AN OVERVIEW OF THE IMPACT x86 BINARY REOPTIMIZATION FRAMEWORK

BY

MATTHEW C. MERTEN
AND
MICHAEL S. THIEMS

ECE 498 Independent Study Report
Spring, 1998

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Computer software is most often distributed in binary form, so that users do not have access to the original source code. When vendors compile their software to binary form, they optimize for a certain target processor. If the user does not have that particular processor, the performance of the binary on the user's machine may not be optimal. To avoid excluding users with older machines, software vendors often avoid using the newest architectural features. Therefore, the user's processor may have new, performance-enhancing features that are not utilized by the distributed binary. Differences in the underlying microarchitecture may also favor the use of different instruction sequences to achieve the same functional purpose. In all of these situations, the end user would benefit from having an executable that is fully optimized for his or her specific machine.

While traditional compiler technology may be adapted to generate more optimal code for a certain processor, the unavailability of source code to users and even microprocessor manufacturers makes this technology inapplicable to the stated problem. The first goal of this project is to create a bridge between distributed binary programs and an internal representation of the IMPACT compiler. Doing so enables the application of IMPACT's established compiler technology tools [1], [2] to the reoptimization of binary programs, the primary aim of the project.

The Intel x86 instruction set [3] and Microsoft's 32-bit Windows operating systems (Windows 95 and Windows NT) are well-established standards in the market. A huge 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 [4]. For these reasons, this platform was chosen as an appropriate one in which to develop our 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 remainder of this report is organized as follows: Chapter 2 provides an overview of the x86 binary reoptimization system. Chapter 3 describes our use of IMPACT's low-level intermediate representation. Chapter 4 describes each of the software modules in more detail. Chapter 5 provides a summary and concluding remarks.

2. SYSTEM OVERVIEW

2.1. Input Requirements

The structure of a Windows 32-bit executable file is the Portable Executable (PE) file format [5], [6], [7]. 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, allowing our 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, and so the base relocation table is required by our system. A symbol table may also be present in the executable. The information provided by the table is helpful in locating the beginning of functions in the program, but is not required by our system.

2.2. Processing Steps

Figure 2.1 outlines the process of reoptimizing binaries using our system. First, 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 [8], [9]. Several

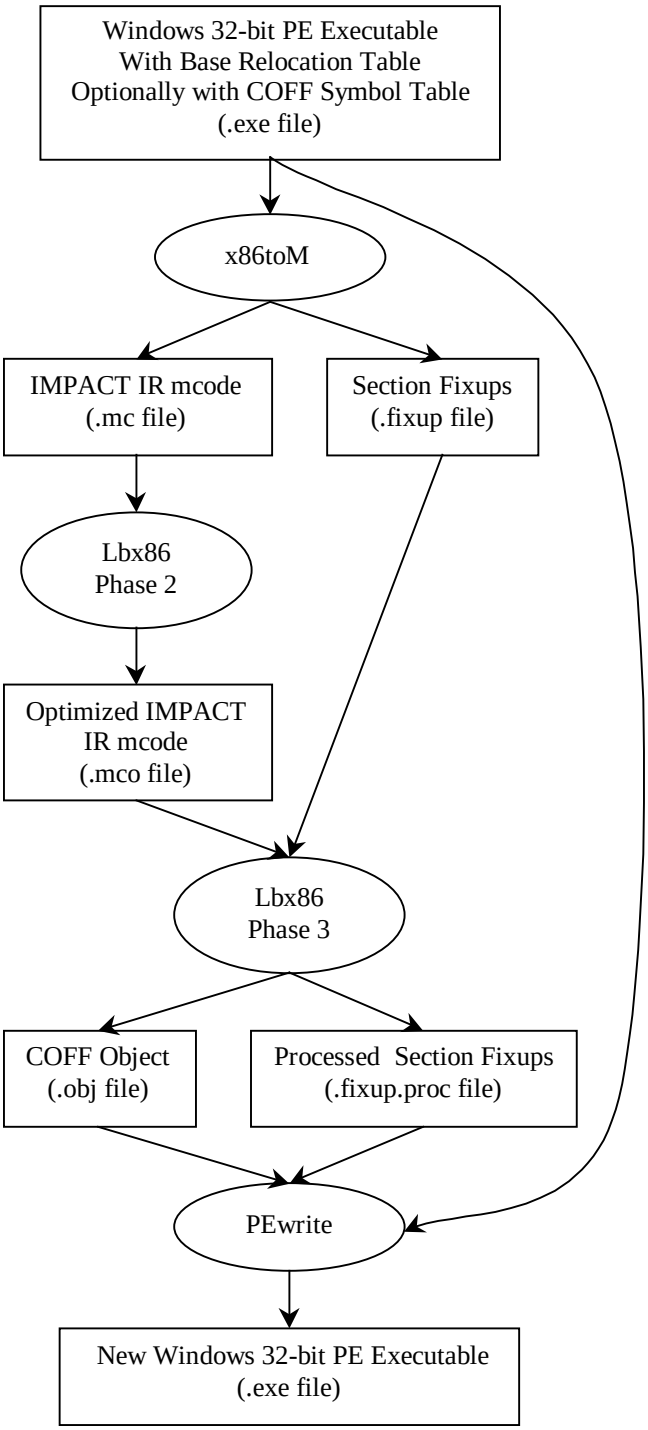

Figure 2.1 Binary reoptimization steps

other sections exist in the binary but are not converted since they will not be significantly modified by our system. However, these other sections do contain pointers into sections that will be modified, and thus those pointers will require modification at link time. For this reason, a *fixup* file is also produced, containing a mapping of the pointers to Mcode symbolic names.

The Lbx86 program performs the next two processing steps in two separate phases. The first phase, which for historical reasons is called phase 2, performs 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 the second phase of Lbx86, called 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 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, our 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 our system be capable of representing the entire x86 user instruction set. To enable dependence analysis, all register accesses by an instruction should be explicitly represented in its Mcode representation. For this system we 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 flags registers in the x86 architecture.

3.2. Typical Instruction Format

An example of an instruction represented in Mcode is shown in Figure 3.1. This instruction 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 `add` instruction. Inside the first set of square brackets are the destination operands. This particular addition instruction writes its results into the `eax` register and modifies six of the condition code flags: *overflow*, *sign*, *zero*, *auxiliary carry*, *parity*, and *carry*. Inside the next set of square brackets are the source operands. This instruction adds four to the contents of the `eax` register, and then adds one more if the carry flag is set. Inside the angle brackets are the attributes. One of the required attributes is the processor-specific opcode mnemonic, called `gen_opc`, which is provided as a text string for readability. Another required attribute is the `popc`, a numeric value use internally by the system that corresponds to the `gen_opc`. This will be explained further in Section 4.2.1.

```
(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*

This example also illustrates a general principle followed in the design of our 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 unused operand positions.

Because x86 is a CISC 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. Figure 3.2 illustrates the instruction "`or BYTE PTR [eax+ebx*4+24], 64`," in which a single

memory location is both a source and a destination. Our system uses the `addr` macro operand to indicate the presence of a memory address specification in an instruction. This specification is 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. An attribute is used to specify the type of memory access performed, either `mem_read`, `mem_write`, or `mem_read_write`. The `mem_size` attribute specifies the number of bytes accessed.

```
(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

3.3. 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, no operands are specified; they are all implicit. This instruction had the *repeat* prefix in the original binary that made it a one-instruction loop. This is also evident in the Mcode as the `ecx` loop control operand is read, which is the bounds check, and written, which is the decrement. Register `edi` contains the destination memory address, which 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* is read to determine whether to increment or decrement the memory address registers. 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. The purpose of this *string move* illustration is to show that even complex, non-conforming instructions can be accurately represented in Mcode.

```
(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 movsw))(popc (i 1472))>)
```

Figure 3.3 Mcode representation of a *string move*

4. SOFTWARE

4.1. X86toM

The primary function of x86toM is to convert the executable code in the binary into a Mcode representation. This conversion process proceeds in two phases. The first phase consists of code discovery, which involves instruction decoding and control flow analysis. During the second phase, the program is converted to the Mcode representation. Finally, the fixup file is produced.

4.1.1. Instruction decoding and control flow analysis

Instructions are decoded starting at the initial entry point of the program. Note that the variable length instruction feature of the x86 architecture makes decoding more complicated than for RISC architectures, since in RISC architectures instructions can always be found at word boundaries. As branching instructions are encountered, their target addresses are pushed onto a stack. Instruction decoding continues along fall-through paths until an unconditional branch, a return, or a previously decoded instruction is reached. The address stack is then popped, and decoding continues from the popped address.

Special care must be taken when jumps through tables are encountered. Heuristics have been developed to find jump tables based on the displacement contained in the jump instruction. The displacement is the only information available, without extensive analysis of prior instructions, that can be used to locate the jump table. Unfortunately, the displacement can be misleading as some jump tables are not indexed in the positive direction and some have non-zero starting offsets. Figure 4.1 depicts a negatively indexed jump and its jump table from

the Microsoft Visual C++ 5.0 library function `memmove`. Note that the displacement in the jump instruction in control block 30 actually points to another *instruction* in control block 10. During the initial search for the jump table, code was found instead of a pointer to code, which would have been marked as a relocatable address. The heuristic then locally searches both lower and higher addresses for relocatable entries. Note that this is just a heuristic, and that the jump table found might not be the correct one. Plans for more extensive analysis have been formulated and will be explored in the future, but it is unclear that this problem can be solved completely.

```
## control block 30:
   004171B4 : FF 24 8D B8 72 41 00 jmp dword ptr [4*ECX+004172B8h)]

@@ jump table 6:
   004172A8 : 004172B8 (cb 10)
   004172AC : 004172C0 (cb 9)
   004172B0 : 004172CC (cb 8)
   004172B4 : 004172E0 (cb 7)

## control block 10:
   004172B8 : 8B 45 08            mov     EAX, dword ptr [EBP+08h]
```

Figure 4.1 X86toM phase 1 output of a jump through jump table with negative indexing

When a call is encountered, its target address is pushed onto a call address stack. Once the branch target address stack is empty, a new function is decoded from the call address stack. If a symbol table is available, its information can be used to locate additional function entry points that have not previously been detected. Other heuristics are employed to detect functions that are only accessed via a callback, i.e. the address of the function is passed as data to another function. At the end of this first phase, a complete map of the code section is produced.

4.1.2. Conversion to Mcode

The second phase of conversion consists of conversion to Mcode. This phase is generally straightforward as all decoding and flow analysis was completed in the first phase. When creating control block references in Mcode, such as the target of a branch, the actual control block must exist. Therefore each Mcode function is created first, along with its control blocks, but without any instructions. Then in a second pass, the instructions are added to the correct control blocks. Each instruction is converted to a Mcode representation by a set of annotation functions. In addition to the instructions themselves, *flow arcs* must be created, which specify the potential flow of control between the control blocks [10]. As control flow altering instructions are processed, flow arcs are created to their destinations. In a third pass, fall-through flow arcs are added, and a special function is used to analyze jump tables in order to create the appropriate flow arcs.

Figure 4.2 depicts the Mcode representation of the jump though jump table with negative indexing shown earlier. Note the addressing used in the *indirect jump* (`jump_rg`) instruction. The displacement field is represented differently than in the original binary in order to properly identify which jump table is used. The label `_section_text_90792` references the base name of the jump table. The table is given a base name derived from its location in the original executable so that the tables can be distinguished from each other. The `+16` indicates an offset from the base of the jump table. Together, they represent the displacement from which to index. Finally, note the four flow arcs attached to control block (`cb`) 30, which are denoted by the `flow` token. The first integer in each flow arc represents the condition code, which in the case of a jump table is the index into the table. In this example, they are all negative. The second integer represents the target control block, and

the last floating-point value is reserved for a profile weight if one is know. Also shown is the Mcode representation of the jump table itself. Each entry in this representation of the jump table consists of an offset from the base label followed by a label corresponding to the correct control block.

```
(function _memmove_453_ 0.000000 <L>
    <(jump_tbls (i 6)(i 5)(s_l_abs "renamed"))>)
 ...
  (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 4.2 X86toM phase 2 output of a jump through jump table with negative indexing

One assumption made by the IMPACT compiler is that a function must be represented as having a single entry point. However, real functions found in binaries may have several entry points. If during decoding a branch is found whose target is in a different function, or if one function falls through to another, the two functions are merged into a single function with multiple entry points. The representation of multiple entry point functions in Mcode was developed in conjunction with IMPACT's region-based compilation [11]. Special control blocks and instructions are added to the Mcode, and marked accordingly, that make the function appear to IMPACT to have only a single entry point. In order to represent this

situation correctly, a special control block is added to the beginning of the function called the *prologue*. In this control block is a single *indirect jump* instruction that branches to each of the programs entry points, and is also marked `prologue`. Those branch targets are indicated by flow arcs out of the prologue control block to each of the entry point control blocks. Each entry point control block is tagged with an attribute that contains that entry point's original name. This organization physically connects each of the real entry points of the function to the single IMPACT entry point. In addition to the physical representation, this system preserves proper data and control flow, which will be discussed further in Section 4.2.2.

```
(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 4.3 X86toM phase 2 output for a function with multiple entry points

Figure 4.3 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.

4.1.3. Fixup file

Finally, x86toM creates the fixup file. This file represents changes that need to be made at link time to the unconverted sections. All unconverted sections are scanned for relocatable entries, which are pointers, that point into the converted sections. Due to optimization and other conversion issues, the addresses of the code or data pointed to by the relocatables will be different when the executable is put back together. For each pointer that will be different in the new executable, because the code or data was moved, an entry is made in the fixup file mapping the contents of that pointer to a symbolic intermediate representation name. An example of a fixup is shown in Figure 4.4. Here, the *read-only data section*, `_section_rdata_`, contains a relocatable address that points into the code section, specifically to control block 9 of the function `mainCRTStartup`. Since the aforementioned control block and function will not be located at the same address in the new executable as in the old, the fixup allows the linker to correct the pointer. Since the linker will need to point to

the overall entry point for the program, the fixup file also includes the appropriate program entry point symbol.

```
_section_rdata+164        cb9_mainCRTStartup_1_
```

Figure 4.4 An example fixup file entry

4.2. 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.

4.2.1. Instruction type identification

One important feature of the Mcode data structure is the numeric processor-specific opcode, known as the `proc_opc` or `popc`, which is associated with each instruction. Though this field may be used in any desired way for a given Mcode format, it is also used by the general IMPACT scheduling system [12], [13]. 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 opcode 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 portions in our system. The two portions are known as the *general opcode*, or *genopc*, and the *variant*. The genopc corresponds to the processor-specific opcode mnemonic, such as `add`, and thus is directly correlated to the `gen_opc` text string in the Mcode format. 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. 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.

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 zero in this original Mcode. One of the preprocessing steps taken by Lbx86 is to add the proper variant values for use by the rest of Lbx86. This is accomplished by using the genopc value to choose an appropriate variant annotation function. That function deduces the appropriate variant for the operation based primarily on its operands. Later, if a transformation adds a new operation to the code, the appropriate variant annotation function must be called.

4.2.2. Dependence analysis

The IMPACT compiler system includes a *scheduler manager*, called SM, which facilitates code analysis for transformation and scheduling purposes [14]. 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 *dataflow analysis* component to provide information about which operands are live in to and out of the current control block [15].

Dependence analysis by SM and dataflow 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 4.5, 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. The `mov` and `add` instructions are unrelated to the `cmp` and `jb`, and the condition code flags set as a side effect of the `add` instruction are insignificant to the operation of this code sequence. However, 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 our system by explicitly modeling the condition code flags as registers and including the reads and writes in Mcode, as mentioned in Section 3.2.

```
mov  eax, 1    (writes no condition code flags)
add  ebx, 4    (writes condition code flags as side effect)
cmp  ecx, edx  (writes condition code flags)
jb   $L5$2     (reads condition code flags)
```

Figure 4.5 Assembly language example illustrating importance of side effects

Though the Mcode format used in our system includes features to allow dependence analysis of memory accesses, or *memory disambiguation*, the analysis is not currently implemented. Also, the stack model of x86 floating point instructions invalidates a traditional register-based dependence analysis. Both memory and floating point register accesses are currently treated conservatively for dependence analysis purposes.

In addition to analyzing the data dependence relationships between instructions in the same control block, dataflow 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. 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 our system's representation of multiple entry point functions as discussed in Section 4.1.2. 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 4.3). The

corresponding flow arcs created are recognized by dataflow 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, dataflow analysis will provide correct results for multiple entry point functions.

Functions with multiple exit points, which have been observed to be very common in x86 code, also require special handling for proper dataflow 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 dataflow analysis. Like the multiple entry point representation, this transformation will be undone before the new object file is written out in Lbx86 phase 3.

When SM integrates dataflow 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 dataflow 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.

4.2.3. 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 optimizations are possible. One important special case is rescheduling, which is performed by SM based on a machine description [16], [17]. Different machine descriptions can be created for different microarchitectural implementations of the same instruction set, enabling high quality machine-specific rescheduling. The machine description must classify each possible `proc_opc` value according to its characteristics. It must also model the use of different machine resources by each instruction and their interactions.

4.3. 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 [5]. A corresponding assembly language file is also produced, though this is intended primarily for debugging purposes. To maintain symbol compatibility with the new object file, the fixup file produced by x86toM is processed.

4.3.1. 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 which 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 4.2.2 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 which 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.

4.3.2. 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 encoding specification. In the x86 architecture, the problem is compounded by 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.

In our system, a database is created using the IMPACT meta-description (MD) language facility [17]. 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* [3]. In this way, we strive to make the database specification clear and maintainable. The database specification is converted from its high-level textual form to a low-level form read by Lbx86 phase 3, which includes functionality to interpret the database and generate binary encoded machine instructions based on it.

4.3.3. Address resolution

When machine code is generated, most references to addresses are handled by creating COFF relocation entries, to be filled in 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, and 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. Because relative addresses can often be encoded using different numbers of bytes depending on their magnitude, extra address resolution passes are used to achieve optimal encoding.

4.3.4. 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 its relocation entries are also 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.

4.3.5. 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 output by Lbx86 phase 3, it is also necessary to adjust the symbols in the fixup file accordingly. Also, 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 to resolve the fixups.

4.4. 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. In our system, we have 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. Also, our object file 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 our 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. 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 non-converted 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.

5. CONCLUSION

The binary reoptimization system described herein has been implemented under Windows NT, and handles 32-bit Windows x86 executables. A flexible Mcode format has been developed to accurately represent data and control flow information for the complex x86 instruction set. Existing IMPACT compiler technologies are utilized to provide detailed analysis of the converted programs, to facilitate rescheduling and other optimizations, and to provide a maintainable framework for the generation of new executables. The complete system will allow further research into specific types of optimizations that can be applied to existing binary programs.

REFERENCES

[1] Pohua 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.

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

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

[4] *3DNow! Technology Manual*, Advanced Micro Devices, Incorporated, 1998.

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

[6] R. Kath, "The Portable Executable File Format from Top to Bottom," MSDN Library, Microsoft Corporation, June 1993.

[7] "winnt.h" Source code file distributed with Microsoft Visual C++, 1997.

[8] IMPACT Lcode Tutorial, IMPACT Research Group, University of Illinois, 1998. Located in the pending IMPACT release under *impact/tutorials/lcode_tutorial*.

[9] Roger Bringmann, "A Template for Code Generator Development Using the IMPACT-I C Compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana IL, Sept. 1992.

[10] IMPACT Lcode Control Flow Tutorial, IMPACT Research Group, University of Illinois, 1998. Located in the pending IMPACT release under *impact/tutorials/lcode_controlflow.*

[11] R. Hank, "Region-Based Compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, May 1996.

[12] Roger Bringmann, "Enhancing Instruction Level Parallelism Through Complier-Controlled Speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana IL, May 1995.

[13] J. Gyllenhaal, "A Machine Description Language for Compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, Sept. 1994.

[14] J. Gyllenhaal, “An Efficient Framework For Performing Execution-Constraint-Sensitive Transformations That Increase Instruction-Level Parallelism,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, Sept. 1997.

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

[16] John C. Gyllenhaal, Wen-mei W. Hwu, and B. Ramakrishna Rau, “Optimization of Machine Descriptions for Efficient Use,” *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 349-358.

[17] John C. Gyllenhaal, W. W. Hwu and B. Ramakrishna Rau, “HMDES Version 2.0 Specification,” IMPACT Technical Report, IMPACT-96-03, University of Illinois, Urbana, IL, 1996.