AN OPTIMIZING COMPILER CODE GENERATOR: A PLATFORM FOR RISC PERFORMANCE ANALYSIS

BY

WILLIAM YU-WEI CHEN, JR.

B.S., The Ohio State University, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Wen-mei Hwu, for providing constant guidance and support. His patience was invaluable. I would also like to thank Sabrina Hwu for taking care of me for the past few years.

I wish to thank Pohua Chang for his constant suggestion and help. This thesis would not be possible without his work. Scott Mahlke and I have complemented each other's work during this time, and his cooperation in this project is very helpful. Thomas Conte, Nancy Warter, Sadun Anik and other members of the research group were always there to provide me with their insight and knowledge to further my study.

I wish to thank my friends here at the University of Illinois, especially Jaushin Lee, for their friendship and support.

Finally, I would like to thank my parents, William and Pauline, for their love and support. They have taught me to think before acting, which made my conviction toward attending graduate school much stronger.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

In the process of designing an architecture, the support of a good compiler cannot be ignored. The IMPACT-I C compiler generates a highly optimized code for processors designed within the IMPACT Architectural Framework. The IMPACT-I C compiler is retargetable, and can easily integrate with other optimizers and lower-level tools. This integration is possible through LCODE, the IMPACT-I low-level intermediate code.

The performance implication of the IMPACT-I C compiler is not known until it can be compared with a leading commercial compiler. Also, the worth of a compiler is judged by its ability to generate executable code. The construction of a code generator for the DEC 3100 Workstation satisfies both conditions. The IMPCC/MIPS code generator allows the performance comparison of the IMPACT-I C compiler with that of a leading commercial C compiler, MIPS C compiler. This comparison provides other researchers a basic understanding in the base technology for our architectural studies.

In the environment of an university, it is usually too costly and impractical to perform architectural studies by building the actual hardware; the studies are typically carried out by simulations. For our architectural studies, we constructed two low-level supporting tools: an LCODE profiler and an LCODE trace generator. These two tools provide accurate and efficient ways of interpreting LCODE instruction level information. With an LCODE profiler, accurate instruction count and accurate branch flow directions are measured. With an LCODE trace generator, accurate instruction execution sequence is generated. Both the LCODE profiler and LCODE trace generator conform to a standard interface and are built in addition to the IMPCC/MIPS code generator. Thus, by providing a trace analyzer which conforms to the standard interface, the users aquire immediate support from these two low-level LCODE supporting tools to perform various types of RISC architecture studies.

1.2 Outline of the Thesis

In Chapter 2, the general overview of the IMPCC/MIPS code generator is discussed. The main focus is on how to generate correct and fast execution code. Chapter 3 gives the modification of the IMPCC/MIPS code generator to create useful low-level supporting tools for architectural studies. The use of these low-level supporting tools is shown in Chapter 4. The comparison of static and dynamic code scheduling on a multiple-instruction-issue processor is used as an example. Chapter 5 gives the concluding remarks.

2. THE IMPCC/MIPS CODE GENERATOR

2.1 LCODE Environment

LCODE [1] (low-level intermediate code) is a register transfer language designed to provide an easy interface between high- and low-level languages. All low-level supporting tools are implemented in addition to LCODE, and these tools include the LCODE code optimizer, machine-dependent code generators, LCODE profilers, and LCODE trace generators. By following the LCODE interfacing rules, all high level languages are automatically supported by these low-level tools that are either available or under development. Figure 2.1 provides an overview as to where LCODE fits in the whole IMPACT-I C compiler structure.

LCODE is divided into two distinct sections: data declaration and function definition. Each instruction of LCODE is presented in a Lisp-like format, and two sets of data structures are created to store the LCODE instructions. The first data structure contains the data section, and the second data structure contains the function section. One data declaration instruction or one function are read in at a time.

[1]LCODE documentation is available as an internal report.

Figure 2.1: LCODE within IMPACT-I C compiler.

2.2 MIPS Convention Overview

In order to maximize the performance of a code generator, the MIPS architecture must be carefully studied. The information presented in this section is published by Kane [1] and MIPS Computer Systems [2]. These books should be read for a more detailed explanation. The MIPS memory system is divided into the text and the data sections. The data section is further divided into several smaller sections, each with a special purpose. For simplicity, Figure 2.2 provides a breakdown of the MIPS program memory that is utilized by the IMPCC/MIPS code generator. The text memory is for storing the user program instructions. All instructions are aligned on the boundary of four bytes. The data memory stores global variable values. The 32-bit addressing mode is used for accessing the variable values in the data memory. The small data memory has only 64k of space and is also used for storing global variable values. The small data

Figure 2.2: Program memory divisions.

memory uses the 16-bit addressing mode. Thus, each value within the small data memory can be accessed using only one instruction, where the 16-bit address offset is embedded in the immediate value part of the load/store instruction. The base address for the small data section is contained in *gp*, the global pointer register. On the other hand, the values within the data memory require two instructions to access. The third data section is the stack memory. All information local to the currently executed function is stored on the stack. For the IMPCC/MIPS code generator, the stack is used for storing parameters to be passed, local variables, caller saved registers, callee saved registers, returned structure values, and all spilled register contents. The current stack address is contained in *sp*, the stack pointer register. All contents above the stack pointer are assumed to be safe, and cannot be changed by other processes.

In order to be compatible with the system libraries or other user written assembly code, the calling convention must be coherent. There are several major constraints.

1. Stack layout must be the same.

2. Parameter passing methods must be coherent.

3. Register usage must be consistent.

The MIPS architecture keeps two pointers with respect to the stack; one is the stack pointer and the other one is the frame pointer. All stack accesses are in reference to these two pointers. The positive difference between the current stack pointer and the current frame pointer is the frame size of the current function. The space immediately above the current stack pointer location contains the parameters to be passed, if any, for the function to be called. At least 16 bytes must be allocated on the stack for these parameters that are passed in registers. The space immediately below the current frame pointer is the local variable location. The space immediately above the current frame pointer is for the passing parameters of the current function. All other space on the stack can be used freely.

The MIPS register set and its usage are presented in Tables 2.1 and 2.2. Four integer registers and two double floating-point registers are used for passing registers, but at most, four words (each integer register is a word, and each double floating-point register is two words) can be passed in register for each function call. These register numbers are $4, $5, $6, and $7 for the integer registers, and $f12, and $f14 for the double floating-point registers. Table 2.3 lists some possible parameter passing situations. The notations i and f indicate integer or floating-point parameter type. The number following i or f indicates the occurrence of that parameter type. The algorithm for passing parameters in registers is presented in the following rules.

Table 2.1: General register set.

Register	*Usage*
\$0	Constant value of zero
\$1	Reserved and not usable
\$2	Return register for integer values
\$3	Caller saved register
\$4..\$7	Integer parameter passing registers
\$8..\$15	Caller saved registers
\$16..\$23	Callee saved registers
\$24..\$25	Caller saved registers
\$26..\$27	Reserved and not usable
\$28	Global memory address pointer
\$29	Stack pointer
\$30	Frame pointer
\$31	Return address

Table 2.2: Floating-point register set.

Register	*Usage*
\$f0..\$f1	Return registers for floating point values
\$f2..\$f11	Caller saved registers
\$f12..\$f15	Floating point parameter passing registers
\$f16..\$f19	Caller saved register
\$f20..\$f31	Callee saved registers

Table 2.3: Register usage for parameter passing.

Values	*Register Usage*
f1, f2, ...	f1 → \$f12, f2 → \$f14
f1, i1, f2, ...	f1 → \$f12, i1 → \$6, f2 → stack
f1, i1, i2, ...	f1 → \$f12, i1 → \$6, i2 → \$7
i1, i2, i3, i4, ...	i1 → \$4, i2 → \$5, i3 → \$6, i4 → \$7
i1, i2, i3, f1, ...	i1 → \$4, i2 → \$5, i3 → \$6, f1 → stack
i1, i2, f1, ...	i1 → \$4, i2 → \$5, f1 → (\$6,\$7)
i1, f1, ...	i1 → \$4, f1 → (\$6,\$7)

1. All integer parameters are passed in the integer registers in order.
2. Double-precision floating-point values have to be passed in integer registers if any integer parameter comes before it; otherwise, it is passed in double floating-point registers.
3. Double-precision floating-point values must be aligned to even numbered registers. This applies to either integer or double floating-point registers.
4. If an integer register is skipped over because of the alignment problem, then it cannot be used for parameter passing.
5. Any parameters that cannot fit in the registers are passed on the stack.

2.3 Data Generation

It is important to place the global variables in the right section of the data memory since MIPS has the 16-bit and 32-bit data addressing features. Thus far, LCODE is unable to provide enough information in its data structures as to intelligently guide the code generator to efficiently allocate the small data memory space. All the variables are generated in the data memory, and it is up to the MIPS assembler to move the variables to the small data memory. This is done by assembling the entire program in one command line.

The MIPS assembler cannot handle long algebraic expressions. The IMPCC/MIPS code generator is required to reduce the algebraic expressions that contain labels into a minimal format. For example, we are given the following C declaration statement.

```
int p5 = &(p3[2]) - p3;
```

In LCODE, this statement is in this form.

```
(long 1 _p5 (div (sub (add (l _p3) (i 8)) (l _p3)) (i 4)))
```

The IMPCC/MIPS code generator reduces the LCODE data declaration statement into one simple form.

```
$p5:
    .word 2
```

where $p5 is the address label of the variable.

2.4 Instruction Selection

The first major step in the IMPCC/MIPS code generation is the mapping of each LCODE instruction into the appropriate MIPS assembly instruction. LCODE is broken down into six groups of instruction types: macro, branch, arithmetic, move, type conversion, and load/store. For some of the instruction groups, further distinctions between integer, floating-point, and double floating-point instruction types are made. For these instruction groups, tables are constructed to show the instruction mapping of each LCODE instruction to MIPS assembly instruction. The *LCODE* column gives the LCODE instruction. The *MIPS Code* column gives the equivalent MIPS assembly instruction or MIPS assembly instruction sequences. A branch instruction with an *fs* attached to it is the *forward semantic* version of that branch instruction. *Forward semantic* is used to indicate that the instruction is a likely taken branch.

Table 2.4 shows some LCODE assembly macros. For the *jsr* and the *jsr_fs* instructions, *function* pertains to the callee function name. But sometimes in the language C, we can call a function through a pointer. If this happens, *function* would pertain to a register whose content holds the address of the callee function. The register $31 is the linkage register, or better known as the return address register. The *prologue* and

Table 2.4: LCODE assembly macros.

LCODE	*MIPS Code*
jsr	jal *function*
jsr_fs	jal *function*
rts	j $31
rts_fs	j $31
prologue	(variable)
epilogue	(variable)
define	(ignored)
alloc	addu $2, $29, *structsize*
nop	nop

epilogue instructions translate to none or many MIPS assembly instructions depending upon the function. The roles of *prologue* and *epilogue* instructions are to set the stack pointer ($29) for the current function, and to store and load callee saved registers on and from the stack. *Define* is for the internal use of the code generator, and does not generate any equivalent MIPS assembly instruction. *Alloc* allocates a portion of stack space for a structure that is the size of *structsize* in number of bytes. The register $2 contains the beginning address of that structure, and $2 is passed to the callee function, which will return a structure on the stack space provided.

Tables 2.5 to 2.7 give the LCODE branch instruction mappings. MIPS has no equivalent instruction for the floating-point branch instructions. For these branch instructions, two MIPS assembly instructions are required for one LCODE instruction. The first MIPS assembly instruction compares the two operands, and the second MIPS assembly instruction determines the branch direction depending on the result of the comparison.

Tables 2.8 to 2.10 give the LCODE arithmetic instruction mappings. For the integer arithmetic instructions, only *nand* and *nxor* have no equivalent MIPS assembly instruction. Each floating-point compare and set instruction requires at least four MIPS

Table 2.5: LCODE integer branch instructions.

LCODE	*MIPS Code*
jump	b
jump_fs	b
jump_rg	j
jump_rg_fs	j
beq	beq
beq_fs	beq
bne	bne
bne_fs	bne
bgt	bgt
bgt_fs	bgt
bge	bge
bge_fs	bge
blt	blt
blt_fs	blt
ble	ble
ble_fs	ble
bgt_u	bgtu
bgt_u_fs	bgtu
bge_u	bgeu
bge_u_fs	bgeu
blt_u	bltu
blt_u_fs	bltu
ble_u	bleu
ble_u_fs	bleu

Table 2.6: LCODE single-precision floating-point branch instructions.

LCODE	*MIPS Code*
beq_f	c.eq.s bc1t
beq_f_fs	c.eq.s bc1t
bne_f	c.eq.s bclf
bne_f_fs	c.eq.s bclf
bgt_f	c.le.s bc1f
bgt_f_fs	c.le.s bc1f
bge_f	c.lt.s bc1f
bge_f_fs	c.lt.s bc1f
blt_f	c.lt.s bc1t
blt_f_fs	c.lt.s bc1t
ble_f	c.le.s bc1t
ble_f_fs	c.le.s bc1t

Table 2.7: LCODE double-precision floating-point branch instructions.

LCODE	*MIPS Code*
beq_f2	c.eq.d bc1t
beq_f2_fs	c.eq.d bc1t
bne_f2	c.eq.d bclf
bne_f2_fs	c.eq.d bclf
bgt_f2	c.le.d bc1f
bgt_f2_fs	c.le.d bc1f
bge_f2	c.lt.d bc1f
bge_f2_fs	c.lt.d bc1f
blt_f2	c.lt.d bc1t
blt_f2_fs	c.lt.d bc1t
ble_f2	c.le.d bc1t
ble_f2_fs	c.le.d bc1t

Table 2.8: LCODE integer arithmetic instructions.

LCODE	*MIPS Code*
rem	rem
rem_u	remu
add	addu
add_u	addu
sub	subu
sub_u	subu
mul	mul
mul_u	mul
div	div
div_u	divu
abs	abs
or	or
nor	nor
and	and
nand	and not
xor	xor
nxor	xor not
eq	seq
ne	sne
gt	sgt
gt_u	sgtu
ge	sge
ge_u	sgeu
lt	slt
lt_u	sltu
le	sle
le_u	sleu
lsl	sll
lsr	srl
asr	sra

Table 2.9: LCODE single-precision floating-point arithmetic instructions.

LCODE	*MIPS Code*
add_f	add.s
sub_f	sub.s
mul_f	mul.s
div_f	div.s
abs_f	abs.s
eq_f	li dest, 0 c.eq.s bc1f label: li dest, 1 label:
ne_f	li dest, 1 c.eq.s bc1f label: li dest, 0 label:
gt_f	li dest, 1 c.ngt.s bc1f label: li dest, 0 label:
ge_f	li dest, 1 c.lt.s bc1f label: li dest, 0 label:
lt_f	li dest, 0 c.lt.s bc1f label: li dest, 1 label:
le_f	li dest, 0 c.le.s bc1f label: li dest, 1 label:

Table 2.10: LCODE double-precision floating-point arithmetic instructions.

LCODE	*MIPS Code*
add_f2	add.d
sub_f2	sub.d
mul_f2	mul.d
div_f2	div.d
abs_f2	abs.d
eq_f2	li dest, 0 c.eq.d bc1f label: li dest, 1 label:
ne_f2	li dest, 1 c.eq.d bc1f label: li dest, 0 label:
gt_f2	li dest, 1 c.ngt.d bc1f label: li dest, 0 label:
ge_f2	li dest, 1 c.lt.d bc1f label: li dest, 0 label:
lt_f2	li dest, 0 c.lt.d bc1f label: li dest, 1 label:
le_f2	li dest, 0 c.le.d bc1f label: li dest, 1 label:

Table 2.11: LCODE move instructions.

LCODE	*MIPS Code*
mov	la (if source 1 is a label) li (if source 1 is an integer) la (if source 1 is a string) addu (if source 1 is a macro) move (if source 1 is a register)
mov_f	li.s
mov_f2	li.d

Table 2.12: LCODE type conversion instructions.

LCODE	*MIPS Code*
f2_i	trunc.w.d
f_i	trunc.w.s
i_f2	cvt.d.w
i_f	cvt.s.w
f2_f	cvt.s.d
f_f2	cvt.d.s

assembly instructions. We always set the destination register to a predefined value (0 or 1 depending on the LCODE instruction). If the comparison result is the same as the prescribed result, the destination value remains the same, otherwise the destination value is given a complementary Boolean value. Floating point comparison and set instruction is very expensive to implement in MIPS assembly language, and should be avoided when generating LCODE for the MIPS architecture.

Table 2.11 gives the LCODE move instruction mappings. The integer *mov* instruction has many mapping possibilities depending upon the source type. Each type of condition is presented within the parentheses. A string is always converted to an address label pointing to a preallocated data space that contains the value of the string.

Table 2.13: LCODE load/store instructions.

LCODE	*MIPS Code*
ld_uc	lbu
ld_c	lb
ld_uc2	lhu
ld_c2	lh
ld_i	lw
ld_f	l.s
ld_f2	l.d
st_c	sb
st_c2	sh
st_i	sw
st_f	s.s
st_f2	s.d

Table 2.14: Load/store addressing modes.

Addressing Mode	*Example*
(base register)	($16)
value	64372
value (base register)	4($16)
relocatable address	label
relocatable address ± value	label+4
relocatable adderss ± value (base register)	label+4($16)

Table 2.12 gives the LCODE type conversion instruction mappings. The only difficulty with these instructions is with the single-precision to double-precision floating-point conversion (f_f2). Because of the MIPS parameter passing constraints, the destination of this conversion may not be a floating-point register. It is possible to have the $6 and $7 register pair as the destination registers. In this case, temporary double floating-point registers must be used to receive the result of the conversion and to transfer the result to the $6 and $7 register pairs by using the coprocessor command ($mfc1$).

Table 2.13 gives the LCODE load/store instruction mappings. MIPS assembler can handle several types of addressing modes for the load/store instructions. These addressing modes are shown in Table 2.14.

2.5 Machine Dependent Optimizations

Sometimes it is more beneficial to map the LCODE instruction to another MIPS assembly instruction other than the equivalent one. We will provide an example with the add unsigned instruction.

```
(add_u (r 0 i) ((l _label i) (i 4 i) ()))
```

If we map this LCODE instruction directly, the following MIPS assembly instruction sequence results.

```
la $16, _label

addu $16, $16, 4
```

By taking advantage of the MIPS assembler addressing modes, we only have to generate one MIPS assembly instruction.

```
la $16, _label+4
```

This provides a saving of one MIPS machine instruction cycle.

MIPS assembly language has a subset of the LCODE addressing mode [1]. Thus, one LCODE instruction may map to one or many MIPS assembly instructions. The following example shows the problem of having a separate floating-point coprocessor for the MIPS architecture.

```
(f2_i (r 0 i) ((r 1 f2) () ()))
```

This can be matched to two MIPS assembly instructions.

```
trunc.w.d $f22, $f20
mfc1 $16, $f22
```

The results of the floating-point operation have to be moved from the floating-point coprocessor to the main processing unit. This mapping inefficiency is unavoidable since IMPACT architecture assumes an integrated floating-point function unit.

Sometimes, it is possible to achieve one-to-one mapping with some machine-dependent optimizations. For example, we are given the following LCODE instruction.

```
(add_u (r 0 i) ((i 4) (l _label i) ()))
```

Before any machine-dependent optimizations, three MIPS assembly instructions result.

```
la $16, _label
li $17, 4
addu $18, $17, $16
```

But after the machine-dependent optimizations, the following LCODE results.

```
(add_u (r 0 i) ( (l _label i) (i 4 i) ()))
```

Here, only two MIPS assembly instructions are required.

```
la $16, _label
addu $18, $16, 4
```

This machine-dependent optimization uses the fact that the MIPS assembler can handle an immediate field for this particular type of instruction. This type of machine-dependent optimization is called *reordering of constant fields*.

For this example, one-to-one mapping can be achieved in some situations. We can see that the load address instruction (*la*) is present in both cases. For the example shown, if the LCODE instruction is within a heavily used loop (detected either by static loop detection or by profile information), the extra load address instruction can cause the program to slow down. This is the reason behind the *constant preloading* machine-dependent optimization. If we can move the load address instruction from within the loop to the loop preheader, then within the loop, we can achieve one-to-one mapping between this particular LCODE instruction and MIPS assembly instruction.

```
(add_u (r 0 i) ( (r 1 i) (i 4 i) ()))
```

Here, only one MIPS assembly instruction remains.

```
addu $18, $16, 4
```

Lcode generates sign extending instructions for memory loads. On the other hand, MIPS architecture automatically sign extends memory loads [1] for signed instructions. Thus by using the signed instructions, a number of left and right shifts can be avoided.

Some MIPS assembly instructions are composed of more than one MIPS machine instruction. For example, *beq* and *bne* are atomic instructions for both the MIPS assembly and machine instructions. It is beneficial to replace conditional branches to *beq* or *bne* whenever possible. If we are given the following C loop,

```
for (i=0; i<10; i++);
```

the resulting LCODE contains a *blt* instruction.

```
(blt () ((r 0 i) (i 10 i) (l _loop i)))
```

By applying the *strength reduction* of conditional branch optimization, the following LCODE results.

```
(bne () ((r 0 i) (i 10 i) (l _loop i)))
```

Although only one MIPS assembly instruction is generated for both cases, the second situation will result in one fewer MIPS machine instructions.

2.6 Register Allocation

Register allocation maps each virtual register to one or more physical registers. It is important to perform good register allocation because of the following reasons.

1. We want to put all variables in registers to reduce unnecessary memory references since register access is generally faster than memory access.
2. The number of physical registers available is limited, and a proper allocation is required to increase the resource utilization.
3. Inlining, code scheduling, and optimizations increases register *live* range and pressure (the number of live registers at a given point).

The algorithm used for the code generator is Chaitin's graph coloring algorithm [3]. Here, we map many virtual registers to one physical register. The Chaitin's graph coloring algorithm assumes one live range for each virtual register. The main steps for the allocation scheme consist of the following:

1. Compute the *uses* and *defines* of each virtual register.
2. Compute the live range for each virtual register.
3. Construct the interference graph.
4. Color the graph with N colors.

A statement which contains one destination and one or many sources *defines* the destination register and *uses* the source registers. A variable is *live* at a given point within a function when it is used after that point. Thus a virtual register can said to be alive between a define and a use.

Two registers are said to be in *conflict* when their live ranges overlap, and they cannot reside in the same physical register. Virtual registers that cannot be assigned a physical register are said to be *spilled*.

To construct an interference graph for coloring, for each virtual register we create a node, or vertex, representing it. Between each vertex, an edge represents the conflict between the two virtual registers. When two vertices are connected with an edge, they are *adjacent* nodes. The *degree* of the graph is the maximum number of edges incident on a vertex.

After the construction of the interference graph, the maximum number of usable physical registers, or colors, is used to color the graph. All adjacent nodes cannot have the same color as their neighbors. If the degree of the graph is less than or equal to the number of colors available, then the graph is said to be colorable, or all vertices can be assigned a color without conflict. If the degree of the graph is larger than the number of colors available, then a decision must be made to spill a node. When we remove the vertex to be spilled from the graph, the edges associated with it are also removed. Graph coloring is a repetitive application of the coloring and spilling process until all virtual registers are either assigned to a physical register or spilled.

A simple but effective heuristic is devised to reduce the cost of spilling, which is essential to good code performance. The spill cost function is stated in Equation (2.1). W_1 is proportional to the *defines* and *uses*. The *defines* and *uses* are calculated using loop analysis and profile weights. W_2 is inversely proportional to the variable live range.

With this cost calculated for all of the nodes, the node with the smallest cost value is spilled.

$$COST = (W_1^3) * W_2 \tag{2.1}$$

We have found that Equation (2.1) works best for large function bodies where the register live ranges are long and the register pressures are great.

2.7 Register Assignment

After every virtual register has an assignment, either to a pseudo physical register or spilled, we have to decide to which real physical register it should be mapped. This is due to the fact that in the MIPS architecture, there are two groups of physical registers. The first type of registers is the *caller saved*. These register values are not guaranteed to be preserved across the function call boundaries. Thus, the caller has to initiate the saving of the value if necessary. The second type of registers is the *callee saved*. These register values are guaranteed to be preserved across the function call boundaries. When a function uses callee saved registers, it is necessary to save them in the function prologue and restore them in the function epilogue. When a function uses caller saved registers, it is necessary to save them before a function call if it is *live*, and restore them after the function call.

It is a tradeoff in terms of costs when deciding which physical registers should be caller saved and which ones should be callee saved. The follow heuristic uses both profile information and static loop detection for the decision making. The objective is to minimize the loads and stores of register on the stack; this can happen in the function prologue and epilogue for the callee saved registers, before and after function calls for caller saved registers.

There are three cost functions to be considered. Equations (2.2) and (2.4) are used to guide the mapping of the caller saved registers. Equation (2.3) is used to guide the mapping of the callee saved registers.

$$((F_n \rightarrow W) - (JSR \rightarrow cost)) \times 2 \tag{2.2}$$

$$(register \rightarrow weight) - ((F_n \rightarrow W) \times 2) \tag{2.3}$$

$$(register \rightarrow weight) - (((jsr \rightarrow cost) - (F_n \rightarrow W)) \times 2) \tag{2.4}$$

$F_n \rightarrow W$ is the number of times the function has been executed indicated by the profile information. If it is zero, one is assumed. $Jsr \rightarrow cost$ and $register \rightarrow weight$ are different for each physical register, and are calculated by the following algorithm.

```
for (i=0; i<all_assigned_physical_register; i++) {
    if (register[i].assigned != SPILLED) {
        id = phys_register_id(i);
        for (j=0; j<all_jsr_instr_in_function; j++) {
            if (instruction_in_live_range(register[i].id,instr[j].id)) {
                jrs->cost[id] += instr[j].weight;
            }
        }
        register->weight[id] += register[i].weight;
    }
}
```

The basic algorithm for register assignment is as follows.

1. For each physical register, using cost function (2.2), we calculate the cost to be mapped in a caller saved register.
2. From the largest to the smallest cost, if the cost is larger than zero, and there are still more caller saved registers left, then we assign the physical register as caller saved.
3. For each of the remaining physical registers, using cost function (2.3), we calculate the cost to be mapped in a callee saved register.
4. From the largest to the smallest cost, if the cost is larger than zero, and there are still more callee saved registers left, then we assign the physical register as callee saved.
5. For each of the remaining physical registers, using cost function (2.4), we calculate the cost to be mapped in a caller saved register.
6. From the largest to the smallest cost, if the cost is larger than zero, and there are still more caller saved registers left, then we assign the physical register as caller saved.
7. Any remaining registers that have not been assigned to caller saved or callee saved are spilled.

2.8 MIPS Assembly Code Output

The last phase in the code generation process is to output the transformed LCODE instructions into the MIPS assembly instruction format. In addition to the straight instruction-to-instruction printing, three major details have to be observed. First, it is not until this point in code generation that all of the information is available to calculate the stack size required for this particular function. Second, it is up to the output routine to insert any instructions for loading and storing values of all spilled registers. Finally, it is only at this point that the instructions to save and restore caller saved registers can be inserted. The process of code generation is completed after this point. An example program is listed in Appendix A.

3. LOW-LEVEL SUPPORTING TOOLS

Two low-level supporting tools have been created using the IMPCC/MIPS code generator: LCODE profiler and LCODE trace generator. The construction of these tools gives an example on how the code generator can be used to further the IMPACT architecture studies. An example of the MIPS assembly output of the LCODE profiler and LCODE trace generator is presented in Appendix A.

3.1 Probing LCODE

With the completion of the IMPCC/MIPS code generator, it is easy to modify the source code of the code generator to perform execution-driven simulations [4]. Depending upon the run-time information needed, the code generator can insert the appropriate probes into the program to extract this information to form a trace. By consuming the trace at runtime, unnecessary I/O traffic and system calls can be avoided. With execution-driven simulation, we can save both simulation time and storage space.

3.1.1 Statically available information

Any information that can be statically obtained directly from the LCODE data structure does not have to be generated dynamically into the trace. This information includes the function name, the maximum number of control blocks within each function, the maximum number of control operations within each function, the instruction number, the control block number, the operation number, the opcode, the register number, and the instruction address. By using the statically available information, we reduce the amount of dynamically generated information. The trace generation time is reduced.

The function name is used to hash the generated profile information back into the LCODE data structure. The information provided by the maximum number of control blocks and the maximum number of control operations is used to provide static program space and to maintain the run-time profile data structure. The opcode for the control operations is used to evaluate the branch direction using the run-time information. By using the statically available information, both the profiling speed and the trace generation speed are greatly enhanced.

3.1.2 Dynamically available information

Some information is not known until runtime. The information of interest is the instruction execution sequence, the precise branching behavior of each control operation, and the data address of each load or store operation. Using this information, we can perform simulations on different aspects of the IMPACT architecture. For example, the instruction traces can be used for simulating an instruction cache. The branch behavior can be used to study branch prediction. Since the execution speed of the probed program is proportional to the amount of data extracted, it is up to the user to probe for only the needed data.

3.2 Supporting Tool to Analyzer Interface

To promote the compatibility of the trace analyzers used for different architectures, a standard interface between the low-level supporting tools and the trace analyzer should be defined. In this way, the trace analyzers are reusable for the code generators of other machines.

3.2.1 LCODE profiler interface

For the LCODE profiler, each line of the trace contains six separate pieces of information, which are presented in the following data structure.

```
typedef struct L_Trace_Info {
        int fnid;
        int cbid;
        int cntid;
        int type;
        union value src1;
        union value src2;
};
```

The value union contains three data types: integer, single-precision floating-point, and double-precision floating-point values.

The *fnid* field contains the function number indicating in which function the operation is located. Each function has its own unique function number, which is generated at the time of code generation. The *cbid* field contains the control block number of the operation. Only the values of 1 to 4 can go into the *type* field. Each value indicates the type of operation of the trace. A value of 1 indicates the beginning of a control block.

A value of 2 indicates that the present operation is a conditional branch and that the opcode of the operation is found in the *cntid* field. A value of 3 indicates the present operation is an unconditional branch with only two destination locations. A value of 4 indicates a hashing jump. Fields *src*1 and *src*2 hold the source values of the conditional branch, or in the case of a hashing jump, the condition code and the destination location.

3.2.2 LCODE trace generator interface

For the LCODE trace generator, the following data structure is provided.

```
typedef struct L_Trace_Info {
        int fnid;
        int cbid;
        int opid;
        int opcode;
        union value src1;
        union value src2;
};
```

The value union has the same definition as described in Section 3.2.1.

fnid and *cbid* have already been described in Section 3.2.1. The opcode identification number and the operation opcode number are stored in *opid* and *opcode*, respectively. Fields *src*1 and *src*2 hold the source values of the operation. These values are used to determine the branch direction and load/store address.

3.3 LCODE Profiler

The construction of the LCODE profiler is motivated by the fact that some code optimizations (e.g., loop unrolling) approximate profile information, and accurate profile information is needed to generate accurate performance statistics. LCODE profiling is fairly fast, with only approximately 25 times slow down. The LCODE profiler generates execution frequencies for each function and each basic or super block within each function. For each control operation, both the execution frequency and the branch taken frequency are recorded. This information is then used to reconstruct the profile data to be mapped back into the LCODE data structure.

The LCODE profiler is a natural extension to the code generator. The information that is needed to be generated into each line of the traces is the function number, the control block number, the control operation identification number, the operation type, and the value of the sources that are needed to determine the branch direction. This is the dynamically available information.

The function number is set upon entry into the function, and resets after each subroutine call and each control block header. For each control block, the control block number is updated and a line of trace is added into the trace buffer. Extra operations are inserted before each conditional branch or unconditional branch to record the function number, the control block number, the control operation identification number, and the value of the sources.

At the beginning of the main function, before the execution of the prologue operation, the static information is read into a hash table. When generating the actual trace output, the dynamic information is used to access the hash table. After the epilogue operation of the main function, we have to call a special profile routine to clear the trace buffer and close all files before the termination of the program. The profile information is mapped back into LCODE after the probed program terminates.

3.4 LCODE Trace Generator

The LCODE trace generator is used to generate trace information for all operations executed. The implementation method is very similar to the LCODE profiler except that different types of information have to be generated statically and dynamically.

Statically, for each operation within each function, the instruction number, the opcode number, the registers used, and the simulated instruction address must be generated. We assume each LCODE instruction is I bytes in size. Starting with the address zero, and for each of the succeeding executable LCODE instructions, the constant size of I is added to generate the simulated instruction address.

Dynamically, the function number, the control block number, the operation number, and the source values have to be generated. Thus, unlike LCODE profiling, probes must be inserted to gather the trace information for every LCODE instruction. The LCODE trace generator is similar to the LCODE profiler in all other implementation aspects.

3.5 Implementation Hazards and Difficulties

In the C language, we have two unique library functions that can alter the control flow of the program execution: *setjump* and *longjump*. This problem is eliminated by the way the probes are inserted. Since longjump can only return to a location that has a label or right after the setjump call, by resetting the function number and control block number after each occurrence of a control block header and after each subroutine call, the trace generated will always be correct.

Programs can be terminated prematurely thru the *exit* library. This will cause the loss of the last trace of the buffer contents. A solution to this problem is to replace every occurrence of the exit function with a predefined profile function which will clean up the trace buffer before calling the actual *exit* library.

The probes inserted are the instructions that load a global data structure with the needed values. All source values are recorded before the executed instruction but after the loading of the spill registers. All destination values are recorded after the execution of the instruction, either before or after the storing of the spilled destination registers. When all of the values are recorded into the global data structure, a special function call is made to copy the values of the gathered values into a large buffer. Only after the buffer is full is the dynamic information combined with the static information. This combined information is then sent to either the LCODE profiler function or the LCODE trace generator function to be processed.

4. CODE SCHEDULING STUDIES USING THE CODE GENERATOR

By using the integrated LCODE supporting tools, it is possible to study two alternative approaches to support code scheduling for multiple-instruction-issue processors. An existing code scheduler is used to perform aggressive static code scheduling on a conventional architecture. For the software code scheduling method, an architecture with a set of non-trapping instructions is provided, and additional code scheduling can be performed on top of the base static scheduling model. For the hardware code scheduling method, out-of-order execution is supported in the microarchitecture so that the hardware can perform additional dynamic code scheduling, and the compiler performs as much global code motion as allowed by the instruction set architecture specification.

4.1 Static Code Scheduling

Code scheduling done at compile time is called static scheduling [5], [6], [7]. The LCODE code scheduler moves code both upward and downward across branch operations. Moving operations from above a branch operation to below is always safe. On the other hand, moving operations from below a branch to above is not always safe. There are two major restrictions on upward code motion.

1. The moved operation must not destroy a value that is needed when the branch is taken.

2. The moved operation must not cause an exception that may terminate the program execution.

For example, it is not safe to move a memory load operation above a branch because of the possibility of memory access violation. This algorithm is referred to 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 garbage value is returned. Page faults can be handled in the usual manner. This code scheduling model is referred to as *general code percolation.*

4.2 Hardware Scheduling Approach

Code scheduling can be done at run time [8], [9], [10]. The instruction pipeline model is partitioned into several stages: instruction fetch, instruction decode and register operand fetch, instruction issue, instruction execute, and result distribution. Given two operations op_A and op_B, such that op_B depends on the result of op_A, and op_A takes n cycles to execute, static code scheduling inserts independent operations between op_A and op_B so that op_A and op_B are fetched by the processor at least n cycles apart. In practice, the compiler may not be able to find enough independent operations to execute between op_A and op_B. With in-order execution, the instruction fetch and decode stages are stalled until the result of op_A becomes available. Dynamic code scheduling alleviates this problem by allowing subsequent operations, that are independent of op_A and op_B, to proceed to the function units while op_B waits for op_A.

Three major cases in which dynamic scheduling can improve performance on top of static code scheduling have been identified.

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

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

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

Table 4.1: Benchmarks.

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

4.3 Experiments

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

4.3.1 Benchmark programs

We have collected *C* application programs from several domains, including text processing, CAD design, and UNIX utilities. Table 4.1 shows the benchmark programs that are used in this thesis. The *name* column shows the names of the benchmark programs. The *description* column briefly describes the nature of the benchmark program.

Table 4.2: Operation latencies.

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

4.3.2 Base architecture

A single-instruction-issue in-order processor supporting the restricted code percolation model is used as the base architecture, which includes a 64-entry integer register bank and a 32-entry floating-point register bank. The architecture uses a squashing branch scheme and profiled-based branch prediction. One branch slot (one instruction) is allocated for each predict-taken branch. All function units are fully pipelined with deterministic latencies. Table 4.2 shows the operation latencies. Asynchronous events such as cache misses stall the processor pipeline. The compiler performs all optimizations and restricted code percolation for the base architecture. On the average, the base architecture executes more than 0.9 operation per cycle.

4.3.3 Multiple-instruction-issue architectures

Three multiple-instruction-issue architectures are evaluated. Each supports a different code schedule method. All three architectures duplicate hardware resources of the base architecture. Additional access ports to the register file and the cache memory are provided to satisfy the increased demands due to multiple-instruction-issue processors. All operation latencies remain the same as in Table 4.2.

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

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

4.3.4 Measurement tools

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

To measure the performance of out-of-order execution, we use the LCODE trace generator and a trace analyzer. The trace analyzer uses entire instruction traces and simulates a simple dynamic code scheduling model that has an infinite number of function units and an infinite number of reservation station entries for each function unit.

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

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

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

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

4.4 Results and Analysis

Figures 4.1 through 4.3 show the performance of multiple-instruction-issue processors. Each data point represents the harmonic mean of speedup over the base architecture

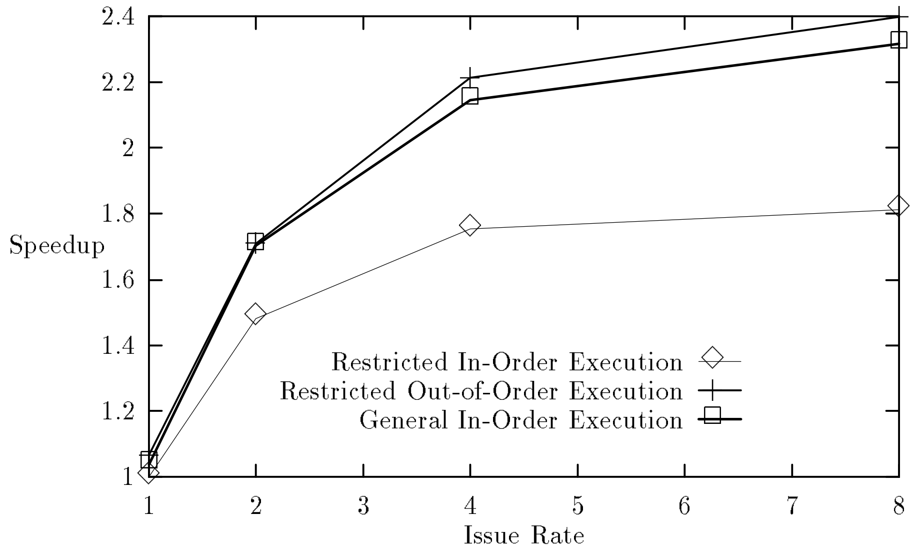

Figure 4.1: Speedup for ideal cache.

for all benchmark programs. The speedups of a machine configuration over the base architecture for individual benchmark programs are listed in Tables 4.3 through 4.5. Each column of Tables 4.3 through 4.5 is labeled XYZ, where X is the issue rate, Y indicates either restricted (r) or general (g) code percolation, and Z indicates in-order (i) or out-of-order (o) execution.

4.4.1 Ideal cache results

Figure 4.1 and Table 4.3 present speedup results for an ideal data cache. Overall, restricted out-of-order execution performs slightly better than general in-order execution. They both achieve substantial improvement over the restricted in-order execution model.

A closer look at the benchmark programs shows that load bypassing is the most beneficial feature of dynamic code scheduling. *Lex* and *qsort* are examples in which general in-order execution is severely limited by memory dependencies. For these programs, load bypassing at run time allows memory load operations on the critical path to execute

Table 4.3: Individual benchmark program results for ideal cache.

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

early. This resulted in the clear performance advantage of restricted in-order execution for *lex* and *qsort*.

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

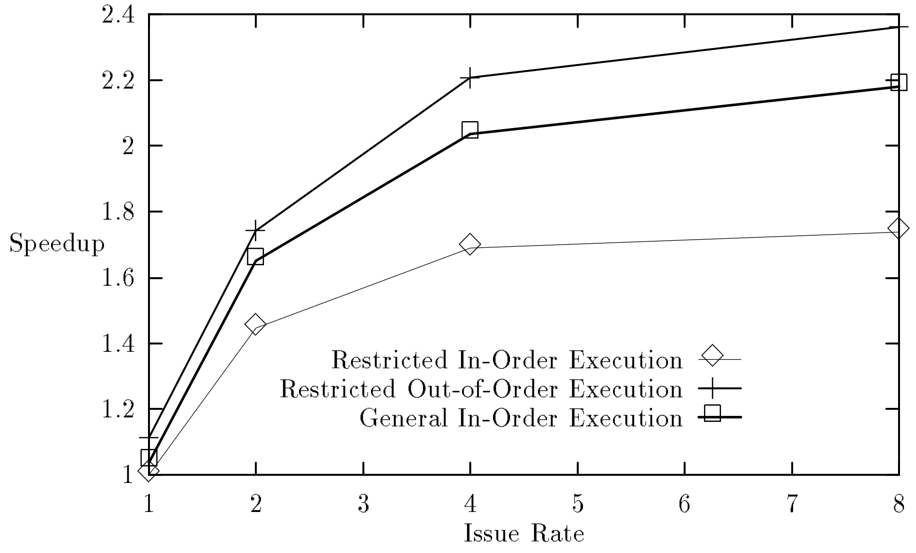

Figure 4.2: Speedup for 8KB data cache.

a result, general in-order execution has a clear performance advantage over restricted out-of-order execution for *wc* and *compress*.

4.4.2 Small cache results

Figure 4.2 presents speedup results for an 8K data cache. Each data point represents the harmonic mean of speedup over the base architecture for all benchmark programs. Cache misses degrade the performance of all architectures. Restricted out-of-order executions tolerate the cache misses better than the in-order execution models.

Figure 4.3 presents speedup results for a 16K data cache. The performance of the in-order execution models in Figure 4.3 is slightly better than was for those in Figure 4.2. On the other hand, the performance of restricted out-of-order execution was virtually identical in both cases. This shows that the performance of restricted out-of-order execution is less sensitive to cache size than in-order execution models.

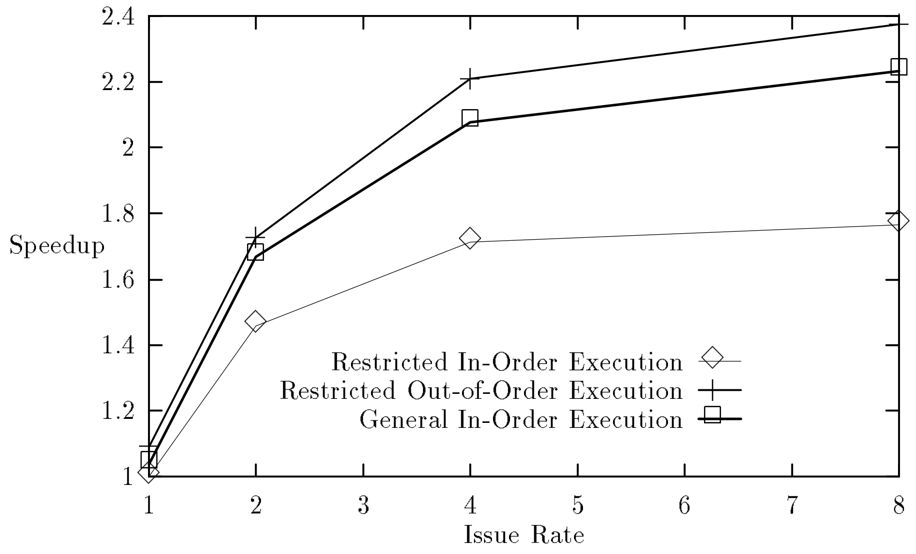

Figure 4.3: Speedup for 16KB data cache.

Table 4.4: Individual benchmark program results for 8KB cache.

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

Table 4.5: Individual benchmark program results for 16KB cache.

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

5. CONCLUSIONS

In the process of designing an architecture, the support of a good compiler cannot be ignored. The IMPACT-I C compiler is retargetable, and can easily integrate with other optimizers and lower-level tools. This integration is possible through LCODE, the IMPACT-I low-level intermediate code. By constructing the IMPCC/MIPS code generator, a platform for RISC architecture studies has been established. In addition to the code generator, we have constructed two low-level supporting tools: an LCODE profiler and an LCODE trace generator. These two tools provide accurate and efficient ways of interpreting LCODE instruction level information. With an LCODE profiler, accurate instruction count and accurate branch flow directions are measured. With an LCODE trace generator, an accurate instruction execution sequence is generated. We have shown the applicability of these low-level supporting tools by an example: the study of two alternative approaches to supporting code scheduling for multiple-instruction-issue processors. Overall, dynamic code scheduling performs slightly better than static code scheduling, but they both achieve substantial improvement compared to the model with no scheduling.

The development of low-level supporting tools is an ongoing process. The number of supporting tools is expected to grow. By conforming to the standard interface, future RISC architecture studies acquire immediate support from all of the supporting tools.

REFERENCES

[1] G. Kane, *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[2] MIPS Computer Systems, *MIPS Language Programmer's Guide*, 1986.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, and J. Cocke, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47–57, 1981.

[4] C. B. Stunkel and W. K. Fuchs, "TRAPEDS: producing traces for multicomputers via execution driven simulation," in *Proc. ACM SIGMETRICS '89 and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems.*, (Berkeley, CA), pp. 70–78, May 1989.

[5] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, no. 7, pp. 478–490, July 1981.

[6] J. L. Hennessy and T. Gross, "Postpass code optimization of pipelined constraints," *ACM Transaction on Programming Languages and Systems*, vol. 5, pp. 422–448, July 1983.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "Impact: An architectural framework for multiple-instruction processors," in *Proceeding of the 18th Annual International Symposium on Computer Architecture*, (Toronto, Canada), June 1991.

[8] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

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

[10] Y. N. Patt, W. W. Hwu, and M. C. Shebanow, "Hps, a new microarchitecture: Rationale and introduction," in *Proceeding of the 18th Annual Workshop on Microprogramming and Microarchitectures*, (Asilomar, CA.), Dec. 1985.

APPENDIX A AN EXAMPLE OF CODE GENERATION

A.1 Sample Program

```
#define N 10

double sum[N];

main()
{
int i;
    for (i=0; i<N; i++) {
        sum[i] = i;
        printf("sum[%d] = %2.2f\n",i,sum[i]);
    }
}
```

A.2 LCODE Program

```
(ms bss)
(global _sum)
(align 8 _sum)
(reserve 80)
(ms text)
(global _main)
(function _main 1.000000)
  (cb 1 1.000000 (flow 0 2 1.000000))
    (op 0 define ((mac $return_type i))(()()()))
    (op 1 define ((mac $local i))((i 0)()()))
    (op 2 define ((mac $param i))((i 16)()()))
    (op 3 prologue (())(()()()))
  (cb 2 1.000000 (flow 0 3 1.000000))
    (op 4 mov ((r 1 i))((i 0)()()))
    (op 5 mov ((r 10 i))((l _sum)()()))
  (cb 3 10.000000 (flow 1 3 9.000000) (flow 0 4 1.000000))
    (op 6 i_f2 ((r 2 f2))((r 1 i)()()))
    (op 7 st_f2 (())((r 10 i)(i 0)(r 2 f2)))
    (op 8 mov_f2 ((mac $P6 f2))((r 2 f2)()()))
    (op 9 mov ((mac $P0 i))((s "sum[%d] = %2.2f\n")()()))
    (op 10 mov ((mac $P1 i))((r 1 i)()()))
    (op 11 jsr (())((l _printf)()()))
    (op 12 add ((r 1 i))((r 1 i)(i 1)()))
    (op 13 add_u ((r 10 i))((r 10 i)(i 8)()))
    (op 14 bne (())((r 1 i)(i 10)(cb 3)))
  (cb 4 1.000000 (flow 1 5 1.000000))
  (cb 5 1.000000)
    (op 15 epilogue (())(()()()))
    (op 16 rts (())(()()()))
(end _main)
```

A.3 MIPS Code Generation Result

```
        .data
        .globl  sum
        .align  3
        .comm   sum     80
        .sdata
$$$_main_0:
        .ascii  "sum[%d] = %2.2f\n\X00"
        .text
        .globl  main
        .ent    main
main:
$_main_1:
        subu    $sp,    $sp,    40
        sw      $31,    36($sp)
        sw      $17,    32($sp)
        sw      $16,    28($sp)
        .mask   0x80030000,     -4
        .frame  $sp,    40,     $31
$_main_2:
        move    $16,    $0
        la      $17,    sum
$_main_3:
        mtc1    $16,    $f10
        cvt.d.w $f8,    $f10
        s.d     $f8,    0($17)
        mfc1.d  $6,     $f8
        la      $4,     $$$_main_0
        move    $5,     $16
        jal     printf
        addu    $16,    $16,    1
        addu    $17,    $17,    8
        bne     $16,    10,     $_main_3
$_main_4:
$_main_5:
        lw      $16,    28($sp)
        lw      $17,    32($sp)
        lw      $31,    36($sp)
        addu    $sp,    $sp,    40
        j       $31
        .end    main
```

Table A.1: Definition of the probing variables.

Name	*Usage*
IMPACT_BUF	An array where each element of the structure contains a line of the trace.
IMPACT_BUF_PTR	A pointer pointing to the current element of IMPACT_BUF.
IMPACT_Lcode_Read_Table	A function reading in all the static available information.
IMPACT_Lcode_fnid	A global variable contains the function identification number.
IMPACT_Lcode_cbid	A global variable contains the control block identification number.
IMPACT_Lcode_profile_1	A function process the trace information.
IMPACT_Lcode_profile_2	A function process the trace information.

A.4 Inserting Probes by the LCODE Profiler

Table A.1 provides the definition of the probing variables.

```
        .data
        .globl  sum
        .align  3
        .comm   sum     80
        .text
        .globl  main
        .sdata
$$$_main_0:
        .ascii  "sum[%d] = %2.2f\n\X00"
        .text
        .ent    main
main:
        la      $3,     __IMPACT_BUF
        sw      $3,     __IMPACT_BUF_PTR
        subu    $sp,    $sp,    16
        sw      $31,    ($sp)
        sw      $4,     4($sp)
        sw      $5,     8($sp)
        jal     __IMPACT_Lcode_Read_Table
        lw      $31,    ($sp)
        lw      $4,     4($sp)
        lw      $5,     8($sp)
        addu    $sp,    $sp,    16
$_main_1:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    1
        sw      $24,    __IMPACT_Lcode_cbid
        li      $25,    1
        lw      $3,     __IMPACT_BUF_PTR
        subu    $sp,    $sp,    8
        sw      $31,    ($sp)
        jal     __IMPACT_Lcode_profile_1
        lw      $31,    ($sp)
        addu    $sp,    $sp,    8
        subu    $sp,    $sp,    48
        sw      $31,    44($sp)
        sw      $17,    40($sp)
        sw      $16,    36($sp)
        .mask   0x80030000,     -4
        .frame  $sp,    48,     $31
$_main_2:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    2
        sw      $24,    __IMPACT_Lcode_cbid
        li      $25,    1
        lw      $3,     __IMPACT_BUF_PTR
        jal     __IMPACT_Lcode_profile_1
        move    $16,    $0
        la      $17,    sum
$_main_3:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    3
        sw      $24,    __IMPACT_Lcode_cbid
        li      $25,    1
        lw      $3,     __IMPACT_BUF_PTR
        jal     __IMPACT_Lcode_profile_1
        mtc1    $16,    $f10
        cvt.d.w $f8,    $f10
        s.d     $f8,    0($17)
        mfc1.d  $6,     $f8
        la      $4,     $$$_main_0
        move    $5,     $16
        jal     printf
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    3
        sw      $24,    __IMPACT_Lcode_cbid
        addu    $16,    $16,    1
        addu    $17,    $17,    8
        lw      $3,     __IMPACT_BUF_PTR
        sw      $16,    16($3)
        li      $24,    10
        sw      $24,    24($3)
        li      $24,    0
        li      $25,    2
        jal     __IMPACT_Lcode_profile_1
        bne     $16,    10,     $_main_3
$_main_4:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    4
        sw      $24,    __IMPACT_Lcode_cbid
        li      $25,    1
        lw      $3,     __IMPACT_BUF_PTR
        jal     __IMPACT_Lcode_profile_1
$_main_5:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    5
        sw      $24,    __IMPACT_Lcode_cbid
        li      $25,    1
        lw      $3,     __IMPACT_BUF_PTR
        jal     __IMPACT_Lcode_profile_1
        lw      $16,    36($sp)
        lw      $17,    40($sp)
        jal     __IMPACT_PRINT_BUF
        lw      $31,    44($sp)
        addu    $sp,    $sp,    48
        j       $31
        .end    main
```

A.5 Inserting Probes by the LCODE Trace Generator

```
        .data
        .globl  sum
        .align  3
        .comm   sum     80
        .text
        .globl  main
        .sdata
$$$_main_0:
        .ascii  "sum[%d] = %2.2f\n\X00"
        .text
        .ent    main
main:
        la      $3,     __IMPACT_BUF
        sw      $3,     __IMPACT_BUF_PTR
        subu    $sp,    $sp,    16
        sw      $31,    ($sp)
        sw      $4,     4($sp)
        sw      $5,     8($sp)
        jal     __IMPACT_Lcode_Read_Table
        lw      $31,    ($sp)
        lw      $4,     4($sp)
        lw      $5,     8($sp)
        addu    $sp,    $sp,    16
$_main_1:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    1
        sw      $24,    __IMPACT_Lcode_cbid
        subu    $sp,    $sp,    48
        sw      $31,    44($sp)
        sw      $17,    40($sp)
        sw      $16,    36($sp)
        .mask   0x80030000,     -4
        .frame  $sp,    48,     $31
$_main_2:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    2
        sw      $24,    __IMPACT_Lcode_cbid
        move    $16,    $0
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    4
        jal     __IMPACT_Lcode_profile_2
        la      $17,    sum
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    5
        jal     __IMPACT_Lcode_profile_2
$_main_3:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    3
        sw      $24,    __IMPACT_Lcode_cbid
        mtc1    $16,    $f10
        cvt.d.w $f8,    $f10
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    6
        jal     __IMPACT_Lcode_profile_2
        lw      $3,     __IMPACT_BUF_PTR
        sw      $17,    16($3)
        li      $24,    0
        sw      $24,    24($3)
        s.d     $f8,    0($17)
        li      $24,    7
        jal     __IMPACT_Lcode_profile_2
        mfc1.d  $6,     $f8
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    8
        jal     __IMPACT_Lcode_profile_2
        la      $4,     $$$_main_0
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    9
        jal     __IMPACT_Lcode_profile_2
        move    $5,     $16
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    10
        jal     __IMPACT_Lcode_profile_2
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    11
        jal     __IMPACT_Lcode_profile_2
        jal     printf
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    3
        sw      $24,    __IMPACT_Lcode_cbid
        addu    $16,    $16,    1
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    12
        jal     __IMPACT_Lcode_profile_2
        addu    $17,    $17,    8
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    13
        jal     __IMPACT_Lcode_profile_2
        lw      $3,     __IMPACT_BUF_PTR
        sw      $16,    16($3)
        li      $24,    10
        sw      $24,    24($3)
        li      $24,    14
        jal     __IMPACT_Lcode_profile_2
        bne     $16,    10,     $_main_3
$_main_4:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    4
        sw      $24,    __IMPACT_Lcode_cbid
$_main_5:
        li      $24,    0
        sw      $24,    __IMPACT_Lcode_fnid
        li      $24,    5
        sw      $24,    __IMPACT_Lcode_cbid
        lw      $3,     __IMPACT_BUF_PTR
        li      $24,    16
        jal     __IMPACT_Lcode_profile_2
        lw      $16,    36($sp)
        lw      $17,    40($sp)
        jal     __IMPACT_PRINT_BUF
        lw      $31,    44($sp)
        addu    $sp,    $sp,    48
        j       $31
        .end    main
```