A ROBUST FOUNDATION FOR BINARY TRANSLATION OF X86 CODE

BY

LIANGCHUAN HSU

B.S., Chung-Cheng Institute of Technology, 1986
M.S., Naval Postgraduate School, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

A ROBUST FOUNDATION FOR
BINARY TRANSLATION OF X86 CODE

Liangchuan Hsu, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Wen-mei W. Hwu, Advisor

Software venders are unwilling to compile new software for a new processor until the marketplace of the processor becomes large enough. As a result, utilization of most new hardware features is delayed. Binary Translation allows programs compiled for older machines to take advantage of all the new features that have since been added. Binary translation can help introduce new instruction set architecture (ISA) features to the software base.

This dissertation develops a solid foundation for binary translation. A hybrid approach to resolving all of branch target problems is proposed and evaluated. It attempts to resolve as many targets as possible statically, and uses a run-time support when the target cannot be resolved at translation time.

Self-modifying code is one of the major hurdles to binary translation. This dissertation describes a self-modifying code detector to prevent the binary translator from translating self-modifying code. In order to illustrate the usefulness of the binary translation framework, an optimizer that improves the load time of executable is designed and implemented in the framework.

To my wife, Chiu-hua, and my children, Amy and Andy.
Thank you for your love, support and longsuffering!

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Wen-mei W. Hwu, for his insight and guidance throughout my studies. Not only was it an honor to work with someone of his caliber, but it was also a pleasure. He truly cares about the needs of his students. Next, I would like to thank Sabrina. It always enjoyed having a barbecue at "Hwuville".

This research would not have been possible without the support of the IMPACT research group. Members of the group were always willing to provide any help required - from research discussions to practice talks. I deeply appreciate the assistance of Roger Bringmann, Scott Mahlke, Dave Gallagher, Richard Hank, Dan Lavery, John Gyllenhaal, David August, Andrew Hsieh, Teresa Johnson, Brian Deitrich, Le-chun Wu, Ben-Chung Cheng, and Matt Trommer.

Thanks to Kemal Ebcioglu and Eric Altman of IBM T. J. Watson Research Laboratory for their valuable discussions about the VLIW architecture. Their technical ability and insight across the entire spectrum of computer architecture are remarkable. I also would like to thank Alfons Hoogervorst, a friend I made on an internet newsgroup. He answered a lot of my questions about x86 protected-mode system programming.

Finally, I must thank my wife, Chiu-hua, and my children, Amy and Andy, for their love and support during this difficult time in graduate school. They are the ones who truly sacrificed to make my graduate studies possible. I also thank my mother-in-law, Yang, for helping to take care of our children in the past year. During the last year of my graduate studies, Chiu-hua had to return to Taiwan to resume her job. She works on week days in Taoyuan and takes the train to Chiayi on weekends to spend time with our children. Thanks, and we are eagerly awaiting again having a complete family!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software venders are unwilling to compile new software for a new processor until the marketplace of the processor becomes large enough. For example, when the Pentium Pro was introduced to the market, most new software was still compiled for the Pentium until the Pentium Pro market matured. As a result, utilization of most new hardware features is delayed.

Also, profiling and optimizing library and third-party code often require binary translation. The lifespan of such software is usually longer than hardware upon which it was originally developed. However, recompilation of the source code may not be possible.

Binary Translation is a technique used to transform one executable program generated for an old processor into an executable for another newer processor. This allows programs compiled for older machines to take advantage of all the new features that have since been added. Binary translation can help introduce new *Instruction Set Architecture* (ISA) features to the software base.

The objective of this thesis is to develop a solid foundation for binary translation. The most important problems solved in the thesis work are:

- Analyzing binary code

- Allowing code size to change
- Enabling partial translation, conservative mixed execution of original code and translated code
- Improving program load time
- Detecting self-modifying code
- Minimizing the overhead of translation

This thesis consists of 9 Chapters. Chapter 2 surveys some industry and academia related research work on binary translation. Industry primarily uses binary translation during processor development to facilitate the transformation of a code base developed for one architecture to the other architecture. Binary translation is also of academic interest and Chapter 2 explores some reasons why.

Chapter 3 introduces the framework of the binary translator presented in this thesis. Some background knowledge used throughout the thesis will be introduced in this chapter as well. Some of the background material needed throughout the thesis, such as executable file format and dynamic linking, will also be discussed in this chapter.

Chapter 4 discusses the front-end of the binary translator. The front-end deals with reading in a binary program, decoding it into an intermediate representation, and invoking the code translator. This chapter analyzes the binary code and performs control-flow analysis. In order to verify the decoding process, two essential components were implemented in the thesis: a binary code generator and a execution switching mechanism. The binary code generator produces binary code from the intermediate representation. After the translated binary code is generated, the execution switching mechanism redirects the program control to the translated code during execution.

After decoding the original binaries, some instrumentation code may be inserted. This code is used to make measurements, such as performance of the binary code. The binary code may also be optimized. In either case, the addresses of some instructions may have to be relocated due to the change in code size. Chapter 5 discusses this instruction relocation problem. Relocation of code can affect the branch target in the translated code. This target may or may not be known during translation. This thesis proposes a hybrid approach to resolving branch target problems. The hybrid approach resolves as many targets as possible statically, and uses a run-time support when the target cannot be resolved at translation time. In order to evaluate the run-time overhead of the proposed approach, this chapter presents a detailed analysis performed over some popular benchmark programs.

Self-modifying code is one of the major hurdles in binary translation, and by its nature is machine dependent. Since the x86 architecture was used to develop the binary translation framework in this thesis, Chapter 6 covers the x86 memory architecture description. This thesis implements a self-modifying code detector to prevent the binary translator from translating self-modifying code.

Writing the translated code to secondary storage as a new executable file marks the full translation cycle. Saving the translated code helps the translator avoid retranslating it when it is executed again. Some information dynamically maintained by the translator needs to be included in the new executable file. In order to make the new executable file work, the translator also needs to modify the original executable file when it creates the new file. Chapter 7 discusses some of these post translation issues.

The program load time is one of the most perceptible performance factors from the user's point of view. The segment loading pattern of a program can be profiled during development. However, the profiling data used in development may not match that of individual users. Chapter 8 addresses this problem, and provides a solution to improve the load time for an executable program. This solution is based on dynamic profiling of users workload. The translator uses the collected information to optimize the load time by modifying the segment attributes in the executable file.

Finally, Chapter 9 draws some conclusions about the work, presents the contribution of the thesis, and outlines a few research directions in the future from different aspects.

Chapter 2

Previous Work

Microprocessor technology has evolved quickly in recent years due to the improvement of VLSI technology. As a result, hardware systems have become out of date in just a few years. The lifetime of software, on the other hand, often outlives the system for which it was developed. To ensure that existing software can be run on a new machine, hardware manufacturers need to provide backward compatibility. Currently, compatibility is preserved by fixing the ISA. Preserving compatibility through the ISA, however, limits the performance improvement that the new hardware can have for old programs. Software compiled for one generation of hardware may not be able to take advantage of features present in the next generation.

This chapter surveys some of the related work on binary translation. Section 2.1 surveys the related work in industry and Section 2.2 surveys the related work in academia.

2.1 Related Work in Industry

Digital Equipment Corporation used binary translation to allow VAX code and MIPS code to be run on an Alpha AXP machine [27]. They created two translators: *VEST*, which translates OpenVMS binary images to OpenVMS AXP images, and *mx*, which translates ULTRIX MIPS images to DEC OSF/1 AXP images. They also ported the Open VMS operating system from the VAX architecture to the Alpha AXP architecture [13]. The migration tool contains two components: a binary translator and a run-time environment. The VEST translator performs backward symbolic execution [28] of VAX instructions to resolve as many computed branch targets as possible. A run-time lookup is used when more than one possible computed target exists. The run-time environment supports completely automatic translation by including a fallback interpreter of old code, and extensive run-time feedback to avoid using the interpreter except for dynamically created code. When the binary translator encounters a branch with a non-determinable target, it generates code to call lookup routines. The lookup routine maps an instruction address in the old architecture to a new address. If an address mapping exists, the control switches to the translated code for execution. Otherwise, the interpreter will be invoked to execute the destination code. The interpretation continues until a control flow change. There is no detailed report about run-time overhead in [27].

Recently, Digital Equipment Corporation introduced a product, *FX!32* [8] [30], which incorporated partial translation. FX!32 combines both interpretation and binary translation techniques to translate x86 Win32 applications on Windows NT Alpha. FX!32 consists of three interoperating components: a *run-time environment (runtime)*, a *binary translator*, and a *server*. The runtime contains an emulator that implements the entire x86 user-mode instruction set and the complete x86 Win32 environment. When the user runs an application program, Windows NT invokes the runtime. When the application is first executed, the runtime will emulate it. When the application is unloaded, the server looks for a new or enlarged profile. A new profile means that a previously unseen x86 image has been executed and may require optimization. An enlarged profile indicates that the runtime found more new code in the previous execution. In either case, the server invokes the background optimizer to translate the x86 code into Alpha code.

The binary translator implemented in this thesis differs from FX!32 in several aspects. First, FX!32 implemented the translation by modifying the NT loader. The binary translator in this thesis, on the contrary, implemented the translation on top of Windows. This difference affects almost all the implementations of the translation. Second, FX!32 redirects X86-based NT API calls to corresponding Alpha-based NT calls. Alpha derives most of the performance gain from the native compiled NT API source which is intensively used in Windows application programs. This thesis assumes no existing source code.

IBM proposed an experimental binary translator in order to port binary programs from their S/390 architecture to a *Very Long Instruction Word (VLIW)* architecture [26]. The architectural framework consists of the *migrant engine*, the *native engine*, and the *switching monitor*. A translator takes the base object code and produces the native object code to run on the native engine. The switch table is used during fallback from the native code to the base code or vice versa. This table maps groups of base instructions to groups of native instructions.

The execution of application is started by setting the migrant engine's program counter (mPC) to the entry address into base code. The switching monitor continually checks the mPC, looking for a match with one of the base code address entries in the switching table. If a match occurs, the corresponding native code address is loaded into the native program counter, or shadow program counter (sPC). The control is then switched to the native code for execution. The execution continues until the end of the translated code is reached. This may be a branch whose target cannot be determined at translation time. If there is no corresponding native code entry point for this target, the mPC is set to point at the base instruction which is the logical successor to the last instruction in the previously executed base code. [26] also proposed a hardware support for detecting self-modifying code.

DAISY [9] described some hardware features for a VLIW machine. It intended to emulate existing architectures, so that existing software, including the operating system, runs without changes on the VLIW machine. DAISY partitioned the memory into 3 sections. The low portion is mapped to the physical address space of the old (base) architecture. The middle portion consists of read-only virtual machine monitor software. The top section stores the translated code. Each address in the low portion is mapped to an address in the top portion. The code translation unit is a *page*. Several VLIW primitive branch instructions were also defined. The VLIW primitive for branches with a non-determinable target can be implemented either by hardware or by software. The software approach is similar to [15].

Tandem used binary translation for migrating software from the TNS (Tandem NonStop Series) CISC-based computer family onto the TNS/R (Tandem NonStop Series/RISC) computer family based on the MIPS RISC architecture [1]. The approach to resolving various puzzles about the unpredictable dynamic effects of the original CISC code is to make a best guess based on static analysis. If the guess turns out to be incorrect, the processor falls into interpretive execution mode for a short time This approach requires the presence of a CISC interpreter and all of the original CISC code for potential use by the interpreter. Tandem defined the points for entry to and recovery from interpreter mode. There is no clear description about how the switching between the translated RISC code and the original CISC code was implemented.

Motorola used PowerPC Migration Tools to enable a smooth transition of existing applications (guest) to the PowerPC architecture (target) [2]. Migration tools consist of 4 components: A *Front-End* (parser) that parses the object code or assembly code, a *Back-End* (code generator) that generates the target code, an *Interpretive*, which is a macro that interprets the instruction's semantics, and finally, the *Runtime support*, which provides the target machine with an execution environment for the guest code. There were four different configurations for the Migration tools. In the *Emulator* configuration, a binary parser decodes one instruction at a time and invokes a handler that uses "C" code to compute the semantics of the guest instruction. In the *Translator* configuration, an assembly parser reads in assembly (source) code and invokes a handler that constructs the intermediate data structures. These data structures are input to the subsequent code optimizations and generation. It takes one pass through the whole source code. In the *Translating Interpreter* configuration, the interpreted instruction can be directly generated from the translating interpreter without needing to interpret the same instruction when it is visited again. The *Binary Translator* configuration is similar to the *Translator* except that its input is the binary code. The paper did not give a detailed discussion of this configuration.

Sun Microsystems introduced *Wabi* to run Microsoft Windows applications on the Solaris desktop [29]. Three core Windows dynamic link libraries: USER.DLL, KERNEL.DLL, and GDI.DLL have their equivalent in Solaris. Without the need to emulate most of the operating system code, much of the performance loss can be avoided. Other Windows executables can have three options for translation. If the architectural platform is x86, the executable will pass through to the hardware without needing to emulate the instructions. If the platform is not x86, Wabi either interprets op-code by op-code or translates a block of instructions at a time. For the latter case, the translated instructions are stored in the translator cache. Loops in application code are translated once and executed multiple times.

2.2 Related Work in Academia

Larus and Ball [15] described the analysis of rewriting executable files for their profiling and tracing tool, *qpt*. In order to solve the non-determinable target problem, *qpt* uses a program's original code segment as a translation table to map from an address in the original program to addresses in the new program. A few instructions are inserted before a branch with an unknown target. These instructions compare the target of the branch at run-time. If the target turns out to be within the old code, the code dereferences the translation table to find the new target for the branch. In this thesis, the approach to resolving a non-determinable branch target is similar to that of *qpt*.

Wahbe *et al* [31] presented *adaptable binaries* for implementing robust binary transformation. They demonstrated that the information necessary to support adaptable binaries can be compactly recorded. However, they assumed that this necessary information was propagated from the compiler. This thesis makes no assumption on any rules or compilers used for generating the original executables.

Wahbe *et al* [31] also indicated several difficult problems in binary transformation. Among other things, there is one issue which is of particular interest to this thesis. [31] mentioned that there is no robust method for distinguishing code from data in the presence of indirect control transfers. Therefore, one solution to this problem, as implemented in Pixie [7], is to duplicate the code segment and instrument only the duplicated code. This thesis uses a different approach to solving this problem. For example, not all kinds of the hashing jump targets are difficult to resolve. Chapter 5 will give a more detailed discussion on this problem.

The *Morph Project* [4] was proposed to enable the evolution of executables by repartitioning the compilation process and by re-writing the executable. Morph consists of four software components. The *Morph back-end* generates executable programs with annotations that can be used by the editor to retarget the code. The *Morph editor* performs host-specific optimizations for the specific hardware platform. *PostMorph* analyzes existing executables and creates application-specific optimization templates for retargeting. The *Morph continuous monitoring system* generates profile data.

PostMorph uses conservative methods, such as single-step debugging or simulation, to trace the control flow and data reference activity. The output of this process is classified as safe, unsafe, and rarely used. Code marked as rarely used will not be optimized. Unsafe code may require manual transformation. Morph did not examine the issue on transforming unsafe code.

Etch [3] was proposed to evaluate and optimize the x86 application program. Etch is invoked with an executable and a Dynamic Link Library (DLL). The DLL includes a set of routines which are called for instructions in the executable. After scanning and instrumenting the executable, Etch writes a new executable that can be run. The new executable includes all referenced functions in the callback routines as well as the Etch runtime library. The instrumentation routines run as a side effect of running the new program. Lee *et al* [3] did not discuss the instruction address relocation problem that arises due to editing binaries.

Chapter 3

Background

3.1 Binary Translation Model

The framework proposed in this thesis is designed to handle x86 code. The original executable is a 16-bit Windows application program. The translated executable will contain instrumentation code or will be optimized code. Figure 3.1 illustrates the overall system.

The framework contains a front-end for reading in a binary program and decoding CISC instructions. This thesis will introduce the translator front-end in Chapter 4. It also contains an execution editor which performs insertion of instrumentation code or optimization. Editing binary program issues will be discussed in Chapter 5. The translator also includes a code generator that dynamically generates new binary code. After the new code is generated, the translator may run it and measure the performance via either hardware execution or simulation.

More specifically, this thesis addresses the issues involved in implementing a solid translation framework. Issues addressed include reading binaries, decoding, solving address relocation problems, enabling partial translation, supporting a run-time switching mechanism, and measuring the overhead of translation. At the final stage of translation, the translated code will be written to disk for reuse.

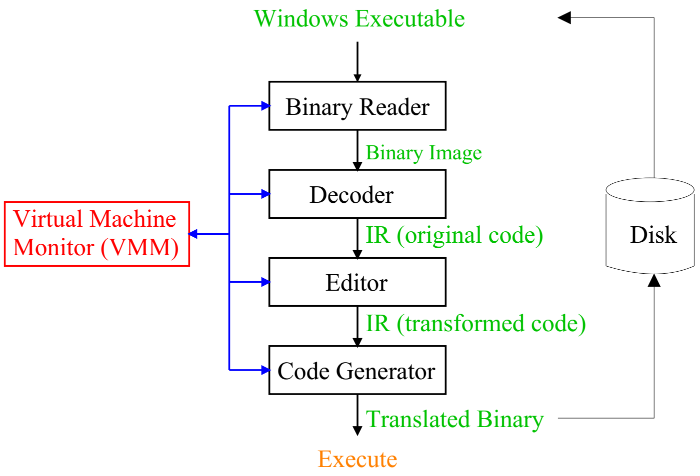

Figure 3.1 The binary translation framework.

3.2 Executable File Header

This section describes the 16-bit Windows executable file format. Since much of the work in the thesis was implemented based on the information given in the executable file, it is better to introduce the executable file format in this section.

The 16-bit Windows executable file is also called *New Executable*, or *NE*, as distinguished from old DOS executables. Instead of giving a comprehensive NE file description, this section only introduces those portions which are pertinent to the thesis. For a complete description, the reader may refer to [18]. Figure 3.2 illustrates the NE file format [18], with some modifications.

The NE file contains a header, followed by a few tables, then followed by each segment data information. Section 3.2.1 will introduce the NE header. Tables that will be used during translation will be introduced in Section 3.2.2. Segment data information will be discussed in Section 3.2.3.

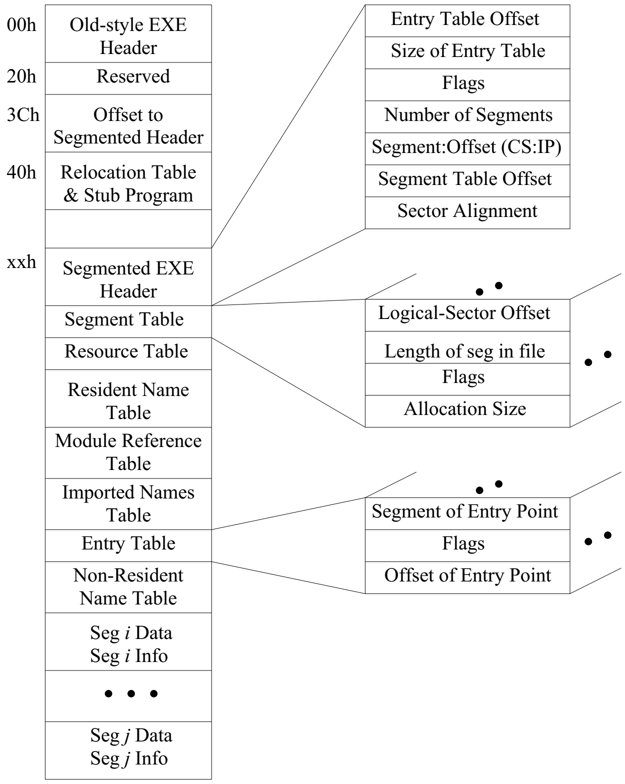

Figure 3.2 The NE file format.

3.2.1 The NE Header

The first two bytes in the NE header are the NE signature. The translator uses this signature to determine whether it is a valid application for translation. In the header there are also offsets to several tables such as the Entry Table, Segment Table, and Resource Table. The header also contains the number of entries in these tables. By using these offsets, the translator can easily move the file pointer to locate the tables for accessing information. The number of entries in the table usually helps the translator determine the boundary when accessing information from within the table. The NE header also contains the address of the first instruction to be executed. This address has the form of $segment{:}offset$ pair. Since the segment address cannot be known until the NE file is loaded into memory, the value in segment portion has the form of a logical ID.

At offset 32_{hex} to the NE header there are two bytes defining the *logical sector alignment shift count*. This word indicates the alignment size for the NE file. It will be used when the translator performs post translation discussed in Chapter 7. The default value for this count is 9. The offsets 37_{hex} through 3F_{hex} are reserved with current value 0's. The translator may utilize this space to store some valuable translation information. For example, the space may be used to store the file pointer to a new added area in the NE file, such as whether a segment was translated or not. Chapter 7 will discuss more details on these issues.

3.2.2 Tables in the NE File

One of the tables most frequently used by the translator is the *Segment Table*. The translator may use the segment table when a segment is loaded into memory for execution. It may also need to use the segment table when it writes the translated code to secondary storage. This

```
Value   Type            Description
-----   ---------       -------------------------------------
0007h = TYPE_MASK       Segment-type field
0000h = CODE            Code-segment type
0001h = DATA            Data-segment type
0010h = MOVEABLE        Segment is not fixed
0040h = PRELOAD         Segment will be preloaded
0100h = RELOCINFO       Set if segment has relocation records
F000h = DISCARD         Discard priority
```

Figure 3.3 The segment flag word in the NE file.

table contains an entry for each segment in the executable file [18]. There are 4 words in each entry. The first word defines the logical-sector offset to the per-segment data, relative to the beginning of the file. To locate the per-segment data, this number is left shifted by the value of logical sector alignment shift count described in Section 3.2.1. The second word defines the length of the segment in the file. The third word defines the flags for the segment. As the flags defined in the third word are used in almost all the remaining chapters, this section gives a full description for these flags. The fourth word in the segment entry defines the minimal allocation size for the segment. In most code segments, this value will be identical to the segment length.

Figure 3.3 defines all the possible attributes for each segment [18]. Each segment in the NE file can be either $CODE$ or $DATA$, but not both. The $PRELOAD$ attribute indicates whether or not the loader should load the segment before the program begins execution. The $MOVEABLE$ attribute indicates whether or not the operating system can move it in linear address space. The $DISCARD$ attribute indicates whether or not Windows may discard it from memory if there is not enough space. If there are several target locations referenced by the segment, and these locations cannot be determined before run-time, then this segment will contain the relocation information for these targets. The *RELOCINFO* attribute indicates whether or not the segment contains any relocation information.

The *Resource Table* contains the entries for each resource used in the executable file. In Windows, a resource can be an icon, cursor, mouse, or others. Although the translator only translates code, getting information about the resource table can be crucial as far as post translation and program reuse is concerned. Modifying the original executable file implies that the offset of resource in the file may be changed as well. By carefully adjusting the offsets, the translated executable file can be executed without error. Chapter 7 will discuss the modification of the Resource Table after translation.

The *Entry Table* contains bundles of entry-point definitions. When a function in the segment is to be exported, there must be an entry in the entry table. Because Windows is a dynamic linking environment, the callers of this exported function cannot know the final address of this function until the segment is loaded and the address is resolved by the loader. The information in this table will be used both during translation and after translation.

3.2.3 Per-Segment Data Information

The contents of each segment can be indexed by a combination of both the logical sector from the entry of the segment table and an alignment shift count from the header. If the segment has relocation fixups, as indicated in the segment table entry flags words, these fixup records immediately follow the content. The addresses of exported functions in an application program cannot be known before the program is loaded. Therefore, the program may call some functions whose addresses may not be immediately available. One example is that most Windows application programs may call Windows *Application Program Interface (API)* functions.

The target addresses for these API functions will be fixed when the application is loaded into memory.

If the operand of an instruction references a target that cannot be resolved before loading, then there is an entry containing the address of this operand. The translator does not need to know the values for these fixup records. However, it may need to know the address of each operand which needs a fixup record. If the translator modifies the content of the original segment and if such modification results in changes of the offsets of some instructions, then it needs to adjust the location for these operands, so that an operand of translated code can be correctly fixed when it is loaded into memory again. These issues are discussed in more detail in Chapter 5 and Chapter 7.

3.3 Dynamic Linking

One feature of the Windows operating system is that all executable programs, including applications and the operating system itself, are dynamically linked before execution. As opposed to static linking in which the linker links together all the object code referenced into an executable at compile time, dynamic linking is a method in which the program loader links the modules at run-time.

To illustrate the merit of dynamic linking, let us look at an example that compares the dynamic linking with static linking. In Figure 3.4, the application program contains a library function call, $printf()$. In static linking, the linker inserts the object code of $printf()$ into the object code of the program $foo.c$ before it generates the executable file. For example, assuming there are 100 user programs executing simultaneously with each one calling $printf()$, then there are 100 instances of $printf()$ loaded into main memory. With dynamic linking, on the other

```c
#include <stdio.h>
int main(void) {
    /* do something, and then call system library function */
    printf("Hello world\n");
}
```

Figure 3.4 A source program written in C.

hand, only one instance of $printf()$ needs to be loaded into main memory. As a result, dynamic linking utilizes memory better than static linking.

The translator implemented in this thesis performs dynamic translation for the binary program, which means that the translation of a code segment begins after the loader loads this segment. As such, most targets that cannot be determined statically have been resolved by the loader during dynamic linking. However, even after loading, not all of the targets can be found until the program actually runs. Chapter 5 will address this problem.

For a more detailed introduction to dynamic linking and dynamic link library, the reader may refer to [16] [17].

Chapter 4

Binary Front-End

The frond-end of the binary translator consists of reading in binary code and decoding it to an intermediate representation. This chapter discusses these issues in Section 4.1 and Section 4.2 respectively. Verifying the decoding process is also important to the preliminary stage of translation. This chapter addresses this problem in Section 4.3. The execution of translated code will be discussed in Section 4.4.

4.1 Parsing Binaries

Parsing the executable program can be done statically or dynamically. There are a few differences between these two approaches. For each executable file, there is a complete file header describing all information about the program. When the executable program is loaded into main memory for execution, some information will be discarded. One example is that the relocation fixup record associated with the segment is discarded after the loader loads the program into main memory. On the other hand, the instruction in an executable program stored in the secondary storage may contain unsolved external addresses. This is because in Windows all the libraries are dynamically loaded.

This thesis focuses on reading binaries from main memory. Translating the code when it is loaded into memory provides us with more accurate information. For example, most unknown far targets will be available before the translator starts to translate the instruction. Moreover, since not all of the code segments will be executed, performing dynamic translation guarantees that only code that is being translated will be executed. Put another way, dynamic translation supports on-demand translation.

In Windows, each segment in the executable program can be dynamically loaded into main memory. To intercept the dynamically loaded segment, the translator installs a notification callback function before the application program starts execution. A *callback function* is a function that will be called by Windows. Among other notifications, the translator is particularly concerned with the following messages sent by Windows:

(1) When Windows loads a segment into memory

(2) When Windows frees a segment from memory

(3) When Windows is about to start a task, and

(4) When Windows is about to exit a task.

The translator maintains an internal data structure for keeping track of each in-memory segment. Figure 4.1 illustrates the data structure implemented in the translator. Upon receiving message (1) from Windows, a $NFY\_LOADSEG$ data structure will be passed to the translator. This structure includes six attributes: $Size$, $Selector$, $SegNum$, $Type$, $Instance$, and $ModuleName$. These attributes represent the size allocated for the segment, the selector to the segment when the segment is loaded into memory, the segment's logical segment ID in the NE file, the segment type, the number of instances that share this segment, and the name of its owner. The selector of the segment will be defined in Section 6.1. The size, the logical ID, and the type of segment were previously described in Section 3.2. Most importantly, the data structure includes the selector's value to the segment. The selector value uniquely identifies the segment in memory. This value plays a central role in the switching mechanism. The $DWORD$ in Figure 4.1 stands for double word which is a 4-byte unsigned integer. When the translator receives message (1) from Windows, it first checks whether the segment is code or data. If it is a code segment, it needs to check whether the segment has been translated or not. It is likely that a code segment was loaded into memory by the loader, translated by the translator, unloaded from memory by the loader, then loaded again by the loader. In this case, the translator can avoid retranslation by simply checking the $Translated$ bit for the segment. The $Present$ bit in Figure 4.1 indicates whether or not the translated code for the original code segment is in memory. The $NewSelector$ in Figure 4.1 is the selector to the translated code, provided that the code is in memory.

When the translator receives message (2) from Windows, it updates the segment data structure. The $Present$ bit will be cleared. Any reference to a segment whose $Present$ bit is cleared will be considered as an invalid access.

When the user starts an application program, Windows first loads all of the $PRELOAD$ segments into memory for the program, then starts execution. After loading all $PRELOAD$ segments and before the execution is about to start, Windows will send message (3) to the translator. Upon receiving this message, the translator obtains the program's initial program counter with a $selector : offset$ pair, or CS:IP. After translation, the translator will redirect the execution flow from this CS:IP value to its new counterpart. The translator needs the new

```
typedef struct SegNodeTypeTag {
    /* NFY_LOADSEG data structure */
    DWORD     Size;
    WORD      Selector;
    WORD      SegNum;
    WORD      Type;
    WORD      Instance;
    LPCSTR    ModuleName;

    /* New code segment information */
    BYTE      Translated;      /* Translation bit of the segment */
    BYTE      Present;         /* Presence bit of the segment    */
    WORD      NewSelector;     /* Pointer to the new code seg    */

    /* Info for control flow analysis */
    OPER_TYPE *head_op, *tail_op;
    DWORD     index;
    MEM_TYPE  code_map[FLOW_MEM_CHUNK_MAP_SIZE];
    MEM_TYPE  fallthru_stack[MAX_STACK_SIZE+1];
    DWORD     top_fallthru_stack;

    struct    SegNodeTypeTag   *nextSeg;
} SegNodeType;
```

Figure 4.1 Internal data structure for in-memory segments.

CS:IP not only to start executing the new code, but also to generate the new executable file when the translation is completed. Chapter 7 will discuss this point.

When the running program is about to exit, the translator will receive message (4) from Windows. After receiving this message, the translator can perform post translation task such as writing the translated code to the secondary storage.

4.2 Decoding CISC Instructions

4.2.1 The Decoding Process

After intercepting binary code from memory, the next step is to decode it into an *Intermediate Representation (IR)*. Decoding is basically a table lookup process in that the translator uses the hexadecimal value to determine the operation code. Since the length of an x86 instruction is variable, the translator may need to decode additional binaries such as addressing mode in order to determine the instruction length.

There are several options in decoding the intercepted binaries. First, the translator can decode one instruction at a time, translate it into new operations, and execute these operations. This is actually an interpretation approach. Interpretation has one advantage in that the translator will never translate data into code. However, in general interpretation is very slow [8] [10].

The second option is to decode the whole intercepted code segment, perhaps one function at a time. Decoding the whole segment provides us more opportunities for optimizing the new code. However, it is likely that the translator may decode data or patched bytes into instructions. There are two approaches to solving this problem. First, the translator may interpret the instruction the first time it parses the original code, or whenever the untranslated code is found. Second, the translator may perform a quick control-flow analysis over the code segment before decoding it. For example, unless an address is the target of some branch instructions, the address that immediately follows a non-conditional branch will not be decoded. In this thesis, the decoding process is guided by control-flow analysis which will be discussed in Section 4.2.2.

4.2.2 Control-Flow Analysis

Given a dynamically-loaded code segment and a starting instruction pointer, the control-flow analyzer starts decoding instructions without actually running them. If the instruction being decoded is not a branch, then the analyzer only needs to advance the program counter to the next instruction, provided that the program counter does not exceed the length of segment. The starting instruction pointer can be either the initial IP, or any entry point in the segment. Section 4.4 will discuss how the translator gets the instruction pointer before the control-flow analyzer is invoked. In order to find as many instructions as possible, the control-flow analyzer may need to iterate several times until all the entry points that belong to this segment are visited.

During analysis, the translator maintains some information in its internal segment data structure. Control-flow information is included in the data structure illustrated in Figure 4.1. The variables *headOp* and *tailOp* in Figure 4.1 are used to maintain a list of operations in the segment. The attribute *index* keeps track of the total contiguous instruction areas visited for the segment. The attribute *code_map* defines the lower and upper boundaries for the each contiguous instruction area. Both the attributes $fallthru\_stack$ and *top_fallthru_stack* are used when the decoded instruction is a conditional branch or a function call. If the translator encounters a conditional branch or a function call, it will first save the fallthrough target to a stack, and visit the taken target of conditional branch. If the translator cannot decode further, it then pops the fallthrough target and keeps decoding, until the fallthrough stack is empty. As soon as any adjacent instruction areas are found, the translator merges them to form a larger single area.

The control-flow analysis module built in the thesis is a conservative approach in that all the code area it detects must be instructions. While a conservative approach is needed to guarantee correct translation, sometimes the results may be weak. To remedy this problem, the translator must be able to support incremental translation which will be discussed in more detail in Section 5.3.

4.3 Verifying the Decoding Process

After the decoding process, the translator may need to verify the output code before it can be used for further transformation. The verification of the decoding process is difficult in general, due to the irreversibility of compilation. One might use a commercial disassembler, such as Sourcer [6], to help verify the decoding process. The translator may generate assembly code from the IR, and compare the result with the assembly code generated by Sourcer. However, this is not a reliable way to do verification. The assembly code generated by Sourcer may not be accurate. In our approach to debugging the decoding process, the translator also contains a binary code generator which translates the IR back to binaries for execution. This step bypasses the editing process, thus helping to distinguish the bugs between the decoding process and the editing process. The dashed arrow in Figure 4.2 illustrates the verification process.

The binary code generator is similar to an assembler. The main difference is that the input to the generator is not assembly code, but the IR. One could use a commercial assembler to

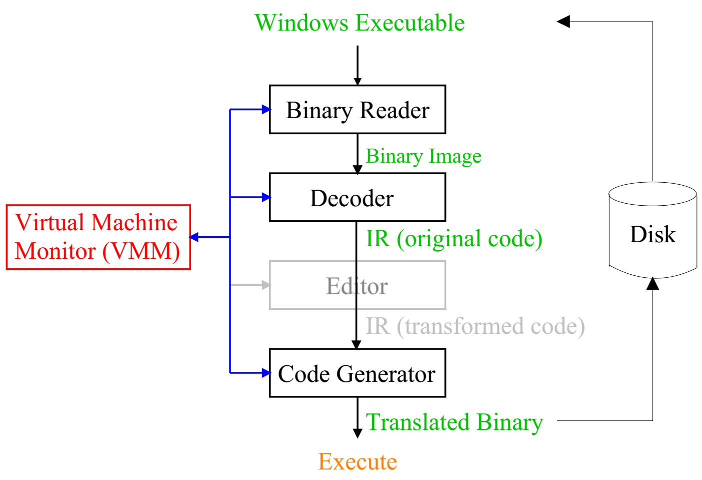

Figure 4.2 Verifying the decoding process.

generate the new binary code. However, there are two major concerns with this method. First, the input to the assembler must be assembly code. Sticking to a specific assembly representation usually is not a good idea for later optimization. Second, the binary code is generated from the IR after the executable program was loaded into memory for execution. Relying on an assembler usually takes more time to produce new code and sometimes the generated code must be modified to conform to the specific assembler syntax. For these reasons, this system bypasses using an assembler to generate binary code.

Since some operations share the same pattern of addressing modes, the code generator uses several templates to generate binary code. For example, Figure 4.3 shows that there are 8 x86 operation codes that share the same pattern of 4 addressing modes. The code generator may group them together and use a template to generate the binary code.

In addition to fully generating the x86 code from the IR, a slim version of a code generator was also implemented to accelerate the develop time. In this version, the original binary code is copied to the new code directly from the IR, provided that the instruction has not been

```
   Operations:             Description:
       ADC                 Add with Carry
       ADD                 Add
       AND                 Logical AND
       CMP                 Compare Two Operands
       OR                  Logical Inclusive OR
       SBB                 Integer Subtraction with Borrow
       SUB                 Integer Subtraction
       XOR                 Logical Exclusive OR

Addressing Modes:
      Destination          Source
      {AL,AX,EAX},         {imm8,imm16,imm32}
      {r/m8,r/m16,r/m32},  {imm8,imm16,imm32}
      {r/m8,r/m16,r/m32},  {r8r16,r32}
      {r8,r16,r32},        {r/m8,r/m16,r/m32}

imm:      immediate value
r/m:      either register or memory operand
r:        register
8/16/32: size of operand, in byte
```

Figure 4.3 Operations that share the same addressing mode.

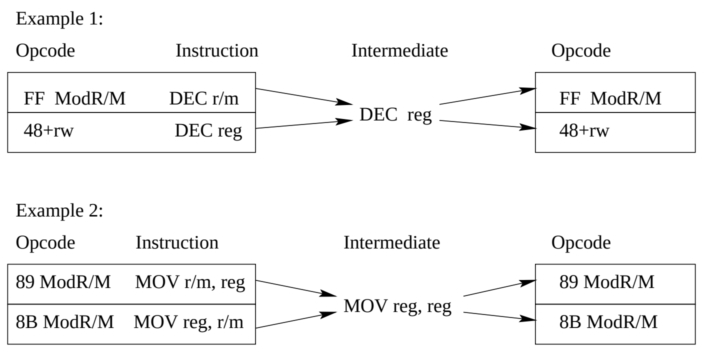

Figure 4.4 Ambiguous situations in generating new binary code.

changed. For each IR, the hexadecimal code is also stored as soon as the instruction length is determined during decoding. The hexadecimal code is copied to the new code during code generation.

Since it is likely to have more than one binary code corresponding to an x86 assembly instruction, generating new code from decoded binaries may result in ambiguity. Consider Figure 4.4 for examples. In Example 1, there can be two binary codes for the decrement register instruction. The first encoding contains two bytes: FF and a $ModR/M$ byte which determines whether the operand is memory or register. The second encoding contains only one byte since the register value is encoded as part of the operation code. Special care must be taken during code generation, since different sizes of instructions may affect the relative addresses of other instructions. This is particularly true after performing optimizations on the new code. In each IR, instruction length as well as hexadecimal code are stored. Either information can be used to determine the encoding. In Example 2, the assembly instruction moves the value of one register to the other register. This assembly may have two different binary codes that perform the identical operation. This case, however, does not affect the result since the length of these two encodings are identical.

```
i = logical segment ID;
allocate a memory space, Mem[i];
translate Seg[i] into Mem[i];
make Mem[i] executable;
obtain the selector, Sel[i], for Mem[i];
Ofs[i] = new IP;
replace the initial CS:IP with Sel[i]:Ofs[i];

return control to operating system;
```

Figure 4.5 An algorithm for translating and executing the new segment.

Another way to verify the decoding process is for the binary code generator to generate a binary image to secondary storage, and compare the output file with the one generated by commercial product. For example, the *TDUMP* command associated with *Turbo Debugger* can be used to dump the specific binary image of an executable program. This can then be compared with the output generated by the binary code generator.

4.4 Executing the Translated Code

After the intercepted code is translated, the next step is to execute the new code. At the beginning, this code will start execution from its new initial program counter. Figure 4.5 shows the algorithm for redirecting execution to the new initial program counter.

The logical segment ID in Figure 4.5 can be obtained from the internal segment data structure described in Figure 4.1. First, the translator requests a memory space from the operating system for storing the new code, then starts translation. Since allocated memory cannot be executed, the translator needs to change the access right for allocated memory before it can be executed. At the last stage, the translator replaces the original $CS{:}IP$ value with its new counterpart and returns control back to the operating system. The new initial code segment

```
for each loaded segment {
    find all entry points which belong to this segment;
    for ( j=1; j<=Entry_Found; j++ ) {
        backup the byte content MEM[OFFSET(j)]
        mask the byte content MEM[OFFSET(j)] with a breakpoint;
    }
}

return control to operating system;
```

Figure 4.6 An algorithm for trapping each loaded segment.

address, or $Sel[i]$, is available as soon as the translator makes the memory executable. The new initial offset, or $Ofs[i]$, is fully under the control of the translator. When the new code was modified, $Ofs[i]$ may or may not be the same as IP. The translator is responsible for keeping track of such change.

In addition to redirecting execution to the new initial program counter, the translator also needs to set up some extra traps when each code segment is loaded into memory in order to capture more instructions later. Figure 4.6 shows the algorithm used by the translator during segment loading.

The entry points in Figure 4.6 can be found in the module table of an application program. For each loaded code segment, the translator will replace the byte content at all entry points with breakpoints if the segment contains any of them. The $OFFSET(j)$ in Figure 4.6 stands for the offset of an entry point, j, to the beginning of the segment. $MEM[OFFSET(j)]$ stands for the byte content in address $OFFSET(j)$. As mentioned previously, the translator can obtain the program's initial program counter from Windows. Starting from this initial program counter, the translator can decode and analyze the binaries. If any instruction in the initial segment branches to another segment, the translator must detect such a branch, or this target

```
foo.c
foo( ) {
    func_1( );
    func_2( );
}
```

```
bar.c
func_1( ) {
}
func_2( ) {
}
```

Figure 4.7 Switching mechanism: a code example.

segment could never get a chance to be translated. As such, the translator sets up breakpoints at all entry points in the segment as traps. Whenever the target in the segment is trapped, the translator may then translate the segment.

Let us consider the code example in Figure 4.7 to demonstrate how the switching mechanism works. The source program contains two files, $foo.c$ and $bar.c$. $Foo.c$ contains calls to functions defined in $bar.c$. If the program was compiled by using a large memory model, the compiler will generate far calls for the calling instructions in $foo.c$. Figure 4.8 shows the executable file for the source program. In Figure 4.8, the function $foo()$ was stored in segment 1, the functions $func\_1()$ and $func\_2()$ were stored in segment 2. Since the address of segment 2 was not available at compile-time, the compiler generated 2 entry points in the executable header, one for $func\_1()$ and the other for $func\_2()$.

Figure 4.9 shows the code image after the executable file is loaded into memory. After segment 2 is loaded, its address is available. Now the loader fills this address in the target of calling instructions in segment 1. Figure 4.10 shows the graphical description of how the translator detects that $func\_1()$ in segment 2 is executed. The left column in Figure 4.10 shows the in-memory code image. The middle column shows the in-memory code image after the translator sets up the traps. The right column shows the execution flow. After the trap is

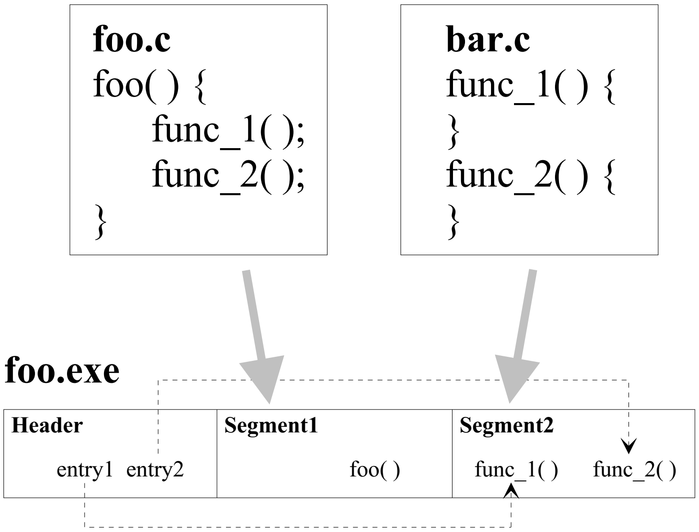

Figure 4.8 Switching mechanism: the executable file.

received, the translator starts translating segment 2 and redirects the execution flow to the new code. The black arrows in the right column indicate that the call to $func\_1()$ branches to a trap, then the translator redirects the branch to $func\_1'()$.

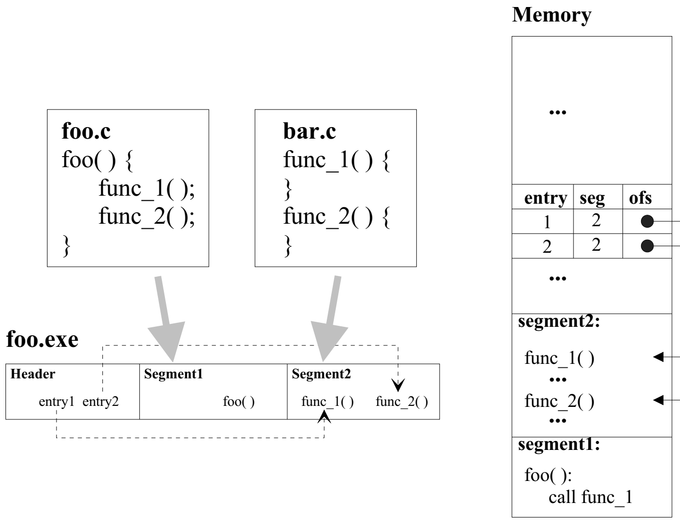

Figure 4.9 Switching mechanism: the code in memory.

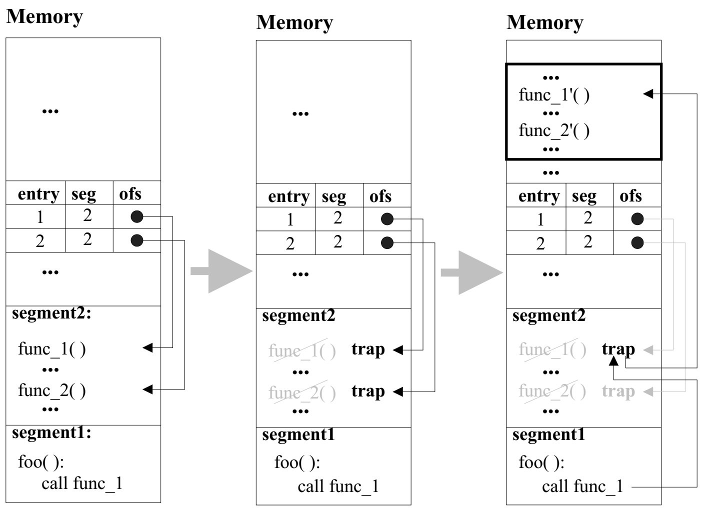

Figure 4.10 Switching mechanism: trapping and redirecting execution flow.

Chapter 5

Editing Binaries

Instrumenting code or performing optimization requires editing the code and may result in code size changes. This chapter discusses the instruction relocation problem resulting from this change of code size. Section 5.1 discusses adjusting the branch target whose value can be determined at translation time. A method is proposed in Section 5.2 to solve the relocation problem in situations where the branch target cannot be determined at translation time. Since the translator needs run-time support to resolve the branch target, in this case the impact to performance is also analyzed in this section. Because of the fact that the translator may not be able to detect all possible instructions during initial translation, it may be able to find more instructions as soon as the undetermined targets discussed in Section 5.2 are resolved at run-time. Section 5.3 discusses the issue of enabling incremental translation.

5.1 Relocation for Determinable Targets

If the target of a branch can be determined during translation, then the translator is able to solve all the relocation problems resulting from editing. A branch instruction with a statically identifiable target can be either intra-segment or inter-segment. The solution to intra-segment instruction relocation differs from that of inter-segment. These problems will be discussed in

Section 5.1.1 and 5.1.2, respectively. The callback function and the regular hashing jump are also determinable. Section 5.1.3 discusses the problem of editing a code segment that contains callback functions, and hashing jumps with regular patterns will be discussed in Section 5.1.4 Some DLL function calls are compiled as indirect calls. Indirect DLL function calls can be statically determined and will be discussed in Section 5.1.5.

5.1.1 Intra-Segment Targets

Figure 5.1 shows how to adjust a branch target after insertion or deletion. All the targets of branches in Figure 5.1 are PC-relative. Though there are intra-segment branches whose targets are not PC-relative, this solution still applies. All of the intra-segment targets are relative to the beginning of segment if they are not PC-relative. For those grey-arrow branches indicated in the figure, the translator does not need to adjust the target. For example, assume the user inserted a few bytes at location 3000_{hex}, the locations for those instructions after 3000_{hex} will be changed accordingly, but not for the instructions whose locations are prior to 3000_{hex}. Now if there is a branch from location 1000_{hex} to location 2000_{hex}, or vice versa, then no adjustment needs to be made. Likewise, if there is a branch from location 4000_{hex} to 5000_{hex}, or vice versa, no adjustment needs to be made either. Assume the user inserted 100_{hex} bytes at location 3000_{hex}, the location of 4000_{hex} and 5000_{hex} will become 4100_{hex} and 5100_{hex}, respectively. The net difference does not vary.

If the source and destination of the branch straddle the insertion or deletion line, however, adjustment may need to be made. These situations are indicated by the black-arrow branches in Figure 5.1. Assume the user inserted 100_{hex} bytes starting from location 3000_{hex} and there is a branch from 2000_{hex} to 4000_{hex}. Also assume the branch is PC-relative and the branch

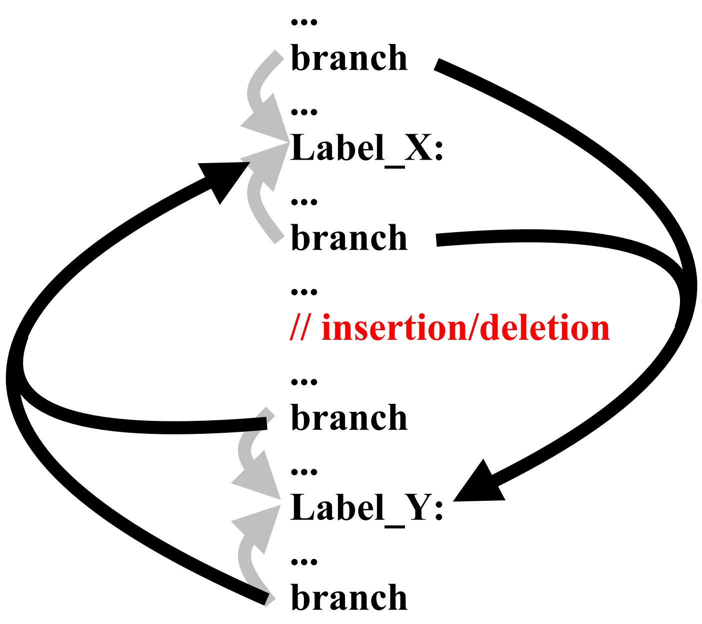

Figure 5.1 Adjusting the branch target.

instruction takes 4 bytes. If the insertion was not made, the value of operand in the branch instruction would be $1FFC_{hex}$ (4000_{hex} - 2004_{hex}). Now if the user inserts 100_{hex} bytes in between these locations, the value of operand should be added by 100_{hex} as well, that is, $20FC_{hex}$.

5.1.2 Inter-Segment Targets

Adjusting determinable inter-segment branch targets is slightly more complicated than intra-segment, but not difficult. When a code segment is loaded into memory, the loader needs to fix up all the branch targets in other segments if these targets reference the current code segment. Likewise, changing the location of instructions in the current code segment may require the translator to adjust these targets accordingly.

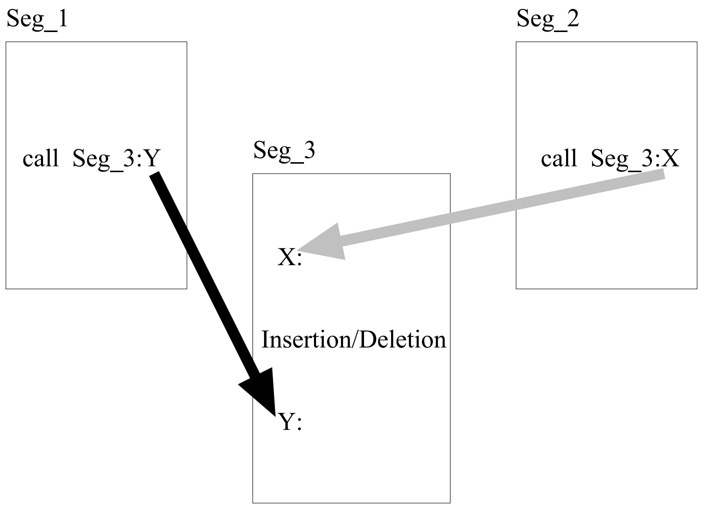

Figure 5.2 Adjusting the inter-segment branch target.

Figure 5.2 shows two cases for the inter-segment targets. The black arrow in Figure 5.2 indicates that the operand value in the calling instruction must be modified. The insertion or deletion in *Seg_3* results in the change of offset Y. This change, in turn, affects the operand (target Y) of the calling instruction in *Seg_1*. On the other hand, the insertion or deletion in *Seg_3* does not affect the offset X. As such, no adjustment needs to be made for the calling instruction in *Seg_2*.

After editing a code segment, the translator needs to iterate through all other present code segments. During iteration, the translator looks up all the relocation fixup record to find if the segment of the fixup records matches the current code segment. A detailed description of relocation fixup records may be found in Section 3.2.3. If the segment that belongs to the fixup record matches the editing segment, and the offset of fixup record is larger than an insertion or deletion point, then the translator will replace the operand with a new value. This value can be calculated during editing.

5.1.3 Callback Functions

Some functions in application programs can be declared as call-back. The description of a callback function may be found in Section 4.1. The address of a callback function may also be determinable. If a code segment contains any callback function and this segment is translated, then its original segment must reside in memory as well. Section 4.4 mentioned that when a code segment is loaded into memory, the translator sets up a trap for each entry point in the original segment. When the user sends a command, for example, by pushing a window button, the operating system sends a message to the application program in response to user's action, which means that the operating system calls back the function in the application program. Assume there is an operating system instruction I which calls back the function in the application program. After the application program is loaded into memory, the loader has filled the operand of I with the address of the callback function. This address is an old address. Even if the translator translated the callback function and created a new address for the callback function, the instruction I will still reference the old address unless the translator can also modify the operand of I.

Therefore, the translator relies on the trap in the original segment to redirect the execution. As soon as the trap is received, the translator looks up the address mapping table which will be defined in Section 5.2.2, and switches the control to the new code.

As opposed to directly overwriting the offset as previous approaches did, the new address of the callback function was not overwritten by the translator at run-time. After the translation is finished, this information will then be written to disk when the translator creates a new executable file. Writing the new code to secondary storage will be discussed in Chapter 7. For now, it suffices to say that the old code segment is not required to be present when the translated program is run again. All the corresponding entry addresses in the code segment will be updated when the translator generates the new executable file. When the program is loaded again, the address of the callback function will have already become the new one.

5.1.4 Hashing Jumps with Regular Patterns

A hashing jump is another branch whose targets may be determinable. The compiler usually generates a hashing jump based on a regular pattern. Several programs were analyzed with four patterns of hashing jumps summarized in Figure 5.3. All the hashing jumps in Figure 5.3 are intra-segment and indirect. The intra-segment jump implies that the size of the target specifier is 2 bytes. The indirect jump implies that the operand of the jump instruction is not the target, but rather, the offset to the target. All the hashing jump patterns in Figure 5.3 examine the number of offsets before accessing the eligible indirect target. This is accomplished by a compare instruction (*cmp*) and a conditional jump instruction (*ja* or *jbe*). For example, the value 7 in Pattern 1 of Figure 5.3 indicates that there are 8 possible indirect targets for the branch, ranging from 0 to 7. The "shift left (*shl*)" instruction or "*add*" instruction Figure 5.3 converts the byte index (1 byte) to the word index (2 bytes). Since register *ax* is used for computation and register *bx* is used as the base register, the "exchange (*xchg*)" instruction is introduced before the indirect jump instruction.

If the hashing jump was generated based on these regular patterns, the translator is able to resolve all of the hashing jump's possible targets. This is done by going to the offset of the indirect targets (for example, *ofs_0856* in Pattern 1), and fetching the values accordingly.

```
Pattern 1: (from Microsoft Paint Brush)
        cmp     ax,7                       ; set range: 0 - 7
        ja      short ofs_3462             ; punt if range > 7
        shl     ax,1                       ; double the index range, BYTE->WORD
        xchg    bx,ax                      ; bx serves as base
        jmp     word ptr cs:ofs_0865[bx]   ; base indirect jump

Pattern 2: (from Microsoft Excel)
        cmp     ax,0Ch
        ja      ofs_1148
        add     ax,ax                      ; another way to double the index
        xchg    bx,ax                      ; range
        jmp     word ptr cs:ofs_1121[bx]

Pattern 3: (from QVT Terminal)
        mov     ax,si                      ; another way to double the index
        cmp     ax,7                       ; range, si can be other register
        ja      short ofs_5026             ; as well
        add     ax,si
        xchg    bx,ax
        jmp     word ptr cs:ofs_0866[bx]

Pattern 4: (from Microsoft Calculator)
        cmp     ax,0Ch
        jbe     short ofs_0977             ; another way to skip if the
        jmp     ofs_1044                   ; range is beyond the boundary
    ofs_0977:
        shl     ax,1
        xchg    bx,ax
        jmp     word ptr cs:ofs_0367[bx]
```

Figure 5.3 Hashing jumps with regular patterns.

```
Lib_Instance = LoadLibrary("Library_Name");
if ( Lib_Instance > 32 ) {
    Func_Pointer = GetProcAddress(Lib_Instance, "Function_Name");
    ( *Func_Pointer )();
}
else {
    ErrorHandler();
}
```

Figure 5.4 Example of calling a dynamically loaded function.

5.1.5 Indirect DLL Function Calls

Application programs may dynamically request the operating system to load a library function and then call the loaded function. In Windows, this can be achieved by two system API function calls: *LoadLibrary*(), which requests Windows to load a specific dynamic link library, and *GetProcAddress*(), which obtains the address of the function in the loaded library. Figure 5.4 illustrates a code example for calling the dynamically loaded function. Upon returning from *LoadLibrary*(), the operating system tells the programmer whether or not the library is successfully loaded, and, if the library is not successfully loaded, the reason why. In Figure 5.4, the number 32 indicates that the library is successfully loaded if the return value is greater than 32.

Calling a dynamically loaded function is usually compiled as an indirect call. Figure 5.5 illustrates the assembly code found in Microsoft Word. At first glance, the indirect call instruction at line 14 seems unable to be determined since the content in *data_0262e* is not defined until the call to *GetProcAddress*() at line 8 is executed. Since the translator is able to determine whether the dynamically loaded function is translated or not, the target of this type of branches is determinable.

```
(1)     call    far ptr LoadLibrary           ; call LoadLibrary()
(2)     mov     ds:data_0009e,ax
(3)     cmp     ax,20h                        ; if return value greater than 32
(4)     jbe     short loc_2781
(5)     push    ax
(6)     push    cs
(7)     push    0C7h
(8)     call    far ptr GetProcAddress        ; obtain an address to function
(9)     mov     ds:data_0262e,ax              ; address is stored in
(10)    mov     word ptr ds:data_0262e+2,dx   ; data_0262e
(11)    mov     ax,dx
(12)    or      ax,ds:data_0262e
(13)    jz      short loc_2781
(14)    call    dword ptr ds:data_0262e       ; then call data_0262e
```

Figure 5.5 Indirect call to the dynamically loaded function (from Microsoft Word).

5.2 Relocation for Non-Determinable Target

Sometimes a code segment may contain some branches whose targets are prohibitively expensive to analyze on the fly. A code segment may also contain some branches whose targets cannot be determined without running the code. Section 5.2.1 will describe these cases. Section 5.2.2 proposes a run-time mechanism to solve this problem, and analyzes the occurrence for these instructions.

5.2.1 Non-Determinable Branch Targets

Not all of the hashing jump patterns can be easily analyzed. Figure 5.6 shows some irregular hashing jump patterns. These patterns are found in *QVT Terminal Emulator*. The common situation with these patterns is that the range of the hashing jump cannot be easily determined. Determining the range often demands a complicated calculation. There is no general rule to outline a calculation pattern. For example, the value 3 in a right-shift instruction (*shr*) in

```
Pattern 5: (from QVT Terminal)
        and     al,0F0h
        sub     ax,0F000h
        test    al,0Fh
        jnz     ofs_0653
        shr     ax,3
        cmp     ax,26h
        ja      ofs_0653
        xchg    bx,ax
        jmp     word ptr cs:ofs_0593[bx]

Pattern 6: (from QVT Terminal)
        mov     cl,3
        shl     al,cl
        add     al,[bp-7]
        xlat
        inc     cl
        shr     al,cl
        mov     [bp-7],al
        cbw
        mov     bx,ax
        shl     bx,1
        jmp     word ptr cs:ofs_0636[bx]
```

Figure 5.6 Hashing jumps with irregular patterns.

Pattern 5 can be easily changed, depending on the compiler or assembly code writers. In this pattern, the potential targets that follow the hashing jump consist of 95 bytes. This is an odd number, which conflicts with the fact that the total byte size should be an even number for the hashing jump, since the "word ptr" cast expected a 2-byte target. For Pattern 6, determining the range may also require the translator to perform a tedious back trace and complicated calculation. It is even more difficult for the translator to automatically resolve the target if the old assembly code was hand optimized.

Figure 5.7 illustrates an example in which the target of the branch cannot be determined until the program is executed. The piece of code was detected in Microsoft Calculator. Before

```
...                                  ; [ds:ofs_04C8] contains value 0
call    far ptr INITTASK             ; This Windows system API sets
...                                  ; [ds:ofs_04C8] as side effect.
jmp     word ptr ds:ofs_04C8         ; Now [ds:ofs_04C8] contains 004Bh
```

Figure 5.7 A dynamically determined branch target.

```
call    sreg:[reg+disp]          ; indirect target is in memory
call    reg                      ; indirect target is in register

jmp     word ptr sreg:[reg+disp] ; not including regular hashing jumps
jmp     reg

sreg: segment registers
reg:  general registers
disp: a constant value denoting the offset
```

Figure 5.8 Branches with non-determinable targets.

the call to function *INITTASK*, the memory at offset [*ds*:*ofs_04C8*] contained value 0. This value was set to 004B_{hex} after the function call. Since the value cannot be determined until the function is called, the target of the indirect jump cannot be statically analyzed. Assuming a user edited the code segment, which caused the change of offset 004B_{hex}, then the target may land at a wrong location when the jump instruction is executed.

All the non-determinable branch targets can be summarized in a few patterns showed in Figure 5.8. As soon as these instructions are detected, the translator may need to rewrite them into the VMM calls, as described in Section 5.2.2.

5.2.2 Adjusting Non-Determinable Branch Targets

In order to resolve a non-determinable branch target at run-time, the translator replaces the branch with a VMM call and some parameters. Figure 5.9 illustrates an example for such

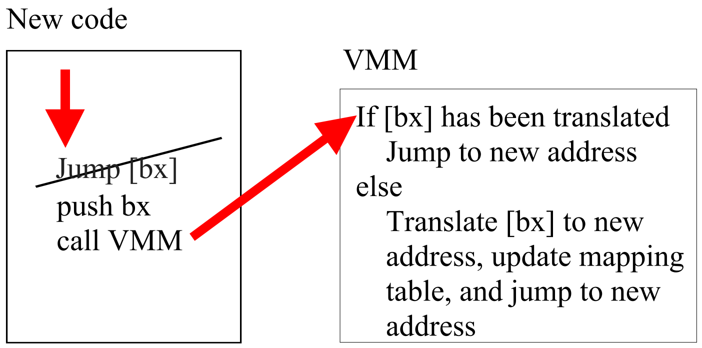

Figure 5.9 Replacing a non-determinable branch instruction with a VMM call.

replacement. If the value of register BX in the instruction "*call [BX]*" cannot be determined statically, this instruction will be replaced with a VMM call with two parameters: register DS and register BX. Upon receiving a call from the program, the VMM obtains the target from [DS:BX]. The value of this target belongs to old code. The translator then checks whether the corresponding new location is the same as the old one, and looks up the new location if they are different. Finally, the translator switches execution to the new target.

Ideally, the translator may pass all necessary information to the VMM by using the space of the original branch instruction. For instance, if the size of the original call instruction is 6 bytes, the translator will replace these 6 bytes with some $PUSH$ instructions followed by the VMM call. The $PUSH$ instructions may be needed to pass a parameter to the VMM. Sometimes there may not be enough space to accommodate these instructions. For example, the instruction "*jmp [bx]*" takes only 2 bytes, which is not enough for calling VMM with parameters. To solve this problem, the translator also maintains an internal fixup table for each code segment.

As shown in Figure 5.8, branch instructions with non-determinable targets include the following attributes: a segment register, a general register, and a displacement. The entry in the internal fixup table must also cover these attributes. Each entry in the internal fixup table consists of 6 bytes: 2 bytes for the location of the requesting instruction, 1 byte for the instruction type, segment register and 1 byte for the general register, and 2 bytes for the displacement. Since the x86 has only 6 segment registers, it suffices to use 1 byte to cover the instruction type and segment register. The most significant bit of the byte denotes the type of instruction. The value 0 stands for *CALL*, and the value 1 stands for *JMP*. The translator uses 1 byte to store 24 possible general x86 register usages: 8 byte addressable registers, 8 word addressable registers, and 8 double word addressable registers. Note that it is likely for 16-bit code to use 32-bit registers. As will be discussed later in this section, the occurrence of a branch with a non-determinable target is rare in most programs, this extra memory needed by the translator should not be a concern.

The translator must also maintain an *Address Mapping Table, AMT* for each translated code segment. This table keeps track of address changes during editing. The format of the AMT is illustrated in Figure 5.10. When a program requests a VMM call, the translator uses this table to find the new offset for the target. The AMT does not need to maintain an entry for each instruction. The number of entries in the AMT depends on how frequently the insertion or deletion occurs. The internal data structure maintained by the translator must be modified as well. Figure 5.11 is derived from Figure 4.1 in Section 4.1 with editing information augmented. The *NewSize* attribute may be used by the translator when it is writing the translated code back to the new executable file.

5.2.3 Analysis of Non-Determinable Branch Targets

The occurrence of non-determinable branch targets is rare. Figure 5.12 and 5.13 show the distribution of the call and jump instructions, respectively, from several benchmark programs.

Offset at Old Code	Offset at New Code
0028h	0038h
...	...
1F27h	202Bh

Figure 5.10 The address mapping table.

```
typedef struct SegNodeTypeTag {
    /* Info for supporting run-time target resolution and editing */
    AMT_TYPE  amt;             /* Address Mapping Table           */
    FIX_TYPE  fixup_table;     /* For undeterminable branches     */
    WORD      NewSize          /* New segment size after editing  */

    /* NFY_LOADSEG data structure */
    DWORD     Size;
    WORD      Selector;
    WORD      SegNum;
    WORD      Type;
    WORD      Instance;
    LPCSTR    ModuleName;

    /* New code segment information */
    BYTE      Translated;      /* Translation bit of the segment  */
    BYTE      Present;         /* Presence bit of the segment     */
    WORD      NewSelector;     /* Pointer to the new code seg     */

    /* Info for control flow analysis */
    OPER_TYPE *head_op, *tail_op;
    DWORD     index;
    MEM_TYPE  code_map[FLOW_MEM_CHUNK_MAP_SIZE];
    MEM_TYPE  fallthru_stack[MAX_STACK_SIZE+1];
    DWORD     top_fallthru_stack;

    struct    SegNodeTypeTag   *nextSeg;
} SegNodeType;
```

Figure 5.11 Internal segment data structure augmented with editing information.

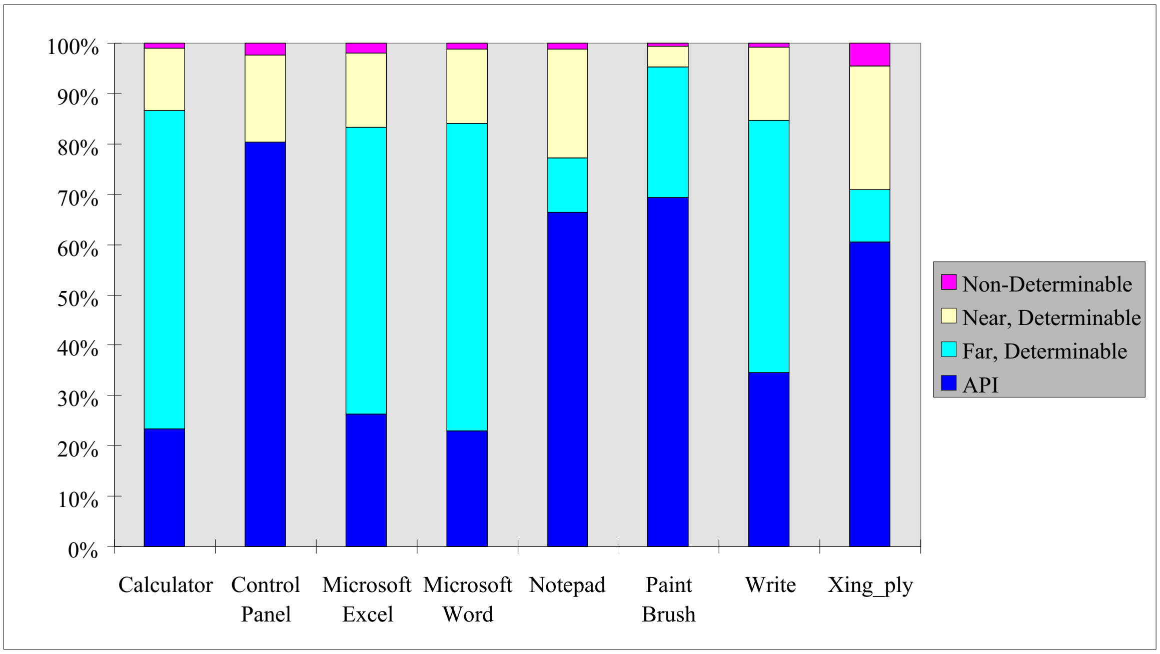

Figure 5.12 Distribution of *CALL* instructions.

There are four types of *CALL* instructions illustrated in Figure 5.12. The first type is an API call. Since the translator currently does not intend to translate Windows operating system code, the target address for the API call does not change during editing. The second type of *CALL* is an inter-segment branch described in Section 5.1.2. The third type of *CALL* is intra-segment branch described in Section 5.1.1. All the above three types of *CALL* instructions do not require the translator's help during run-time. The address adjustment was already performed during translation. Only the fourth type of *CALL* instruction needs the translator's help at run-time. Fortunately, this kind of instruction only accounts for a very small portion of all *CALL* instructions, as shown in Figure 5.12. Most of these instructions are in the startup code, meaning that they are executed only once.

Figure 5.13 shows the distribution of *JMP* instructions from several benchmark programs. Note that Figure 5.13 does not count the number of conditional jumps. The target of an x86

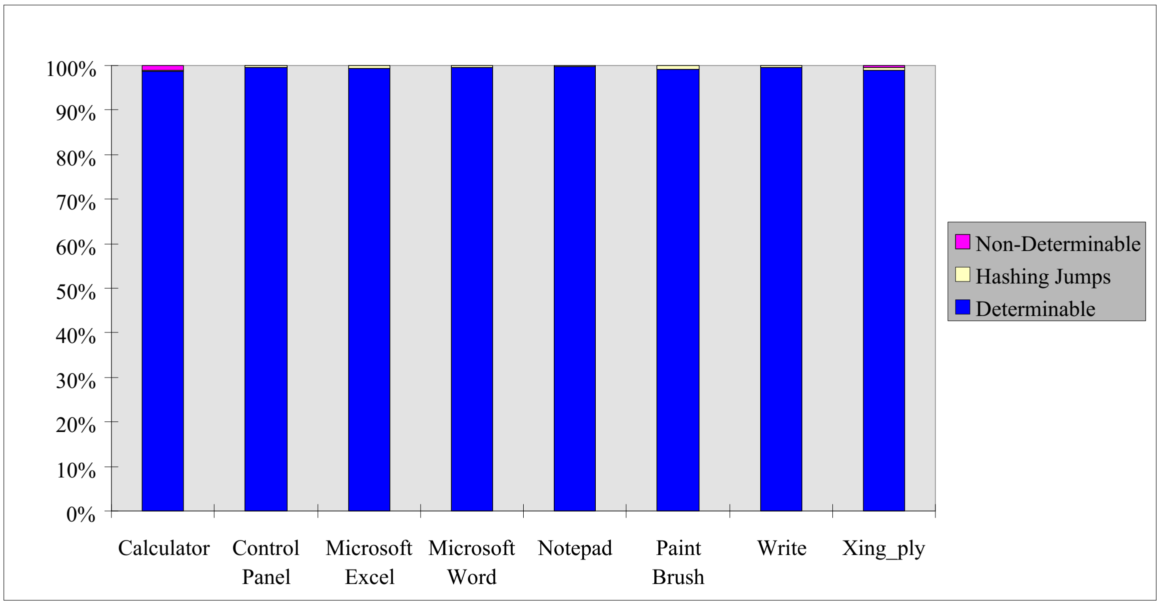

Figure 5.13 Distribution of *JMP* instructions.

conditional branch can always be determined. From Figure 5.13, more than 99 percents of the unconditional jump instructions are determinable. These instructions also contain hashing jumps with regular patterns.

In order to analyze dynamic instruction execution, a simulator was implemented to count the instructions during program execution. Table 5.1 shows the dynamic instruction count for several applications. Since the focus is on the applications, each API call was counted as one instruction. Table 5.1 indicates that non-determinable branch instructions are rare, compared to overall instructions. We also noticed that most of the non-determinable branch instructions occur in program startup, which means that these instructions will be executed only once when the program is executed.

Table 5.2 shows the percentage of non-determinable branch instructions that were executed for several special operations. Two types of applications were chosen: computation-intensive and text processing. For each benchmark program in Table 5.2, the experiment compared the

	Total Dynamic Instruction Count	*Non-Determinable Branch Instruction Count*	*Percentage of Non-Determinable Branches*
Calculator v3.10	126,947	26	0.020%
Control Panel v3.10	61,340	19	0.031%
Microsoft Excel v5.0	1,043,594	891	0.085%
Microsoft Word v6.0	534,675	38	0.007%
Notepad v3.10	61,493	1	0.001%
Paint Brush v3.10	500,602	176	0.035%
Write v3.10	369,466	8	0.002%
Xing_ply v1.0	99,689	20	0.020%

Table 5.1 The percentage of non-determinable branch targets based on dynamic instruction count.

instruction count of the program's startup and the instruction count of repetitive operations. For example, in *Calculator* the repetitive operations are sine, cosine, and other mathematic calculations. For computation-intensive programs, the experiment indicates that the percentage of non-determinable branch instructions decrease when the number of computation operations increases, meaning that a major portion of the non-determinable branches are executed during program startup, as we mentioned earlier. Similarly, for text processing applications, the experiment found that there are no non-determinable branch instructions executed for handling text processing, such as character input and pattern search.

5.3 Incremental Translation

Since some targets of branches may not be known during initial translation, it could be that the target falls into an undecoded area. In this case, the translator may need to dynamically decode additional instructions. Figure 5.14 illustrates an example from Microsoft Calculator. The first block and the third block were decoded during initial translation. The grey arrow in Figure 5.14 denotes the control flow which was traced during the initial translation. Since the translator failed to find any control flow to the second block during the initial translation, this

Computation-intensive programs:

	Total Instruction Count	Non-Determinable Instruction Count	Percentage of Non-Determinable Instructions
XING PLY: Startup	6,314	7	0.11%
Repeated operations (repeated play)	192,413	15	0.01%
Calculator: Startup	15,484	8	0.05%
Repeated operations (sin, cos, ln, log, 1/x)	76,600	12	0.02%

Text processing programs:

	Total Instruction Count	Non-Determinable Instruction Count	Percentage of Non-Determinable Instructions
Notepad: Startup	7,201	1	0.11%
Repeated operations (char input, search)	66,763	0	0%
Write: Startup	55,608	7	0.01%
Repeated operations (text input, format,…)	476,324	0	0%

Table 5.2 The percentage of non-determinable branch targets based on dynamic instruction count of special operations.

block was marked as unknown area. The translator also noticed that somewhere in the third block there exists a branch with a non-determinable target. It replaced the branch instruction with a VMM call.

When the partially translated code is executed, the program will first display a window. If the user hits any key in the window, the code in the third block of Figure 5.14 will be executed. Immediately before the *JMP* instruction is executed, the translator obtains the value of register BX, 0588_{hex}, and the value of register DS, $3F9F_{\text{hex}}$. It then queries the content in location [$3F9F_{\text{hex}}$:0588_{hex}] and finds that the value in location [$3F9F_{\text{hex}}$:0588_{hex}] is $0C88_{\text{hex}}$. This value is actually the offset of the second block in Figure 5.14. If the new code has been changed during initial translation, this value may or may not be the same as its new counterpart. If it is not, the translator will look up the address mapping table to find the new offset to the translated code.

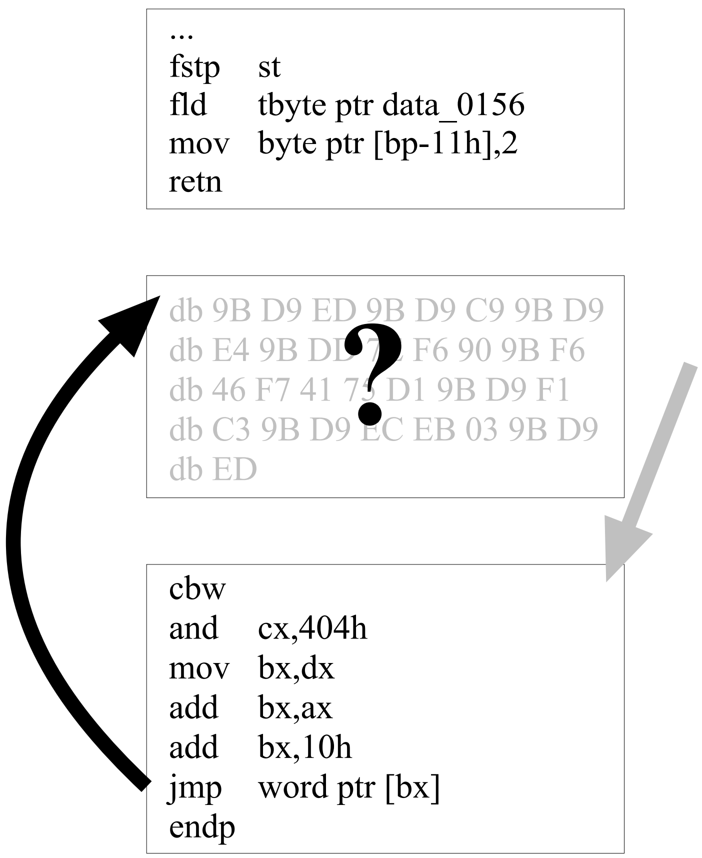

Figure 5.14 A code example of incremental translation (*CALC.EXE*).

Offset	Instruction	Executed?	Target already translated?	Code Detected (%) 73.25% (Base)
041A	jmp dword ptr ss:data_0018e	No	--	--
09A4	jmp dword ptr ds:data_0014e	Yes	Yes	--
1651	jmp word ptr [di]	No	--	--
1D82	jmp word ptr [bx]	Yes	No	73.54%
1DE5	jmp word ptr [bx]	Yes	No	75.49%
28E8	jmp word ptr [bp+LOCAL_3]	No	--	--

Table 5.3 The percentage of detected code for *CALC.EXE*.

Table 5.3 shows the percentage of incrementally detected code for a segment in program *CALC.EXE* during translation. From Figure 5.13, there are six unknown targets for the *JMP* instruction. The first two columns in Table 5.3 show the offset and instruction, respectively. The third column indicates whether the instruction is executed during run-time. Finally, the fourth column shows whether the area pointed by the target of branch has been decoded or not. If the area to which the new target points has not been previously translated, the translator dynamically translates it and calculates the percentage of decoded binaries. The percentage is calculated by dividing the size of code segment by the size of all decoded instructions. Note that it is unlikely that the percentage can reach 100. This is because a code segment may contain data, patched bytes, or any other garbage bytes. This table is for us to understand that more code can be detected through the support of a run-time mechanism. When the program is executed, 2 of them fall into undecoded areas, as indicated in the table. The remaining *JMP* instructions either do not execute or fall into decoded areas during execution.

Chapter 6

Self-Modifying Code

Self-modifying code refers to a piece of code that can be modified by the program itself. The translator needs to detect self-modifying code in order to determine whether or not it should be invalidated. Binary code translation becomes very difficult in the presence of self-modifying code since code translated at a certain time may be different at a later time. If the translator is not aware of such change, the program will most likely crash. Even worse, it is in general not possible to debug a translated self-modifying program when a crash occurs.

An operating system may prevent a user program from generating self-modifying code. In general, Windows 3.1 and later versions of Windows operating systems only allow application programs to be run under the x86 protected mode. This means a code segment may not be written to, and a data segment may not be executed. In this case, self-modifying code seems less likely to happen. On the other hand, Windows allows application programs to alias a code segment into a data segment. Application programs may write to a data segment first, create a code selector, make it aliased to the data segment, and execute the code in the data segment. This is how Windows application programs may contain self-modifying code.

Previous work used hardware to detect self-modifying code [12] [26]. In this thesis, however, a software approach is proposed. Since the proposed approach relies heavily on x86 protected

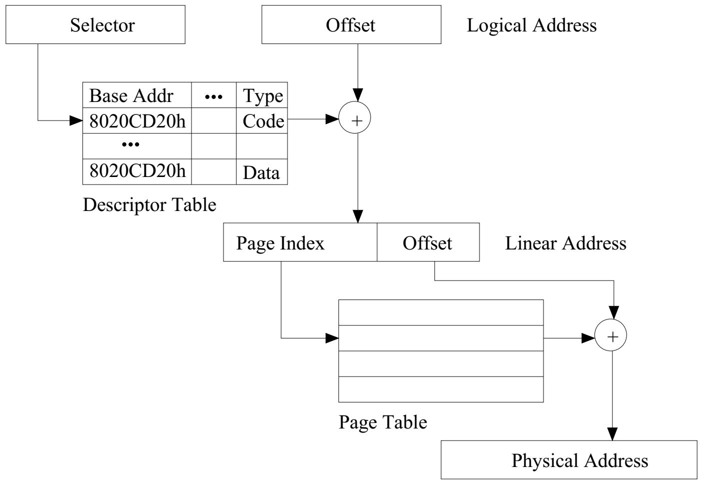

Figure 6.1 The x86 memory address translation process.

mode architecture, Section 6.1 will briefly review the x86 protected mode. Section 6.2 will describe self-modifying code detection strategies and give a concrete example to show how the translator detects the self-modifying code. Finally, Section 6.3 will discuss what the translator should do in the presence of self-modifying code.

6.1 Overview of X86 Protected Mode Architecture

Self-modifying code is processor-dependent. Creating self-modifying code, as well as detecting self-modifying code in x86 code cannot avoid dealing with its processor memory address architecture. Therefore, this section will briefly review the x86 memory address architecture.

The memory address translation process in the x86 is two-folded. Figure 6.1 illustrates these two address translation steps. The address that is visible to the programmer is called the *Logical Address*. The *Physical Address* refers to an address in the *Random Access Memory* where the instructions or data are stored. If the processor is operating in *Protected Mode*, there is one additional layer in the translation from a logical address to a physical address. The segment value in this mode is used by the processor as an index to a descriptor table. This value is called a *selector* in the protected mode. Each entry in the descriptor table contains some attributes such as selector limit, whether it is code or data, etc. It is this layer that prevents code from being written and data from being executed, thus the name "protection". It is also this layer in which an application program can create self-modifying code. The address obtained by looking up the descriptor table is known as a *Linear Address*.

The linear address output from the descriptor table still is not the address where instructions or data reside. By enabling the paging mechanism, each linear address can be mapped to a physical address. The size of each page is 4K bytes. Note that addresses that are continuous in linear address space may or may not be continuous in physical address space. In general, an application does not, and should not, know where the final data is in physical memory.

There are two types of descriptor tables in the x86 processor. There is one *Global Descriptor Table (GDT)* for all tasks, and a *Local Descriptor Table (LDT)* for each task being run. However, Windows only uses one LDT for all user programs. There are also several operating modes for Windows. The various modes of Windows reflect the various modes of the Intel CPUs [20]. This thesis focuses on the Windows enhanced mode which utilizes the x86 protected mode architecture. For a more detailed description of the x86 architecture, the reader may refer to [12].

6.2 Detection Strategies

The example in Figure 6.2 illustrates a sample C code segment that produces self-modifying code by calling a few system functions. It first allocates heap memory from the operating system by calling $GlobalAlloc()$. It then acquires a pointer by calling $GlobalLock()$, so that this memory

```
// .....
// Create a memory heap
hMem = GlobalAlloc( GPTR, MemSize );
// Get a pointer to this memory
lpMem = GlobalLock( hMem );
// Write some data into the memory heap
MemoryWrite( SELECTOROF(lpMem), OFFSETOF(lpMem),
             DataBuffer, sizeof(DataBuffer) );
// Create an aliased code selector to execute the data
lpSel = AllocDSToCSAlias( SELECTOROF(lpMem) );
// Switch control to lpSel and start execution
// .....
```

Figure 6.2 A piece of code that generates self-modifying code.

can be accessed. After this, the code writes the hard-coded instructions which were previously stored in a data buffer into this memory block by calling $MemoryWrite()$. At the last stage, it calls the function $AllocDSToCSAlias()$ to request a code selector from the operating system. This selector shares the same base address as the heap memory selector. Finally, the subsequent instructions can now start executing the aliased data code. This section uses the $HELLOWIN$ program as the example. After some modification, this program contains the code illustrated in Figure 6.2.

The detection strategy used in the translator consists of comparing the information of the LDT entries that are obtained before and after the user program is executed. If there are more entries found after the user program is executed, and the additional entries map to the same linear address as that of original entry but with a different type (code or data), then there may exist self-modifying code. Note that an application program may create segment aliases but never make self-modifying code. For example, in the code example, after the function $AllocDSToCSAlias()$ is called, the program may not necessarily switch the execution control to $DataBuffer$. Thus, this approach provides a more strict examination in that it may expect a greater potential presence of self-modifying code than actually present in real situations. This thesis's policy, however, is to minimize the risk of translating the self-modifying code.

Both Windows and the x86 processor maintain LDT information. Each LDT entry in the x86, however, takes only 8 bytes. It cannot provide as much information needed for Windows to manage the overall system. As a result, Windows also maintains auxiliary data called *global arena* [22] [25]. These global arena structures are chained together in a linked list. For each memory object, the global arena also indicates its owner. This additional information helps the translator filter out the unrelated descriptors when it traverses the global arena.

Figure 6.3 shows the results obtained when the translator traverses the Windows LDT, after the sample program HELLOWIN is executed. For each memory object, the data structure includes its linear address, size, handle (or selector) in the LDT, and its owner handle. The type field in Figure 6.3 indicates the type of the memory object. For example, value 2 means data, value 3 means code, etc. A detailed description for all values can be found in the Windows Software Development Kit (SDK) [19]. The data field contents in Figure 6.3 depend on the value in the type field. If the type of a memory object is neither code nor a Windows resource, then the value in its data field is 0. As with the type field, detailed information can be found in the SDK.

Windows provides a set of system functions for debugger writers. Among other things, this set provides a couple of heap walk functions. The translator provides an empty data structure and calls these functions. Upon returning from one of these functions, this data structure will be filled, with the descriptor and other information in it. As indicated in the figure, in addition to HELLOWIN, the linked list contains Windows system memory such as $KRNL386$, GDI,

```
LinAddr     Size   Handle   Owner   Type  Data  FileName
============================================================================
    FC00h  B400h     117h    10Fh     3h    1h   C:\WINDOWS\SYSTEM\KRNL386.EXE
   9D460h   B40h     10Eh    10Fh     6h    0h   C:\WINDOWS\SYSTEM\KRNL386.EXE
80805C00h    C0h     146h    10Fh     0h    0h   C:\WINDOWS\SYSTEM\KRNL386.EXE
802F0140h   220h     D66h    D66h     4h    0h   C:\BC5\BIN\BCW.EXE
806FB9E0h   340h    2686h    D66h     0h    0h   C:\BC5\BIN\BCW.EXE
   1C200h    20h    1F6Eh    237h     5h    6h   C:\WINDOWS\SYSTEM\GDI.EXE
803CF040h  EDC0h    36E6h    237h     0h    0h   C:\WINDOWS\SYSTEM\GDI.EXE
801789A0h    A0h    2C77h   17CFh     0h    0h   C:\WINDOWS\SYSTEM\USER.EXE
801C2000h  1300h    17C7h   17CFh     3h    1h   C:\WINDOWS\SYSTEM\USER.EXE
......
   2A580h   120h    3B4Fh   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
8020CD20h 10000h    1256h   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
8021CD20h 10000h    124Eh   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
80804900h    E0h    3B46h   3B47h     6h    0h   C:\WORK\HELLOWIN.EXE
808059E0h   220h    1126h   1126h     4h    0h   C:\WORK\HELLOWIN.EXE
808064C0h  2640h    315Eh   3B47h     1h    2h   C:\WORK\HELLOWIN.EXE
80808B00h    20h    1236h   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
80808B20h   140h    122Eh   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
80808C60h   620h    1226h   1126h     0h    0h   C:\WORK\HELLOWIN.EXE
81540BC0h   8C0h    11CEh   3B47h     3h    1h   C:\WORK\HELLOWIN.EXE
```

Figure 6.3 Information in selector table.

and *USER*, as well as other applications, such as *BCW* (the Borland C compiler). Though the file name is not included in the list, the translator can derive it from the handle in the node. The *handle* of a memory object serves as an identifier that makes the object unique in Windows.

For each running program, Windows maintains per-task data called the *Task Database*. A task is a thread of execution through code segments loaded by Windows [22]. The task database owns a memory object whose type field has the value 4_{hex}, as indicated in Figure 6.3. From Figure 6.3, the HELLOWIN object with type value 4_{hex} has a handle value 1126_{hex}. This handle value is exactly the same as the owner of some other memory objects that belong to HELLOWIN.

Though using global arena traversing functions in *TOOLHELP* provides some information needed for the translator, it does not provide all the necessary information. To give an example, let us consider the memory objects that belong to HELLOWIN. From Figure 6.3, all the linear addresses are unique. While the translator expects to find two entries with identical linear addresses but with different handle values for HELLOWIN, as it does own aliased segments, only one of the entries is found. Therefore, the translator needs to query the x86 LDT for some additional information.

Querying the x86 LDT requires the translator to run the code in the higher x86 privileged level. In order to achieve this, the *DOS Protected Mode Interface (DPMI)* function described in [24] was used. Basically, the idea behind this approach is for an application program to request a DPMI service which runs at the most privileged level. As such, it can acquire the x86 LDT and return the information to the application program. As it turned out, the translator was able to find out aliased segments at this time.

The translator queried the x86 LDT twice, one before HELLOWIN was executed and the other after execution, dumping the x86 LDT contents for each query. Figure 6.4 shows the result of several commands used to compare these two content files. The file *LDT_DUMP.OLD* in Figure 6.4 illustrates the x86 LDT contents obtained before HELLOWIN was executed. The file *LDT_DUMP.NEW* illustrates the x86 LDT contents obtained after HELLOWIN was executed. The only difference is that there was one more entry in x86 LDT after HELLOWIN was executed. The linear address for this additional descriptor is $8020CD20_{\mathrm{hex}}$, and the type value is FB_{hex}. This linear address is exactly identical to that of one of the HELLOWIN entries that appeared in Figure 6.3.

An explanation needs to be made for the one-byte *Type* attribute in the x86 LDT contents files, as it is different from the aforementioned one defined by Windows. For each 8-byte x86 LDT entry, the least 4 significant bits in this byte indicate the *Application Segment Types*. Table 6.1 shows the 16 types based on these 4 bits [12]. From Figure 6.4, there is only descriptor whose linear address has value $8020CD20_{hex}$ in LDT_DUMP.OLD with type value $F3_{hex}$. The least 4 significant bits in the type value are 0011_{bin} (Number 3), meaning that the descriptor type is data which can be read/written, and has been accessed. Now let us examine the file LDT_DUMP.NEW. In addition to the entry in LDT_DUMP.OLD, there is another descriptor which has the same linear address and has type value FB_{hex}. The least 4 bits are 1011_{bin} (Number 11), meaning that the descriptor type is code which can be executed/read, and has been accessed. This means that there are two entries that map to the same linear address, with one writable and the other executable. Therefore, the self-modifying code may exist by writing to this linear address first and executing the written data. The translator has now successfully detected the potential self-modifying code in HELLOWIN.

6.3 Handling Self-Modifying Code

Since there may be multiple segments in an application program, not all containing self-modifying code, the translator may still translate those segments which are not self-modifying. If an instruction in a translated segment calls a function in the self-modifying segment, it will use the old target. Since the translator did not translate the self-modifying segment, this target did not change either. If, on the other hand, the instruction in the self-modifying segment

```
C:\work\fc ldt_dump.old ldt_dump.new
Comparing files LDT_DUMP.OLD and LDT_DUMP.NEW
****** LDT_DUMP.OLD
LinAddr = 8038A300h, Type = F3h
LinAddr = 807B64C0h, Type = F3h
****** LDT_DUMP.NEW
LinAddr = 8038A300h, Type = F3h
LinAddr = 8020CD20h, Type = FBh
LinAddr = 807B64C0h, Type = F3h
******

****** LDT_DUMP.OLD
LinAddr = 825A9000h, Type = F3h
LDT entry count = 1576
****** LDT_DUMP.NEW
LinAddr = 825A9000h, Type = F3h
LDT entry count = 1577
******

C:\work\grep 8020CD20 LDT_DUMP.OLD
File LDT_DUMP.OLD:
LinAddr = 8020CD20h, Type = F3h

C:\work\grep 8020CD20 LDT_DUMP.NEW
File LDT_DUMP.NEW:
LinAddr = 8020CD20h, Type = F3h
LinAddr = 8020CD20h, Type = FBh
```

Figure 6.4 Comparison of LDT entries: one without aliased segments in application program and the other with aliased segments.

Number	E	W	A	Descriptor Type	Description
0	0	0	0	Data	Read-Only
1	0	0	1	Data	Read-Only, accessed
2	0	1	0	Data	Read/Write
3	0	1	1	Data	Read/Write, accessed
4	1	0	1	Data	Read-Only, expand-down
5	1	0	1	Data	Read-Only, expand-down, accessed
6	1	1	0	Data	Read/Write, expand-down
7	1	1	1	Data	Read/Write, expand-down, accessed
Number	**C**	**R**	**A**	**Descriptor Type**	**Description**
8	0	0	0	Code	Execute-Only
9	0	0	1	Code	Execute-Only, accessed
10	0	1	0	Code	Execute/Read
11	0	1	1	Code	Execute/Read, accessed
12	1	0	1	Code	Executed-Only, conforming
13	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	Code	Execute/Read-Only, conforming, accessed

Table 6.1 Application segment types.

calls the function in the translated segment, then there may be trouble if special care is not taken. This is due to the fact that the target of a call or jump may land anywhere in the translated segment, and the target offset in the translated segment may have been changed. The translator may maintain an address mapping table that keeps track of pairs of old addresses and new addresses for the translated segment. The problem, however, is that it is too late to use this information unless the run-time environment can suppress the current instruction execution in the self-modifying segment. One way to solve this problem is to single-step the instruction execution in the self-modifying segment. Though this sounds costly, it is rare for an application program to contain self-modifying code. It is even rarer that a self-modifying program contains a large volume of multiple segments.

At the final stage, the translator may need to write the self-modifying code detection results back to the executable file. A *Self-Modifying Code Detection Table* is added to the executable file. For each segment, the table maintains a flag to indicate whether the segment has potential self-modifying code. When the segment is loaded into memory, the translator checks this flag in order to determine whether it should translate the segment or not.

The translator must be able to locate the Self-Modifying Code Detection Table. Two of the reserved bytes described in Section 3.2.1 are used for the translator to access this table. A detailed design for the new executable file will be discussed in Chapter 7.

Chapter 7

Post Translation

Binary translation may not be completed without saving the translated code for reuse. Since run-time translation can be time consuming, the translator should avoid retranslating the same code. A full translation cycle can be achieved after the translated code is written back to the secondary storage. To reuse the translated code, however, future invocations of the translator must also be able to detect the fact that the file has been translated. Therefore, both the executable file and translator need to be modified. This chapter describes the details of writing the translated code into disk. Since some of the NE file format information is used in this chapter, the reader may wish to review the NE file format covered in Section 3.2.

In the first step of writing to disk, the translator reads in the original executable file. Both the header and body of the original executable file are needed in order to create the new executable file. The translator needs header information to determine the new offset of some data such as the segment table. Modifying the file header will be discussed in Section 7.1. Modifying the segment body will be discussed in Section 7.2.

Some of segments in the original executable file may be needed when the translator generates the new executable file. If the segment has never been loaded for execution, then there is no chance that the translator has translated it. To handle this, the translator just copies the whole segment into the new executable file. If the segment has been changed, then the translator may need to modify all the relocation data following the segment to reflect the code size change. Section 7.3 addresses this problem. Finally, Section 7.4 will discuss the overall design for the new executable file.

7.1 Modification of Executable Header

Each NE header contains the initial program counter in a double word, that is, CS:IP in Figure 3.2. The initial program counter is actually the starting address of the program. If the initial program counter was changed during editing, the translator needs to replace it with the new value. Note that though the segment that contains the initial program counter must be loaded into memory, modification of the initial program counter may not be necessary. This is because editing the code segment may not change all the addresses of instructions. For instance, if a few bytes are inserted at an offset that follows the initial program counter, the offset of initial program counter is not changed, even though the size of segment was expanded.

Secondly, the translator may need to modify entries in the segment table. For some entries, offsets may need to be updated even though their corresponding segments do not change. This is because the size change of a segment may affect all of the segments' offsets whose values are larger than current one. In addition, for those segments whose sizes have been changed, the translator needs to modify the segment length as well as allocation size.

The translator must next modify the entry table. Each entry in the entry table keeps track of the address of an exported function. Put another way, it provides an address *thunk* for other functions to call. If the segment size was changed, and such change results in a different offset for the exported function, then its entry addresses also need to be adjusted accordingly.

One side effect of changing the code size is the requirement for the translator to update the resource table. As described in Section 3.2.2, resources can be an icon, cursor, mouse, or others. Basically, the translator should never change a resource in the executable file, since a resource is not code. The problem is that the location for each resource is relative to the beginning of the executable file, which may be affected due to the change of the segment size. As such, the translator needs to check the offset for each resource, and adjust them whenever necessary.

7.2 Modification of Executable Body

If a segment was changed during editing, it will be written back to disk from main memory. This in-memory image will replace the original segment when the translator writes to the new executable file. If the segment was not changed, then the translator will just copy the whole segment from the original executable file to the new executable file. In addition, the translator may also need to patch a few bytes from the original file to the new one. The number of bytes being patched is calculated by subtracting the offset of the last byte from the offset of the next segment. Figure 7.1 (a) illustrates the calculation of the needed patching size.

Care must be taken for the permutation of segments in the executable file. From [18] it seems that all per-segment data are attached to the end of the executable header in a descending order; that is, the body of segment 2 should follow that of segment 1. However, this is not true in general. This sequence holds only in the segment table. In the segment table, assume there are n entries, then the logical segment ID of the first entry must be 1. However, the offset of segment 2 could be less than that of segment 1. This may be the result of *gangload*, which will be defined in Section 8.3. When writing to disk, segment order must be chosen according its offset value, rather than its ID, so that the translator can keep advancing the file pointer.

When the linker encounters a target that it cannot resolve at link-time, it generates a fixup record in the executable file for the loader. Given an instruction that contains a far target, the linker will fill the segment field with value $\mathrm{FFFF}_{\mathrm{hex}}$. Also, for API call instructions, the linker will generate a value of 0000_{hex} for the offset field of the target. Although these two values will be replaced by the loader at load time, when the translator generates the new executable file, it still needs to mask the segment value with $\mathrm{FFFF}_{\mathrm{hex}}$ for all instructions that need fix-ups, and mask the *segment*:$offset$ values with a $\mathrm{FFFF}_{\mathrm{hex}}$:$0000_{\mathrm{hex}}$ pair for all API calls. Thus, the binary translator needs to update the information generated by the linker.

Incidentally, the translator implemented in this thesis differs from Digital's FX!32 [8] in that FX!32 is capable of modifying the Windows NT loader. As such, determining whether the portion of binaries have been translated or not is the loader's task. Since our translator is not able to modify the loader, the modification information must be passed through through the loader, letting the translator deal with all remaining tasks.

After the code segment is written to disk, the translator writes relocation data if the current segment has any. Relocation data tells the Windows loader that at some offset in the segment there is an address that cannot be determined until run-time. The translator does not care about this address as far as generating an executable file is concerned. However, the offset that indicates the need of fixup is crucial if the segment size was changed during editing. The translator only needs to modify the original offset if it was already different from the new offset.

7.3 Dealing with Size Expansion

There are two situations that occur after rewriting a code segment. Figure 7.1 illustrates these two situations. The first situation happens when the expanded segment does not overlap

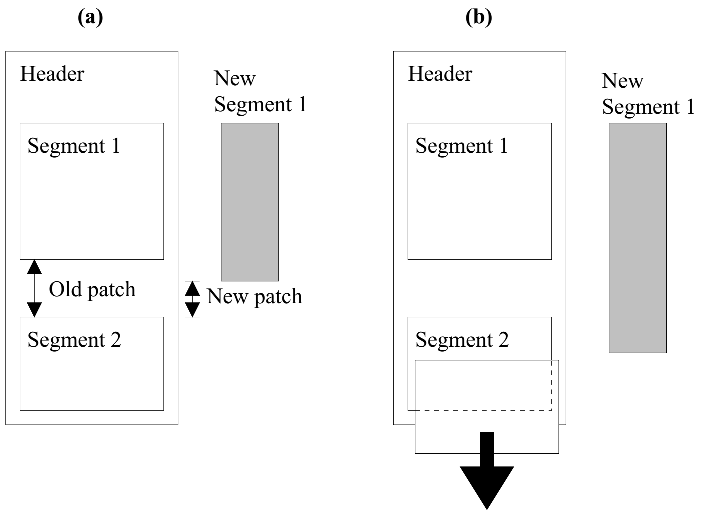

Figure 7.1 Two situations resulted after segment size expansion.

with the offset of the segment immediately following it. In this case, the header does not need to be modified. The translator only needs to decrement the size of the patching code by the the size of the net expansion. In Figure 7.1 (a), assuming the *Oldpatch* area occupies 20 bytes and Segment 1 only increases by 10 bytes, then the offset of Segment 2 will remain the same value. After writing the new Segment 1 to disk, instead of writing additional 20 bytes, the translator only needs to patch another 10 bytes, since the file pointer has already been advanced by 10 bytes.

If segment expansion results in overlap of segments, as indicated in Figure 7.1 (b), then the translator needs to modify the offsets of all following segments whose offset values are larger than that of the current segment. The offset of a segment in the executable file does not consist of a double-word or DWORD. Rather, it is a combination of the logical sector offset and the file alignment size, as described in Section 3.2.1. To determine the offset of a segment, the translator needs to left shift the sector offset value by the size of the alignment count. For each segment, the sector offset is stored in a corresponding entry in the segment table. The alignment shift count is stored at offset 32_{hex} to the beginning of the NE file.

Let us use an example to show how the segment offset is calculated. Assume at offset 32_{hex} to the NE file there is a WORD (2 bytes) whose value is 09_{hex} and a segment whose sector offset is 04_{hex}. The segment offset is then determined to be

$$(DWORD)((DWORD)04_{\text{hex}} << 09_{\text{hex}}) == 0100000000000_{\text{bin}} == 800_{\text{hex}} \tag{7.1}$$

Since the segment offset consists of two components, there are two approaches to modifying it. The translator could change the value of the alignment shift count, or change the sector offset. There are several disadvantages for the first approach. The major disadvantage of this approach is that, in general, it wastes space. Incrementing the alignment shift count by one results in an unnecessary waste of space. Let us take the program $HELLOWIN.EXE$ as an example. $HELLOWIN.EXE$ contains a code segment and a data segment. The size of the code segment is $8B4_{\text{hex}}$. Assume the user inserted 100_{hex} bytes at offset $7D6_{\text{hex}}$ to the segment. By using techniques that were mentioned in previous chapters, the new code can execute correctly. Now the translator is going to save the translated code by writing to a new executable file. The value of the alignment shift count for $HELLOWIN$ is 9, and the sector offsets for the code segment and data segment are 4 and 9 respectively. The original offsets of the code and data segments are then

$$(DWORD)((DWORD)04_{\text{hex}} << 09_{\text{hex}}) == 0100000000000_{\text{bin}} == 800_{\text{hex}} \tag{7.2}$$

and

$$(DWORD)((DWORD)09_{\text{hex}} << 09_{\text{hex}}) == 1001000000000_{\text{bin}} == 1200_{\text{hex}} \tag{7.3}$$

respectively. While examining the original code we found that the code segment size is $8B4_{\text{hex}}$, and there are 23_{hex} fix-up records immediately following this segment. Each fix-up record occupies 8 bytes. There are 2 additional bytes indicating the number of fix-up records. So, after dumping, the position of the file pointer should be at

$$800_{\text{hex}} + 8B4_{\text{hex}} + (23_{\text{hex}} * 8) + 2 == 11CE_{\text{hex}} \tag{7.4}$$

Now since inserting 100_{hex} bytes at offset $7D6_{\text{hex}}$ in the code segment causes the file pointer to become $12CE_{\text{hex}}$ which is larger than the offset of data segment (i.e., 1200_{hex}), the translator needs to adjust the data segment offset to avoid overlapping.

If the translator increments the alignment shift count by 1, then the final data segment offset will become

$$(DWORD)((DWORD)09_{\text{hex}} << 0A_{\text{hex}}) == 10010000000000_{\text{bin}} == 2400_{\text{hex}} \tag{7.5}$$

If the translator increments the sector offset by 1 instead, then the final offset will become 1400_{hex} as calculated below, a saving of 1000_{hex}.

$$(DWORD)((DWORD)0A_{\text{hex}} << 09_{\text{hex}}) == 1010000000000_{\text{bin}} == 1400_{\text{hex}} \tag{7.6}$$

Moreover, the alignment shift count is global to all segments, but not all the offsets of segments need to be changed. For the above example, if the translator changes the alignment shift count, it needs to change the sector offset of the code segment in order to make the final offset unchanged.

We used Borland's Turbo Dump Version 4.2.19.3 to dump the binary images of new generated file called *NEW_HELLOWIN*, and compared these with the original image. Figure 7.2 shows all of the differences encountered.

The alignment shift count is 9, meaning that the minimal unit in this file is 200_{hex} bytes. From Figure 7.2, the new DOS file size and old DOS file size differ in 200_{hex} bytes. Since the code size has increased by 100_{hex} bytes, both the allocation size and file length have also increased by 100_{hex} bytes, as indicated in Figure 7.2. The next change takes place in the data segment. Though the size of the data segment did not change, its sector offset must be updated due to the size expansion of the proceeding code segment. Since the minimal unit is 200_{hex} bytes, the translator only needs to increase the sector offset by one, from 09_{hex} to 0A_{hex}. The remaining changes in Figure 7.2 are relocation fixups in the code segment. For example, "*PTR 07F8h KERNEL.5*" in Figure 7.2 indicates that at offset 07F8h there is a referencing target to function $KERNEL$.5. The location of $KERNEL$.5 is not important. However, since 100_{hex} bytes have been inserted at $07\text{D}6_{\text{hex}}$, the location $07\text{F}8_{\text{hex}}$ where a fixup is needed must be changed to $08\text{F}8_{\text{hex}}$. Note that not all entries need to be updated. For example, the location 061Dh is not required to be updated, as it is less than $07\text{D}6_{\text{hex}}$.

7.4 Designing the New Executable File

The translator may need to store some important information in the new generated executable file. This information must be visible to the translator to accelerate the translation process.

```
Comparing files hellowin.dmp and new_hellowin.dmp

***** hellowin.dmp
DOS File Size                 1600h ( 5632. )
***** new_hellowin.dmp
DIS File Size                 1800h ( 6144. )

***** hellowin.dmp
  Segment Type:  CODE         Alloc Size:  08B4h
  Sector Offset: 0004h        File length: 08B4h
***** new_hellowin.dmp
  Segment Type:  CODE         Alloc Size:  09B4h
  Sector Offset: 0004h        File length: 09B4h

***** hellowin.dmp
  Sector Offset: 0009h        File length: 0224h
***** new_hellowin.dmp
  Sector Offset: 000Ah        File length: 0224h

***** hellowin.dmp
  PTR   07F8h                 KERNEL.5
  PTR   088Dh                 KERNEL.6
  PTR   0837h                 KERNEL.7
  PTR   061Dh                 KERNEL.137
  PTR   08ABh                 KERNEL.10
***** new_hellowin.dmp
  PTR   08F8h                 KERNEL.5
  PTR   098Dh                 KERNEL.6
  PTR   0937h                 KERNEL.7
  PTR   061Dh                 KERNEL.137
  PTR   09ABh                 KERNEL.10

***** hellowin.dmp
  PTR   07ECh                 KERNEL.23
  PTR   0804h                 KERNEL.24
***** new_hellowin.dmp
  PTR   08ECh                 KERNEL.23
  PTR   0904h                 KERNEL.24
```

Figure 7.2 Comparison of original and new HELLOWIN.EXE files.

The translator may use 4 bytes starting at offset 37_{hex} to the NE file to store the sector offset to the Self-Modifying Code Detecting Table indicated in Section 6.3. This sector offset, combined with the file alignment shift count, can be used by the translator to locate the table in the new executable file. In normal situations, using 2 reserved bytes is enough to locate the table. In the worst case, however, the original file may be too large to be indexed by the combination. As such, the translator uses 4 bytes as the sector offset.

As described in Chapter 5, the Address Mapping Table may also need to be saved to the new executable file. Similarly, the translator can use 4 bytes starting at offset 3B_{hex} to the NE file to store the sector offset to this table.

Chapter 8

Improving the Program Load Time

In order to illustrate the usefulness of the binary translation framework, an optimizer that improves the load time of the executable is designed and implemented in the framework.

8.1 Description of Load Process and Costs

The program load time is crucial to its overall performance. When the user launches a program for execution, the operating system must load part or all of the program into main memory from secondary storage. In comparing secondary storage access time with register or main memory access times, accessing data from secondary storage requires significantly greater amounts of time, due to the physical characteristics of secondary storage.

In Windows executable files, each segment has several attributes, as described in Section 3.2.2. One of the attributes, *PRELOAD*, has implied two possible values: PRELOAD and LOADONCALL. When the user invokes an executable file, all PRELOAD segments will be loaded into memory before the program starts execution. A *LOADONCALL* segment will not be loaded into memory until any instruction in this segment is executed. When the loader loads the program into memory, it first sets up the program's *Module Database*, or *Module Table*. The *Module Database* is essentially an in-memory version of the NE file header, as described in

Section 3.2, and contains the information necessary to perform dynamic linking [25]. Among other things, the module database includes a segment table that keeps track of all segments for the executable program in memory. This segment table serves as a placeholder for all the segments belonging to the executable. For example, the final address of a segment cannot be known in a dynamic linking system until it is loaded into memory. As soon as the final address for the segment is available, the loader fills the address field in the segment table for this segment.

After setting up the module database, the loader allocates a selector in the *Local Descriptor Table (LDT)* for each PRELOAD segment, loads these segments into memory, and fills the final address fields in the module database for these segments. For those segments whose attributes are LOADONCALL, the selectors are allocated, but the loader does not load the segments into memory. Rather, it clears the *segment-present bits* for these segments in their corresponding LDT entries. A more detailed description of the LDT can be read in Chapter 6. Later on, if any of these LOADONCALL segments is referenced, the x86 processor will generate a segment-not-present fault. Upon receiving this fault, the operating system loads this segment into memory and resumes execution. For a detailed description of the executable program invocation process, the reader may refer to Chapter 3 in [22].

Loading too many segments that are not referenced during program execution both incurs excessive segment load time and wastes precious memory resources. The latter may, in turn, degrade the overall system performance by evicting other programs from memory. On the other hand, those segments with the LOADONCALL attribute, but needed during program execution, may also increase the total load time. This additional time results from exception handling (segment-not-present fault) overhead incurred when unloaded segments are referenced.

In order to better understand which aspects we can improve during program loading, this chapter will define the term *startup*. Since there are several meanings for the term *startup* in the context of Windows programming, in order to avoid confusion, throughout this chapter the term *startup* refers to the time from when the user invokes a Windows executable program until the program is ready for user interaction. Based on this definition, the program startup time is different from the load time and the operating system turnaround time. Indeed, it includes all of the following five different components:

(1) The time to load all the PRELOAD segments

(2) The time to execute instructions in part or all of the PRELOAD segments, until the program falls into its message loop

(3) The time to load LOADONCALL segments, if these segments are referenced by any previously executed code.

(4) The time to execute instructions in (3).

(5) The operating system overhead, including multi-tasking.

Note that instructions in (2) may belong to only some of the PRELOAD segments since not all preloaded segments will be executed during program startup. On the contrary, LOADONCALL segments are not loaded unless they are executed. This chapter will discuss the feasibility of reducing the time spent in (1) and (3).

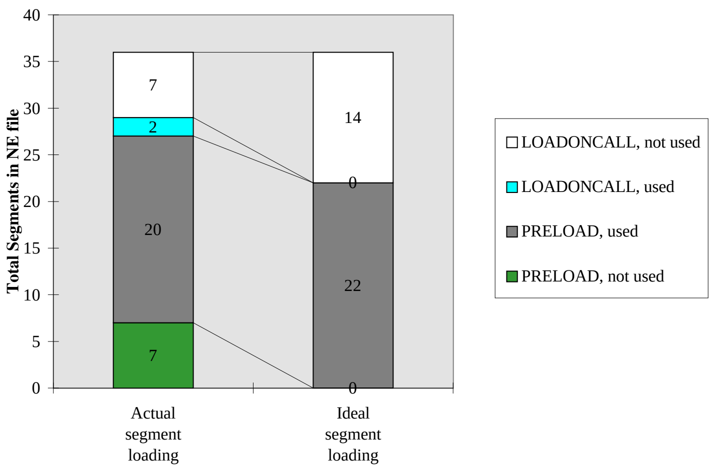

Figure 8.1 Segment loading distribution for MicroEmacs startup.

8.2 Examples of Load Time Inefficiencies

In order to examine segment loading information, a segment loading profiler was implemented to keep track of the segments used and their loading attributes during startup. The segment loading profiler was then used to analyze the segment loading behaviors of both MicroEmacs and Microsoft Excel version 5.0. The analysis results indicate that the time spent in (1) and (3) from Section 8.1 may be reduced for both MicroEmacs and Microsoft Excel.

Figure 8.1 shows the statistical information gathered during MicroEmacs startup. MicroEmacs contains 36 code segments in the NE file, with 27 marked as PRELOAD segments. During program startup 22 segments were used. Among these segments, 20 were PRELOAD segments and 2 were LOADONCALL segments. Only 20 out of 27 PRELOAD code segments were referenced, which means that more than 25% of the PRELOAD segments were not used during startup. Loading segments without using them is far more wasteful than using LOAD-

ONCALL segments. This is because loading a segment usually takes longer than processing a segment-not-present fault. Moreover, as mentioned earlier, loading a segment into memory may evict other useful code.

The rightmost bar in Figure 8.1 illustrates the ideal loading situation. The ideal case occurs when those segments which are needed during startup are marked PRELOAD, and those segments marked LOADONCALL are not used during startup. This ideal loading configuration can be achieved when segment loading profiling information is available and the translator is able to modify the NE file.

Figure 8.2 shows the statistical information gathered during Microsoft Excel startup. Excel contains 248 code segments, including 32 PRELOAD segments. After the user invokes Excel, 31 out of the 32 PRELOAD segments are used during startup. However, Figure 8.2 shows that there are 70 code segments used during startup. This implies that there are 39 LOADONCALL code segments referenced during startup. As a result, the processor needs to process 39 segment-not-present faults while loading these LOADONCALL segments. This overhead can be avoided if these segments are marked as PRELOAD in the original executable file.

It is the responsibility of programmers to tell the linker which segments should be PRELOAD and which should be LOADONCALL [21]. After a binary program is generated, this information becomes invariant throughout the program's lifespan. If the programmer did not correctly set the attributes for the segments, extra time will be incurred whenever the program is loaded.

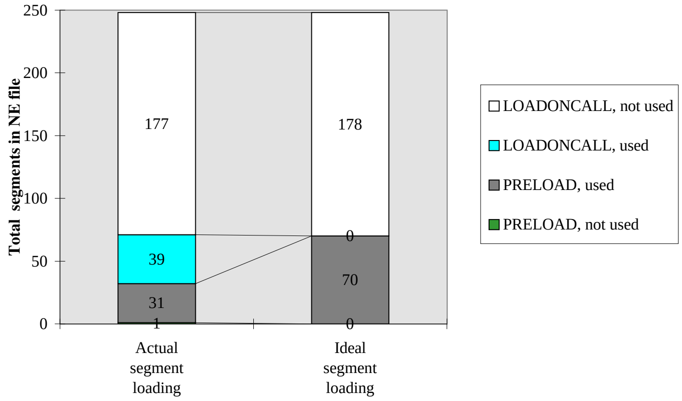

Figure 8.2 Segment loading distribution for Excel startup.

8.3 Improving the Load Time

8.3.1 Profile-Driven Compilation

Profiling the executables during development can help detect the segments used at run-time. Pietrek [23] discusses how to improve program load time by running the compiled executable program and collecting segment loading information. Using this information, the developer can then guide the compiler or linker to generate a new executable program. However, profiling during development may not always reflect all users' application needs. One user may use a program to perform a certain task while another user may use the same program to perform a different task. Different tasks may exercise different portions of the code, and thus, different code segments. Moreover, after a profile guided executable is generated, there is no way for each user to customize their segment loading configuration. The next section discusses how the translator can improve the program load time. In addition, a scheme is proposed that allows users to participate in configuring the segment loading sequence based on their preferences.

8.3.2 Executable Translation

Improving the load time involves two steps. In the first step, the original load information is collected. Then in the second step, the translator modifies the executable file in order to reduce the overhead of unnecessary segment loading and/or exception handling. The net result is that the load time improvement is transparent to the loader.

In order for users, who do not have source code available, to optimize the load time based on their needs, the translator must first collect all segment load information during program startup and throughout the user's program usage. Any commands issued during the program's lifetime may also result in loading LOADONCALL segments. To further reduce the segment-not-present faults, these LOADONCALL segments should also be changed to PRELOAD segments, provided that there is enough memory to hold all PRELOAD segments.

After collecting load information, the next step is to modify the executable file. The NE executable file format was introduced in Section 3.2. This chapter focuses on the segment attribute field. Each entry of the segment table in the NE file contains 4 words, or 8 bytes, representing each segment. The third word defines some flags. Figure 3.3 illustrates these flags and their corresponding meanings [18]. Equipped with this information and the segment load information gathered from the user, the translator is able to modify the PRELOAD flag for the accessed segments by setting the 8th most significant bit (the 4 from 0040h in Figure 3.3 has its 8th bit set). The PRELOAD flag for the PRELOAD segments that were not accessed is reset to 0 to indicate LOADONCALL.

Special care must be taken when the translator modifies the segment attribute. When the loader initializes the module database for an NE file, it will automatically make the following segments PRELOAD, even though the PRELOAD bit is clear in the NE file [22]:

(1) All code segments that are MOVEABLE and NONDISCARDABLE

(2) All code segments that are NONMOVEABLE

The MOVEABLE flag and DISCARD flag are defined in Section 3.2.2. Most LOADONCALL code segments have their MOVEABLE and DISCARD attributes set in the segment table entry in the NE file. However, when the translator detects a LOADONCALL segment whose MOVEABLE bit is off, or whose MOVEABLE bit is on but DISCARD bit is off, it should regard this as a PRELOAD segment. When the translator needs to change a LOADONCALL segment to PRELOAD, it only needs to set the PRELOAD bit in the segment entry of the NE file. If, on the other hand, the translator needs to change a PRELOAD segment to LOADONCALL, it needs to clear the PRELOAD bit, clear the MOVEABLE bit, and set the DISCARD bit.

One important point is that the segment access pattern may vary even after the executable file is generated. For example, most application programs allow the user to configure their application preferences. The segments needed for one configuration may be different from the other. Users may reinvoke the translator to perform load optimizations when the segment access pattern changes.

8.3.3 Gangload

Starting with Windows 3.0, Microsoft introduced a technique called *gangload*, or *fastload*, to accelerate program loading [22]. Suppose there are 10 code segments in the compiled code, with segments 1, 3, 5, 7, and 9 marked PRELOAD and segments 2, 4, 6, 8, 10 marked LOADONCALL. If the loader is going to load all the PRELOAD segments, it will take much time seeking and reading each individual segment. When programmers invoke the resource compiler to generate a final executable file, they may tell the resource compiler to create a gangload area. The task of the resource compiler is to translate the ASCII-based resource into a binary file. The resource compiler will rewrite the original binary file and put all these PRELOAD segments into a contiguous area. Gangload can reduce the program load time, but it is only optional. If the programmer did not use gangload development, then the loader will not improve the load time, unless the generated executable file is modified later.

The reader should not get confused between gangload and our approach. Our approach can help reduce the program load time, with or without the presence of gangload. Gangload does not modify the segment attribute, but simply rearranges segment placement in the executable file. Though they are two different optimizations, our approach can take advantage of gangload. For example, if the original executable file does not contain a gangload area, the translator can create a gangload area after optimizing the segment attributes. If the original file already has a gangload area and the number of PRELOAD segments is increased, the translator can expand the original gangload area. The only limitation is that the size of the gangload area must be less than 1 MB. If the final PRELOAD size turns out to be larger than 1MB, gangload may only be used for part of the segments.

Chapter 9

Conclusions

Binary Translation plays an important role in software migration. In the foreseeable future, software will become the major driving force of computer evolution since it usually outlives hardware. Both developers and users tend to reuse software in order to reduce any unnecessary cost. Developers tend to reduce the software development cost by adding new features to the original kernel module. Users tend to use original software due to budget and familiarity.

Improving hardware technology may not be able to speed up the original binary program since the code compiled for one generation of hardware may not run faster on the next generation of hardware. Binary translation makes it possible to improve the performance of an executable program by creating a new executable capable of utilizing some if not all of the new processor's features.

9.1 Contribution

This thesis addressed some of the issues in binary translation. A robust framework was designed and implemented to translate the original binary code and execute the translated code. A run-time switching mechanism was proposed to allow the code size to change as well as to allow partial translation of the executable. A hybrid approach to resolving non-determinable branch targets was designed to minimize the run-time overhead. The switching mechanism allows the translator to avoid making potentially incorrect assumptions about the program's control flow. A self-modifying code detector was also implemented to prevent the binary translator from translating self-modifying code. Unlike other related work that is based on special hardware, a software approach based on the standard address mapping mechanism is used here to detect self-modifying code.

Since translation takes time, the translated code should be written to secondary storage for reuse. This thesis documents the detailed implementation issues for the write-back module to generate a correct executable file in place of the old file. Finally, this thesis proposed to improve the program loading time without recompiling the program or modifying the operating system.

9.2 Future Work

Though this thesis has made progress in solving some of the important problems in binary translation, there are still many research issues that are left to examine. The following sections outline some future research directions.

9.2.1 Performing Machine-Level Optimizations

Choosing a large memory model during compilation sometimes incurs performance loss by generating unnecessary far calls. This phenomena is ubiquitous in executable files. One example is the Windows Control Panel. This program contains only one code segment. However, there are several far calls whose targets turn out to be in the same segment as the callers. Each far call takes 4 machine cycles while each near call takes only one machine cycle. Each far call takes 5 bytes while each near call takes only 3 bytes.

This thesis provided a robust framework that allows the code size to change in the presence of undeterminable instructions. After run-time profiling information is available, advanced optimizations can be applied to the binary program to further improve the performance.

9.2.2 Removing Segment Operations

16-bit code may contain a lot of segment loading, which has a major negative impact on performance [5]. The address used in 16-bit code is formed by a 16-bit offset and a 16-bit segment or selector. 32-bit code uses a flat addressing model. In this model, the offset is 32-bit and all segments point to the same linear address. Since the translator is able to rewrite the binary code, it is possible to unfold the segmented address in 16-bit code to a 32-bit flat address.

Another way to reduce segment loading is by merging small segments to form a large segment. Most application programs do not contain large sized code segments. One of the largest benchmark programs, Excel, contains 248 code segments. The average segment size of this program is only 0x4026 bytes, which is approximately one fourth of the maximum segment size. Merging small segments can reduce the number of segments, which in turn, can reduce the overall time of segment loading. Merging code segments can also improve the situation described in Section 9.2.1. Inter-segment call instructions may be rewritable as near calls after code segment merging.

9.2.3 Translating 32-Bit Code

Translating a 16-bit Windows executable differs from translating a 32-bit Windows executable, depending on the operating system. In Windows 3.x, all programs share a single address space. In Windows 95, however, each 32-bit program has its own address space while all 16-bit programs share a common address space. In Windows NT, each program has its own address space. Having a private address space prevents a malfunctioning program from crashing other programs, or even the operating system. This additional restriction is imposed by the operating system. At machine level, it should not matter since all code is running as binaries, including the operating system code. However, since the kernel run-time manager, VMM, is built on top of Windows, translating 32-bit code may require modification of the VMM implementation.

Other issues in migrating 32-bit code include augmenting the binary front-end and the run-time switching mechanism. 32-bit executables use the *Portable Executable (PE)* file format. Reading the binaries as well as generating new binaries need to conform to this file format. For example, 32-bit programs use a flat memory model. As such, there are no inter-segment references any more. Every memory access is a near reference. Thus, the run-time support must be modified as well.

Though the 32-bit executable has no segment structure, there is still space for improving program loading time. Consider a source program that consists of 30 code segments after using the 16-bit compiler. Among these 30 code segments, 10 are marked as PRELOAD. Though these segments do not exist if the source program was compiled by the 32-bit compiler, not all portions of the executable code need to be loaded into memory at one time. By recording some run-time commands and analyzing the loading behavior, the translator may improve program loading time.

9.2.4 Porting 32-Bit Code to 64-Bit Code

By the time this thesis is written, the next generation of 64-bit x86 microprocessor, Merced, will be nearing completion of development [32]. In response to the new architecture platform, Microsoft is developing a 64-bit version of their NT operating system [11]. We expect to see a large volume of 16-bit and 32-bit binary programs running on the Merced. Given the speed of processor development, we believe binary translation will be critical to the performance of 64-bit processors for typical users.

References

[1] K. Andrews and D. Sand, "Migrating a CISC Computer Family onto RISC via Object Code Translation," *ASPLOS V*, pp.213-222, October 1992.

[2] T. Afzal, M. Breternitz, M. Kacher, S. Menyhert, M. Ommerman, and W. Su, "Motorola PowerPC Migration Tools - Emulation and Translation," *COMPCOM*, June 1996.

[3] D. Lee, T. Romer, G. Voelker, A. Wolman, W. Wong, B. Chen, B. Bershad, and H. Levy, "Instrumentation and Optimization of WIN32/Intel Executables," *http: // www.cs.washington.edu/homes/bershad/Etch/index.html*, University of Washington.

[4] B. Chen, M. Smith, and B. Bershad, "Morph: A Framework for platform-Specific Optimization," *http://www.eecs.harvard.edu/morph/*, Harvard University, March 1996.

[5] Personal talk with Susan Corwin, Center for Reliable and High-Performance Computing, Urbana, IL., March 15, 1997.

[6] V COMMUNICATIONS, INC, *Sourcer(TM) Commenting Disassembler* Reading, V COMMUNICATIONS, INC, San Jose, CA 95129, August 1994.

[7] Digital Equipment Corporation, *Ultrix v4.2* pixie Manual Page.

[8] Digital Equipment Corporation, "Technical Introduction to Digital FX!32," *http:// www.digital.com/info/semiconductor/amt/fx32/fx.html*, March 1996.

[9] K. Ebcioglu and E. Altman, "DAISY: Compilation for 100Compatibility," Report No. RC 20538, IBM T.J. Watson Research Center.

[10] C. Hunter and J. Banning, "DOS at RISC," *BYTE*, pp. 361–368, November 1989.

[11] "NT will have three faces," News, *Info World Electric*, Vol. 19, Issue 15, April 14, 1997. The web site address is http://www.infoworld.com/cgi-bin/displayArchive.pl?/97/15/t02-15.1.htm

[12] Intel, "Pentium Family User's Manual, Volumn 3: Architecture and Programming Manual," Intel Corporation, 1994.

[13] N. Kronenberg, T. R. Benson, W. M. Cardoza, R. Jagannathan, and B. J. Thomas, "Porting OpenVMS from VAX to Alpha AXP," *Communication of the ACM*, pp. 45–53, February 1993.

[14] J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *ACM SIGPLAN '95 Conferences on Programming Languages Design and Implementation (PLDI)*, pp.291-300, June 1995.

[15] J. R. Larus and T. Ball, "Rewriting Executable Files to Measure Program Behavior," Software Practice and Experience, 24(2): 197-218, February 1994.

[16] D. Long and D. Ruder, "Introduction to Microsoft Windows Dynamic-Link Libraries," Microsoft Developer Network Library, Technical Articles: Windows Articles: Kernel Articles, August 1992.

[17] D. Long and D. Ruder, "Mechanics of Dynamic Linking," Microsoft Developer Network Library, Technical Articles: Windows Articles: Kernel Articles, January 1993.

[18] Microsoft, "Executable-File Header Format," *Microsoft Knowledge Base*, Article number: Q65122, February 1996.

[19] Microsoft, Microsoft Windows Software Development Kit, Version 3.1, Microsoft Part No. 30211, 1992.

[20] D. A. Norton, "Writing Windows Device Drivers," Addison-Wesley, 1992.

[21] C. Petzold, *Programming Windows: the Microsoft guide to writing applications for Windows 3.1*, Microsoft Press, Redmond, WA, 1992.

[22] M. Pietrek, *Windows Internals*, Addison-Wesley, 1993.

[23] M. Pietrek, "Windows Question and Answer," *Microsoft Systems Journal*, Number 9, September 1994.

[24] M. Pietrek, "Windows Question and Answer," *Microsoft Systems Journal*, Number 10, October 1994.

[25] A. Schulman, D. Maxey, and M. Pietrek, *Undocumented Windows*, Addison-Wesley, 1992.

[26] G. M. Silberman and K. Ebcioglu, "An architectural framework for supporting heterogeneous instruction-set architectures," *IEEE Computer*, pp. 39–56, June 1993.

[27] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communication of the ACM*, pp. 69–81, February 1993.

[28] R. Sites, "Branch resolution via backward symbolic execution," *U.S. Patent*, No. 5428786, Mar 1991.

[29] Sun Microsystems Incorporation, "The Wabi(TM) White Paper," Sun Microsystems Incorporation, Mountain View, CA, 1996.

[30] T. Thompson, "An Alpha in PC Clothing," *BYTE*, pp.195-196, February 1996.

[31] R. Wahbe, S. Lucco, and S. L. Graham, "Adaptable binary programs," Tech. Rep. CMU-CS-94-137, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1994.

[32] A. Wolfe, "Parallelism is part of P7 picture," EE Times, Oct 12, 1996. The web site address is http://techweb.cmp.com/eet/news/p7web.html.

Vita

Liang-Chuan Hsu was born in 1962, in Ping-Tung, Taiwan. He pursued his undergraduate studies at the Chung-Cheng Institute of Technology, Tao-Yuan, Taiwan, where he received the B.S. degree in Vehicle Engineering in 1986. After receiving the B.S. degree, he joined the Taiwan Army with the rank of First Lieutenant, and served as a Teaching Assistant at the Chung-Cheng Institute of Technology. In 1990, he completed the M.S. degree in Computer Science at the Naval Postgraduate School, California. After receiving the M.S. degree, he continued to serve in the Taiwan Army as an Instructor at the Chung-Cheng Institute of Technology. In the fall of 1993, he began his graduate studies in Computer Science at the University of Illinois in Urbana, Illinois. During his graduate tenure at the University of Illinois, he has been a member of the Center of Reliable and High-Performance Computing and the IMPACT project directed by Professor Wen-mei W. Hwu. In 1994, he advanced the military rank to Major. After completing the Ph.D. work, he will continue to serve in the Taiwan Army as an Assistant Professor at the Chung-Cheng Institute of Technology.