IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

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

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

Abstract

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

We ran experiments to characterize the following architectural design issues: code scheduling model, instruction issue rate, memory load latency, and function unit resource limitations. Based on the experimental results, we propose the IMPACT Architectural Framework, a set of architectural features that best support the IMPACT-I C compiler to generate efficient code for multiple-instruction-issue processors. By supporting these architectural features, multiple-instruction-issue implementations of existing and new architectures receive immediate compilation support from the IMPACT-I C compiler.

1 Introduction

Computer engineers have been striving to improve uniprocessor performance since the invention of computers. This paper is concerned with exploiting instruction level concurrency to achieve high performance. The traditional approach to exploiting concurrency is to provide the necessary support for instruction pipelining and overlapping [Kogge 81]. By optimizing a simple instruction pipeline structure, current pipelined processors can execute nearly one operation per cycle [Hennessy 81]. A natural extension to instruction pipelining is to provide parallel datapaths 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.

An important problem in the design of multiple-instruction-issue processors is to ensure that the compiler can generate efficient code for the hardware. To solve this problem, we have constructed the IMPACT-I C compiler, a retargetable compiler with code optimization components especially developed for multiple-instruction-issue processors. These code improving techniques include function inline expansion, instruction placement, loop unrolling, loop peeling, memory disambiguation, register renaming, branch prediction, critical path depth reduction, and an integrated register allocation and code scheduling algorithm.

Using the IMPACT-I C compiler, we conducted experiments to characterize the performance implications of engineering tradeoffs including alternative code scheduling models, instruction issue rate, memory load latency, and function unit resource limitations. All experimental results are derived from important non-numerical programs with realistic input data. Based on the experimental results, we have identified a set of architectural features that best support the IMPACT-I C compiler to generate efficient code for multiple-instruction-issue processors. We call the collection of these architectural features the IMPACT Architectural Framework.

1.1 Related Work

In a multiple-instruction-issue machine it is important to increase the scheduling scope in order to expose more parallelism. In processors with hardware support for dynamic code scheduling, the scope of instruction scheduling is limited by the instruction fetch mechanism [Patt 85] [Hwu 86] [Smith 89]. However, simple in-order execution architectures rely on aggressive compile-time optimizations to increase the scope. Trace scheduling has been proposed which combines basic blocks that tend to execute in sequence

0

into a trace and then schedules operations within traces [Fisher 81] [Ellis 86] [Howland 87]. Several researches have used loop unrolling, software pipelining, and other techniques to enlarge the scheduling scope. Weiss and Smith showed that loop unrolling and software pipelining effectively improve the performance of a deeply pipelined processor [Weiss 87]. Lam described a software pipelining technique which handles loops with conditional branches and showed its effectiveness for a VLIW architecture [Lam 88]. Anantha and Long use loop peeling and branch replication to increase the scope for parallel code compaction [Anantha 90].

For numerical applications, the above techniques expose large amounts of parallelism and code scheduling is relatively straight forward. However, non-numerical code tends to be control intensive. 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 latencies, they report that there are between 1.6 and 2.1 concurrently executable operations per cycle [Jouppi 89]. This suggests the need for more aggressive code scheduling techniques. Nicolau presented a percolation code scheduling method to support parallel execution [Nicolau 85]. Smith, Lam, and Horowitz have proposed a speculative execution technique which allows operations to be moved across a preceding branch operation. Using a four-issue microarchitecture and a single-instruction-issue compiler, they report about 1.63 speedup against a scalar processor which executes approximately 0.9 instructions per cycle. These results reinforce the need for more aggressive compile-time optimizations to improve the performance of multiple-instruction-issue processors.

The architectural support for these code optimizations and code scheduling techniques has been investigated by several researchers [Acosta 86] [Golumbic 90] [Rau 81] [Rau 89]. Colwell *et. al.*, use non-trapping instructions in the Multiflow Trace Computer to enable more compiler code motion and optimization than a traditional approach to exception handling would have allowed [Colwell 87]. Several researchers have analyzed the architectural tradeoffs of multiple-instruction-issue processors. Sohi and Vajapeyam investigated tradeoffs in instruction format design [Sohi 89]. Cohn, Gross, Lam, and Tseng studied resource performance tradeoffs for the iWarp processor [Cohn 89]. These studies have focused on numerical applications.

In this paper we present an architectural framework based on an aggressive multiple-instruction-issue compiler for non-numerical programs. Non-numerical programs are characterized by increased frequency of branches, smaller loop bodies, and fewer loop iterations.

1.2 Organization Of This Paper

This paper is organized into five sections. Section 2 presents our compiler technology and static code scheduling techniques. Section 3 presents experimental results. Section 4 describes the IMPACT Architectural Framework. Section 5 provides concluding remarks.

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 1: Benchmarks.

2 The IMPACT-I C Compiler

The IMPACT-I C compiler serves two important purposes. First, it is intended to generate highly optimized code for existing commercial microprocessors. We have constructed code generators for the MIPS-R2000, SPARC, AMD29K, and the i860 processors. Second, it provides a platform for studying new code optimization techniques for multiple-instruction-issue architectures. These new code optimization techniques, once validated, can be immediately applied to the multiple-instruction-issue implementations of existing and new commercial architectures.

2.1 Traditional Code Optimizations

Code improving techniques in the IMPACT-I C compiler can be roughly categorized into two groups: machine-independent optimizations and machine-dependent optimizations. Machine-independent optimizations include classical local and global code optimizations [Aho 86], function inline expansion [Hwu 89.2], instruction placement optimization [Chang 88] [Hwu 89.1], loop unrolling, intelligent generation of switch statements [Chang 89.2], and jump optimization. Machine-dependent optimizations include profile-based branch prediction, constant preloading, graph-coloring-based register allocation [Chaitin 82] [Chow 84], and code scheduling. A profiler has been integrated into the IMPACT-I C compiler. When hardware resources are scarce, the profile information helps to identify the most frequently executed program sections and variables.

2.2 Base Performance

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 2: Execution time comparison.

It is important to measure the performance of multiple-instruction-issue architectures using highly optimized code, because a naive compiler may produce redundant operations that show deceptive parallelism. To calibrate the quality of the code used in our experiments, we compare the execution time of code generated by IMPACT-I with code generated by a leading commercial compiler (MIPS CC release 2.1) and a leading public domain compiler (GNU CC release 1.37.1) on a DEC 3100 workstation.

Table 1 shows the benchmark programs that are used in this paper. Table 2 shows the execution time ratio of the MIPS CC (-O4) and the Gnu CC (-O) compilers over the IMPACT-I C (-O5). The quality of the code generated by the IMPACT-I C compiler is comparable to that of the MIPS C compiler which is known for its excellent code optimization capabilities. Therefore, the speedup numbers that we report for multiple-instruction-issue architectures in Section are based on very efficient sequential code.

2.3 Multiple-Instruction-Issue Code Optimization

The IMPACT-I C compiler performs several code transformations that enlarge the scope of static scheduling, including function inline expansion, instruction placement, loop unrolling, loop peeling, and branch expansion [Chang 90]. The compiler also performs several code transformations that reduce the depth of critical paths, including induction variable expansion, register renaming, global variable register allocation, operation combining, operation folding, and memory disambiguation [Chang 90].

2.4 Code Scheduling Algorithm

Prepass code scheduling is performed prior to register allocation to reduce the effect of artificial data dependencies that are introduced by register assignment [Hwu 88] [Goodman 88]. Postpass code scheduling is performed after register allocation.

Both prepass and postpass code scheduling algorithms consist of the following steps: 1) Form traces from basic blocks that are likely to be executed as a sequence. 2) Form a large *superblock* from each trace of basic blocks by code duplication. A superblock has a unique entry point, and one or more exit points. Basic blocks within a superblock are placed sequentially in memory. 3) Construct a dependence graph for each superblock. 4) Improve the dependence graph by removing dependence arcs that can be resolved at compile-time. 5) Compute live-variable information. For each branch path, live-variable information tells us what variables must not be destroyed when that branch path is taken. 6) Schedule the refined dependence graph according to machine constraints.

2.5 Code Scheduling Models

Our code scheduler moves code both upward and downward across branch operations within a superblock. With the exception of branch operations, the order of any two operations may be reversed if there are no data dependencies between the two operations. Let X and Y denote two operations where X is the operation to move and Y is a branch operation. Also, let *live-out(Y)* be the set of variables which may be used before defined when Y is taken.

For downward code motion, e.g., X precedes Y, if Y does not depend on X then X can be moved below Y. Note that if X is to be scheduled after Y and the destination register of X is in *live-out(Y)*, a copy of X must be inserted between Y and its target instruction.

For upward code motion there are two major restrictions.

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

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

For example, it is not safe to move a division operation above a branch because of the possibility of dividing by zero. As another example, it is not safe to move a memory load operation above a branch because it may cause a 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.*

The restricted code percolation model assumes that the processor supports conventional non-trapping instructions. It is possible to completely free the code scheduler from the second restriction if a comprehensive set of non-trapping instructions are available. We refer to this code scheduling model as *general code percolation.* Instead of trapping on divide by zero or illegal memory access, a garbage value is returned. There are two possible scenarios when an exception occurs. First, when the branch is taken Restriction 1 ensures that the garbage value will not be used. The second case is when the branch is not taken. Normally, the program would trap and halt on a divide by zero or memory access violation. In our execution model, the program

model	*Restricted*	*General*	*Speculative*
restrictions	1 and 2	1	none
hardware support	conventional	non-trapping instructions	shadow structures
page fault	handle as usual as soon as it occurs		
divide by zero	trap	ignore	trap if branch taken
access violation	trap	ignore	trap if branch taken

Table 3: Features of the code scheduling models.

fn	*base*	*MIPS-R3000*	*SPARC*	*i860*
integer alu	1	1	1	1
barrel shifter	1	1	1	1
integer mul	3	12	47	11
integer div	25	35	-	59
load	2	2	2	2
store	-	-	-	-
FP alu	3	6	10	3
FP conv	3	4	10	4
FP mul	4	6	12	5
FP div	25	12	64	38

Table 4: Operation latencies.

does not trap and thus the output will likely be incorrect. This is not wrong since the program is faulty. However, it does make the program more difficult to debug. As is done for many compiler optimizations, the programmer should debug the program without general code percolation.

Using aggressive hardware support, the first restriction can also be removed. Smith, Lam, and Horowitz have described such a scheme [Smith 90]. This scheme requires a shadow register file and a shadow store buffer. These shadow structures hold the results of percolated operation until the branch is committed. Once the direction of the branch is known, the percolated operations are either committed or nullified if the branch is not taken or taken respectively. In this scheme, exceptions are handled after a branch commits. We have implemented a scheduling method where operations can be freely moved above N branch operations in the same superblock, where N is a design parameter. We refer to this scheduling model as *speculative execution*.

A side effect of code percolation is that the number of page faults may increase. An instruction that is moved above a branch may cause a page fault. For a branch that is taken, this page fault does not occur before percolation and thus it degrades performance. Since most working sets can be kept in memory, it is unlikely to have a page fault on a legal instruction or data access. We suspect that a page fault will most likely occur for out-of-bound array references. We are currently studying the impact of page faults on the performance for code percolation scheduling.

Table 3 summarizes the features of the three static code scheduling models. In the next section, we show the relative performance of the three static code scheduling models.

2.6 Code Generation

A machine description file has been written to describe the instruction set, the microarchitecture, and the code scheduling model of each processor architecture under study. A code generator for this parameterized multiple-instruction-issue architecture has been implemented. The code generator performs profile-based branch prediction to support the squashing branch scheme [McFarling 86] [Chang 89.1].

3 Experiments

3.1 Evaluation Methodology

Using a profiler, we measure the execution count of every operation and collect branch statistics. Compile-time decisions are based on a composite of 20 profiles (i.e., a different input per profile). Using another input, we derive the best and the worst case execution time of each superblock, assuming ideal cache. The worst case is due to long operation latencies that protrude from one superblock to another superblock. For the benchmark programs used in this paper, the difference between the best case and the worst case execution time is always negligible. In the following discussion, we use the worst case execution time measure.

The experiment is to study the speedup of multiple-instruction-issue processors versus a single-instruction-issue processor for various scheduling models, memory load latencies, and function unit resource limitations. For the speculative execution model, instructions are scheduled within a superblock with N set to 32. We report the harmonic mean of the speedup numbers of all benchmarks[1].

3.2 Base Architecture

The base architecture is a single-instruction-issue processor that uses the general code percolation model. We have chosen an instruction set that is a superset of the MIPS instruction set to establish a strong single-instruction-issue base architecture. All function units are pipelined. The *base* column of Table 4 [2] shows the operation latencies. We assume in-order execution and deterministic operation

[1]To report results conservatively, the harmonic mean is used instead of the arithmetic mean.

[2]The integer multiplication and division latencies of the commercial processors are based on software implementation.

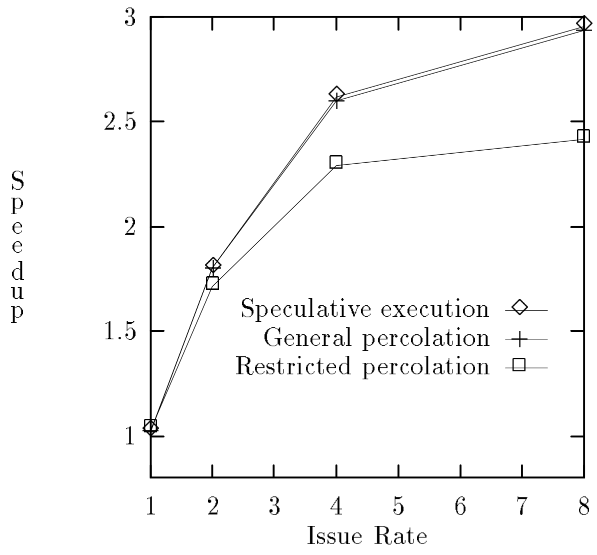

Figure 1: Comparison of scheduling models for load delay 1

latencies. Each processor includes a 64-entry integer register bank and a 32-entry floating-point register bank. The architecture uses a squashing branch scheme and profile-based branch prediction. One branch slot (one instruction) is automatically allocated for each instruction that contains a predict-taken branch operation.

Considering one cycle branch latency, the base architecture has achieved an execution rate of better than 0.95 operations per cycle for the benchmark programs.

3.3 Comparison Of The Three Scheduling Models

Figures 1 through 3 show the speedup of all three code scheduling models over the base architecture for issue rates from one to eight. By issue rate we mean that the processor can issue up to that many instructions per cycle. The graphs show the speedup when the memory load operation latency is one, two, and three cycles respectively. Except for the memory load latency, operation latencies are the same as that of the base architecture. No limitation has been placed on the function unit resources. Every operation code can be executed from every operation slot of an instruction.

The experimental results show that general code percolation and speculative execution substantially out-perform restricted code percolation. They also show that speculative execution consistently performs better than general code percolation, but the improvement is insignificant. This is true for all three memory latencies.

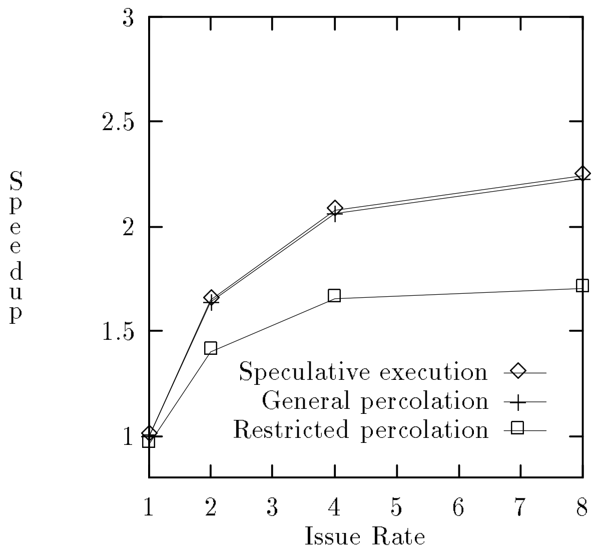

Figure 2: Comparison of scheduling models for load delay 2

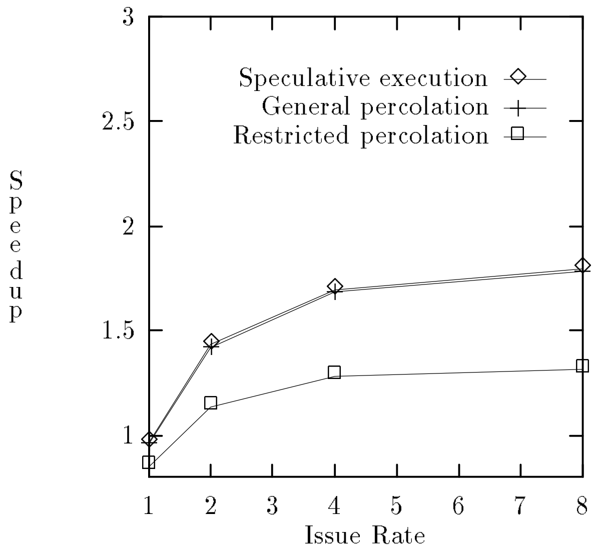

Figure 3: Comparison of scheduling models for load delay 3

3.4 Limited Resource

The cost to replicate all function units for each additional operation slot in the instruction format can be very high. Therefore, we have evaluated performance degradations due to limited function unit resources. The results are shown in Figures 4 and 5. These results motivate our architectural framework decisions in Section 4.

4 The IMPACT Architectural Framework

Based on the experimental results, in this section we identify a set of architectural features that best support the IMPACT-I C compiler to generate efficient code for multiple-instruction-issue processors.

4.1 Instruction Issue Rate

Figures 4 and 5 show that there is significant speedup as the issue rate increases from 2 to 4. After 4 instructions per cycle, the speedup starts to level off. For example, assuming load delay 1 and no resource constraints, the performance of a four-issue processor is 45% higher than a two-issue processor. However, an eight-issue processor only improves the performance by 12% over a four-issue processor. For load delays 2 and above, the performance improvement from a four-issue processor to an eight-issue processor is even less. For the current compiler technology and the benchmarks analyzed, the most efficient issue rate included in the IMPACT Architectural Framework is 4.

4.2 Limited Function Unit Resources

The experimental results in Figures 4 and 5 show the consequences of limiting the function unit resources. Limiting the number of stores per cycle to one does not significantly degrade the performance. However, if in addition the number of loads is limited to one per cycle the performance degrades significantly. Note that in cache design it is easier to provide multiple load ports than multiple store ports. Therefore, it is cost effective to support multiple memory loads per cycle.

Limiting the number of branches to one per cycle also significantly degrades the performance. Furthermore, when the branches are limited to one per cycle and there is at most one store or load per cycle, the performance of a four-issue machine approaches that of a two-issue machine.

The IMPACT-I C compiler is designed to support multiple branch operations per cycle. We have developed a variant of the squashing branch, called *inline target insertion* [Hwu 90] [Chang 89.1] which allows concurrent execution of branch operations. Furthermore, inline target insertion allows branch operations to be fetched from branch slots and independent of the length of the control unit pipeline, only one program counter needs to be saved in order to return from an interrupt. A new set of branches supporting

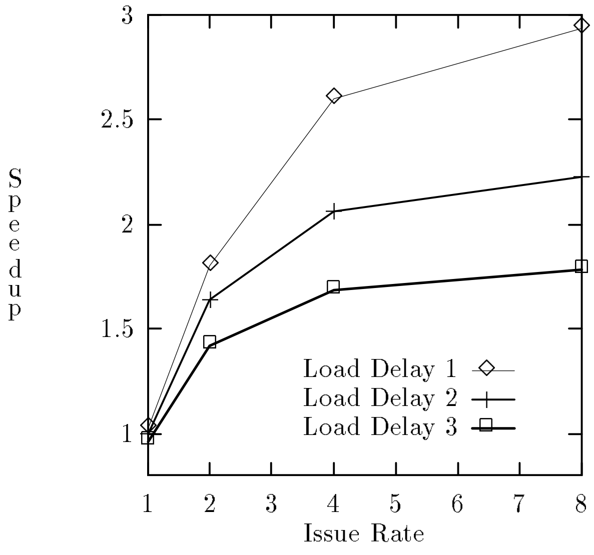

Figure 6: Comparison of load delays for general percolation

inline target insertion can be added to existing commercial architectures as an upward compatible feature.

4.3 Code Scheduling Model

The experimental results show that general code percolation significantly out-performs restricted code percolation. They also show that speculative execution achieves very little speedup beyond general code percolation. Considering the extra hardware overhead of supporting speculative execution, general code percolation is the most cost effective scheduling model in the IMPACT Architectural Framework The hardware support for this model includes disabling exceptions for non-trapping instructions. Many recent processors, such as MIPS-R2000, already support a set of arithmetic operations that do not signal overflow exception [Kane 87].

4.4 Memory Load Latency

For higher issue rates, the IMPACT-I C compiler can most effectively schedule instructions if the load delay is 1. For single-issue architectures there is a sufficient number of independent operations available to the scheduler to hide long memory load latencies. However, the demand for independent operations to schedule after a load grows as a multiple of the issue rate. As a result, for higher issue rates the limited supply of independent instructions can no longer hide a high memory load latency. This is clearly shown in Figure 6.

Most existing commercial RISC architectures assume load delay 2. Reducing load delay to 1 without stretching

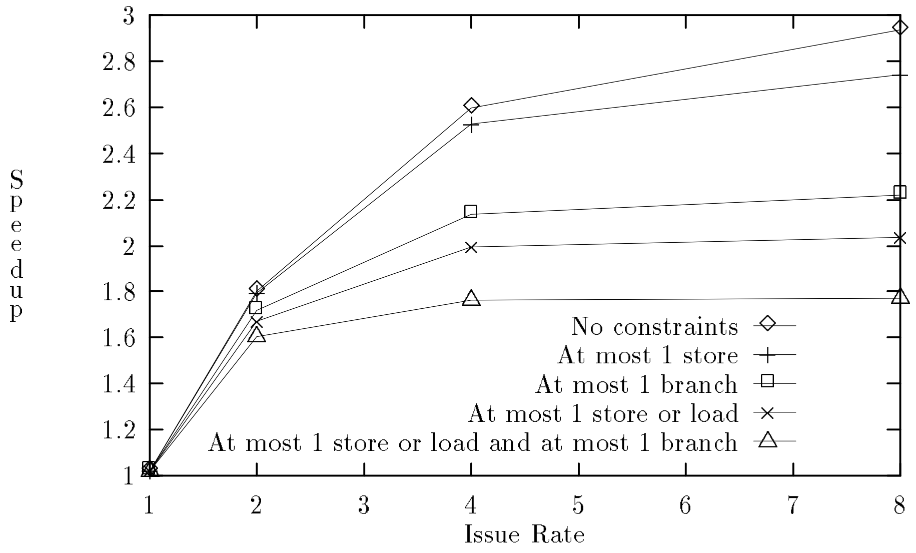

Figure 4: Effects of limited resources for load delay 1

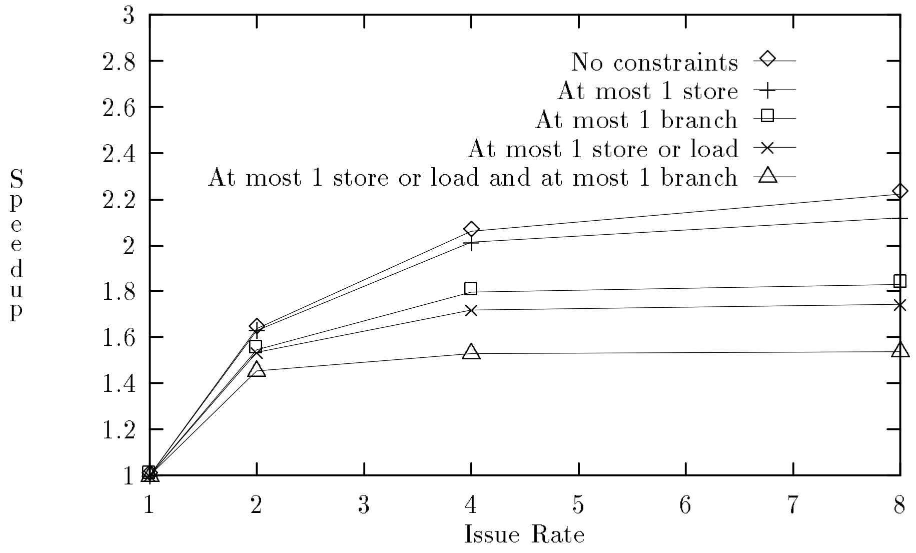

Figure 5: Effects of limited resources for load delay 2

cycle time poses difficult engineering problems. However, the performance payoff for high issue rate processors is significant. We are currently studying various techniques to achieve this goal.

5 Conclusion

Our approach to the design of multiple-instruction-issue processors can be summarized into three steps. In step one, we constructed IMPACT-I, a highly optimizing C compiler for multiple-instruction-issue processors. In step two, we conducted experiments to derive the IMPACT Architectural Framework. Multiple-instruction-issue processors designed within this framework receive effective compilation support from the IMPACT-I C compiler. In step three, we verified that multiple-instruction-issue processors designed within the IMPACT Architectural Framework have achieved solid speedup over a high-performance single-instruction-issue processor.

The IMPACT Architectural Framework consists of the following features. First, for scheduling general code percolation provides substantial performance improvement over restricted code percolation without the hardware cost of speculative execution. Second, as the instruction issue rate increases, the memory load delay becomes the major limiting factor of processor performance. Unlike single-instruction-issue architectures, decreasing the load delay from 2 to 1 cycles significantly improves performance in multiple-instruction-issue architectures. For example, reducing load delay from 2 to 1 cycles improves the performance of a four-issue processor by approximately 30%. Third, multiple-instruction-issue processors should have the ability to execute multiple branch and load operations in each cycle. For example, for a four-issue machine, limiting to one memory load and one branch per cycle reduces the performance by about 30%.

Future directions of the IMPACT Architectural Framework project include supporting multiple-instruction-issue implementations of major commercial architectures, enhancing the code optimization capabilities of the IMPACT-I C compiler, analyzing scientific applications, and extending the framework to support multiprocessor architectures. In addition, we are analyzing the effects of: code expansion due to optimizations on the instruction cache design, page faults on performance of percolation code scheduling, and register file size on the architectural framework design decisions.

Acknowledgements

The authors would like to thank Bob Horst, Andy Glew, Roland Ouelette, 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

[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.

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[Anantha 90] K. Anantha and F. Long, "Code Compaction for Parallel Architectures", Software Practice & Experience, vol.20, no.6, pp.537-554, June, 1990.

[Chaitin 82] G. J. Chaitin, "Register Allocation & Spilling Via Graph Coloring", ACM SIGPLAN Notice, vol.17-6, June 1982.

[Chang 88] 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.

[Chang 89.1] P. P. Chang and W. W. Hwu, "Forward Semantic: A Compiler-Assisted Instruction Fetch Method For Heavily Pipelined Processors", Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture, Dublin, Ireland, August, 1989.

[Chang 89.2] P. P. Chang and W. W. Hwu, "Control Flow Optimization for Supercomputer Scalar Processing", Proceedings, 1989 International Conference on Supercomputing, Crete, Greece, June 5-9, 1989.

[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.

[Chow 84] F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring", Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions, June, 1984.

[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.

[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.

[Goodman 88] 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.

[Hennessy 81] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, October 1981.

[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 88] W. W. Hwu and P. P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator", Proceedings, 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May, 1988.

[Hwu 89.1] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", Proceedings, 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, June, 1989.

[Hwu 89.2] 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 P. P. Chang, "Efficient Instruction Sequencing with Inline Target Insertion", Coordinated Science Laboratory Report, UILU-ENG-90-2215, CSG-123, May, 1990.

[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, ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, pp.318-327, Atlanta, Georgia, June, 1988.

[McFarling 86] S. McFarling and J. L. Hennessy, "Reducing the Cost of Branches", The 13th International Symposium on Computer Architecture Conference Proceedings, pp.396-403, Tokyo, Japan, June, 1986.

[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 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.

[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.