Comparing Static And Dynamic Code Scheduling for Multiple-Instruction-Issue Processors

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

Center for Reliable and High-performance Computing
University of Illinois
Urbana, IL 61801

Abstract

This paper examines two alternative approaches to supporting code scheduling for multiple-instruction-issue processors. One is to provide a set of non-trapping instructions so that the compiler can perform aggressive static code scheduling. The application of this approach to existing commercial architectures typically requires extending the instruction set. The other approach is to support out-of-order execution in the microarchitecture so that the hardware can perform aggressive dynamic code scheduling. This approach usually does not require modifying the instruction set but requires complex hardware support.

In this paper, we analyze the performance of the two alternative approaches using a set of important non-numerical C benchmark programs. A distinguishing feature of the experiment is that the code for the dynamic approach has been optimized and scheduled as much as allowed by the architecture. The hardware is only responsible for the additional reordering that cannot be performed by the compiler. The overall result is that the dynamic and static approaches are comparable in performance. When applied to a four-instruction-issue processor, both methods achieve more than two times speedup over a high performance single-instruction-issue processor. However, the performance of each scheme varies among the benchmark programs. To explain this variation, we have identified the conditions in these programs that make one approach perform better than the other.

1 Introduction

Instruction pipelining has become a standard feature for improving the performance of commercial processors [Kogge 81] [Kane 87] [Intel 89] [IBM 90] [Amd] [Sparc 87]. A natural extension to instruction pipelining is to provide parallel data-paths in order to fetch, decode, and execute several operations per cycle. Such processors have been referred to as *multiple-instruction-issue* processors in recent literature [Smith 89].

A critical issue regarding the design of multiple-instruction-issue processors is code scheduling. Code scheduling methods ensure that control dependencies, data dependencies, and resource limitations are properly handled during concurrent execution. The goal is to produce a code schedule that minimizes the execution time in addition to enforcing the correctness of execution.

Code scheduling can be done at compile time (static scheduling) [Fisher 81] [Hennessy 83] and/or at run time (dynamic scheduling) [Tomasulo 67] [Thornton 70] [Patt 85]. Static scheduling requires intelligent compilation support whereas dynamic scheduling requires sophisticated hardware support. In practice, dynamic scheduling is assisted by static scheduling to improve performance and to reduce hardware cost. On the other hand, static scheduling is often assisted by hardware interlocking to enforce the correctness of execution.

Code scheduling decisions can have a major impact on the performance of multiple-instruction-issue processors. Therefore, many dynamic and static techniques for multiple-instruction-issue processors have been studied [Fisher 81] [Rau 81] [Fisher 83] [Nicolau 85] [Patt 85] [Ellis 86] [Hwu 86] [Colwell 87] [Howland 87] [Weiss 87] [Cohn 89] [Jouppi 89] [Rau 89] [Smith 89] [Sohi 89] [Golumbic 90] [Warren 90] [Smith 90].

Hardware concurrency detection and scheduling algorithms have been used in early high-end machines such as the IBM 360-91 [Tomasulo 67] and CDC 6600 [Thornton 70]. Weiss and Smith have compared Thornton's Scoreboarding Algorithm and Tomasulo's Algorithm [Weiss 87]. Sohi has proposed an extension to the Tomasulo's Algorithm to support precise interrupts [Sohi 87]. Acosta, Kjelstrup, and Torng have described a dispatched stack hardware scheme [Acosta 86]. These studies have been mostly based on numerical applications and loop kernels. The results presented in this paper are based on non-numerical programs.

Patt and Hwu have adapted Tomasulo's Algorithm to a class of multiple-instruction-issue processors called HPS [Patt 85] [Hwu 86]. They have analyzed the performance of this multiple-instruction-issue processor with limited compilation support. Smith, Johnson, and Horowitz have used trace-based simulations to determine that dynamic scheduling can achieve an execution rate of about two operations per cycle [Smith 89]. These works focus on the

performance of dynamic code scheduling. Our work differs from these previous works in two important ways. First, we have provided static code scheduling for our dynamic scheduling experiments. Second, we characterize important tradeoffs between static and dynamic scheduling.

Fisher demonstrated that trace scheduling can find sufficient instruction-level parallelism to exploit VLIW architectures [Fisher 81]. Code scheduling and resource allocation for VLIW machines are done at compile-time [Fisher 81] [Ellis 86] [Colwell 87]. Rau has designed the ESL Polycyclic processor [Rau 81] and the Cydra 5 supercomputer [Rau 89]. He has also studied code scheduling techniques for those machines. Cohn, Gross, Lam, and Tseng have studied the architecture and compiler tradeoffs in the design of iWarp which is capable of specifying up to nine operations in an instruction [Cohn 89]. Lam has applied software pipelining to a Systolic Array compiler [Lam 88]. Weiss and Smith have shown that loop unrolling and software pipelining are effective in increasing parallelism [Weiss 87]. Hwu and Chang have studied the ability of a code generator to exploit various multiple-instruction-issue processors [Hwu 88]. Uht, Polychronopoulos, and Kolen show that a combination of compiler and hardware techniques is most effective for exploiting parallelism [Uht 87]. These studies have focused mainly on numerical kernels and applications. Our application domain is non-numerical programs.

Jouppi and Wall have measured the instruction-level parallelism of some non-numerical Modula-2 and C programs using an optimizing compiler that performs local code scheduling. Assuming unit-time operation delay, they reported that there are between 1.6 and 2.1 concurrently executable operations per cycle [Jouppi 89]. In this paper, we have implemented more aggressive static scheduling techniques, and have considered non-unit-time operation delays.

Smith, Lam, and Horowitz have proposed and studied a static scheduling scheme called instruction boosting that allows operations to be moved across a preceding conditional branch [Smith 90]. The hardware support relieves the compiler from the first restriction described in Section 2.3. They concluded that static scheduling supported by instruction boosting can be comparable in performance with dynamic scheduling. Our work has two important differences from their work in instruction boosting. First, we provide more powerful compiler optimizations beyond trace scheduling. They have only assumed local code scheduling for their dynamic scheduling results. As a result, we have achieved a higher performance level. Second, our static scheduling model requires only a set of non-trapping instructions whereas theirs needs shadow register and write buffer structures to implement instruction boosting.

1.1 Our Approach and Contribution

The objective of this work is to study two alternative approaches to supporting code scheduling for multiple-instruction-issue processors. One approach is to extend the architecture with a set of non-trapping instructions so that the compiler can perform aggressive static code scheduling. The other is to provide out-of-order execution support in the microarchitecture so that the hardware can perform dynamic code scheduling on top of static scheduling. The question is how much performance improvement can each approach produce for real programs.

Using the IMPACT-I C compiler, we generate optimized machine code for each approach. For the static approach, the compiler performs aggressive global code motion using non-trapping instructions. For the dynamic approach, the compiler performs as much global code motion as allowed by the instruction set architecture specification. Both approaches receive full-scale advanced compilation support which allow them to achieve higher performance for non-numerical C programs than previously reported by other researchers.

Our experimental results show that both approaches have achieved more than two times speedup over a good base architecture. Overall, the two approaches are comparable in performance. However, the relative performance varies among the individual benchmark programs. To explain this variation, we have identified the conditions in these programs that make one approach perform better than the other. These results lead to further understanding of both dynamic and static code scheduling techniques.

2 Compilation Support

2.1 Base Compiler Technology

It is important to evaluate processor architectures using highly optimized code. Naive code may contain redundant operations which show deceptive parallelism. Also, unnecessary operations could introduce artificial dependencies which restrict the effectiveness of static scheduling. The IMPACT-I C compiler generates code for several machines, including MIPS-R2000 [Kane 87], SPARC [Sparc 87], AMD 29K [Amd], Intel i860 [Intel 89], and a family of multiple-instruction-issue processors within the IMPACT architectural framework. To calibrate the quality of the code generated by the IMPACT-I C compiler, we have compared the execution time of its output code with those of a leading commercial compiler (MIPS CC release 2.1) and a leading public domain compiler (GNU CC release 1.37.1) on the DEC 3100 workstation. Table 1 shows the output code execution time ratio of the MIPS C compiler and the GNU C compiler over the IMPACT-I C compiler. For the benchmark programs used in this paper, our compiler achieves an average of 1.04 speedup with a standard deviation of 0.04 over the MIPS C compiler using the (-O4) option. Therefore, the speedup numbers that we record for multiple-instruction-issue architectures are based on very efficient sequential code.

2.2 Multiple-Instruction-Issue Optimizations

The scope of static code scheduling is generally small for integer C programs because of the high frequency of branches. To enlarge the scope of static scheduling, we have added several code transformations to the IMPACT-I C compiler, including function inline expansion, instruction placement, branch target expansion, loop unrolling, loop peeling, and superblock formation [Chang 90]. To reduce the depth of critical paths, we have incorporated register renaming, global variable migration to registers,

name	*IMPACT -O5*	*MIPS -O4*	*GNU -O*
cccp	1.00	1.08	1.09
cmp	1.00	1.05	1.05
compress	1.00	1.02	1.06
eqn	1.00	1.15	1.15
eqntott	1.00	1.04	1.33
espresso	1.00	1.02	1.15
grep	1.00	1.03	1.24
lex	1.00	1.01	1.04
qsort	1.00	1.01	1.08
tbl	1.00	1.02	1.07
wc	1.00	1.04	1.15
yacc	1.00	1.00	1.11

Table 1: Execution time comparison.

operation combining, operation folding, and memory disambiguation [Chang 90].

Function inline expansion expands frequently called functions into the calling function. Instruction placement groups instructions that tend to execute in sequence into sequential memory locations. Branch target expansion copies the target basic block of a frequently taken branch into its fall-through path. Loop peeling fully unrolls loops with small number of iterations. Memory disambiguation determines non-conflicting memory addresses for memory load/store operations. Superblock formation increases code optimization and scheduling freedom by duplicating frequently executed code sections.

2.3 Static Code Scheduling

In the IMPACT-I C compiler, static code scheduling is done twice, before and after register allocation. Our code scheduler moves code both upward and downward across branch operations. Moving operations from above a branch operation to below is always safe. On the other hand, moving operations from below a branch to above is not always safe. There are two major restrictions on upward code motion.

1. The moved operation must not destroy a value that is needed when the branch is taken.
2. The moved operation must not cause an exception that may terminate the program execution.

For example, it is not safe to move a memory load operation above a branch because of the possibility of memory access violation. We have implemented a code scheduling algorithm that enforces the above two restrictions. We refer to this algorithm as *restricted code percolation.*

It is possible to free the code scheduler from the second restriction if the division operation and the memory load operation do not cause exceptions. Instead of trapping on divide by zero or illegal memory access, a magic value is returned. Page faults can be handled as they occur. We refer to this code scheduling model as *general code percolation.*

3 Hardware Scheduling

The instruction pipeline model assumed in this paper is partitioned into several stages: instruction fetch, instruction decode and register operand fetch, instruction issue, instruction execute, and result distribution. Given two operations op_A and op_B, such that op_B depends on the result of op_A, and op_A takes n cycles to execute, static code scheduling inserts independent operations between op_A and op_B so that op_A and op_B are fetched by the processor at least n cycles apart. In practice, the compiler may not be able to find enough independent operations to execute between op_A and op_B. With in-order execution, the instruction fetch and decode stages are stalled until the result of op_A becomes available. Dynamic code scheduling alleviates this problem by allowing subsequent operations, that are independent of op_A and op_B, to proceed to the function units while op_B waits for op_A.

We have identified three major cases where dynamic scheduling can improve performance on top of static code scheduling.

Load Bypassing: Memory load operations often reside on the critical path of program execution. Therefore, allowing memory load operations to bypass memory store operations may improve performance by making the load results available early. This is referred to as *load bypassing* . Load bypassing can be performed by the static code scheduler and/or the dynamic code scheduler. To enforce the correctness of execution, a memory load operation is allowed to bypass an earlier store if their addresses do not conflict. Because the compiler does not know the address of some memory access operations, a static code scheduler may not be able to take advantage of all opportunities for load bypassing. At run-time, a dynamic code scheduler can detect opportunities missed by the static code scheduler.

Loop Iteration Overlapping: Within a big loop or any outer loop, loop optimizations, such as loop unrolling and loop peeling, are turned off to control code size expansion. Thus, with only static scheduling, operations from different iterations cannot execute concurrently. With out-of-order execution, the next iteration of the loop can proceed, and overlap its execution with that of the previous iteration.

Tolerance To Data Cache Miss Delay: For static scheduling, the instruction pipeline is stalled on a data cache miss. With out-of-order execution, the hardware allows independent operations to bypass the waiting memory operations. Therefore, the delay due to cache miss may be hidden.

4 Experiments and Analysis

Experiments have been conducted to evaluate the performance of static and dynamic code scheduling. The scheduling methods examined are restricted code percolation with in-order execution, general code percolation with in-order execution, and restricted code percolation with out-of-order execution. For each scheduling approach, we show the speedup achieved with instruction issue rates of one, two, four, and eight. Based on the experimental results and on the details of the benchmark programs, we discuss the strengths and limitations of each scheduling method.

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
qsort	quick sort
tbl	format tables for troff
wc	word count
yacc	parsing program generator

Table 2: Benchmarks.

function	*base*
integer alu	1
barrel shifter	1
integer mult	3
integer div	25
load	2
store	-
FP alu	3
FP conv	3
FP mult	4
FP div	25

Table 3: Operation delays.

4.1 Benchmark Programs

We have collected *C* application programs from several domains, including text processing, CAD design, and UNIX utilities. Table 2 shows the benchmark programs that are used in this paper. The *name* column shows the names of the benchmark programs. The *description* column briefly describes the nature of the benchmark program.

4.2 Base Architecture

A single-instruction-issue in-order processor supporting the restricted code percolation model is used as the base architecture. The instruction set architecture is a superset of the MIPS R2000 architecture with extensions in register file size, branch semantics, and floating point unit pipelining. The base architecture includes a 64-entry integer register bank and a 32-entry floating-point register bank. The architecture uses a squashing branch scheme and profiled-based branch prediction. One branch slot (one instruction) is allocated for each predict-taken branch. All function units are fully pipelined with deterministic delays. Table 3 shows the operation delays. Asynchronous events such as cache misses stall the processor pipeline.

The compiler performs all optimizations and restricted code percolation for the base architecture. On the average, the base architecture executes more than 0.9 operations per cycle.

4.3 Multiple-Instruction-Issue Architectures

Three multiple-instruction-issue architectures are evaluated. Each supports a different code scheduling method. All three architectures duplicate hardware resources of the base architecture. Additional access ports to the register file and the cache memory are provided to satisfy the increased demands due to multiple-instruction-issue. All operation delays remain the same as in Table 3.

The first architecture supports restricted code percolation and in-order execution. We refer to this architecture as *restricted in-order execution* . The second architecture supports restricted code percolation and out-of-order execution. This architecture is referred to as *restricted out-of-order execution* . The third architecture supports general code percolation and in-order execution. This architecture is derived from the first architecture by adding non-trapping instructions. This last architecture is referred to as *general in-order execution* .

For the first and third architecture, which implement in-order execution, cache misses stall the processor pipeline. In addition to their individual code scheduling algorithms, the compiler performs full-scale code optimizations for all three architectures.

4.4 Measurement Tools

To analyze the performance of in-order execution architectures, we have implemented a profiler to record the execution count of every instruction and the branch statistics. Because all operation delays are deterministic, we can derive the best and worst case execution times for the benchmark programs. The worst case is due to long operation delays that protrude from one basic block to an off-trace basic block. For the benchmark programs used in this paper, the difference between the best and worst case execution times is negligible. We will use the worst case execution time.

To measure the performance of out-of-order execution, we have implemented a trace generator and a trace analyzer.

Trace Generator: The code generator has been modified to insert probes into the user program. Executing the modified program with sample input data produces an instruction trace.[1] The instruction traces are then processed by a trace analyzer which simulates out-of-order execution hardware.

Trace Analyzer: The trace analyzer uses entire instruction traces. The trace analyzer simulates a simple dynamic code scheduling model that has an infinite number of function units and an infinite number of reservation

[1] For all benchmark programs, the length of the instruction traces accurately matches the estimation made by the profiling tool that has been described earlier.

station entries for each function unit. The control unit fetches one instruction (N operations) per cycle, except when an incorrectly predicted branch operation causes the control unit to refill the pipeline. After an instruction has been decoded, operations that do not have both source operands are placed into the reservation stations. Otherwise, operations are directly submitted to the function units. An operation is moved from a reservation station to a function unit as soon as its source operands are available.

Memory load operations are allowed to bypass preceding memory store operations if the memory addresses do not conflict. Cache misses do not stall the instruction pipeline. When a data cache miss occurs, the processor can continue to execute independent operations. This allows the dynamic scheduler to overlap data cache refill with the execution of other operations.

A two-level direct-mapped cache model is assumed in the simulation. We simulated three different first level cache sizes: infinite, 8KB, and 16KB. A miss from the first level cache adds four cycles to the access. The simulation assumes a 128KB second level cache. The miss ratio for the second level cache is negligible for all benchmarks used.

A branch operation that has been decoded but not yet executed is called a *pending branch* . The trace analyzer allows instructions to bypass an infinite number of pending branch operations. This feature is especially useful when the static code scheduler is limited by the restricted percolation model. Allowing operations to bypass branches is also useful when loops are not unrolled at compile-time due to code size and register constraints. It increases the overlap between the execution of adjacent loop iterations.

4.5 Results and Analysis

Figures 1 through 3 show the performance of multiple-instruction-issue processors. Each data point represents the harmonic mean of speedup over the base architecture for all benchmark programs. The speedup of a machine configuration over the base architecture for individual benchmark programs are listed in Tables 4 through 6 that are attached to the end of this paper. Each column of Tables 4 through 6 is labeled XYZ, where X is the issue rate, Y indicates either restricted (r) or general (g) code percolation, and Z indicates in-order (i) or out-of-order (o) execution.

Ideal Cache Results

Figure 1 and Table 4 present speedup results for an infinite first level data cache. Overall, restricted out-of-order execution performs slightly better than general in-order execution. They both achieve substantial improvement over the restricted in-order execution model.

A closer look at the benchmark programs show that load bypassing is the most beneficial feature of dynamic code scheduling. *Lex* and *qsort* are examples where general in-order execution is severely limited by memory dependencies. For these programs, load bypassing at run time allows memory load operations on the critical path to execute early. This resulted in the clear performance advantage of restricted out-of-order execution for *lex* and *qsort* (see Table 4).

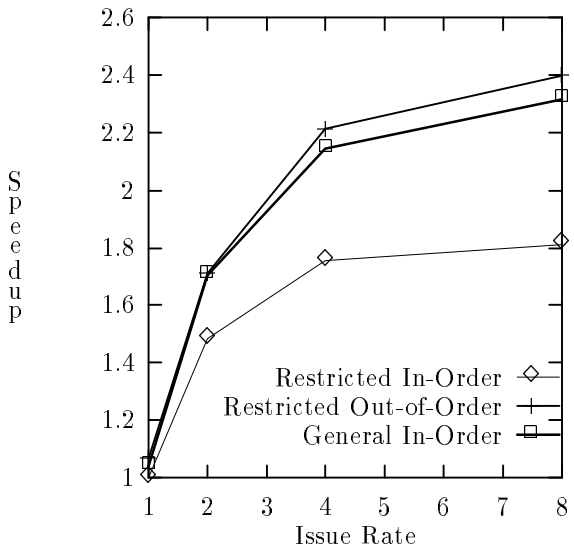

Figure 1: Speedup for Ideal Cache.

The ability to examine a large section of code to make scheduling decisions gives static code scheduling most of its performance advantage. With general code percolation support, the static code scheduler can concurrently execute instructions from different iterations of a loop. *Wc* and *compress* are examples where general in-order execution works better than restricted out-of-order execution. The bodies of several important loops in these programs start with memory load operations. With loop unrolling and general code percolation support, the static code scheduler is able to concurrently execute operations from different iterations to improve performance. In the restricted code percolation model, however, the static code scheduler does not allow the load operations to percolate into previous iterations. Therefore, the iterations are fetched sequentially from memory. By the time the operations from one iteration are fetched, it is already too late to execute them in parallel with the operations from the previous iteration. As a result, general in-order execution has a clear performance advantage over restricted out-of-order execution for *wc* and *compress*.

Small Cache Results

Figure 2 and Table 5 present speedup results for an 8KB data cache. Cache misses degrade the performance of all architectures. Restricted out-of-order execution tolerate the cache misses better than the in-order execution models. The effect is most visible for *compress* (compare Tables 4 and 5). From more detailed measurements, we found that *compress* has a large number of cache misses whose delay can be hidden by the dynamic code scheduler.

Figure 3 and Table 6 present speedup results for a 16KB data cache. The performance of the in-order execution models in Figure 3 is slightly better than in Figure 2. On the other hand, the performance of restricted out-of-order execution were virtually identical in both cases.

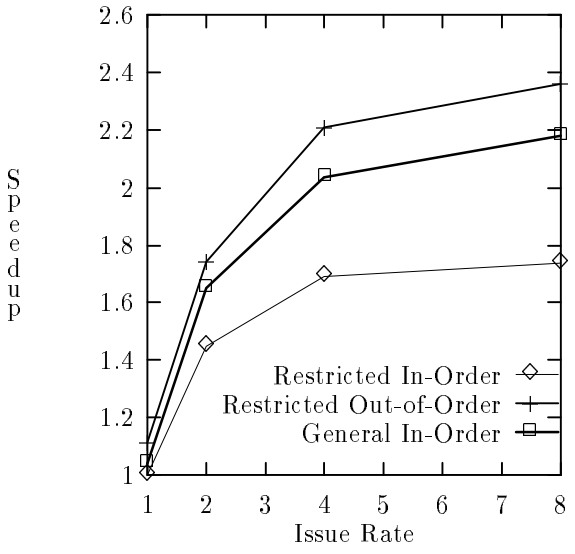

Figure 2: Speedup for 8KB Data Cache.

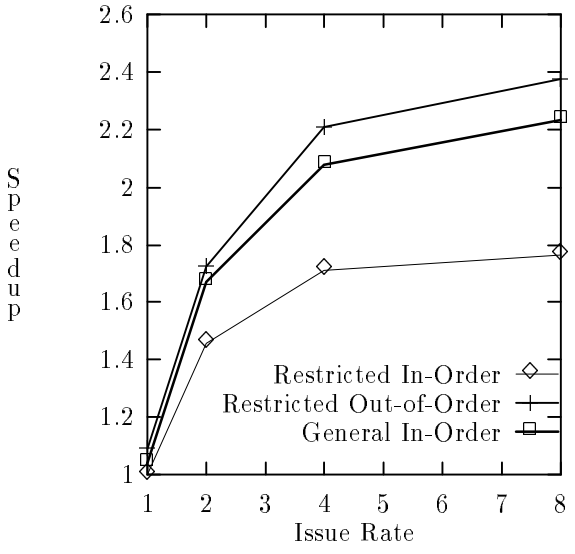

Figure 3: Speedup for 16KB Data Cache.

This shows that the performance of restricted out-of-order execution is less sensitive to cache size than in-order execution models.

5 Conclusion

This paper addresses a seemingly simple question: how should one support code scheduling for multiple-instruction-issue processors. To give useful answers to this question, we provide full-scale compilation support for each method under study. Only by doing so can one be sure that experimental results are reproducible in real product development. With the completion of the IMPACT-I C compiler, we have satisfied this important requirement. With code generators for MIPS R2000, SPARC, Intel i860, and AMD 29K, the results reported in this paper readily apply to the multiple-instruction-issue implementations of these commercial architectures.

We focused on two promising code scheduling candidates: general in-order execution and restricted out-of-order execution. For a set of important non-numerical programs, four-instruction-issue processors supporting both methods have achieved more than two times speedup over a high-performance single-instruction-issue processor. Both of them perform substantially better than restricted in-order execution.

General in-order execution is especially attractive because it only requires a set of non-trapping instructions. These instructions can be added to most existing commercial architectures in an upward compatible manner. On the other hand, restricted out-of-order execution should be considered if speeding up existing binary code is the major objective. Therefore, architecture compatibility requirements and hardware design complexity are the two major decision factors for choosing between these two methods.

In general, restricted out-of-order execution has better tolerance against data cache misses than in-order execution models. By examining the benchmark programs in detail, we have identified conditions that make one method more preferable than the other. Load bypassing is the most important feature that allows restricted out-of-order execution to perform better than general in-order execution for two benchmarks. The ability to schedule memory load operations across many loop iterations is the most important feature that allows general in-order execution to outperform restricted out-of-order execution for another two of the benchmarks.

Combining general code percolation with dynamic code scheduling has the potential to further improve performance beyond each individual approach. We intend to study the performance and tradeoffs of this combination in our future research.

Acknowledgements

The authors would like to thank Nancy Warter 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, Matsuhita Electric Corporation, the National Aeronautics

and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[Acosta 86] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", *IEEE Transactions on Computers,* vol.C-35, no.9, pp.815-828, September, 1986.

[Amd] Advanced Micro Devices, "Am29000 Streamlined Instruction Processor, Advance Information", Publication Number 09075, Rev. A, Amendment /0, Sunnyvale, California.

[Chang 90] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Code Optimization Techniques for Multiple-instruction-issue Architectures," Center for Reliable and High-Performance Computing Report, Uiversity of Illinois, in preparation.

[Cohn 89] R. Cohn, T. Gross, M. Lam, and P.S. Tseng, "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessors", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April, 1989.

[Colwell 87] 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.

[Ellis 86] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* The MIT Press, 1986.

[Fisher 81] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction", *IEEE Transactions on Computers,* vol.c-30, no.7, July 1981.

[Fisher 83] J. A. Fisher, "VLIW architectures and the ELI-512", *Proceedings of the 10th Annual Symposium on Computer Architecture,* June, 1983.

[Golumbic 90] M.C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks", *IBM Journal of Research and Development,* vol.34, no.1, pp.93-97, January, 1990.

[Gross 86] T. Gross and M. S. Lam, "Compilation for a High-Performance Systolic Array", *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction,* June, 1986

[Hennessy 83] J. L. Hennessy and T. Gross, "Postpass Code Optimization of Pipelined Constraints", *ACM Transactions on Programming Languages and Systems,* vol.5, pp.422-448, ACM, July, 1983.

[Howland 87] M. A. Howland, R. A. Mueller, and P. H. Sweany, "Trace Scheduling Optimization in a Retargetable Microcode Compiler", *Proceedings of the 20th International Microprogramming Workshop,* Colorado Springs, December, 1987.

[Hwu 86] 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.

[Hwu 89.3] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling Realistic C Programs", *Proceedings, ACM SIGPLAN'89 Conference on Programming Language Design and Implementation,* Portland, Oregon, June 21-23, 1989.

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

[IBM 90] IBM, Special Issue on IBM RISC System/6000 Processor, *IBM Journal of Research and Development,* vol. 34, no. 1, January, 1990.

[Intel 89] Intel, "i860(TM) 64-Bit Microprocessor", Order Number 240296-002, Santa Clara, California, April, 1989.

[Jouppi 89] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April, 1989.

[Kane 87] G. Kane, *MIPS R2000 RISC Architecture,* Prentice Hall, Englewood Cliffs, NJ, 1987.

[Kogge 81] P. M. Kogge, *The Architecture of Pipelined Computers,* pp.237-243, McGraw-Hill, 1981.

[Lam 88] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June, 1988.

[Nicolau 85] A. Nicolau, "Uniform parallelism exploitation in ordinary programs", *Proceedings of the International Conference on Parallel Processing,* pp.614-618, August, 1985.

[Patt 85] Y. N. Patt, W. W. Hwu, and M. C. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction", *Proceedings of the 18th International Microprogramming Workshop,* pp.103-108, Asilomar, CA, December, 1985.

[Rau 81] 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.

[Rau 89] B. Rau, D. Yen, W. Yen, and R.A. Towle, "The Cydra 5 departmental supercomputer", *Computer,* vol.22, pp.12-35, January, 1989.

[Smith 89] 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.

[Smith 90] 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.

[Sohi 87] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High Performance, Interruptible Pipelined Processors", *Proceedings of the 14th Annual Symposium on Computer Architecture,* June, 1987.

[Sohi 89] G. S. Sohi and S. Vajapeyam, "Tradeoffs in Instruction Format Design for Horizontal Architectures", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April, 1989.

[Sparc 87] *The SPARC^{TM} Architecture Manual,* Part No. 800-1399-07, Revision 50, SUN, Mountain View, California, August 1987.

[Thornton 70] J. E. Thornton, *Design of a Computer: The Control Data 6600,* Glenview, IL: Scott, Foresman and Co., 1970.

[Tomasulo 67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development,* vol.11, pp.25-33, January, 1967.

[Uht 87] A. K. Uht and C. D. Polychronopoulos and J. F. Kolen, "On the Combination of Hardware and Software Concurrency Extraction Methods", *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture,* pp.133-141, December, 1987.

[Warren 90] H.S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor", *IBM Journal of Research and Development,* vol.34, no.1, pp.85-92, January, 1990.

[Weiss 87] 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.

config	*1ri*	*2ri*	*4ri*	*8ri*	*1ro*	*2ro*	*4ro*	*8ro*	*1gi*	*2gi*	*4gi*	*8gi*
cccp	1.00	1.42	1.63	1.65	1.03	1.56	1.83	1.89	1.03	1.55	1.74	1.83
cmp	1.00	1.29	1.48	1.48	1.15	1.66	2.12	2.24	1.14	1.98	2.23	2.23
compress	1.00	1.61	1.90	1.92	1.03	1.73	2.24	2.33	0.99	1.82	2.70	3.05
eqn	1.00	1.41	1.61	1.63	1.17	1.80	2.24	2.30	1.13	1.74	1.98	2.02
eqntott	1.00	1.47	1.57	1.58	1.03	1.61	1.88	1.92	1.00	1.47	1.58	1.59
espresso	1.00	1.45	1.68	1.71	1.04	1.59	1.94	2.02	0.99	1.53	1.85	1.91
grep	1.00	1.72	2.34	2.68	1.02	1.93	3.17	4.19	1.01	1.93	2.86	4.00
lex	1.00	1.53	2.02	2.03	1.03	1.80	2.80	3.02	1.01	1.61	2.19	2.27
qsort	1.00	1.67	2.25	2.66	1.00	1.68	2.29	3.22	1.00	1.66	2.22	2.61
tbl	1.00	1.44	1.63	1.70	1.04	1.65	2.11	2.30	1.02	1.64	2.22	2.46
wc	1.00	1.40	1.61	1.64	1.23	1.88	2.33	2.40	1.21	2.08	2.94	3.38
yacc	1.00	1.39	1.62	1.64	1.06	1.60	2.04	2.15	1.00	1.65	2.11	2.29

Table 4: Ideal Speedup.

config	*1ri*	*2ri*	*4ri*	*8ri*	*1ro*	*2ro*	*4ro*	*8ro*	*1gi*	*2gi*	*4gi*	*8gi*
cccp	1.00	1.40	1.59	1.61	1.06	1.57	1.83	1.88	1.03	1.52	1.69	1.78
cmp	1.00	1.28	1.47	1.47	1.17	1.69	2.15	2.27	1.13	1.95	2.19	2.19
compress	1.00	1.46	1.64	1.65	1.20	1.77	2.08	2.15	0.99	1.60	2.08	2.25
eqn	1.00	1.39	1.58	1.59	1.20	1.83	2.23	2.27	1.12	1.69	1.91	1.94
eqntott	1.00	1.44	1.54	1.54	1.07	1.65	1.92	1.94	1.00	1.44	1.54	1.55
espresso	1.00	1.43	1.64	1.67	1.07	1.63	1.96	2.04	0.99	1.51	1.80	1.85
grep	1.00	1.72	2.33	2.68	1.02	1.94	3.17	4.20	1.01	1.93	2.85	3.97
lex	1.00	1.52	1.98	1.99	1.05	1.80	2.75	2.95	1.01	1.59	2.14	2.21
qsort	1.00	1.54	1.94	2.19	1.12	1.78	2.33	2.91	1.00	1.53	1.92	2.16
tbl	1.00	1.43	1.60	1.66	1.09	1.69	2.10	2.26	1.02	1.61	2.12	2.33
wc	1.00	1.39	1.60	1.64	1.40	1.89	2.33	2.40	1.21	2.07	2.92	3.35
yacc	1.00	1.38	1.60	1.62	1.08	1.63	2.05	2.16	1.00	1.62	2.05	2.22

Table 5: Speedup With 8K Cache Miss.

config	*1ri*	*2ri*	*4ri*	*8ri*	*1ro*	*2ro*	*4ro*	*8ro*	*1gi*	*2gi*	*4gi*	*8gi*
cccp	1.00	1.40	1.60	1.62	1.05	1.56	1.82	1.88	1.03	1.52	1.70	1.79
cmp	1.00	1.29	1.47	1.47	1.16	1.67	2.14	2.25	1.14	1.96	2.21	2.21
compress	1.00	1.50	1.71	1.72	1.14	1.78	2.17	2.24	0.99	1.66	2.24	2.45
eqn	1.00	1.41	1.60	1.62	1.18	1.81	2.23	2.30	1.13	1.72	1.96	2.00
eqntott	1.00	1.45	1.55	1.55	1.06	1.63	1.91	1.94	1.00	1.45	1.56	1.56
espresso	1.00	1.44	1.66	1.68	1.05	1.60	1.93	2.01	0.99	1.52	1.82	1.88
grep	1.00	1.72	2.33	2.68	1.02	1.93	3.17	4.20	1.01	1.93	2.86	3.99
lex	1.00	1.52	1.99	2.01	1.04	1.80	2.78	2.98	1.01	1.59	2.16	2.23
qsort	1.00	1.58	2.03	2.33	1.07	1.73	2.31	3.01	1.00	1.57	2.01	2.30
tbl	1.00	1.43	1.62	1.69	1.05	1.65	2.10	2.29	1.02	1.63	2.19	2.43
wc	1.00	1.39	1.61	1.64	1.23	1.89	2.33	2.40	1.21	2.07	2.93	3.38
yacc	1.00	1.38	1.61	1.63	1.06	1.62	2.05	2.16	1.00	1.63	2.09	2.26

Table 6: Speedup With 16K Cache Miss.