1. INTRODUCTION

Recent compiler research has provided insight into many sophisticated compilation techniques for wide-issue superscalar RISC machines. However, the vast majority of computers still use processors based on CISC instruction set architectures (such as the x86) with limited register sets, complex addressing modes, and special-purpose instructions. Many of the sophisticated compiler techniques which are effective, and even necessary, for optimal performance on wide-issue RISC microprocessors exhaust the limited number of x86 registers. In many cases, these aggressive optimizations cause performance degradation. On the x86 architecture, superscalar optimizations must be guided and controlled by register pressure heuristics, and peephole optimizations that provide minor performance gains on wide-issue processors are significantly more important.

This thesis describes the modification and optimization of the IMPACT compiler [1], an aggressive superscalar RISC compiler, to generate efficient code for the 486 [2], Intel Pentium [3], and AMD K5 processors. By retargeting an existing compiler, the new code generator can utilize existing IMPACT tools for optimization, register allocation, and code scheduling. The design of the compiler can then focus on machine-dependent optimizations for the x86 architecture. Performance of the IMPACT x86 compiler is shown to exceed several commercially and publicly available compilers.

The thesis is divided into seven chapters. The remainder of this chapter briefly describes the architecture of the AMD K5 microprocessor and introduces the notation used for the x86 assembly examples in the thesis. Chapter 2 gives an overview of the IMPACT C compiler. Chapter 3 describes changes in the intermediate language that were necessary to support non-RISC elements in the x86 instruction set, such as the two-operand instruction format and the separate compare-branch instructions. Chapter 4 describes optimizations that take advantage of x86-specific features, including complex addressing modes and arithmetic instructions with memory operands. Chapter 5 presents peephole optimizations that, as mentioned earlier, are extremely important for producing efficient x86 code. Chapter 6 presents performance results. Finally, Chapter 7 offers concluding remarks and suggestions for future work.

1.1 The AMD K5 Microprocessor

One of the goals of the x86 IMPACT project is to produce efficient code for AMD's K5 microprocessor. The K5 is a four-issue superscalar machine with two integer units, a dual-ported load/store unit, a branch unit, and a floating point unit. The K5 translates each x86 instruction into one or more microinstructions [4], referred to as RISC operations or *ROPs* . The ROPs are issued in-order, but may execute out-of-order according to dataflow constraints. After the instructions finish execution, they wait in a reorder buffer and are retired in-order. Thus, the K5 is similar to superscalar processors from other vendors (such as Motorola's PPC604) and can benefit from the general superscalar optimizations in the IMPACT compiler. This thesis describes general-purpose optimizations that are effective on the K5, 486, and Pentium processors. For a discussion of superscalar scheduling and optimizations for the K5, refer to [5].

1.2 Notation

The x86 IMPACT code generator produces code for 32-bit x86 UNIX implementations. Currently, IMPACT supports Linux, Novell's Unixware, and the Pharlap 32-bit environments. The assembly code examples in this thesis use the standard UNIX x86 notation. The following rules should help to clarify the code examples:

1. Destination registers appear last in the instruction format. For example, the instruction "addl %eax, %ebx" performs the operation %ebx <- %ebx + %eax.
2. The operand size is usually encoded in the instruction mnemonic. For example, **addl** operates on longwords (thirty-two bits), while **addb** operates on bytes (eight bits).
3. Constant operands are preceded by a dollar sign, as in "$5." Register operands are preceded by a percent sign, as in "%eax."

2. OVERVIEW OF THE IMPACT C COMPILER

IMPACT is a complex compiler system composed of many optimizing, simulation, profiling, and code generation tools. IMPACT supports multiple target architectures (MIPS R2000/R3000, HPPA, SPARC, i860, AMD 29K, and x86). Recently, a template-guided approach was introduced to minimize the amount of redundant work being done in each code generator [6]. In particular, code generators now share a common machine-independent optimizer, scheduler, and register allocator. This chapter provides a brief overview of the compilation process, an introduction to the Lcode intermediate language, and a description of the standard general-purpose code generator. Knowledge of Lcode and the code generation process is important for understanding the optimization techniques described in the remainder of this thesis.

2.1 Overview of Compilation Process

Figure 2.1 shows a high-level block diagram of the IMPACT C compiler. The compilation process begins with preprocessing, which must be done on the target architecture so that correct system header files are included. Almost all other steps in the compilation process, except for profiling and assembling the final code, need not be done on the target architecture. Compiling on other machines is important for x86 compilation, since the other workstations are typically much faster than x86 machines.

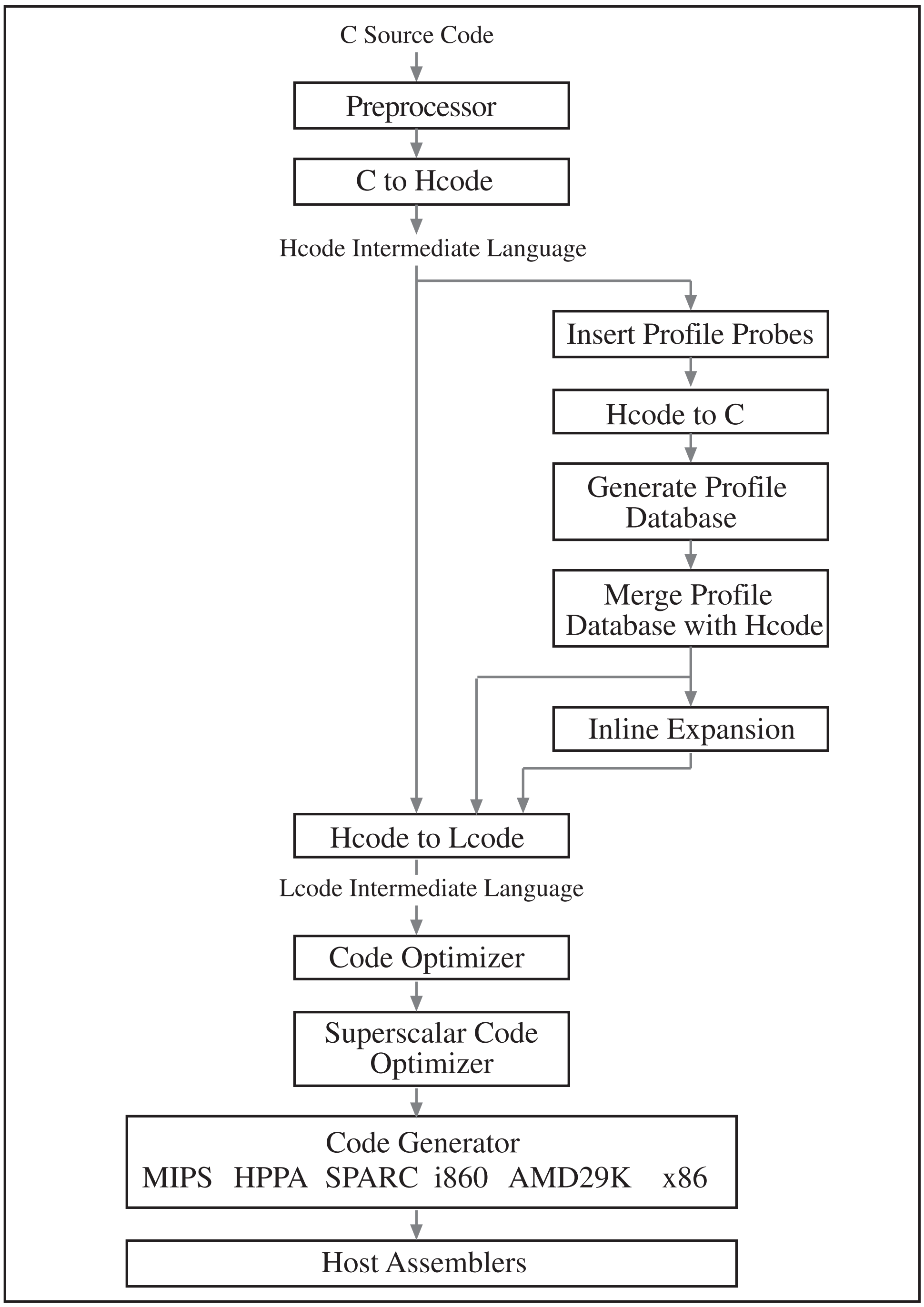

Figure 2.1: Block diagram of the IMPACT C compiler.

After preprocessing, the C files are converted into Hcode, a high-level intermediate language. If profile information is desired, tools are available for inserting profile probes into the Hcode and reverse translating the probed Hcode to probed C code. Compiling and running the probed C code on the target architecture produces a profile database. The profile database contains the invocation count of each function, the execution count of each basic block, and branch direction information [7]. The profile database is merged with the Hcode files to produce profiled Hcode. If profile information is not desired, compilation can proceed directly from Hcode generation to HtoL conversion.

If inline expansion [8] is chosen as one of the compile-time optimizations, the program will be broken up into one Hcode file per function. This simplifies the merging process required by inlining and makes it easier to isolate bugs in the compiler, since good and bad functions can be swapped by simply exchanging files. Inline expansion is much more effective with good profile information, although it is possible to inline functions without performing the profiling stages.

The HtoL tool then converts Hcode to Lcode. Lcode is a machine-independent assembly language that serves as the input language for a number of tools in the IMPACT environment, including Lopti, a machine-independent optimizer [7]. Lopti performs classical optimizations such as copy propagation, constant folding, common subexpression elimination, strength reduction, loop-invariant code removal, jump optimizations, and dead code removal.

The superscalar optimizer uses profile information to form superblocks [9]. Superblocks are groups of basic blocks that tend to execute in a sequence. The superblock has one entry point (at the start) but may have multiple exit points. Superblocks expose additional opportunities for optimization; thus all of the classical optimizations performed by Lopti are performed again on the superblock code. In addition, the superscalar optimizer may perform superscalar optimizations such as loop unrolling and software pipelining to expose parallelism between loop iterations.

After passing through the machine-independent optimizers, a code generator for the target architecture translates Lcode into machine language. Code generation involves three phases. Phase I converts the Lcode into Mcode. Mcode is similar to Lcode, but the Mcode operations are closer to the target architecture, and Mcode maps directly to the target assembly. Phase II performs additional code annotations, register allocation, scheduling, and machine-specific optimizations. Finally, Phase III converts the optimized Mcode into assembly instructions for the target architecture. The host assembler can then be used to assemble and link the final executable. Section 2.3 describes the code generation process in more detail.

2.2 Introduction to Lcode

Lcode is a compiler intermediate language that is similar to RISC instructions sets such as MIPS and SPARC. Lcode assumes a load/store architecture: arithmetic instructions are register-to-register operations, and data transfers between registers and memory are accomplished by explicit memory load/store instructions. Instructions are described with a three-operand notation. Lcode supports infinite virtual registers and basic synchronization operations [10].

Figure 2.2 shows an example of an Lcode function in the ASCII representation. The ASCII representation interfaces the various tools in the compilation process. For example, the machine-independent optimizer Lopti reads an ASCII Lcode file as input and writes an optimized ASCII Lcode file as output. Lcode also has standard internal data structures that are common across all tools. For a detailed description of the internal representation of Lcode, refer to [6].

```
(ms  data)
(global  _maxbits)
(long 1 _maxbits (i 16))
(ms  text)
(global  _rindex)
(function _rindex 1.000000 <L>)
  (cb 1 1.000000 [(flow 0 2 1.000000)])
    (op 1 define [(mac $tm_type i)] [(mac $IP i)(i 4)] <(tm (i 300))>)
    (op 2 define [(mac $tm_type i)] [(mac $IP i)(i 8)] <(tm (i 301))>)
    (op 3 define [(mac $return_type i)] [])
    (op 4 define [(mac $local i)] [(i 0)])
    (op 5 define [(mac $param i)] [(i 0)])
    (op 6 prologue [] [])
    (op 7 ld_i <F> [(r 2 i)] [(mac $IP i)(i 4)] <(tm (i 300))>)
    (op 8 ld_i <F> [(r 3 i)] [(mac $IP i)(i 8)] <(tm (i 301))>)
  (cb 2 1.000000 [(flow 1 5 0.000000)(flow 0 3 1.000000)])
    (op 9 mov [(r 1 i)] [(i 0)])
    (op 10 ld_c [(r 4 i)] [(r 2 i)(i 0)])
    (op 11 beq [] [(r 4 i)(i 0)(cb 5)])
  (cb 3 13.000000 [(flow 1 6 0.000000)(flow 0 4 13.000000)])
    (op 12 ld_c [(r 5 i)] [(r 2 i)(i 0)])
    (op 13 beq [] [(r 5 i)(r 3 i)(cb 6)])
  (cb 4 13.000000 [(flow 1 3 12.000000)(flow 0 5 1.000000)])
    (op 14 mov [(r 6 i)] [(r 2 i)])
    (op 15 add_u [(r 2 i)] [(r 2 i)(i 1)])
    (op 16 ld_c [(r 7 i)] [(r 2 i)(i 0)])
    (op 17 bne [] [(r 7 i)(i 0)(cb 3)])
  (cb 5 1.000000 [(flow 1 7 1.000000)])
    (op 18 mov [(mac $P0 i)] [(r 1 i])
  (cb 7 1.000000 [])
    (op 19 epilogue [] [])
    (op 20 rts [] [] <(tr (mac $P0 i))>)
  (cb 6 0.000000 [(flow 1 4 0.000000)])
    (op 21 mov [(r 1 i)] [(r 2 i)])
    (op 22 jump [] [(cb 4)])
(end  _rindex)
```

Figure 2.2: Sample Lcode function.

Lcode divides the program into data and function blocks. In Figure 2.2, the **(ms data)** directive begins the data block and identifies a memory space for non-constant data that can be statically initialized to any value. Other memory spaces include the **data1** space for read-only data, the **bss** space for non-initialized data, and the **sync** space for synchronization variables [10].

In Figure 2.2, the **(long 1 _maxbits (i 16))** operation declares storage space for one long-size variable at label **_maxbits** and initializes its value to sixteen. The **global** operation on the preceding line makes the **_maxbits** label visible outside the scope of the current file. In addition to declaring storage for long-size data, operations may also declare storage for bytes, words, single-precision floating point values, and double-precision floating point values.

The **(ms text)** directive defines a text space that will contain executable instructions and is the beginning of the function block in this example. Function blocks are composed of control blocks containing the Lcode instructions. Function and control blocks are annotated with the profile information gathered in the earlier compilation stages, or null information if profiling is not performed. In this case, the profile information indicates that the function is called one time.

The first control block (*cb*) begins with the **(cb 1)** directive. A unique number identifies each control block. Control blocks have one or more exit points, but only one entry point. Control blocks with one entry point are the same as basic blocks. Control blocks with more than one exit point are referred to as superblocks. The flow information in the first control block header indicates that **cb 1** always flows to **cb 2**.

The first control block contains **define** operations that indicate the number and location of incoming parameters (two stack-based parameters), the size of the return value (integer), the size of the local variable space required (zero bytes), and the maximum outgoing parameter size (zero bytes). This information can be used to calculate the necessary stack frame size for the function after the register allocator determines the spill code size. The

prologue operation is expanded during Phase II of code generation to allocate the stack frame and to save the callee-saved registers.

Lcode instructions begin with the word **op** and are composed of four major parts: operation number, opcode, operands, and attributes. The operation number is unique for each Lcode instruction in the function. Opcodes include the **define** operations seen in **cb 1** above, and more traditional opcodes such as those shown in Figure 2.3 below.

```
ld_i      Load Integer
ld_c      Load Character
mov       Move to Register
beq       Branch if Equal
add_u     Add Unsigned
rts       Return from Subroutine
```

Figure 2.3: Examples of Lcode opcodes.

Opcodes are broken down into assembler macros, integer and floating-point ALU operations, memory access, and control flow. For a more detailed description of each opcode, refer to [10].

Lcode instructions support multiple destination and source operands. Each operand has two fields. The first field specifies the operand type as either register (r), label (l), integer (i), single precision floating point (f), double-precision floating point (f2), macro (mac), or control block number (cb). Macro registers correspond directly to the registers that are present in the target architecture; for example, the x86 integer macro registers are **eax**, **ebx**, **ecx**, **edx**, **esp**, **ebp**, **esi**, and **edi**. The second field is the data of the specified type.

Finally, instructions can contain attributes to store additional information that cannot be contained in the other fields of the Lcode instruction format. For example, Lopti adds attributes to identify loop and inner-loop control blocks. In Figure 2.2, operation **op1** contains the attribute "(tm (i 300))" that indicates the first incoming parameter is passed through a memory location to the function. Later chapters in this thesis discuss some of the attributes that are added by the x86 code generator.

2.3 Code Generation

The creation of a code generator for the IMPACT involves two tasks. First, a machine-specific file (*mspec*) must be developed. The mspec file conveys the architectural characteristics and limitations of the target processor to earlier compilation stages. For example, the mspec may contain information on the data alignment requirements of the architecture or details on the addressing mode for accessing local variables.

Once the mspec is complete, a code generator must be written to convert Lcode instructions into assembly files for the target architecture. Code generation is divided into three phases, as shown in Figure 2.4.

Phase I performs code annotation to convert the Lcode into an Mcode format. Mcode is very similar to Lcode (they have identical data structures), but Mcode instructions are closer to the target instruction set, and can be easily mapped to the target assembly. For example, the x86 code generator translates three-operand instructions into two-operand instructions during Phase I annotation. The two-operand instructions can be directly converted to x86 assembly instructions. Chapter 3 describes the two-operand conversion and other Phase I annotations.

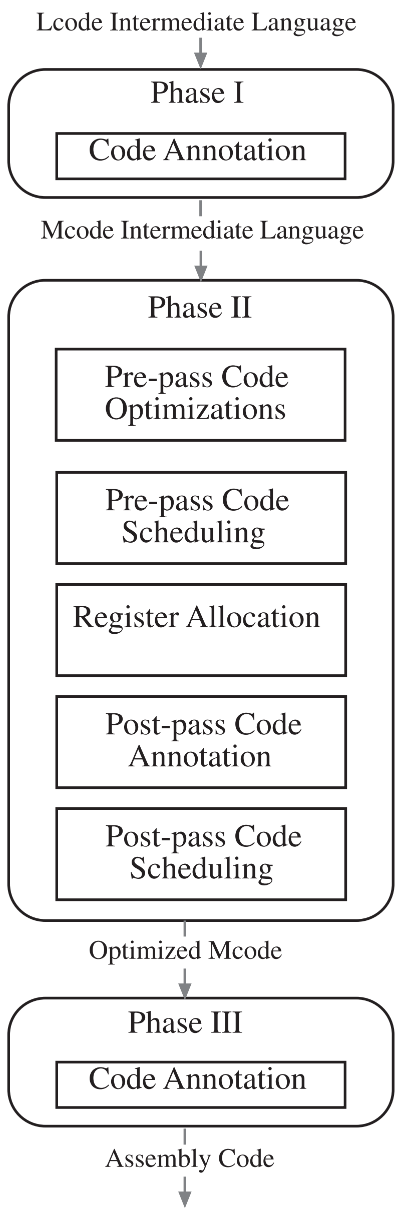

Figure 2.4: Block diagram of an IMPACT code generator

Phase II performs Mcode optimizations to remove inefficient code introduced by Phase I and to select better code sequences for the target architecture. Most code generators can use a standard library of Mcode optimizations, including common subexpression elimination, copy propagation, and dead code elimination. For the x86, the standard Mcode copy propagation reverses the two-operand conversion. Thus, the x86 code generator uses a custom copy propagation routine that is described in Chapter 2. In addition, many of the machine-specific peephole optimizations described in this thesis are performed during pre-pass code optimization. Optimizations at this stage are easier to write because they are close to the Mcode file output from Phase I, and scheduling cannot prevent the optimization opportunity from materializing.

Pre-pass code scheduling has been shown [11] to provide a better schedule than post-pass since it is not restricted by anti- and output dependences that may be introduced by the register allocator. The scheduler attempts to minimize dependences and efficiently utilize machine resources by moving instructions into fill slots and separating data definition from data use. Unfortunately, in the x86, a schedule with fewer dependences typically increases register pressure and forces large amounts of spill code. The effects are especially detrimental on narrow-issue machines such as the 486 and Pentium, which are not able to hide spill code as well as the four-issue K5.

A machine-independent register allocator [12] converts the virtual registers to the machine-specific physical registers. The register allocator decides whether to use a callee-save or caller-save register convention. If insufficient registers exist in the architecture, as is typically the case with the eight registers in the x86, the register allocator will insert spill and unspill code to break live ranges.

Post-pass code annotation and optimizations are performed after register allocation. Function prologue and epilogue, which depend on the spill code size information provided by the register allocator, are annotated at this point. In addition, instructions that reference addresses on the stack frame (such as incoming and outgoing parameters, local variables, and spill locations) are annotated to contain the correct stack offsets. Optimizations that operate on the spill code are also run at this point.

Finally, post-pass scheduling reorders instructions within the constraints of the register allocation. For the narrow-issue x86 processors, post-pass is the only scheduling performed. Post-pass scheduling is "free" in the sense that it cannot increase register pressure, but it is typically limited in its effectiveness.

3. SUPPORT FOR X86 INSTRUCTION FORMAT

The x86 instruction set includes several unusual features not found in traditional RISC architectures and not used in any of the code generators previously supported by IMPACT. For example, all x86 ALU instructions are two-operand instructions, in which one of the source registers also serves as the destination register. RISC architectures and the Lcode intermediate language use more general three-operand instructions. Other unique x86 features include separate compare and branch instructions, the stack-based x86 floating point architecture, and the concept of byte and word-sized subset registers. To make basic code generation possible, IMPACT must support these x86-specific features.

3.1 Two-Operand Instructions

The x86 instruction set was designed to minimize the instruction length at a time when memory and storage space were extremely limited. Thus the instruction set overlaps the destination register with one of the source registers to reduce the number of operands that have to be encoded in the instruction. Unfortunately, instructions in the two-operand format destroy one of the source registers. Modern RISC architectures use a fixed instruction size and a more general three-operand format. As described in Chapter 2, the Lcode intermediate language also uses a three-operand format. Thus the first task of the x86 code generator is to convert all instructions into a two-operand format. By delaying this conversion until the first phase of code generation, the machine-independent optimizers can be used. Figure 3.1 shows the conversion process.

```
R1 <- R2 + R3;
```

(a)

```
R1 <- R2;
R1 <- R1 + R3;
```

(b)

Figure 3.1: Mapping of three-operand Lcode into two-operand Mcode: (a) three-operand Lcode, (b) two-operand Mcode.

Note that conversion is only necessary if **R1** and **R2** refer to different registers.

3.2 Custom Copy Propagation

The two-operand conversion process inserts a large number of move instructions into the code. Frequently, one of the source registers is not needed after the operation, and the move instruction is useless. The code generator attempts to eliminate the unnecessary move instructions through forward and reverse copy propagation during Phase 2 optimizations. Figure 3.2 shows an example of how forward copy propagation can eliminate a move introduced during the Phase 1 conversion to two-operand instructions.

```
I1: R2 <- R1
I2: R2 <- R2 + R3;
```

(a)

```
I1: R2 <- R1;
I2: R1 <- R1+ R3;
(change all other uses
 of R2 to R1)
```

(b)

Figure 3.2: Forward copy propagation example: (a) original Mcode, (b) after forward copy propagation

Forward copy propagation requires that **R1** not be used after instruction **I2** in the original Mcode, and that **R1** not be live out of the cb. After changing all of the uses of **R2** to **R1** after instruction **I2** in Figure 3.2, the move instruction **I1** will be deleted by dead code removal.

Figure 3.3 shows an example of how reverse copy propagation can eliminate a move introduced during the Phase 1 conversion to two-operand instructions.

```
I0: R1 <- ...                I0: R2 <- ...
I1: R2 <- R1                 I1: R2 <- R2;
I2: R2 <- R2 + R3;           I2: R2 <- R2 + R3;

      (a)                          (b)
```

Figure 3.3: Reverse copy propagation example: (a) original Mcode, (b) after reverse copy propagation

Reverse copy propagation attempts to change the destination of the instruction that defines **R1**. This requires that **R1** is not used after instruction **I2** in the original Mcode, and that **R1** is not live on any path out of the cb. After reverse copy propagation, the move instruction **I1** will be deleted by dead code removal.

The generic Mcode copy propagation routines used by other code generators reverses the two-operand conversion done in Phase I and is therefore unusable in the x86 code generator. Consider the reverse copy propagation example in Figure 3.3 above: if **I0** is an subtract instruction such as "**R1** <- **R1** - 5," reverse copy propagation will create the instruction "**R2** <- **R1** - 5," which cannot be generated in the x86 architecture. To prevent this error from occurring, the x86 has customized copy propagation routines that do not propagate registers back into any instructions that only support two operands. Essentially, this means that copy propagation is limited to load operations and load effective address (**lea**) instructions.

3.3 Separate Compare-Branch Instructions

The x86 architecture implements conditional branching with separate compare and branch instructions. The compare instruction is used to set a system flags register, and the branch direction is determined from conditions in the system flags register. In addition, all x86 arithmetic and logic operations modify the flags register, and thus the compare instruction can sometimes be eliminated if the preceding ALU instruction sets the flags register correctly. Section 4.4 discusses an optimization to delete redundant compare instructions. The flags register causes problems when scheduling instructions: it is necessary to prevent the scheduler from moving ALU instructions between the compare and the branch. This scheduling problem was new to IMPACT: other architectures supported by IMPACT either use a combined compare and branch instruction (HP), or do not have arithmetic instructions which modify the flags register as a side effect.

The initial solution was to add the flags macro register as an additional destination for all the arithmetic and logic instructions. This successfully prevents ALU instructions from being scheduled between compare and branch pairs; however, it also severely restricts scheduling freedom. The scheduler cannot reorder any instructions that modify the flags register because of the apparent output dependences between the instructions. For example, consider the following:

```
I1:   [R1, flags]     <- R1 + R2;
I2:   [R3, flags]     <- R3 + r4;
I3:   [flags]         <- cmp (R3, R1)
I4:   jge loop
```

To the scheduler, it appears as if **I1** and **I2** have an output dependence on the flags register and therefore cannot be scheduled. In fact, the flags register written by instructions **I1** and **I2** is "dead" because the flags register is overwritten by **I3** before being used in **I4**. The notion of live instructions writing to dead registers is completely foreign to the scheduler, and it was decided not to modify the scheduler to support this situation.

Instead, the compare-and-branch pairs are left as single instructions until after scheduling. Scheduling cannot move ALU instructions between the compare-branch pairs because they are still single instructions when they are scheduled. This approach has the disadvantage that compare instructions cannot be scheduled, but frequently this is impossible and x86 architectures such as the Pentium can execute the compare-branch pair at the same time. If scheduling of compare instructions is still desired, a post-pass optimization after compare-branch splitting could move the comparison instruction as far as possible from the branch. By delaying compare-branch splitting until after scheduling, the general-purpose IMPACT scheduler could be used.

3.4 Floating Point

The x86 floating point architecture is implemented using a stack of eight registers. Most floating point instructions use the top of the floating point stack for one of their operands. The stack organization makes it difficult, if not impossible, to allocate the floating point registers on the register stack. To generate efficient floating point code, floating point values are allocated to memory locations on the stack frame rather than attempting to allocate the stack registers. Floating point instructions with memory operands execute at the same speed as those with stack register operands [13].

Traditional RISC architectures have a large number of floating point registers (typically thirty-two or more) that are allocated using standard register allocation techniques. Since all the x86 floating point registers are actually located on the stack frame, register allocation for the x86 simply assigns each floating point operand a home location on the stack. To facilitate this stack allocation, a large (1000 entry) register bank is passed to the register allocator. The macro register numbers returned by the register allocator are then used as indexes into the floating point register section on the stack frame.

The use of stack locations for floating point registers can lead to inefficient code that shuffles floating point operands between memory and the floating point register section on the stack frame. For example, if a floating point operand is already located in memory, it can be directly accessed from the memory location and does not need to be loaded into its home location on the stack. To eliminate these inefficiencies, a pass during Phase II detects and optimizes these sequences.

3.5 Subset Registers

The x86 architecture supports byte and word-sized instructions that access subsets of the 32-bit general-purpose registers. For example, the instruction "**addb** %al, $5" adds five to the 8-bit **al** register, which is actually the low-order byte of the 32-bit **eax** register. The subset registers can be modified without changing the rest of the register. Unfortunately, only half of the eight registers in the architecture have 8-bit versions (**eax**, **ebx**, **ecx**, and **edx**). The register allocator must allocate one of these four registers to the byte-size instruction.

Since the register allocator did not initially support the ability to restrict allocation to a subset of the available registers, the original solution was to avoid byte-size instructions whenever possible and to use an explicit macro register when the byte-size instructions could not be avoided. Byte size instructions can be avoided by zero or sign-extending 8-bit operands to occupy the entire 32-bit register, and then using the 32-bit register in 32-bit operations. Figure 3.4 shows this approach.

(a)	(b)

`movb (), %al ;load`
`...`
`addb $5, %al`
	`movzbl (), %eax`
	`...`
	`addl $5, %eax`

Figure 3.4: Elimination of byte registers via sign extension: (a) byte-sized instructions, (b) sign-extension to 32-bit instructions.

However, some instructions, such as **setcc**, always write to 8-bit destinations. For these instructions, the **eax** (or **al** in the 8-bit version) macro register was explicitly coded as the destination. The register allocator is prepared to deal with explicitly coded macro registers in the Mcode input, as these macro registers are used in many architectures for function call and return parameters, and are used in the x86 architecture for some special-purpose instructions such as shift-by-register and divide.

While the register allocator produces a correct allocation when macro registers are used in the input stream, the allocation may be less than optimal, particularly when many instructions that write to the same macro appear in close proximity. Also, the initial approach of extending all 8-bit operands to 32-bit versions is expensive, especially on 486 and Pentium processors. A zero or sign-extended load requires three cycles on a Pentium and may restrict pairing due to an opsize prefix. For these reasons, the register allocator was modified to support the concept of illegal registers (*ill_regs*) for an operand in an instruction. Byte-size ALU and **setcc** instructions are marked during Phase 1 Annotation and Phase 2 Optimization with the *ill_reg* notation to indicate that the **esi**, **edi**, **ebp**, and **esp** registers cannot be allocated to the destination operand in these instructions. The register allocator uses the *ill_reg* attribute to guarantee a legal register allocation. *Ill_regs* provide much more flexibilty than the explicitly coded macro register approach, and generate a more efficient register allocation.

4. OPTIMIZATIONS FOR X86 ARCHITECTURE FEATURES

The preceding chapter describes how IMPACT was changed to make basic x86 code generation possible. This chapter describes optimizations that take advantage of features in the x86 architecture that are not found in traditional RISC architectures. Section 4.1 describes merging address calculation operations into memory instructions using the x86 complex address modes. Section 4.2 discusses the use of CISC instructions, which reduce register pressure and instruction size by combining load and store instructions with ALU operations. Section 4.3 presents optimizations that eliminate compare instructions when the flags register is set appropriately by an arithmetic instruction. Finally, Section 4.4 lists simple transformations that reduce instruction size and increase instruction density in caches and main memory.

4.1 Complex Addressing Modes

When the x86 architecture was introduced in the late 1970s, one of the figures of merit for an instruction set was its "richness" and "flexibility" [14]. Thus, the x86 was designed with a wide variety of complex addressing modes: x86 addresses can contain a base register, plus a scaled index register that is multiplied by two, four, or eight, plus a label operand, plus a sign-extended offset, plus a segment offset. The segment offset will not be discussed here since x86 UNIX implementations use a flat memory model without the segment registers. On modern x86 implementations (such as Pentium and K5), the calculation of the complex addressing mode can be done in a single cycle, and merging the shift and add operations otherwise used to calculate the address into a complex addressing mode reduces register pressure and eliminates many dependent instructions.

For store instructions, a total of six possible source operands (store source, base, index, scale, label, and offset) are possible, requiring an expansion of the number of operands in the intermediate Mcode format. Because a larger number of operands impacts the performance of the compiler, and because addressing modes such as base plus index plus label are unlikely, the memory operand format shown in Figure 4.1 was used:

```
source 0:  source operand for stores and CISC operations
source 1:  Base Register or Label Operand
source 2:  Index Register
source 3:  Scale Factor
source 4:  Constant Offset or Label Operand
```

Figure 4.1: Internal format for x86 memory operations.

To take advantage of the complex addressing modes, memory operations are converted to the five-operand format in Phase 1 of the code generator. Before Phase 1, the memory operations are in standard Lcode format (two-operand register plus register or label or offset) and can be optimized by the machine-independent optimizers. During Phase 2 of code generation, **lea** (load effective address) instructions are generated for left-shifts of two, four, or eight, and for three-operand addition operations. Figure 4.2 shows an example of how these **lea** instructions can be merged into a complex addressing mode. The complex addressing mode in Figure 4.2(d) requires no registers to calculate the address and eliminates a chain of three dependent operations.

```
long buf[80];

a = buf[i+4];
```

(a)

```
R1 <- buf;
R2 <- R1 + 4;
R3 <- i*4;
R4 <- (R3 + R4);
```

(b)

```
R2 <- buf + 4;
R3 <- i*4;
R4 <- (R2 + R3);
```

(c)

```
R4 <- (i*4 + buf + 4);
```

(d)

Figure 4.2: Formation of complex addressing modes: (a) original C source code, (b) Lcode pseudo-code, (c) after **lea** optimization, (d) after merging into complex addressing mode.

The primary impact of complex addressing modes is on the memory disambiguation performed during code scheduling. Memory disambiguation attempts to prove that two memory instructions always refer to different memory addresses. If the load and store instructions can be disambiguated, then the load can be scheduled ahead of the store. If a load inside a loop can be disambiguated from all of the stores in the loop, then it may be possible to pull the load outside the loop during a loop-invariant code removal optimization.

Because the five-operand complex addressing mode differs from the standard Lcode format, the x86 code generator requires a custom memory disambiguation routine. The x86 memory disambiguation maps the complex memory format into two operands, and uses the same heuristics as the Lcode routine. The scale and index registers, which are not used in the Lcode heuristics, are required to be identical. For example, x86 memory addresses that are different offsets from the same register can be disambiguated if the base operands are the same, the offsets are different, and the index and scale factors for the two addresses are identical.

4.2 CISC Instructions

All arithmetic and logic instructions in the x86 instruction set support memory-to-register and register-to-memory operations, in addition to the register-to-register operations supported by traditional RISC architectures. Memory-to-register instructions load one of the operands from memory, perform the appropriate operation with a register operand, and store the result in the register. Register-to-memory instructions load one of the operands from memory, perform the appropriate operation with a register operand, and store the result back into the memory location. Because the memory-to-register and register-to-memory instructions are unique to older architectures such as the x86 and 68K, they are referred to as *CISC instructions.*

Figure 4.3 shows the formation of a memory-to-register CISC instruction.

```
R1 <- (mem);                                      TEMP <- (mem);
R2 <- R2 - R1;      R2 <- R2 - (mem);     R2 <- R2 - TEMP;

    (a)                       (b)                         (c)
```

Figure 4.3: Formation of a memory-to-register CISC instruction: (a) original code, (b) after CISC conversion, (c) after decoding into K5 ROPs.

The single CISC instruction in Figure 4.3(b) will require fewer bytes than the two instructions in Figure 4.3(a). The real advantage, however, of the CISC instruction is the reduction in register pressure since the instruction directly accesses one of its operands from memory and does not require that it first be loaded into a register. On the K5, the CISC instruction will be decoded into a load ROP and a subtract ROP, as shown in Figure 4.3(c). The ROPs use one of the internal registers of the K5, labeled as **TEMP** in Figure 4.3(c). The disadvantage of the CISC instructions is that the ROPs within the CISC instruction cannot be scheduled to eliminate true dependences. For example, the subtract ROP in Figure 4.3(c) is dependent on the load ROP, and will not execute until the load ROP completes. If the CISC instruction was not formed, the scheduler might be able to move the load ahead of the subtract, and the subtract could execute immediately after issuing.

One solution to the scheduling problem is to only merge CISC instructions if the scheduling opportunities do not materialize. The optimization merges only immediately adjacent load and ALU instructions. However, we have found that the scheduler is frequently overaggressive, and so a more aggressive conversion to CISC instructions actually helps by limiting scheduling freedom. Thus, the optimization that is currently implemented is capable of merging load instructions that have been separated from the ALU instruction by the scheduler. To prevent the CISC optimization from negating register allocation by converting all instructions to CISC operations, a heuristic disables the CISC conversion if the load destination is used three or more times.

Figure 4.4 shows the formation of a register-to-memory CISC instruction.

```
R1 <- (mem)                                        TEMP <- (mem);
R1 <- R1 - R2;        (mem) <- (mem) - R2;         TEMP <- TEMP - R2;
(mem) <- R1;                                       (mem) <- TEMP;

      (a)                     (b)                        (c)
```

Figure 4.4: Formation of a register-to-memory CISC instruction: (a) original code, (b) after CISC conversion, (c) after decoding into K5 ROPs.

Again, the CISC instruction in Figure 4.4(b) has a more efficient encoding and uses fewer registers than the original code in Figure 4.4(a). The K5 will translate the CISC instruction into three ROPs and use an internal temporary register, as shown in Figure 4.4(c). The register-to-memory CISC conversion suffers from the same scheduling problems as the memory-to-register CISC. However, since the register-to-memory CISC instruction stores the result back to memory, multiple uses of the load destination are much less likely. Nevertheless, the register-to-memory CISC optimization uses the same heuristic as the memory-to-register CISC optimization.

Both of these optimizations restrict the pairing abilities on a Pentium processor and may cause performance degradation. On the 486, which does not pair instructions, and on the K5, which has a more general decode mechanism, CISC instructions improve performance by reducing register pressure and, to some extent, by encoding the instructions more efficiently. Further research is necessary to explore the trade-offs between scheduling and CISC instructions. As the scheduler improves and incorporates register pressure heuristics appropriate for the x86, the current heuristics may have to be revised to make this optimization less aggressive.

4.3 Elimination of Redundant Compare Instructions

All x86 arithmetic and logic instructions modify the system flags register used for conditional branching. Occasionally, the ALU instruction sets the flags correctly for a subsequent conditional branch instruction, and the intervening compare can be deleted. Figure 4.5 shows an example of a sequence in which the compare instruction can be deleted.

```
I0: add %eax, $5          I0: add %eax, $5
I1: cmp %eax, $0          I1:
I2: jge loop              I2: jge loop

      (a)                       (b)
```

Figure 4.5: Elimination of redundant compare instructions: (a) original code, (b) after compare elimination.

As discussed in Section 3.3, compare-branch instructions are not split into two instructions until after post-pass code scheduling has occurred. After the splitting, another pass through the code deletes the unnecessary compares. A compare is considered unnecessary if

1. One of the operands of the compare is a constant zero,
2. The other operand of the compare is the destination of a preceding ALU instruction,
3. No other instruction writes to the flags register between the ALU instruction and the compare, and
4. The preceding ALU instruction correctly sets the conditions needed by the subsequent branch instruction.

The last condition is necessary because some ALU instructions (such as multiply) destroy the flags but do not set any of them correctly, and other operations (such as logic and shift instructions) do not set all of the flags. Another more subtle problem involves the **incl** and **decl** instructions, which do not set the carry flag but are otherwise identical to "**addl** $1, %reg" and "**subl** $1, %reg" instructions. A branch such as **jge** that requires the carry condition flag cannot use the flags register as set by these instructions.

To maximize the opportunities for eliminating redundant compare instructions, two optimizations are performed at earlier stages in the compilation path to create as many compare zero instructions as possible. First, compare-zero instructions can sometimes be produced by applying the transformation shown in Figure 4.6 below.

`I0: R1 <- R2;`
`I1: R1 <- R1 - C1;`
`I2: Jcc R2, C1, LOOP`
	`I0: R1 <- R2;`
	`I1: R1 <- R1 - C1;`
	`I2: Jcc R1, $0, LOOP`

(a)	(b)

Figure 4.6: Example of creating compare-zero instructions: (a) original Mcode, (b) after make compare zero optimization.

C1 is a typically a constant operand, although the optimization does not require it to be so. The **Jcc** instruction (**I2**) refers to any conditional jump instruction that compares **R2** to **C1**. Since instruction **I1** sets **R1** to (**R2** - **C1**), if **R2** is equal to **C1**, then **R1** must be equal to zero. The optimization changes the jump instruction so the comparison is between **R1** and zero instead of **R2** and **C1**. At the time this optimization is performed, compare-branch pairs have not been split. Thus this optimization, referred to as *make_cmp_zero*, changes the branch instruction. When the branch is split, a compare-zero operation will be created. If the conditions listed above can be met, then the compare can be deleted.

This optimization introduces an additional dependence on **R1** between **I1** and **I2**, but it frequently eliminates a comparison instruction. Also, **I2** is often the only use of **R2**, and thus instruction **I0** can be eliminated through copy propagation. Figure 4.7(b) shows the code which is frequently produced by this optimization after compare elimination and copy propagation.

```
I0: movl %ebx, %eax
I1: addl %ebx, $1
    cmpl %ecx, $1
I2: jne  loop

      (a)
```

```
I1: addl %eax, $1
I2: jne  loop

      (b)
```

Figure 4.7: Results of make compare zero optimization: (a) original assembly, (b) optimized assembly.

Table 4.1 shows three simple instruction transformations that create additional compare-zero instructions.

Table 4.1: Transformations to create compare-zero instructions.

Original Code	Compare-zero code
`cmpl $-1, %reg` `jg   label`	`cmpl $0, %reg` `jge  label`
`cmpl $1,  %reg` `jl   label`	`cmpl $0, %reg` `jle  label`
`cmpl $-1, %reg` `jb   label`	`cmpl $0, %reg` `jbe  label`

Table 4.1 uses the standard x86 notation for conditional branches: **jg** refers to a signed branch-greater-than instruction, **jge** is a signed branch-greater-than-or-equal instruction, and **jb** is an unsigned branch-less-than instruction. Consider the transformation for instruction **jg**. This transformation compares two integers, and therefore any number that is greater than negative one is either a zero or a number greater than zero. The optimization is done as part of the *make_cmp_zero* optimization described above, and relies on subsequent passes to create and then delete the redundant compare instruction if possible.

4.4 Instruction Size Optimizations

CISC architectures such as the x86 were designed with a tight encoding to minimize the instruction size, leading to more efficient use of instruction caches. For some operations, the x86 supports more than one opcode that accomplish nearly identical results. The smaller instructions may have an impact on performance for cache-intensive programs. This section discusses transformations that utilize the smaller instruction opcodes when appropriate. Typically, the smaller instructions affect different bits in the flags register, and thus care must be taken that the optimization does not change the meaning of the original code.

4.4.1. **movl** $0, %reg instruction

The "**movl** $0, %reg" operation requires five bytes because a full 32-bit constant zero must be encoded in the instruction. A smaller sequence, requiring only two bytes, is shown in Figure 4.8 below.

(a)	(b)

`movl $0, %eax`
`(b8 00 00 00 00 )`
`5 bytes`
	`xorl %eax, %eax`
	`(31 c0)`
	`2 bytes`

Figure 4.8: Transformation of **movl** to **xorl**: (a) original code, (b) smaller code.

The **xorl** operation destroys the flags register, whereas the original **movl** instruction does not. Thus, care must be taken to ensure that the flags are not live across the original instruction. Phase II transforms **movl** instructions to **xorl** operations after compare instructions are deleted, because it is preferable to eliminate the compare rather than use the smaller instruction in the case where the flags are live across the **movl** instruction.

4.4.2. **movl** $-1, %reg instruction

This instruction requires five bytes because a full 32-bit negative one must be included in the opcode. A smaller sequence using a sign-extended immediate constant, requiring only three bytes, is shown in Figure 4.9 below.

(a)	(b)

`movl $0, %eax`
`(b8 ff ff ff ff)`
`5 bytes`
	`orl $1, %eax`
	`(83 c8 ff)`
	`3 bytes`

Figure 4.9: Transformation of **movl** to **orl**: (a) original code, (b) smaller code.

This optimization takes advantage of the fact that the x86 supports a sign-extended immediate **orl** operation, but not a sign-extended immediate **mov**. It can be applied to any **mov** instruction with a constant in the range -1 to -255. Again, the optimized code destroys the flags register and the same precautions discussed in Section 4.4.1 must be applied here as well.

4.4.3. **cmpl** $0, %reg instruction

If the branch instruction that uses the result of a compare-with-zero instruction is a **jne** or **jeq** instruction, then the smaller sequence shown in Figure 4.10 below may be used.

```
cmpl $0, %eax             testl %eax, %eax
(83 f8 00)                (85 c0)
3 bytes                   2 bytes

    (a)                       (b)
```

Figure 4.10: Transformation of **cmpl** to **testl**: (a) original code, (b) smaller code.

The **test** instruction performs a nondestructive bitwise **and** operation, setting the zero flag in the condition codes register but not modifying either of the source operands. The **test** instruction in Figure 4.10 sets the zero flag in the flags register if **eax** is zero but clears it otherwise. The conversion to test instructions occurs during the pre-pass optimization pass in Phase II of the code generator.

4.4.4. **addl** $1, %reg / **subl** $1, %reg instructions

An add or subtract instruction in which one of the source operands is a constant with a value of one may be more efficiently coded using the **incl** and **decl** instructions, as shown in Figure 4.11 below.

```
addl $0, %eax             incl %eax
(83 c0 01)                (40)
3 bytes                   1 byte

   (a)                       (b)
```

Figure 4.11: Transformation of **addl** to **incl**: (a) original code, (b) smaller code.

The **incl** and **decl** instructions require only one byte. On a Pentium processor, single-byte instructions may execute more efficiently since they can pair with other instructions immediately after being fetched from memory. Other instructions can pair only if they are fetched from the instruction cache.

The smaller instructions do not set the carry flag in the condition flags register, and thus unsigned branches cannot delete a compare instruction if the preceding operation is an **incl** or **decl**. This problem is solved by performing the translation before compare-branch splitting and by modifying the redundant compare optimization so that it does not inappropriately delete compare instructions when the flags are set by **incl** or **decl** instructions.

5. ADDITIONAL X86 OPTIMIZATIONS

Previous chapters focused on creating a functional x86 code generator and on optimizing the code generator to use some of the architectural features in the x86. This chapter describes additional machine-specific optimizations, focusing on instruction selection, improvements to relieve register pressure, and the control of superscalar optimizations.

Optimizations discussed in the beginning of this chapter utilize specific x86 instructions such as **lea**, **test**, **setcc**, and **sbb**. The **lea** instruction can be used as a three-operand add or shift operation, and is also useful for performing constant multiply instructions. The **test** instruction masks bits in a register, and may eliminate **mov-and** sequences introduced in Phase I to support the two-operand format. The **setcc** and **sbb** instructions are used to eliminate branches by predication.

Other transformations remove inefficient code introduced in earlier compilation stages. For example, Lcode follows all character load instructions with an "**and** $255, %reg" instruction to clear the rest of the register. Since the x86 supports a zero-extend character load operation, the **and** instruction is redundant. Other examples, including improved bit field handling and multiple return statements, are discussed below.

Many of these optimizations produce good general-purpose improvements, but have not been implemented in the general optimizers because they provide relatively small gains on wide-issue machines. For example, almost all machines have zero-extend character load instructions, but much more performance can be gained from good scheduling and superscalar transformations than from removing a single instruction. In fact, some superscalar optimizations are register intensive and have a negative impact on performance with the limited number of x86 registers. Section 5.9 discusses how these register-intensive superscalar optimizations were controlled and, in some cases, disabled.

Next, strategies for handling the register pressure problem are discussed. Section 5.10 presents an argument for the caller-save register convention. Section 5.11 describes the use of the base pointer as the seventh general-purpose register. Under some circumstances, even the stack pointer can be used as a general-purpose register. Section 5.12 describes these circumstances and the methods for using the stack pointer as the eighth register.

Section 5.13 describes an optimization to eliminate flow dependences in the function prologue. Finally, Section 5.14 describes the use of compiler built-in functions to further improve performance on the benchmarks that use them.

5.1 Load-Effective Address Instruction

The load effective address (**lea**) instruction is designed to calculate a complex address by adding base plus scaled index plus constant offset in a single cycle. It is also the only three-operand instruction in the x86 instruction set and can sometimes be used as a three-operand add or shift to eliminate an otherwise necessary **mov** instruction. Figure 5.1 shows an example of an **lea** add optimization to remove an unnecessary **mov** instruction.

```
I0: movl %ebx, %eax
I1: addl %ecx, %eax       I1: lea [%ebx+%ecx], %eax
...                              ...
(use of %ebx)             (use of %ebx)

   (a)                           (b)
```

Figure 5.1: **Lea** add optimization: (a) original code, (b) after **lea** optimization.

The use of **ebx** in the sequence shown in Figure 5.1 prevents copy propagation from eliminating the move instruction. The three-operand **lea** instruction adds **ebx** and **ecx** together as base and index registers and stores the result of this "address calculation" in **eax**. Because the **lea** is a three operand instruction, the **movl** instruction (**I0**) used to shuffle the operands for the two-operand add is no longer necessary and will be deleted as dead code.

The **lea** instruction generates an address generation interlock (*AGI*) on the 486 and Pentium processors. An AGI occurs when the instruction preceding an **lea** or a memory instruction writes to one of the registers (either base or index) used in the address calculation. For example, an AGI exists on the **eax** register in the following example:

```
add $5, %eax
lea [%eax], %ebx
```

The AGI occurs between instructions that issue on consecutive cycles on the 486 and Pentium processors. On a 486 processor, AGI can occur only between immediately adjacent instructions. On the Pentium, with its two superscalar integer pipes, instructions may have to be separated by up to three intervening instructions to avoid AGI. The K5 does not suffer from the AGI problem, but **lea** instructions are executed in the load/store unit. This may cause a bottleneck if the code around the **lea** contains a large number of memory instructions. For these reasons, a standard **add** instruction is preferable to the **lea** if the preceding **mov** instruction can be eliminated by copy propagation. In the IMPACT x86 code generator, copy propagation runs before the **lea** optimization so that copy propagation has priority over the **lea** transformation.

Frequently, the **lea** instructions created are actually address computations and can be merged into complex addressing modes in subsequent memory instructions. Figure 5.2 shows a frequently occurring example.

```
I0: mov $iob_buf, %eax
I1: add $16, %eax              I1: lea %eax, $iob+16

          (a)                               (b)
```

Figure 5.2: Common address calculation with **lea** instruction: (a) original code, (b) after lea optimization.

The code in Figure 5.2 is generated by calls to the **getc** and **putc** system macros. A subsequent load or store instruction uses the address to move a character from or to the system buffer at iob_buf+16. The optimization that creates complex addresses for memory instructions will recognize the **lea** instruction as an address calculation and may be able to merge it into the memory instruction. To further facilitate complex address merging, left-shifts of one, two, or three are converted during Phase I annotation into **lea** instructions using the scale factor to multiply the index operand by two, four, or eight. This annotation may also eliminate a **mov** instruction with the three-operand **lea**.

5.2 Constant Multiply Instructions

The conversion of multiply instructions with constant operands into a series of less expensive add, shift, and subtract operations is a traditional code optimization. The optimization is motivated by the long latency of multiply instructions (ten to eleven cycles on a Pentium). On the x86 architecture, the add and shift operations can frequently be combined into **lea** instruction, further improving the effectiveness of this optimization. The IMPACT x86 code generator uses two algorithms to convert constant multiply instructions into a more efficient sequence of simpler instructions. The first is a version of the modified Booth's algorithm [15] that identifies strings of ones in the binary representation of the constant and then converts each string into a series of simple operations. The operations for each string can then be added to produce the final result. The second algorithm attempts to factor the constant into numbers that can be easily generated by a sequence of three or fewer simple instructions, and then multiplies the factors to get the final result.

Figure 5.3 shows an example of the Booth's algorithm approach for converting constant multiply instructions into a sequence of simpler operations.

```
R1 <- R2  * 23;
```

(a)

```
I0: R1 <- R2 ;
I1: R1 <- R1 << 3;
I2: R1 <- R1 - R2;
I3: R3 <- R2 << 5;
I4: R1 <- R1 + R3;
```

(b)

Figure 5.3: Example of Booth's algorithm: (a) constant multiply instruction, (b) sequence of simpler operations for performing the multiply.

The binary representation of twenty-three is "10111_2." Booth's algorithm breaks the binary representation of the number into strings of ones. For example, "10111" is broken into

$$23 = 10111_2 = 00111_2 + 10000_2 .$$

For each string, the algorithm performs the multiplication by left-shifting the multiplicand by the number of bits in the string, and then subtracting the original multiplicand. For example, the multiplication by seven in the above example would be performed as

$$x * (111_2) = x * (1000_2 - 0001_2) = x * (8 - 1).$$

Figure 5.3 shows how "1000_2" and "0001_2" are each computed with simple shift instructions. Instruction **I0** and **I1** serve to copy the original number and multiply it by 1000_2. Instruction **I2** subtracts the original number to complete the multiply by seven. For long strings, Booth's Algorithm uses substantially fewer instructions than a straight-forward shift and add sequence for each bit. For example, the above multiplication by seven could also be performed somewhat inefficiently as

$$x * (111_2) = x * (001_2 + 010_2 + 100_2).$$

Booth's algorithm will calculate the contribution of all strings and add them together. In this case, the only other string is "10000_2". Since it contains only one bit, this number can be efficiently calculated with a single shift instruction rather than the shift-add-subtract sequence shown above for the string of seven. Instruction **I3** from Figure 5.3 performs the shift to multiply the original number by sixteen, and instruction **I4** adds the partial sums to produce the final result.

Lea instructions can be useful for combining **mov-shift** sequences such as instructions **I0** and **I1** in Figure 5.3, and for combining **mov-add** sequences that are also frequently produced by the multiply algorithm. Another special-case optimization is to replace arithmetic left-shifts with add instructions. For example, consider

R1 <- R1 << 1.

This operation is identical to

R1 <- R1 + R1.

However, both the Pentium and the K5 have a shifter capable of executing the first instruction in only one of their two integer units. The second instruction can be executed in either integer unit, and is therefore more desirable.

The second multiplication algorithm breaks the constant into factors that can be easily generated by a few instructions. Easily generated factors include powers of two, powers of two plus one, and powers of two minus one. Any power of two can be computed with a single arithmetic left shift. Powers of two plus or minus one can be calculated by copying the number, left-shifting it, and then adding or subtracting the original number. For example, Figure 5.4 shows a multiplication by thirty-three using this technique.

```
R1 <- R2  * 33;

      (a)
```

```
I0: R1 <- R2 ;
I1: R1 <- R1 << 5;
I2: R1 <- R1 + R2;

      (b)
```

Figure 5.4. Easily generated multiply by thirty-three: (a) constant multiply instruction, (b) sequence of simpler operations.

The **lea** instructions provide additional easily generated numbers, since they can perform a shift and addition in the same instruction. For example, a multiply by nine can be done with one **lea** instruction:

R1 <- R2 + R2 *8;

Similarly, two **lea** instructions can perform a multiply by seventeen:

R1 <- R2 + R2 *8;

R1 <- R1 + R2 *8;

The factoring constant multiply algorithm searches for factors in the constant by starting with large powers of two and working down to smaller numbers. Factors are pushed onto a stack as they are found. If the constant can be factored into easily generated numbers, then the factors are popped from the stack and converted into an appropriate sequence of simple operations. Figure 5.5 shows an example of the factoring algorithm.

```
R1 <- R2  * 155;
```

(a)

```
I0: R1 <- R2 + R2 *4 ;
I1: R3 <- R1;
I2: R1 <- R1 << 5;
I3: R1 <- R1 - R3;
```

(b)

Figure 5.5: Example of factoring algorithm: (a) constant multiply instruction, (b) sequence of simpler operations for performing the multiply.

The second algorithm factors 155 as thirty-one times five. Instruction **I0** performs the multiply by five. Instructions **I1**, **I2**, and **I3** multiply the result of instruction **I0** by thirty-one to produce the final result.

The cost for a multiply instruction has rapidly decreased in recent processor generations due to much better multiplication hardware. For example, on the 486, multiply instructions have a latency of thirteen to forty-two cycles, since the 486 uses an early-out algorithm.

On the Pentium, the latency is least ten cycles. On the K5, a multiply takes only four cycles. The reduction in multiply latencies lessens the effectiveness of this optimization. Additionally, multiply instructions on the K5 are executed in the infrequently used floating point unit, making the threshold for a useful optimization even lower. The simple operations used to calculate the multiply tend to form a chain of dependent operations, implying that even a superscalar processor such as the K5 will serialize and execute only one instruction per cycle during the dependent chain.

Determining the optimal sequence of simple operations for performing a constant multiply is believed to be an NP-complete problem [16]. To prevent this optimization from converting constant multiply instructions into less efficient sequences of simple operations, each algorithm computes a cost function for the number of instructions it expects to generate for the given constant. If the cost of the best algorithm is below the threshold for the target processor, it is run again to perform the transformation. An optimal algorithm might combine the two techniques so that the number could be factored into constants that were computed with Booth's algorithm. However, this appears to provide marginal gain over the straightforward approach of two separate algorithms, especially considering the low multiplication latency of the modern processors.

5.3 Bit Masking with Test Instruction

Bit masking operations are typically done with the "&" operation in the C source code, and appear in Lcode as **and** opcodes. After conversion to Mcode, a **mov** instruction may be added to support the two-operand format. If the operand that is being masked is needed after the **and** instruction, the **mov** cannot be eliminated by copy propagation. This can happen if more than one bit test is performed on the same source operand. The **test** instruction performs a nondestructive **and** operation: the condition codes are set correctly, but the original operand is not modified. Figure 5.6 illustrates the effectiveness of the **test** instruction. Code in the "espresso" benchmark from the SPECint92 suite [17] performs a sequence of bit tests similar to that shown in Figure 5.6.

```
I0: movl %r1, %r2;
I1: andl $1, %r2;
I2: jne  label_1
I3: movl %r1, %r2;
I4: andl $2, %r2;
I5: jne  label_2

        (a)
```

```
I0: testl %r1, $1
I1: jne   label_1
I2: testl %r2, $2
I3: jne   label_2

        (b)
```

Figure 5.6: Example showing effectiveness of **test** instruction optimization: (a) bit-masking with **and** instruction, (b) bit-masking with **test** instruction.

The test instruction is similar to a compare in that its only effect is to modify the flags register. Thus, it suffers from the same scheduling problems as standard compare instructions: the scheduler must be prevented from inserting instructions that modify the flags between the test and the branch that uses the flags. To solve this problem, the test optimization tags the branch instruction with a *test_compare* attribute. When the compare-branch pair with a *test_compare* attribute is split, a **test** instruction is generated instead of the standard **cmp** operation.

5.4 X86 Predication

As wide-issue x86 microprocessors such as AMD's K5 and Intel's P6 become available, mispredicted branches are becoming increasingly expensive. For example, the P6 has a branch misprediction penalty of at least eleven cycles. These microprocessors speculatively execute past branches, and rely on accurate branch prediction so that the results computed by the speculatively executed instructions can be used. One way for the compiler to improve the branch prediction accuracy is with intelligent code layout [18] so that the most frequently executed branch direction is the fall-through path. Another technique is to eliminate the branches entirely with predication.

The x86 supports predication with the **setcc** instruction, and in some special cases, the **sbb** instruction. The **setcc** instruction sets the destination byte if the condition specified in the instruction is met, but clears the byte if not. Figure 5.7 shows an example of predication using the **setcc** instruction.

```
  testl $1, %R1;          testl $1, %R1;
  jeq   label;            setne %R3;
  addl  $1, %R2;          addl  %R3,%R2;
label:

      (a)                        (b)
```

Figure 5.7: Predication using the **setcc** instruction: (a) original code, (b) after predication.

The **setcc** instruction modifies only the least significant byte in **R3**, so an extra instruction may have to be added to clear the upper part of the register. The original code in Figure 5.7 requires two instructions if the branch is taken, and three if not. After predication, three instructions are always required; however, predication eliminates the branch and the misprediction penalties. A more typical case might involve loading a register, adding one, and then storing the register back to memory in the fall-through path of the branch, as shown in Figure 5.8.

```
    testl $1, %R1;           testl $1,  %R1;
    jeq   label;             setne %R3;
    movl  (), %R2;           movl  (),  %R2;
    addl  $1, %R2;           addl  %R3, %R2;
    movl  %R2,();            movl  %R2, ();
  label:

         (a)                      (b)
```

Figure 5.8: More typical example of predication using the **setcc** instruction: (a) original code, (b) after predication.

In this case, the predicated code always requires five instructions, whereas the original code requires five instructions for the fall-through path but only two instructions when the branch is taken. The predicated code still avoids the branch penalty, which may be sufficiently substantial to justify the optimization. Additionally, superscalar processors such as the K5 have many functional units, and five instructions are not equivalent to five cycles. In this example, the code will execute in four cycles since the load is independent of the first two instructions and can execute in parallel. However, accurate profile information is required to identify sequences for which profiling is appropriate. In the above code, if the branch is taken nearly all of the time, predication is not a good choice. On the other hand, if the branch is taken closer to half of the time, it will probably be difficult to predict, the fall-through path will be executed frequently, and predication will improve performance.

Figure 5.9 shows the use of the **sbb** to eliminate branch instructions using another form of predication. Instruction **I0** sets the carry flag if **R1** is greater than **R2**. The **sbb** instruction subtracts its first operand from the second operand, and then subtracts a constant one if the carry flag is set. Since the first and second operands are **R3** in instruction **I1** in Figure 5.9, **I1** will set **R3** to negative one if the carry flag is set, or zero if the carry flag is clear. Finally, instruction **I2** sets **R3** to positive one if it is zero, but does not change **R1** if it is negative one. This has the same effect as the code in Figure 5.9(a), but requires fewer instructions and eliminates two branches.

```
  cmpl %R1, %R2
  jg  label1
  movl $-1, %R3
  jmp  label2
label1:
  movl $1,  %R3
label2:

     (a)
```

```
I0: cmpl %R1, %R2
I1: sbb  %R3, %R3
I2: orl  $1 , %R3

     (b)
```

Figure 5.9: Predication using the **sbb** instruction: (a) original code, (b) optimized code.

The opportunities for **sbb** predication are limited, and the example in Figure 5.9 seems contrived and unlikely to occur. However, sequences like this are frequently used in comparison functions passed to the **qsort** library function. For example, the **strcmp** library call could benefit from this optimization. Unlike the **setcc** predication example above, **sbb** predication does not require profile information to be effective, since the fall-through and taken path both execute the same number of instructions. When the conditions are met, the optimization will always improve performance.

5.5 Zero-Extended Loads

To guarantee compatibility with all architectures, the HtoL conversion process follows all unsigned character loads with an "**and** $255, %reg" operation. The **and** instruction clears the upper part of the register, preserving only the byte loaded by the **ld_uc** instruction. Many architectures, including the x86, support a zero-extended character load instruction that makes the **and** instruction useless. Since the HtoL conversion process places the **and** instructions immediately following the **ld_uc** operation, it is relatively simple to detect this situation and delete the redundant **and** operations during Phase I annotation.

5.6 Bit Field Handling

Figure 5.10 shows another example of inefficient code introduced by the Lcode conversion. The code is generated for C source code that uses bit fields.

```
andl $8, %eax              andl $8, %eax
sarl $3, %eax
jne  label                 jne  label

     (a)                        (b)
```

Figure 5.10: Better bit field handling: (a) original code, (b) after optimization.

If **eax** is not live on either path of the branch, then the optimization can eliminate the right-shift. This simple optimization performed during Phase II pre-pass peephole optimizations. If multiple bit field operations exist, then the test optimization discussed earlier may convert the **and** instructions into **test** operations.

5.7 Pointer Post-Increment

Figure 5.11 shows another example of inefficient code introduced by the Lcode conversion. The code is generated from pointer post-increments in code such as "*ptr++".

```
I0: R1 <- (ptr);
I1: R2 <- R1;
I2: R1 <- R1 + 1;
I3: (ptr) <- R1;
I4: R3 <- (R2);
```

(a)

```
I0: R1 <- (ptr);
I4: R3 <- (R1);
I2: R1 <- R1 + 1;
I3: (ptr) <- R1;
```

(b)

Figure 5.11: Pointer post-increment optimization: (a) original code, (b) optimized code.

The optimized code schedules instruction **I4** above the pointer increment instruction **I2**, provided that the addresses in instructions **I4** and **I5** can be disambiguated. Instruction **I1** is then no longer necessary (provided **R2** is not live outside the sequence shown above) and can be deleted. Since the optimized code in Figure 5.11(b) is closer to the original source than the code in Figure 5.11(a), **I4** and **I5** always reference different addresses and the optimization is legal.

5.8 Multiple Return Statements

Lcode functions are limited to a single epilogue block containing the function cleanup and return statement. The single epilogue restriction is necessary for dataflow analysis and for the trace selection and tail duplication algorithms used in superblock formation. For functions that have multiple return statements, the Lcode contains jump instructions that branch to the common epilogue control block. Since the function cleanup on the x86 typically requires a single instruction to deallocate the stack frame, more efficient code can be generated by replacing the jump instruction with a copy of the epilogue. To minimize the interaction of this optimization with other parts of the IMPACT environment, it is performed at the end of Phase II after register allocation and post-pass scheduling.

5.9 Controlling Inappropriate Superscalar Optimizations

Some register-intensive optimizations in the machine independent optimizers have severely detrimental effects on the x86 architecture. For example, Lopti can perform strength reduction on operations that are linear functions (add, subtract, multiply) of induction variables within a loop by replacing the linear function with an increment of a new loop induction variable [19]. Figure 5.12 shows an example of this transformation.

```
loop:   ...
        R2 <- R1 * 160;
        ...
        R1 <- R1 + 1;
        jge loop;
```

(a)

```
loop:   ...
        R2 <- R2 + 160;

        R1 <- R1 + 1;
        jge loop;
```

(b)

Figure 5.12: Induction variable strength reduction for multiply instructions: (a) original code, (b) after induction variable strength reduction.

In Figure 5.12, the multiply of the induction variable **R1** is replaced by a less expensive addition operation. Unfortunately, the code in Figure 5.12(b) increases register pressure by extending the live range of **R2** so that it is live for the entire loop body. Additionally, the machine-independent version of this optimizations performs strength reduction for all linear functions, including add, subtract, and shift instructions to break flow dependences and increase scheduling freedom. On the x86 architecture, the advantages of additional scheduling freedom are greatly outweighed by the disadvantages of increased register pressure. In fact, many of the linear functions that are candidates for induction variable strength reduction are multiplies of two, four, or eight that would otherwise be merged into complex addressing modes and executed at no cost. These multiply instructions are common in loops that access memory using the induction variable as an array index.

Strength reductions such as the one in Figure 5.12 above are beneficial since they reduce the cost of performing the multiply operation. To reap these benefits but avoid the penalties associated with strength reduction for simple linear functions, Lopti now performs strength reduction only for complicated multiply instructions when the target architecture is the x86. Because Lopti is a machine-independent optimizer, it is important to make these changes as unobtrusive as possible. Fortunately, the machine specification (mspec) mentioned in Chapter 2 provides a convenient mechanism for describing machine-specific features, such as the scaled address modes supported by the x86. Lopti selects the algorithm to use for induction variable strength reduction based on the mspec information: if scaled address modes are supported, then only multiply instructions can be strength reduced. Otherwise, all linear functions of induction variables are candidates.

5.10 Register Saving Convention

Research has shown that the best register-saving convention is a mix of callee-save and caller-save registers [20]. However, this research was conducted on machines that have significantly more (at least 50%) registers than the x86. Still, the standard x86 UNIX register-saving convention follows this guideline and designates some of the x86 registers (**eax**, **ebx**, **ebx**) as caller-save and the others as callee-save.

The x86 IMPACT code generator designates all registers as caller-save. Since the architecture has so few registers, it seems likely that any reasonably large-sized function will use all of them. IMPACT aggressively inlines smaller functions to create larger functions that require more registers. Also, registers are allocated in a round-robin fashion to maximize the opportunities for post-pass scheduling. These optimizations tend to increase the number of registers that are used in a function, and the majority of functions use all of the available registers and generate spill code to compensate. The caller-save register convention is easier to implement than a hybrid technique, and several examples in the SPECint92 suite clearly benefit from the caller-save convention. For example, consider the function "eval" from the sc benchmark, shown in Figure 5.13.

```
double
eval(e)
     register struct enode *e;
{
    switch (e->op) {
     ...
      case '+':   return (eval(e->left) + eval(e->right));
      case '-':   return (eval(e->left) - eval(e->right));

      case '=':   return (eval(e->left) == eval(e->right));

     ...
  }

}
```

Figure 5.13: A section of the "eval" function from the SC benchmark.

Each case in the switch statement makes a recursive call to eval until the leaves in the expression tree are reached. With the caller-save register convention, no registers have to be maintained across the function calls. The standard hybrid register-save convention must save the scratch registers each time the eval is called, which can be significant because evaluating the case block does not require many other instructions.

Library functions called from IMPACT-compiled functions still use the hybrid register-save convention. Thus it is not necessary to save registers that are designated as callee-save in the hybrid calling convention (**eax**, **ebx**, and **edx**) across library function calls. Since the register allocator is not able to distinguish between library and other function calls, the redundant spill operations are removed in a separate pass in the code generator. This function uses a table of library function names to determine whether the function call is to a library routine or a procedure that is part of the program being compiled.

5.11 The Seventh Register

Figure 5.14 shows a standard x86 function entry and exit using the **esp** register as a base pointer, and Figure 5.15 shows the stack frame that is allocated by the standard function entry.

```
pushl %ebp
movl  %esp, %ebp
subl  $24, %esp

      (a)
```

```
movl %ebp, %esp
popl %ebp
ret

      (b)
```

Figure 5.14: Base pointer stack frame: (a) function entry, (b) function exit.

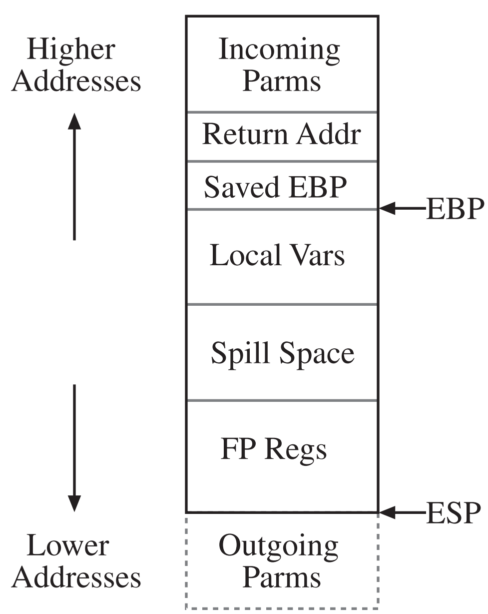

Figure 5.15: Stack frame layout for with base pointer stack frame.

Incoming parameters are accessed as positive offsets from the base pointer starting at **ebp**+8. Local variables, register spill locations, and floating point variables are accessed at negative locations from the **ebp** register. Outgoing parameters are pushed onto the stack before a function call, moving the stack pointer down as each argument is pushed. After the function returns, the stack pointer must be adjusted to deallocate the outgoing arguments and return the stack pointer to its original location. Figure 5.16 shows an example of a function call using the base pointer stack frame.

```
pushl $msg
pushl %ebx
pushl $_stderr
call  _fprintf
addl  $12, %esp
```

Figure 5.16: Function call using base pointer stack frame.

The base pointer and the stack pointer perform redundant functions in the stack frame shown in Figure 5.15. Since the stack frame size is known at compile time, incoming parameters, local variables, spill code, and floating point registers can all be accessed using fixed offsets from the stack pointer, freeing up the base pointer for use as another general-purpose register. At function entry, a single subtract instruction allocates the space on the stack. At function exit, the stack frame is deallocated by adding the size of the stack frame to the stack pointer.

However, the stack pointer can no longer move during outgoing parameter calls (i.e., **push** instructions cannot be used), since local variables and other values on the local stack frame would no longer be at the same offset from the stack pointer. To solve this problem, an area for outgoing parameters is reserved when the stack frame is allocated, as shown in Figure 5.17.

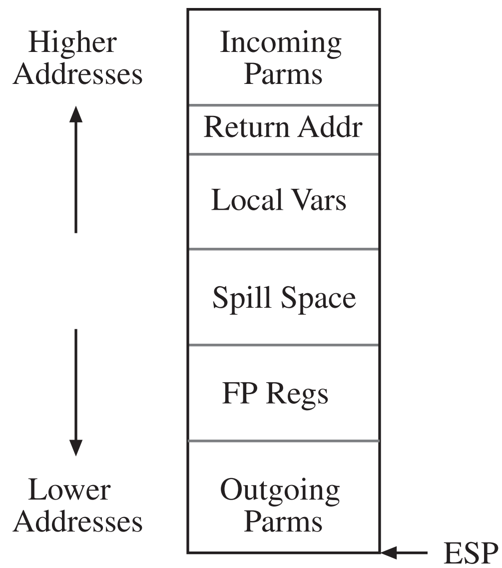

Figure 5.17: New stack frame.

The outgoing parameter area must be large enough to accommodate the largest number of parameters to any function called from the function allocating the stack frame. Fortunately, this information is calculated during CtoH conversion, and is accessible through a **define** operation in the Lcode representation. Figure 5.18 shows a function call with the new stack frame.

```
movl $msg, 8(%esp)
movl %ebx, 4(%esp)
movl $_stderr, (%esp)
call _fprintf
```

Figure 5.18: Function call using new stack frame.

Since the stack frame is statically allocated during function entry, parameters no longer have to be deallocated after a function call. This eliminates the **addl** instruction that was necessary in Figure 5.16 above. In addition, the **movl** instructions in the new stack frame may be cheaper to execute than **pushl**, since the **movl** needs only a store operation while the **pushl** requires a store operation and an operation to update the stack pointer. The K5 is one such processor that will execute **movl** instructions faster than **pushl** instructions. The **movl** instructions are larger than the **pushl** instructions (three bytes instead of one byte), but code expansion is less important than the savings in CPU cycles. Additionally, since the new stack frame uses fewer instructions for function entry and exit, and requires no cleanup after a function call, the code size may not be significantly different.

The real gain from the new stack frame is the availability of **ebp** as the seventh general-purpose register. It is important to view the additional register not as merely a gain of one register, but as a sixteen percent increase in the number of registers in the machine. The seventh register reduces spill code and improves the effectiveness of aggressive optimizations.

5.12 The Eighth Register

With the base pointer reclaimed as a general-purpose register, the only remaining register is the stack pointer, which appears to be necessary for accessing local variables and making function calls. This section describes a strategy for using the stack pointer as a general-purpose register by placing the local variables, spill space, and floating point registers in a static stack frame.

At function entry, the current stack pointer is saved in a global label. Incoming parameters to the function are accessed at positive offsets from the stack pointer and are loaded into registers. Once the incoming parameters have been loaded, the **esp** register may be allocated as a general-purpose register. However, the **esp** register cannot be used as an index register in a complex memory mode. This problem is handled using the *ill_reg* mechanism described in Section 3.5. Local variables and spill space accesses are converted from stack pointer references to fixed offsets from a GLOBAL_STACK_PTR. The GLOBAL_STACK_PTR is a label that points to a large statically allocated section of memory, as shown in Figure 5.19 below.

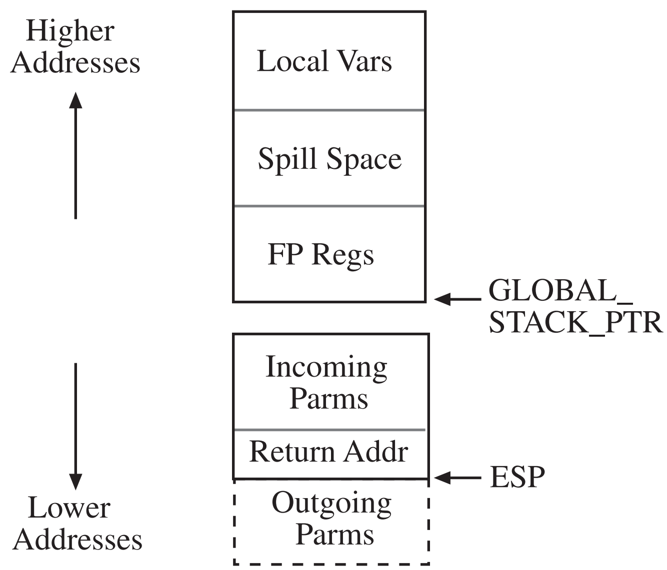

Figure 5.19: Eighth register stack frame.

As long as the eighth register function is a leaf function (i.e., the function contains no call instructions), the stack pointer is not required until the end of the function and can be allocated as a general-purpose register. The epilogue restores the original stack pointer, pops the return address from the original stack, and returns to the calling function. Even nonleaf functions can use the eighth register if the original stack pointer is restored before the outgoing parameters are pushed onto the stack. If the **esp** register is in use at the time of the function call, the register allocator will insert the necessary spill code to save it before the saved stack pointer is reloaded. Figure 5.20 shows an example of a function call from an eighth register function in which the **esp** register is live at the time of the call.

```
I0: movl %esp, GLOBAL_STACK_FRAME+76;

I1: movl SAVED_STACK_PTR, %esp
I2: pushl %ebx
I3: pushl $5
I4: call _foo

I5: movl GLOBAL_STACK_FRAME+76, %esp
```

Figure 5.20: Function call from eighth register function.

Instructions **I0** and **I6** are only necessary when **esp** is live at the time of the function call. Instruction **I1** restores the original stack pointer that was saved in the function prologue. Instructions **I2** and **I3** push the outgoing parameters onto the stack. Since the function prologue does not allocate stack space for the outgoing parameters, **pushl** instructions are required to store the argument and update the stack pointer. This technique is similar to the function call used in the base pointer stack frame described in Section 5.11, and suffers from many of the same disadvantages. However, unlike the base pointer stack frame, parameters do not need to be deallocated since the original stack pointer is not needed after the function returns. If another function call must be made, the original stack pointer can be loaded from the SAVED_STACK_POINTER location.

The main caveat in the function call strategy described above is that no eighth register function can ever call another eighth register function, since they both use the same global stack frame. The callee function would overwrite the local variables and spilled registers of the caller function. One solution to this problem is to use local static stack frames for each eighth register function, but this has a negative effect on data cache performance. Another solution is to guarantee that the functions called from an eighth register function are not also eighth register functions. This requires a complicated call graph analysis and a heuristic for selecting the eighth register functions. The current implementation does not use the call graph, but simplifies the problem by transforming only leaf functions and functions that call only library routines. Future work may implement a more aggressive algorithm if the technique significantly improves performance.

When using the stack pointer as a general-purpose register, the effect of interrupts for handling I/O events and task switching must be considered. Since the **esp** register will most likely not point to the user stack, it might be possible for interrupts that save state information on the stack to crash the system. Fortunately, the x86 UNIX implementations execute interrupts at a different processor privilege level. When the interrupt occurs, the processor switches to a new stack when it changes privilege level [3]. The stack swapping is completely transparent, and the value of **esp** in the user program is completely isolated from the stack environment of the interrupt.

Eighth register functions are only slightly more expensive than the standard seventh register functions described in the previous section. At function entry, the eighth register function uses a store instruction to save the stack pointer instead of a subtract operation to allocate the stack frame. At function epilogue, the eighth register function loads the saved stack pointer while the standard function deallocates the stack frame with an add operation. The overhead for function calls requires an extra load instruction to restore the saved stack pointer and extra overhead because push instructions must be used instead of stores for the outgoing parameters. This overhead makes it desirable to limit the number of calls in eighth register functions.

Since the stack frame is global for all eighth register functions, the impact on cache performance is probably negligible. Private static stack frames might decrease the cache hit ratio by reducing the frequency of stack references, and this should be considered if the optimization is made more aggressive.

5.13 Improved Function Entry

The new stack frame described in Section 5.11 decrements the stack pointer at function entry and then loads the incoming parameters from the stack. Because the stack pointer is decremented first, the function entry creates a flow dependence between the allocation instruction and the instructions that load the incoming parameters. Figure 5.21 shows this dependence, along with a simple instruction reordering to eliminate it.

```
I0: subl $40, %esp        I0: movl 4(%esp), %edi

I1: movl 44(%esp), %edi   I1: movl 8(%esp), %esi
I2: movl 48(%esp), %esi   I2: subl $40, %esp

        (a)                         (b)
```

Figure 5.21: Scheduling function entry to remove flow dependence: (a) original function entry, (b) improved function entry.

The improved function entry replaces the true dependence on the **esp** register between instruction **I0** and the other two instructions with an anti-dependence. The anti-dependence can be eliminated by register renaming in the K5 hardware, and will execute in one cycle as opposed to two in the original code.

5.14 Built-in Compiler Functions

Library functions that are very small can be dominated by the function call overhead. For functions such as **strcmp** and **strlen**, it may be beneficial for the compiler to automatically inline them as compiler built-in functions. Other compilers (such as gcc) use compiler built-ins to utilize the special string-handling instructions of the x86, such as **movs** and **scas**. On processors such as the K5 and P6, these instructions trap to microcode routines and are likely to be more expensive than sequences of simple instructions. Profile information indicates that strings passed to **strcmp** and **strlen** are typically very short (approximately four characters), which implies that the function call overhead may be significant.

Built-in functions are implemented by replacing the function call in Phase II annotation with a sequence of simpler operations that perform the same function. Annotation is done very early in Phase II, and thus the instructions that store outgoing parameters and move the function result are easy to locate and integrate into the inlined code. The code for the built-in function may be optimized and scheduled with the surrounding code by later passes in Phase II.

Another important built-in function is **alloca**, used in the gcc benchmark from the SPECint92 suite [17]. Alloca allocates temporary space on the local stack frame that is automatically freed when the function exits. A function to emulate this behavior is included with the gcc benchmark, but it is rather slow. The code uses malloc to allocate the memory, and employs a heuristic to determine approximately when the function has ended and the memory should be deallocated. If alloca is implemented as a built-in function, only a few instructions are required to move the stack pointer to allocate the requested memory.

Since the built-in alloca function moves the stack pointer, the function cannot reference the local stack frame at fixed offsets from the **esp** register. Thus, the function must use the base pointer stack frame described in the beginning of Section 5.11. However, the alloca stack frame uses a shifting outgoing parameter space so that push instructions and outgoing parameter deallocation are not necessary. Figure 5.22 shows the alloca stack frame.

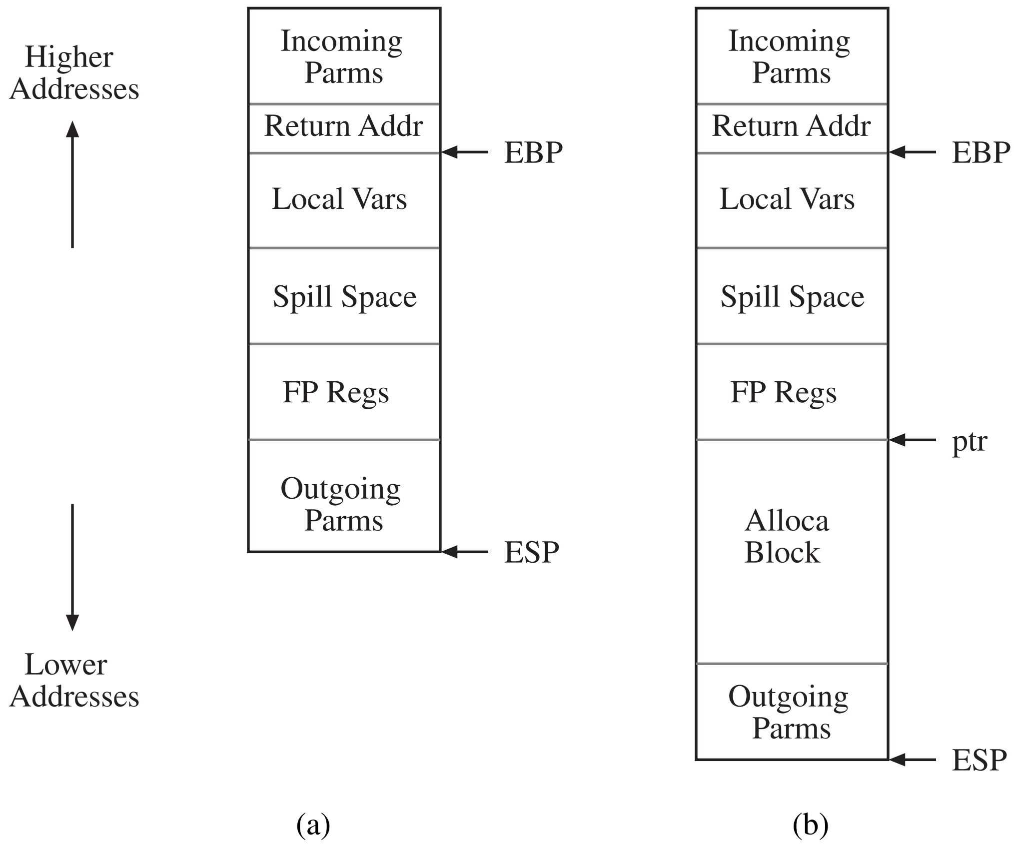

Figure 5.22: The alloca stack frame: (a) before **alloca** call, (b) after **alloca** call.

The alloca call returns a pointer (marked as "ptr" in Figure 5.22) to the beginning of the outgoing parameters section using an **lea** instruction. The stack pointer is moved downward by subtracting the size of the alloca request. Because outgoing parameters are accessed with positive offsets from the stack pointer, this effectively moves the outgoing parameter space with the stack pointer. Since the outgoing space has been allocated in the stack frame, arguments can be passed without using **pushl** instructions, and the parameter space does not need to be deallocated. Local variables, spill space, and floating-point registers are all accessed using negative offsets from the base pointer.

Since IMPACT uses a caller-save convention, spill code must be inserted to maintain the base pointer across function calls. The register allocator will not insert this spill code for macro registers, so the save and restore instructions are explicitly inserted by the code generator.

6. PERFORMANCE EVALUATION

The performance of the IMPACT x86 code generator was evaluated on a 90 MHz Pentium system running Unixware version 1.1. Table 6.1 shows the hardware features of the evaluation machine.

Table 6.1: Hardware features of evaluation machine.

Processor:	90 MHz Intel Pentium
Main Memory:	48 MB RAM
Cache:	256K L2
Disk:	520 MB Connor IDE

Tables 6.2 and 6.3 below show the performance of the Intel Reference C Compiler (version 2.0.8) and GNU gcc (version 2.5.8) compared to the IMPACT x86 compiler. Intel publishes SPEC results with the Intel Reference C Compiler. The flags specified in the SPEC reports for the compiler (-tp p5 -ip -dn) are used here. These flags specify the Pentium as the target processor (-tp p5), interprocedural analysis for inlining and function cloning (-ip), and static linking (-dn). Additionally, the Intel Reference compiler supports feedback directed optimizations (profiling), as does IMPACT. The "Intel Profile" column in Table 6.2 reflects the performance of the compiler when the benchmarks are profiled on

the same inputs used for IMPACT profiling. The programs are recompiled using the Intel feedback-directed optimizations. The "IMPACT Profile" columns in Tables 6.2 and 6.3 reflect the performance of the IMPACT compiler using feedback-directed superscalar optimizations (with the Ltrace superscalar optimizer). The baseline IMPACT numbers use profile information for inline function expansion and classic code optimizations. The flags used for gcc specify the highest level of optimization for the compiler (-O2 -fomit-frame-pointer -m486 -static). Gcc does not optimize specifically for the Pentium, and thus the machine type was specified as 486 (-m486).

The benchmarks used in the evaluation include the six benchmarks from the SPECint92 benchmark suite [17] and five other common application programs. Table 6.2 shows the speedup comparison for the three Unixware compilers, and Figures 6.1 and 6.2 show graphs that present the same information.

Table 6.2: Speedup comparison of Unixware compilers.

Benchmark	*Intel*	*Intel Profile*	*IMPACT*	*IMPACT Profile*	*gcc*
espresso	1.00	1.11	1.00	1.02	0.82
li	1.00	1.00	1.15	1.21	0.83
eqntott	1.00	1.00	1.11	1.12	0.61
compress	1.00	0.99	0.96	0.96	0.92
sc	1.00	1.02	1.04	1.06	0.35
gcc	1.00	0.99	1.10	1.10	1.13
SPEC Avg.	1.00	1.02	1.06	1.08	0.78
cmp	1.00	0.97	0.84	1.10	0.90
eqn			1.00	1.10	0.75
grep	1.00	1.04	0.82	0.87	0.75
lex	1.00	0.99	0.95	1.00	0.95
wc	1.00	1.09	1.17	1.19	0.75
Application Avg.	1.00	1.02	0.96	1.05	0.82
Total Avg.	1.00	1.02	1.01	1.07	0.80

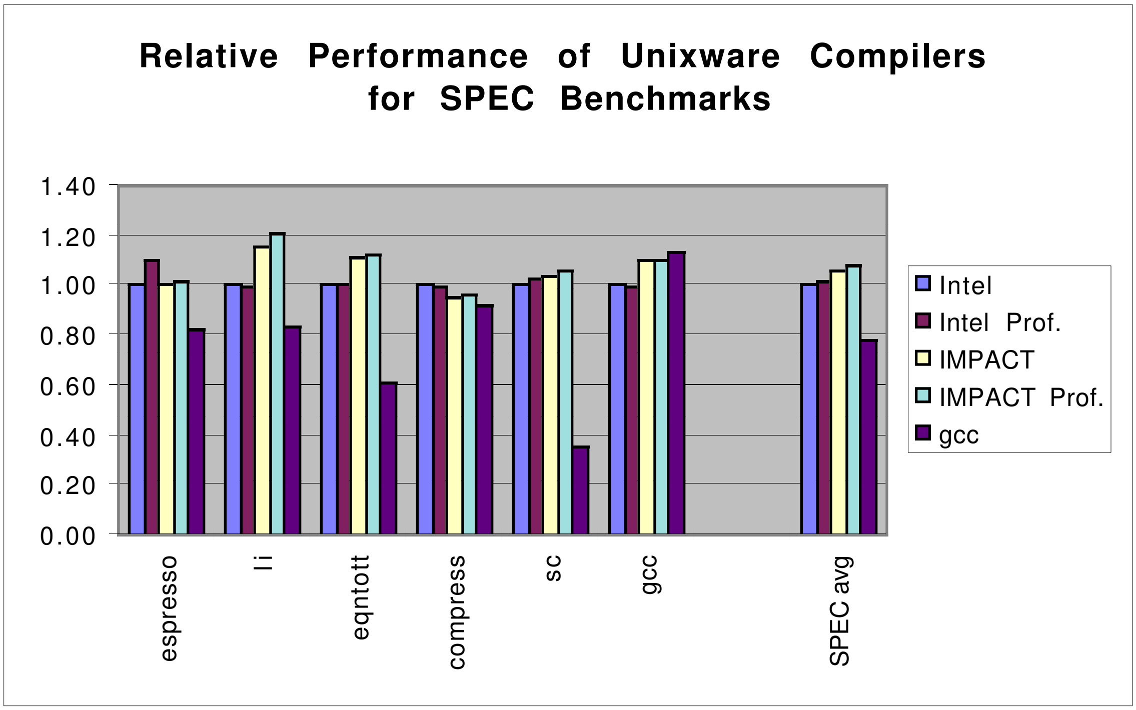

Figure 6.1: Relative performance of Unixware compilers for SPEC benchmarks.

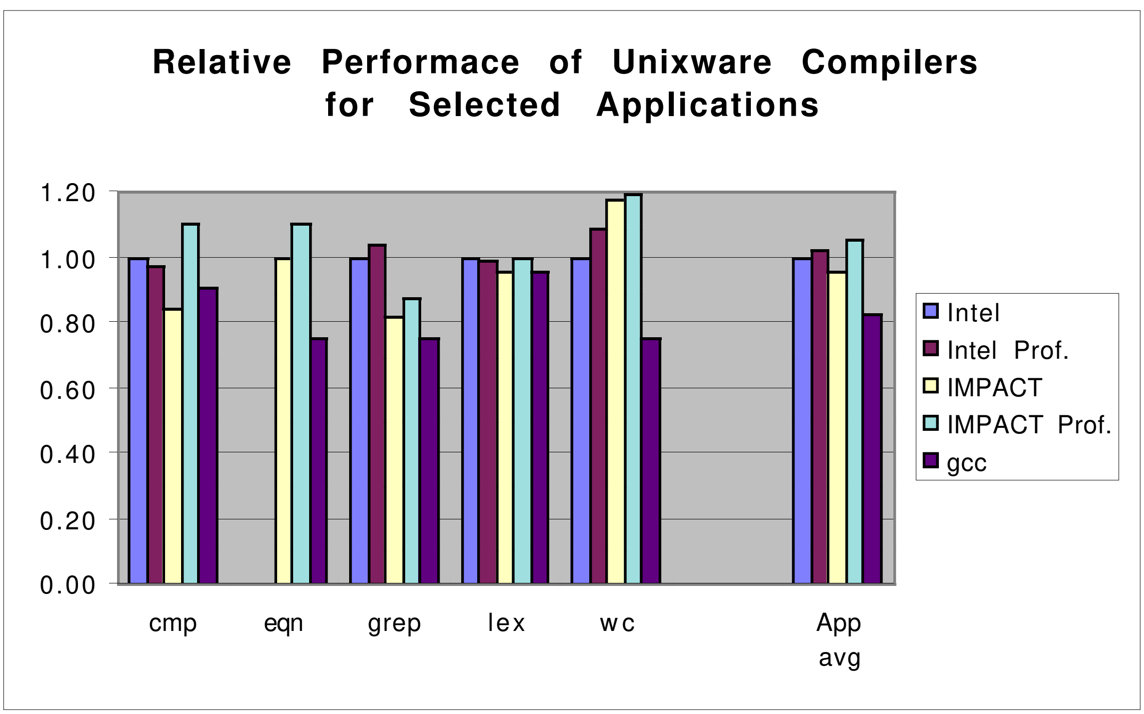

Figure 6.2: Relative performance of Unixware compilers for selected applications.

The Intel Reference Compiler failed to correctly compile the **eqn** application program. Therefore the **eqn** speedup in the above graph uses IMPACT as the reference. Also, the Intel Reference Compiler failed when compiling **eqntott** with profile information, and thus the **eqntott** performance without profiling was used in Table 6.2 and Figure 6.1. The Intel Reference Compiler failed when using feedback-directed optimizations and interprocedural analysis (-ip) for the **espresso** and **gcc** benchmarks, and thus interprocedural analysis was disabled to generate the numbers for these benchmarks in the "Intel Profile" column. Due to time constraints, the IMPACT numbers for the **gcc** benchmark in Table 6.2 and Figure 6.1 above do not use feedback-directed superscalar optimizations. Finally, the gcc compiler uses an additional flag (-fwritable-strings) to correctly compile some badly written code in the **sc** benchmark. This may partially explain the especially poor performance of the gcc compiler on the **sc** benchmark.

The IMPACT x86 compiler exceeds all Unixware compilers when compiling for the SPEC benchmarks, and is very competitive with the Intel Reference Compiler for the selected application benchmarks. When the IMPACT superscalar optimizations are enabled, IMPACT is the fastest compiler for nine of the eleven benchmarks programs and the fastest in both the SPEC average and the application average. The results are even more impressive considering that IMPACT does no special scheduling for the Pentium processor. The IMPACT executables used in this study were scheduled for the 486 processor. On new processors such as the K5, we expect that better scheduling and IMPACT's advantage in superscalar optimizations will further widen the gap between IMPACT and the other compilers.

Table 6.3 compares the performance of gcc and IMPACT on the Linux operating system for the SPEC benchmarks, and Figure 6.3 shows a graph that presents the same information.

Table 6.3: Speedup comparison of Linux compilers.

Benchmark	*gcc*	*IMPACT*	*IMPACT Profile*
espresso	1.00	1.18	1.18
l i	1.00	1.39	1.43
eqntott	1.00	1.74	1.76
compress	1.00	1.14	1.13
sc	1.00	3.44	3.53
gcc	1.00	1.04	1.04
avg.	1.00	1.66	1.68

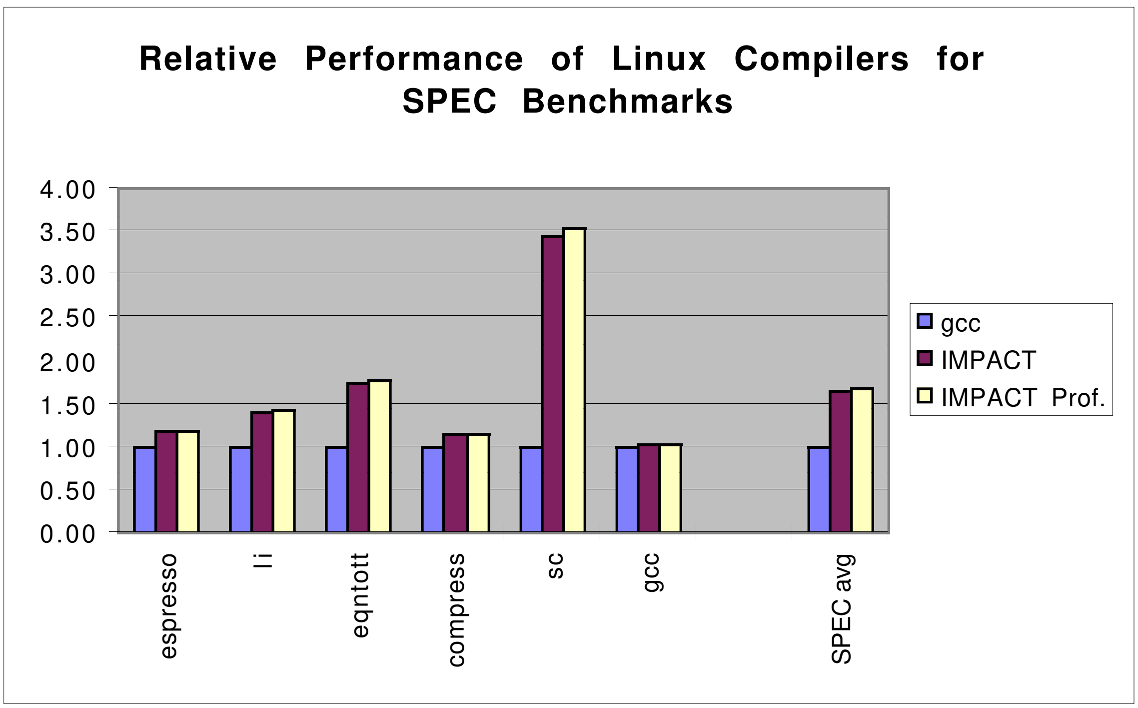

Figure 6.3: Relative performance of Linux compilers for SPEC benchmarks.

Table 6.3 shows that IMPACT is more than 60% faster than gcc for the SPEC benchmarks. As mentioned previously, a large part of this advantage is due to the tuning in the IMPACT x86 code generator for the SPEC suite. Also, gcc appears to have some difficulty in producing an efficient executable for the **sc** benchmark.

7. CONCLUSIONS AND FUTURE WORK

This thesis discusses the modifications made to the IMPACT compiler to produce efficient code for the x86 architecture. The thesis is intended to be used as the reference document for the IMPACT x86 code generator for future work on the project. The performance of the IMPACT x86 compiler is shown to exceed that of several commercially and publicly available compilers for the Pentium processors, including the compiler that Intel uses for generating SPEC performance numbers. It is expected that the performance gap will widen when IMPACT is run against these compilers on superscalar x86 machines such as the AMD K5.

The basic optimizations in the code generator are reaching a state of diminishing returns. The work described in this thesis has brought the code generator to the point at which it is extremely competitive with other compilers for 486 and Pentium machines. Future work will focus on register pressure heuristics to guide superscalar optimizations and on effective scheduling for superscalar x86 processors. As the x86 market moves towards wide-issue superscalar processors such as the AMD K5 and the Intel P6, these superscalar optimizations will become increasingly important.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture,* pp. 266-275, Toronto, Canada, May 1991.

[2] *Intel486 Microprocessor Family Programmer's Reference Manual.* Intel Co., 1992.

[3] *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual.* Intel, Co., 1993.

[4] W. M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[5] W. F. Dugal, "Code scheduling and optimization for a superscalar x86 microprocessor," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[6] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[7] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[8] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," *IEEE Transactions on Computers*, vol. 41, pp. 1537-51, December 1992.

[9] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229-248, January, 1993.

[10] P. P. Chang and W. W. Hwu, "The Lcode language and its environment," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, Internal Report, April 1991.

[11] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, Tech. Rep. CRHC-91-18, May 1991.

[12] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[13] Intel Co., "Optimizations for Intel's 32-Bit Processors," Intel Co., Application Note AP-500, Intel Co., February 1994.

[14] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, pp. 8-21, January 1985.

[15] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics,* vol. 4, no. 4, pp. 81-85, August 1984.

[16] Y. Wu, "Strength reduction of multiplications by integer constants," *ACM SIGPLAN Notices*, vol. 30, no. 2, February 1995.

[17] SPEC Benchmark Suite, 1992.

[18] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 296-403, Tokyo, Japan, June 1986.

[19] P. P. Chang, S. M. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software: Practice and Experience*, vol. 21, pp. 1301-21, December 1991.

[20] J. W. Davidson and D. B. Whalley, "Methods for saving and restoring register values across function calls," *Software: Practice and Experience*, vol. 21, pp. 149-65. February 1991.