February 1991

UILU-ENG-91-2208
CRHC-91-4

Center for Reliable and High-Performance Computing

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

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2208 CRHC-91-4	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (*If applicable*) N/A
7a. NAME OF MONITORING ORGANIZATION NASA, National Science Foundation	
6c. ADDRESS (*City, State, and ZIP Code*) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (*City, State, and ZIP Code*) NASA Langley Res. Ctr., Hampton VA 23665 NSF 1800 G St. Washington DC 20552
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA and NSF	8b. OFFICE SYMBOL (*If applicable*)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NASA NAG 1-613 NSF: MIP-88-09478	
8c. ADDRESS (*City, State, and ZIP Code*) see 7b.	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

12. PERSONAL AUTHOR(S) Chang, Pohua P.; Mahlke, Scott A.; Chen, William; Warter, Nancy J.; Hwu, W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	91-02-05	38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Multiple-instruction issue processors, optimizing compiler code, scheduling code, transformation concurrency

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

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 is 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 have achieved solid speedup over a high-performance single-instruction-issue processor.

To address architecture design issues, we ran experiments to characterize the engineering tradeoffs such as the code scheduling model, the instruction issue rate, the memory load latency, and the 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.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

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

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

Center for Reliable and High-Performance Computing
University of Illinois
hwu@crhc.uiuc.edu
January 31, 1991

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 is 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 have achieved solid speedup over a high-performance single-instruction-issue processor.

To address architecture design issues, we ran experiments to charaterize the engineering tradeoffs such as the code scheduling model, the instruction issue rate, the memory load latency, and the 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 the computer. 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. Many hardware and software techniques for improving the performance and cost-effectiveness of 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].

1.1 Our Approach And Contribution

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, 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 such as alternative code scheduling models, 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.

The IMPACT-I C compiler generates highly optimized code for processors designed within the IMPACT Architectural Framework. Experimental results show that multiple-instruction-issue processors in this framework have achieved solid speedup over a high-performance single-instruction-issue processor. Supported by the advanced code optimization capabilities of the IMPACT-I C compiler, these multiple-instruction-issue processors have achieved a very encouraging performance level for non-numerical C programs.

1.2 Related Work

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] that issue several operations in a single cycle. 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]. Weiss and Smith have shown that loop unrolling and software pipelining are effective in increasing parallelism [Weiss 87]. Sohi has shown that restricted horizontal instruction formats are compa-

rable to a format that can issue every operation to every function unit[Sohi 89]. These studies have focused mainly on numerical kernels and applications. In this paper, we have dealt with non-numerical programs.

Patt and Hwu have described a single-chip microarchitecture which allows the compiler to schedule several operations into an instruction word, and also supports dynamic code scheduling and branch lookahead to further explore concurrencies between instructions [Patt 85]. This paper focuses on compile-time optimizations for simple in-order execution architectures.

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 latency, 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 code scheduling techniques, and have considered non-unit-time operation latencies and machine constraints.

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 for non-numerical *C* programs [Smith 89]. This paper focuses on the static code scheduling approaches.

Smith, Lam, and Horowitz have proposed and studied a static scheduling architecture that allows operations to be moved across a preceding branch operation [Smith 90]. They have reported about 1.63 speedup against a scalar processor which executes approximately 0.9 operations per cycle, using a four-issue microarchitecture. In this paper, we have implemented more levels of static scheduling support. With aggressive code transformation optimizations, we have achieved a higher level of performance.

1.3 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.

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 [Kane 87] and the SPARC [Sparc 87] processors. We are constructing code generators for the AMD29K [Amd], the IBM RS/6000 [IBM 90], and the i860 [Intel 89] 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 Code Optimizations

Code improving techniques in the IMPACT-I C compiler can be 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

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.

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

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 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) and a leading public domain compiler (GNU CC) on the DEC 3100 workstation.

Table 1 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 each benchmark program. Table 2 [1] shows the speedup of the IMPACT-I C compiler and the Gnu

[1]** GNU -O failed, reported speedup is that without optimization.

name	*MIPS -O3*	*GNU -O*	*IMPACT*
cccp	1.0	0.99	1.07
cmp	1.0	0.99	1.01
compress	1.0	0.77 **	1.00
eqn	1.0	0.98	1.07
eqntott	1.0	0.85	0.98
espresso	1.0	0.88	1.00
grep	1.0	0.86	1.01
lex	1.0	0.97	1.01
qsort	1.0	0.94	1.00
tbl	1.0	0.93	1.00
wc	1.0	0.83	1.04
yacc	1.0	0.62 **	0.96

Table 2: Code Quality.

CC (-O) over the MIPS CC (-O3).

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 the Section 3 are based on very efficient sequential code.

2.3 Code Generation For Multiple-Instruction-Issue Machine

2.3.1 Code Optimization Techniques

A code generator for a 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]. The IMPACT-I C compiler performs several code transformations that enlarge the scope of static scheduling, including function inline expansion, instruction

```
;; for (i=0; i<120; i++) {
;;   c[i] = max(a[i], b[i]);
;;   m = max(m, c[i]);
;; }
;;
    (1)  mov i, 0;
L0: (2)  load $P1, _a, i;       /* a[i], $P1 is the 1st parameter register */
    (3)  load $P2, _b, i;       /* b[i], $P2 is the 2nd parameter register */
    (4)  jsr _max;              /* max(a[i], b[i]), $P0 is the return register */
    (5)  store _c, i, $P0;      /* c[i] = max(a[i], b[i]) */
    (6)  load $P1, _m, 0;       /* m */
    (7)  mov, $P2, $P0;         /* c[i] */
    (8)  jsr _max;              /* max(m, c[i]) */
    (9)  store _m, 0, $P0;      /* m = max(m, c[i]) */
    (10) add i, i, 4;           /* i++ */
    (11) blt i, 120, L0;        /* i<120 */
L1:
```

Figure 1: A Simple Program

placement, super-block formation, loop unrolling, loop peeling, and branch expansion. 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 folding, and memory disambiguation.

Figure 1 illustrates a simple program that determines the maximum of array elements. This program is used in the following paragraphs to show the functionality of each code optimization.

Branch prediction: There are two unconditional jumps (subroutine calls) and one conditional branch in Figure 1. If the code segment is invoked exactly once, the profile information should indicate that the two subroutine calls (4, 8) have been executed 120 times, and the conditional branch (11) has been taken 119 times and not taken 1 time. For machines that support squashing branch, all three branches should be marked as likely branches and the branch slots should be filled

```
    (1)  mov i, 0;
L0: (2)  load $P1, _a, i;
    (3)  load $P2, _b, i;
    (4a) bge $P1, $P2, L2;   /* if !($P1>=$P2) { */
    (4b) mov $P0, $P2;       /*   $P0 = $P2;     */
    (4c) jump L3;            /* } else {         */
L2: (4d) mov $P0, $P1;       /*   $P0 = $P1;     */
L3: (5)  store _c, i, $P0;   /* }                */
    (6)  load $P1, _m, 0;
    (7)  mov $P2, $P0;
    (8a) bge $P1, $P2, L4;   /* if !($P1>=$P2) { */
    (8b) mov $P0, $P2;       /*   $P0 = $P2;     */
    (8c) jump L5;            /* } else {         */
L4: (8d) mov $P0, $P1;       /*   $P0 = $P1;     */
L5: (9)  store _m, 0, $P0;   /* }                */
    (10) add i, i, 4;
    (11) blt i, 120, L0;
L1:
```

Figure 2: After function inline expansion

by instructions from the target paths.

Function inline expansion: To perform static code scheduling in the inter-function (inter-procedure) level in Figure 1, the function (procedure) calls must be inline expanded. Figure 2 illustrates the result of function inline expansion.

Instruction placement: Although we have eliminated the two function calls from the simple program in Figure 1, the expanded program has several additional branches ($4a, 4c, 8a, 8c$). Instruction placement is an optimization that reorganizes code to increase sequential locality and to reduce the number of taken branches. The basic idea is to group basic blocks that tend to execute in a sequence together into a linear trace. We use the code segment in Figure 2 as an example. Assume that the elements in b[] are usually larger than the elements in a[]. The most

```
    (1)  mov i, 0;
L0: (2)  load $P1, _a, i;
    (3)  load $P2, _b, i;
    (4a) bge $P1, $P2, L2;    /* unlikely */
    (4b) mov $P0, $P2;
L3: (5)  store _c, i, $P0;
    (6)  load $P1, _m, 0;
    (7)  mov $P2, $P0;
    (8a) bge $P1, $P2, L4;    /* unlikely */
    (8b) mov $P0, $P2;
L5: (9)  store _m, 0, $P0;
    (10) add i, i, 4;
    (11) blt i, 120, L0;      /* likely */
L1:
    .....
L2: (4d) mov $P0, $P1;
    (4e) jump L3;             /* likely */
L4: (8d) mov $P0, $P1;
    (8e) jump L5;             /* likely */
```

Figure 3: After instruction placement

likely instruction sequence is ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4a \rightarrow 4b \rightarrow 4c \rightarrow 5$). Assume that the elements in the beginning of $a[]$ and $b[]$ tend to be bigger than the rest. Then, the most likely instruction sequence is ($6 \rightarrow 7 \rightarrow 8a \rightarrow 8b \rightarrow 8c \rightarrow 9$). The profiler can capture the sequencing information that is needed to guide the instruction placement optimization. The result of instruction placement is shown in Figure 3. Instruction placement forms groups of basic blocks that can be compacted by trace scheduling [Fisher 81]. Instruction (2) to (11) in Figure 3 forms a single trace. After instruction placement, the loop body contains no likely branch, except the loop back-edge. Note that ($4c$) and ($8c$) have been eliminated and ($4e$) and ($8e$) have been added to the program. Because ($4c$) and ($8c$) appear on frequently executed paths and ($4e$) and ($8e$) appear on infrequently executed paths, we can expect some speedup for single-instruction-issue architectures as well as for multiple-instruction-issue architectures.

Super-block formation: The flow-dependence arc ($4b \rightarrow 5$) can be eliminated if the result of ($4b$) can be copy propagated to (5). This is impossible because instruction ($4d$) modifies $\$P0$. We propose an optimization called super-block formation which allows traces to be customized. A super-block is a sequence of instructions that can be executed only from the top instruction and may contain multiple branch instructions. If a trace is not already a super-block, the trace can be converted to a super-block by creating a copy of the trace and redirecting all control transfers to the middle of the trace to the duplicate copy. From Figure 3, super-block formation produces Figure 4.

In Figure 4, label $L3$ and label $L5$ can be eliminated because all control transfers have been redirected to label $L6$ and label $L7$. More classical code optimizations can be applied to the super-block. The result is shown in Figure 5.

Loop unrolling: To increase the number of instructions in the super-block in Figure 5, we can unroll the loop N times by duplicating the loop body $(N - 1)$ times. Figure 6 shows the new super-block after loop unrolling ($N = 2$). For multiple-instruction-issue processors, the IMPACT-I C compiler typically unrolls small loops 8 or more times.

Loop peeling: Many loops iterate very few times (< 10) in the benchmark programs that we have studied (see Table 1). For these loops, loop unrolling and software pipelining are less effective because the execution time that is spent in the parallel section (optimized loop body) is not substantially longer than in the sequential section (loop prologue and epilogue). An alternative approach is to peel off enough iterations, such that the loop typically executes as a piece of straight-line code. Figure 7 shows the result of applying loop peeling ($N = 1$) on the code segment in Figure 5. Typically, the IMPACT-I C compiler peels off about 4 iterations. The peeled iterations

```
    (1)  mov i, 0;
L0: (2)  load $P1, _a, i;
    (3)  load $P2, _b, i;
    (4a) bge $P1, $P2, L2;    /* unlikely */
    (4b) mov $P0, $P2;
L3: (5)  store _c, i, $P0;
    (6)  load $P1, _m, 0;
    (7)  mov $P2, $P0;
    (8a) bge $P1, $P2, L4;    /* unlikely */
    (8b) mov $P0, $P2;
L5: (9)  store _m, 0, $P0;
    (10) add i, i, 4;
    (11) blt i, 120, L0;      /* likely */
L1:
    .....
L6: (5')  store _c, i, $P0;
    (6')  load $P1, _m, 0;
    (7')  mov $P2, $P0;
    (8a') bge $P1, $P2, L4;    /* unlikely */
    (8b') mov $P0, $P2;
L7: (9')  store _m, 0, $P0;
    (10') add i, i, 4;
    (11') blt i, 120, L0;      /* likely */
    (12)  jump L1;             /* likely */
    .....
L2: (4d) mov $P0, $P1;
    (4e) jump L6;             /* likely */
L4: (8d) mov $P0, $P1;
    (8e) jump L7;             /* likely */
```

Figure 4: After super-block formation

```
    (1)  mov i, 0;
L0: (2)  load $P1, _a, i;
    (3)  load $P2, _b, i;
    (4a) bge $P1, $P2, L2;    /* unlikely */
                              /* delete (4b) */
    (5)  store _c, i, $P2;
    (6)  load $P1, _m, 0;
                              /* delete (7) */
    (8a) bge $P1, $P2, L4;    /* unlikely */
                              /* delete (8b) */
    (9)  store _m, 0, $P2;
    (10) add i, i, 4;
    (11) blt i, 120, L0;      /* likely */
L1:
    .....
```

Figure 5: After super-block formation and classical code optimization

```
    (1)   mov i, 0;
L0: (2)   load $P1, _a, i;
    (3)   load $P2, _b, i;
    (4a)  bge $P1, $P2, L2;   /* unlikely */
    (5)   store _c, i, $P2;
    (6)   load $P1, _m, 0;
    (8a)  bge $P1, $P2, L4;   /* unlikely */
    (9)   store _m, 0, $P2;
    (10)  add i, i, 4;
    (11)  bge i, 120, L1;     /* unlikely */
    (2")  load $P1, _a, i;
    (3")  load $P2, _b, i;
    (4a") bge $P1, $P2, L2;   /* unlikely */
    (5")  store _c, i, $P2;
    (6")  load $P1, _m, 0;
    (8a") bge $P1, $P2, L4;   /* unlikely */
    (9")  store _m, 0, $P2;
    (10") add i, i, 4;
    (11") blt i, 120, L0;     /* likely */
L1:
    .....
```

Figure 6: After loop unrolling

```
     (1)   mov i, 0;
     (2")  load $P1, _a, 0;
     (3")  load $P2, _b, 0;
     (4a") bge $P1, $P2, L2;    /* unlikely */
     (5")  store _c, 0, $P2;
     (6")  load $P1, _m, 0;
     (8a") bge $P1, $P2, L4;    /* unlikely */
     (9")  store _m, 0, $P2;
     (10") mov i, 4;
L0:  (2)   load $P1, _a, i;
     (3)   load $P2, _b, i;
     (4a)  bge $P1, $P2, L2;    /* unlikely */
     (5)   store _c, i, $P2;
     (6)   load $P1, _m, 0;
     (8a)  bge $P1, $P2, L4;    /* unlikely */
     (9)   store _m, 0, $P2;
     (10)  add i, i, 4;
     (11)  blt i, 120, L0;      /* unlikely */
L1:
     .....
```

Figure 7: After loop peeling

```
     .....
L2:  (4d)  mov $P0, $P1;
     (5')  store _c, i, $P0;
     (6')  load $P1, _m, 0;
     (7')  mov $P2, $P0;
     (8a') bge $P1, $P2, L4;    /* unlikely */
     (8b') mov $P0, $P2;
     (9')  store _m, 0, $P0;
     (10') add i, i, 4;
     (11') blt i, 120, L0;      /* likely */
     (12)  jump L1;             /* likely */
     .....
L4:  (8d)  mov $P0, $P1;
     (9")  store _m, 0, $P0;
     (10") add i, i, 4;
     (11") blt i, 120, L0;      /* likely */
     (12") jump L1;             /* likely */
```

Figure 8: After branch expansion

can usually be further optimized by classical code optimizations, e.g. copy propagation.

Branch expansion: Instruction placement and super-block formation introduce many spurious branch instructions (12), (4e), and (8e) in Figure 4. In realistic programs where there are many nested $if-then-else$ statements, the situation is much worst than in Figure 4. Branch expansion helps to eliminate some branch instructions by replacing them by their target basic blocks. The number of static instructions increases due to this optimization. But the number of dynamic (executed) instructions decreases. Figure 8 shows the result of branch expansion. Note that two super-blocks ($L2, L4$) are developed by this optimization.

Induction variable expansion: After loop unrolling (see Figure 6), induction variables and variables that are used as accumulators are often the source of anti-dependencies and output-

```
    (1)   mov i, 0;
L0: (2)   add i2, i, 4;       /* rename i to i2 for the 2nd iteration */
    (3)   load $P1, _a, i;
    (4)   load $P2, _b, i;
    (5)   bge $P1, $P2, L2;
    (6)   store _c, i, $P2;
    (7)   load $P1, _m, 0;
    (8)   bge $P1, $P2, L4;
    (9)   store _m, 0, $P2;
    (10)  add i, i, 4;            /* replace i by i2 after here */
    (11)  bge i2, 120, L1;
    (12)  load $P1, _a, i2;
    (13)  load $P2, _b, i2;
    (14)  bge $P1, $P2, L2;
    (15)  store _c, i2, $P2;
    (16)  load $P1, _m, 0;
    (17)  bge $P1, $P2, L4;
    (18)  store _m, 0, $P2;
    (19)  add i, i2, 4;
    (20)  blt i, 120, L0;
L1:
    .....
```

Figure 9: After induction variable expansion

dependencies across iterations. In order to overlap the execution of different iterations, it is necessary to rename these induction variables. Unfortunately, register renaming cannot be applied because an induction variable is alive in the loop. Induction variable expansion is designed specifically to rename induction variables. Figure 9 shows the result of induction variable expansion. For a loop which is unrolled N times, $(N-1)$ temporary variables are introduced to represent the induction variable in iteration 2..N. In Figure 9, instruction (3) to (9) use the original induction variable. Instruction (2) creates a new induction for the second loop iteration. Instruction (11) to (18) use the new induction variable. Note that instruction (10) and (19) update the original induction variable to enforce consistency.

Register renaming: Register renaming needs to be applied to the code segment in Figure 9 in order to overlap the execution of different loop iterations. The result is shown in Figure 10. $\$P1$ in instruction (3) is renamed to *temp*1 and $\$P2$ in instruction (4) is renamed to *temp*2. Subsequent uses of $\$P1$, prior to a new definition of $\$P1$, are also renamed to *temp*1. When $\$P1$ is redefined in instruction (12), it is renamed to a new temporary variable *temp*5. When a register is alive in the taken path of a branch, a bookkeep instruction needs to be inserted between the branch and the target instruction to undo the effect of register renaming. For example, $\$P1$ is used before defined in $L2$ super-block. Therefore, instruction (5) must first branch to $L2'$. In $L2'$, $\$P1$ is updated before jumping to $L2$.

Note that the bookkeep cost can be reduced with branch expansion and classical code optimization by forming $(L2' \rightarrow L2)$, $(L2" \rightarrow L2)$, $(L4' \rightarrow L4)$, and $(L4" \rightarrow L4)$ super-blocks. Repeated applications of branch expansion and classical code optimization increase the static program size and decrease the number of dynamic (executed) instructions. However, increasing program size

```
    (1)   mov i, 0;
L0: (2)   add i2, i, 4;
    (3)   load temp1, _a, i;
    (4)   load temp2, _b, i;
    (5)   bge temp1, temp2, L2';   /* $P1 is live in L2 */
    (6)   store _c, i, temp2;
    (7)   load temp3, _m, 0;
    (8)   bge temp3, temp2, L4';   /* $P1 is live in L4 */
    (9)   store _m, 0, temp2;
    (10)  add i, i, 4;
    (11)  bge i2, 120, L1;
    (12)  load temp5, _a, i2;
    (13)  load temp6, _b, i2;
    (14)  bge temp5, temp6, L2";   /* $P1 is live in L2 */
    (15)  store _c, i2, temp6;
    (16)  load temp7, _m, 0;
    (17)  bge temp7, temp6, L4";   /* $P1 is live in L4 */
    (18)  store _m, 0, temp6;
    (19)  add i, i2, 4;
    (20)  blt i, 120, L0;
L1:
    .....
    .....
L2': (100) mov $P1, temp1;
     (101) jump L2;
L2": (102) mov $P1, temp5;
     (103) jump L2;
L4': (104) mov $P1, temp3;
     (105) jump L4;
L4": (106) mov $P1, temp7;
     (107) jump L4;
    .....
```

Figure 10: After register renaming

```
(19)  add i, i2, 4;     -->   (19')  add i, i2, 4;
(20)  blt i, 120, L0;   -->   (20')  blt i2, 116, L0;
```

Figure 11: After operation folding

may degrade instruction cache performance. The tradeoff between optimizing for efficiency and optimizing for instruction cache performance is not clear.

Operation folding: Operation folding eliminates flow-dependencies between instructions by observing special instruction patterns of the form $f(g(src_g), src_f)$ which can be transformed to $(g(src_g), f'(src_g, src_f))$. For example, the flow-dependence between instruction (19) and (20) in Figure 10 can be eliminated as shown in Figure 11.

Global variable register allocation: Global variable register allocation identifies all memory accesses to a specific memory location in a loop, replaces all memory accesses by register accesses, and adds appropriate code to load and restore the memory location in the loop prologue and epilogue sections. Some flow-dependencies that are due to memory load and store instructions can be eliminated by global variable register allocation. In Figure 10, all memory accesses to $\_m$ can be converted to register accesses. Figure 11 shows the result of global variable register allocation for the most important super-block of this example. Note that after global variable register allocation, there are more opportunities for register renaming.

Memory disambiguation: Memory disambiguation does not by itself perform code transformation. Its duty is to discover optimization opportunities for other code optimizations, such as global variable register allocation and code scheduling. For the C programming language, memory disambiguation is very difficult due to pointers.

```
     (1)   mov i, 0;
     (100) load tm, _m, 0;          /* tm = mem[_m] */
L0:  (2)   add i2, i, 4;
     (3)   load temp1, _a, i;
     (4)   load temp2, _b, i;
     (5)   bge temp1, temp2, L2';
     (6)   store _c, i, temp2;
     (7)   mov temp3, tm;
     (8)   bge tm, temp2, L4';
     (9)   mov tm, temp2;
     (10)  add i, i, 4;
     (11)  bge i2, 120, L1;
     (12)  load temp5, _a, i2;
     (13)  load temp6, _b, i2;
     (14)  bge temp5, temp6, L2";
     (15)  store _c, i2, temp6;
     (16)  mov temp7, temp2;
     (17)  bge temp2, temp6, L4";
     (18)  mov tm, temp6;
     (19)  add i, i2, 4;
     (20)  blt i2, 116, L0;
L1:  (101) store _m, 0, tm;         /* mem[_m] = tm */
     .....
```

Figure 12: After global variable register allocation

2.3.2 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 super-block from each trace of basic blocks by code duplication. 3) Construct a dependence graph for each super-block. 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.

We have shown how super-blocks are formed in the previous subsection (Figure 1 - 12).

The construction of a dependence graph from a super-block is very straight forward by observing every pair of instructions in the super-block. There are three major categories of dependencies: data dependencies, control dependencies, and synchronization dependencies. Data dependencies can be further partitioned to six different types: register flow-dependence, register anti-dependence, register output-dependence, memory flow-dependence, memory anti-dependence, and memory output-dependence. For example, instruction (5) in Figure 12 is register flow-dependent on instruction (3) because instruction (5) uses the result of instruction (3). Instruction (9) is register anti-dependent on instruction (8) because instruction (9) modifies a source register of instruction (8). Instruction (19) is register output-dependent on instruction (10) because they modify the same register. Without memory disambiguation, we have to assume that all memory load and store instructions

may access the same location. For example, instruction (12) is memory flow-dependent on instruction (6). Instruction (6) is memory anti-dependent on instruction (3). Instruction (15) is memory output-dependent on instruction (6). Control dependencies exist between a branch instruction and any other instruction in the super-block. For example, instruction (5) is control-dependent on instruction (4). Instruction (6) is control-dependent on instruction (5). Dependencies due to synchronization instructions are beyond the scope of this paper and will not be discussed further.

After a dependence graph is constructed for a super-block, some memory dependencies can be eliminated by memory disambiguation.

Dataflow analysis[Aho 86] is applied to a super-block to determine for each branch what variable values must not be destroyed when that branch is taken. Let *live-out(x)* denote the set of variables which may be used before defined when a branch x is taken. The result of dataflow analysis on Figure 12 is shown in Figure 13.

2.3.3 Code Scheduling Models

Our code scheduler moves code both upward and downward across branch operations in a super-block. If not because of branch operations, the order of any two operations may be reversed if there is no data dependencies between the two operations. Let X and Y denote two operations in the same super-block and X precedes Y in the original code. If X and Y are both branch operations, their order may not be changed, except when they branch to the same location and are consecutive in the super-block. For simplicity, a static code scheduler typically does not schedule a branch before another preceding branch. If Y is a branch and X is not a branch, then the static code scheduler is allowed to schedule Y ahead of X. If Y is to be scheduled ahead of X and the destination register of X is in *live-out(Y)*, then a copy of X must be inserted between Y and its

```
    (1)   mov i, 0;
    (100) load tm, _m, 0;
L0: (2)   add i2, i, 4;
    (3)   load temp1, _a, i;
    (4)   load temp2, _b, i;
    (5)   bge temp1, temp2, L2';  /* live-out(5) = {temp1, tm, i} */
    (6)   store _c, i, temp2;
    (7)   mov temp3, tm;
    (8)   bge tm, temp2, L4';     /* live-out(8) = {temp3, tm, i} */
    (9)   mov tm, temp2;
    (10)  add i, i, 4;
    (11)  bge i2, 120, L1;        /* live-out(11) = {tm, i} */
    (12)  load temp5, _a, i2;
    (13)  load temp6, _b, i2;
    (14)  bge temp5, temp6, L2";  /* live-out(14) = {temp5, tm, i} */
    (15)  store _c, i2, temp6;
    (16)  mov temp7, temp2;
    (17)  bge temp2, temp6, L4";  /* live-out(17) = {temp7, tm, i} */
    (18)  mov tm, temp6;
    (19)  add i, i2, 4;
    (20)  blt i2, 116, L0;        /* live-out(20) = {tm, i} */
L1: (101) store _m, 0, tm;
    .....
```

Figure 13: After dataflow analysis

target instruction. 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. If X is a branch and Y is not a branch, then the static code scheduler is allowed to schedule Y ahead of X if the following two restrictions are not violated.

Restriction 1: Y is not in *live-out(X)*.

Restriction 2: Y 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 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 in the usual manner. We refer to this code scheduling model as *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 squashes percolated operations if the branch direction is incorrectly predicted. We have implemented a scheduling method where operations can be freely moved above N branch operations in the same super-block, where N is a design parameter. We refer to this scheduling model as *speculative execution.*

In Section 3, we show the relative performance of the three static code scheduling models. We set N to 32 for the speculative execution model.

```
time operations [precedence is from left to right]
1    (2)(3)(4)(7)
2    (5)(6)(8)(9)(10)(11)(12)(13)(16)
3    (14)(15)(17)(18)(19)(20)
```

Figure 14: After code scheduling (restricted code percolation)

```
time operations [precedence is from left to right]
1    (2)(3)(4)(7) (12)(13)
2    (5)(6)(8)(9)(10)(11) (14)(15)(16)(17)(18)(19)(20)
```

Figure 15: After code scheduling (general code percolation)

Assuming that all operation latencies are unit time and we have infinite computation resource, we apply static code scheduling to the code segment in Figure 13 using aforementioned code scheduling models. There is an implicit linear precedence order among operations that are issued concurrently. We assume that a taken branch can squash all lower precedence operations that are issued concurrently with the taken branch. A legal schedule using the restricted code percolation model is shown in Figure 14. Each row gives the operations that may be issued concurrently in a single cycle. A legal schedule using the general code percolation model is shown in Figure 15. A legal schedule using the speculative execution model is shown in Figure 16.

```
* = speculative execution
time operations [precedence is from left to right]
1    (2)(3)(4)(7) (10*) (12)(13) (19*)
2    (5)(6)(8)(9)(11) (14)(15)(16)(17)(18) (20)
```

Figure 16: After code scheduling (speculative execution)

3 Experiments

3.1 Evaluation Methodology

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. The machine description file is used to guide the IMPACT-I C compiler to optimize each benchmark program for each processor architecture. Using a profiler, we measure the execution count of every operation and collect branch statistics. From the profile information, we can derive the best and the worst case execution time of each super-block, assuming ideal cache. The worst case is due to long operation latencies that protrude from one super-block to another super-block. 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. The experiment produces a total of $(X * Y)$ numbers, where X is the number of processor configurations under study, and Y is the number of benchmark programs. Let $cycle(i, j)$ denote a function that returns the number of cycles to execute the benchmark program j on the machine i. Let $cycle(1, j)$ denote the number of cycles to execute the benchmark j on the base architecture. We define the $speedup(i)$ function as the harmonic [2] mean of $(cycle(1, .)/cycle(i, .))$ over all benchmarks.

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

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 3: Operation latencies.

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 3 [3] shows the operation latencies. We assume in-order execution and deterministic operation 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]The integer multiplication and division latencies of the commercial processors are based on software implementation.

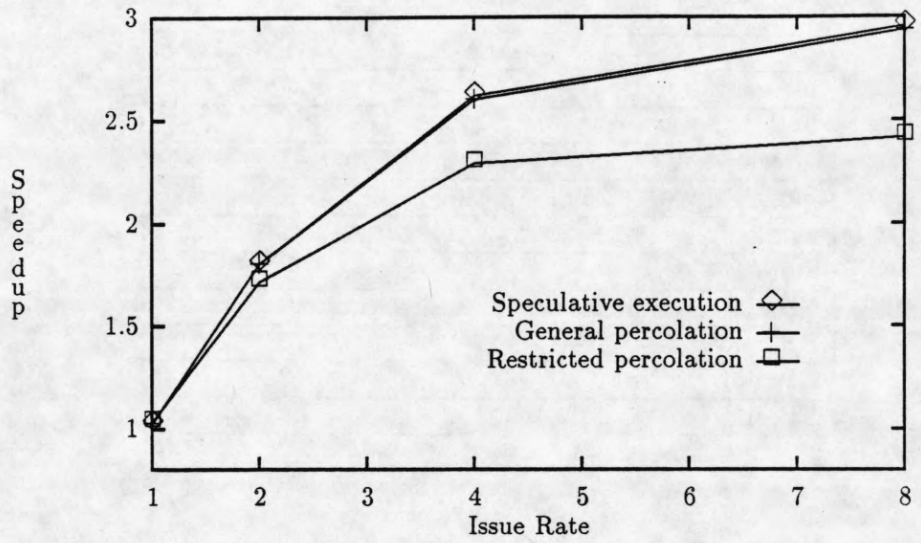

Figure 17: Comparison of Scheduling Models for Load Delay 1

3.3 Comparison Of The Three Scheduling Models

Figures 17 through 19 show the speedup of all three code scheduling models over the base architecture for issue rates from one to eight. 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.

The experimental results indicate that increasing the memory load operation latency substantially degrades the performance of multiple-instruction-issue architectures. This degradation is

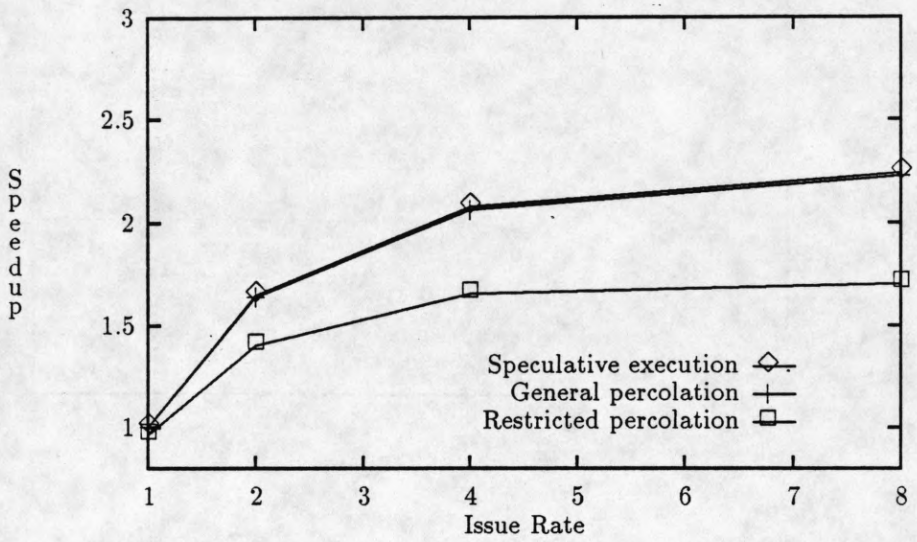

Figure 18: Comparison of Scheduling Models for Load Delay 2

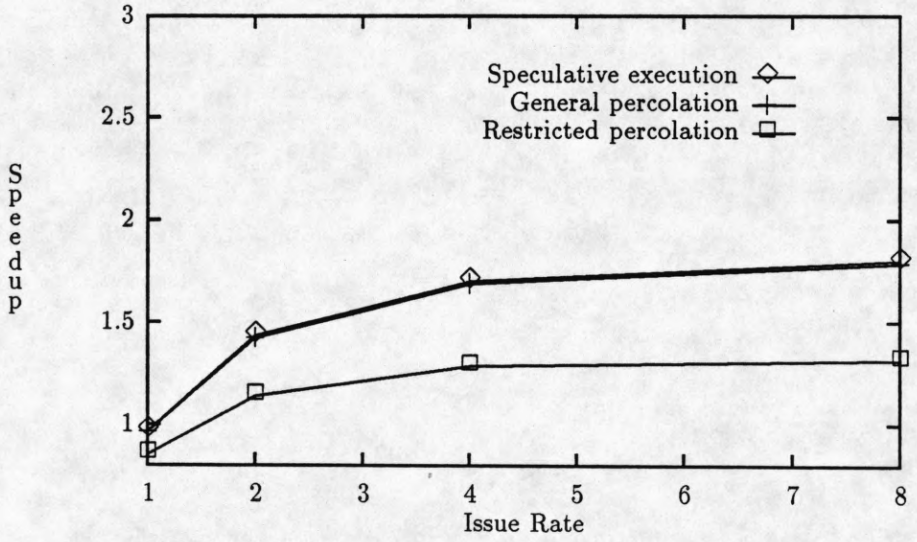

Figure 19: Comparison of Scheduling Models for Load Delay 3

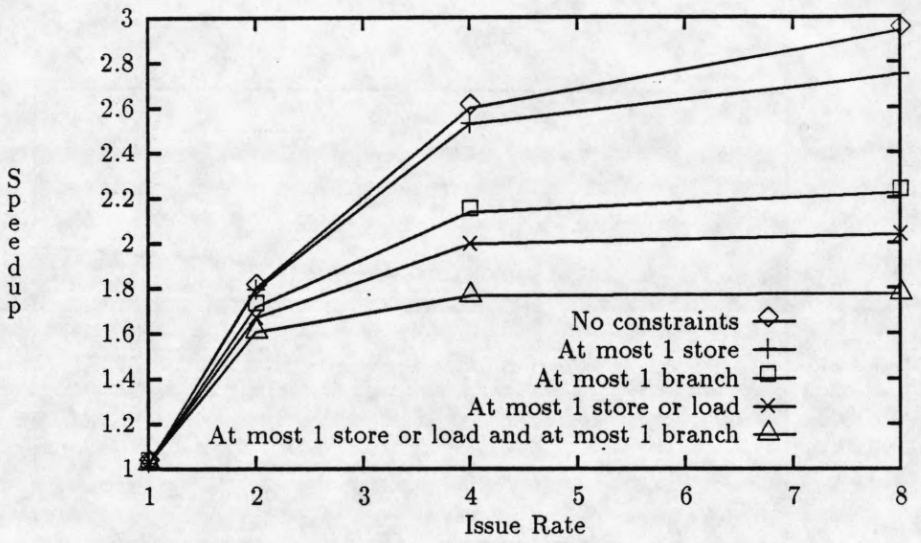

Figure 20: Effects of Limited Resources for Load Delay 1

most pronounced for high issue rates.

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 20 and 21.

The experimental results indicate that the ability to issue multiple branch and memory load operations per cycle is important for high issue rate architectures.

4 The IMPACT Architectural Framework

Figure 22 shows a high-level block diagram of the IMPACT Architectural Framework. In the ideal case, instructions are fetched and decoded every cycle. The control unit issues instructions to the function units in the order they are fetched.

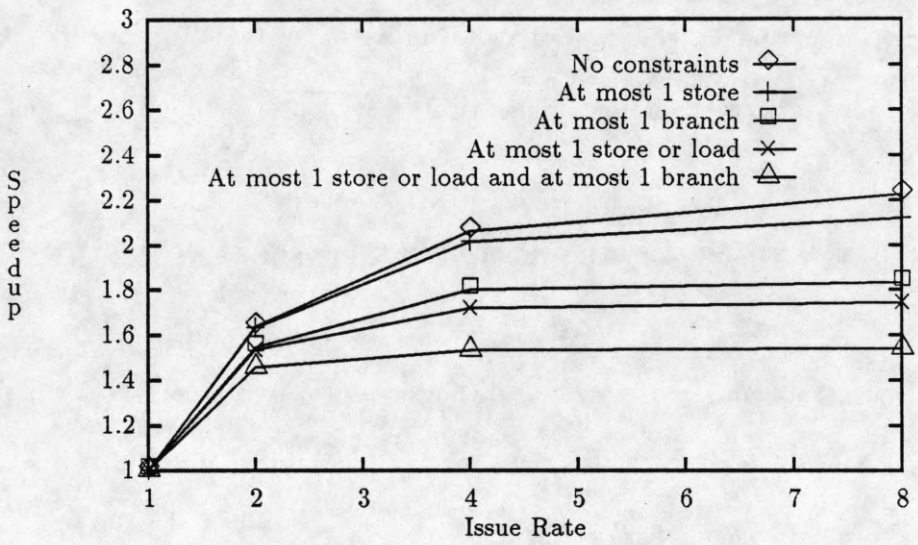

Figure 21: Effects of Limited Resources for Load Delay 2

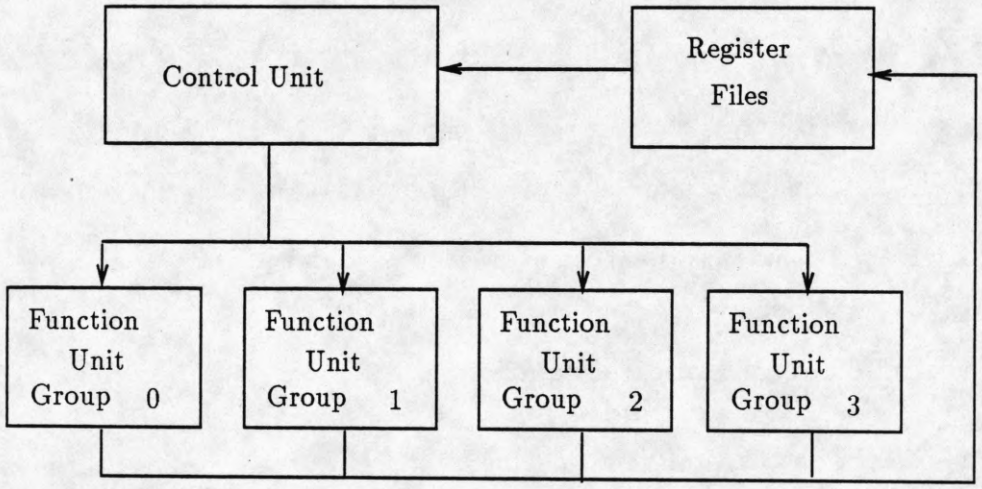

Figure 22: Block diagram of the IMPACT Architectural Framework.

4.1 Instruction Issue Rate

The number of operations that can be packed into an instruction is an architectural parameter. We have developed a variant of squashing branch, called *inline target insertion* [Hwu 90] [Chang 89.1], that uses profile-based branch prediction. Inline target insertion allows multiple branch operations to be issued per cycle, and allows branch operations to be fetched from branch slots. Independent of the length of the control unit pipeline, only one program counter needs to be saved in order to return from interrupt.

The ability to execute multiple branch operations per cycle is important for high issue rate architectures. Without this ability, the execution rate of four-issue architectures will be limited to below two times speedup over the base architecture.

4.2 Limited Function Unit Resources

The decoded instructions are forwarded directly to several independent function units. Figure 22 shows four function units for an issue rate of four operations per cycle. Each function unit can be as simple as an integer ALU, or as complex as a composite of a cache interface, a floating-point ALU, an integer ALU, and branch logic.

The experimental results in Figures 20 and 21 indicate that the ability to execute multiple branch and memory load operations is important for high issue rate architectures.

4.3 Support For General Code Percolation

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. Therefore, multiple-instruction-issue architectures should support the

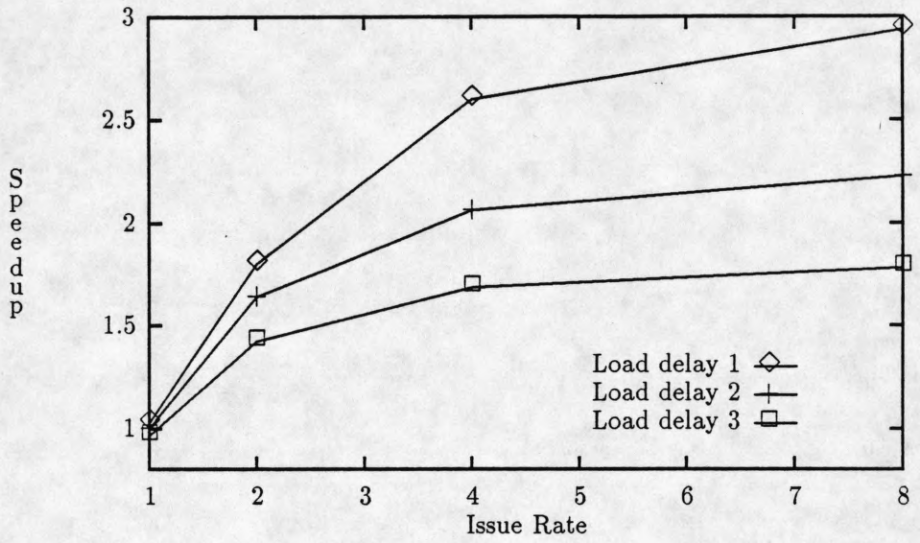

Figure 23: Comparison of Load Delays for General Percolation

general code percolation model. The hardware support includes disabling exceptions that are caused by divide by zero and by illegal memory accesses. Some recent processors already support a set of arithmetic operations that do not signal overflow exception [Kane 87] [Sparc 87] [Amd] [IBM 90] [Intel 89] .

4.4 Memory Load Latency

The experimental results in Figure 23 clearly indicate that increasing the memory load latency substantially degrades the performance of high issue rate architectures. Memory load operations frequently appear on critical paths. As instruction issue bandwidth increases, critical paths become more visible. Therefore, we strongly recommend a short memory load operation latency for multiple-instruction-issue machines.

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 ran experiments to verify 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-I C compiler has code generators for several major commercial architectures either available or under construction. Within the IMPACT Architectural Framework, multiple-instruction-issue implementations of these architectures receive immediate compilation support from the IMPACT-I C compiler. This would significantly reduce the uncertainties and misconceptions about software during hardware development. Furthermore, processors designed within the framework would have compiler support to deliver promised performance in their production use.

In terms of engineering tradeoffs, we recommend the following. First, general code percolation should be supported as the code scheduling model for multiple-instruction-issue processors. Second, high issue rate processors should support short memory load latency. Third, high issue rate processors should have the ability to execute multiple branch and load operations in each cycle. All these features can be incorporated into existing commercial architectures in an upward compatible manner.

Future directions of the IMPACT Architectural Framework project include supporting multiple-instruction-issue implementations of major commercial architectures, enhancing the code optimiza-

tion capabilities of the IMPACT-I C compiler, and extending the framework to support multiprocessor architectures.

Acknowledgements The authors would like to thank 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, a donation from NCR, 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.

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

[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.

[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.

[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.

[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.

[IBM 90] IBM, Special Issue on IBM RISC System/6000 Processor, IBM Journal of Research and Development, Volume 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.

[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.

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