A FRAMEWORK FOR INSTALL-TIME OPTIMIZATION OF BINARY DYNAMIC-LINK LIBRARIES

BY

CHRISTOPHER NEITH GEORGE

B.S., University of Illinois at Urbana-Champaign, 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, 1999

Urbana, Illinois

ACKNOWLEDGMENTS

The work described in this thesis has undergone many evolutions and owes many debts to many people. I would like to thank my advisor, Professor Wen-mei Hwu, for his teaching and advice and for giving me a chance to prove that I could get the job done. John Gyllenhaal was a source of much advice and a fountain of valuable information regarding the machine description language and the IMPACT schedule manager. Matthew Merten and I have been working side-by-side on this project for the past year and a half, and he has been a tremendous help in working through ideas, debugging my code, and doing anything else humanly possible to see that the project continued. Matt developed the instruction conversion to Mcode in the x86toM module and has done the work to incorporate object files into the framework. Mike Thiems helped to develop much of the framework and worked extensively on the optimizations and binary code generation in the Lbx86 module. The decoding framework and PEwrite code were developed in the initial phase of the project with John Sias, Guanyao Cheng, and myself, and John was an excellent mentor. Thanks to all of the members of the IMPACT group and AMD's Advanced Architecture Labs for their technical guidance. Special thanks go to my family for all the years of support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Today's computer systems have evolved from highly specialized research units into multi-functional machines with a wide application base. Computers are equipped to fulfill many needs, and many different machines can be supplied to satisfy the same demand. Consequently, the application software available on the market is written and compiled generically to run on a variety of machines in many different situations.

As a result of this generic compilation, commercial compilers do not incorporate targeted optimizations designed to take advantage of processor-specific features. The use of a reoptimization system can allow individual users to tailor their software to the particular features of their machine. By incorporating the reoptimization into the software installation, the additional time necessary to process the binary becomes relatively transparent to the user. Furthermore, install-time optimization allows the system to analyze the processor's features and take advantage of the information included in the binary by the compiler, without adding overhead to the actual execution of the binary.

The *IMPACT binary reoptimization framework* accomplishes this static reoptimization without the need for program source code or any sort of access to the linker or loader. The framework can be configured to perform the transformations at install-time. The framework currently targets the Intel x86 instruction set [1] plus MMX and AMD

3DNow! [2] extensions and executes under Microsoft's 32-bit Windows operating systems (Windows 95/98 and Windows NT).

The framework is capable of analyzing *binary executables* (EXEs), *object files* (OBJs), and *dynamic-link libraries* (DLLs). The increasingly common trend among software developers is to insert commonly and intensively executed code in DLLs for shared access by many executables. Additionally, software patches and updates can be released through DLLs rather than by supplying a new copy of the main executable. By focusing on the optimization of DLLs, the framework can achieve a benefit for a wide variety of applications without having to modify the main executables. As users update their systems, or as new enhancements reach the industry, the DLLs can be reoptimized with relative ease.

This thesis describes the static code discovery algorithms employed by the IMPACT binary reoptimization framework and expands upon some optimization and binary write-back issues not explored in the theses of M.S. Thiems [3] and M.C. Merten [4]. When taken together, the three theses provide a complete description of the methods and benefits of the optimization framework, as well as the issues and future work involved. In this thesis, Chapter 2 summarizes some related work in the area of optimization. Chapter 3 provides an overview of the binary optimization framework. Chapter 4 describes the format of the binary input to the optimizer. Chapter 5 details the process of code discovery and decoding the binary. Chapter 6 explains some optimizations related to code rescheduling based upon machine resources. Chapter 7 concerns the generation of the optimized binary code, and Chapter 8 describes the writeback of the code to a new binary file. Chapter 9 provides some experimental results to validate the approach and operation of the framework. Chapter 10 analyzes the impact of the work and discusses possible future enhancements. The appendices contain the structures used to represent the components of the binary, as well as a machine resource description for the AMD K6-2 processor.

CHAPTER 2

PREVIOUS WORK

At this point, others in both industry and academia have also undertaken efforts to analyze executables. Researchers at Compaq's Digital Equipment Corporation have developed the Digital FX!32 software package to allow 32-bit x86 Windows applications to run on a Digital Alpha workstation running Windows NT [5]. FX!32 employs seven modules to facilitate the translation. First, a transparent agent launches a 32-bit x86 application. Next, a runtime initializes the execution environment and loads the code image along with code fragments, or jackets, allowing the use of Alpha Windows NT API routines. An emulator then runs the x86 application.

During the first run of the program, the emulator executes the entire application on the Alpha NT platform and gathers profile information. A translator operates in the background and generates a translated file image based upon the information from the emulator. As translated code becomes available, it is fed to the emulator. On subsequent runs of the program, the emulator serves mainly as a backup to the translated code and activates when untranslated code is encountered. A database collects the profile data from the emulator, as well as the translated code images, for further use by the system. A server retains this database and initiates operation of the translator.

The FX!32 suite differs greatly from the framework established in this thesis. FX!32 relies on runtime profile data, rather than static analysis of the binary, and accomplishes translation through examination of this profile data and modification of the NT loader. Furthermore, the translator relies on the insertion of jackets, or code hooks, to direct the application to call the Alpha Windows NT API instead of the x86 API. While the goal of both frameworks is the rapid execution of applications on a certain platform, the translations are implemented in vastly different ways. FX!32 relies heavily on the NT loader and source code specific to the DEC Alpha, while the IMPACT binary translator is a more general-purpose, nonintrusive framework that also incorporates a large number of optimization capabilities.

The Morph system from Harvard University [6] integrates operating system and compiler technology to support program optimizations based upon profile data. While Morph does not require application source code, it needs an intermediate-form representation to perform its optimizations and cannot derive the intermediate form on its own. Additionally, Morph requires modifications to the UNIX kernel to collect the profile data upon which the optimizations are based.

The Morph system employs three main elements for optimization: the Morph Monitor, Morph Editor, and Morph Manager. The Monitor watches the application execution and collects profile data in conjunction with the modified UNIX kernel. The Manager processes the profile data off-line to convert it into a form usable by the Editor. The Editor uses the profile data and the supplied intermediate representation to perform optimizations. The compiler portion of the Morph Editor is based upon Stanford's SUIF compiler [7]. This thesis proposes a binary translation and optimization framework that creates its own intermediate representation from the supplied compiled binary and performs rapid optimization at install time with no modification to the operating system or kernel.

At the University of Washington, researchers have designed an API that can be used to develop tools for the measurement and optimization of Win32 x86 binaries. The system, known as Etch [8], requires no source code and allows the user to analyze the binary, evaluate program performance, and improve program performance. Etch differs from the IMPACT binary optimization framework in the method of code discovery. It also relies on outside developers to use the Etch API in their optimization efforts, while the IMPACT framework presents a unified system of decoding, optimizing, and binary rewriting. Additionally, Etch retains a copy of the original code in the optimized binary, while the IMPACT binary optimizer writes one optimized binary to prevent code size explosion.

CHAPTER 3

PROCESS OVERVIEW

The focus of the IMPACT binary optimization framework is to provide accurate and reproducible function-level profile data for common Win32 applications and dynamic-link libraries and to develop an interface into x86 Portable Executable (PE) binaries for binary reoptimization. The framework operates without any source code, and symbol information is optional. The system functions under both Windows NT and Windows 95/98 and does not modify any kernel, loader, or system components. The binary optimization framework is separated into four phases: decoding the binary, optimizing the code, regenerating the new code and data bytes, and reconstructing the binary executable, as shown in Figure 3.1.

Disassembly of the binary takes place during Phase I in the x86toM module, and, in order to facilitate code discovery, the optimizer extracts information stored in the PE file headers and tables. The header contains the sizes and offsets of the .text, .data, and other sections. The base relocation table is extracted from the header and is used by the optimizer to verify that addresses encountered are valid pointers and not simply data bytes. If present, the import and export tables reveal functions imported from external DLLs and list the addresses of functions that are being made available to other executables. Similarly, if available, the symbol table is read in and used to ensure that all

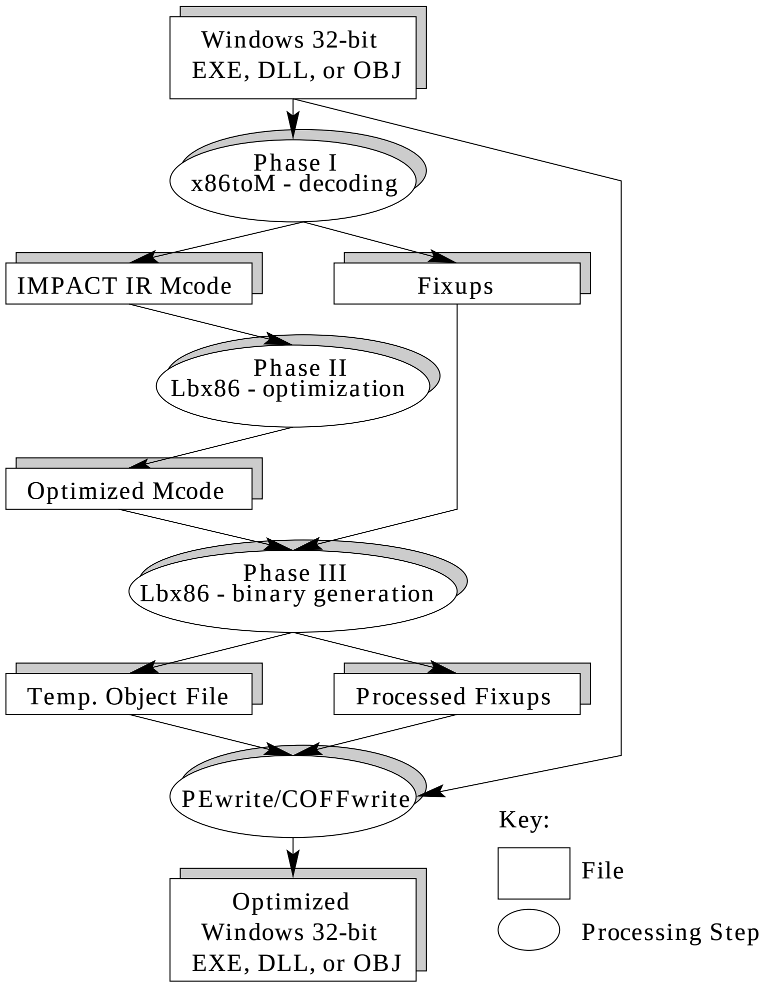
Windows 32-bit EXE, DLL, or OBJ
Phase I x86toM - decoding
IMPACT IR Mcode
Fixups
Phase II Lbx86 - optimization
Optimized Mcode
Phase III Lbx86 - binary generation
Temp. Object File
Processed Fixups
PEwrite/COFFwrite
Key:
File
Processing Step
Optimized Windows 32-bit EXE, DLL, or OBJ

Figure 3.1 Steps in the binary optimization framework.

function entry points have been examined. The pointers within the headers and tables are adjusted during the translation process to reflect the reordering of functions and sections and the increase in size of the sections due to optimizations and profile probe insertions.

The initial entry point extracted from the file header and the other decode points taken from the export table are pushed onto an entry point stack. Code discovery begins by popping the initial entry point off the stack, and decoding continues through the corresponding basic block. If a call is encountered, the call address is explored, and the return address is pushed on the stack for future decoding. Jump addresses are pushed onto the stack for exploration after the current basic block has been decoded. Once the current stream has been completely decoded, the next decode point is popped off of the stack. Potential callback addresses and jump table offsets are also stored for further exploration. Immediate operands and memory references are also pushed on the stack, so they can be examined for addresses in the text section or tables of text section addresses residing in the data section. The entries in the symbol and base relocation tables are examined as a backup method to ensure that all functions have been decoded.

All instructions are fully decoded into IMPACT's intermediate Mcode representation, a format that allows complete representation of the x86 instruction set, including MMX and 3DNow! instruction set extensions. As instructions are decoded, functions and their corresponding basic blocks are identified and labeled. From this information, program control flow can be determined for use in the optimization phase.

After all functions and their constituent instructions have been converted into Mcode, the optimizer shifts to Phase II and the Lbx86 module to perform optimizations and code rescheduling. The information stored in the Mcode, MemoryMap, and InstrInfo structures allows the programmer to perform any number of generic or machine-specific optimizations on individual instructions, basic blocks, or functions. In addition, since all of the functions have been separated and stored in their own individual objects, the functions can be reordered to capitalize on machine cache size or cache boundaries.

Finally, the IMPACT schedule manager can employ a description of the available machine resources and reschedule the code appropriately.

After the optimizations are complete, Phase III (also in Lbx86) of the binary optimizer takes over to translate the Mcode into a binary object file. Simultaneously, an equivalent assembly file can be output for use in debugging. After the object file has been generated, the PEwrite module produces a new binary executable. PEwrite creates a copy of the original executable and inserts the new text and data sections over the original versions. The other sections, which have not been converted to Mcode or optimized, are simply copied from the original with pointers adjusted to correctly point to locations within the new text and data. A new base relocation table and symbol table are created based upon the updated executable and inserted in place of the original tables, and the other tables and headers are adjusted to reflect new addresses and pointer locations.

CHAPTER 4

BINARY INPUT

The binary optimization framework leverages off of the information inherent in the DLL or EXE itself for code discovery and the ensuing optimizations. Due to the difficulties inherent in decoding mixed 16- and 32-bit code, the system handles strictly 32-bit Windows executables and dynamic-link libraries in the *PE/COFF format.* With the mainstream acceptance of Windows 98 and Windows NT 4.0, the use of 16-bit libraries has become a rarity rather than the standard, and optimizations geared toward 32-bit code have the potential to achieve performance improvement for a widespread application base. From the 32-bit PE file format, the optimizer can extract information regarding the sizes and locations of the various sections, the contents of function import and export tables, the pointers located in the base relocation table, and symbol information, if available.

4.1 PE File Format

Under Microsoft's 32-bit operating systems, executable and object files are organized according to the *Portable Executable* (PE), or *Common Object File Format* (COFF), specification. The PE format is segmented into several headers, tables, and sections that are detailed in [9] and [10]. The binary optimizer relies upon the information encapsulated in the PE to decode the binary file, to identify pointers within and among the various sections, and to differentiate between instruction code bytes and data. The header portion of the PE is made up of an MS-DOS stub program, a file signature, a *COFF header*, and an *optional header.*

The file signature is used to confirm that the file is compiled in valid PE format. The COFF header indicates how many sections are contained in the file and includes both a pointer to the symbol table and the number of entries in that symbol table. The *Characteristics* variable in the file can be used to denote that the file is a DLL rather than an EXE and also records the presence of a base relocation table. Additionally, the PE image file contains an *optional (or PE) header* used by the Windows linker and loader. The optional header is always present in compiled binaries but is not needed for object files. The binary optimizer reads this header information to determine the relative addresses of the initial entry point into the executable, the start of the code section, and the start of the data section.

The optional header also includes data directory entries that provide indices into tables that are mapped in memory for use by the system during execution. Three of those tables are important to the operation of the optimizer: the *export table*, *import table*, and *base relocation table.* While the optimizer requires the base relocation table for normal operation, the export and import tables simply provide additional information. The base relocation table is vital to the operation of the optimizer and is used to determine whether bytes detected are valid pointers into the code or are data mistakenly interpreted as code by the decoder. The export table indicates the names and offsets of functions that are being made available to other applications and is particularly useful when examining DLLs. The import table is used to identify external DLL calls within functions and lists the external function references, with entry point information for each reference.

The PE/COFF header can also contain a pointer to a *symbol table*, an optional table that contains an array of symbol records. The records describe the symbol name, the value of that symbol, the section in which the symbol is located, the type of the symbol, and the number of auxiliary symbols that follow the current record. Immediately after the symbol table lies the *string table* comprised of a series of null-terminated strings pointed to by symbols in the symbol table. Symbols can represent file names, section names, function names, and pieces of code or data. This information can be used by the binary optimizer to assign names to discovered functions and to confirm that all pointers into the code section have been explored.

Following the optional header is the *section table*, consisting of section header entries. Each section header details the name, size, and starting offset of that particular section. For object files, the section header also points to a listing of relocation entries for that section. These relocation entries provide the addresses of the items that must be relocated, indices for the relocated symbols in the symbol table, and the types of relocations that should be completed.

The sections containing the actual code and data of the file follow the headers and are located via pointers and information discussed above in the header tables. The *.edata* section of an image file describes symbols that are exported and can be utilized by other files through dynamic linking. The *.idata* section contains information on symbols being imported from other dynamically linked sources. The *.rsrc* section details the resource data included in the application. The *.reloc* section of a binary image includes a list of all addresses that must be fixed up if the executable cannot be loaded at its preferred address. The *.data* section is comprised of blocks of initialized data bytes used by functions within the file, as well as index tables for indirect jumps and calls. The *.text* section contains the instruction code for the application, and the initial entry point indicates where execution begins.

Although Microsoft's compilers have set the .text and .data standards, the binary optimization framework can also operate on files generated by both Borland and Watcom compilers. Borland refers to the sections as *code* and *data*, and Watcom uses *auto* and *dgroup*. Throughout the remainder of this thesis, these sections will be referred to as the text and data sections.

4.2 Dynamic-Link Libraries

In order to enable a large array of programs to utilize a common set of functions, the 32-bit Windows (Win32) application programming interface (API) relies upon dynamic-link libraries (DLLs) to implement major portions of its functionality. DLLs contain collections of separate functions without the standard Windows startup and support code. Additionally, DLLs allow the programmer to subdivide an application into many portions that can be individually loaded into memory as needed and, thus, minimize use of system resources.

Either standalone executables or other DLLs can call the functions within the DLL, and multiple processes can access the same DLL. In order for outside processes to utilize functions defined within a DLL, they must be defined as EXPORT functions and listed within the DLL's export table. Microsoft Visual C++ prefers the use of *_declspec (dllexport)* before the declarations of functions exported from a DLL and *_declspec (dllimport)* before the corresponding declarations of functions imported from a DLL. The keywords and the corresponding table entries alert the OS and the linker.

DLL object files are compiled and organized in the same PE format as a regular executable. At link time, however, the DLL object files must incorporate information that allows the operating system to differentiate between a DLL and a standard application. Unlike a typical Win32 application that executes in its own linear address space, a DLL should be accessible across multiple address spaces. The system, therefore, loads the DLL into a global heap that is then memory mapped into the relevant application space(s).

When an executable contains calls to functions within a DLL, the DLL file image can be mapped into the caller's address space through *implicit load-time linking* or through *explicit runtime linking*. With implicit linking at load-time, an *LIB* file must be generated to list the DLL functions available to external binaries. The linker then uses the information in the LIB files to embed DLL information into the executable file image. The OS loader will load the referenced DLLs when the main application is loaded and free them from memory only after the calling application terminates. Alternatively, using the LoadLibrary function from the Windows API, a DLL file image can be explicitly retrieved and mapped into the application space during the course of execution and then released immediately after the application has finished referencing it.

After mapping, the text and data sections within the DLL appear as additional text and data sections that can be accessed by the parent application. Under 32-bit Windows, all processes share the DLL code, but each process creates a copy of the DLL's data. The programmer, however, can specify that global variables be shared among processes using the same DLL.

If both a master EXE and its DLLs statically link the same standard libraries, the linker will include separate but identical copies of the library code in both the EXE and DLL. Additionally, it appears that the OS may have difficulty initializing static copies of the standard C library in both the EXE and the DLL, and, thus, standard C functions within the DLL may not execute properly. When using EXEs with DLLs, therefore, it is important not to mix statically and dynamically linked C libraries. DLL functions should be carefully selected not to rely on C library functions, or both the DLL and EXE should utilize the C runtime DLL and, thus, access the same C library functions.

Under 32-bit Windows, a single main entry point is used for loading and unloading the DLL, as opposed to multiple primary entry points under Windows 3.x. Additionally, only the variables through which functions are externally accessed need be exported, rather than the functions declarations themselves, and the applications using the DLL see only generic function numbers and not actual function names.

Win32 employs dynamically linked libraries rather than statically linked libraries for several reasons. For static linking, the linker needs both the object code (.LIB files) for the linked function and the desired location of that function in memory. The accessed function code is integrated into the compiled executable to be run as a single unit. Dynamic linking does not require the library source code to be compiled with the executable and allows library functions to be loaded at various memory locations throughout normal system operation by using PE import tables and by postponing function resolution until program execution.

Since DLLs are separate from the applications that reference them, DLL code can be modified without having to alter and relink the main executable. As a result, optimizations can be performed on common DLLs for a transparent benefit to all applications accessing those DLLS. This is the focus of the thesis and one key advantage of the framework that has been implemented.

CHAPTER 5

PHASE I - DECODING THE BINARY

5.1 PE/DLL/OBJ File Input

In the first phase of the binary optimization framework, the x86toM module calls the decoder to parse the binary input, whether it be an executable, dynamically-linked library, or object file. The PE/COFF file is memory-mapped, and information is extracted from the file header and placed into a *BinaryFileInfo* structure, detailed in Figure A.1 of Appendix A. Symbol data, import/export functions, and other PE header information are collected using the data structures defined in *windows.h* and *winnt.h.* As each section of the binary is examined, its headers are stored in *SectionInfoNode* structures, shown in Figure A.2. As the code section of the binary is disassembled, its instructions and related data bytes are placed into various informational structures for use in conversion and optimization.

5.2 Instruction Information Collection

When an instruction is decoded, the x86toM translator stores its components in an internal instruction information, or *InstrInfo*, structure, shown in Figure A.4. This structure is utilized to ensure accurate conversion of x86 instructions to Mcode instructions, as discussed in Section 5.10. The structure allows the framework to track the instruction opcode, the general type of the instruction, and all of the operands, flags, and descriptive bytes of data associated with that instruction.

The *DecodeOneInstr(DWORD decodeIP, InstrInfo *Instr)* function in the x86toM module parses a single instruction extracted from the bytes in the binary and fills in the fields of the InstrInfo structure according to the algorithm presented in Figure 5.1. The decode function begins with the first byte of the instruction and performs a series of table lookups to determine the various components of the instruction. The tables were compiled from the x86 instruction specification establish by Intel in [1]. The decoder steps through the bytes searching for prefixes, addressing modes, opcode bytes (one- or two-byte opcodes), modR/M and SIB bytes (which contain operands and displacements), and immediates. If the opcode is 0x0F0Fh, then the instruction is an AMD 3DNow! [2] instruction, and the immediate byte contains the actual opcode specifier for the instruction. The decoder also obtains the flags modified by the instruction and advances the file pointer to the next instruction to be decoded.

Figure 5.2 reveals a sample instruction taken from *speedsim.dll*, a dynamically linked library used in conjunction with the IMPACT timing simulator, *speedsim.exe*. The decoder parses this instruction based upon the Intel x86 instruction set specification [1] and determines from the opcode of 83h that it is an ADD instruction. Based on that information, it knows that the following byte represents the register or memory operand of the modR/M byte. Based on the value of C4h, it identifies the ESP register. Then it detects an immediate byte to be added to the contents of the ESP — in this case 04h.

```
DecodeOneInstruction (){
    Look up byte in x86 opcode table
    If byte == prefix
        Set InstrInfo prefix to byte and advance to next byte of file
    If two-byte opcode
        Advance to next byte of file and look in two-byte section of opcode table
    If 3DNow! opcode
        Set amd3d flag to look at immed byte for real opcode
    Set InstrInfo opcode to byte from opcode table and advance to next byte of file
    If opcode indicates presence of modR/M byte
        If modR/M indicates presence of SIB byte{
            Set InstrInfo basereg, indexreg, and scale assigned from SIB table lookup
            Advance to next byte of file
        }
    For each operand byte i specified by modR/M {
        Look up operand byte in table
        Set InstrInfo opertype[i], opersize[i], and regnum[i] to table lookup values and advance
    }
    If displacement bytes present{
        Record number of displacement bytes present
        If DWORD displacement && displacement value is a relocation
            Set InstrInfo displacement to address and set InstrInfo's reloc32MemDisp flag to 1
        Else set InstrInfo displacement to bytes
    }
    If immediate value present{
        If amd3d == 1
            Use immediate value to set InstrInfo opcode to value from 3DNow! lookup table
        Else set InstrInfo immediate field to immediate byte
    Set InstrFlags
    }
}
```

Figure 5.1 Pseudocode algorithm for decoding one instruction in the binary.

5.3 Instruction Flow Decoding

Beginning with the initial entry point specified in the file header, code discovery proceeds using a depth-first traversal of the text section according to the algorithm in

```
1000B73E : 83 C4 04   add ESP, 04h
```

Figure 5.2 Sample decoded instruction from the speedsim.dll binary.

Figure 5.3. The x86toM decoder parses the bytes of the first instruction as discussed in Section 5.2, and, after those bytes are interpreted, the location of the next instruction is ascertained. If the decoded instruction does not alter the control flow, the next instruction is assumed to begin with the next consecutive code byte and the instruction decoding process is repeated in linear order.

Changes in control flow throughout the binary further guide the decoding process and organize the code into basic blocks. A *basic block* is defined as a linear series of instructions with control flow in at the beginning of the block and control flow out at the end of the block. No control flow transfers occur inside the block itself. For a function call, the target address is pushed onto a *decode stack*, and decoding continues with the next consecutive (return) address. A function call does not signify the end of a basic block because control will return to that next consecutive address after the called function has completed execution. For an unconditional branch instruction, the taken address also is pushed onto the decode stack, and decoding resumes with the fall-through path. Since the conditional branch is a control transfer point, the fall-through path represents the start of a new basic block for that function. For an unconditional branch, the target address is subsequently decoded, and a new basic block begins. When a return instruction is decoded or decoding progresses to a previously discovered instruction, the decode stack

```
Push export entry points from export table and initial entry point onto decode stack
DecodeFlowAnalysis(){
    While decode stack is not empty{
        While linear, undiscovered instructions{
            DecodeOneInstruction()
            If instruction is a function call{
                If indirect call through memory
                    Examine memory location for target address
                Push target address onto decode stack
                Decode instruction at next consecutive (return) address
            }
            If instruction is a conditional branch{
                If indirect call through memory
                    Examine memory location for target address
                Push taken address onto decode stack
                Decode instruction at next consecutive (fall-through) address
            }
            Else if instruction is an unconditional branch
                If indirect call through memory
                    Examine memory location for target address
                Decode instruction at target address
            Else if instruction is a return
                Pop address from decode stack and resume decoding at popped address
            Else if instruction has immediate operand
                If immediate is a relocatable address
                        Push relocatable address onto decode stack
            Else continue linearly to decode next consecutive instruction
        }
    }
}
Push base relocation entries and repeat DecodeFlowAnalysis()
Push symbol table entries and repeat DecodeFlowAnalysis()
```

Figure 5.3 Pseudocode algorithm for instruction flow decoding.

is popped, and decoding continues with that address. The Global Memory Map facilitates the identification of previously discovered code and will be discussed in Section 5.4.

```
### Decoding func 1 bb 1 starting at addr 10001000h

10001000 : 55                   push    EBP
10001001 : 8B EC                mov     EBP, ESP
10001003 : 51                   push    ECX
10001004 : C7 45 FC 01 00 00 00 mov     dword ptr [EBP-04h], 00000001h
1000100B : 83 7D 0C 00          cmp     dword ptr [EBP+0Ch], 00h
1000100F : 75 10                jnz     10001021

### Decoding func 1 bb 2 starting at addr 10001011h

10001011 : 83 3D 60 59 02 10 00 cmp     dword ptr [10025960h], 00h
10001018 : 75 07                jnz     10001021

### Decoding func 1 bb 3 starting at addr 1000101Ah

1000101A : 33 C0                xor     EAX, EAX, EAX
1000101C : E9 CC 00 00 00       jmp     100010ED

### Decoding func 1 bb 4 starting at addr 100010EDh

100010ED : 8B E5                mov     ESP, EBP
100010EF : 5D                   pop     EBP
100010F0 : C2 0C 00             ret     000Ch
### exit pt of fn 1 bb 4.

### Decoding func 1 bb 5 starting at addr 10001021h

10001021 : 83 7D 0C 01          cmp     dword ptr [EBP+0Ch], 01h
...
```

Figure 5.4 Sample instruction flow from the decoding of speedsim.dll.

Figure 5.4 illustrates the flow decoding process. Instructions are disassembled linearly through a basic block until a control flow altering instruction is reached. In this case, the *jnz* to a direct address causes the taken address of 10001021h to be pushed on the decode stack and the fall-through path at 10001011h to be explored. When the *jmp* to

Table 5.1 Types of decode/entry points stored in EntryPt structure.

EXPORT_TGT	CALL_TGT	JCC_T_TGT	JCC_F_TGT
JMP_TGT	JMP_RM_TGT	CALLBACK_TGT	CALL_SPLIT_TGT
RELOC_TGT	EXCPT_EXPR	EXCPT_BODY	EXCPT_HDLR

100010EDh is encountered, the target address is explored until the *ret* signals the end of the block and the current decode path. The address on the top of the decode stack, 10001021h, is popped, and decoding continues.

5.3.1 Entry point identification

In order to facilitate the control flow decoding of the target binary, the x86toM module employs a decode stack to store fall-through addresses, possible callback function addresses, exported entry points, and the initial entry point for the binary. All of these addresses, or decode points, are stored in *EntryPt* structures defined in Figure A.3. The EntryPt structure identifies the type of the decode point (jump target, call target, exception handling code, relocatable address, etc), as shown in Table 5.1. Additionally, the EntryPt stores the basic block to which it belongs, the address to be decoded, and the identification number of its parent function. The structure also tracks the address of the instruction that generated the control flow change.

The x86toM module must have a starting point from which to decode the binary and statically explore the various functions. First, the entry points of functions being exported from the binary are extracted from the export table and pushed onto the decode stack. The initial entry point of the binary is identified from the file header and pushed onto the top of the decode stack. The decoder begins by popping the decode stack and analyzing this first instruction. After the first instruction is disassembled and stored as an InstrInfo, the decoder determines where the next instruction must follow. As discovery progresses, additional decode points are stored as EntryPts and pushed onto the decode stack. Once x86toM has exhausted the current linear decode path, an EntryPt is popped off of the stack for further decoding. As a fallback measure, all undiscovered entries in the base relocation table and the symbol table from the original binary may be converted to EntryPts and pushed onto the decode stack to ensure that all of the code is discovered.

5.3.2 Basic block formation

As the static instruction stream is decoded, the instructions are grouped into basic blocks (refer back to the algorithm in Figure 5.3) to aid in function reconstruction and dataflow analysis. These basic blocks are stored in *BlockInfo* structures, shown in Figure A.5. Although superblock [11] capability and several corresponding optimizations are currently being added to the framework, the existing framework and rescheduler rely exclusively on basic block boundaries to perform optimizations and reorder instructions. Thus, it is vital to reliably maintain detailed records of the basic blocks found in the binary and to clearly define the relationships between those blocks.

The example of Figure 5.4, taken from speedsim.dll, illustrates a typical basic block analyzed by the decoder. Decoding progresses linearly until the conditional branch (*jnz*) is reached. This signals a control flow transfer and the start of another basic block within the function.

During the course of decoding the binary, x86toM may discover a branch or even a call into the middle of a previously decoded block. This new piece of control flow information requires that the target block be split along the target address to form two separate blocks. The second block would begin with the target address of the branch or call. At other times, the decoder may discover that it is translating invalid instructions or is searching through data in the text section. At that point, the current block and references to it must be deleted. Such deletions may reverse the effects of a prior block split, so the decoder has the capability to merge blocks as well. These basic block manipulations are facilitated by the presence of a *global memory map* of all basic blocks and jump/call tables, which is discussed in Section 5.4.

5.4 Global Memory Map Construction

As basic blocks are translated and organized, they are collected into a *global memory map*. This memory map is used to ensure that decoded instruction addresses lie within the boundaries of the text section and, therefore, are likely to be valid instructions rather than data bytes. The memory map also keeps track of all the basic blocks that have been decoded in the binary, along with their constituent instructions, and facilitates the insertion and removal of basic blocks. Additionally, the memory map can be used to aid in the expansion, reduction, or splitting of existing basic blocks due to further code discovery. In addition to storing blocks of regular code in the global memory map, jump and call tables are also included. The memory map can be used to detect repeated references to these tables and allow enlargement of the tables.

The memory map may also be used for debugging and analysis to identify areas of code that were not explored by the static decoder. A possible extension of the decoding framework would be to explore those unknown regions, even if the decoder cannot find a connection to the discovered code. This approach has been implemented but is not actually employed due to the extremely high risk of encountering data blocks in the middle of the text section and mistakenly interpreting the bytes as code. See Figure A.6 for the composition of the *MemoryMap* object.

5.5 Function Formation

Groups of basic blocks are also analyzed to determine function boundaries, and all of the blocks comprising a function are assigned to a *FunctionNode*, shown in Figure A.7. The understanding of function boundaries enables the optimizer to rearrange the order of functions in the output binary. Furthermore, delineation of functions allows the optimizer to insert probe code used to extract function profile weights upon execution of the new binary. As functions are identified, they are placed in a *FunctionTable* that enables rapid querying during analysis, optimization, and profiling. If symbol information is present in the binary, function names can be associated with the FunctionNode structures.

If the decoding of one function falls through into another function, the system assumes that the two functions are actually a single function with additional entry points within the function body. The decoder integrates the two functions into a single function in the FunctionTable to ensure proper conversion to Mcode.

The FunctionNodes also track all of the entry points and return points of the function, as well as a list of the addresses from where the function is called. As code discovery alters the composition of a function's basic blocks, the entry and exit point lists are updated accordingly. The FunctionNode not only maintains a list of constituent blocks but also stores jump/call tables that are used with the function. The block and table information is used by x86toM to establish Mcode control flow arcs within and among the functions. The lists of entry and exit points are also used during Mcode conversion to ensure that multiple entry point functions are translated correctly.

5.6 PE Table Usage

The exact structure and code layout of the target binary are hidden from the binary optimization framework due to lack of original source code. Therefore, the techniques used for code discovery are based upon heuristics that rely heavily upon the PE and x86 instruction specifications and information inherent in the PE file. In order to both verify and augment the code discovery performed by the decoder, the information found in various tables associated with the PE file headers is utilized by x86toM.

5.6.1 Base relocation table

The PE base relocation table is a vital component of the x86toM decoder. In order to properly execute almost all of the code discovery heuristics, the system must know that the address being investigated is in fact a valid pointer to a function entry point or other point of control transfer. All Win32 binaries have a preferred load address listed in the file header. The loader always attempts to map the binary image into memory at that preferred address. However, if another binary is already mapped at that address in the program space, the loader maps the image at a different address and must adjust absolute pointers within the binary to reflect the new positioning. The base relocation table lists all such addresses so that the program loader can successfully relocate the executable or dynamically linked library in memory and fix up up the pointers within the file. The jump/call table and callback heuristics discussed in Sections 5.7 and 5.8, as well as the normal decoding flow for branches and function calls, all reference the list of relocations to ensure that they are pursuing valid code targets.

5.6.2 Symbol table

If a symbol table is present in the original binary, the information contained in the symbol table and the corresponding string table can be used to identify the functions that are being decoded and can provide more detailed output and debugging information to the user. More importantly, the function addresses in the symbol table can be used as additional decode points in a cleanup measure to explore functions that were not already discovered in the initial depth-first search of the binary. The addresses are pushed onto the decode stack, and control returns to the x86toM decoder. Additionally, the symbol information can be used to allow the user to selectively profile specific functions by name.

5.6.3 Import and export tables

The import table associated with the original binary is employed by x86toM to identify calls to external functions in other DLLs or supplementary EXEs. When the decoder encounters a call to an address outside of the known code space, it refers to the list of imports to confirm that the operation is a valid function call and to associate a DLL and function name with that call. The decoder also analyzes the binary's export table to identify functions that are made available to other executables. The addresses of these functions are pushed onto the decode stack to ensure that they are decoded by x86toM.

5.7 Jump and Call Table Discovery

When the decoder encounters an indirect branch or call, the target may point to a table of jump or call targets that is accessed through a base offset plus some index. Using the algorithm laid out in Figure 5.5, the x86toM decoder uses the base offset to iterate through the table entries and to extract all of the valid target addresses. If these addresses lie within the bounds of the text section and are listed as relocatable pointers, then they are pushed onto the decode stack for further code discovery.

However, many of these tables have a more complex instantiation and require further exploration. While most tables begin at the base offset and are positively indexed from that address, some tables end at the given base offset and are negatively indexed. The decoder heuristically examines addresses both before and after the base offset to locate all valid pointers to push onto the decode stack. Often, the call or jump tables are implemented in a two-level format, with the first address pointing to a secondary table of valid function entry points. If the decoder cannot find any valid entries in either the positive or negative direction of the first-level table, it treats the entries as pointers to

```
Extract displacement from instruction and convert to mapped address
Check for positively-indexed jump table{
    While assuming possible targets exist{
        If table entry points to a valid relocatable address
            Push target address onto decode stack
        Else if table entry points to an out-of-bounds address
            Assume possible targets no longer exist
        Advance to next table entry
    }
}
If no valid entries found in the positive direction{
    While assuming possible targets exist{
        Move back one DWORD in table
        If table entry points to a valid relocatable address
            Push target address onto decode stack
        Else if table entry points to an out-of-bounds address
            Assume possible targets no longer exist
        Move back in table to previous entry
    }
}
If no valid entries found in either the positive or negative directions
    Assume a two-level jump table
```

Figure 5.5 Pseudocode algorithm for analyzing jump and call tables.

a second-level table and performs a similar validity analysis on the second-level table entries.

```
### Decoding func 67 bb 19 starting at addr 56688E29h
56688E29 : 33 C9                xor     ECX, ECX, ECX
56688E2B : 8A 88 F0 94 68 56    mov     CL, byte ptr [EAX+566894F0h]
56688E31 : FF 24 8D AC 94 68 56 jmp     dword ptr [4*ECX+566894ACh]
### check for jump table

Raw Data Dump
  566894A0: 04 04 04 04 04 04 04 04 04 04 03 90 38 8E 68 56
  566894B0: 07 90 68 56 07 90 68 56 1C 8F 68 56 10 91 68 56


Contains Addresses:
  56688E38
  56689007
  56688F1C
  56689110

### Decoding func 67 bb 132 starting at addr 56688E38h
56688E38 : 8B 44 24 10          mov     EAX, dword ptr [ESP+10h]
56688E3C : A9 30 08 00 00       test    EAX, 00000830h
56688E41 : 75 07                jnz     56688E4A
```

Figure 5.6 Sample one-level jump table from Microsoft DirectX's d3dim.dll binary.

For an example illustrating the presence of indirect jump tables, refer to Figure 5.6. The decoder examines the information at the offset for the indirect jump, and, by searching through the data bytes, discovers table of four jump target addresses. Upon further examination, those target addresses all reveal valid basic blocks within the function. The block at the first jump table target is shown simply as a validating example.

5.8 Callback Detection

If a nonbranch or noncall instruction, such as a *mov* or a *push*, has a 32-bit immediate or memory displacement value, it is possible that the value is a pointer to a function entry point and is being passed to the operating system or server application for later callback execution. To ensure that these functions are detected, the decoder relies upon the heuristic presented in Figure 5.7 to examine these instructions with 32-bit immediate or memory displacement values. The decoder checks if the value falls within the range of the text section and also checks if the value is a relocatable address in the base relocation table. Alternatively, the pointer could reference a table of callback addresses. The routine searches through the table for callbacks until a valid nonzero, nonpointer is found. These valid addresses are pushed onto the decode stack for code discovery.

```
If instruction has an immediate value{
    If value at immediate target address is a relocatable
        Look for code at address stored at target
    Else{
        If value at target is in text section
            Push target onto decode stack
        Else look for table of callbacks, ignoring zero entries
    }
}
```

Figure 5.7 Pseudocode algorithm for analyzing callbacks.

From Figure 5.8, it can be seen that what appears to be a valid address is pushed onto the stack. When investigated by the decoder, it becomes clear that this is an entry

```
### Decoding func 43 bb 1 starting at addr 5666C744h
...
5666C74E : 68 48 61 68 56       push    56686148h
...

### Decoding func 45 bb 1 starting at addr 56686148h
56686148 : 55                   push    EBP
56686149 : 8B EC                mov     EBP, ESP
5668614B : 83 EC 08             sub     ESP, 08h
5668614E : 53                   push    EBX
...
```

Figure 5.8 Sample callback from Microsoft DirectX's d3dim.dll binary.

point for a legitimate function in the binary. This function may not have been decoded without the exploration of the callback address by x86toM.

5.9 Exception Handler Identification

Some Win32 programs contain special code called exception handlers inserted to correct an exception or error encountered during program execution. An exception could be an operation such as an invalid memory access, INT 3, or division by 0. If an exception is detected, the exception handling code block is executed to identify the exception and to attempt to remedy the problem in order to return to normal program execution.

Under Win32, each process thread contains a *thread information block* (TIB) which stores important values and settings for the thread. Included at offset 0 in the TIB is the head of the structured exception handling chain for that thread [12]. Win32 uses the FS segment register to point to the TIB for the currently executing thread, so a reference to

FS:[0] is a reference to the head of the exception handler. The binary optimizer heuristically searches for references to the FS segment register to identify exception handlers in the code. The binary optimizer will not perform dataflow analysis and the corresponding optimizations on blocks with exception handling mechanisms because not enough information is known about their structure and function and the resulting dependences.

```
...
5666C753 : 64 A1 00 00 00 00    mov     EAX, dword ptr FS:[00000000h]
5666C759 : 50                   push    EAX
5666C75A : 64 89 25 00 00 00 00 mov     dword ptr FS:[00000000h], ESP
...
```

Figure 5.9 Suspected exception handling code from DirectX's d3dim.dll binary.

In Figure 5.9, the manipulation of offset 0 of the FS segment register indicates interaction with the exception handler. The decoder will heuristically assume that exception handling mechanisms are present and indicate that the basic block is an exception handling block. The Lbx86 optimizer will not perform optimizations on this block. Exception handler code tends to be deliberately organized to recover from an unforeseen event, and any sort of optimizations may unintentionally render the exception handler inoperable.

5.10 Mcode Conversion

After the binary has been decoded, the instruction information contained in each InstrInfo is converted into the IMPACT intermediate Mcode representation for use in code scheduling and optimization. A detailed explanation of the Mcode format and the conversion from x86 instruction into Mcode can be found in [3], [4], and [13], but a brief overview is included here for thoroughness.

Mcode is a machine-specific version of IMPACT's generic Lcode [14] representation, and the version of Mcode used in the binary reoptimization framework is geared toward the x86 architecture. It provides a method of representing all instructions in the x86 instruction set, as well as data and control flow information. The Mcode format accommodates a variable number of source and destination fields, a processor-specific opcode mnemonic, a general opcode mnemonic, and an attribute list that provides additional information to the optimizer. For the purposes of the binary optimization framework with the x86 instruction set, a maximum of four destination fields and seven source fields allows all register accesses and instructions which read or write many operands to be explicitly detailed in Mcode.

In Mcode, each instruction is assigned a numeric processor-specific opcode, known as the *proc_opc* or *popc*, and this value is used to distinguish between different types of operations and resource usage. Since different optimizations may require different degrees of detail regarding the instructions (e.g., identify only register-register addition instructions or identify all addition instructions), the proc_opc is separated into two fields: the general opcode or *genopc* and the *variant*. The genopc indicates the processor-specific mnemonic, and the variant furnishes additional information about the specific form of the instruction (operand types).

When the x86 instructions are converted to Mcode through x86toM, the Mcode includes only the genopc portion of the proc_opc, and the variant is zero. The Lbx86 preprocessor examines the genopc for an instruction and calls the appropriate variant annotation function to determine the variant from the instruction operands. Thus, the IMPACT code scheduler will have complete information about the instruction.

Figure 5.10 provides examples of the general Mcode formats for various types of instructions. The Mcode format begins with an operation indicator, the identification number assigned to the operation by the scheduler, and the functional opcode mnemonic (such as *add*) for that operation. The first set of square brackets specifies the destination operand information and flags to be modified. The second set of square brackets delineates the source operands and flags to be read by the operations. The set of angle brackets following the destination and source information contains the instruction attributes, such as the genopc and the popc.

```
 (op n opc [(reg:intel_dest)(reg:intel_src)(reg/df)(flags)]
[(reg/addr:intel_dest)(reg/addr/imm:intel_src)(reg/flags)(addr_base)
 (addr_index)(addr_scale)(addr_offset)] (popc ))
```

(a) General instructions, such as an ADD.

```
 (op n opc [(edi/reg:intel_dest)(esi/reg:intel_src)(ecx)(flags)]
       [(edi/reg:intel_dest)(esi/reg:intel_src)(ecx)() (df)(flags)] (popc ))
```

(b) String instructions.

```
 (op n ld_i [(reg)()()()] [()(addr)()(b)(i)(s)(o)] (popc MOV_TO_REG))
 (op n st_i [()()()()] [(addr)(reg/imm)()(b)(i)(s)(o)] (popc MOV_TO_MEM))
```

(c) Load and store instructions.

Figure 5.10 Some sample Mcode formats.

CHAPTER 6

PHASE II - OPTIMIZATION

During *Lbx86 Phase II*, the binary optimization framework performs various instruction transformations and instruction and function reordering using the Mcode structures which have been assembled in x86toM. Details of those optimizations can be found in [3] and [4]. This chapter will focus on the rescheduling that occurs following the optimizations discussed by Thiems and Merten.

6.1 Schedule Manager

The IMPACT compiler's *scheduler manager* (SM) [15] supports code analysis for tranformation and scheduling. It enables the system to track operand usage via dependence arcs. SM examines one basic block at a time but employs IMPACT's dataflow analysis tools to track which operands are live-in and live-out of the current basic block. The Mcode represents reads and writes of instruction operands and encodes condition code flags as registers to allow accurate dependence and dataflow analysis by SM.

After dataflow analysis has been completed, SM generates dependence arcs to branches for operands that are live-out along that branch's taken path. Unconditional dummy jumps are inserted by the Lbx86 preprocessor at the end of each basic block with a fall-through path. This step ensures that complete live-out information is present for all transformations. The Lbx86 postprocessor removes these dummy jumps.

6.2 Machine Descriptions

The resources of the platform on which an application is executed play a vital role in the speed and efficiency of that application. Furthermore, one cannot take advantage of all available scheduling and optimization opportunities without considering the configuration of the target machine. For this reason, the IMPACT binary optimization framework employs SM in combination with a *machine description* file (MDES). The MDES is written in the *Machine Description Language* [16] and represents the resources, latencies, and dependences of a processor and its instruction set in a form that the scheduler can comprehend and utilize. Programmers can construct the MDES in a high-level, easy-to-use language that is translated into a low-level format that allows rapid access by the scheduler manager.

The availability of accurate data and control flow information, along with complete instruction identification, provides a framework with multiple optimization opportunities. An important optimization opportunity is presented in the form of code rescheduling based upon a machine description. Machine descriptions can be altered for different microarchitectural implementations of the same instruction set to facilitate precise machine-specific rescheduling. The machine description must classify each possible *proc_opc* value based upon its characteristics. It must also model the use of different machine resources by each instruction, and it must model the interaction between the instructions. Using an accurate machine description, SM can avoid resource conflicts and data dependence interlocks when scheduling the code.

6.2.1 Anatomy of an MDES

As shown in Figure B.1, an MDES is divided into several different sections describing different characteristics of the target processor. A *Format* section describes the operands that may be used with each processor operation. A *Resource Usage* section reveals which processor resources are occupied during the execution of a particular operation. The *Latency* section provides the information necessary to calculate dependence distances between operations. The *Operations* section combines format, resource usage, and latency information into descriptions of valid operations. The *Registers* section lists register overlaps and helps to facilitate register allocation by the compiler. A final *Compiler-specific* section allows the programmer to include any additional configuration information that the target compiler can utilize.

Reservation tables within the machine description detail how and when various operations utilize processor resources during execution. Often, the resources listed in the reservation tables correspond to instantiations of the target processor's scheduling rules. By defining resource usages, the machine description can aid the instruction scheduler in effectively ordering program operations. For example, if two instructions utilize two separate processor resources, then they can be paired and scheduled simultaneously by the processor. Without the machine description, the scheduler would not know how to order the code effectively.

It is desirable for the scheduler to interact with the machine description as quickly as possible in order to minimize the impact on optimization and overall installation time. While traditionally the reservation tables employed in the machine description utilize an *OR-tree* internal representation, an alternative representation may be made using *AND/OR-trees* [17]. By more compactly representing the processor's resource constraints, the MDES size can be reduced, and the scheduler can more efficiently parse the description. In the OR-tree representation, the scheduler simply checks if the resources for any execution options are available and subsequently schedules the operation. With few resource constraints, this method works quite well, but the scheduler cannot use information about why an option was not available. If an option is unavailable due to the prior use of a particular resource, many other options will be similarly unavailable. The new representation contains an AND-tree structure of OR-trees and allows a rapid resource constraint check algorithm to identify available resources without unnecessary overhead.

6.2.2 AMD K6-2 MDES

A machine description was constructed for this thesis to model the resource constraints of the AMD K6-2 processor and was written based upon the K6-2 optimization guide [18]. While some key resources of the K6-2 processor have been modelled in the current MDES,

Table 6.1 Types of instruction decode categories in the AMD K6-2 processor.

Decode Type	Bytes	RISC86 Ops	Example
short	max. 7	max. 2	ADD mreg8,reg8
long	max. 11	max. 4	ADD mem8,reg8
vector	complex	long sequences	ADC mem16/32,imm8

many other configurations remain to be simulated. Rescheduling results can vary greatly based upon the degree of constraint in the machine description.

The K6-2 (and the K6) decode x86 instructions into fixed-length RISC86 operations (Rops) and process the x86 instructions in three different categories: short decode instructions, long decode instructions, and vector decode instructions, as shown in Table 6.1. Short decodes are x86 instructions that are no longer than 7 bytes and that translate into no more than two Rops each. Long decodes are x86 instructions that are no more than 11 bytes in length and that translate into no more than four Rops apiece. The K6-2 includes two short decoder units, so two short decode instructions can be decoded simultaneously. Long decodes require both short decoders, and vector decodes need both a vector decoder unit and an on-chip ROM. Since the instruction decode was deemed to be a potential scheduling bottleneck, the AMD K6-2 machine description reported in this thesis models the parallel short decoders. The excerpts from the *Resource* and *Resource_Usage* sections shown in Figure 6.1 illustrate the instantiation of the decoder resources in the K6-2 MDES.

The RISC86 instruction set is comprised of eight operation groups, and the K6-2 has a separate execution unit to process each group. The various groups and corresponding

```
SECTION Resource
{
  slot0 (slot(0)); /* scheduling slots */
  slot1 (slot(1));
  decode0 ();          /* 1st short decoder */
  decode1 ();          /* 2nd short decoder */
  load_port ();        /* single load unit */
  store_port ();       /* single store unit */
}

SECTION Resource_Usage
{
  RU_slot0_t0_0  (use(slot0) time(0));
  RU_slot1_t0_0 (use(slot1) time(0));
  RU_decode0_t0_0 (use(decode0) time(0));
  RU_decode1_t0_0 (use(decode1) time(0));
  RU_load_port_t1_1 (use(load_port) time(1));
  RU_store_port_t1_1 (use(store_port) time(1));
}
```

Figure 6.1 Excerpts from Resource and Resource_Usage sections of K6-2 MDES.

execution resources are listed in Table 6.2. For evaluation purposes, the K6-2 MDES in this thesis also constrains scheduling with one load unit and one store unit. The *time(1)* shown in Figure 6.1 alerts the scheduler that the load or store instruction uses the load or store unit in cycle 1 after the decode is completed in cycle 0. The order of resource usage is also listed in the *Reservation_Table* entry in Figure 6.2.

Using the decoder and one load/one store unit information, the scheduler manager can hopefully pair applicable instructions and minimize stalls of the decode pipeline. The full version of the AMD K6-2 MDES is included in Appendix B. Further resource

Table 6.2 RISC86 operation groups and corresponding K6-2 execution units.

Operation Group	Execution Unit
Memory load ops	load unit
Load immediate	instruction control unit
Memory store ops	store unit
Integer reg ops	alu or alux
MMX reg ops	multimedia execution unit
3DNow! reg ops	AMD-3D execution unit
Floating-point reg ops	floating point unit
Branch condition eval	branch logic

constraints can be added to the machine description which may either improve or degrade performance, but those experiments remain future directions of the research.

```
SECTION Table_Option
{
  two_issue (one_of(RU_slot0_t0_0 RU_slot1_t0_0));
  short_decode (one_of(RU_decode0_t0_0 RU_decode1_t0_0));
}

SECTION Reservation_Table
{
  RL_ALL  (use(two_issue));
  RL_Load  (use(two_issue RU_load_port_t1_1));
  RL_Store  (use(two_issue RU_store_port_t1_1));
  RL_Short_Dec  (use(two_issue short_decode));
  RL_Long_Dec  (use(two_issue RU_decode0_t0_0 RU_decode1_t0_0));
  RL_Short_Load  (use(two_issue short_decode RU_load_port_t1_1));
  RL_Long_Load
(use(two_issue RU_decode0_t0_0 RU_decode1_t0_0 RU_load_port_t1_1));
  RL_Short_Store (use(two_issue short_decode RU_store_port_t1_1));
  RL_Long_Store
(use(two_issue RU_decode0_t0_0 RU_decode1_t0_0 Ru_store_port_t1_1));
}
```

Figure 6.2 Excerpts from Table_Option and Reservation_Table of K6-2 MDES.

CHAPTER 7

PHASE III - BINARY GENERATION

The process of binary generation was developed by Michael Thiems, and a detailed discussion can be found in [3]. Only a brief overview will be presented here. After the Mcode has been optimized by Lbx86 in Phase II, Lbx86 produces machine code output for the text and data sections that have been translated and optimized in Phase III. The output is generated as a temporary COFF object file to be reintegrated with the other sections in the final phase of the process. As a tool for debugging, assembly language output is also generated. Additionally, the existing fixup file is modified to correct symbol name changes that occurred during Mcode conversion and optimization.

First, the Phase III preprocessor removes all of the extraneous operations that were inserted during Phases I and II to facilitate optimization and scheduling. In Phase III, Lbx86 utilizes a database created from the IMPACT machine description language facility. The database contains one entry for each possible value of the proc_opc, and each entry has a direct correspondence to a member of the x86 instruction set, complete with MMX and 3DNow! extensions. The database entries are translated in advance from high-level textual form into a low-level format much like the MDES discussed in Chapter 6.

When the machine code is generated, the system must ensure that all address references are resolved correctly. While most references can be handled through COFF relocation entries to be completed at link time, some relative addresses within the code section must be resolved during code generation. The binary generator accomplishes this resolution in a series of processing passes over the program. First, instruction lengths and spaces between functions, tables, etc., are determined. Then, further fine-tuning passes are performed to achieve near-optimal encoding based upon the various sizes available for encoding the relative offset.

A final pass over all the Mcode generates the binary machine code for the optimized binary and its data. The relocation entries from the text and data sections are also written to the object file, along with the COFF symbol and string tables. Simultaneously, complete assembly code is generated to aid in debugging of the optimized binary executable. Phase III updates some names in the fixup file, which are used to integrate the optimized code and data with the remaining untranslated sections in PEwrite.

CHAPTER 8

RECONSTRUCTING THE BINARY

The final phase in the binary reoptimization process is the reconstruction and writing of the complete binary executable or dynamic-link library. The PEwrite module combines the original executable with the optimized temporary COFF object file, the processed fixup file, and the updated PE import, export, symbol, and base relocation tables. This process simulates the work of the conventional linker to generate the optimized binary. Since statically linked library code and system startup code reside in the optimized object file, no special linking is necessary, and PEwrite can perform a rapid relinking of the binary components.

8.1 Merge Modifications with the Original

In order to construct and link together the new optimized binary, PEwrite begins by copying the sections and headers of the original binary executable into the new binary to preserve the file and section structure and to retain the sections that were not optimized by the framework. The optimized text and data sections are extracted from the new temporary COFF object file generated by Lbx86 and inserted in the new binary in place of the original text and data sections. Additionally, the file headers must be altered to reflect changes in section locations and sizes. PEwrite must ensure that proper alignment is maintained through section or table placement or via padding of the affected sections to artificially round up to the desired file or section alignment.

8.2 Address Fixups

Since functions have been re-ordered and file and section sizes and alignments have been altered, pointers to old locations within the binary must be adjusted to reflect the new values. The optimized temporary COFF object file contains its own symbol and COFF relocation tables. Since the optimized text and data sections from the object file are copied directly into the new binary, the symbol and relocation references must simply be adjusted for the new starting offsets of the text and data sections. However, the pointers in the untranslated sections that reference the text and data must be corrected using the information found in the fixup file produced during the translation and optimization of the binary.

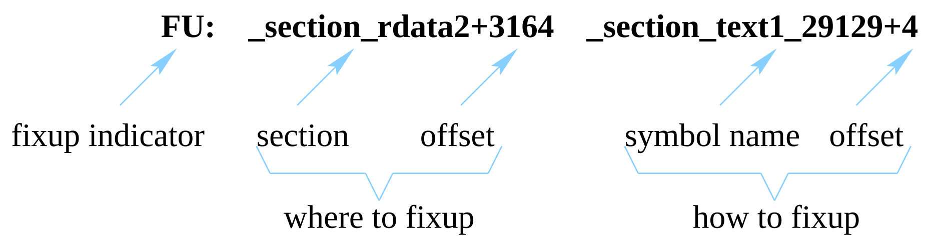

Figure 8.1 Sample fixup file entry from 129.compress.

The fixup file contains relocatable addresses from sections outside the text and data and represents those relocations in the form shown in Figure 8.1. PEwrite parses the fixup file according to the method presented in Figure 8.2 and first searches for each *FU* keyword signifying a fixup entry. Each fixup entry begins with *_section_* followed by a section name indicating in which section the reference can be found. The number after the + is the offset that must be added to the base virtual address of the section to find the pointer reference. Once PEwrite has determined where to fix up the address, the module must decide how to fix up the pointer. The final portion of the fixup entry lists the symbol name of the address to which the reference is pointing. A symbol table lookup reveals the symbol address, and the value following the + tells PEwrite how to fix up the pointer reference. The corrected reference is then written to the new binary.

```
Open pipe to processed fixup file
Process fixups (){
    For each FU entry in file{
        Identify section where reference is found
        Determine offset into that section
        Lookup up reference in symbol table
        Write fixed up pointer reference to new binary
    }
}
```

Figure 8.2 Pseudocode algorithm for processing fixups outside the text and data.

8.3 PE Table Adjustments

Just as pointers within the file sections must be updated as a result of the effects of the translation, optimization, and reordering, the pointers to and the function addresses within many of the PE tables must be adjusted to ensure the correct operation of the new binary.

8.3.1 Import table fixup

Each external function call in a binary references an entry in the import table to determine the export ordinal of the function and the DLL which contains that function. Since the identification of the DLL and its constituent function has not changed during the reoptimization process, the pointer from the file header to import table simply must be updated to record its new position in the output binary. Additionally, the pointers to the import address table and import directory table arrays within the import table must be correctly represented in the output binary.

8.3.2 Export table fixup

The export table is used by the OS and the linker to identify and link functions within the binary to external binaries calling those functions. Due to function reordering and the possible shift of section base addresses, the export table must be rebuilt so that the exported functions and variables can be located by other binaries. In x86toM, the export table entries were added to the fixup file. PEwrite searches the fixup file for entries labeled *EX* and processes those values as the updated components of the new export table. Two sample export fixup entries are shown in Figure 8.3.

First, the existing export table is copied to the new binary, and pointers from the header to the table and its *Name* and *Address* arrays are updated appropriately. Second,

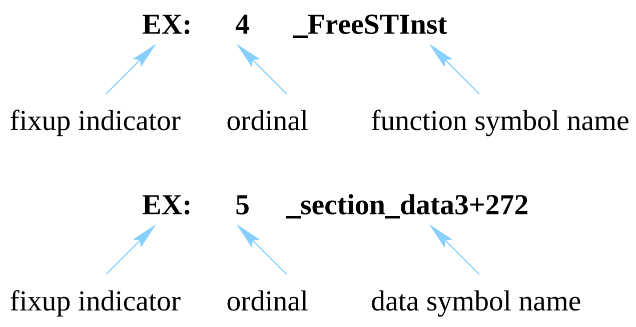

Figure 8.3 Sample export fixup file entry from speedsim.dll.

PEwrite examines each *EX* entry which consists of an export ordinal and a symbol name corresponding to an exported function or variable. The symbol must be looked up in the new symbol table, and the new address is substituted for the original address in the binary's export table.

8.3.3 Symbol table fixup

Realignment and function reordering also mandate the adjustment of the symbol table, as described in Figure 8.4. The new symbol table is formed by updating symbols from the original binary file and the temporary COFF object file. The symbols from the text and data of the COFF object file can be copied from the COFF's symbol table, updated to reflect the offsets of the optimized binary, and inserted in the new symbol table. Additionally, symbols from the other file sections are examined and integrated into the new symbol table, and external data symbols from the original table are updated and added to the new symbol table. After all symbols have been added, a sorting routine ensures proper ordering of the table. An updated string table is generated to describe

```
Open original binary symbol table
Open temporary COFF object file symbol table
Initialize symbol table structure for new binary
Process symbols (){
    For each section in new binary{
        Search original binary symbol table for symbols in section
        Look up old symbol name in new table
        If not in new table
            Add updated symbol to new table
    }
    Integrate undefined symbols
    For each external data symbol in original table{
        If not in new table
            Add update symbol to new table
    }
    Sort symbols
    Generate updated string table for symbol entries
    Write new sorted symbol table and associated string table to new binary
}
```

Figure 8.4 Pseudocode algorithm for new symbol table construction.

the symbols, and the new symbol and string tables are written to the new optimized binary.

8.3.4 Base relocation table fixup

The new base relocation table can be produced based upon the COFF relocation information contained in the optimized object file. The PE base relocation table is a simplified version of the COFF relocation table and can be obtained by pairing down the COFF relocation to include an offset and a relocation type. The information is parsed and sorted by PEwrite and written back as PE format base relocation table blocks. The blocks are delineated by a starting relative virtual page address and a block size, and the fixups for that page follow. Each fixup entry is represented by a WORD which combines the *Type* field of the fixup and the *Offset* field of that fixup within the page. During the generation of the new base relocation table, fixup entries are stored in *BaseRelocEntry* structures shown in Figure A.8.

Additional relocations that are not included in the text and data sections of the optimized object file can be found by parsing both the *FU* and *EX* entries in the processed fixup file maintained by the binary optimization framework. PEwrite converts these fixup entries into additional base relocations and organizes all of the relocations into blocks based on virtual page address. Figure 8.5 illustrates the general algorithm used to generate the new base relocation table.

```
Construct New Base Relocation Table (){
    For each section in temporary COFF object file{
        Determine number of relocations in section
        For each relocation in the section{
            Extract Virtual Address of relocation
            Extract Type of relocation
            Merge values to construct the combined TypeOffset field for entry
            Mask out the Relative Virtual Address of the block in which the fixup resides
        }
    }
    Open fixup file
    For each FU: or EX: entry{
        Locate section and offset into section for the relocation
        Set Virtual Address equal to the updated offset of the relocation
        Set Type of relocation
        Merge values to construct the combined TypeOffset field for entry
        Mask out the Relative Virtual Address of the block in which the fixup resides
    }
    Sort relocations according to Relative Virtual Address (RVA)
    All relocations with the same RVA belong in one fixup block
    For each fixup block{
        Write fixup block RVA to new base relocation table in optimized binary
        Calculate size of block based upon number of entries
        Write size of block to new base relocation table
        For each fixup entry in block{
            Write TypeOffset field to new base relocation table
        }
        If block size is an odd number
            Pad with NULL entry
    }
}
```

Figure 8.5 Pseudocode algorithm for new base relocation table construction.

CHAPTER 9

EXPERIMENTAL RESULTS

The install-time binary optimization framework has been implemented by members of the IMPACT research group to validate the static approach to binary code discovery, optimization, and writeback that has been discussed in this thesis, as well as in the theses by Thiems [3] and Merten [4]. The project leveraged off of the easily manipulated Mcode format already in use with the IMPACT ILP compiler [19]. The Mcode had to be slightly modified to take into account the complete x86 instruction set and all of its variants. The framework runs under both Windows NT and Windows 95/98 and can translate most 32-bit Windows x86 console binaries with base relocation table information. The system can optimize executables or dynamic-link libraries for potential performance improvement.

In order to demonstrate the feasibility of the binary install-time optimization framework and illustrate the accurate translation of sample binaries, this chapter details the results of some sample experiments performed using executables from the SPEC95 benchmark suite and the executable and DLL which constitute the IMPACT timing simulator. The AMD K6 optimizations implemented by Thiems [3] and an AMD K6-2 machine description compiled for this thesis were used to demonstrate the potential for performance improvement.

Table 9.1 Benchmarks used with the optimization framework in thesis experiments.

Benchmark Name	Input Run with Benchmark
124.m88ksim	SPEC95 clt.in reference input
129.compress	SPEC95 bigtest.in reference input
130.li	SPEC95 *.lsp reference inputs
speedsim	simulator EXE and DLL with MSWord trace input

The experiments were performed on a 300-MHz AMD K6-2 system with 64-MB RAM running Windows 98. The K6-2 processor implements a superscalar x86 architecture, and its microarchitecture allows parallel short decodes and the issue of six RISC86 microoperations per cycle to eight execution units. Although the K6-2 is architecturally similar to the Intel Pentium or Pentium Pro, its distinct microarchitecture requires optimizations different from those employed by Pentium programmers. The popular Microsoft Visual C++ 5.0 does not target code generation for the AMD K6-2, so the processor remains a good candidate for reoptimization.

The binaries analyzed in the experiments (shown in Table 9.1) were compiled using Microsoft Visual C++ 5.0 with optimization for maximum speed. To collect performance data, each version of each benchmark was run three times, and the wall clock time was recorded for each run. The average of the three times is displayed for each version of the benchmark. The Microsoft utility *timethis.exe* was used to produce the wall clock time. The results are given in terms of the speedup over the original executable or dynamic-link library. Speedup can be calculated as

$$SPEEDUP = \frac{original\_runtime}{new\_runtime}. \tag{9.1}$$

The translation process used in the reoptimization framework results in the reordering of the functions in the binary based upon the depth-first traversal used to discover the functions. The reorganization can impact program performance due to cache and paging effects, but the details of these effects have not been thoroughly studied. Although the reordering often inherently decreases the binary runtime, it has also been shown to occasionally slow down the executable. Further research is necessary to determine the causes of and potential solutions for these effects.

9.1 Using the Framework to Reschedule with MDES

A few of the SPEC95 applications were run through the binary optimization framework to examine the benefit of employing accurate processor machine descriptions to facilitate improved code scheduling for faster, more efficient program execution. The results shown in Figure 9.1 illustrate that much remains to be learned about the relationship between the rescheduler and the processor. One possible explanation for the behavior is also the effects of cache size and paging on program execution. Additionally, the machine description does not yet have the capability to ensure that rescheduling does not place an instruction across a cache line boundary. Instructions that span cache lines are automatically considered to be long decode instructions, which would eliminate the benefit of rescheduling for two parallel short decodes. Furthermore, basic memory disambiguation capability is currently being added to the framework and was not available for these experiments. Static code scheduling is substantially restricted by undetermined dependences between memory instructions. Even limited memory disambiguation will allow the scheduler manager more freedom in reordering those memory instructions and their dependents.

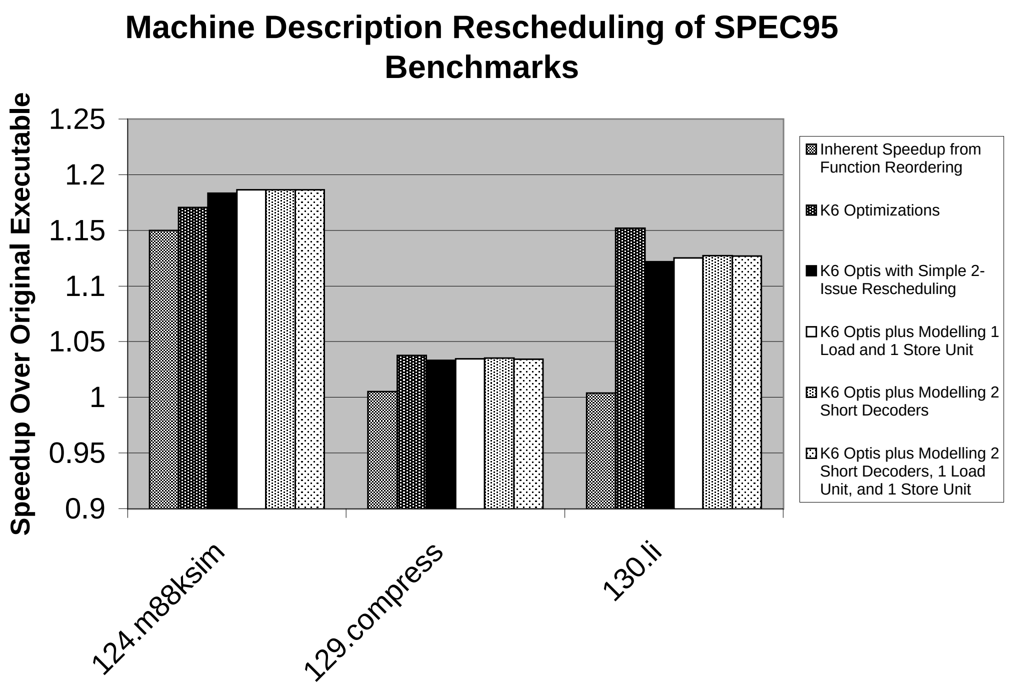

Figure 9.1 Reoptimization of sample SPEC95 benchmarks using K6-2 MDES.

The graph shown in Figure 9.1 illustrates that a speedup does indeed result from optimization within the framework, but the effect of the machine description can be questionable in some cases. Further research is required to determine the exact effects and to find new ways to correct for the negative effects. As mentioned above, the negative effects most likely center around cache boundary and ambiguous memory dependence issues.

Three benchmarks from the SPEC95 benchmark suite (124.m88ksim, 129.compress, and 130.li) were optimized in a variety of ways. The first bar of each set represents execution speedup from the inherent function reordering of the framework. The second bar adds the K6-specific optimizations implemented by Thiems [3]. These optimizations include replacing a reference to the *esi* register with *[esi+0]*, using *mov 0* to clear a register, and replacing certain chains of instructions with a single *imul* operation. The third bar performs the optimizations and reschedules based upon a simple two-issue machine description. Bar four replaces the simple machine description with an AMD K6-2 MDES constraining rescheduling with only one load unit and one store unit. The next result also adds two parallel short decoders to the MDES. The final bar removes the load/store constraint to only model the effects of the parallel short decode.

9.2 Using the Framework to Optimize a DLL

In order to demonstrate that the optimization framework can successfully translate and optimize both 32-bit Windows EXEs and DLLs, the IMPACT timing simulator was separated into an EXE and a single DLL. The timing simulator, speedsim, takes an input instruction trace and processes each instruction in the trace. Speedsim models processor resources and dependences and determines the number of processor clock cycles necessary to execute the instruction stream. It has been modified for the runtime optimization work done in [20] to identify hot spots in the code which represent optimization opportunities. The trace piped through speedsim for this experiment was recorded by a special hardware unit, known as *SpeedTracer*, that captures dynamic instruction traces on an AMD K6 platform. The trace was taken while editing a document in Microsoft Word 97. The instructions which read from the instruction trace pipe and disassemble each instruction were placed in the DLL. The initialization routines, main program loop, processor constraint simulators, and hot spot detection functions resided in the EXE.

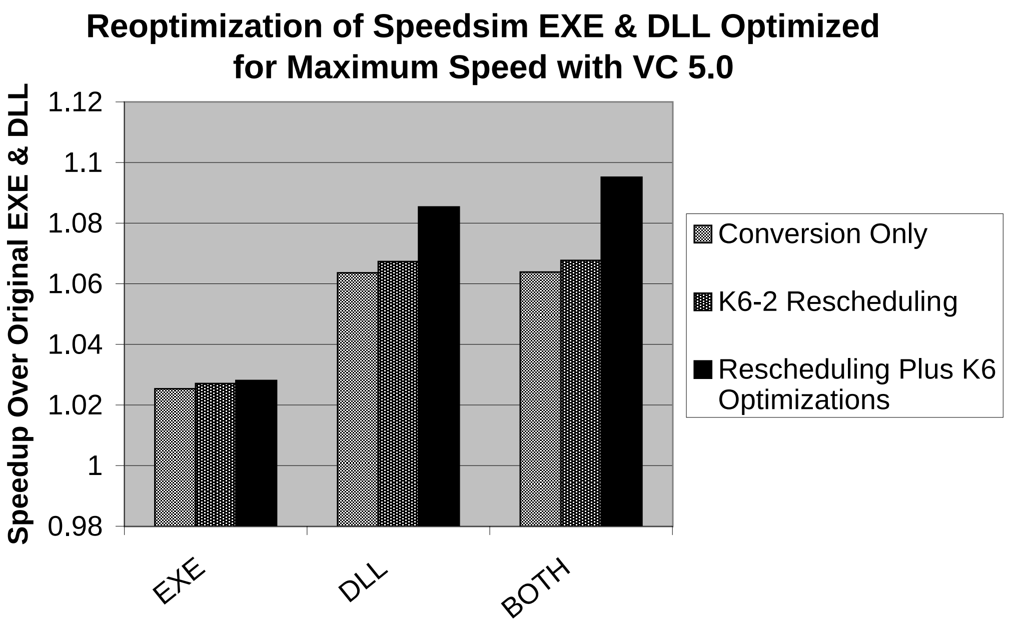

Figure 9.2 Reoptimization of speedsim.dll and speedsim.exe.

The binary reoptimization framework was used to optimize the EXE and the DLL, and performance results were recorded for each binary. The first set of data represents the translated EXE run with the original DLL. The second set records the performance of the translated DLL run with the original EXE. The third set records the results when the EXE and DLL are both translated. The first bar in each set illustrates the performance improvement inherent in the function reordering performed by Lbx86 and demonstrates that even the depth-first reorganization can positively impact run time. The second bar in each group shows the performance improvement resulting from instruction rescheduling based upon the AMD K6-2 machine description discussed in Section 6.2. The final bar combines the effects of rescheduling with the K6 optimizations implemented by Thiems in [3].

The data clearly show that both the executable and the dynamic-link library can be correctly translated by the framework and reexecuted. Furthermore, the results demonstrate that significant performance improvement can be gained through the DLL optimization, as well as the EXE optimization. These experiments support the assertion of this thesis that DLLs can be optimized with important benefits to the overall program.

CHAPTER 10

CONCLUSION

This thesis provides a description of the static code discovery mechanisms utilized in the IMPACT binary reoptimization framework and discusses some of the optimization, binary regeneration, and binary writeback issues involved in the process. Thiems [3], Merten [4], and the present thesis represent a complete treatise on the framework and its methodology. The system facilitates the decoding, optimization, and writeback of binary executables, object files, or dynamic-link libraries, and provides an extremely flexible framework for additional optimizations and even for the inclusion of run-time information.

Although the binary reoptimizer has been implemented for the x86 architecture, its capabilities are easily applicable to a wide variety of architectures, provided that the corresponding instruction sets can be decoded and converted into the Mcode internal representation. An additional machine description would ensure that the scheduler targets the different processor resources. This flexibility ensures that the framework will remain viable despite shifts in platform preference and architectural development. Although some of the heuristics employed in the framework are not fool-proof, the majority of system experiments have resulted in the successful decoding and optimization of 32-bit Windows x86 executables and dynamic link libraries. These experiments and the data presented in this thesis show that the system is indeed functional and applicable to real-world applications.

The open-ended nature of the binary reoptimization framework allows great potential for future work. Many classical compiler optimizations can be ported from the IMPACT compiler to the framework, and memory disambiguation is in the process of being added to the system. The scheduling issues surrounding the machine description and the modeling of system constraints also offer further research directions. Much work is also being done on a runtime optimization project, and information gleaned from the runtime monitoring of a binary can be utilized in a first or second pass of the binary reoptimizer to further tailor performance optimizations. Furthermore, information generated by the binary reoptimizer can be used to identify and target certain code areas during runtime optimization.

REFERENCES

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

[2] Advanced Micro Devices, "3DNow! technology manual," Tech. Rep. 21928, Advanced Micro Devices, Sunnyvale, CA, May 1998.

[3] M. S. Thiems, "Optimization and executable regeneration in the IMPACT binary reoptimization framework," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[4] M. C. Merten, "A framework for profile-driven optimization in the IMPACT binary reoptimization system," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1999.

[5] R. J. Hookway and M. A. Herdeg, "Digital FX!32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, August 1997.

[6] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proc. of the 16th ACM Symposium of Operating Systems Principles*, October 1997, pp. 15–26.

[7] Stanford Compiler Group, "SUIF: A parallelizing and optimizing research compiler," Tech. Rep. CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[8] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of Win32/Intel executables using etch," in *Proceedings of the USENIX Windows NT Workshop*, August 1997, pp. 1–7.

[9] M. Pietrek, "Peering inside PE: A tour of the Win32 portable executable format," *Microsoft Systems Jouranl*, vol. 9, pp. 15–34, March 1994.

[10] Visual C++ Business Unit, "Microsoft portable executable and common object file format specification 4.1," tech. rep., MSDN Library, Microsoft Corporation, August 1994.

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

[12] M. Pietrek, *Windows 95 System Programming Secrets.* Foster City, CA: IDG Books Worldwide, 1995.

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

[14] University of Illinois, Urbana, IL, *IMPACT Lcode Tutorial*, 1998.

[15] J. C. Gyllenhaal, "An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism," PhD dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[16] J. C. Gyllenhaal, "A machine description language for compilation," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[17] 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.

[18] Advanced Micro Devices, "AMD-K6-2 processor code optimization application note," Tech. Rep. 21924, Advanced Micro Devices, Sunnyvale, CA, May 1998.

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

[20] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," to appear in *Proceedings of the 1999 International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

APPENDIX A

BINARY DECODER STRUCTURES

Chapter 5 details the algorithms employed by the x86toM decoder to disassemble the binary input file and convert its x86 instructions into the intermediate Mcode representation used by the Lbx86 optimizer. To facilitate conversion and optimization, the components of the binary, the functions, and the constituent blocks of x86 instructions are all stored in various structures in memory. For completeness, these structures are reproduced here in Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, and A.8.

```
typedef struct _BinaryFileInfo
{
  HANDLE fileHandle;
  HANDLE fileMappingHandle;
  LPVOID fileBase; /* location at which file is memory-mapped */
  PIMAGE_FILE_HEADER imgFileHdr;
  PIMAGE_SECTION_HEADER origSectTable;
  WORD numSections;
  PBYTE stringTabBase;
  struct _BinarySymbolTable *symTab; /* pointer to symbol table */
  SectionInfoNode *sectList; /* list of section structures */
  SectionInfoNode *processSection;
  BinaryType fileType;
  RelocList *relocList; /* pointer to list of relocations */
  BYTE reloc_info;
  DWORD imageBase;
  /* general shortcut buffers to speed up section lookups */
  SectionInfoNode *XLTM_buf,
  *XMTL_buf,
  *GSIBN_buf,
  *GSIBA_buf,
  *XLTON_buf;
  DWORD sectionAlignment;
  DWORD fileAlignment;
  PIMAGE_OPTIONAL_HEADER imgOptHdr;
  TableInfo tableInfo[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} BinaryFileInfo;
```

Figure A.1 BinaryFileInfo structure.

```
typdef struct _SectionInfoNode
{
  /* Use for the original mapped file. */
  PIMAGE_SECTION_HEADER info;
/* pointer to original section information */
  WORD sectionNum;   /* numeric id of section */
  char *Name; /* name of section */
  /* Use for the new, grown sections. */
  DWORD newfbase; /* new base file address of section */
  DWORD newfsize; /* new file size of section */
  DWORD newvbase; /* new virtual base address of section */
  DWORD newvsize; /* new virtual size of section */
  DWORD newrbase;
  PIMAGE_RELOCATION newrelocs;
  struct _SectionInfoNode *next;
} SectionInfoNode;
```

Figure A.2 SectionInfoNode structure.

```
typedef struct EntryPt
{
  EntryType type; /* type of entry pt (CALL, Jcc, JMP, etc.) */
  BlockInfo *block; /* ptr to bock jumped/called from */
  DWORD nextIP; /* IP of next instruction to be decoded */
  DWORD origin; /* address of instruction that generated EP */
  DWORD fnid; /* numeric id of parent function */
  DWORD cbid; /* numeric id of parent block */
} EntryPt;
```

Figure A.3 EntryPt structure.

```
typedef struct InstrInfo
{
   OPC_ENUM opcode;
   BYTE *instrPtr;          /* address of current instruction in virtual RAM */
   BYTE addrmode;           /* selects addressing mode (16- or 32- bit) */
   OPERAND_ENUM opertype[3];/* specifies type of operand */
   BYTE opersize[3];        /* specifies operand size (8-, 16- or 32- bit) */
   REG_ENUM regnum[3];      /* specifies actual register numbers for operands */
   BYTE immediate[4];     /* instruction immediate value */
   BYTE displacement[6];    /* displacement value for the operation */
   BYTE numDispBytes;
   REG_ENUM basereg; /* base reg of the SIB */
   REG_ENUM indexreg; /* index reg of the SIB */
   BYTE scale; /* scale value of the SIB */
   BYTE InstrLen; /* overall length (in bytes) of the instruction */
   OPC_ENUM prefix; /* instruction prefix value */
   OPC_ENUM segOveride;
   FLAG_ENUM InstrFlags;
   BYTE immdOfst;
   BYTE dispOfst;
   BYTE reloc32Imm;
   BYTE reloc32MemDisp;
   int amd3d;               /* Is this a 3DNow instruction? */
} InstrInfo;
```

Figure A.4 InstrInfo structure.

```
typedef struct _BlockInfo
{
  struct _BlockInfo *next; /* pointer to next block */
  struct _BlockInfo *prev; /* pointer to prev block */
  DWORD start; /* starting address of block */
  DWORD length; /* length of block */
  DWORD relocatedStart; /* start of block after probe ins. */
  int delta; /* change in size of this block */
  DWORD fnid; /* Mcode function id */
  DWORD cbid; /* Mcode control block id */
  BlockType type; /* block type (jmp tbl, reg code, etc.) */
  RelocNode *relocs; /* linked-list of immediate addresses in
   block that may need fixup after probe
   insertion */
  L_Datalist *datalist;
  DWSNode *jump_rg_targets; /* only for tables that are out of bounds */
  DWSNode *jump_rg_conds; /* only for tables that are out of bounds */
} BlockInfo;
```

Figure A.5 BlockInfo structure.

```
typedef struct _MemoryMap
{
  BlockInfo *headBlock; /* 1st block of memory map */
  BlockInfo *currBlock; /* pointer to current block of memory map */
/* current block is block being modified by
   the decoder */
  BlockInfo *tailBlock; /* last block of memory map */
  BlockInfo **blockArray; /* array for shortcut table which allows easy
   and rapid block lookup */
  DWORD chunkSize, /* spacing (bytes) of shortcut entries */
arraySize, /* size of shortcut table */
baseOfst, /* base of mapped memory section */
length; /* length of mapped memory section */
} BlockMap;
```

Figure A.6 MemoryMap structure.

```
typedef struct FunctionNode
{
  DWORD fnid; /* numeric id of the function */
  DWSNode *headEntryPt; /* pointer to list of function entry points */
  DWSNode *headExitPt; /* pointer to list of function exit pointers */
  DWSNode *calledFrom; /* pointer to list of function call sites */
  CBNode *headCB; /* pointer to list of constituent blocks in function */
  CBNode *tailCB;
  int entryCount; /* count of entry points listed in headEntryPt array */
  int exitCount; /* count of exit points listed in headExitPt array */
  int cbCount; /* count of constituent blocks in headCB array */
  int lastCbidAssigned; /* last cb id assigned for function */
  char *name; /* dynamically allocated fn name */
  /* FunctionTable supports function lookup, addition, merging, and deletion */
  struct FunctionTable *table; /* for deletion of function */
  struct FunctionNode *nextFunction; /* for table's contents list */
  struct FunctionNode *prevFunction;
  /* hash table allows rapid and easy access to functions in table using id */
  struct FunctionNode *nextHash; /* for table's hash table */
  struct FunctionNode *prevHash;
  int livePoints; /* number of entry points pending */
  DWSNode *pushedLoc; /* pointers pushed to stack */
  char poison; /* if function is invalidated, poison
   flag is set to true. */
  L_Func *mcodeFn; /* pointer to mcode structure for function */
  int numberOfJumpTables; /* number of jump tables used by function */
} FunctionNode;
```

Figure A.7 FunctionNode structure.

```
typedef struct _BaseRelocEntry
{
  DWORD VirtualAddress; /* complete virtual address of fixup */
  DWORD RVA; /* relative virtual address of fixup page */
  WORD Type; /* type of fixup (HIGHLOW == 3, ABSOLUTE == 0) */
  WORD TypeOffset; /* combination of type and offset into virtual page */
} BaseRelocEntry;
```

Figure A.8 BaseRelocEntry structure.

APPENDIX B

MACHINE DESCRIPTION STRUCTURE

Chapter 6 described the format of a machine description used by the scheduler manager to reschedule instructions for a target processor. Figure B.1 illustrates the various sections of a machine description and identifies the relationships between the machine description components.

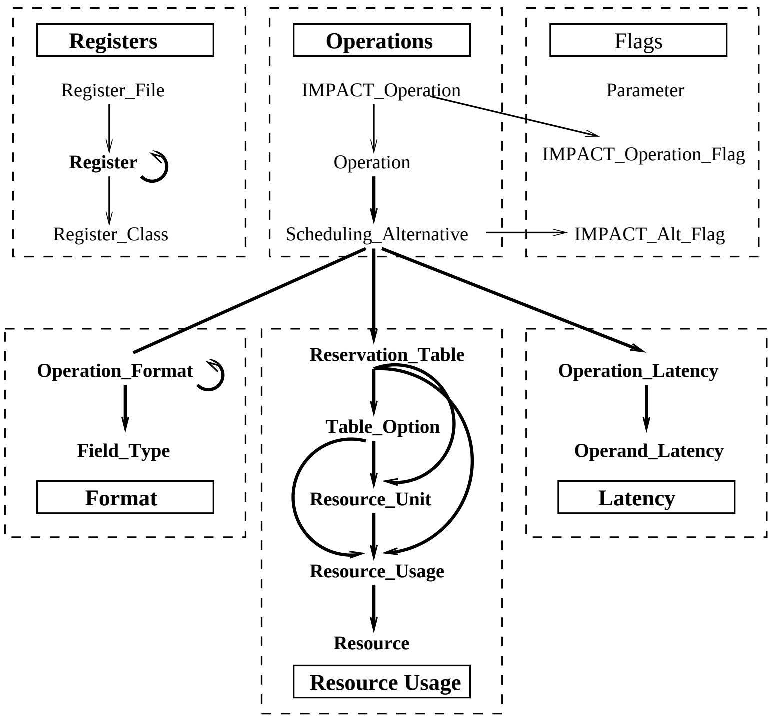

Figure B.1 Basic structural relationship of a machine description. Bold sections and arrows indicate compiler-independent features.

APPENDIX C

AMD K6-2 MACHINE DESCRIPTION

The experiments presented in this thesis utilize a machine description which reflects the resources and constraints of the AMD K6-2 microprocessor. The machine description file is included below for reference, and the description reflects the dual short decode units, the single load unit, and the single store unit.

```
# AMD K6-2 machine description.
#
# Author: Christopher N. George with John Gyllenhaal
# Date:   3/19/99
#
$include "${IMPACT_REL_PATH}/mdes/structure/structure_IMPACT.hmdes2"

SECTION Parameter
{
  processor_model
                (value("superscalar"));
  customization_headers
                (value("defs"
                       "${IMPACT_REL_PATH}/src/include/Lcode/l_opc.h"
                       "${IMPACT_REL_PATH}/src/include/Lcode/limpact_phase1.h"
                       "${IMPACT_REL_PATH}/src/include/machine/m_spec.h"
                       "${IMPACT_REL_PATH}/src/include/machine/m_bx86.h"
                       "${IMPACT_REL_PATH}/src/include/Lcode/l_flags.h"));
}

SECTION Field_Type
{
  Lit           ();
  Label         ();
  NULL          ();
  REG           ();
  ANY           (compatible_with(Lit Label NULL REG ANY));
}

/* Allow any instruction format */
SECTION Operation_Format
{
  IOI_ANY       (pred() src(ANY ANY ANY ANY ANY ANY ANY)
                 dest(ANY ANY ANY ANY));
}

SECTION Resource
{
  /* Minimum requirement, one scheduling slot */
  slot0 (slot(0));
  slot1 (slot(1));
  slot2 (slot(2));
  decode0 (); /* 1st short decoder */
  decode1 (); /* 2nd short decoder */
  load_port (); /* single load unit */
  store_port (); /* single store unit */
}

SECTION Resource_Usage
{
  RU_slot0_t0_0 (use(slot0) time(0));
  RU_slot1_t0_0 (use(slot1) time(0));
  RU_slot2_t0_0 (use(slot2) time(0));
  RU_decode0_t0_0 (use(decode0) time(0));
  RU_decode1_t0_0 (use(decode1) time(0));
  RU_load_port_t1_1 (use(load_port) time(1));
  RU_store_port_t1_1 (use(store_port) time(1));
}

SECTION Resource_Unit
{
}

SECTION Table_Option
{
  two_issue (one_of(RU_slot0_t0_0 RU_slot1_t0_0));
  three_issue (one_of(RU_slot0_t0_0 RU_slot1_t0_0 RU_slot2_t0_0));
  short_decode (one_of(RU_decode0_t0_0 RU_decode1_t0_0));
}

SECTION Reservation_Table
{
//  RL_ALL        (use(RU_slot0_t0_0));
    RL_ALL        (use(two_issue));
    RL_Load   (use(two_issue RU_load_port_t1_1));
    RL_Store   (use(two_issue RU_store_port_t1_1));
//  RL_ALL        (use(three_issue));
    RL_Short_Dec  (use(two_issue short_decode));
    RL_Long_Dec   (use(two_issue RU_decode0_t0_0 RU_decode1_t0_0));
    RL_Short_Load (use(two_issue short_decode RU_load_port_t1_1));
    RL_Long_Load  (use(two_issue RU_decode0_t0_0 RU_decode1_t0_0
RU_load_port_t1_1));
    RL_Short_Store (use(two_issue short_decode RU_store_port_t1_1));
    RL_Long_Store  (use(two_issue RU_decode0_t0_0 RU_decode1_t0_0
RU_store_port_t1_1));
}
```

```
SECTION Operand_Latency
{
  s0            (time(0));
  d1            (time(1));
  p0            (time(0));
  ss0           (time(0));
  sd0           (time(0));
}

SECTION Operation_Latency
{
  Lat_ALL       (dest(d1 d1)
                 src(s0 s0 s0 s0)
                 pred(p0)
                 mem_dest(sd0)
                 ctrl_dest(sd0)
                 sync_dest(sd0)
                 mem_src(ss0)
                 ctrl_src(ss0)
                 sync_src(ss0));
}

SECTION Scheduling_Alternative
{
  ALT_ALL       (format(IOI_ANY)
                 resv(RL_ALL)
         latency(Lat_ALL));

  ALT_Load       (format(IOI_ANY)
                 resv(RL_Load)
         latency(Lat_ALL));

  ALT_Store     (format(IOI_ANY)
                 resv(RL_Store)
         latency(Lat_ALL));

  ALT_Short_Dec (format(IOI_ANY)
 resv(RL_Short_Dec)
 latency(Lat_ALL));

  ALT_Long_Dec  (format(IOI_ANY)
 resv(RL_Long_Dec)
 latency(Lat_ALL));

  ALT_Short_Load (format(IOI_ANY)
 resv(RL_Short_Load)
 latency(Lat_ALL));

  ALT_Long_Load (format(IOI_ANY)
 resv(RL_Long_Load)
 latency(Lat_ALL));

  ALT_Short_Store (format(IOI_ANY)
 resv(RL_Short_Store)
 latency(Lat_ALL));

  ALT_Long_Store (format(IOI_ANY)
 resv(RL_Long_Store)
 latency(Lat_ALL));
}

SECTION Operation
{
  OP_ALL        (alt(ALT_ALL));
  OP_Load       (alt(ALT_Load));
  OP_Store      (alt(ALT_Store));
  OP_Short_Dec  (alt(ALT_Short_Dec));
  OP_Long_Dec   (alt(ALT_Long_Dec));
  OP_Short_Load (alt(ALT_Short_Load));
  OP_Long_Load  (alt(ALT_Long_Load));
  OP_Short_Store (alt(ALT_Short_Store));
  OP_Long_Store (alt(ALT_Long_Store));
}

SECTION IMPACT_Operation
{
$include "classify_decode"

}



# classify_decode
# associate operations with AMD K6-2 machine resources and operations
#
# Author: Christopher N. George with John Gyllenhaal and Mike Thiems
# Date:   3/19/99
#
$include "${IMPACT_REL_PATH}/mdes/structure/structure_IMPACT.hmdes2"

  // Ignored ops
  $for (NAME in
         P_WAIT
         P_DIRECTIVE
       )
   {
     ${NAME}  (op(OP_Long_Dec) flags(IGNORE));
   }

  // Synchronization ops
  $for (NAME in
         P_F2XM1
         P_FABS
         P_FADD_REG_REG
         P_FADDP
         P_FCHS
         P_FCMOVB
         P_FCMOVBE
         P_FCMOVE
         P_FCMOVNB
         P_FCMOVNBE
         P_FCMOVNE
         P_FCMOVNU
         P_FCMOVU
         P_FCOM_REG_REG
         P_FCOMI
         P_FCOMIP
         P_FCOMP_REG_REG
         P_FCOMPP
         P_FCOS
         P_FDECSTP
         P_FDIV_REG_REG
         P_FDIVP
         P_FDIVR_REG_REG
         P_FDIVRP
         P_FFREE
         P_FINCSTP
         P_FLD_REG_REG
         P_FLDLG2
```

```
          P_FLDLN2
          P_FLDL2E
          P_FLDL2T
          P_FLDPI
          P_FLDZ
          P_FLD1
          P_FMUL_REG_REG
          P_FMULP
          P_FNOP
          P_FPATAN
          P_FPREM
          P_FPREM1
          P_FRNDINT
          P_FSCALE
          P_FSETPM
          P_FSIN
          P_FSQRT
          P_FST_REG_REG
          P_FSTP_REG_REG
          P_FSUB_REG_REG
          P_FSUBP
          P_FSUBR_REG_REG
          P_FSUBRP
          P_FTST
          P_FUCOM
          P_FUCOMI
          P_FUCOMIP
          P_FUCOMP
          P_FUCOMPP
          P_FXAM
          P_FXCH
          P_FYL2X
          P_FYL2XP1
        )
    {
      ${NAME}  (op(OP_Short_Dec) flags(SYNC));
    }

   // Synchronization ops
   $for (NAME in
P_HLT
P_MOV_SREG_SREG_REG
P_FCLEX
```

```
P_FINIT
P_FPTAN
P_FSINCOS
P_FSTSW_REG_REG
P_FXTRACT
)
   {
${NAME}  (op(OP_Long_Dec) flags(SYNC));
   }

  // JSR ops
  $for (NAME in
         P_CALL_REL_NEAR
       )
   {
     ${NAME}  (op(OP_Short_Dec) flags(JSR NI));
   }

  // JSR ops
  $for (NAME in
         P_CALL_REG_NEAR
       )
   {
     ${NAME}  (op(OP_Long_Dec) flags(JSR NI));
   }

  // JSR with LOAD ops
  $for (NAME in
         P_CALL_MEM_NEAR
       )
   {
     ${NAME}  (op(OP_Long_Load) flags(JSR EXCEPT LOAD NI));
   }

  // RTS ops
  $for (NAME in
         P_RET_WITH_IMM
         P_RET_NO_IMM
         P_RETF_WITH_IMM
         P_RETF_NO_IMM
       )
   {
     ${NAME}  (op(OP_Long_Dec) flags(RTS NI));
```

```
 }

// JUMP ops
$for (NAME in
       P_JMP_REL_NEAR
     )
 {
   ${NAME}  (op(OP_Short_Dec) flags(JMP NI));
 }

// JUMP ops
$for (NAME in
       P_JMP_REG_NEAR
     )
 {
   ${NAME}  (op(OP_Long_Dec) flags(JMP NI));
 }

// JUMP with LOAD ops
$for (NAME in
       P_JMP_MEM_NEAR
     )
 {
   ${NAME}  (op(OP_Long_Load) flags(JMP EXCEPT LOAD NI));
 }

// Conditional branch ops
$for (NAME in
       P_JA_CBR
       P_JAE_CBR
       P_JB_CBR
       P_JBE_CBR
       P_JG_CBR
       P_JGE_CBR
       P_JL_CBR
       P_JLE_CBR
       P_JO_CBR
       P_JNO_CBR
       P_JP_CBR
       P_JNP_CBR
       P_JS_CBR
       P_JNS_CBR
       P_JZ_CBR
       P_JNZ_CBR
       P_LOOP_CBR
     )
 {
   ${NAME}  (op(OP_Short_Dec) flags(CBR NI));
 }

// Conditional branch ops
$for (NAME in
       P_JCXZ_CBR
       P_LOOPZ_CBR
       P_LOOPNZ_CBR
     )
 {
   ${NAME}  (op(OP_Long_Dec) flags(CBR NI));
 }

// LOAD ops
$for (NAME in
       P_ADD_REG_MEM
       P_AND_REG_MEM
       P_CMP_MEM_IMM
       P_CMP_MEM_REG
       P_CMP_REG_MEM
       P_MOV_REG_MEM_ACC
       P_MOV_REG_MEM
       P_MOVSX_REG_MEM
       P_MOVZX_REG_MEM
       P_OR_REG_MEM
       P_POP_eSP
       P_POP_REG
       P_SUB_REG_MEM
       P_XOR_REG_MEM
       P_MOVD_REG_MEM
       P_MOVQ_REG_MEM
       P_PACKSSDW_REG_MEM
       P_PACKSSWB_REG_MEM
       P_PACKUSWB_REG_MEM
       P_PADDB_REG_MEM
       P_PADDD_REG_MEM
       P_PADDW_REG_MEM
       P_PADDSB_REG_MEM
       P_PADDSW_REG_MEM
```

```
P_PADDUSB_REG_MEM
P_PADDUSW_REG_MEM
P_PAND_REG_MEM
P_PANDN_REG_MEM
P_PCMPEQB_REG_MEM
P_PCMPEQD_REG_MEM
P_PCMPEQW_REG_MEM
P_PCMPGTB_REG_MEM
P_PCMPGTD_REG_MEM
P_PCMPGTW_REG_MEM
P_PMADDWD_REG_MEM
P_PMULHW_REG_MEM
P_PMULLW_REG_MEM
P_POR_REG_MEM
P_PSUBB_REG_MEM
P_PSUBSB_REG_MEM
P_PSUBSW_REG_MEM
P_PSUBUSB_REG_MEM
P_PSUBUSW_REG_MEM
P_PSUBD_REG_MEM
P_PSUBW_REG_MEM
P_PUNPCKHBW_REG_MEM
P_PUNPCKHWD_REG_MEM
P_PUNPCKHDQ_REG_MEM
P_PUNPCKLBW_REG_MEM
P_PUNPCKLWD_REG_MEM
P_PUNPCKLDQ_REG_MEM
P_PXOR_REG_MEM
P_PAVGUSB_REG_MEM
P_PF2ID_REG_MEM
P_PFACC_REG_MEM
P_PFADD_REG_MEM
P_PFCMPEQ_REG_MEM
P_PFCMPGE_REG_MEM
P_PFCMPGT_REG_MEM
P_PFMAX_REG_MEM
P_PFMIN_REG_MEM
P_PFMUL_REG_MEM
P_PFRCP_REG_MEM
P_PFRCPIT1_REG_MEM
P_PFRCPIT2_REG_MEM
P_PFRSQIT1_REG_MEM
P_PFRSQRT_REG_MEM
```

```
        P_PFSUB_REG_MEM
        P_PFSUBR_REG_MEM
        P_PI2FD_REG_MEM
        P_PMULHRW_REG_MEM
      )
  {
    ${NAME}  (op(OP_Short_Load) flags(EXCEPT LOAD));
  }

 // LOAD ops
 $for (NAME in
P_ADC_REG_MEM
        P_BOUND
        P_BSF_REG_MEM
        P_BSF_REG_MEM_BIG
        P_BSR_REG_MEM
        P_BSR_REG_MEM_BIG
        P_BT_MEM_IMM
        P_BT_MEM_REG
        P_BT_REG_MEM
        P_BTC_REG_MEM
        P_BTR_REG_MEM
        P_BTS_REG_MEM
        P_CMPS_REPZ
        P_CMPS_REPNZ
        P_CMPS_REP
        P_CMPS_NO_REP
        P_DIV_MEM_32
        P_DIV_MEM_8
        P_DIV_MEM_16
        P_IDIV_MEM_32
        P_IDIV_MEM_8
        P_IDIV_MEM_16
        P_IMUL_REG_MEM_IMM
        P_IMUL_MEM_32
        P_IMUL_MEM_8
        P_IMUL_MEM_16
        P_IMUL_REG_MEM
        P_LEAVE
        P_LODS_REPZ
        P_LODS_REPNZ
        P_LODS_REP
        P_LODS_NO_REP
```

```
         P_MUL_MEM_32
         P_MUL_MEM_8
         P_MUL_MEM_16
         P_OUTS_REPZ
         P_OUTS_REPNZ
         P_OUTS_REP
         P_OUTS_NO_REP
         P_POPA
         P_POPF
         P_PUSH_MEM
         P_SBB_REG_MEM
         P_SCAS_REPZ
         P_SCAS_REPNZ
         P_SCAS_REP
         P_SCAS_NO_REP
         P_TEST_MEM_IMM
         P_TEST_MEM_REG
         P_TEST_REG_MEM
         P_XLAT
 P_PREFETCH_MEM_8
)
   {
${NAME} (op(OP_Long_Load) flags(EXCEPT LOAD));
   }

  // Synchronizing LOAD ops
  $for (NAME in
         P_FADD_REG_MEM
         P_FCOM_REG_MEM
         P_FCOMP_REG_MEM
         P_FDIV_REG_MEM
         P_FDIVR_REG_MEM
         P_FIADD_REG_MEM
         P_FIADD_REG_MEM_BIG
         P_FICOM_REG_MEM
         P_FICOM_REG_MEM_BIG
         P_FICOMP_REG_MEM
         P_FICOMP_REG_MEM_BIG
         P_FIDIV_REG_MEM
         P_FIDIV_REG_MEM_BIG
         P_FIDIVR_REG_MEM
         P_FIDIVR_REG_MEM_BIG
         P_FILD_MEM
```

```
       P_FILD_MEM_BIG
       P_FIMUL_REG_MEM
       P_FIMUL_REG_MEM_BIG
       P_FISTP_MEM
       P_FISTP_MEM_BIG
       P_FISUB_REG_MEM
       P_FISUB_REG_MEM_BIG
       P_FISUBR_REG_MEM
       P_FISUBR_REG_MEM_BIG
       P_FLD_REG_MEM
       P_FLDENV_MEM
       P_FLDENV_MEM_BIG
       P_FMUL_REG_MEM
       P_FSUB_REG_MEM
       P_FSUBR_REG_MEM
     )
 {
   ${NAME}  (op(OP_Short_Load) flags(EXCEPT LOAD SYNC));
 }

// Synchronizing LOAD ops
$for (NAME in
       P_LDS
       P_LES
       P_LFS
       P_LGS
       P_LSS
       P_MOV_SREG_SREG_MEM
       P_POP_CS
       P_POP_SS
       P_POP_DS
       P_POP_ES
       P_POP_FS
       P_POP_GS
       P_FBLD
       P_FLD_REG_MEM_BIG
       P_FLDCW
       P_FRSTOR_MEM
       P_FRSTOR_MEM_BIG
       P_FSAVE_MEM
       P_FSAVE_MEM_BIG
       P_FSTENV_MEM
       P_FSTENV_MEM_BIG
     )
 {
   ${NAME}  (op(OP_Long_Load) flags(EXCEPT LOAD SYNC));
 }

// STORE ops
$for (NAME in
       P_ADC_MEM_IMM
       P_ADC_MEM_REG
       P_ADD_MEM_IMM
       P_ADD_MEM_REG
       P_AND_MEM_IMM
       P_AND_MEM_REG
       P_ARPL_MEM_REG
       P_BTC_MEM_IMM
       P_BTC_MEM_REG
       P_BTR_MEM_IMM
       P_BTR_MEM_REG
       P_BTS_MEM_IMM
       P_BTS_MEM_REG
       P_DEC_MEM_IMM
       P_ENTER
       P_INC_MEM_IMM
       P_INS_REPZ
       P_INS_REPNZ
       P_INS_REP
       P_INS_NO_REP
       P_MOV_MEM_IMM
       P_MOV_SREG_MEM_SREG
       P_MOVS_REPZ
       P_MOVS_REPNZ
       P_MOVS_REP
       P_MOVS_NO_REP
       P_NEG_MEM_IMM
       P_NOT_MEM_IMM
       P_OR_MEM_IMM
       P_POP_MEM
       P_PUSH_CS
       P_PUSH_SS
       P_PUSH_DS
       P_PUSH_ES
       P_PUSH_FS
       P_PUSH_GS
```

```
P_PUSH_REG
P_PUSH_IMM
P_PUSHA
P_PUSHF
P_RCL_MEM_BY_1
P_RCL_MEM_BY_IMM
P_RCL_MEM_BY_CL
P_RCR_MEM_BY_1
P_RCR_MEM_BY_IMM
P_RCR_MEM_BY_CL
P_ROL_MEM_BY_1
P_ROL_MEM_BY_IMM
P_ROL_MEM_BY_CL
P_ROR_MEM_BY_1
P_ROR_MEM_BY_IMM
P_ROR_MEM_BY_CL
P_SAL_MEM_BY_1
P_SAL_MEM_BY_IMM
P_SAL_MEM_BY_CL
P_SAR_MEM_BY_1
P_SAR_MEM_BY_IMM
P_SAR_MEM_BY_CL
P_SHR_MEM_BY_1
P_SHR_MEM_BY_IMM
P_SHR_MEM_BY_CL
P_SBB_MEM_IMM
P_SBB_MEM_REG
P_SETA_MEM_IMM
P_SETAE_MEM_IMM
P_SETB_MEM_IMM
P_SETBE_MEM_IMM
P_SETG_MEM_IMM
P_SETGE_MEM_IMM
P_SETL_MEM_IMM
P_SETLE_MEM_IMM
P_SETO_MEM_IMM
P_SETNO_MEM_IMM
P_SETP_MEM_IMM
P_SETNP_MEM_IMM
P_SETS_MEM_IMM
P_SETNS_MEM_IMM
P_SETZ_MEM_IMM
P_SETNZ_MEM_IMM
```

```
       P_SHLD_MEM_BY_IMM
       P_SHLD_MEM_BY_CL
       P_SHRD_MEM_BY_IMM
       P_SHRD_MEM_BY_CL
       P_STOS_REPZ
       P_STOS_REPNZ
       P_STOS_REP
       P_STOS_NO_REP
       P_SUB_MEM_IMM
       P_SUB_MEM_REG
       P_XADD_MEM_IMM
       P_XADD_MEM_REG
       P_XADD_REG_MEM
       P_XCHG_MEM_REG
       P_XCHG_REG_MEM
       P_XOR_MEM_IMM
       P_XOR_MEM_REG
       P_CPUID
    )
 {
   ${NAME}  (op(OP_Long_Store) flags(EXCEPT LOAD STORE));
 }

// STORE ops
$for (NAME in
       P_MOV_MEM_REG_ACC
       P_MOV_MEM_REG
       P_MOVSX_MEM_IMM
       P_MOVSX_MEM_REG
       P_MOVZX_MEM_IMM
       P_MOVZX_MEM_REG
       P_OR_MEM_REG
       P_MOVD_MEM_REG
       P_MOVQ_MEM_REG
    )
 {
   ${NAME}  (op(OP_Short_Store) flags(EXCEPT LOAD STORE));
 }

// Synchronizing STORE ops
$for (NAME in
       P_FIST
       P_FST_MEM_REG
       P_FST_MEM_REG_BIG
       P_FSTP_MEM_REG
       P_FSTP_MEM_REG_BIG
     )
 {
   ${NAME}  (op(OP_Short_Store) flags(EXCEPT STORE SYNC));
 }

// Synchronizing STORE ops
$for (NAME in
       P_FBSTP
       P_FSTCW
       P_FSTSW_MEM_REG
     )
 {
   ${NAME}  (op(OP_Long_Store) flags(EXCEPT STORE SYNC));
 }

// JSR with SYNC ops
$for (NAME in
       P_INTP_I_3
       P_INTP_I_OTHER
       P_INTO
     )
 {
   ${NAME}  (op(OP_Long_Dec) flags(JSR SYNC NI));
 }

// The rest
$for (NAME in
       P_AAA
       P_AAD
       P_AAM
       P_AAS
       P_ADC_REG_IMM_ACC
       P_ADC_REG_IMM
       P_ADC_REG_REG
       P_ARPL_REG_REG
       P_BSF_REG_REG
       P_BSF_REG_REG_BIG
       P_BSR_REG_REG
       P_BSR_REG_REG_BIG
       P_BSWAP
```

```
P_BT_REG_IMM
P_BT_REG_REG
P_BTC_REG_IMM
P_BTC_REG_REG
P_BTR_REG_IMM
P_BTR_REG_REG
P_BTS_REG_IMM
P_BTS_REG_REG
P_CBW
P_CLC
P_CLD
P_CLI
P_CMC
P_CWD
P_DAA
P_DAS
P_DIV_REG_32
P_DIV_REG_8
P_DIV_REG_16
P_IDIV_REG_32
P_IDIV_REG_8
P_IDIV_REG_16
P_IMUL_REG_REG_IMM
P_IMUL_REG_32
P_IMUL_REG_8
P_IMUL_REG_16
P_IMUL_REG_IMM_SPC
P_IMUL_REG_REG
P_INP_REG_IMM
P_INP_REG_REG
P_LAHF
P_MOV_SREG_REG_SREG
P_MUL_REG_32
P_MUL_REG_8
P_MUL_REG_16
P_OUTP_REG_IMM
P_OUTP_REG_REG
P_RCL_REG_BY_1
P_RCL_REG_BY_IMM
P_RCL_REG_BY_CL
P_RCR_REG_BY_1
P_RCR_REG_BY_IMM
P_RCR_REG_BY_CL
```

```
P_ROL_REG_BY_1
P_ROL_REG_BY_IMM
P_ROL_REG_BY_CL
P_ROR_REG_BY_1
P_ROR_REG_BY_IMM
P_ROR_REG_BY_CL
P_SAHF
P_SBB_REG_IMM_ACC
P_SBB_REG_IMM
P_SBB_REG_REG
P_SETA_REG_IMM
P_SETAE_REG_IMM
P_SETB_REG_IMM
P_SETBE_REG_IMM
P_SETG_REG_IMM
P_SETGE_REG_IMM
P_SETL_REG_IMM
P_SETLE_REG_IMM
P_SETO_REG_IMM
P_SETNO_REG_IMM
P_SETP_REG_IMM
P_SETNP_REG_IMM
P_SETS_REG_IMM
P_SETNS_REG_IMM
P_SETZ_REG_IMM
P_SETNZ_REG_IMM
P_SHLD_REG_BY_IMM
P_SHLD_REG_BY_CL
P_SHRD_REG_BY_IMM
P_SHRD_REG_BY_CL
P_STC
P_STD
P_STI
P_TEST_REG_IMM_ACC
P_TEST_REG_IMM
P_XADD_REG_IMM
P_XADD_REG_REG
P_XCHG_REG_REG_ACC
P_XCHG_REG_REG
P_EMMS
P_FEMMS
P_RDMSR
P_RDPMC
```

```
       P_RDTSC
     )
 {
   ${NAME}  (op(OP_Long_Dec));
 }


// The rest
$for (NAME in
       P_ADD_REG_IMM_ACC
       P_ADD_REG_IMM
       P_ADD_REG_REG
       P_AND_REG_IMM_ACC
       P_AND_REG_IMM
       P_AND_REG_REG
       P_CMP_REG_IMM_ACC
       P_CMP_REG_IMM
       P_CMP_REG_REG
       P_DEC_REG_IMM
       P_DEC_REG_IMM_BIG
       P_INC_REG_IMM
       P_INC_REG_IMM_BIG
       P_LEA
       P_MOV_REG_IMM
       P_MOV_REG_REG
       P_MOVSX_REG_IMM
       P_MOVSX_REG_REG
       P_MOVZX_REG_IMM
       P_MOVZX_REG_REG
       P_NEG_REG_IMM
       P_NOP
       P_NOT_REG_IMM
       P_OR_REG_IMM_ACC
       P_OR_REG_IMM
       P_OR_REG_REG
       P_SAL_REG_BY_1
       P_SAL_REG_BY_IMM
       P_SAL_REG_BY_CL
       P_SAR_REG_BY_1
       P_SAR_REG_BY_IMM
       P_SAR_REG_BY_CL
       P_SHR_REG_BY_1
       P_SHR_REG_BY_IMM
```

```
P_SHR_REG_BY_CL
P_SUB_REG_IMM_ACC
P_SUB_REG_IMM
P_SUB_REG_REG
P_TEST_REG_REG
P_XOR_REG_IMM_ACC
P_XOR_REG_IMM
P_XOR_REG_REG
P_MOVD_REG_REG_BIG
P_MOVD_REG_REG
P_MOVQ_REG_REG_BIG
P_MOVQ_REG_REG
P_PACKSSDW_REG_REG
P_PACKSSWB_REG_REG
P_PACKUSWB_REG_REG
P_PADDB_REG_REG
P_PADDD_REG_REG
P_PADDW_REG_REG
P_PADDSB_REG_REG
P_PADDSW_REG_REG
P_PADDUSB_REG_REG
P_PADDUSW_REG_REG
P_PAND_REG_REG
P_PANDN_REG_REG
P_PCMPEQB_REG_REG
P_PCMPEQD_REG_REG
P_PCMPEQW_REG_REG
P_PCMPGTB_REG_REG
P_PCMPGTD_REG_REG
P_PCMPGTW_REG_REG
P_PMADDWD_REG_REG
P_PMULHW_REG_REG
P_PMULLW_REG_REG
P_POR_REG_REG
P_PSLLW_REG_BY_IMM
P_PSLLW_REG_BY_MEM
P_PSLLW_REG_BY_REG
P_PSLLD_REG_BY_IMM
P_PSLLD_REG_BY_MEM
P_PSLLD_REG_BY_REG
P_PSLLQ_REG_BY_IMM
P_PSLLQ_REG_BY_MEM
P_PSLLQ_REG_BY_REG
```

```
P_PSRAD_REG_BY_IMM
P_PSRAD_REG_BY_MEM
P_PSRAD_REG_BY_REG
P_PSRAW_REG_BY_IMM
P_PSRAW_REG_BY_MEM
P_PSRAW_REG_BY_REG
P_PSRLD_REG_BY_IMM
P_PSRLD_REG_BY_MEM
P_PSRLD_REG_BY_REG
P_PSRLQ_REG_BY_IMM
P_PSRLQ_REG_BY_MEM
P_PSRLQ_REG_BY_REG
P_PSRLW_REG_BY_IMM
P_PSRLW_REG_BY_MEM
P_PSRLW_REG_BY_REG
P_PSUBB_REG_REG
P_PSUBSB_REG_REG
P_PSUBSW_REG_REG
P_PSUBUSB_REG_REG
P_PSUBUSW_REG_REG
P_PSUBD_REG_REG
P_PSUBW_REG_REG
P_PUNPCKHBW_REG_REG
P_PUNPCKHWD_REG_REG
P_PUNPCKHDQ_REG_REG
P_PUNPCKLBW_REG_REG
P_PUNPCKLWD_REG_REG
P_PUNPCKLDQ_REG_REG
P_PXOR_REG_REG
P_PAVGUSB_REG_REG
P_PF2ID_REG_REG
P_PFACC_REG_REG
P_PFADD_REG_REG
P_PFCMPEQ_REG_REG
P_PFCMPGE_REG_REG
P_PFCMPGT_REG_REG
P_PFMAX_REG_REG
P_PFMIN_REG_REG
P_PFMUL_REG_REG
P_PFRCP_REG_REG
P_PFRCPIT1_REG_REG
P_PFRCPIT2_REG_REG
P_PFRSQIT1_REG_REG
```

```
      P_PFRSQRT_REG_REG
      P_PFSUB_REG_REG
      P_PFSUBR_REG_REG
      P_PI2FD_REG_REG
      P_PMULHRW_REG_REG
    )
{
  ${NAME}  (op(OP_Short_Dec));
}
```