fault is ignored. 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 Speculative Code Percolation

Speculative code percolation is similar to Smith et. al.'s speculative execution model [22]. Instructions which violate Restrictions 1 and 2 can be moved above a branch because no action is committed until the branch commits. A buffer such as the shadow register file in [22] is required to store the results 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 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 [22] [4]. Therefore, this model assumes that instructions are only moved across one conditional branch. [4]

When a branch commits, a boosted instruction has either finished execution or is still executing. 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, the values in the shadow register are cleared. If stores are boosted then a shadow write buffer similar to the one in [22] can be used. 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. Our experiments have shown that boosting stores does not improve the performance significantly and thus in our model a shadow write buffer is not required.

[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.

Since boosted instructions may 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 register 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 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

[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.

is cleared and instruction fetch returns to normal operation.

5 Side Effects of General Code Percolation

Code compiled with general code percolation will not necessarily raise an exception when an exception condition exists. When an exception condition exists for a boosted 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. However, the output will likely be incorrect. Since the program has an error (i.e., an exception condition exists), 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.

6 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.

6.1 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 [4] 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] [5]. Therefore, the scheduling results reported in this paper are based on highly optimized code.

The IMPACT-I C compiler uses profiling 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, microarchitecture, and the scheduling 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 scheduling 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, the best and worst case execution times of each benchmark are derived. The worst case 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. The results presented in Section 6.3 are the harmonic mean of the speedup numbers of all benchmarks using the worst case execution time to calculate the speedup.

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

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

Table 1: Benchmarks.

6.1.1 Processor Architecture

In this study, we analyzed the speedup of the three superblock scheduling models for superscalar and superpipelined processors. 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 [15]. Table 2 shows the instruction latencies. Instructions are issued in order. Read-after-write hazards are handled by stalling the instruction-unit pipeline. The microarchitecture uses a squashing branch scheme [14] and profile-based branch prediction. For the base processor, one branch slot is allocated by the compiler for each predicted-taken branch. 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-

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

Table 2: 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

after-write hazard between it and a previous instruction. All the subsequent instructions are also 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, every instruction can be executed from every instruction slot. When the issue rate is greater than one, the number of branch slots increases [14].

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 [14].

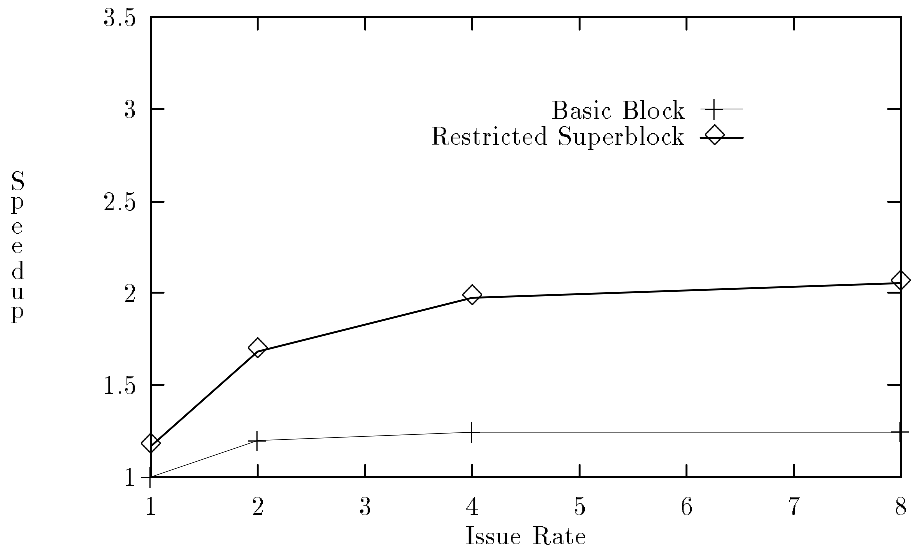

Figure 10: Comparison of Basic Block and Restricted Superblock Scheduling.

6.2 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.

6.2.1 Basic Block vs. Superblock Scheduling

First, we want to verify the need for superblock scheduling. Figure 10 shows that the most speedup that can be achieved using basic block scheduling is approximately 1.25. Whereas for superblock scheduling using the restricted model, a speedup larger than 2 can be achieved for an issue 8 processor with uniform function units.

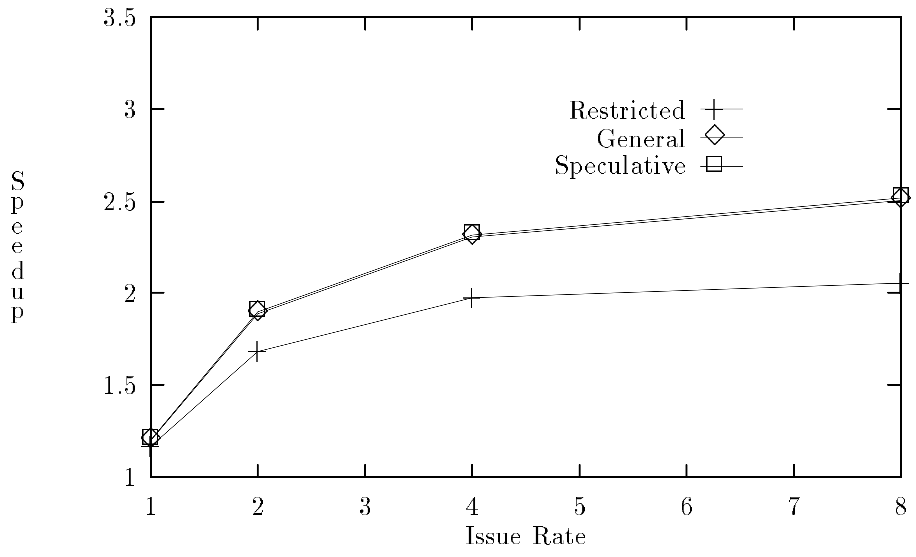

Figure 11: Comparison of scheduling models for a superscalar processor model.

6.2.2 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. Thus, the type of instructions that can be issued in a cycle is not limited by hardware. Figure 11 shows the speedup of the three scheduling models for a superscalar processor model. The speedup for the general and speculative code percolation models with uniform function units is approximately the same. For an issue 8 processor, these two models perform approximately 22% better than the restricted code percolation model.

Figures 12 and 13 show the speedup of the three scheduling models when superpipelining is added. The degree of superpipelining in Figures 12 and 13 is 2 and 4 respectively. The issue rate of the combined superscalar/superpipelined processor ranges from 1 to 8. A pure superpipelined processor corresponds to issue rate 1. The relative performance among the three models remains the same for superpipelined as for superscalar. Comparing the performance of the three models

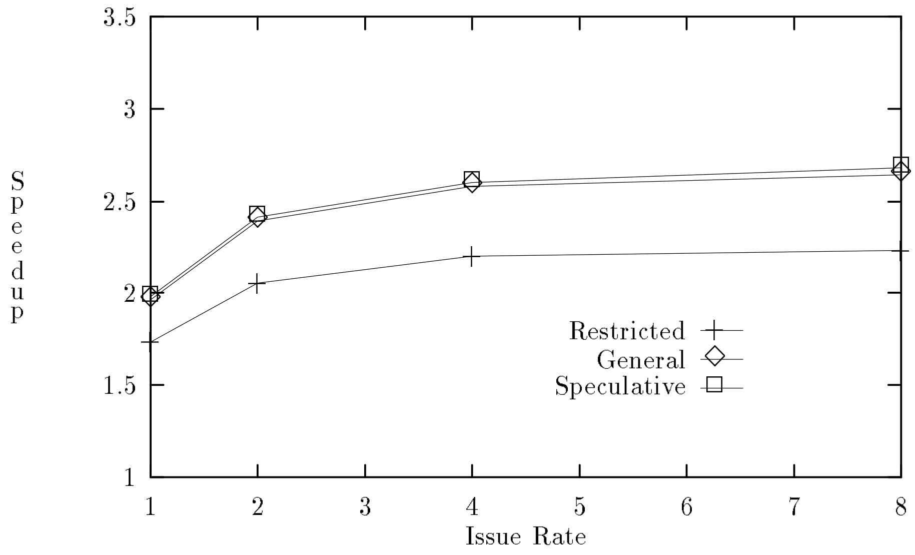

Figure 12: Comparison of scheduling models for a superpipelined processor model with (**P** = 2).

on a superscalar processor for issue rates 2 and 4 (Figure 11) with the performance of the models for the pure superpipelined processors in Figures 12 and 13 it can be seen that all models perform slightly better on the pure superpipelined processors.

Relative to the pure superscalar processor, the processors in Figures 12 and 13 can respectively execute two and four times as many instructions in a given time period. As the number of instructions executed per cycle increases, the difference between the speedup for each of the scheduling models increases. However, the speedup for all three scheduling models starts to level off for the processor with degree of superpipelining equal to 4 and issue rate 8. At this point speculative performs slightly better than general (1.1%) and much better than restrictive (28%).

Figures 11 - 13 show that the general code percolation model performs almost as well as the speculative code percolation model even though code motion is more restricted. This implies that there are not many cases where code moved across a branch is in the live-out set of the branch,

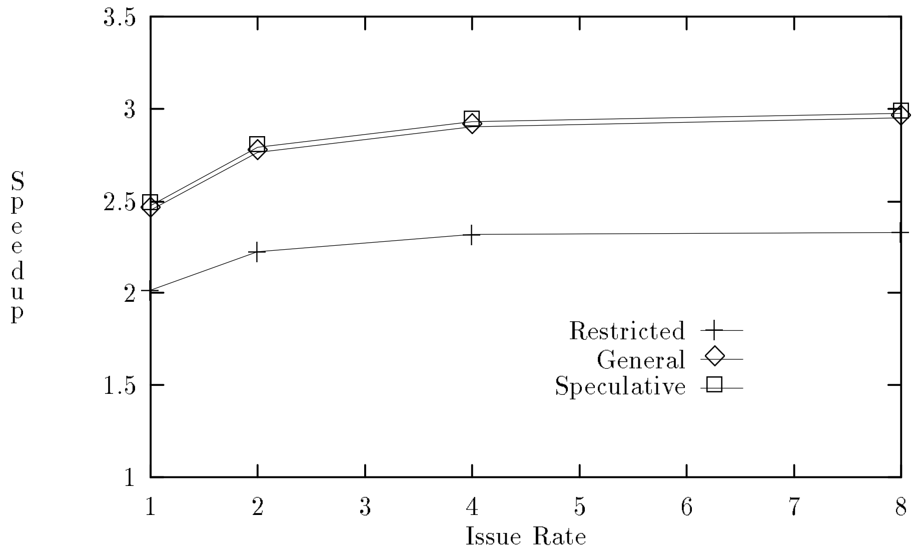

Figure 13: Comparison of scheduling models for a superpipelined processor model with ($\mathbf{P} = 4$).

which is expected after the application of register renaming, loop induction variable expansion, and accumulator expansion. The fact that both the general and speculative 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.

6.2.3 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.

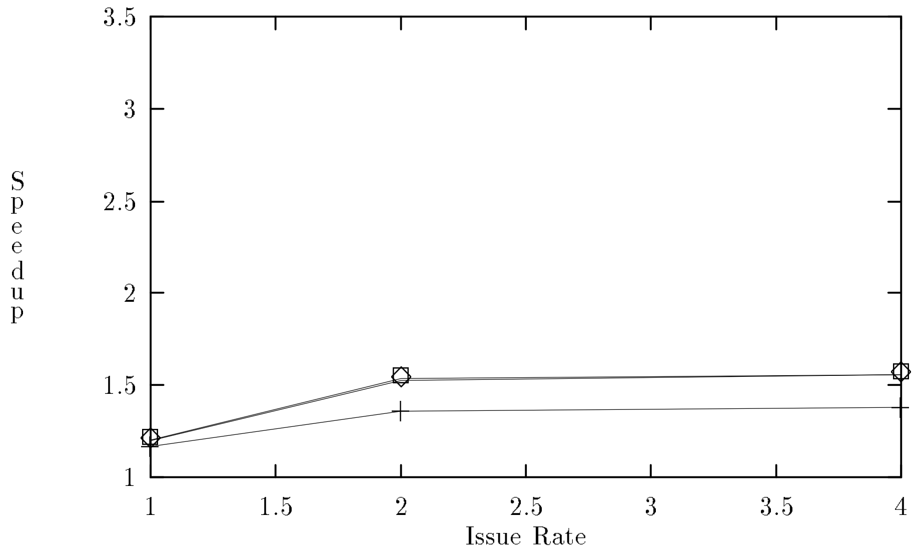

Figure 14: Comparison of scheduling models for a superscalar with limited resources.

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 14 shows the speedup of the three scheduling models for a superscalar processor with one cache port, one integer ALU, one FPU, and branch logic to issue one branch per cycle. Since there are only four function units, there is no speedup beyond issue rate 4. For an issue 4 machine with the general and speculative code percolation models, there is a performance drop of approximately 34% with these resource limitations. Whereas for an issue 4 machine with the restricted code percolation model there is approximately a 29% performance drop with the same resource limitations. The difference in the performance drops indicates that the general and speculative models can take advantage of additional resources better than the restricted model.

6.2.4 Effect of Load Delay

Because it is not possible to perform perfect memory disambiguation, memory references often cannot be determined until run time. Thus, 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 limited supply of independent instructions can no longer hide a high memory load latency. The benefit of reducing the load delay is clearly shown in Figures 15 and 16 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 speculative and general code percolation increases when the load delay is decreased. The difference between general and speculative remain small (0.6%) as the load delay decreases. In other words, when the load delay is decreased from 3 to 2, the speedup for general and speculative code percolation increases by 22% while the speedup for restricted code percolation increases by 27%. Likewise, when the load delay is decreased from 2 to 1, the speedup for general and speculative code percolation increases by 27% while the speedup for restricted code percolation increases by 37%. 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 speculative models. Therefore, restricted code percolation is more sensitive to increasing the memory access delay.

6.2.5 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

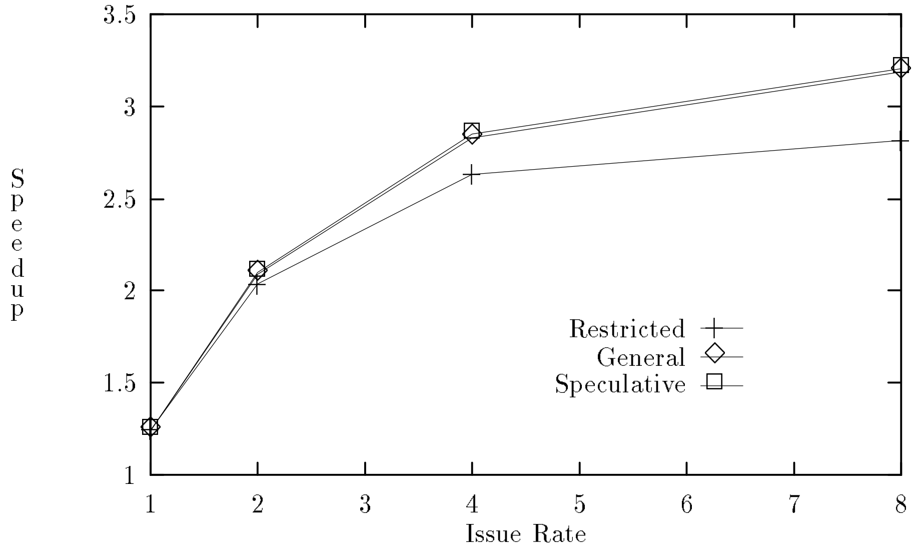

Figure 15: Comparison of scheduling models for a superscalar with load delay 1.

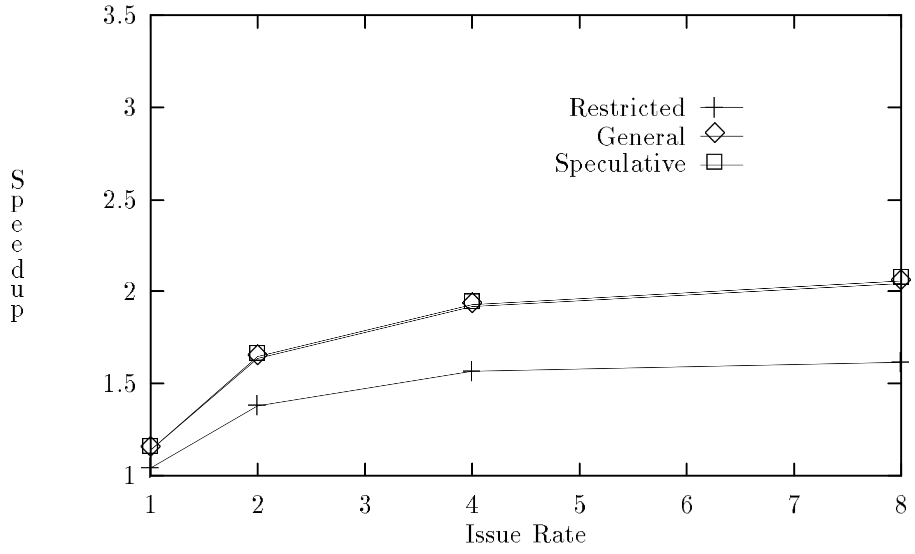

Figure 16: Comparison of scheduling models for a superscalar with load delay 3.

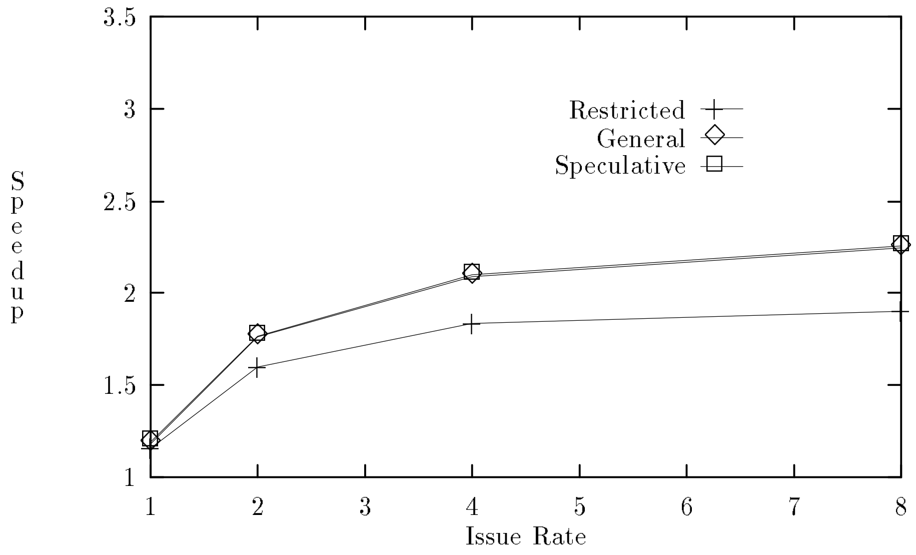

Figure 17: Comparison of scheduling models for a superscalar with 8K data cache.

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 curves shown in Figure 17 and those in Figure 11. 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 17 shows that the effect of the data cache misses effectively decreases the speedup of speculative and general by 10% and of restricted code percolation by 7%. As expected, the performance of the data cache has a greater impact on the more aggressive scheduling models.

7 Conclusion

In this paper we have analyzed three superblock code scheduling models 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 almost 200% speedup for an issue 4 uniform function unit superscalar processor with a restricted code scheduling 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 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 code scheduling models for moving code across branches: restricted code percolation, general code percolation, and speculative code percolation, use varying degrees of hardware support to remove the restrictions on code motion.

Restricted code percolation assumes traditional trapping instructions (integer divide, memory access, floating point arithmetic). The non-trapping instructions can be moved across a branch 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 percolated, 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. A garbage value is returned when there is an exception on an instruction that was moved across a branch. No extra hardware support is required to support page fault handling in either the restricted or general code percolation model. Speculative execution 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 and shadow register file 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 speculative code percolation model is the least restrictive, however, it is also requires the most hardware support. In this paper, we analyzed the speedup of all three models on superscalar and superpipelined processors. Both the speculative and general code percolation models perform considerably better (approximately 22%) than restrictive code percolation. Speculative performs slightly better (approximately 1%) than general code percolation. Similar trends have been shown for processors with varying resource assumptions. Therefore, general code percolation is an efficient scheduling model since it has good performance with low hardware overhead.

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-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, 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), and the Office of Naval Research under Contract N00014-88-K-0656.

References

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

[2] K. Anantha and F. Long, “Code Compaction for Parallel Architectures”, Software Practice & Experience, vol.20, no.6, pp.537-554, June, 1990.

[3] P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode", Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures, pp.21-29, San Diego, California, November, 1988.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N.J. Warter, and W. W. Hwu, "IMPACT: An Archtectural Framework for Multiple-Instruction-Issue Processors," Proceedings of the 18th Annual Symposium on Computer Architecture, pp.266-275, May, 1991.

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

[6] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," Center for Reliable and High-Performance Computing Technical Report, University of Illinois at Urbana-Champaign, May, 1991.

[8] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler", Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California, October, 1987.

[9] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction", IEEE Transactions on Computers, vol.c-30, no.7, July 1981.

[10] J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks", Proceedings of the 1988 International Conference on Supercomputing, St. Malo, July, 1988.

[11] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing", Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 386-395, June, 1986.

[12] W. W. Hwu and Y. N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality", The 13th International Symposium on Computer Architecture Conference Proceedings, pp. 297-306, June, 1986.

[13] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing Software and Hardware Schemes For Reducing the Cost of Branches", Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, May, 1989.

[14] W. W. Hwu and P. P. Chang, "Efficient Instruction Sequencing with Inline Target Insertion", Coordinated Science Laboratory Report, UILU-ENG-90-2215, CSG-123, May, 1990.

[15] G. Kane, MIPS R2000 RISC Architecture, Prentice Hall, Englewood Cliffs, NJ, 1987.

[16] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", Proceedings, ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, pp.318-327, Atlanta, Georgia, June, 1988.

[17] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, January, 1984.

[18] B.R. Rau and C.D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", Proceedings of the 14th Annual Workshop on Microprogramming, pp.183-198, October, 1981.

[19] B. Rau, D. Yen, W. Yen, and R.A. Towle, "The Cydra 5 departmental supercomputer", Computer, vol.22, pp.12-35, January, 1989.

[20] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", *IEEE Transactions on Computers*, vol. C-21, no. 12, December 1972, pp. 1405-1411.

[21] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instruction Issue", Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.

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

[23] S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1987.