There are four main areas in which we see opportunities for future work. The first is in measuring the effect of hardware extensions to our current machine model for supporting unsafe code boosting. The second is implementing a software mechanism similar to those proposed by Bernstein et al. [10] for proving the safety of speculative loads and measuring its impact on performance. The third is augmenting our scheduler to allow weak ordering of memory references by performing memory reference disambiguation [4, 32, 13, 5]. The fourth is in integrating a mechanism for register reallocation and spill code insertion into our scheduling framework (for related work, see [6]). Some of these efforts are currently underway.

Acknowledgments

This research was supported by NSF Grant No. MIP-9007678 and by an NSF Graduate Fellowship.

References

[1] R.D. Acosta, J. Kilestrup, H.C. Torng. An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors. *IEEE Trans. on Computers.* C35(9):815-828, Sep, 1986.

[2] A.V. Aho, R. Sethi, J.D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[3] A. Aiken, A. Nicolau. *Perfect Pipelining: A New Loop Parallelization Technique.* TR 87-873, Cornell University, October, 1987.

[4] R. Allen, K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. on Prog. Lang. and Syst.* 9(3):319-349, July, 1987.

[5] U. Banerjee, S. Chen, D. Kuck, R. Towle. Time and Parallel Processor Bounds for Fortran-Like Loops. *IEEE Trans. on Computers.* C-28(9), Sept., 1979.

[6] D.G. Bradlee, S.J. Eggers, R.R. Henry. Integrating register allocation and instruction scheduling for RISCs. *ASPLOS IV,* pp. 122-131. Santa Clara, CA, April, 1991.

[7] F.C. Chow, J.L. Hennessy. Register allocation by priority-based coloring. *ACM SIGPLAN '84 Symposium on Compiler Construction,* pp. 222-232. June, 1984.

[8] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, R.K. Rodman. A VLIW Architecture For a Trace Scheduling Compiler. *ASPLOS II*, pp. 180-192. Palo Alto, CA, October, 1987.

[9] D. Bernstein, M. Rodeh. Global instruction scheduling for superscalar machines. *ACM SIGPLAN '91 on Programming Language Design and Implementation*, pp. 241-255. Toronto, Ontario, Canada, June, 1991.

[10] D. Bernstein, M. Rodeh, M. Sagiv. Proving safety of speculative load instructions at compile time. *ESOP 92, 4th Europe Symposium on Programming,* pp. 344-354. February, 1992.

[11] J. Ferrante, K.J. Ottenstein, J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. on Programming Languages and Systems.* 9(3):319-349, July, 1987.

[12] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers.* C-30(7):478-490, July, 1981.

[13] G. Goff, K. Kennedy, C. Tseng. Practical Dependence Testing. *ACM SIGPLAN 91, Conf. on Programming Language Design and Implementation*. June, 1991.

[14] G.F. Grohoski. Machine Organization of the IBM RISC System/6000 processor. *IBM Journal of Res and Development.* 34(1):37-58, January, 1990.

[15] T.R. Gross, J.L. Hennessy. Optimizing delayed branches. *15th Annual Workshop on Microprogramming*, pp. 114-120. Palo Alto, CA, October, 1982.

[16] J.L Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[17] M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] M.S. Lam, R.P. Wilson. Limits of Control Flow on Parallelism. *19th ISCA*, pp. 46-57. 1992.

[19] N.P. Jouppi, D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *ASPLOS III*, pp. 272-282. Boston, MA, April, 1989.

[20] A. Nicolau. *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report 85-678, Cornell University, May, 1985.

[21] R. R. Oehler, R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Res. & Development.* 34(1):23-36, January, 1990.

[22] A. Rogers, K. Li. Software support for speculative loads. *ASPLOS V,* pp. 38-50. Boston, MA, Oct., 1992.

[23] J.E. Smith, A.R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. on Computers.* 37(5):562-573, May, 1988.

[24] M.D. Smith, M.A. Horowitz, M.S. Lam. Efficient superscalar performance through boosting. *ASPLOS V,* pp. 248-259. Boston, MA, October, 1992.

[25] M.D. Smith, M. Lam, M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. *17th ISCA*, pp. 344-354. May, 1990.

[26] M.D. Smith, M. Johnson, M.A. Horowitz. Limits on multiple instruction issue. *ASPLOS III*, pp. 290-302. Boston, MA, April, 1989.

[27] G.S. Sohi, S. Vajapeyam. Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors. *14th ISCA*, pp. 27-34. 1987.

[28] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, J. Shen. Instruction Level Profiling and Evaluation of the IBM RS/6000. *18th ISCA*, pp. 180-189. 1991.

[29] R.F. Touzeau. A Fortran Compiler For the FPS-164 Scientific Computer. *ACM SIGPLAN '84 on Compiler Construction*, pp. 48-57. June, 1984.

[30] D.W. Wall. Limits of instruction-level parallelism. *ASPLOS IV,* pp. 176-188. Santa Clara, CA, April, 1991.

[31] H.S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Res. & Development.* 34(1):85-92, Jan., 1990.

[32] M. Wolfe, U. Banerjee. Data Dependence and its Applications to Parallel Processing. *International J. of Parallel Programming.* 16(2):137-8, 1987.

[33] M. Wolfe. *Optimizing Supercompilers For Supercomputers.* MIT Press, Cambridge, MA, 1990.

of the two types of functional units, FXU and MEM functional units, for executing ALU instructions and load/store instructions, respectively. Since so far we have experimented with integer-intensive benchmarks, the number of FPU's is not considered. For machine model M2, the instruction fetch bandwidth is six instructions per cycle, instead of four instructions per cycle, and the instruction dispatch bandwidth is increased to four instructions per cycle, as opposed to two instructions per cycle. The performance improvement over the original machine, M0, is given in Table 5 for all of the benchmarks. Clearly, significant performance improvements are obtained with M1 and M2 over M0 due to the additional machine parallelism. Additional performance increases of up to 1.86% and 7.16% are found for boosting across two conditional branches on machine models M1 and M2, respectively.

TABLE 5. Performance Improvement due to AC Boosting

	Performance Improvement with respect to M0			
	Scheduled for M1, Executed on M1		Scheduled for M2, Executed on M2	
Benchmark	No Boosting	Across 2 Branches	No Boosting	Across 2 Branches
compress	47.37 %	+0.94 %	67.84 %	+4.07 %
eightq	24.19 %	+0.00 %	41.51 %	+0.00 %
grope	22.38 %	+0.03 %	34.76 %	+7.16 %
qslarge	23.04 %	+0.07 %	29.13 %	+0.07 %
queen	25.28 %	+1.86 %	44.14 %	+1.24 %
quick	17.47 %	+0.00 %	19.08 %	+0.00 %
tree	26.45 %	+0.27 %	36.36 %	+0.93 %
xlisp	19.48 %	+0.00 %	37.99 %	+0.00 %
AVERAGE	25.71 %	+0.40 %	38.84 %	+1.68 %

While these performance improvements are not impressive, it is clear that some benchmarks benefit significantly from AC boosting. Nonetheless, the concept of AC boosting is shown to be feasible on realistic machines. There is room for improvement on some of the scheduler heuristics, and further experimentation on more benchmarks is needed. Unlike the data presented in the original boosting proposal [25], the above data does not include the boosting of any load instructions. In the next subsection, we explore this possibility.

4.4. AC Boosting of (Potentially Unsafe) Loads

Unsafe code motion allows load instructions to be moved across conditional branches. Doing so exposes a greater amount of program parallelism by increasing the number of potential candidates for upward code motion. Table 6 compares the performance improvement between safe and unsafe code motion with respect to the performance of the original machine, M0. The potential of having more instructions for code motion increases the possibility of the scheduler producing overly aggressive code. For those benchmarks that show additional performance improvement of unsafe code motion over safe motion, up to 28.23% and 29.74% gains are observed for boosting across two and five branches, respectively. These results indicate that restricting speculative code motion to only safe instructions can severely limit the amount of instruction-level parallelism that can be exploited. The results clearly indicate that speculative boosting of loads can result in significant performance gain for some benchmarks.

TABLE 6. Performance Improvement due to AC Boosting of Load Instructions

	Performance Improvement with respect to M0			
	Scheduled for M2, Executed on M2			
	Across 2 Branches		Across 5 Branches	
Benchmark	Safe	Gain	Safe	Gain
compress	71.92 %	+4.49 %	71.92 %	+4.49 %
eightq	41.51 %	+28.23 %	41.51 %	+29.74 %
grope	41.93 %	+2.00 %	41.93 %	+2.00 %
qslarge	29.20 %	+0.14 %	29.20 %	+0.14 %
queen	45.38 %	+7.25 %	45.38 %	+11.60 %
quick	19.08 %	+1.51 %	19.08 %	+1.51 %
tree	37.29 %	+1.24 %	37.29 %	+1.24 %
xlisp	37.99 %	+5.01 %	37.99 %	+5.18 %
AVERAGE	40.54 %	+6.23 %	40.54 %	+6.99 %

5. Summary and Conclusion

Our results indicate that Architecture-Compatible (AC) Boosting is a feasible and effective technique to improve the performance of superscalar processors. AC boosting has been implemented on the RS/6000 without requiring any additional hardware support nor modifying the instruction set architecture. Although no significant performance improvement is found on the current implementation of the RS/6000, performance increase of up to 7.16% is attained by performing AC boosting across two conditional branches on an extended, but realistic, implementation of the RS/6000.

By permitting unsafe code motion, AC boosting across two conditional branches can achieve up to 28.23% performance increase. This result shows that confinement to only safe code motion limits the amount of instruction-level parallelism that can be exploited. Unsafe code motion however requires hardware support to handle possible exceptions at an incorrect point of the control flow. Further experimentation is needed to determine the viability of an architecture-compatible, software-based approach to this problem. This is similar to the approach proposed by Bernstein et al. in [10], in which they indicate that a significant number of loads can be proven to be non-excepting at compile time and, therefore, can be safely boosted without hardware support. A combination of hardware and software techniques may prove to be the most effective approach, however, as indicated by previous results [24, 25, 22]

As processor designs become more superscalar and superpipelined, the under-utilization of the processor functional units is likely to increase. As resource contention is reduced with more machine parallelism, the sources for stalls shift from structural dependences to data dependences and control dependences. With an implementation that has more machine parallelism, more instructions must be found to cover the penalty due to control dependences. We believe that AC boosting will be more effective in these implementations because more opportunities to boost instructions and to improve performance are available.

TABLE 1. Benchmark Set Description

Benchmark	Description
compress	data compression (SPEC benchmark)
eightq	eight queen (recursive version)
grope	version of grep
qslarge	quicksort (iterative version)
queen	eight queen (Stanford benchmark)
quick	quicksort (Stanford benchmark)
tree	treesort (Stanford benchmark)
xlisp	xlisp (SPEC benchmark)

have a greater number of taken conditional branches while others have more not-taken conditional branches.

4.2. Performance of Current RS/6000 Machine Model

The baseline machine, M0, models the current implementation of the RS/6000. The instruction fetch bandwidth is four instructions per cycle. The dispatch policy allows condition code and branch instructions that do not have data dependences to be executed in the dispatch stage within the ICU. The ICU transfers all other instructions, at most two per cycle, to the buffer servicing the FXU and the FPU. The buffer stores at most four instructions. When a buffer is full, a stall occurs and lasts until the buffer has room to accept more instructions. When they are free, the two functional units, FXU and FPU, retrieves instructions from the buffer that do not have data dependences nor structural dependences and execute them.

Table 2 shows the performance of the machine model M0 running the unmodified benchmark set. The instruction traces span from twenty-thousand to over seven million instructions, depending on the benchmark. The instruction count for the entire benchmark set is over twenty million instructions. The performance, in terms of the average number of instructions issued per cycle (IPC), varies from 0.76 to 1.18, with a mean of 1.02. Table 3 exhibits the distribution of control, data, and structural hazards of the benchmark set. Six of the seven benchmarks have structural hazards as the most dominant of the three types of hazards. This result indicates that performance is limited most severely by resource contention, namely, the FXU functional unit. Table 4 presents the distribution of branch instructions with respect to their branch latencies. The latency of a branch is defined as the number of extra cycles needed to resolve the branch condition once a branch is decoded in the ICU. A zero-latency branch resolves the direction that the branch takes in the same cycle that the branch is decoded in the ICU. The number of cycles to resolve a branch on the RS/6000 is typically three. A branch latency greater than three represents a branch that depends on a condition code setting instruction, which in turn also depends on another instruction. To reduce the effects of control dependences is equivalent to minimizing the total number of cycles of all branch latencies. One way of achieving that is to reduce the branch latency to zero cycle for as many branches as possible. In the benchmark set, *quick* has the lowest percentage of zero-latency instructions, while *grope* has the highest. In other words, *grope* has the least potential for performance improvement.

TABLE 2. Performance on Current RS/6000 Model

Benchmark	Instruction Count (unmodified)	Cycle Count (unmodified)	Average IPC
compress	7,121,836	7,070,747	1.01
eightq	20,862	21,820	0.96
grope	6,525,825	5,544,277	1.18
qslarge	1,000,252	1,159,338	0.86
queen	972,537	972,287	1.00
quick	792,205	1,035,690	0.76
tree	1,272,596	1,533,912	0.83
xlisp	3,168,297	3,098,972	1.02
TOTAL	20,874,410	20,437,043	1.02

TABLE 3. Distribution of Control, Data, Structural Hazards

Benchmark	Total Number of Hazards	Ratio of Control Hazards to All Hazards	Ratio of Data Hazards to All Hazards	Ratio of Structural Hazards to All Hazards
compress	7,793,818	19.88 %	13.55 %	66.57 %
eightq	23,151	28.84 %	16.24 %	54.92 %
grope	6,703,479	48.74 %	33.90 %	17.36 %
qslarge	1,286,087	44.51 %	8.73 %	46.76 %
queen	1,049,542	29.63 %	9.21 %	61.16 %
quick	1,111,920	36.19 %	22.84 %	40.98 %
tree	1,743,038	34.11 %	23.89 %	42.00 %
xlisp	4,161,551	16.37 %	42.49 %	41.47 %
AVERAGE	2,984,073	32.28 %	20.36 %	46.40 %

TABLE 4. Distribution of Latencies in Resolving Branches

Benchmark	Branch Latencies (in cycles)				
	0	1	2	3	4 - 6
compress	44.05 %	28.31 %	9.80 %	~0.00 %	17.90 %
eightq	46.31 %	0.00 %	22.10 %	29.60 %	1.97 %
grope	81.50 %	0.03 %	4.57 %	0.27 %	13.63 %
qslarge	44.59 %	14.26 %	~0.00 %	0.32 %	40.82 %
queen	44.22 %	19.02 %	7.81 %	19.37 %	9.59 %
quick	43.26 %	4.24 %	7.35 %	2.45 %	42.70 %
tree	56.50 %	1.52 %	6.06 %	1.51 %	34.40 %
xlisp	51.26 %	6.82 %	8.36 %	14.28 %	19.28 %
AVERAGE	51.46 %	9.28 %	8.26 %	8.48 %	22.54 %

4.3. AC Boosting on Current & Extended Machine Models

To perform AC boosting effectively requires the scheduler to make good decisions on which instructions to boost. Boosting an instruction without having a functional unit to execute it can degrade performance by interfering and possibly lengthening the critical path of the original code. In order to avoid this undesirable effect, the scheduler maintains a record of the functional unit utilization as it schedules instructions within each fragment.

Three machine models are used in our experiments. First, AC boosting is performed for the machine model, M0, with boosting distance of crossing up to two conditional branches. For most cases, little or no performance increase is found; in the other cases, slight performance degradation is observed. The reason is that the machine model M0 does not have enough functional units to deal with the resource requirements. Hence, two machine models with additional functional units are introduced, as shown in Figure 7. The machine model M1 has one additional FXU, while the machine model M2 has two of each

instructions in the guest ready set. Once an instruction has been scheduled for each available functional unit (or, in the absence of available instructions, the functional unit has stalled), the scheduler increments its cycle count and proceeds to reconstruct the ready sets for scheduling the next cycle.

The final schedule in Figure 5 illustrates how the scheduler boosted two instructions from the A1 guest basic block to cover the latency between the `cmpi` and `bc` instructions; the second of these also had its destination renamed to `r6` in order to prevent it from overwriting `r4`, which must be preserved in case the branch falls through.

3.3. Retargetable Trace-driven Pipeline Simulator

The simulator takes as input an instruction trace, models the execution of the instructions, and produces as output the performance data. All resources that are of interest within the processor are represented at the register transfer level. As an instruction flows through the pipeline in the absence of structural, data, and control hazards, the simulator records the resource utilization trace: the resources that an instruction uses in each cycle during its execution. Performance data is obtained by accumulating the resource utilization trace over the entire instruction trace.

3.3.1. Retargetability

The simulator consists of two components: a knowledge base and an engine. The description of the structure/organization of a processor is stored in the knowledge base, while the behavior of the processor is coded into the engine. For example, the size of the instruction dispatch buffer is stored in the knowledge base, and the dispatch policy is coded in the engine as a C function that describes the criteria that must be satisfied for an instruction to be removed from the dispatch buffer and considered to have been issued. The simulator is capable of modeling different implementations of the RS/6000 architecture. In particular, the instruction fetch bandwidth, the instruction dispatch bandwidth, the number and type of functional units, and the number of read and write ports are all easily reconfigurable by simply changing the parameters in the knowledge base.

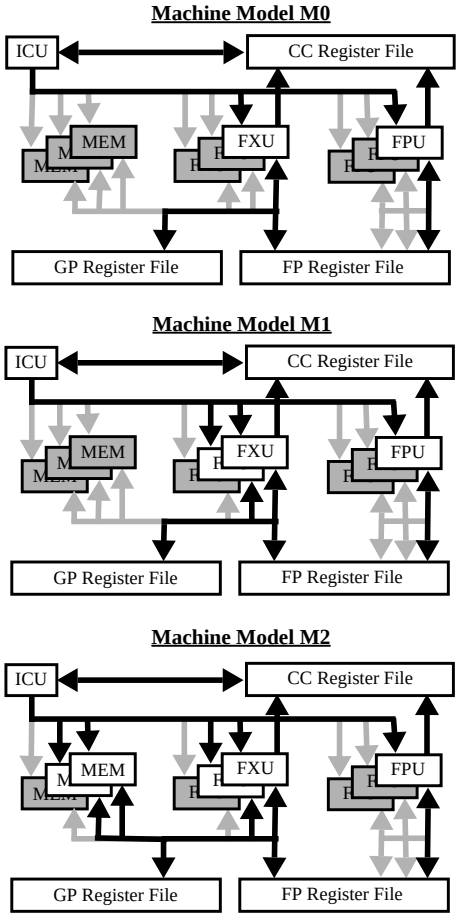
3.3.2. Simplifying Assumptions Used

The simulator does not model any memory hierarchy effects. It assumes that all loads and stores take one cycle to access memory, as in the case of a cache hit. Some instructions, such as string instructions, have operands that depend on the data in the registers, and their result latency varies with respect to the operand values. The simulator assumes the result latency for such instructions as fixed, regardless of the value of the operands.

3.4. RS/6000 Machine Models

The machine model used by the scheduler abstracts the structure of the implementation of the IBM RS/6000 architecture presented in [21] and [14]. This implementation incorporates an instruction cache unit (ICU), which handles instruction fetching and dispatching, as well as the execution of branch and condition code instructions, a fixed point unit (FXU), which executes fixed point and all load/store instructions; a floating point unit (FPU), which executes floating point instructions; and disjoint condition code (CC), general purpose (GP), and floating point (FP) register files. The scheduler's machine model can be configured to extend the existing implementation by including one or more FXUs, FPUs, and zero or more load/store units (MEMs), which are dedicated to executing load/store instructions. Figure 7 summarizes the machine models used in this paper. The machine model M0 represents the current implementation of the RS/6000.

FIGURE 7. Configurable Machine Models



4. Experimental Results

4.1. Benchmark Set

The benchmark set used to evaluate the feasibility and the effectiveness of AC boosting consists of eight benchmarks, as shown in Table 1. All of the benchmarks are integer-intensive, written in C, compiled with optimizations turned on, and run to completion. The focus is on benchmarks that are branch-intensive but not loop-intensive; no floating-point benchmarks were used. The dynamic branch behavior characteristics of the benchmark set is diverse. The percentage of branch instructions encountered with respect to all instructions executed is between 15.45% and 35.28%. The average run length, or the number of instructions executed before finding a branch, is between 3 and 7 instructions. Some benchmarks

forms control and dataflow analysis on it. This information, along with branch and execution statistics generated by a profiled version of the image, are fed to the scheduler. The disassembly, control flow analysis, and profiling are all performed by an enhanced version of the *goblin* profiling system [28].

The scheduler is designed to take advantage of profiling information, not only for predicting the outcomes of conditional branches and boosting from likely successor basic blocks, but also for favoring those portions of the code executed most often by scheduling them first. This approach gives the best possible execution schedule for those portions that contribute the most to the dynamic cycle count of the program. We expect the importance of profile-guided scheduling to increase as we augment our scheduler to support live-range splitting [7] and spill-code insertion, since the scheduler will be able to include execution-frequency statistics into its spill code cost/priority functions [7].

3.2.1. Scheduling Framework

For reasons of computational efficiency, the scheduling framework divides the program image into fragments. A *fragment* is defined as a sequence of basic blocks that has one or more external entry points (i.e. subroutine call destinations) followed by one or more exit points (subroutine return statements). Each fragment is processed in turn through the following sequence of steps:

1. Compute the control flow and control dependence information for the fragment.
2. Compute register dataflow and def-use chains for the fragment.
3. Order the basic blocks in the fragment by their execution frequency.
4. Select the basic block with the highest execution frequency and add it to the scheduling scope as the host basic block.
5. Find the most likely successor basic block, and add it to the scheduling scope as a guest.
6. Apply Step 5 recursively, until the number of conditional branches crossed matches that specified by the user.
7. Construct a dependence graph for the scheduling scope.
8. Schedule instructions in the scope, subject to the dependence graph, until all instructions in the host basic block have been scheduled. Allow upward motion of instructions from successor basic blocks to the host by performing the actions specified for preserving semantic correctness.
9. Repeat from Step 4 until all basic blocks in the fragment have been scheduled.

The number of guest basic blocks is indirectly determined by the number of conditional branches that the user specifies should be crossed.

3.2.2. List Scheduling Algorithm

The list scheduling algorithm employed by our scheduler is fairly straightforward. As the host and guest basic blocks are added to the scheduling scope, the instructions in them are scanned for dependences, which are then added to the dependence graph. Our dependence graph contains three types of edges; RAW edges, which represent read-after-write, or true dependences between operations; WAW edges, which represent write-after-write, or output dependences; and MISC edges, which are introduced in the graph to impose miscellaneous ordering constraints. These are used to constrain the scheduler to keep all memory accesses strongly ordered, to prevent upward code motion of exception-causing instructions, and to prevent downward code motion. Figure 6 shows an example dependence graph, one that corresponds to the scheduling example in Figure 5. Note that the RAW edges are

FIGURE 5. Example of Code Scheduling

Original Code

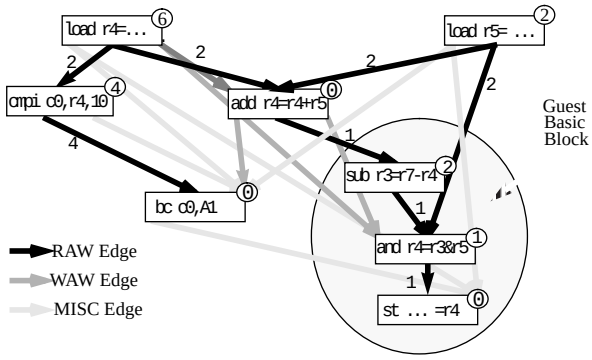
```
        load   r4= ...
        load   r5= ...
        cmpi   c0,r4,10
        add    r4=r4+r5
        bc     c0, A1
        st     ... =r4
A1:     sub    r3=r7-r4
        and    r4=r3&r5
        st     ... =r4
```

Scheduled Code

```
        load   r4= ...
        load   r5= ...
        cmpi   c0,r4,10
        add    r4=r4+r5
        sub    r3=r7-r4
        and    r6=r3&r5
        bc     c0, A1
        st     ... =r4
A1:     st     ... =r6
```

weighted with the result latencies of the instructions in question. The weights on these edges are used to compute weights for each node in the graph, based on the longest path from the instruction to any other instruction in the same basic block.

FIGURE 6. Sample Dependence Graph



The scheduler computes two ready sets, one for the host and one for the guests. The ready set is the set of instructions whose reverse RAW and MISC dependences have all been satisfied. Each ready set is then ordered by the following cost function: where *weight* is the original

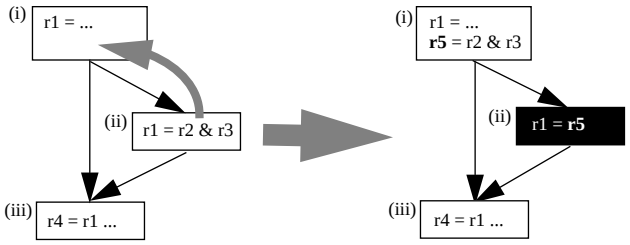
$$Cost = weight \times 10 + |kill| - |gen| - 2 \times |ren| - CP \times |copy|$$

weight assigned earlier, *kill* is the set of last uses, *gen* is the set of register values generated, *ren* is the set of destination registers that must be renamed, *CP* is the user-specified cost of introducing a copy instruction, and *copy* is the set of copy instructions that must be generated. The *kill* and *gen* terms are a heuristic for decreasing register pressure, while the *ren* and *copy* terms are used to discourage excessive resource consumption.

Once the ready sets are available, the scheduler attempts to find instructions from the host ready set for each idle functional unit; if it fails to do so, it looks for

Figure 2. If the definition of r1 in basic block (ii) is

FIGURE 2. Example of Register Copy Creation



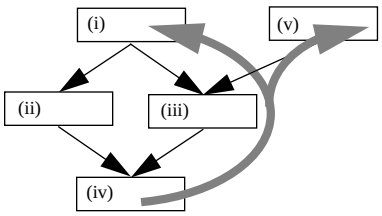
boosted into basic block (i), it must be renamed to prevent it from overwriting the previous value of r1, which must be preserved in case the left branch is taken. However, since multiple definitions of r1 reach the use in basic block (iii), that use cannot be renamed. Instead, a register copy is added to basic block (ii). This way, whichever branch direction is taken, the correct value residing in r1 is ensured when it is used in basic block (iii). The register copy that is explicitly added to basic block (ii) corresponds to the shadow structure copy that is implicitly encoded within each conditional branch instruction in the original boosting technique [25]. An additional instruction is introduced for each register copy that is specified explicitly in software. This cost corresponds to the additional hardware support needed to make that copy implicitly from the shadow register file.

In future implementations of the architecture, some form of hardware support can be beneficial for executing the additional register copy instructions. One means of support would be to add instruction opcodes that encode multiple register copy operations. Another would be to have specialized and dedicated functional units to execute these instructions. The primary resource required by these functional units would be additional register file ports.

2.3. Instruction Replication

To support the general case of upward speculative code motion, the case in which instructions are moved across either an entry point or a control flow join must be considered. A mechanism is needed to ensure that the boosted instructions are executed irrespective of which control flow path is taken to reach the guest basic block in which they originally reside. A simple solution is to replicate the boosted instruction at the end of each basic block that precedes the guest in the control flow graph. In cases where the guest is not an immediate successor of the host, however, the solution is not quite so simple. For example, in the control flow graph shown in Figure 3, an instruction

FIGURE 3. Example of Instruction Replication



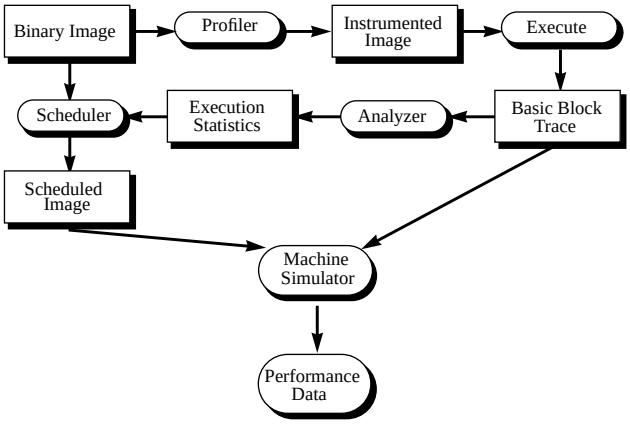
boosted from basic block (iv) to basic block (i) must be replicated in basic block (v), but not in (ii) or (iii).

A replicate set for each guest basic block in the scope is found by searching all paths from the guest to the host and marking each node that is on a path. The predecessor set of each marked basic block is checked for unmarked basic blocks; these are added to the replicate set of the guest. As instructions are boosted into the host, they are appended to each basic block in the replicate set.

3. Experimentation Framework

Figure 4 gives the overview of the experimentation framework. The program image under consideration is processed by the *goblin* profiler [28], which inserts calls to profiling routines at basic block boundaries, to produce a profiled image. Running that image with a data set produces a basic block trace. This basic block trace is analyzed for all conditional branches in the program to generate branch statistics, which are then fed to the scheduler, along with the original program image. The scheduler produces a new image, rescheduled for the machine configuration specified. A retargetable trace-driven pipeline simulator is used to determine the performance of the original and the new images.

FIGURE 4. Overview of Experimentation Framework



3.1. Profiler

goblin is an instruction-level profiling tool for the IBM RS/6000 machines [28]. The goal of goblin is to provide a precise instruction-level view of a program execution on the RS/6000; this goal is accomplished via invasive profiling. Invasive profiling inserts instrumentation code into the user program which does not perturb the program state, but is sufficient for maintaining an exact record of the program's execution. The *goblin* system can be used to generate a wide variety of execution and resource-utilization statistics. Our use of it is limited to collecting a basic block trace that captures control flow information for use by the scheduler.

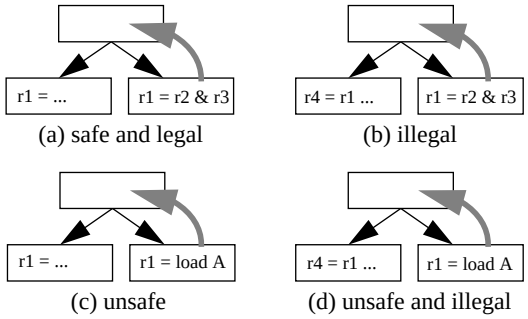
3.2. Scheduler

The instruction scheduler for performing AC boosting takes as input a binary executable image, generated by the IBM XL compiler back end [31], disassembles it, and per-

question. At this point, the results in the shadow structures are either committed or discarded, depending on the outcome of the branch.

Smith et al. [24] identify two characteristics of speculative code motion—safety and legality—that describe how program semantics can potentially be changed by unwanted side effects. The cross-product of the two produces four different types of code motion: safe and legal, safe but illegal, legal but unsafe, and unsafe and illegal, as illustrated in Figure 1. The *safety* characteristic of code

FIGURE 1. Types of Speculative Code Motion



motion describes its potential to cause an exception at an incorrect point in the program flow, whereas the *legality* of code motion refers to its destruction of program state that may be needed later. A combination of compiler and hardware techniques are proposed by Smith et al. to address these two potential violations of program semantics. To maintain correctness in the case of unsafe code motion, boosted instructions are labeled as non-excepting, and exception handling is delayed until the control flow has been resolved. Similarly, to maintain correctness for illegal code motion, instructions that are labeled as boosted temporarily have their results stored in shadow structures, again until their control dependences have been resolved.

1.3. Weakness of the Original Boosting Idea

While we agree that some hardware support is necessary for maintaining correctness in the case of unsafe code motion, we claim that the elaborate hardware scheme using shadow register files for handling illegal code motion is unnecessary. Rather than extending the hardware to contain multiple copies of the entire register file (one for each conditional branch that an instruction is allowed to move across) to prevent destructive side effects, or limiting performance by employing the less aggressive alternatives presented in [24], we claim that non-destructive storage allocation can be performed effectively by the compiler. Modifying the register allocation for instructions that have no implicit side effects is sufficient for guaranteeing semantic correctness for illegal code motion. Without any hardware support, we must forfeit unsafe code motion as well as code motion of instructions with destructive implicit side effects. While the performance results of our approach are not expected to be as impressive as those presented by Smith et al., our approach is applied to a real machine and uses existing hardware. Also, our scheduler is compatible with the existing instruction set architecture, since our approach does not require modifying the instruction formats to encode control dependence information within them.

2. Architecture-Compatible Code Boosting

Our scheduling framework operates on scheduling scopes, program fragments within which code motion is allowed; these are scheduled one after another. A scheduling *scope* is defined as a collection of basic blocks with one *host* basic block and one or more *guest* basic blocks, which are control-flow successors of the host. The number of guest basic blocks is determined by the maximum number of conditional branches that can be crossed as specified by the user. Instructions are scheduled primarily from the host, but whenever there is an idle functional unit and no available instruction in the host's data ready set, the algorithm looks for schedulable instructions from guest basic blocks and boosts them into the host basic block subject to semantic correctness constraints.

To preserve semantic correctness while moving instructions up from guest basic blocks to the host, some combination of the following three simple transformations is performed: *register renaming*, *register copy creation*, and *instruction replication*. These transformations are applied subject to dataflow, control flow, and control dependence information and employed only when required to preserve semantic correctness.

2.1. Register Renaming

Register renaming is the key to avoiding the need for having multiple shadow register files. To preserve program semantics while performing illegal code motion of the type illustrated in Figure 1(b), the result of the instruction in question must be stored in a location that is not live on the alternate control flow path. Smith et al. accomplish this by creating a duplicate shadow register file structure from which the value is copied once the branch condition is resolved. Instead, we rename the destination register as well as all of its dependent uses to an unused register.

The scheduling algorithm maintains a register scoreboard for determining which registers are in use, and uses a first-fit policy for selecting a replacement register for instructions that are boosted. The register scoreboard, which is the static scheduler's counterpart to the *scoreboards* used in dynamic scheduling mechanisms [17], is also used to determine when renaming is necessary. It is initialized to mark those registers that are live on alternate exits from the scheduling scope as busy, so that any instructions that are moved beyond those exit points and overwrite the live storage locations are renamed. Also, as instructions are scheduled, the scoreboard is updated to reflect the number of outstanding uses of the current variable residing in each register, so that once all uses have been scheduled, the register becomes available for reuse.

2.2. Register Copy Creation

Register renaming is not always sufficient for maintaining correct program semantics. In cases where the boosted instruction is one of multiple reaching definitions [2] of a variable, a mechanism is needed for choosing the appropriate definition based on the outcome of control flow dependences. An example of such a case is shown in

Architecture-Compatible Code Boosting for Performance Enhancement of the IBM RS/6000

Trung A. Diep, Mikko H. Lipasti*, John P. Shen

Computing Systems Center
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Boosting, first introduced by Smith et al. [25], is an instruction scheduling technique that increases the instruction-level parallelism by allowing the compiler to move instructions speculatively up past conditional branches and providing hardware support to delay committing the side effects of the boosted instructions until the conditional branches have been resolved. This paper proposes an enhanced compilation technique similar to boosting that provides performance improvements while maintaining instruction set architecture compatibility and eliminating the need for complex hardware support. The technique, called *Architecture-Compatible (AC) Boosting*, has been implemented for the IBM RS/6000 architecture. Code scheduling and machine simulation tools have been implemented, and experiments have been performed to demonstrate the feasibility of AC boosting on the current as well as future implementations of the IBM RS/6000 architecture.

1. Motivations

Instruction scheduling is a compilation technique that seeks to maximize functional unit utilization within a processor. By scheduling long latency instructions early in the instruction stream and filling dependence-induced stall slots with independent instructions, its goal is to increase the instruction-level parallelism exposed to the processor, increase the average number of instructions issued per cycle (IPC) [16], and consequently improve program performance. Most modern compilers implement some form of instruction scheduling, ranging from load and branch delay slot filling [15] to *trace scheduling* [8, 12] and *percolation scheduling* [20] for VLIW processors.

Instruction scheduling techniques can be subdivided into three main categories: those that are restricted to code motion within basic blocks, those that allow code motion between basic blocks only in the context of techniques such as loop unrolling [33, 12] or software pipelining [29, 3], and those that allow speculative code motion between basic blocks. *Speculative code motion* consists of moving instructions between basic blocks that are not control flow equivalent [11, 9], where the execution of one basic block does not necessarily imply the execution of the other. The *control flow equivalence* of two basic blocks A and B means that any control flow path used to reach B must pass through A (A *dominates* B in the control flow graph), while any control flow path that passes through A must also reach B (B *postdominates* A). Performing speculative code motion requires adding corrective actions to ensure that the semantics of the program are not changed.

1.1. Control Dependence Limitation

Heavily pipelined superscalar processors such as the IBM RS/6000 [14, 21] or the DEC AXP suffer a performance penalty when conditional branches are not resolved early enough to keep the pipelines filled. The penalty paid by early RISC designs with a single branch delay slot is magnified both by the length and the number of pipelines. Hence, locating instructions to fill the branch delay slots becomes increasingly more important so that sufficient *program parallelism* [19] is available to match the increased *machine parallelism* [19] offered by the processor.

Others [18, 30, 26] have argued that speculative code motion is necessary for extracting instruction-level parallelism from branch-intensive, non-scientific programs. While compilation techniques like loop unrolling and software pipelining work well at extracting parallelism from loop-intensive programs, they perform less effectively with programs that are dominated by if-then-else control flow constructs. Numerous approaches have been proposed for implementing speculative code motion, both in hardware and in software [17, 23, 27, 1].

1.2. Effective Static Code Scheduling Support

One of the most promising approaches for supporting speculative code motion combines both hardware and software techniques. It relegates code motion decisions to the compiler and leaves the elimination of unwanted side effects to the hardware. This approach is called *boosting* and was introduced by Smith et al. [25]. The authors present a convincing argument against dynamic instruction scheduling methods, pointing out their hardware complexity, limited instruction selection scope, and inability to make use of control dependence information. In contrast, they point out a static instruction scheduler's ability to exploit control dependence information and to select instructions from a much larger scope.

To enhance available program parallelism, boosting speculatively moves instructions up past conditional branches in the compiler and provides hardware support to delay committing the results and side effects of the boosted instructions until the conditional branches have been resolved. This is accomplished via tagging boosted instructions with their control dependence information and storing their side effects temporarily in shadow structures until the program control flow resolves the branches in

*. Currently with IBM Corporation at Rochester, Minnesota