OPTIMIZATION AND EXECUTABLE REGENERATION IN THE IMPACT BINARY REOPTIMIZATION FRAMEWORK

BY

MICHAEL STEPHEN THIEMS

B.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

ACKNOWLEDGMENTS

Many people have provided invaluable support throughout my education, in my research, and in the writing of this thesis. I wish to thank my advisor, Professor Wen-mei Hwu, for his gifted teaching and advice that has helped guide me to where I am and to where I am going in my career. John Gyllenhaal served as a mentor and provided many useful suggestions in the development of this system and in the writing of this thesis. Besides developing the IMPACT scheduler manager and machine description technology, he also had a hand in many other parts of the IMPACT infrastructure. Matthew Merten, who developed x86toM and with whom I have spent many hours working over the past year, has also been a great help in working out the many problems in this system. The PEwrite program is based on binary profiling work done by John Sias, Chris George, and Guanyao Cheng. As my cubicle-mate, Qudus Olaniran provided many useful insights after listening to both my ideas and my frustrations. Justin Donoho helped build both the original versions of the NT benchmarks and the testing environment. David August provided data flow support, and Dan Connors helped with very practical thesis advice. Many thanks are also due to the past and present members of the IMPACT group, whose excellent compiler infrastructure has made this work possible.

I owe so much to my parents for their loving support and guidance over the entire course of my life. Finally, I wish to thank my wife Natalie for lovingly helping me to keep my priorities straight while always remaining my biggest fan. The way that my family believes in me has been a constant source of encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Advances in microprocessor design have continued to increase the performance of computer systems while extending the life span of architecture families. The rapid pace of development has made the simultaneous presence of many different processor versions within the same basic architecture an important fact in the computer market. However, the binary executable program, along with binary dynamic-link libraries, remains the dominant form for computer software distribution. In binary form, a program is targeted towards a single processor version or towards a blend of different processors. For this reason, programs fail to achieve maximum performance on many, or even all, processor versions.

Performance-enhancing features of microprocessors may be grouped in two general categories. *Architectural* features add new instructions or registers to the processor design. Until such extensions become supported by the majority of processor versions in the market, many software vendors will not support the new features to avoid alienating consumers with older machines. *Microarchitectural* features do not change the programmer-visible properties of the processor, but instead seek to achieve a certain cost-performance point by altering how the architecture is implemented. To achieve optimal performance, different implementations often require different sequences of instructions that perform the same functional tasks. In all of these situations, the end user would benefit from having a binary program that is optimized specifically for his or her particular processor version. Manufacturers of different processor versions would also benefit, since specially optimized executables would better highlight the performance of their products.

Although traditional compiler technology may be adapted to generate more optimal code for a certain processor [1], source code must be available as input to a compiler. For most commercial software, the high-level source code is available to neither consumers nor even microprocessor manufacturers. The goal of the IMPACT binary reoptimization framework is to apply compiler technology to programs starting from binary form and thus to reoptimize the programs for a specific processor version. In particular, the proposed framework takes advantage of the established compiler technology tools of the IMPACT compiler system [2], [3].

The Intel x86 instruction set [4] and Microsoft's 32-bit Windows operating systems (Windows 95 and Windows NT) are well-established standards in the market. A large body of binary-distributed commercial software exists for this platform. Furthermore, the platform continues to be advanced with new processors and new operating system versions, while backward compatibility is maintained for economic reasons. More recently, x86 processors produced by companies other than Intel have also gained significant market share, having different microarchitectures and even different instruction set extensions. Competing instruction set extensions to improve three-dimensional graphics processing performance from Intel [5] and AMD [6] may create an environment in which no single set of extensions will ever become the standard. For these reasons, the Windows/x86 platform was chosen as an appropriate one for which to develop the binary reoptimization framework. However, the components and techniques developed may be viewed as specific instances of a general framework, which could be applied to other processor and operating system platforms.

The primary purpose of this thesis is to describe the backend of the IMPACT binary reoptimization framework, which involves optimization and executable regeneration. By detailing the environment in which optimization is performed, this thesis provides information necessary for the further development of optimizations in the framework. By explaining how new binary executables are generated for the optimized programs, it offers insight on how to extend the functionality for enhanced architectures, and how to adapt it to a different architecture. The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the reoptimization framework. Chapter 3 provides background on the intermediate representation utilized. Chapters 4, 5, and 6 describe the different steps that make up the backend of the framework. Chapter 7 validates the system and its implementation by evaluating the performance impact of a number of example optimizations. Finally, Chapter 8 discusses conclusions and possible future work.

2. SYSTEM OVERVIEW

2.1. Input Requirements

The structure of a Windows 32-bit executable file is the Portable Executable (PE) file format [7], [8], [9]. A PE file contains a number of sections, including binary images of the code and data segments of the program. Several sections, such as code and data, are required by the operating system to correctly execute the program. In addition to the required sections, additional information may be present in the file that assists in debugging or that might be needed under special circumstances. One such additional section is the *base relocation table*, which is used by the operating system to adjust addresses in the program image if the program cannot be loaded at its preferred base address. This table indicates the location of all pointers that are present in the binary, and so it is also useful to the proposed reoptimization system to distinguish between pointers and other data. It is necessary to make this distinction in order to successfully convert the program to the intermediate representation; therefore, the base relocation table is required by the proposed system. A symbol table may also be present in the executable, though it is not required by the proposed system. If present, the information provided by the table is used to supplement the system's ability to locate the beginning of functions in the program being reoptimized.

2.2. Processing Steps

Figure 2.1 outlines the process of reoptimizing binary programs within the proposed framework. The complete system consists of three separate programs, called x86toM, Lbx86,

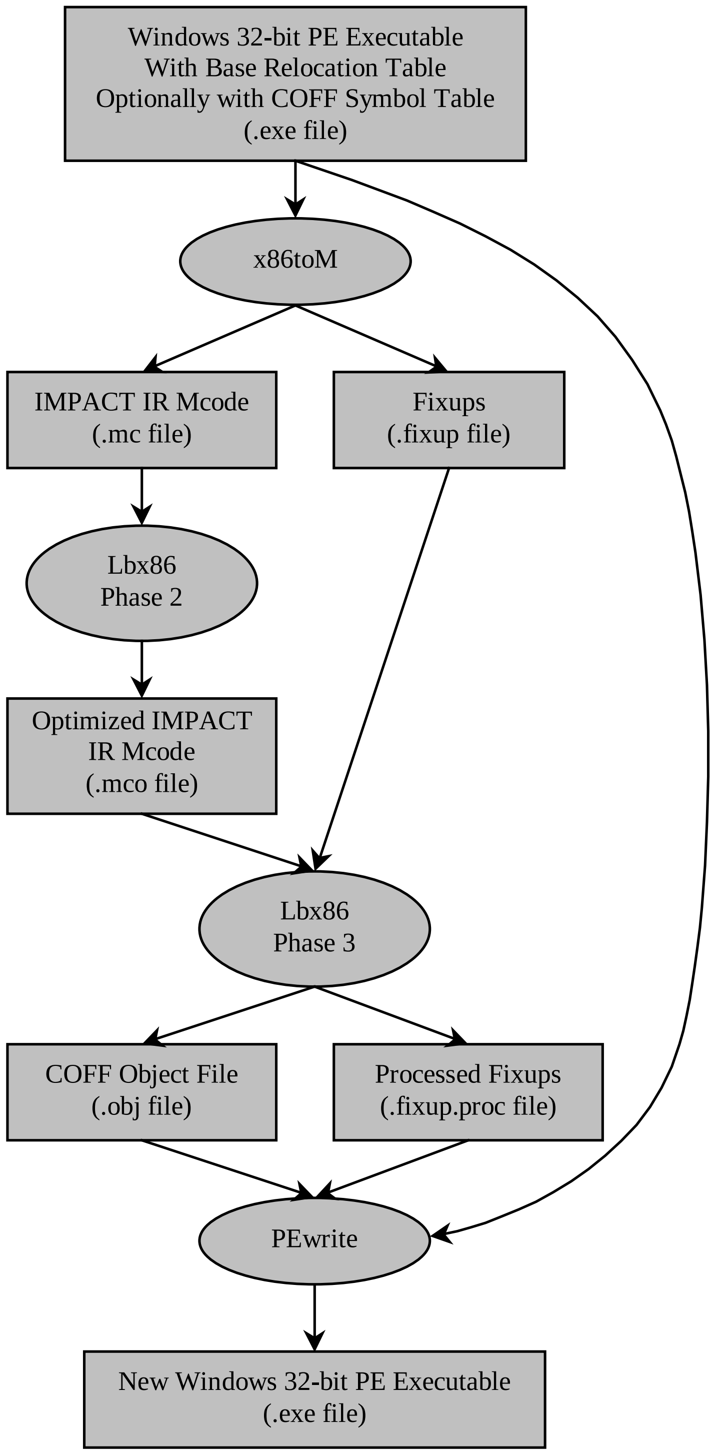


Figure 2.1 Binary reoptimization steps

and PEwrite. The Lbx86 program itself consists of two phases, which may be executed together or separately. The four processing steps are executed sequentially, and intermediate results are communicated from one step to the next in the form of files. Note that if both phases of Lbx86 are executed at once, the corresponding intermediate file (.mco) is not produced.

2.2.1. Conversion to intermediate representation

In the first step, the original executable file is processed by x86toM, which converts the code and data sections into *Mcode*, an intermediate representation used by the IMPACT compiler [10], [11]. This conversion process involves decoding binary machine instructions while discovering the control flow structure of the program. The corresponding Mcode representation of the code section is created, including the grouping of the individual operations into control blocks and functions. Elements of data, which are often present in the code section in addition to the data section, are also converted to Mcode.

Besides the code and data sections, several other sections often exist in the binary. Since these others will not be significantly modified by the proposed system, they are not converted to Mcode. References from converted sections to addresses in nonconverted sections are treated in the Mcode representation as external references to intrinsically defined symbolic names. For instance, references to the *read-only data section* use the label `_section_rdata`, which is then properly resolved at link time. Similarly, the nonconverted sections may contain references to addresses in the converted sections. Such pointers will also require modification at link time. To handle the latter case, a *fixup* file is also produced by x86toM, containing a mapping of the pointers from nonconverted sections to Mcode symbolic names.

The output produced by x86toM provides input to the remaining steps in the proposed system. However, the x86toM program is not the topic of this thesis. For more details on its operation, see [12].

2.2.2. Optimization and executable regeneration

The remaining steps of the binary reoptimization framework are described in detail in this thesis. The next two steps are performed by the Lbx86 program. Because x86toM takes on the role historically filled by phase 1 of an IMPACT code generator [11], these next two steps are called phase 2 and phase 3 in Lbx86. As is the historical case with phase 2 of IMPACT code generators, the primary function of Lbx86 phase 2 is to perform optimization on the program code. The program is represented as Mcode throughout this phase. Optimizations performed during this phase may include traditional machine-independent optimizations and processor-specific optimizations along with rescheduling.

During Lbx86 phase 3, the Mcode representation is converted back into a binary object file. A corresponding text assembly file is also generated for debugging purposes. At this time, the fixup file is also processed to convert Mcode symbolic names into object file symbolic names.

Finally, the PEwrite step forms a new executable file. A copy of the original executable is made, with the original code and data sections replaced by the optimized and converted versions. The only change made to the sections that were not converted is the linking of symbols to addresses as specified by the processed fixup file.

3. MCODE FORMAT

3.1. Overview

The machine-dependent, low-level intermediate representation used by the IMPACT compiler is known as Mcode. The format was designed to be flexible so that it could be used to represent the instruction set of any target processor. It typically provides a one-to-one mapping to the instructions of the specific target processor's machine language, while also providing facilities to represent data and control flow. The format allows for a variable number of source and destination fields, a processor specific opcode mnemonic in addition to a functional opcode mnemonic, and a variable length list of attributes that provide extra information.

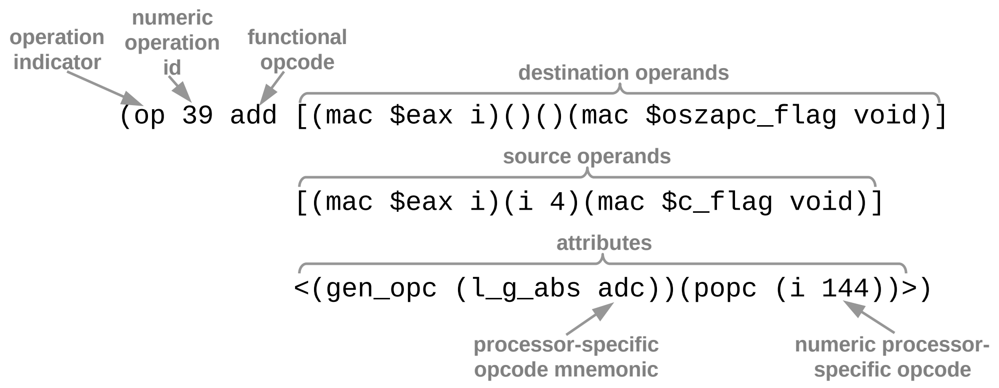
Unlike traditional compiler systems that only need to represent those instructions used by the compiler, the binary reoptimization system has no control over the subset of instructions that will need to be processed. Any user-mode instruction could appear in an application binary. It is therefore necessary that the proposed system be capable of representing the entire x86 user instruction set. To enable dependence and data flow analysis, all register accesses by an instruction should be explicitly represented in its Mcode representation. For this system, it was determined that a maximum of four destination fields and seven source fields are necessary to achieve this requirement. This format allows for accurate representation of complex instructions that may read or write many operands, such as the reads and writes to the various flag registers in the x86 architecture.

It should be noted that this particular Mcode format is different from the one used by Lx86, the x86 code generator for IMPACT's standard compilation path [1]. However, the challenges faced in the development of Lx86 have influenced this new design.

3.2. Instruction Format

3.2.1. Typical register instructions

An example of an instruction represented as a Mcode operation is shown in Figure 3.1, with annotation to clarify the format. The particular instruction being represented is an *add with carry*, and would appear in Intel's x86 assembly language as "`adc eax, 4`." The instruction has been classified for functional purposes as an *addition* (`add`) instruction.



```
(op 39 add [(mac $eax i)()()(mac $oszapc_flag void)]
           [(mac $eax i)(i 4)(mac $c_flag void)]
           <(gen_opc (l_g_abs adc))(popc (i 144))>)
```

Figure 3.1 Mcode representation of an *add with carry*, with format annotation

The first set of square brackets contains the destination operands. This particular instruction writes its results into the `eax` register, as represented by the first destination operand. The Mcode operand in this case, as well as many others, is of *macro* type (`mac`). This type allows the particular Mcode format to define a set of machine-specific operands that correspond directly to features of the target machine. The third element of the macro operand provides further information about the type of the macro operand; in this case, the macro type is integer (`i`), since the x86 `eax` register holds a 32-bit integer. This instruction also implicitly modifies six of the condition code flags: *overflow*, *sign*, *zero*, *auxiliary carry*, *parity*, and *carry*. This write to the flags is represented by the fourth destination operand, which is also a macro, whose name (`oszapc_flag`) indicates that it represents all six of the aforementioned flags. The `void` macro type is used in this case since the collection of six bits does not fit any of the standard types of the Mcode format, which include `i` for *integer*, `f` for *float*, and so on. The empty sets of parentheses denote unused destination operands.

Inside the next set of square brackets are the source operands. This instruction adds 4 to the contents of the `eax` register, and then adds 1 if the carry flag is set. The first source operand represents the fact that the `eax` register is read in addition to being written. The next operand is the immediate source, a constant integer (`i`) of value 4. The implicit read of the carry flag is also modeled explicitly, but because only the carry flag is read, the name of the macro operand is simply `c_flag`. In this case, the last four source operands are not printed out in Mcode, implying that they are unused.

The set of angle brackets at the end of each Mcode operation encloses the attributes of the instruction. This particular instruction needs only the two attributes required by every operation. The first is the processor-specific opcode mnemonic, called `gen_opc`, which is provided as a text string and used only for readability. Another required attribute is the `popc`, a numeric value used internally by the system that corresponds to the `gen_opc`. This will be explained further in Section 4.2.

This example also illustrates a general principle followed in the design of this particular Mcode format. The explicit operands that would appear in the assembly code representation of an instruction are generally mapped in order to the first few destination or source operands in Mcode. This allows for easier readability and guides the uniform placement of operands for different instruction types. Implicit operands are placed in the otherwise unused operand positions; the specific positions used vary for different instruction types.

3.2.2. Memory access instructions

Because x86 is a *CISC* (*complex instruction set computer*) architecture, a memory location can be a source, a destination, or both, in many different types of instructions. In this architecture, there exists a uniform manner for specifying memory addresses that is common to most instructions. The four components that make up such an address specification are referred to as the *base*, *index*, *scale*, and *displacement*. Figure 3.2 illustrates the Mcode representation of the instruction "`or BYTE PTR [eax+ebx*4+24], 64`," in which a single memory location is both a source and a destination. In this memory address, `eax` is the base register and `ebx` is the index register. The index register is scaled by a factor of 4, and a displacement of 24 is also present.

```
(op 908 or [()()()(mac $oszapc_flag void)]
           [(mac $addr void)(i 64)()(mac $eax i)(mac $ebx i)(i 4)(i 24)]
           <(mem_size (i 1))(mem_read_write)
            (gen_opc (l_g_abs or))(popc (i 1584))>)
```

Figure 3.2 Mcode representation of a *logical or* to memory

The way in which an instruction accesses memory is not explicitly represented by the Mcode operands. Instead, an attribute is used to specify the type of memory access performed, either `mem_read`, `mem_write`, or `mem_read_write`; the latter case indicates that memory is both read and written. The `mem_size` attribute further specifies the number of bytes of memory accessed. The proposed system uses the `addr` macro operand to indicate the presence of a memory address specification in an instruction. For instance, if this example had been a register-immediate form of the `or` instruction, a register macro operand would have appeared as both the first source and the first destination. In the memory-immediate example, the `addr` macro is used instead as a placeholder in the first source position. The corresponding address specification is always represented by the last four source operands in the Mcode representation. Note that these operands always appears as sources; the operands that specify a memory address are always read, even if the memory location indicated by that address is being written.

3.2.3. Exceptions to the typical instruction format

Several types of instructions in the x86 instruction set do not conform well to the uniform operand layout principle. The *string move* instruction is one example, as shown in Figure 3.3. In the assembly language representation, "`rep movsd`," no operands are specified; they are all implicit. The presence of the *repeat* prefix on the instruction makes it a one-instruction loop, so it will execute the number of times specified by the `ecx` register. This is also evident in the Mcode, in which the `ecx` loop control operand is both a source and a destination, because it must be read to check the loop boundary and written for a decrement.

Register `edi` contains the destination memory address for the string move; it is read to reference memory and written to increment or decrement. The `esi` register behaves just like the `edi` register, but contains the source memory address. Note that the *direction flag* (d_flag) is read to determine whether to increment or decrement the memory address registers.

```
(op 302 mov [(mac $edi i)(mac $esi i)(mac $ecx i)]
            [(mac $edi i)(mac $esi i)(mac $ecx i)()()(mac $d_flag void)]
            <(mem_size (i 4))(mem_read_write)(str_inst)
             (gen_opc (l_g_abs movs))(popc (i 1472))>)
```

Figure 3.3 Mcode representation of a *string move*

As in the previous example, the mem_read_write attribute identifies the memory access type. However, in this case, the mem_size attribute indicates the access size of a single iteration of the repeated instruction, which is also the amount by which esi and edi are incremented or decremented after each iteration. Finally, the use of the str_inst attribute clearly identifies this as a string instruction, which is another way that this can be distinguished from a normal *move* as the functional opcode might suggest. This string move illustration shows that even complex, nonconforming instructions can be accurately represented in Mcode.

3.3. Control Flow Organization in the Mcode Format

3.3.1. Program organization and control flow information

Beyond the ability to represent individual instructions, the Mcode format also represents the organizational structure of the program. Figure 3.4 shows an example taken from the Microsoft Visual C++ 5.0 library function memmove. The code for a complete program is separated into individual functions, each represented by a function and end

```
(function _memmove_453_ 0.000000 <L>
    <(jump_tbls (i 6)(i 5)(s_l_abs "renamed"))>)
 ...
  (cb 21 0.000000 [(flow 1 30 0.000000)(flow 0 22 0.000000)])
   ...
    (op 25 sub [(mac $ecx i)()()(mac $oszapc_flag void)]
               [(mac $ecx i)(i 4)]
               <(gen_opc (l_g_abs sub))(popc (i 2336))>)
    (op 26 blt_u [] [()(cb 30)(mac $c_flag void)]
                 <(gen_opc (l_g_abs jb))(popc (i 816))>)

  (cb 22 0.000000 [...])
   ...

  (cb 30 0.000000
      [(flow -4 10 0.000000)
       (flow -3 9 0.000000)
       (flow -2 8 0.000000)
       (flow -1 7 0.000000)])
    (op 30 jump_rg [] [()(mac $addr void)()
                       ()(mac $ecx i)(i 4)(l_g_abs _section_text_90792+16)]
                   <(mem_size (i 4))(mem_read)
                    (gen_opc (l_g_abs jmp))(popc (i 1088))>)
 ...
(end _memmove_453_)
 ...
(align 4 _section_text_90792)
(reserve 16)
(wi (add (l _section_text_90792)(i 0)) (l cb10_memmove_453_))
(wi (add (l _section_text_90792)(i 4)) (l cb9_memmove_453_))
(wi (add (l _section_text_90792)(i 8)) (l cb8_memmove_453_))
(wi (add (l _section_text_90792)(i 12)) (l cb7_memmove_453_))
```

Figure 3.4 Mcode representation of control flow and data elements

statement in Mcode. Each function is then broken up into control blocks, represented by the `cb` grouping. Each control block contains one or more operations, each indicated by an `op` tag, based on certain control flow rules. A control block is made up of a group of instructions that can be entered only at the top. Put another way, no control-flow-altering instruction can branch to an instruction in the interior of a control block. A control block can have multiple points from which to branch out of the block. However, the x86toM program typically creates *basic blocks*, which are control blocks with no more than one control flow transfer instruction at the end of the block. If the block has no control flow transfer instruction, or if the branch is conditional, then the block has a fall-through path to the next consecutive control block. In the example, control block 21 has a *conditional branch* to control block 30 as well as a fall-through path to control block 22.

Figure 3.4 also illustrates a number of features related to the representation of control flow within a function. The two possible paths out of control block 21 are represented by a set of *flow arcs*, which are the elements of the `cb` entry designated by the `flow` tag [13]. Flow arcs are present in all nonterminal control blocks, representing the potential flow of control to other control blocks from the given block. The first integer in each flow arc represents the condition code, which in the general case is 0 for the fall-through path or 1 for a taken path. The second integer represents the target control block, and the last floating-point value is reserved for a profile weight if it is known. For control block 21, note the correlation between its control flow structure as described above and the flow arcs.

In the above example, operation 30 represents an *indirect jump* (`jump_rg`) instruction, which in this case is the control transfer point for a jump table. The associated flow arcs for control block 30 indicate the possible targets of the jump table. For jump table flow arcs, the condition code is used to represent the index into the table upon which each control flow path is based.

This example also illustrates the representation of data elements in Mcode. The `align` statement initiates the block of data by specifying that the data should begin on an address that is a multiple of 4, and labels the block as `section_text_90792`. The `reserve` statement then specifies that the block has a size of 16 bytes. Each 4-byte integer data element (`wi`) contains an address expression, which specifies the position of the data element relative to the beginning of the block, followed by a data value.

In this case, the data values are the labels of the target control blocks for the jump table used by operation 30. That indirect jump refers to the displacement label `_section_text_90792+16`, which is just past the end of the jump table. This interesting situation can be understood by considering that the jump table is being accessed with negative index values, as seen in the condition code values of the flow arcs for control block 30. The fact that the index values are consecutive integers agrees with the observation that the index register, `ecx`, is scaled by a factor of 4.

3.3.2. Functions with multiple entry points

One assumption made by IMPACT compiler tools is that each function is represented as having a single entry point. However, real functions found in binaries may have several entry points. The representation of multiple entry point functions in Mcode, which is designed so that they appear to IMPACT as having only a single entry point, was developed in conjunction with IMPACT's region-based compilation [14]. In order to represent this situation correctly, a special *prologue* control block is added to the beginning of the function. This control block contains a single indirect jump instruction that branches to each of the program's entry points, and this block is also marked with a `prologue` attribute. Those branch targets are indicated by flow arcs out of the prologue control block to each of the entry point control blocks. This scheme preserves proper data and control flow, which will be discussed further in Section 4.3.4. Furthermore, since each entry point control block is tagged with an attribute that contains that entry point's original name, the individual entry points can be recreated during code generation (see Section 5.1).

Figure 3.5 shows an example of a multiple entry point function taken from the library code of the SPECint95 Benchmark *130.li* compiled by Microsoft's Visual C++. The function consists of two entry points: `__startTwoArgErrorHandling_267_` at control block 1 and `__startOneArgErrorHandling_267_` at control block 2. In this example, both entry points flow into the same block, control block 5. The conglomerate function name was derived from one of the entry points and appended with the string "`_me`" to indicate that multiple entry points were present. The prologue control block 7 was added with the corresponding prologue jump, and the flow arcs were created to the appropriate entry point control blocks. Note that control block numeric identifiers do not indicate layout order.

```
 (function __startOneArgErrorHandling_267_me 0.000000
    <(jump_tbls (i 0)(i -1)(s_l_abs "renamed"))>)

  (cb 7 0.000000 [(flow 1 1 0.000000)(flow 1 2 0.000000)] <(prologue)>)
    (op 34 jump_rg [] [()(mac $esp i)]
                   <(prologue)(gen_opc (l_g_abs jmp))(popc (i 1088))>)

  (cb 1 0.000000 [(flow 1 5 0.000000)]
      <(entrypt (l_g_abs __startTwoArgErrorHandling_267_))>)
    (op non-control-flow op)
     ...
    (op non-control-flow op)
    (op 9 jump [] [()(cb 5)] <(gen_opc (l_g_abs jmp))(popc (i 1088))>)

  (cb 2 0.000000 [(flow 0 5 0.000000)]
      <(entrypt (l_g_abs __startOneArgErrorHandling_267_))>)
    (op non-control-flow op)
     ...
    (op non-control-flow op)

  (cb 5 0.000000 [(flow 1 4 0.000000)(flow 0 3 0.000000)])
    (op non-control-flow op)
     ...
    (op non-control-flow op)
    (op 32 rts [] [] <(gen_opc (l_g_abs ret))(popc (i 1792))>)

(end __startOneArgErrorHandling_267_me)
```

Figure 3.5 Mcode representation of a function with multiple entry points

4. LBX86 PHASE 2

The important process of code reoptimization takes place entirely within what is known as phase 2 of the Lbx86 program. The functions of the program being reoptimized are processed one at a time, always represented by Mcode data structures. Many aspects of the Mcode format, as well as certain preprocessing steps taken in phase 2, are related to the need to analyze the specific types of instructions in the program, as well as the data and control flow of the program code. This analysis capability is necessary so that the code can be modified without changing its functional behavior.

4.1. Internal Representation of Attributes

While the ability to add any number of attributes to a Mcode operation adds flexibility to the Mcode format, the continued use of attributes is not efficient in terms of run time and storage. Attributes are contained in a linked list, and each attribute is identified only by its name. Every time the need arises to use the data associated with a given attribute, or even to just detect the presence of a certain attribute, the list of attributes must be traversed while performing a series of string match operations.

Because the information supplied by attributes is necessary to completely represent instructions, and because it is accessed frequently, it is desirable to make the storage of such information take as little space as possible and be quickly accessible. This task is made easier because the set of attributes that must be recognized at Lbx86 run time is known for a given implementation. Therefore, a small, fixed amount of space can be set aside for each operation to store attribute information. The few attributes that may also be associated with functions or control blocks can be represented internally by using the existing flag storage space already associated with each function and control block.

When the Mcode representation is read in by Lbx86, a preprocessing step converts the attributes into the internal binary representation. The unnecessary attributes are then removed to save storage space. If phase 2 is run alone, the attributes are regenerated before writing the optimized Mcode output.

4.2. Instruction Type Identification

Throughout the Lbx86 program, the type of each instruction must be recognized for different purposes and at various levels. Table 4.1 summarizes the different ways that the type of an instruction is identified, including examples of each for the memory-immediate form of the *add with carry* instruction. The first two identifiers in this table are those traditionally used by IMPACT and its code generators. However, only in the case of the functional opcode are the individual values recognized by most IMPACT tools, since the values are machine-independent. Though a specific functional opcode value may not exist for each type of operation that a given machine can perform, it is set to correspond as closely as possible to the functionality of the machine instruction being represented. The use of all of the other identifiers is specific to Lbx86 and will be described further in this section.

One important feature of the Mcode data structure is the numeric processor-specific opcode, known as the *proc_opc* and represented by the `popc` Mcode attribute, which is associated with each instruction. This is the second identifier given in Table 4.1. Though this field may be used in any desired way for a given Mcode format, it is also used by the general

Table 4.1 Identifiers for the type of an instruction

Name	**Storage Location**	**Example**	**Explanation**
functional opcode	opc field of Mcode data structure for each operation	Lop_ADD (110_{dec}, seen in external representation as "`add`")	Machine-independent functional identification; recognized by IMPACT tools
proc_opc	proc_opc field of Mcode data structure; `popc` Mcode attribute	P_ADC_MEM_IMM (148_{dec})	Processor-specific numeric value uniquely identifying instruction type and form; recognized by Lbx86
genopc	component of proc_opc	ADC (9)	Numeric value identifying processor-specific instruction type
variant	component of proc_opc	VARI_MEM_IMM (4)	Numeric value further identifying form of instruction
`gen_opc`	`gen_opc` Mcode attribute	"`adc`"	Assembly language mnemonic text string corresponding to genopc; for readability only
binary machine opcode	`opcode` field of machine instruction encoding specification	80_{hex}	Opcode value used in binary machine encoding for instruction; recognized by processor itself

IMPACT scheduling system [15], [16]. The preferred method for its use relative to the scheduler is that unique values be assigned to every type of instruction that will ever need to be distinguished in any way by the scheduler. For instance, the x86 `add` instruction uses the same x86 binary machine opcode (the sixth entry in the table) for its register-register and register-memory forms. However, these two variations may need to be treated differently by the scheduler. It may be convenient for other types of optimizations to quickly identify addition instructions in the register-register form. On the other hand, some optimization algorithms may wish to identify all addition instructions, regardless of their operand types.

These different motives led to the separation of the proc_opc field into two numeric portions in the proposed system. The two portions are known as the *general opcode*, or *genopc*, and the *variant*, which are the third and fourth identifiers in the table. The genopc corresponds to the processor-specific opcode mnemonic, such as `adc`, and thus is directly correlated to the gen_opc text string attribute in the Mcode format (the fifth entry in Table 4.1). The variant provides more information about the specific form of the instruction; it typically indicates the operand types on which the instruction operates. The encoding of the proc_opc is accomplished by separating the integer field into two groups of bits. This separation is illustrated in Figure 4.1, assuming an integer size of 32 bits. In this way, a value can be extracted for the genopc, a separate value can be extracted for the variant, or the entire field can be considered as a single integer that uniquely identifies both the instruction type and its form. This last use is appropriate for the scheduler.

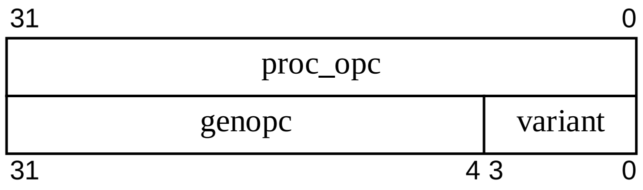


Figure 4.1 Encoding of the proc_opc field

To isolate the complexity of the proc_opc encoding, the Mcode produced by x86toM includes only the genopc portion of the proc_opc; the variant is always 0 in this original Mcode. Within Lbx86, it is important that the proper variant values be added to all operations. This is accomplished by using a table of pointers to *variant annotation* functions. The genopc value for a given instruction is used to index into the table, thereby selecting an appropriate variant annotation function. That function deduces the appropriate variant for the operation based primarily on its operands. Since many instructions have the same set of operand forms in which they can appear, extensive reuse of variant annotation functions is possible.

The process of variant annotation is performed as a preprocessing step in Lbx86, just after the Mcode is read. It is important that the information be kept accurate throughout

Lbx86 phase 2 in the presence of code transformations. Because complete proc_opc values (including variants) are crucial to Lbx86 phase 3, the information must also be correct at the end of phase 2. If a transformation adds a new operation to the code, the appropriate variant annotation function must be called. Similarly, when the operands used by an operation are changed, the variant should be annotated to reflect the change.

4.3. Dependence Analysis

4.3.1. The scheduler manager

The IMPACT compiler system includes a *scheduler manager*, called SM, which facilitates code analysis for transformation and scheduling purposes [17]. It provides easily accessible information about the use of operands in the form of dependence arcs. For instance, a transformation can use data structures provided by SM to follow from a definition of a register to all of its associated uses. It also integrates the scheduler so that the impact of a transformation on the code schedule can be immediately evaluated. SM operates on a single control block at a time, but it also uses IMPACT's *data flow analysis* component to provide information about which operands are live into and out of the current control block [18].

4.3.2. Sources and destinations

Dependence analysis by SM and data flow analysis is based on the appearance of operands as sources and destinations in the Mcode representation. Therefore, in order to prevent incorrect code transformations by the scheduler and other optimizations, it is important that all reads and writes of operands be modeled by the Mcode format. Consider the example in Figure 4.2, which relates to the x86 condition code flags. The *jump if below* (`jb`) instruction depends on the condition code flags set by the *compare* (cmp) instruction, which in turn depends on the value of ecx set by the sub instruction. The mov and add instructions are unrelated to the sub, cmp, and jb. Furthermore, those condition code flags that are set as side effects of the add and sub instructions are insignificant to the operation of this code sequence.

```
mov  eax, 1    ; does not write condition code flags
add  ebx, 4    ; writes condition code flags as side effect
sub  ecx, 19   ; writes condition code flags as side effect
cmp  ecx, edx  ; writes condition code flags
jb   $L5$2     ; reads condition code flags
```

Figure 4.2 Assembly language example illustrating importance of side effects

Transformations performed on this code sequence must not place the add between the cmp and the jb, as this would result in the wrong condition code flow dependence. On the other hand, transformations *are* free to place the mov between the cmp and the jb, since the mov does not affect the condition code flags. Cases such as this are handled correctly in the proposed system by explicitly modeling the condition code flags as registers and including the reads and writes in the Mcode operations, as mentioned in Section 3.2.1. In this way, dependence arcs will be drawn for all relevant sources and destinations.

In the example, an output dependence would exist (for the condition code flags) from the add to the cmp, correctly preventing the add from moving just below the cmp. However, a similar output dependence for the condition code flags would prevent the reordering of the add and sub instructions relative to one another. Since the flags written by these two instructions are never used, it should actually be legal to swap them in this example. Special analysis is employed by SM to prevent such unnecessary output dependences from being drawn, thereby allowing more transformation freedom. This is especially useful in x86 code; most x86 arithmetic and logical instructions also write the condition code flags, but the flag values so written are rarely used. Of course, if they are used, flow dependences will still be drawn correctly.

4.3.3. Overlapping registers

In the typical case, SM indicates dependence arcs between accesses to the same exact register. However, it may be the case that two or more registers overlap in some way, and dependence analysis must take this into account to provide accurate information. For example, the x86 general register `eax` can be broken into a number of subset registers, as shown in Figure 4.3. To zero-extend a 16-bit word, a sequence of operations may write a 0 value to `eax` and then write the desired value to `ax`. In such a case, *both* of these definitions must reach any subsequent uses of `eax`.

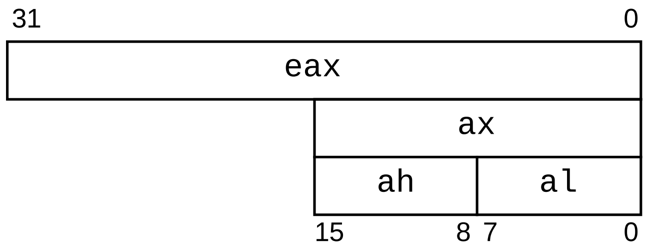


Figure 4.3 Example of overlapping subset registers

Similar issues of overlapping registers exist for many of the macro operands used in the proposed system. The complete condition code flags macro operand, `oszapc_flag`, overlaps with each of its individual components, such as `c_flag` (see Section 3.2.1). To maintain Mcode compactness for the x86 `pusha` and `popa` instructions, which operate on all eight of the x86 general registers, a macro operand called `all_gp_32` is created and overlaps all eight registers (as well as their subsets). The creation of implementation-dependent macro operands, as well as the specification of how they overlap with one another, is part of IMPACT's machine-specific (Mspec) code base [11].

4.3.4. Control flow information

In addition to analyzing the data dependence relationships between instructions in the same control block, data flow analysis is used to understand data dependence relationships between different control blocks. This analysis must take into account the control flow of the program. Therefore, accurate control flow information is necessary in the Mcode representation, and is provided by x86toM in the form of flow arcs. In order to correlate flow arcs with the associated control-transfer instructions, other IMPACT components expect appropriate use of the Mcode functional opcode. A conditional branch instruction, such as *jump if not parity* (`jnp`), needs to use a conditional branch functional opcode. Since general IMPACT tools do not recognize a specific conditional branch based on parity, it is safe to use the *branch if not equal* (`bne`) functional opcode. For similar reasons, the *subroutine call* (`jsr`) functional opcode must also be used appropriately. These are important examples of the use of the functional opcode for more than just readability.

Correct analysis of the program's control flow structure is also an important motivation for the proposed system's representation of multiple entry point functions as discussed in Section 3.3. To clarify the control flow structure relative to the various entry points, a temporary *indirect jump* is present at the beginning of the composite function (see Figure 3.5). The corresponding flow arcs created are recognized by data flow analysis to indicate that the flow of control in the function can go directly from the beginning of the function to the entry point control blocks. In this way, data flow analysis will provide correct results for multiple entry point functions. For more information on call site references to such functions, see Section 5.1.

Functions with multiple exit points, which have been observed to be very common in x86 code, also require special handling for proper data flow analysis. If a function has more than one *return from subroutine* (`rts`) operation, x86toM performs a simple transformation. Each `rts` is changed into an *unconditional jump* (`jmp`) operation that jumps to a new *epilogue* control block. That final epilogue block contains the single `rts` expected by data flow analysis. Like the multiple entry point representation, this transformation will be undone before the new object file is written out in Lbx86 phase 3.

4.3.5. Live-out information and dummy jumps

When SM integrates data flow analysis results into its dependence information, it draws dependence arcs to branches for operands that are live-out along that branch's taken path. This is very convenient for code transformations, because it is not necessary for the associated analysis to also make a separate check of live-out data flow information. However, because no branch instruction is associated with the fall-through path out of a control block, SM has no instruction to which to map the live-out information for the fall-through path. In order to make this convenient information available in all cases, Lbx86 phase 2 performs another preprocessing step. An unconditional *dummy jump* is created at the end of each control block that has a fall-through; the jump target is set to the next block. In this way, complete live-out information is made available to all transformations through the SM dependence arcs. This temporary transformation is undone in a phase 2 postprocessing step.

4.3.6. Limitations

The ability to alter the relative ordering of instructions that access memory requires dependence analysis of memory accesses, or *memory disambiguation*. Conventional disambiguation in the IMPACT compiler utilizes information derived from prior compilation steps, which have acted on high-level source code. Because the proposed system does not have the benefit of starting from source code, information is not readily available to enable conventional disambiguation of references to global variables, stack-allocated local variables, and register allocator spill locations. Though the Mcode format used in the proposed system includes the necessary information to allow a modified, low-level form of memory disambiguation, the analysis is not currently implemented. In the present, conservative implementation, memory is considered a single cell so that every store to memory potentially conflicts with any other memory access.

Standard x86 floating-point instructions operate based on a stack model. The eight registers in the floating-point stack are identified based on their offset from the top of the stack, which changes as computations are performed. This storage model invalidates a traditional register-based dependence analysis. Although a more advanced analysis may be possible, floating-point register accesses are currently treated conservatively for dependence analysis purposes.

The use of exception-handling features, the interface to which is specific to the operating system, significantly complicates the control flow structure of a program. When exceptions are nonrecoverable, which is the default behavior for programs, the extra control flow is not particularly important. However, if context-sensitive, recoverable exception-handling features are added to a program, the control and data flow analysis may be invalid. The x86toM program adds a special attribute to any function in which the installation of exception handlers is detected. Lbx86 phase 2 does not perform optimizations that rely on dependence analysis information on any function that is marked with this attribute.

4.4. Optimization

By providing complete and accurate data and control flow information, in addition to flexible methods for identifying instruction types, Lbx86 phase 2 creates a flexible framework in which any number of transformations and optimizations are possible. These capabilities are described in the remainder of this section. See Chapter 7 for details on specific test optimizations evaluated within the framework.

4.4.1. Rescheduling

One special case of optimization is rescheduling, which is performed by SM based on a machine description [19], [20]. Different machine descriptions can be created for different microarchitectural implementations of the same instruction set, enabling high-quality machine-specific rescheduling.

The development of a machine description typically involves many references to the various instructions of the target machine. As mentioned earlier, the different possible values for the proc_opc field represent all the different instructions (and forms thereof) that are distinguishable in the machine description. It is clear that, for development and maintenance purposes, it is desirable to refer to these proc_opc values by symbolic names. It is therefore necessary to maintain a mapping from these names to their numeric values. However, even maintaining such a mapping file may be difficult when development is ongoing. As shown in Figure 4.4, a script called lbx86_gendefs.pl has been developed to automate this task. This script analyzes Lbx86 source files in order to produce the *defs* file. The resulting file contains C-style symbolic constant definitions for each utilized proc_opc value. The same numeric variant value is reused to represent different variants in different contexts, and most

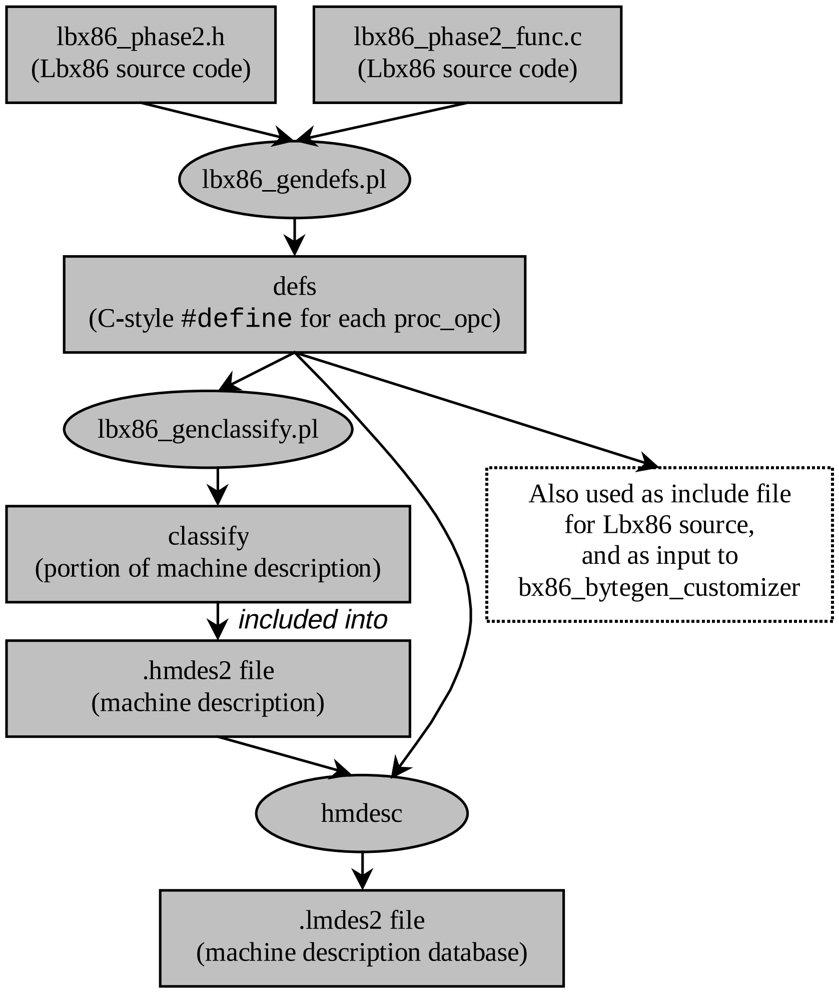


Figure 4.4 Support infrastructure for development of a machine description

instructions do not use the full range of possible variants. For this reason, the Lbx86 source code for the variant annotation functions are analyzed to determine what variants are actually associated with each general instruction type. While the defs file is important to the machine description, it is also used as an include file for the Lbx86 code base to allow references to specific proc_opc symbolic constants in the development of Lbx86 code transformations. Its further use for machine instruction encoding will be described in Section 5.2.2.

A significant section of the machine description involves the classification of each proc_opc value according to its characteristics. Some characteristics to be considered include memory access behavior and control flow behavior. All instruction forms having the same set of characteristics are grouped together. To ease the burden of maintaining this section of the machine description during ongoing development, the lbx86_genclassify.pl script has been developed. It produces the *classify* file, which is then included into the machine description file. The script uses clues provided by the variant of a proc_opc value to help automate the classification. For instance, proc_opc values with the VARI_REG_MEM variant typically involve a load from memory. While some instructions require special handling in the script, the general cases are handled automatically. Note that both the defs and classify files need to be regenerated only when changes are made to the usage of proc_opc values in an implementation of the framework.

To achieve performance-enhancing rescheduling for a specific target processor, a specialized machine description must be created. The same defs and classify files can be used in conjunction with any number of processor-specific machine descriptions. Such a machine description must model the microarchitecture of the target machine, including machine resources and their interactions. The use of certain machine resources by the various instructions must also be represented. Once a complete specification is created, the hmdesc program is used to convert the textual specification to the low-level machine description database. This database is read by Lbx86 phase 2 and used by SM to guide scheduling.

4.4.2. Other transformations enabled

The framework of Lbx86 phase 2 enables any manner of code transformation, and the usefulness of this capability is not limited to optimization. Specific probe code could be inserted to gather information about dynamic program behavior. The platform for research thus enabled may utilize machine-specific performance monitoring hardware [21]. It would also be possible to feed this profiling information back into the reoptimization system, allowing profile-guided optimization. Such optimizations might include the reordering of control blocks and functions, which is supported by the proposed system.

To support the dynamic collection of program statistics, the reoptimization system supports the addition of new initialized space to the program's data section. Such data space is added to the beginning of the data section using one or more new labels. The intrinsic label of the program's original data space, `_section_data`, then refers to a location that is offset from the beginning of the new data section. Since the original data space remains contiguous, original references to both initialized and uninitialized data are kept accurate.

5. LBX86 PHASE 3

The primary function of phase 3 of the Lbx86 program is to generate a binary machine code representation of the optimized Mcode produced by Lbx86 phase 2. Since only certain sections are converted, as opposed to a complete program, the output is produced in the form of a Common Object File Format (COFF) object file [7]. A corresponding assembly language file is also produced, though this is intended primarily for debugging purposes. During this phase, the fixup file previously produced by x86toM is processed in order to maintain symbol compatibility with the new object file.

5.1. Mcode Format Preprocessing

To simplify generation of binary machine code by Lbx86 phase 3, it is convenient for the operations in Mcode to have an exact one-to-one correspondence to x86 machine instructions. However, some extra operations exist in the Mcode that were used to enable correct and convenient dependence analysis during phase 2. Before generating machine code, phase 3 must remove these extraneous operations.

In the case of functions with multiple exit points, the epilogue transformation described in Section 4.3.4 must be reversed. Unconditional jumps to the epilogue control block are converted back into the correct return from subroutine operation. Then the epilogue control block itself is removed.

For functions that have multiple entry points, the prologue control block and its associated flow arcs must be removed. However, the attributes that identify the names of the various entry points are left in place. This allows these entry point names to be made into function labels in the object file. In every function in the program code, subroutine calls that referenced multiple entry point functions must also be adjusted. The form of these calls before and after adjustment is shown in Figure 5.1. Before Lbx86 phase 3, such calls use the name of the conglomerate multiple entry point function as their target operand. This ensures the proper Mcode function being called can be found by other IMPACT tools. For the purposes of phase 3 object file generation, however, the target operand should be the label of the actual entry point being called, which is derived from the `entrypt` attribute attached to the call operation. Note that, because of its large size and relatively infrequent presence and use, this attribute is not converted to a faster-access version with the others as described in Section 4.1.

before Lbx86 phase 3:

```
(op 33 jsr [] [()(l_g_abs _$fn__startOneArgErrorHandling_267_me)]
           <(entrypt (l_g_abs _$fn__startTwoArgErrorHandling_267_))
            (gen_opc (l_g_abs call))(popc (i 336))>)
```

after Lbx86 phase 3 Mcode format preprocessing:

```
(op 33 jsr [] [()(l_g_abs _$fn__startTwoArgErrorHandling_267_)]
           <(gen_opc (l_g_abs call))(popc (i 336))>)
```

Figure 5.1 Mcode representation of a subroutine call to a function with multiple entry points

5.2. Specification of Machine Instruction Encoding

The encoding of binary machine instructions is complicated by the aim of packing such instruction encodings into as few bytes as possible. Every major instruction set architecture has its own binary encoding format with certain idiosyncrasies. In the x86 architecture, one example is the existence of special "short" forms of instructions that can be used in conjunction with certain operands. It is therefore desirable to create an easily maintainable database to specify how to encode instructions, rather than writing many different program functions to encode all the various types of instructions. Ideally, the specification format of such a database should be easy to verify against an appropriate architecture document.

5.2.1. The database specification

In the proposed system, a database is created using the IMPACT meta-description (MD) language facility [20]. In the specification, one entry exists for each possible value of the proc_opc. Each entry uses data field names that correspond to the bit-encoding field names used in the Intel's *Instruction Set Reference* [4]. In this way, the database specification is clear and maintainable.

Figure 5.2 shows sample encoding specification entries for two different forms of the *logical exclusive-or* (`xor`) instruction. In general, each entry begins with information about the instruction's operands within the Mcode format for the operation. Each `operand`*`n`* field includes items to specify the Mcode operand position(s) in which the operand can be found and the expected type of the operand. Each operand is thereby given a number to identify it independent of the Mcode source or destination operand number. In the examples, operand 2 comes from Mcode `src[1]`, while operand 3 is found in Mcode `dest[3]`.

The remaining fields in each entry specify the binary format of the machine instruction. The `opcode` field specifies the base binary machine opcode, while the `s` and `w` fields specify bit positions of that opcode that may also be set depending on the size of various operands. Note that the designations of these two bit fields are the same as those used in [4].

Consider the `modxrm` field, which consists of items to specify how to encode the three separate bit fields of the x86 ModR/M byte. While the first and last bit fields (Mod and R/M) are typically used as part of the specification for a memory address, the middle field is used for different purposes in different contexts. In the first example, the middle bit field uses X_REG_OP1 to indicate that the middle three bits of the ModR/M byte are based on the register specified by operand 1, which in turn can be found at both `src[0]` and `dest[0]` in the Mcode format. In the second example, the middle bit field is simply a constant value of 6, which serves to distinguish the `xor` instruction from the other arithmetic and logic instructions that also use the same base binary machine opcode of 80_{hex}.

```
P_XOR_REG_MEM        (operand1(${SRC0_AND_DEST0}  ${TYPE_GENREG})
                      operand2(${SRC1}            ${TYPE_ADDR})
                      operand3(${DEST3}           ${TYPE_MAC_VOID})
                      opsize(${SIZE_FOLLOW_MEM})
                      opcode(0x32)        w(0)
                      modxrm(${MOD_NOT11} ${X_REG_OP1} ${RM_MEM}));

P_XOR_MEM_IMM        (operand1(${SRC0}            ${TYPE_ADDR})
                      operand2(${SRC1}            ${TYPE_INT_OR_LABEL})
                      operand3(${DEST3}           ${TYPE_MAC_VOID})
                      opsize(${SIZE_FOLLOW_MEM})
                      opcode(0x80)   s(1) w(0)
                      modxrm(${MOD_NOT11} 6 ${RM_MEM})
                      immed(2 ${SIZE_FOLLOW_MEM}));
```

Figure 5.2 Sample machine instruction encoding specification entries

Note also the `immed` field in the second example, whose first item specifies that the value of the immediate operand is based on operand 2, which in turn is found in `src[1]` of the Mcode operation. This extra level of indirection for fields that refer to specific operands allows the Mcode format to be changed more easily. For more detailed information on the various fields of the encoding database, refer to Appendix C.

5.2.2. Development and utilization

Just as the machine description specification required the use of the hmdesc program to convert it into a low-level database form (see Section 4.4.1), the machine instruction encoding specification must be converted from its high-level textual form to a low-level database form. As illustrated in Figure 5.3, three programs accomplish this task. While the first two are standard IMPACT tools, the bx86_bytegen_customizer program was developed specifically for use with the encoding database. The defs file is used by this program to associate the correct value with each proc_opc symbolic constant, which is analogous to the use of the defs file for machine description development.

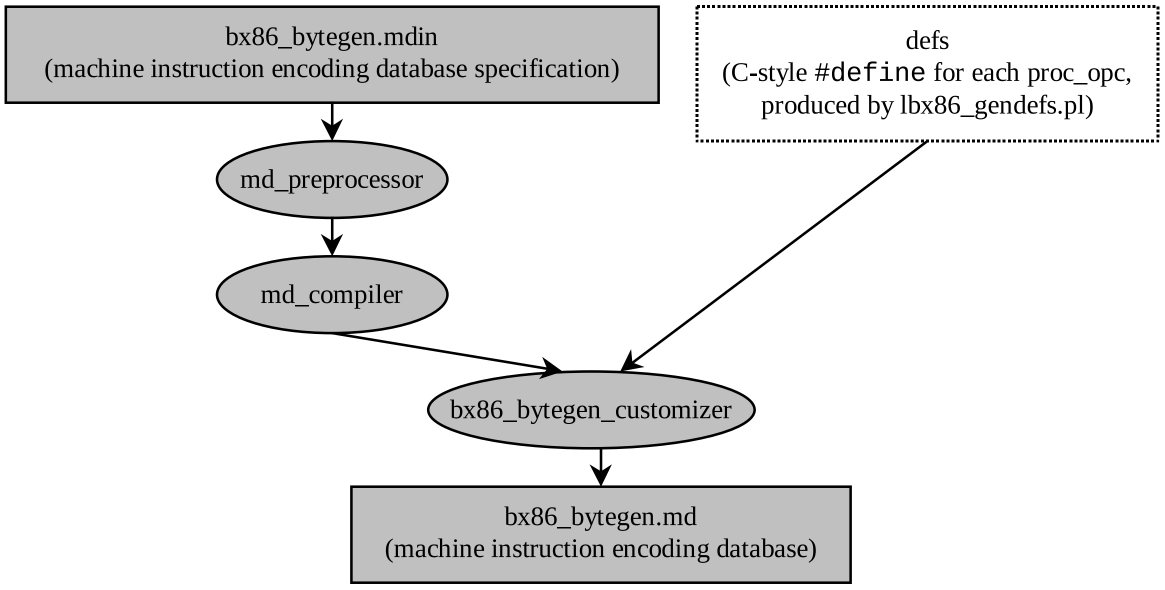

bx86_bytegen.mdin
(machine instruction encoding database specification)
defs
(C-style #define for each proc_opc, produced by lbx86_gendefs.pl)
md_preprocessor
md_compiler
bx86_bytegen_customizer
bx86_bytegen.md
(machine instruction encoding database)

Figure 5.3 Support infrastructure for development of a machine instruction encoding database

The low-level machine instruction encoding database produced by bx86_bytegen_customizer is read by Lbx86 phase 3, which includes functionality to interpret the database. Two sets of phase 3 functions utilize information from the database. The primary set of functions has the complete functionality necessary to generate binary encoded machine instructions. The secondary set of functions is made up of faster, reduced versions that only determine the number of bytes required to encode a given instruction. Though not yet implemented, a third set of functions is envisioned for use during ongoing development of the binary reoptimization framework. Using the fields of the database that specify elements of the Mcode format of each instruction, these functions would provide a consistency check with x86toM. The functions would also be useful for verifying the validity of operations added or modified by Lbx86 phase 2 code transformations.

5.3. Address Resolution

When machine code is generated, most references to addresses are handled by creating COFF relocation entries, which refer to the COFF symbol table and are resolved at link time. However, some instructions require addresses to be specified relative to the current program counter. Such *relative addresses* are always within the code section, so they can be resolved during the process of machine code generation. To calculate the appropriate values for relative address references, the normalized starting address of every function entry point and control block in the program must be known.

To calculate the necessary starting addresses, a preprocessing pass is made over the program. During this pass, the machine instruction encoding database is used to calculate the length of each instruction. The amount of space necessary between each function, both for jump tables and for the alignment of functions, is also calculated.

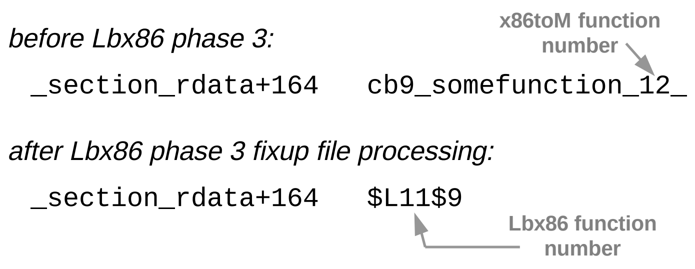
The x86 architecture allows short forms of many instructions that use relative addresses. For instance, without any further analysis a conditional branch would require four bytes for its relative address. If it is known that the relative address falls in the range of -127_{dec} to $+128_{dec}$, the short 1-byte relative address form can be used. Because of the frequency of short jumps and branches, and the related decrease in code size that can be achieved, Lbx86 phase 3 performs an analysis to optimize the size of relative addresses within a function. First, all relative addresses are assumed to be full size, and then an address resolution pass is executed during which each relative address that can be made into a short form is made so. Since the associated decrease in instruction size may cause an address that was previously out of range to come into short form range, multiple passes are made. This continues until no more changes occur within the function. The function's size and address is then known, and address resolution continues with the next function. Calculation of relative addresses between different functions is done during code generation. Since the iterative analysis described above would be too costly to perform across the entire program, such interfunction relative address references are always assumed to be full size.

5.4. Code Generation

After all preprocessing steps are completed, a final pass is made over all the Mcode to produce the binary machine code for the program and its data. At the same time, a related set of Lbx86 phase 3 functions is used to produce corresponding text assembly code output. The code and data sections are processed separately, and, after each section is produced, the relocation entries for that section are output to the object file. The COFF symbol and string tables are added to the end of the object file, after which the COFF headers are finalized to indicate the positions of the various COFF components.

5.5. Fixup File Processing

The symbols in the fixup file as produced by x86toM correspond to the style of function and control block labels used in the Mcode representation. Because these labels have a slightly different format in the object file representation that is output by Lbx86 phase 3, it is also necessary to adjust the symbols in the fixup file accordingly. A sample entry before and after processing is shown in Figure 5.4. The first portion of each entry specifies the location in a nonconverted section where a fixup will need to be performed. The second portion is the label for the address to which the fixup will be resolved. In the example, the Mcode-style control block label is converted to the shorter form used in the object and assembly files produced by Lbx86 phase 3. Note that the function number used to build the new control block label may not be the same as the function number appended by x86toM as part of the symbolic function name.



before Lbx86 phase 3:

```
 _section_rdata+164   cb9_somefunction_12_
```

after Lbx86 phase 3 fixup file processing:

```
 _section_rdata+164   $L11$9
```

Figure 5.4 Sample fixup file entry

In addition to adjusting the names of the labels, if the labels referred to in the fixup file were not already *public* (visible beyond the scope of a single section), they are made public. This ensures that these labels will be present in the COFF symbol table, which is necessary in order for the linker (PEwrite) to resolve the fixups.

6. PEWRITE

A typical linker combines one or more object files together with library code and operating-system-specific startup code to produce an executable program. The proposed system uses a single object file containing only the code and data sections. The other sections must be recovered from the original input executable and adjusted according to the fixup file. Furthermore, the object file produced by the reoptimization system already contains all necessary library and system startup code, since this was converted to Mcode along with the user code. For these reasons, a special-purpose linker called PEwrite has been developed and is used as the last processing step in the system.

The PEwrite program reads the original executable program and creates a new one by replacing the original code and data sections with the optimized and converted ones from the COFF object file. This involves special support for changing the sizes of the optimized code and data sections. All sections must maintain alignment in the address space of the program as it will exist when it is actually loaded into memory for execution. The aligned base address of each section will have a bearing on the address references written into the executable. A separate section alignment must also be maintained within the executable file itself.

After all sizes and positions are determined, addresses can be resolved for the various labels. These addresses will be based on the preferred base address of the executable, which remains unchanged. The relocations for the new code and data sections are performed as specified in the object file, thus linking between the code and data sections as well as to the other, non-converted sections. Then the processed fixup file is read, and the necessary linkages are performed in the nonconverted sections. Finally, the headers for the new PE executable are updated, which includes linking to the program entry point also specified in the processed fixup file. The new, optimized executable program is thus formed.

7. PERFORMANCE EVALUATION

7.1. Overview

In order to validate the approach proposed in this thesis, the binary reoptimization framework has been implemented; the major components execute under Windows NT on an x86 processor. The implementation is capable of processing 32-bit Windows x86 applications. A number of example optimizations have been implemented in order to demonstrate the correctness of the system, its optimization capabilities, and its potential to improve performance.

As a case study reflecting the importance of binary reoptimization, the example optimizations target the AMD-K6 microprocessor. This processor is a superscalar implementation of the x86 architecture. While its architectural features are comparable to those of the Intel Pentium or Pentium Pro processors, including MMX support, its microarchitectural features are distinct. The implementation has the capability to decode up to two instructions per cycle and issue six microinstructions per cycle to seven execution units. It also utilizes speculative and out-of-order execution with register renaming and data forwarding [22]. Because of its distinct microarchitecture, the instruction sequences used to achieve optimal performance for the AMD-K6 [23] differ from those used for the Pentium or Pentium Pro processors [24]. However, the widely used Microsoft Visual C++ 5.0 compiler does not have an option to target code generation for the AMD-K6. Hence, little commercial software is optimized for the AMD-K6, making it an appropriate target for binary reoptimization.

The performance of programs reoptimized by this framework was evaluated on a 233 MHz AMD-K6 system with 64 MB of RAM, running Windows NT Workstation 4.0. The programs reoptimized come from the set of SPECint95 benchmarks; the “original” executables were generated by Microsoft Visual C++ 5.0 with maximum speed optimizations, including function inlining, and targeted for a blend of Intel x86 processors. Complete results are presented for seven of the eight SPECint95 benchmarks. The benchmark *126.gcc* was not evaluated because of difficulties in producing a correct original executable using the Microsoft compiler. Performance data was collected on the evaluation machine by running each version of each benchmark three times and taking the average wall time of the three runs. The complete set of SPECint95 reference input was used for each benchmark run.

All results are presented in terms of speedup, the factor of performance improvement over a certain baseline runtime. The runtime of the original executables may be viewed as an appropriate baseline, since this is the performance the binary reoptimization framework is intended to improve upon. However, the decoding process of the framework changes the order of program functions into an approximate depth-first traversal of the program’s call graph, and the resulting cache and paging effects often produce performance improvements. The resulting speedup shown in Figure 7.1 is based on running the complete system to convert and reproduce executables without using any specific optimizations. Because the new executables functioned correctly, the basic functionality of the framework is verified. Since the speedup demonstrated in this case is inherent in the framework, most of the remaining results presented use the runtime of the “conversion only” executables as a baseline; this isolates the impact of the optimizations being considered.

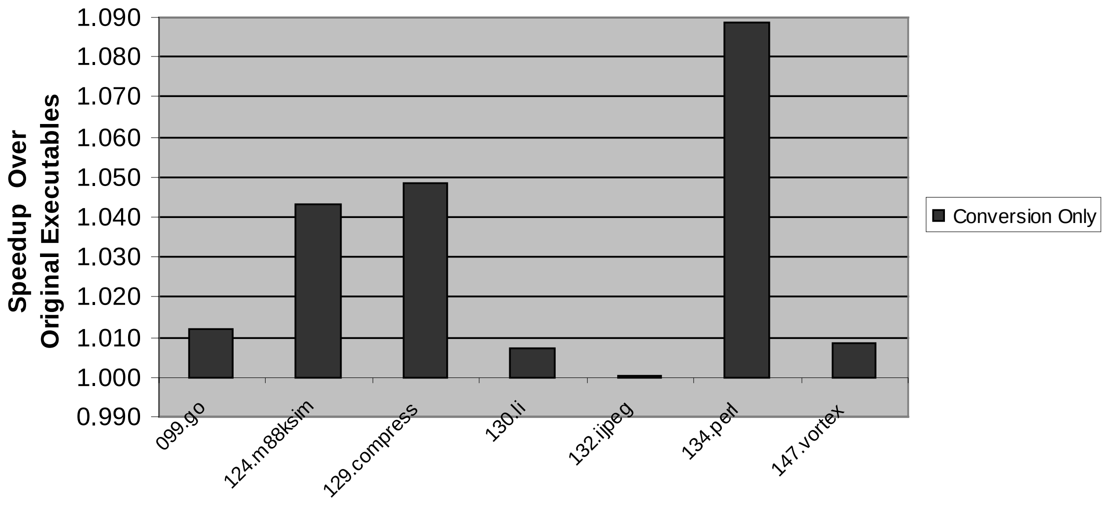


Figure 7.1 Speedup inherent in framework

7.2. Rescheduling

The rescheduling of program instructions is important to optimally exploit the microarchitecture of the target processor. However, the development of a complete machine description to guide scheduling specifically for the AMD-K6 is beyond the scope of this thesis. Instead, a very simple machine description was developed to model a generic x86 machine in which all instructions have the same unit latency. Furthermore, the simple machine modeled is assumed to have two complete sets of uniform functional units, thus allowing any two nondependent instructions to decode and issue simultaneously. This is meant to correspond very roughly to the AMD-K6 maximum decoding bandwidth of two instructions per cycle. Beyond this very simple machine model, the machine description used to produce the following results was simply designed to enforce correctness during scheduling.

The results for this "two-issue rescheduling" optimization are shown in Figure 7.2. While the performance is not optimal for the AMD-K6, this experiment is very important from a validation standpoint. Since the rescheduled executables functioned correctly, it demonstrates the capability of rescheduling to affect performance while maintaining functionality. Because the rescheduling causes significant reordering of program instructions, it also stresses dependence and data flow analysis capabilities, verifying the accuracy of this analysis.

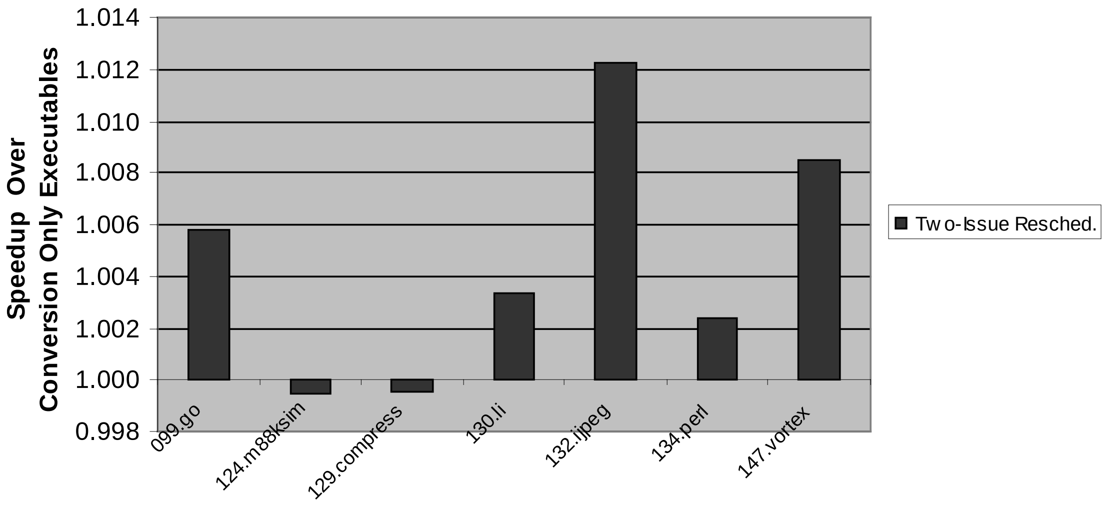


Figure 7.2 Performance results for simple two-issue rescheduling

7.3. An Encoding Optimization

One optimization recommended in [23] simply involves how certain memory addresses are encoded in instructions. Whenever the address for a memory reference involves only the `esi` register, with no scale or displacement, the AMD-K6 will process the associated instruction more efficiently if the address is encoded with a constant displacement of 0. Normally, the displacement byte would be omitted in such cases.

Performing this encoding optimization in the proposed framework requires no data flow or dependence analysis information. At a late stage in Lbx86 phase 2, useless displacement values of 0 are first removed from the Mcode representations of all instructions. Then this optimization algorithm reintroduces a displacement of 0 for any instructions that meet the criteria described above. When instruction encodings are generated by Lbx86 phase 3, this displacement of 0 will be encoded as desired since phase 3 encodes instructions exactly as specified. This property is in contrast to the Microsoft Macro Assembler (MASM), which always assumes that a constant displacement of 0 is not desired. This is one reason why Lbx86 generates binary code directly rather than relying on an assembler.

Figure 7.3 shows the results for the "`[esi+0]`" encoding optimization. While it is often a beneficial optimization, two benchmarks show degraded performance. Since the optimization does increase the size of instruction encodings, these losses may be the result of negative instruction cache effects.

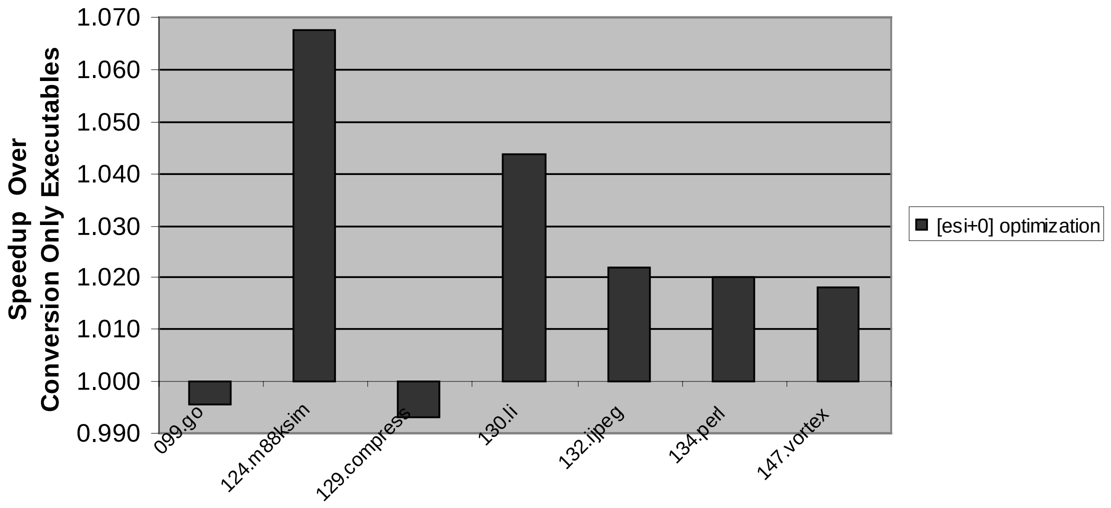


Figure 7.3 Performance results for `[esi+0]` encoding optimization

7.4. An Instruction-Selection Optimization

The x86 instruction set offers two different ways to clear a register to 0. A `mov` instruction can simply move an immediate value of 0 into the register. Alternatively, a `xor` instruction can be used to perform the logical exclusive-or of the register with itself, which has the effect of clearing the register to 0. Though most compilers will favor the `xor` form for its smaller encoding size, [23] suggests that the `mov` form is better for the AMD-K6. The reasoning is based on the observation that, though the prior value of the register doesn't matter, its appearance as a source in the `xor` form causes extra dependence checking and a consequent reduction in issue freedom.

Although performing this optimization is a simple matter of replacing one operation with another, an extra check is necessary to verify that the optimization is safe. This is due to the fact that the `xor` instruction always writes the condition code flags as a side effect, while the `mov` instruction does not. Before converting a `xor` into a `mov`, it is necessary to check that the flags set by the `xor` are not used by any later instruction. This check requires dependence and data flow analysis information, and thus it is performed with the aid of the information provided by SM. The C code for a function to check whether the write to a certain destination is later used is shown in Figure 7.4. This example illustrates the ease with which SM information can help answer important questions in dependence analysis.

The performance impact of the "clear using `mov`" optimization is illustrated in Figure 7.5. While typically beneficial, performance losses are observed for two of the benchmarks; this may again be due to negative cache effects resulting from the larger instruction encoding size of the `mov` instruction.

```
int O_has_uses(SM_Reg_Action * dest_action)
{
  SM_Dep * dep_out;

  /* look for any flow dependences that are not ignored */
  for (dep_out = dest_action->first_dep_out; dep_out != NULL;
       dep_out = dep_out->next_dep_out)
    if ((dep_out->flags & SM_FLOW_DEP) && !(dep_out->ignore))
      return(TRUE);

  return(FALSE);
}
```

Figure 7.4 Function to check for uses of a write to a destination operand

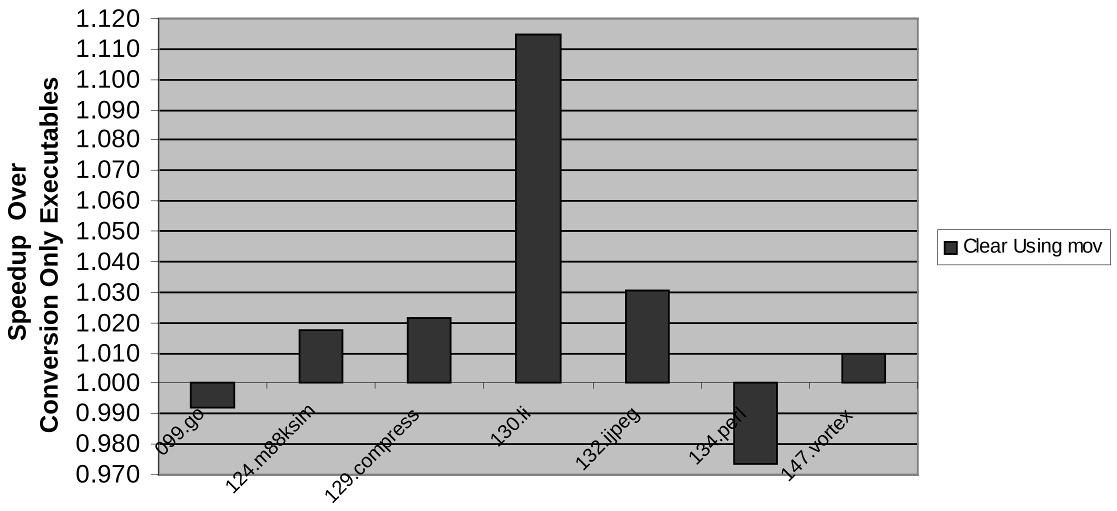


Figure 7.5 Performance results for clear using `mov` instruction-selection optimization

7.5. A More Complex Optimization

In many microarchitectural implementations of the x86 instruction set, *integer multiplication* (`imul`) is a very expensive operation. For this reason, many x86 compilers perform an optimization when it is necessary to perform an integer multiplication by a constant. It is possible to perform such a multiplication by using an equivalent sequence of logical shift, add, and subtract operations, and such a sequence is generated when its latency would be less than that of the corresponding `imul`. However, because the AMD-K6 includes a low-latency integer multiplier, this "optimization" usually decreases performance on this processor. It is therefore usually desirable to reverse this optimization, recreating a single `imul` instruction to replace the chain of instructions (referred to as the *imulchain*). Figure 7.6 shows an example of this optimization in assembly language, with comments to clarify the operation of the original chain of instructions. Note the use of the *load effective address* (`lea`) instruction, which is just an efficient way of performing certain combinations of addition and multiplication by small powers of 2; the latter is the same operation performed by the *arithmetic left shift* (`sal`) instruction.

original chain of instructions:

```
lea  ebx, [ecx+ecx*2]  ; ebx = ecx × 3
sal  ebx, 4            ; ebx = ebx × 16  = ecx × 48
add  ebx, ecx          ; ebx = ebx + ecx = ecx × 49
sal  ebx, 6            ; ebx = ebx × 64  = ecx × 3136
sub  ebx, ecx          ; ebx = ebx – ecx = ecx × 3135
```

after imulchain reverse conversion optimization:

```
imul ebx, ecx, 3135    ; ebx = ecx × 3135
```

Figure 7.6 Assembly language example of integer multiplication by a constant

A number of observations can be drawn from this example. First, the imulchain is made up of a fixed set of instruction types. Second, note that every operation in the chain (except perhaps for the first) uses the same register (`ebx`) as both a source and a destination, and this is the always the destination register of the first instruction in the chain. This register is referred to as the *accumulating register* for the imulchain; `ecx`, on the other hand, is the *base register*.

The optimization algorithm that performs this reverse conversion optimization makes extensive use of the information provided by SM. After an instruction is found that is a candidate for starting an imulchain, dependence information can easily be used to follow the instructions within the chain. The chain terminates when any dependent instruction is found that is of a different type than those expected, or if any one of a number of other conditions is violated. This approach is much easier than a forward serial search of all following instructions because it naturally excludes instructions that do not matter to the chain; such intervening instructions could be present due to instruction scheduling of the original code. An example of an intervening instruction that would be significant would be one that changes the value of the base register. While this instruction would not be inspected as a part of the dependence chain, such a chain-breaking case is detected by using SM information to ensure that every use of the base register by a member of the chain has the same set of reaching definitions as the first use of the base register.

While following the dependence chain, the optimization algorithm keeps track of both the members of the chain and the constant multiplication factor up to that point. Once the end of the imulchain is discovered, the final transformation is only performed if the number of instructions in the chain has reached a user-specified threshold. This ensures that the optimization will only be performed when it is beneficial. If the threshold is reached, the correct `imul` instruction is inserted and the members of the imulchain are deleted.

Figure 7.7 details the performance results of the “constant `imul`” optimization, which are based on a chain length threshold value of 3. It is interesting to note that the optimization is not always beneficial, though it results in fewer instructions with the same or better latency.

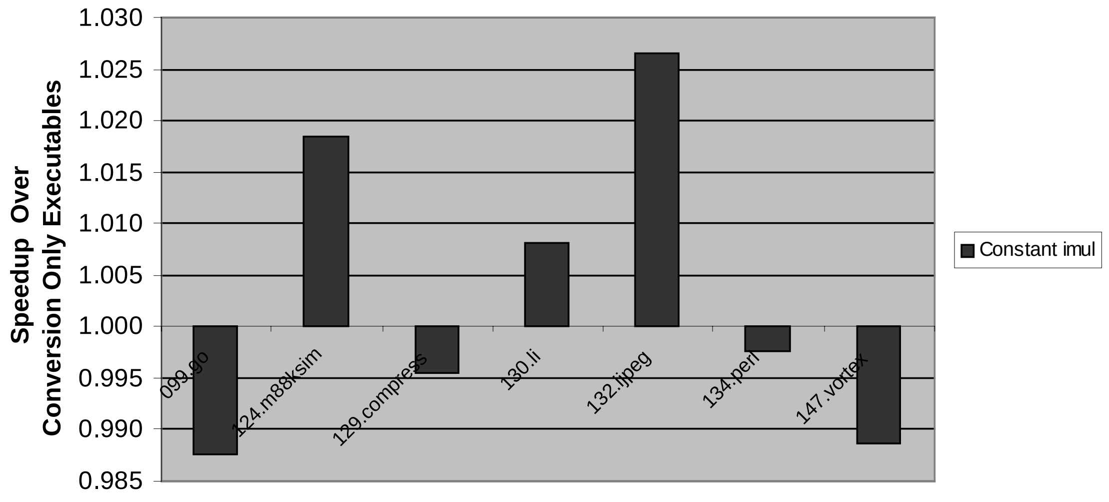
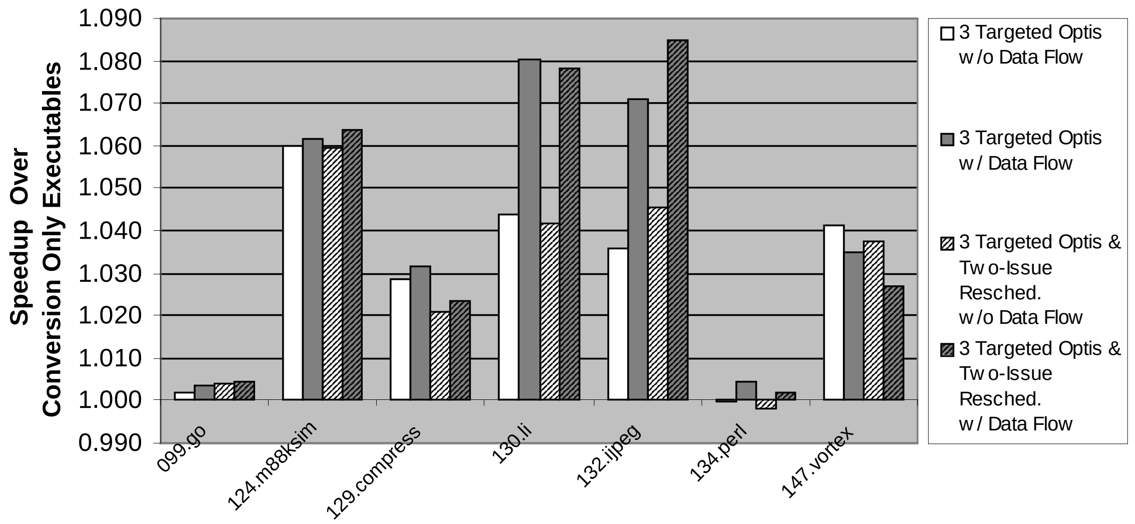


Figure 7.7 Performance results for constant `imul` optimization

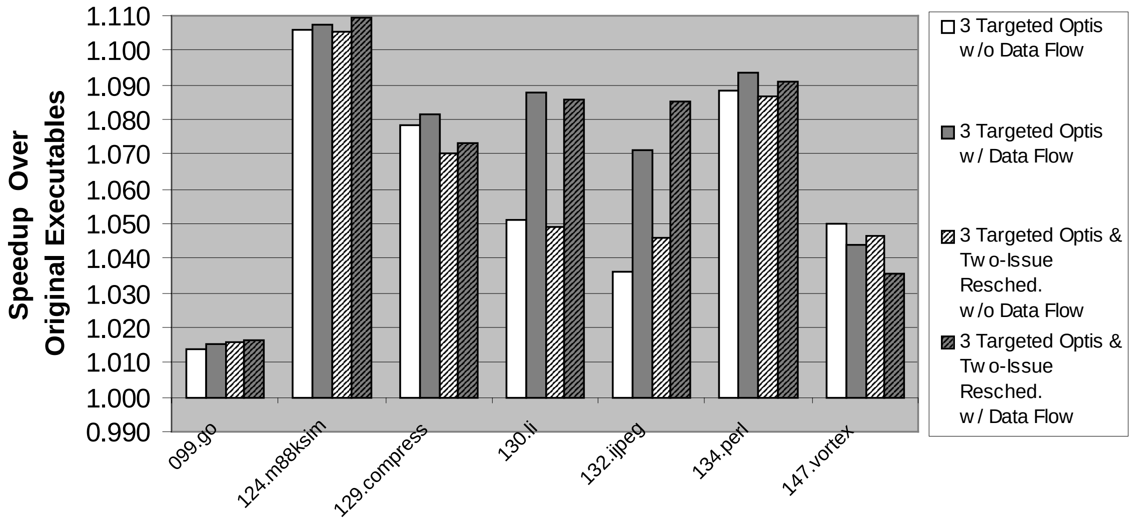
7.6. Summary

Composite performance results for the evaluated optimizations are presented in Figure 7.8. The "3 targeted optis" referred to in this figure are the targeted processor-specific optimizations of Sections 7.3, 7.4, and 7.5. Results for these optimizations applied together are presented, as are the results with the two-issue rescheduling utilized along with the others. To emphasize the benefit of complete, function-level data flow analysis, the results are also broken down into those with and without this complete data flow analysis. Without data flow analysis, conservative assumptions are made, such as assuming that all operands are live out of every control block. With data flow, on the other hand, dependence analysis spans control blocks to provide accurate operand liveness information.

Figure 7.8(a) considers speedup relative to the conversion only executables, as do the figures in this chapter that examine the optimizations individually. To highlight the overall performance benefits of the framework as evaluated, Figure 7.8(b) compares the composite results back to the original executables that were reoptimized.



(a) Compared to conversion only executables



(b) Compared to original executables

Figure 7.8 Composite performance results

8. CONCLUSION

This thesis discusses the backend functionality of the IMPACT binary reoptimization framework. It details the challenges and issues addressed to perform optimization and executable regeneration for the purposes of binary reoptimization. This information is useful as a reference for future extension of capabilities of framework, including the adaptation of the framework to other architectures. The examples of optimizations for a specific platform implementation also offer insight into the analysis and transformation capabilities of the framework, providing a basis for the further development of optimizations and other transformations.

The system proposed by this framework has been implemented for the x86 architecture. Many of the established IMPACT compiler technology tools have been utilized in this framework to ease development and provide a rich set of capabilities. The implementation has been used to successfully transform and reoptimize 32-bit Windows x86 binary executables, thus validating the approach. The performance results generated, while already encouraging, are meant primarily as examples of what is possible.

Because of the powerful code analysis capability enabled within the framework, many different types of optimizations may be explored in the future. For any given target processor, many opportunities exist for machine-specific optimizations. Classical compiler optimizations are also possible, and may be useful if it is observed that existing binaries have not been well optimized in this respect. It would also be useful to develop a highly targeted machine description for a processor such as the AMD-K6. This would allow further study into the possible performance benefits of rescheduling. More advanced analysis capabilities, such as memory disambiguation, may also be explored.

In addition to target-specific optimizations, the analysis capabilities of the framework make possible higher-level optimizations. Data flow analysis information may be used to perform *register deallocation*, in which operands stored in physical architecture registers are mapped to higher-level virtual registers. After performing optimizations using this representation, which would be easier in many cases, standard IMPACT register allocation could be used to achieve beneficial use of the physical registers. If efforts to map the program instructions to a higher-level representation are successful, it would be possible to perform more aggressive transformations, such as those to effectively utilize architectural features not supported in original executables. It may even be possible to use this framework to perform translation of executables to entirely different architecture platforms.

APPENDIX A. SUMMARY OF LBX86 PARAMETERS

The command-line syntax for running the Lbx86 program is the following:

Lbx86 -i *input_file* -o *output_file* -Pphase=*phasenum* -F*parameter_name*=*value* -F...

input_file: The input Mcode file name for the program being reoptimized. This file can come either from x86toM (in which case it must have a .mc extension) or from Lbx86 phase 2 run alone (must have .mco extension). The base name (without the extension) is also used to form the name of the input fixup file for phase 3.

output_file: The name of the file to be output by Lbx86. If phase 2 is run alone, this is an Mcode file (must have .mco extension); otherwise it is an assembly language file (must have .s or .asm extension). The names of the object and processed fixup output files produced (if phase 3 is run) are formed using the base name of this file.

phasenum: A single digit indicating what phase(s) are to be run:

2	for just phase 2,
4	for just phase 3, or
6	for both phase 2 and phase 3.

parameter_name and *value*: Sets a parameter to a certain value as described below.

The IMPACT parameters relevant to running Lbx86 are described below. They can be set either in the IMPACT parameter file (STD_PARMS) or on the command line using the -F option. When used in the parameter file, the parameters must be placed in the appropriate parameter file section. The section is not necessary on the command line; also, note that parameters set on the command line override the values set in the parameter file.

- architecture section
  - arch
    The name of the architecture to use for code generation. Always use bx86.
  - lmdes
    The name of the machine description database file (.lmdes2 extension). See Section 4.4.1.

- Mcode section
  - do_postpass_sched
    Always set to yes because scheduling (if performed at all) is being done on code that is already register-allocated.
- Scheduler section
  - do_postpass_scheduling
    Always set to yes for the same reason as described above.
- global section
  - max_dest_operand
    Always set to 4 as a function of the specific Mcode format used. See Section 3.1.
  - max_src_operand
    Always set to 7 for the same reason as described above.
- Lbx86 section
  - Lbx86_enable_dataflow
    Set to yes or no to control whether or not to utilize complete data flow analysis. See Section 7.6.
  - schedule
    Set to yes or no to control whether or not to perform rescheduling during phase 2. See Section 7.2.
  - k6_esi_plus0_opti
    Set to yes or no to control whether or not to perform the encoding optimization described in Section 7.3.
  - clear_using_mov
    Set to yes or no to control whether or not to perform the instruction-selection optimization described in Section 7.4.
  - imulchain_reverse_conversion_threshold
    Set to an integer value for the desired threshold to control the constant `imul` optimization described in Section 7.5. Set to 0 to disable the optimization.
  - print_Lbx86_opti_stats
    Set to yes or no to control whether or not to output statistics relating to the optimizations performed during a given execution of Lbx86 phase 2.

APPENDIX B. SUMMARY OF PEWRITE PARAMETERS

The command-line syntax for running the PEwrite program is the following:

PEwrite -F*parameter_name*=*value* -F...

parameter_name and *value*: Sets a parameter to a certain value as described below.

The IMPACT parameters relevant to running PEwrite are described below. They can be set either in the IMPACT parameter file (STD_PARMS) or on the command line using the -F option. When used in the parameter file, the parameters must be placed in the appropriate parameter file section. The section is not necessary on the command line; also note that parameters set on the command line override the values set in the parameter file. For more information on the concepts discussed below, refer to Chapter 6.

- PEwrite section
  - output_path
    Set to the name of the directory into which all output files will be written.
  - input_exe_name
    Set to the name (optionally with path) of the original executable file being reoptimized.
  - input_coff_name
    Set to the name (optionally with path) of the reoptimized object file produced by Lbx86 phase 3.
  - input_fixups_name
    Set to the name (optionally with path) of the processed fixup file produced by Lbx86 phase 3.
  - output_exe_name
    Set to the name of the output executable to be produced by PEwrite. The file will be placed into the directory specified by output_path.

APPENDIX C. SUMMARY OF X86 MACHINE INSTRUCTION ENCODING SPECIFICATION FORMAT

The machine instruction encoding database was explained in Section 5.2. Each entry in the specification of this database represents a single proc_opc value; example entries can be seen in Figure 5.2 on page 35. The various fields that can be used as a part of each entry are described below; note that only the `opcode` field is strictly required. The items associated with each field are also explained, including example values. Note that the `${`*`symbol`*`}` notation represents a predefined constant.

- `operand1(`*`Mcode_position type`*`)`

  Names operand 1, specifying where it is located in the Mcode format and its type. The number `1` is significant as the number by which this operand will be referred from other fields.

  - *`Mcode_position`*
    Specifies where the operand should be found in the Mcode format for this operation. Example values include `${SRC0}`, `${DEST2}`, and `${SRC1_AND_DEST1}`.
  - *`type`*
    Specifies the type of the operand. Example values include `${TYPE_GENREG}`, `${TYPE_ADDR}`, and `${TYPE_INT}`.

- `operand2(`*`Mcode_position type`*`)`

  Names operand 2; see `operand1` above.

- `operand3(`*`Mcode_position type`*`)`

  Names operand 3; see `operand1` above.

- `operand4(`*`Mcode_position type`*`)`

  Names operand 4; see `operand1` above.

- `prefix(`*`byte_value`*`)`

  Specifies a single byte to be the very first byte of the instruction encoding. This field is typically used for the repeat prefixes since other prefixes are handled by other fields.

  - *`byte_value`*
    Specifies the actual value of the byte to be used, such as `0xF3` for the `rep` prefix.

- `opsize(`*`size_identifier`*`)`

  Specifies that the instruction *may* need to use an operand-size override prefix to support the use of 16-bit operands.

  - *`size_identifier`*
    Gives information on how to determine whether or not to use the operand-size override prefix. If values such as `${SIZE_FOLLOW_OP1}` are used, the specified operand (as named by `operand1`, etc.) is inspected. Similarly, `${SIZE_FOLLOW_MEM}` causes the memory access size to be inspected. If the value `${OPSIZE_IFDEFAULT32}` is used, an operand-size override prefix will always be used if the default operand size is 32 bits; this is always the case for Windows NT.

- `addrsize(`*`size_identifier`*`)`

  Specifies that the instruction *may* need to use an address-size override prefix to support the use of certain special 16-bit operands.

  - *`size_identifier`*
    Gives information on how to determine whether or not to use the address-size override prefix. In general, values such as `${SIZE_FOLLOW_OP1}` should be used, as explained for `opsize` above. Since 16-bit memory addresses are not currently supported, this field is generally used only in special cases. One such example is when it is necessary to cause the `loop` instruction to use the `cx` register instead of the `ecx` register as its loop counter.

- `popval(`*`fpstack_pop_amt`*`)`

  Specifies how many extra floating-point stack operands are popped by this instruction. This information may be used in the future to enable floating-point stack analysis.

  - *`fpstack_pop_amt`*
    An integer value corresponding to the number of extra floating-point pops performed.

- `opcode(`*`value`*`)`

  Typically specifies the binary machine opcode for the instruction. May also be used to convey special-case handling information.

  - *`value`*
    The normal use of this field is to specify the actual byte value of the instruction's binary machine opcode. A 1-byte opcode might look like `0x80`. A 2-byte opcode for which the first byte is $0F_{hex}$ and the second byte is $9A_{hex}$ must be represented in little-endian fashion as `0x9A0F`. Bit positions into this value are numbered in little-endian fashion as well, so that the least-significant bit of the `0F` byte is bit 0, the least-significant bit of the `9A` byte is bit 8, and so on. The possible special values for this field are `${ILLEGAL}`, meaning that this particular proc_opc is not a valid one, as well as `${OPCODE_CBR}` and `${OPCODE_RELJMP}`. The latter two are used to provide special handling for relative address instructions whose opcodes vary depending on the encoded size of the relative address operand.

- `s(`*`bitpos_into_opcode`*`)`

  Used to indicate the possible need to set a certain bit in the binary machine opcode, depending on whether or not the instruction can use an 8-bit sign-extended immediate value. When this field is present, the `opsize` and `immed` fields must also be present. Furthermore, the *`size_identifier`* items of these two fields must agree. The information provided by these other fields is used to determine whether or not to set the bit. This decision then influences the encoding produced by the `immed` field.

  - *`bitpos_into_opcode`*
    An integer value for the bit position in the binary machine opcode where the bit resides. For more information on bit position numbering, see the `opcode` *`value`* item above. Note that the bit value of the `opcode` field should be 0 at this position, so that it can be selectively set.

- `w(`*`bitpos_into_opcode`*`)`

  Used to indicate the possible need to set a certain bit in the binary machine opcode, depending on whether or not the instruction uses an 8-bit operand. When this field is present, the `opsize` field must also be present. The information provided by this other field is used to determine whether or not to set the bit.

  - *`bitpos_into_opcode`*
    An integer value for the bit position in the binary machine opcode where the bit resides. For more information on bit position numbering, see the `opcode` *`value`* item above. Note that the bit value of the `opcode` field should be 0 at this position, so that it can be selectively set.

- `mf(`*`mf_type`*`)`

  Used to indicate the need to write the value of a certain 2-bit bitfield in the binary machine opcode of a floating-point instruction, depending on the size of memory accessed by the instruction. The bit values at bit positions 1 and 2 in the binary machine opcode should be 0 so that the value of this bitfield can be set.

  - *`mf_type`*
    Must be either `${MF_INT}` or `${MF_REAL}`, depending on whether the instruction form is accessing an integer or a floating-point value in memory, respectively.

- `d(`*`d_type`*`)`

  Used to indicate the possible need to set a certain bit in the binary machine opcode of a floating-point instruction, depending on the order of the two floating-point stack operands accessed by the instruction. It also influences a portion of the ModR/M byte (see `modxrm` below).

  - *`d_type`*
    Must be either `${D_REV}` or `${D_NOTREV}`, depending on whether or not the instruction is a "reversed" floating-point instruction such as `fdivr`. This type does not impact the setting of the binary machine opcode bit described above, but rather influences the way that the ModR/M byte is handled.

- `regfield(`*`whichop`*`)`

  Used to indicate the need to write the value of a certain 3-bit bitfield in the binary machine opcode, depending on which register is being accessed by a certain instruction operand. The bit values at the three least-significant bit positions of the most significant byte in the binary machine opcode should be 0 so that the value of this bitfield can be set.

  - *`whichop`*
    The number of the operand to be inspected in order to determine the value of the bitfield. If this value is `1`, for instance, then the operand specified by the `operand1` field will be inspected.

- `modxrm(`*`mod x rm`*`)`

  Used to indicate the presence of a ModR/M byte in the instruction. Its presence also indicates that a SIB byte and one or more memory address displacement bytes may also be necessary. The rules for encoding these various bytes are explained in [4].

  - *`mod`*
    Must be either `${MOD_11}` or `${MOD_NOT11}` depending on whether the two bits of the Mod field of the ModR/M byte should both be forced to 1 values or should be determined based on the memory address, respectively.

  - *`x`*
    Determines how the middle three bits of the ModR/M byte are encoded. Values such as `${X_REG_OP1}` or `${X_SREG3_OP2}` cause the bits to be encoded based on what specific register is used for a certain operand. It is possible to specify two of the three bits with values such as `${X_REG_01R}`, with the third bit (the R bit) being determined based on the order of floating-point stack operands and the `d` field's *`d_type`*. Finally, this item can simply be set to a constant integer value in the range of 0 to 7, and the three bits will simply be filled in with the binary value of that integer. This last use is common for opcode extensions.

  - *`rm`*
    Determines how the least-significant three bits of the ModR/M byte are encoded. The `${RM_MEM}` value, which is always used in conjunction with `${MOD_NOT11}`, is for encoding memory addresses. Values such as `${RM_REG_OP1}` or `${RM_FSTi_OP2}` cause the bits to be encoded based on what specific register (or floating-point stack element) is used for a certain operand. As with the *`x`* item, a constant integer value in the range of 0 to 7 can also be used.

- `immed(`*`whichop size_identifier`*`)`

  Used to indicate the presence of one or more bytes for the immediate operand in the binary instruction encoding.

  - *`whichop`*
    The number of the operand to be inspected in order to determine the value of the immediate operand.

  - *`size_identifier`*
    Used to determine the number of bytes that should be used to encode the immediate operand. The size is fixed if values such as `${SIZE_BYTE}` or `${SIZE_DWORD}` are used. If instead values such as `${SIZE_FOLLOW_OP1}` or `${SIZE_FOLLOW_MEM}` are used, the size of the immediate operand will be determined based on the size of another operand. In such cases, the presence of the `s` field may also influence the size of the immediate operand. Note that, when

"following" the size of another operand, the operand being followed should not be the same as that indicated by the *`whichop`* item.

- `immed2(`*`whichop size_identifier`*`)`

  Used to indicate the presence of one or more bytes for the second immediate operand in the binary instruction encoding. The second immediate operand will be placed after the first immediate operand.

  - *`whichop`*
    The number of the operand to be inspected in order to determine the value of the second immediate operand.
  - *`size_identifier`*
    Used to determine the number of bytes that should be used to encode the second immediate operand. In this case, only fixed-size values such as `${SIZE_BYTE}` or `${SIZE_DWORD}` are supported.

- `reladdr(`*`whichop size_identifier`*`)`

  Used to indicate the presence of one or more bytes of relative address in the binary instruction encoding.

  - *`whichop`*
    The number of the operand to be inspected in order to determine the target (and thereby the value) of the relative address.
  - *`size_identifier`*
    Used to determine the number of bytes that should be used to encode the relative address. The size is fixed if values such as `${SIZE_BYTE}` or `${SIZE_DWORD}` are used. If instead values such as `${SIZE_FOLLOW_OP1}` are used, the size of the relative address is variable and will be determined based on the relative magnitude needed to reach the target. In such cases, the operand being "followed" should be the same as that indicated by the *`whichop`* item.

- `suffix(`*`byte_value`*`)`

  Specifies a single byte to be the very last byte of the instruction encoding. This field is typically used for advanced opcode extensions.

  - *`byte_value`*
    Specifies the actual value of the byte to be used as a suffix, in a form such as `0x93`.

REFERENCES

[1] B. T. Sander, “Performance optimization and evaluation for the IMPACT x86 compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1995.

[2] P. P. Chang et al., “IMPACT: An architectural framework for multiple-instruction-issue processors,” in *Proceedings of the 18th Annual Int’l Symposium on Computer Architecture*, Toronto, Canada, May 28, 1991, pp. 266-275.

[3] W. W. Hwu et al., “Compiler Technology for Future Microprocessors” in *Proceedings of the IEEE*, Vol. 83, No. 12, December 1995, pp. 1625-1640.

[4] *Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference*, Intel Corporation, 1997.

[5] Intel Corporation, “Software benefits of Katmai New Instructions,” May 1998, http://developer.intel.com/drg/news/katmai.htm.

[6] *3DNow! Technology Manual,* Advanced Micro Devices, Incorporated, May 1998.

[7] Visual C++ Business Unit, *Microsoft Portable Executable and Common Object File Format Specification 4.1,* MSDN Library, Microsoft Corporation, August 1994.

[8] R. Kath, “The Portable Executable file format from top to bottom,” MSDN Library, Microsoft Corporation, June 1993.

[9] “winnt.h” Source code file distributed with Microsoft Visual C++, 1997.

[10] “IMPACT Lcode tutorial,” IMPACT Research Group, University of Illinois, 1998. Located in the pending IMPACT release under *impact/tutorials/lcode_tutorial*.

[11] R. Bringmann, “A template for code generator development using the IMPACT-I C compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1992.

[12] M. C. Merten and M. S. Thiems, “An overview of the IMPACT x86 binary reoptimization framework,” The IMPACT Research Group, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-98-05, 1998.

[13] “IMPACT Lcode control flow tutorial,” IMPACT Research Group, University of Illinois, 1998. Located in the pending IMPACT release under *impact/tutorials/ lcode_controlflow.*

[14] R. Hank, “Region-based compilation,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1996.

[15] R. Bringmann, “Enhancing instruction level parallelism through compiler-controlled speculation,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[16] J. Gyllenhaal, “A machine description language for compilation,” M.S. thesis, University of Illinois, Urbana, IL, 1994.

[17] J. Gyllenhaal, “An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1997.

[18] “IMPACT Lcode Analysis Tutorial,” IMPACT Research Group, University of Illinois, 1998. Located in the pending IMPACT release under *impact/tutorials/ lcode_analysis_tutorial*.

[19] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, “Optimization of machine descriptions for efficient use,” in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 349-358.

[20] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, “HMDES version 2.0 specification,” The IMPACT Research Group, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1998.

[21] K. D. Safford, “A framework for using the Pentium’s performance monitoring hardware,” M.S. thesis, University of Illinois, Urbana, IL, 1997.

[22] *AMD-K6 MMX Enhanced Processor Data Sheet*, Advanced Micro Devices, Incorporated, 1997.

[23] *AMD-K6 MMX Enhanced Processor x86 Code Optimization Application Note*, Advanced Micro Devices, Incorporated, August 1997.

[24] *Intel Architecture Optimization Manual*, Intel Corporation, November 1996.