A FRAMEWORK FOR PROFILE-DRIVEN OPTIMIZATION IN THE IMPACT BINARY REOPTIMIZATION SYSTEM

BY

MATTHEW CARL MERTEN

B.S., University of Illinois at Urbana-Champaign, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ACKNOWLEDGMENTS

I first want to thank Professor Wen-mei Hwu for providing me with the opportunity to develop my research skills in the IMPACT research group. I appreciate his devotion to teaching and to high academic and research standards. While I am the only researcher to have worked on the binary reoptimization project from the beginning, many other people have had a great deal of influence on the project, and on me. Michael Thiems and I, together, built the foundation for each of the tools in our binary reoptimization system. He is a talented and skilled engineer from whom I have learned a great deal, and with whom I have enjoyed working. John Gyllenhaal, who has a vast knowledge of computer architecture and compiler technology, especially concerning IMPACT, assisted us in understanding the necessary features of our reoptimizer and helped us interface several standard IMPACT components to our system. For the past year, Chris George and I strove to implement more advanced AMD K6 optimizations. I especially thank him for listening to me ramble about ideas and questions, and for helping me analyze our optimizations. Thanks to everyone, past and present, in the IMPACT research group, especially John Sias for his early contributions and for his friendship, Dan Connors for his useful insights and his ECE 312 teaching abilities, and Dan Lavery for his mentoring efforts in the early days of my IMPACT membership.

In addition, I want to thank our corporate research partners. First, thanks to Advanced Micro Devices, in particular Dr. David Christie. The AMD team has provided hardware, software tools, and invaluable insight into the K6 microarchitecture and into the entire software development process. Thanks to Microsoft for their donation of the Microsoft Developer

Network tool set and to Microsoft Research for our post-link optimization discussions. And thanks to Hewlett-Packard for my internship opportunity and for general support of the IMPACT research group.

Last, I want to thank my family for their constant support. My wife, Polly, is understanding about my long work hours, encourages me when I get discouraged, celebrates with me in my triumphs, and in general, is the loving support that drives me each day. My parents, parents-in-law, and sister are always confident in my abilities and are always encouraging me to pursue my dreams. Finally, I wish to acknowledge and thank the Lord Jesus Christ for his love, grace, and rich blessings on my life.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1
1.1 Related Work 3

2 BINARY REOPTIMIZATION FRAMEWORK 5
2.1 Basic Optimization Process 5
2.2 Profile-Based Optimization Process 7
2.3 Application Requirements for Reoptimization 9

3 SECTION TRANSLATION 10
3.1 Binary Creation 11
3.2 Binary Analysis and Optimization 11
3.3 Binary Decoding 15
3.4 Symbol Table 16

4 RELOCATION TABLES 20
4.1 Pointer Determination 20
4.2 Base Relocation Table 22
4.3 Relocation Table 25

5 MCODE INTERMEDIATE REPRESENTATION 27
5.1 Mcode Functions and Control Blocks 28
5.2 Mcode Instructions 28
5.3 Mcode Operands 30
5.4 Exceptions to the Typical Instruction Format 33

6 CONVERSION OF ASSEMBLY TO MCODE 38
6.1 Conversion of Functions and Basic Blocks 38
6.2 Conversion of Instructions 42
6.3 Conversion of Operands 43

7 BINARY PROFILING 48
7.1 Profiling Probe Insertion 49
7.1.1 Direct branches and jumps 51
7.1.2 Indirect jumps 51
7.1.3 Call graph collection 55
7.2 IMPACT Probe Descriptor File 57
7.3 Profile Counter Extraction 59
7.4 Speedup of Probed Executables 61

8 BASIC SUPPORT FOR MEMORY DISAMBIGUATION 63

9 PROFILE-BASED OPTIMIZATION 65
9.1 Function Layout 66
9.2 Superblock Formation 69
9.3 Push and Pop Chain Optimizations 71

10 FUTURE WORK AND CONCLUSION 75

REFERENCES 77

LIST OF TABLES

Table | Page

5.1 Data types and their sizes. 30
5.2 Comprehensive list of available register operands. 33

9.1 Superblock formation parameter settings. 70

LIST OF FIGURES

Figure Page

2.1 The basic binary optimization process. 6
2.2 The binary optimization process with branch profiling. 8

3.1 Linking of object files into a linked binary. 12
3.2 Section decode, optimization, and reconstruction of a linked binary. 13
3.3 Example labels used within the optimizer. 14
3.4 Partial disassembly of functions `_main()` and `_readfile()` as compiled by Microsoft Visual C++ 6.0. 17

4.1 Code snippet from `_memmove()`. 21
4.2 Base relocation table and annotated assembly code. 24
4.3 Relocation table and annotated assembly code. 26

5.1 Mcode representation of functions and basic blocks. 28
5.2 Mcode representation of a *multiply* instruction. 29
5.3 An example of overlapping subset registers. 31
5.4 Breakdown of a superset register into its constituent subset registers. 32
5.5 Mcode representation of a *push register* instruction. 35
5.6 Mcode representation of a *push memory* instruction. 35
5.7 Mcode representation of an *ASCII adjust* instruction. 36
5.8 Mcode representation of a *floating-point reverse divide* instruction. 37

6.1 Pseudocode algorithm for converting functions, basic blocks, and instructions into Mcode. 40
6.2 Pseudocode algorithm for converting explicit operands into Mcode. 44
6.3 Mcode representation of an *enter* instruction. 44
6.4 Pseudocode algorithm for converting possible relocatable operands into Mcode. . . . 46
6.5 Pseudocode algorithm for converting label operands into Mcode. 47

7.1 Mcode probes using the *increment memory* instruction with surrounding save and restore of the flags. 50
7.2 Mcode probe insertion for a basic block ending in a *conditional branch*. 52
7.3 Mcode probe insertion for a basic block ending in an *indirect jump*. 54
7.4 Call graph profile mechanism. 56
7.5 IMPACT probe descriptor file record formats. 58
7.6 Example of an IMPACT probe descriptor file for SPEC95 *129.compress*. 60
7.7 Probed application speedups. 62

9.1 Speedups with the three function layout optimizations applied. 68

9.2 Speedups for the superblock formation algorithms in addition to CG function ordering and K6 optimizations. 71
9.3 Example of the stack allocation (push) optimization. 72
9.4 Speedups for the stack allocation algorithms in addition to CG function ordering and K6 optimizations. 74

CHAPTER 1

INTRODUCTION

Since the invention of the computer, higher application performance has been in incessant demand. Even orders-of-magnitude performance improvement has not appeased users as they continue to dream of new, more demanding applications. This pursuit has led to aggressive development at every level of computer design, from semiconductor to compiler technologies. Improved semiconductor technology contributes higher die densities, which allows for an increase in the number of transistors while also allowing higher clock frequencies. Computer architectures now may allow coarse-grained parallelization, speculation, and predication. Advanced computer microarchitectures are able to exploit fine-grained parallelism more effectively while also executing instructions more efficiently than in previous microarchitectures. Memory systems have improved to reduce the average latency of accesses to memory by including multiple levels of caching, and by adding new types of caches, such as victim caches. Compilers provide aggressive classical and instruction-level-parallelism optimizations and pointer analysis to reduce the number of instructions executed, and to provide in-depth analysis of the application. These and other improvements have all lead to exponential growth of computer performance.

In order to achieve maximum application performance, however, the improvements made must be effectively exploited. Within the IMPACT research group, research efforts have focused on co-development of the various hardware components and compiler technologies to ensure that the generated code matches well with the underlying hardware. The system described in this thesis adds another component into the compilation process in an effort to further conform the code to the hardware.

As previously mentioned, high performance can only be achieved when applications effectively use the various resources in the microprocessor. However, software vendors traditionally compile code completely to executable format without much regard for the specifics of the target processor model. In essence, the lowest-common-denominator model is chosen and compiled for, thus allowing users with older machines to run the applications. This software distribution model often prevents the use of newer features (for example, MMX capabilities), and can cause mismatches between the microarchitecture and the code (for example, desired instruction ordering for efficient decoding).

Recent changes to this software distribution system have allowed for the inclusion of multiple versions of the software within a single distribution, each targeting a specific model. Specifically, one approach has been to group functions into libraries, where multiple model-specific versions of those libraries are included and selected at install or library load time. Or, a global variable can be tested during execution that indicates a particular model. Based on this variable, the best code can then be executed. These approaches rely on the software vendor to build and distribute multiple copies of the various code segments.

Another approach is to construct an optimizer that customizes applications when they are installed on the target system. This is the approach that is investigated in this thesis. Such a system is desirable as many different models exist within a processor family with competing, compatible families also available. This makes the inclusion of custom pieces of code for each model infeasible. Use of an install-time optimizing system may lower the time-to-market of applications as less model-specific code may need to be developed. Install-time optimization capabilities may also increase the performance benefit of a new processor model. When a user purchases a new processor, the appropriate reoptimizer may be packaged with it, allowing for immediate reoptimization of the user's executables for the new processor.

1.1 Related Work

Other systems have been proposed and developed that attempt to tackle similar or related problems including Etch [1], Morph [2], and Fx32! [3]. A summary of these other efforts can be found in [4].

Other proposed systems have suggested that the final binary be constructed from an intermediate representation at run time. This would allow for customization for the target model because the model-specific portion of compilation would not be performed by the vendor, but deferred until run time. In such a system, the intermediate representation could be immediately optimized and compiled into binary the first time the code is executed, or interpreted for a period of time while profile information is gathered, followed by optimization and compilation. Current Java systems operate much in this way, where the bytecode can be considered an intermediate representation. However, such a software distribution system *requires* intervention by the compiler on the user's machine because the processor cannot directly execute the intermediate representation. In other words, it is not possible to turn off the compiler, even if it contains bugs. Furthermore, applications running in such environments tend to execute quite slowly during the early phases of execution as many instructions are interpreted or compiled.

Research on compilation for x86 architectures began approximately six years ago within the IMPACT research group [5], [6]. The x86 binary reoptimization framework presented in this thesis employs a new intermediate representation and tools, but leverages heavily off of past experiences.

CHAPTER 2

BINARY REOPTIMIZATION FRAMEWORK

The IMPACT binary reoptimization framework is capable of reoptimizing code contained in three different types of binary files: the *Portable Executable* (PE) format [7], [8], the *Dynamic-Link Library* format (DLL), and the *Common Object File Format* (COFF). The Portable Executable and the Dynamic-Link Library formats are the two common (and similar) formats that contain linked code ready for execution. The Common Object File Format contains compiled object code prepared for linking.

The following sections present an overview of the steps taken to reoptimize a binary. In addition to the descriptions found in this thesis, further detail into select topics can be found in a joint technical report by Merten and Thiems [9], a master's thesis by M. S. Thiems [10], and a master's thesis by C. N. George [4].

2.1 Basic Optimization Process

Figure 2.1 depicts the basic optimization process. The first step, *x86toM*, converts the binary file into IMPACT's intermediate representation called *Mcode*. This step actually consists of two phases. In the first phase, code discovery algorithms are employed to find instructions and collect them into basic blocks and finally into whole functions. A temporary representation of the code, which is similar to Intel assembly code, is used in this first phase. Then in the second phase, the code is converted into Mcode.

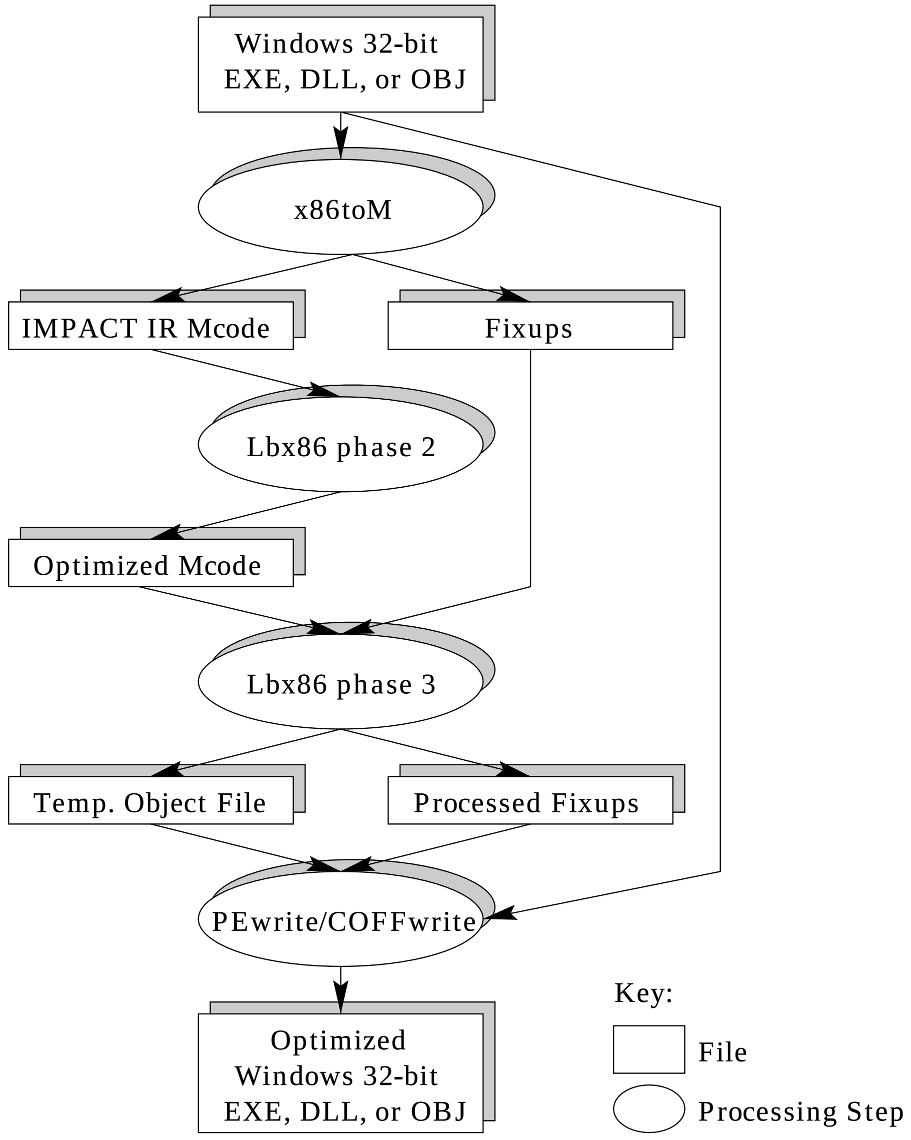


Figure 2.1 The basic binary optimization process.

Optimization is performed in the second step, called *Lbx86 phase 2*. The qualifier *phase 2* is used for historical reasons to indicate the optimization phase of a code generator. A variety of optimizations can be performed on the code in this phase: scheduling, both machine-independent and machine-specific optimizations, etc.

After optimization, the code is converted directly to a binary format in the third step, called *Lbx86 phase 3*. Again, the qualifier *phase 3* is used for historical reasons to indicate the final conversion of the code to assembly format, or in this case, binary format. A custom direct-to-binary converter is desirable because traditional assemblers do not simply convert assembly to binary, but rather perform certain optimizations. One such optimization concerns instruction selection where the particular opcode and operand layouts are chosen to minimize the instruction's encoded size. However, it is sometimes desirable to have instructions of a larger size, as specified by the optimizer. Using a traditional assembler would undo such an optimization performed by the optimizer. The output of this phase is a temporary object file containing the optimized text and data sections.

Finally, the fourth step, called *PEwrite* (for PE and DLL files) or *COFFwrite* (for COFF files) merges the optimized code stored in the temporary object file back into a copy of the original binary file. By merging the code into a copy of the original, certain portions of the original file need not be translated or maintained throughout the optimization process. Large portions of code and data have been converted to the intermediate representation, while some smaller portions along with elements that are not strictly code or data have been communicated through the fixup file.

2.2 Profile-Based Optimization Process

This framework also supports profile-driven optimization of PE and DLL files. Two distinct passes through Lbx86 phase 2, Lbx86 phase 3, and PEwrite must be made, as shown in Figure 2.2. In the first pass, probes to collect edge execution weights are inserted by Lbx86 phase 2. Lbx86 phase 3 and PEwrite generate and reconstruct the binary as usual, including

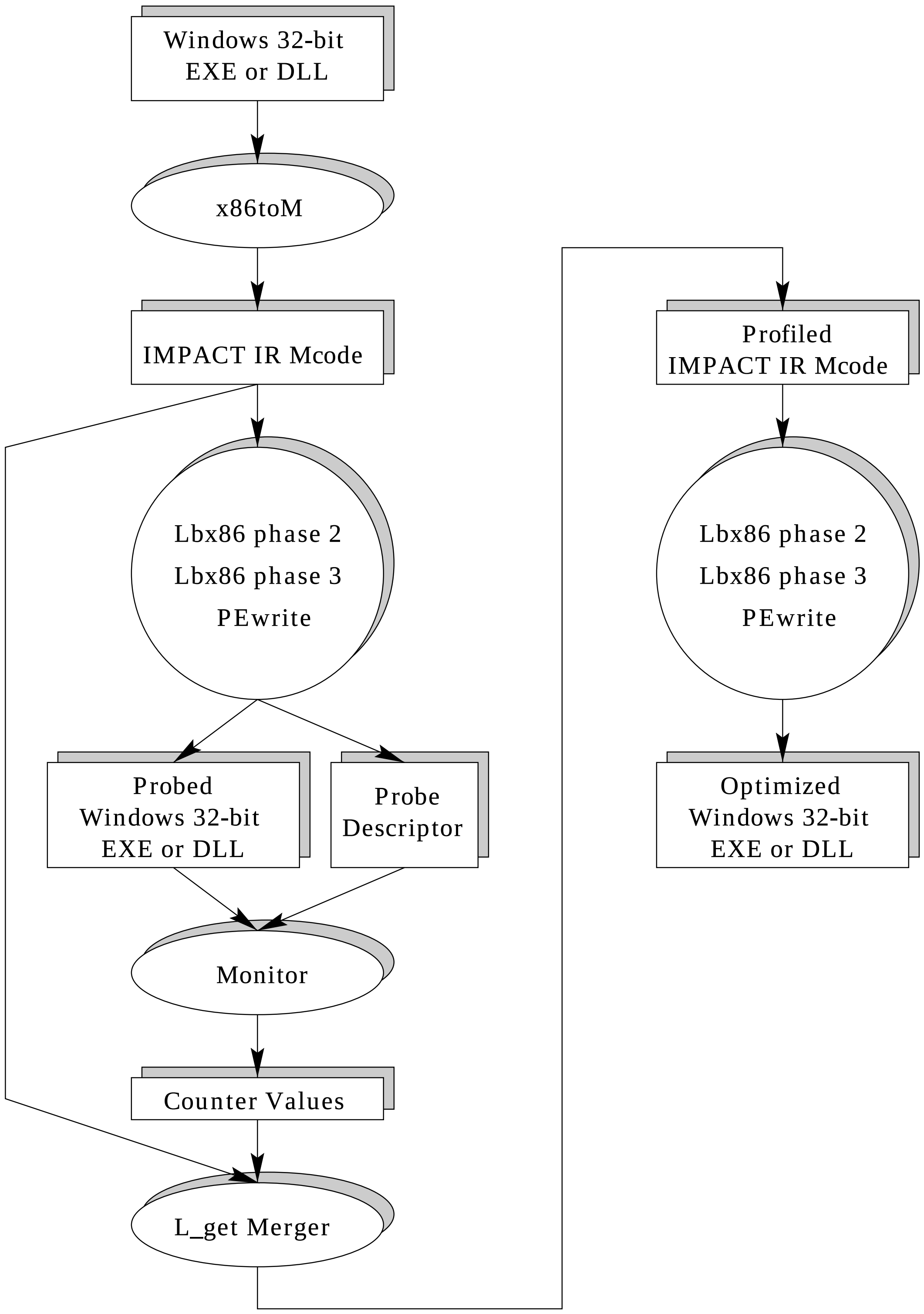


Figure 2.2 The binary optimization process with branch profiling.

the probes and the counter table. When the probed application is executed, the counter table is updated by the probes. In order to read out the counter values from the table, a monitoring application launches the probed application and is notified by the operating system immediately prior to termination of the probed application. Following the extraction of the counter values from the probed application's data space, the application is allowed to terminate. The counter values are then reformatted, allowing *Lget* to annotate the Mcode with profile weights. Profile-driven optimizations can be applied in the second pass of Lbx86 phase 2 to the profile-annotated Mcode.

2.3 Application Requirements for Reoptimization

The reoptimization framework requires that PE and DLL files have a *base relocation table*. This table contains information about the pointers contained within the executable code and data. The binary reoptimization framework simply uses the table to identify the constants that should be treated as pointers and which should be treated as simple numbers. The base relocation table will be further discussed in Chapter 4. In addition, the presence of a *symbol table* helps the framework to delineate function boundaries, but the presence of this table is not required.

COFF files, by definition, must always contain both a *relocation table* and a symbol table for linking purposes. The relocation table is an enhanced base relocation table in the sense that it indicates where the pointers are in the code, but it also indicates what the pointers actually point to. Since both tables are required for linking, the framework uses the information present in those tables. There are no additional requirements for translation of COFF files.

CHAPTER 3

SECTION TRANSLATION

Portable Executable files, Dynamic-Link Libraries, and Common Object File Format files all consist of a collection of *sections.* A section may contain executable code, initialized data, uninitialized data, debugging information, constant data, supplemental information about the binary such as a base relocation table, etc.

In COFF object files, the code for each function is usually contained within its own section. Certain types of constant data, such as jump tables, may also be located with the code. Compilers differ on the grouping of other types of data. For example, the Microsoft Visual C++ compiler groups global initialized data from a single source file together into a section. It also groups global constant data together and global uninitialized data together into their own sections. However, local constant data (such as strings) are often each given their own section. This explicit sectioning allows the linker to easily include in the final binary only the functions that can be potentially called, leaving out the unused functions, or *dead code.*

However, PE and DLL files generally contain only one section for each type of information because the operating system loader must individually load and place each section into memory. By having fewer sections, there is less loader bookkeeping overhead during the loading process. The linker is responsible for merging together sections with similar characteristics during final binary creation.

Microsoft's compiler names sections containing functions as `.text`, global initialized data as `.data`, uninitialized data as `.data` or `.bss`, and global constant data as `.rdata`. Borland's compilers name them `CODE`, `DATA`, `.bss`, and `.rdata`. For simplicity, the Microsoft terminology (without the preceding period) will be used throughout the remainder of this thesis.

3.1 Binary Creation

Before beginning an in-depth discussion on section translation, a brief summary of binary creation is in order. This process is shown in Figure 3.1. In a typical compiler, each function and its associated data (jump table data, etc.) is compiled separately into assembly code, barring any interaction due to function inlining and interprocedural analysis. Each function is then converted to binary representation within its own text and data sections in the object file. An example is shown in column (a) of Figure 3.1 which depicts an object file containing two functions and two associated data sections. The linker then *maps* all of the text sections into a single, large text section, and likewise maps all of the data sections. The result is shown in column (b). Next, all of the sections are *linked* together, as all of the symbolic pointers are resolved into addresses, as shown in column (c). Finally, the binary is written, and most of the information about individual function and data locations is lost. A symbol table may be written which may assist in the delineation of the functions and data within the binary executable. Symbol tables will be discussed further in Section 3.4.

3.2 Binary Analysis and Optimization

In order to optimize the code, the actual instructions, blocks, and functions must be discovered and collected in a process called *binary decode*, as shown in Figure 3.2. The text and data sections are the only sections likely to benefit from optimization and, thus, are the only sections decoded and translated into Mcode. Other sections may contain information required by the

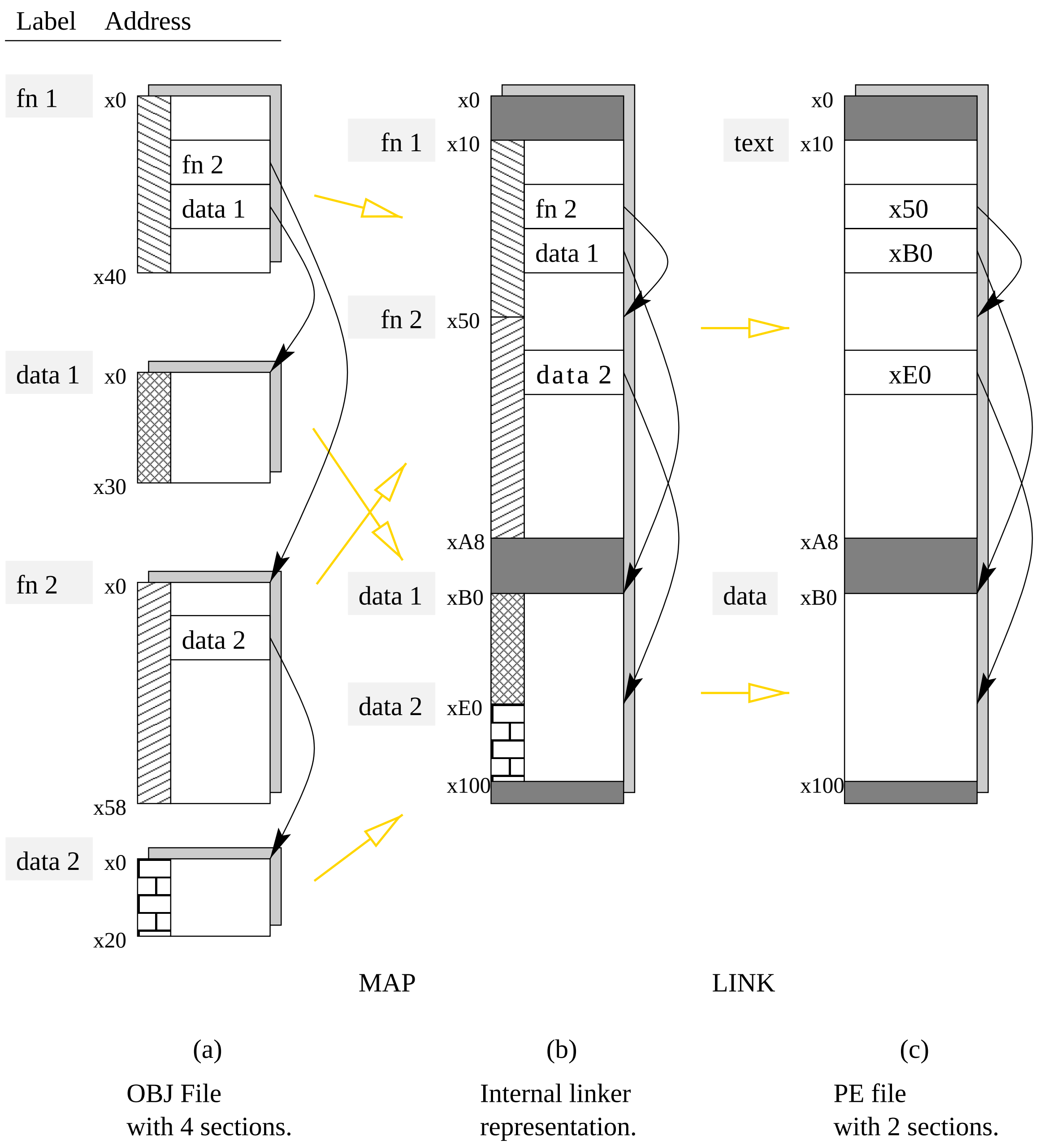


Figure 3.1 Linking of object files into a linked binary.

decoder, but those sections are only partially decoded and referenced upon demand. Because the optimizer cannot efficiently analyze and optimize the code for an entire program at once, it is broken up into *units of compilation*. This framework, like most optimizers, uses *functions* as

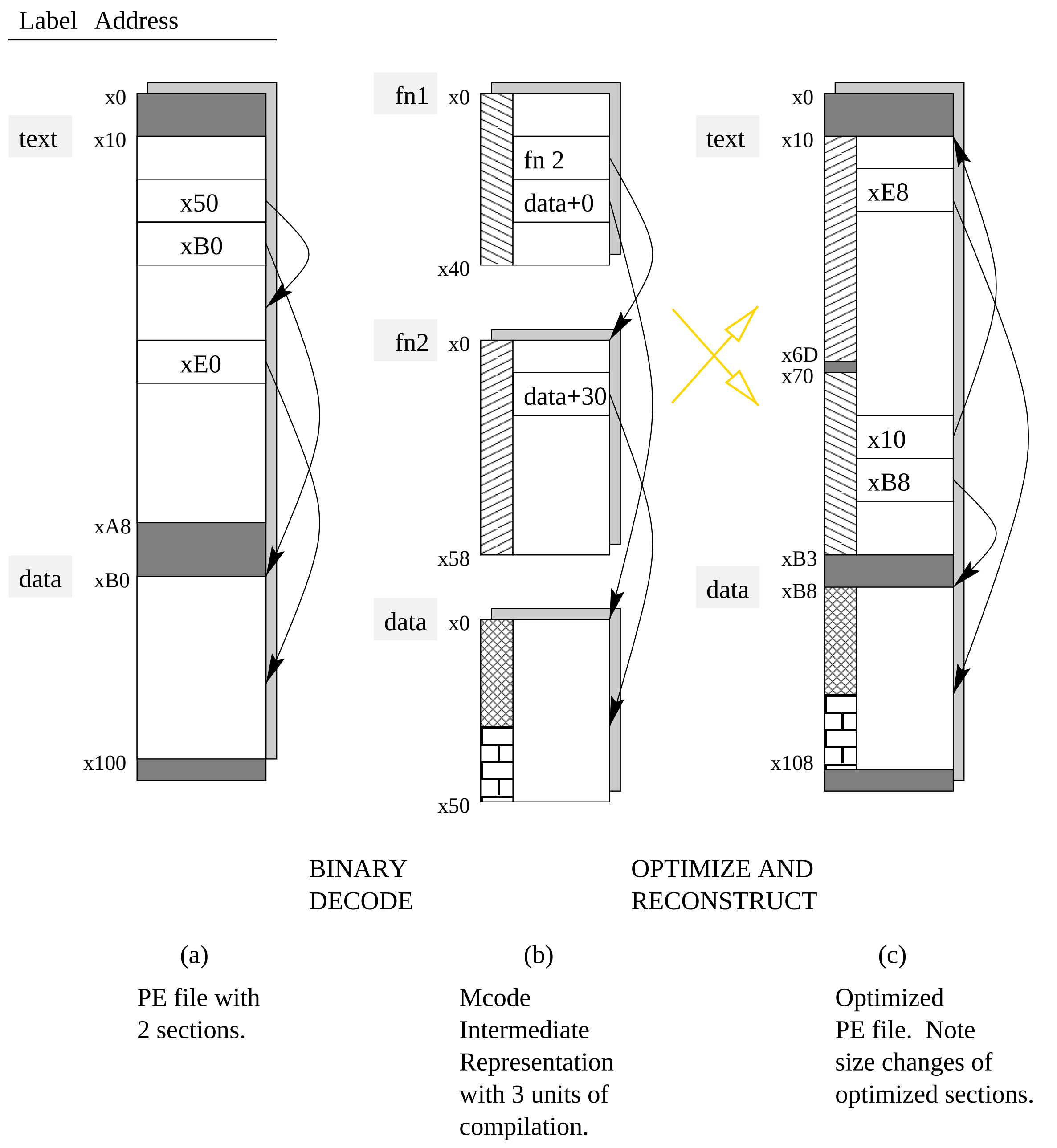
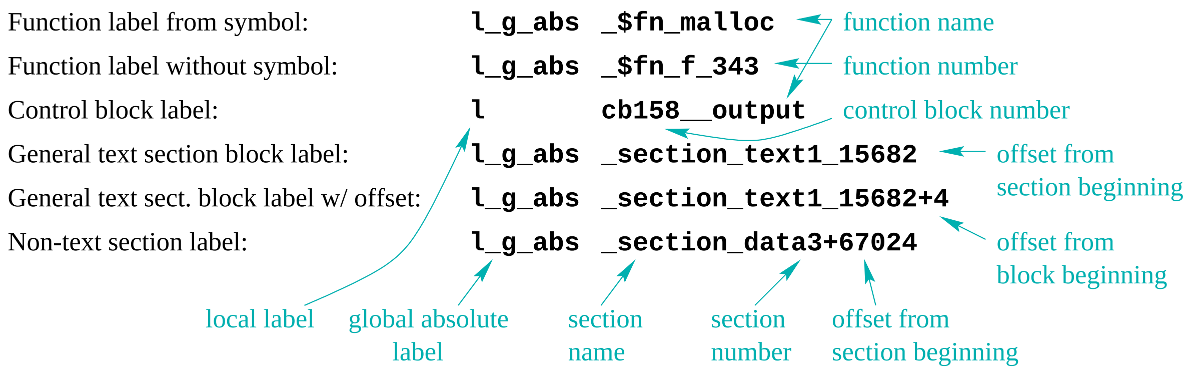


Figure 3.2 Section decode, optimization, and reconstruction of a linked binary.

its unit of compilation, although other units, such as *regions* [11] which are supported within the IMPACT compiler, are also viable options. As shown in column (b) of Figure 3.2, the output of the decoder is a collection of functions and data. Note that the data section, however,



Function label from symbol:	`l_g_abs _$fn_malloc`
Function label without symbol:	`l_g_abs _$fn_f_343`
Control block label:	`l cb158__output`
General text section block label:	`l_g_abs _section_text1_15682`
General text sect. block label w/ offset:	`l_g_abs _section_text1_15682+4`
Non-text section label:	`l_g_abs _section_data3+67024`

Figure 3.3 Example labels used within the optimizer.

is not specifically optimized and remains as a single data section throughout the optimization process.

The functions in the optimized binary will likely be different sizes and located at different addresses than in the original binary. The example in Figure 3.2 column (c) shows that the functions have been reordered and also have changed slightly in size. Since there are layout and size changes, the optimizer cannot use addresses as pointers within instructions. Instead, all functions are given names, or *labels*, possibly taken from the symbol table if it is present, when they are translated into the intermediate representation. All pointers into the text section are converted into label references, examples of which are shown in Figure 3.3, which will be resolved back into addresses before the optimized binary is written. Note that noncode regions within the text section will also be broken up into blocks. Those blocks will be given a block name that consists of the section identifier appended with the block's starting offset from the beginning of the section in the original binary. An additional offset from the beginning of the block may be represented as an offset appended to the block name.

The data section is also translated into the intermediate representation, but with the limited information available within a binary, it cannot yet be reliably broken into pieces. In effect, the entire data section is treated as a single block. Because growth of the text section can change the placement of the data section in memory, all pointers into the data section must also be altered and, therefore, are converted into label references. These pointers consist of a label to the beginning of the data section appended with an offset from the beginning of that section, also shown in Figure 3.3.

The other sections in the binary are not converted into intermediate representation, and thus measures must be taken when information within those sections must be altered. Pointers from the other sections into the unconverted sections are altered in the same way that pointers into the data section pointers are altered. Pointers out of those unconverted sections are tagged as *fixups*. The fixup file contains a list of such pointers and their target labels. Those pointers are then altered, or fixed up, when the new binary is generated.

3.3 Binary Decoding

As shown in Figure 3.2, the first step in the reoptimization process involves decoding the binary and converting it into Mcode intermediate representation. This process is performed by the x86toM tool. The code within the text section is broken into functions via a *decode and flow analysis* algorithm, as described in [4]. Essentially, the process iteratively decodes an instruction and then follows all of its possible control paths. More specifically, the decoder keeps track of the addresses of all previously explored instructions and a stack of addresses of control flow targets yet unexplored. The decoding is performed in a depth-first fashion in which, at a *conditional branch* instruction, the fall-through path is first explored while the address of the instruction in the taken path is pushed onto the unexplored stack. When decoding reaches a *return* instruction or previously decoded instruction, decoding resumes at the address on the top of the unexplored stack. Note that decoding begins with the entry point to the application, often `_mainCRTStartup()` for Microsoft-compiled C applications.

3.4 Symbol Table

The *symbol table* is an inherent component of all COFF files and an optional component of PE and DLL files. In COFF files, inter-section pointers exist as symbol references, or labels, until link time, thus necessitating a list of symbols and their addresses. Since all internal pointers have been resolved in PE and DLL files (external pointers are handled through *import* and *export* tables), symbols are only required for debugging purposes.

Symbol table entries may assist in binary decode in several ways. First, all functions have entries in the symbol table. In the same way that an export table assists in decoding, functions that may have not been previously discovered through standard flow analysis may now be located and decoded. This can be particularly beneficial when determining whether an immediate address within an instruction points to a data block or a function. For instance, a user function pointer may be passed as a parameter to a system or library function in a construct called a *callback*. Likewise, a pointer to a constant character string may be passed to a function such as `_printf()`. Both the function pointer and the string pointer appear as a constant value in a stack *push* instruction. Without interprocedural analysis or a symbol table, the decoder cannot reliably determine which are function and which are data pointers.

A unique feature present in Microsoft Visual C++ 6.0 generated code is the absence of a *return* instruction after calls to the C Library function `_exit()`. Since control will never return

```
_main
  ...
  00401245:E8 E3 FA 02 00      call    _exit
  0040124A:90                  nop
  0040124B:90                  nop
  0040124C:90                  nop
  0040124D:90                  nop
  0040124E:90                  nop
  0040124F:90                  nop
_readfile:
  00401250:83 EC 50            sub     esp, 50h
  ...
```

Figure 3.4 Partial disassembly of functions `_main()` and `_readfile()` as compiled by Microsoft Visual C++ 6.0.

from the `_exit()` function, the compiler does not generate the *return* instruction after the *call* instruction. This poses an interesting situation for the framework's flow decoding routines, as the algorithms always expect that a function call returns and resumes execution. When the decoder begins decoding instructions following the call to `_exit()`, data and filler are incorrectly converted to instructions, as shown in Figure 3.4. This situation poses several more complicated problems.

One such complication is that flow decoding may continue decoding straight into another function. This leads the decoder to believe the two functions are actually a single function with multiple entry points. An example of this situation can be seen in Figure 3.4 as the series of *no operation*, or "`nop`", instructions suggest control flow can follow the *call* into the *subtract*, thus linking the two functions together. This is not necessarily an adverse situation as the framework correctly handles multiple entry-point functions, but this can complicate the optimization process because of false dependences.

A potentially more significant problem is that data may be incorrectly classified as code. Suppose that the series of *no operations* is really a series of data bytes. If optimization is performed on this combined function, the *no operations* might very well be eliminated, thus eliminating the data table! In other cases, series of data bytes translate into a sequence of more diverse instructions. The optimizer may perform optimizations on this supposed code sequence thus altering the data table. Clearly, these situations are unacceptable for correct translation.

These complications are even worse when conversion of data to code alters instruction alignment. Consider the situation when a single byte exists between the call to `_exit()` and the next function. Now, suppose that this byte happens to be decoded as the first byte in a two-byte opcode. When this supposed instruction is finished decoding, the decoder will begin decoding after the supposed instruction, which is actually the second byte of the first instruction of the next function. In this scenario, the second function is now incorrect due to misaligned decoding.

Two solutions are immediately available to solve the `_exit()` problem, both requiring that the decoder be informed of this special case. The simplest solution is to use the symbol table to detect calls to `_exit()`. When such a call is seen, no successive instructions are expected, and decoding continues elsewhere. This method is currently implemented in the binary reoptimization framework. The other solution is to locate the system exit routine in the import table and trace back through the call sequence, marking call sites to which control flow will not return. However, this information must be available before decoding past any call. Therefore, a depth-first search of the call graph must be made until leaf functions are reached where determination of the return can be made. This is generally complicated because the call graph is a potentially cyclic graph. Furthermore, the function `_doexit()` (called by `_exit()` and other functions) may or may not actually call the system exit routine depending on a parameter passed to it. This means that `_doexit()` may return, which suggests that `_exit()` might also return. However, `_exit()` does, in fact, always call `_doexit()` with the system exit parameter, but the only way for the optimizer to determine whether `_exit()` returns is to perform an interprocedural constant propagation analysis on `_doexit()`.

With the presence of a symbol table, it may be possible to reliably break the data section into pieces as is done with the text section. The symbol table could be used to locate the beginning of each data structure and, thus, all boundaries between structures. However, this assumes that each symbol in fact points to the beginning of a structure and never into the middle of one. This is generally true in compiled code but may not be true in assembly generated code. For example, a programmer may want a symbol for both the beginning and ending elements of an array. Using a heuristic like the one mentioned here would cause that one structure to be broken into two pieces. And because the primary reason to break the data section into pieces is to re-layout the structures, the two pieces may not end up back-to-back in the reoptimized binary.

CHAPTER 4

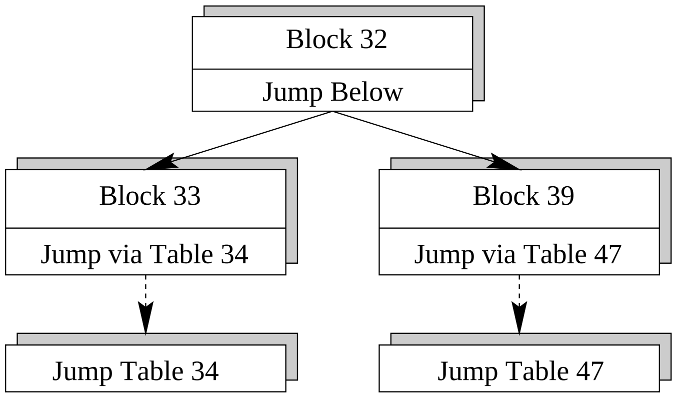
RELOCATION TABLES

The discovery of code reached only via an indirect call or jump is a difficult problem. Compilers, with full knowledge of the original source code, are often able to resolve indirect calls to a set of possible functions. Furthermore, compilers always know the set of possible indirect jump targets, except possibly in the case of a `_longjump()`, because the compiler itself converted specific control flow instructions into an indirect construct. However, it is considerably more difficult for a binary optimizer without the source-level information to draw the same conclusions.

4.1 Pointer Determination

To illustrate the difficulty in resolving indirect jumps into a set of target basic blocks, consider Figure 4.1 which depicts a code snippet from the C library function `_memmove()`. The snippet contains two indirect jumps through jump tables (Blocks 33 and 39) and associated control logic (Block 32) which selects from the two available jump tables.

The code discovery mechanism faces two primary challenges when analyzing code containing pointer references. First, the mechanism must determine which immediate and displacement values within instructions are pointers and which are simply data. For example, the bytes '`08 00 00 00`' in the third instruction of Block 32 could be a pointer to a function or possibly to another basic block, or could be simply data. Likewise, the bytes '`A0 71 43 00`' in



```
control block 32:
  00437068 :  C1 E9 02                ; shr   ecx, ecx, 02h
  0043706B :  83 E2 03                ; and   edx, edx, 03h
  0043706E :  83 F9 08 00 00 00       ; cmp   ecx, 00000008h
  00437074 :  72 0D                   ; jb    00437080

control block 33:
  ...
  00437077 :  FF 24 95 A0 71 43 00    ; jmp   dword ptr [4*edx+ 004371A0h ]

control block 39:
  00437080 :  F7 D9                   ; neg   ecx, ecx
  00437082 :  FF 24 8D 50 71 43 00    ; jmp   dword ptr [4*ecx+ 00437150h ]


jump table 47:  named _section_text1_221492
  00437134 :  00437154                ; (cb 48)
  ...
  00437150 :  00437197                ; (cb 55)

jump table 34:  named _section_text1_221600
  004371A0 :  004371B0                ; (cb 38)
  ...
  004371AC :  004371DC                ; (cb 35)
```

Figure 4.1 Code snippet from _memmove().

the instruction in Block 33 could be either a pointer or raw data. In these two contexts, the determination can be made through a few, very probable heuristics. In the first example, the code discovery algorithm can distinguish between memory regions in 32-bit Windows applications that can contain code and those that cannot. Thus, user code cannot exist at address `0x00000008` which is pointed to by those bytes.

Furthermore, if those bytes were a pointer, then the `ecx` register must also contain a pointer. A complex code analysis algorithm would see `ecx` being shifted to the right by an earlier instruction, which is an unlikely operation on a pointer.

Considering the second example instruction, the jump target must be into user code space. Register `edx` is not likely to contain a pointer because it is being multiplied by a constant and is therefore an index off of a displacement address. However, both determinations have been made by heuristics, and it is very possible for a devious assembly programmer to write functional code that violates these heuristics.

4.2 Base Relocation Table

The *base relocation table* in PE and DLL files, as introduced in Section 2.3, solves this pointer determination problem whenever it exists. This table basically contains a list of addresses within the program that contain pointers. When a binary file is created by the linker, the file is given a *preferred base address*. This address specifies the desired address at which the entire file will be loaded in memory, if possible. By using the preferred base address during the mapping process, the linker can generate absolute addresses when needed. However, occasionally the loader cannot load the binary at its preferred base address and must correct all of the encoded absolute pointers in the binary file. This is simply done by adding the difference between the actual base address and preferred base address to each pointer in the program as delineated by the base relocation table.

The relevant portions of the base relocation table for the previous example are shown in Figure 4.2 along with annotated assembly code. Note that the base relocation table does not contain an entry for offset `0x70`, indicating that the immediate value in the first example, for the value `0x00000008`, is not a pointer.

Figure 4.2 depicts two examples of indirect jumps using jump tables. All of the pointers in the code for this example are italicized and their addresses are all stored in the base relocation table. Note that the displacement in the second indirect jump (to the first jump table) does not actually point to the beginning of the jump table. The indexes stored in the register `ecx` can only be between -7 and 0. The simplest indexing schemes begin with zero and consist of a nonnegative range of contiguous indexes. We will call these *zero-based* jump tables. Other tables, such as the one in this example, are indexed on a contiguous range of indexes that do not begin with index zero and may not even include index zero. We will call these *nonzero-based* jump tables. In fact, several examples have been discovered where index zero is not included in the table, and only negative indexes are used.

During code discovery and translation when an indirect jump is encountered, all of its possible targets must be detected. Jump tables are often zero-based, which means that the displacement address in the indirect jump is the beginning of the jump table. However, this is not always the case, as illustrated n the `_memmove()` example. A heuristic approach has been developed that is further described in [4], but nonzero-based tables with no indexes near zero may be difficult to accurately identify.

```
                Base Relocation Table

             400000 virtual address base
             37000 virtual address

               Offset Type
               ...
               7A     HIGHLOW  ; 0043707A
               85     HIGHLOW  ; 00437085
               ...
```

```
control block 33:
  ...
  00437077 :  FF 24 95  A0 71 43 00    ; jmp   dword ptr [4*edx+  004371A0   h]
                        \__________/                              \________/
                        relocation at                           relocation at
                         offset 7Ah                               offset 7Ah

control block 39:
  00437080 :  F7 D9                    ; neg   ecx, ecx
  00437082 :  FF 24 8D  50 71 43 00    ; jmp   dword ptr [4*ecx+  00437150   h]
                        \__________/                              \________/
                        relocation at                           relocation at
                         offset 85h                               offset 85h

jump table 47:  named _section_text1_221492
  00437134 :  00437154                 ; (cb 48)
  ...
⇒00437150 :  00437197                 ; (cb 55)

jump table 34:  named _section_text1_221600
⇒004371A0 :  004371B0                 ; (cb 38)
  ...
  004371AC :  004371DC                 ; (cb 35)
```

Figure 4.2 Base relocation table and annotated assembly code.

Once the vicinity of the correct jump table is found, the exact bounds on the table must be determined. When a single jump table is nestled within the code, its bounds can be easily determined by the detection of code before and after the table. However, jump tables are sometimes placed one after another in a block of jump tables. This complicates detection because there is no indication where one table ends and another begins, causing them to appear as one large jump table. In this case, a conservative analysis is employed that assumes an indirect jump can jump to any entry in the combined table. This creates extra possible control flow transfers between the blocks. This is a safe assumption because it can only add extra registers to the live-in register sets of the extra target blocks. It also complicates the control flow graph, resulting in functionally correct code, but possibly at the cost of limited optimization opportunities.

Considering the potential inaccuracy of the heuristics, a code analysis approach might provide better results. By using code analysis routines, it may possible to determine the range of values potentially stored in the index register, `ecx` in the example. This would provide information on the possible offsets from the address encoded within the instruction, allowing the decoder to find all of the jump table entries.

4.3 Relocation Table

Figure 4.3 shows the same piece of code with a relocation table. The relocation table not only indicates where the pointers are but also what those pointers point to. Using this table, the correct jump tables can always be associated with their *indirect jump* instructions. This is accomplished by separating the offset into two pieces: the base of the beginning of the table and an offset to correct for nonzero-based indexed tables. The relocation table actually contains an index into the symbol table where the base address of the symbol can be obtained. In the example, the relocation for offset `0x00037085` specifies the symbol `_section_text1_221492` at address `0x00437134` with a table offset of `0x1C`, which sums to `0x00437150`, as encoded in the

Relocation Table

Offset	Type	Symbol Applied To	Symbol Index	Name
...				
0003707A	DIR32	00000000	2F2	_section_text1_221600
00037085	DIR32	0000001C	2F3	_section_text1_221492
...				

```
control block 33:
  ...
  00437077 :  FF 24 95  A0 71 43 00   ; jmp   dword ptr [4*edx+  004371A0   h]
                        \_________/                              \________/
                       relocation at                            relocation at
                       offset 3707Ah                            offset 3707Ah

control block 39:
  00437080 :  F7 D9                   ; neg   ecx, ecx
  00437082 :  FF 24 8D  50 71 43 00   ; jmp   dword ptr [4*ecx+  00437150   h]
                        \_________/                              \________/
                       relocation at                            relocation at
                       offset 37085h                            offset 37085h


jump table 47:  named _section_text1_221492
⇒00437134 :  00437154                ; (cb 48)
  ...
  00437150 :  00437197                ; (cb 55)
```

Figure 4.3 Relocation table and annotated assembly code.

instruction. Note that it still may be difficult to determine the bounds of the jump table, and that a range analysis of the index register may still be still be necessary to accurately determine the table bounds.

CHAPTER 5

MCODE INTERMEDIATE REPRESENTATION

The IMPACT compiler uses several different intermediate representations of an application during compilation. The back-end portions of the compiler employ a configurable representation called *Lcode* [12], [13]. The machine-independent portions of the back-end use an Lcode configuration that maps to a three-operand fictitious, RISC-like architecture. The machine-dependent portions use machine-specific variants of Lcode called *Mcode*. This binary reoptimization framework uses an Mcode configuration tailored to the x86 instruction set.

Some portions of the binary optimization framework require that functions be represented as collections of *basic blocks* (bb). Because of the loose definition of this term, it is necessary to define the term for use within this thesis. A basic block is a single entry-point, single exit-point block of instructions. The exit point may consist of any type of branch or simply a linear control flow instruction, but branches may only exist as the last instruction of a basic block. A *control block* (cb) is a basic block that allows multiple exit points. Most of the IMPACT compiler can operate on control blocks, rather than the simpler basic blocks, and therefore, in general, the term *control block* may be used interchangeably with the term *basic block*. Specifically, the profiling mechanism in Lbx86 phase 2 and the direct-to-binary converter in Lbx86 phase 3 require basic blocks.

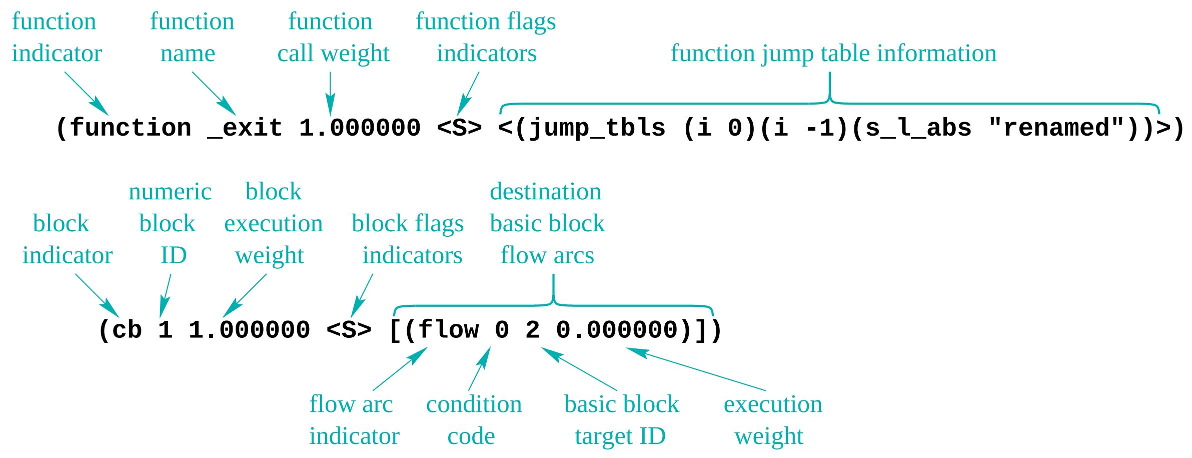

function indicator
function name
function call weight
function flags indicators
function jump table information
(function _exit 1.000000 <S> <(jump_tbls (i 0)(i -1)(s_l_abs "renamed"))>)
block indicator
numeric block ID
block execution weight
block flags indicators
destination basic block flow arcs
(cb 1 1.000000 <S> [(flow 0 2 0.000000)])
flow arc indicator
condition code
basic block target ID
execution weight

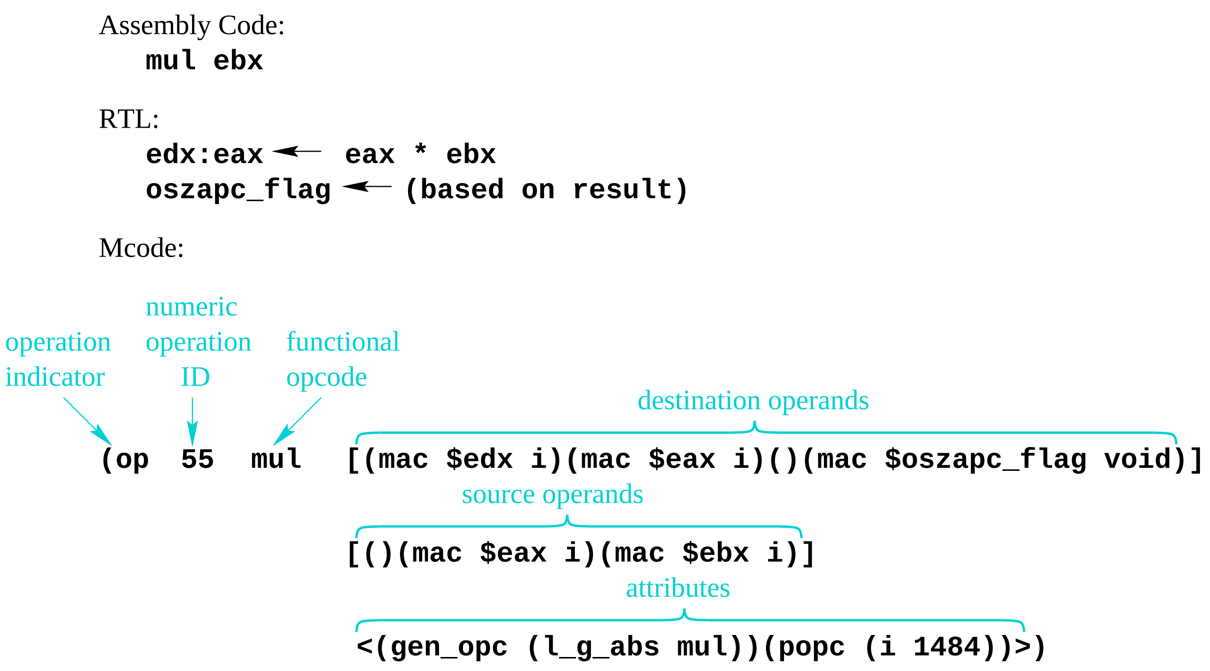
Figure 5.1 Mcode representation of functions and basic blocks.

5.1 Mcode Functions and Control Blocks

The Lcode and Mcode representations utilize data structures called *Lcode Function* for functions, and *Lcode Control Block* for control or basic blocks. Figure 5.1 depicts the printed Mcode for functions and blocks.

5.2 Mcode Instructions

The Lcode and Mcode representations utilize a data structure called *Lcode Operation* to represent instructions. In order to allow for automatic register analysis of the instructions, all registers written by the instruction must be represented in the data structure. Likewise, any registers that are read by the instruction must be represented. Note that the flags bit registers and all implicit registers are included in this requirement and must also be accurately represented. In an Lcode or Mcode configuration, the maximum number of source operands and maximum number of destination operands must be specified. Because some algorithms search all possible operand locations within the instruction, the maximum number of operands



Assembly Code:

```
mul ebx
```

RTL:

```
edx:eax ← eax * ebx
oszapc_flag ← (based on result)
```

Mcode:

```
(op 55 mul [(mac $edx i)(mac $eax i)()(mac $oszapc_flag void)]
           [()(mac $eax i)(mac $ebx i)]
           <(gen_opc (l_g_abs mul))(popc (i 1484))>)
```

Figure 5.2 Mcode representation of a *multiply* instruction.

should be set as low as possible. For the x86 binary reoptimization framework, the maximum number of source operands is seven and destination operands is four.

A brief description of the tailored Mcode will be presented next, while a more detailed description can be found in [9]. Figure 5.2 depicts the Mcode representation of a *multiply* instruction, which would appear in Intel's x86 assembly code as "`mul ebx`". Each instruction, also called an *operation*, begins with an "`op`" tag followed by an identification number that uniquely identifies this instruction within the function. The *functional opcode* classifies the instruction into general categories of operations. The next set of fields represents the *destination operands* of the instruction, followed by the *source operands*. Following the operands are the *attributes* fields which assist in better describing the instruction. The tailored Mcode for this framework requires two attributes: the actual Intel x86 mnemonic and a numeric value representing the

Table 5.1 Data types and their sizes.

dtype	abbrev.	size	dtype	abbrev.	size	dtype	abbrev.	size
integer	i	32 bits	short	sh	16 bits	char	c	8 bits
float	f	64 bits	void	void	none			

actual mnemonic and specific variant of that mnemonic. See Section 4.2 in [10] for additional information.

5.3 Mcode Operands

Each operand is tagged with an operand type, followed by a name and an optional data type. All registers are of operand type *macro*, printed as "`mac`". The data types for the registers are tagged according to their size, as in Table 5.1. All integer immediates in this framework are of operand type *integer*, printed as "`i`", where the data type is implied by the instruction and other operands.

To simplify dataflow and dependence analysis, each register must be independent of all other registers. Consider the `eax` register in Figure 5.3. The x86 architecture was initially designed as a 16-bit architecture, where `ax` could be accessed as the full 16-bit quantity, or as two 8-bit subsets `al` and `ah` that do not overlap. When register dependence analysis algorithms search for the next access to register `al`, accesses to `ax` must also be considered. This situation became more complicated when the original 16-bit architecture was extended to 32-bits, adding a superset register to `ax` called `eax`. This register aliasing problem is also significant at the microarchitecture level as these types of superset and subset dependences can cause pipeline stalls [14].

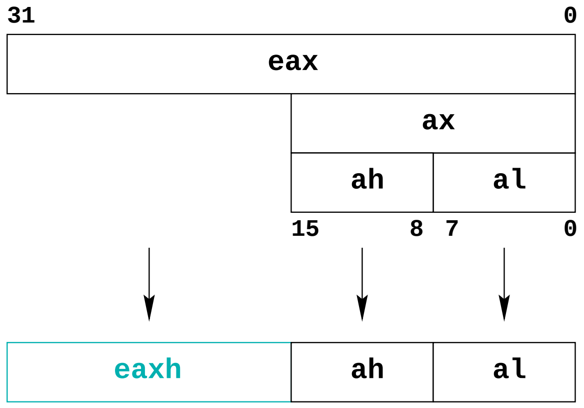


Figure 5.3 An example of overlapping subset registers.

The problem is somewhat simplified inside the reoptimizer by internally replacing each superset register with a set of nonoverlapping, yet completely representing, subset registers. The `ax` register, by architectural design, can be represented in this way by `al` and `ah`. The `eax` register, however, cannot be completely represented by existing subset registers because no high 16-bit register exists. Therefore, a fictitious `eaxh` register is added to solve this problem.

With the capability to break large superset registers into completely representing subset registers, even larger supersets can be created to simplify the Mcode. Figure 5.4 depicts the logical construction of the superset register `all_gp_32` which contains all eight of the 32-bit general purpose registers. Specifically, this is used to simplify the Mcode for the *push all 32-bit general purpose registers onto stack*, "`pusha`", instruction. A 16-bit version of this stack *push* also exists motivating the creation of the `all_gp_16` superset register. Likewise, *pop* instruction counterparts also exist. The shaded registers in the figure represent the subset registers of which all superset registers are assembled.

The flags register is actually modeled as a set of bit registers using the superset/subset model previously described. Detailed description of each of the flags can be found in [15].

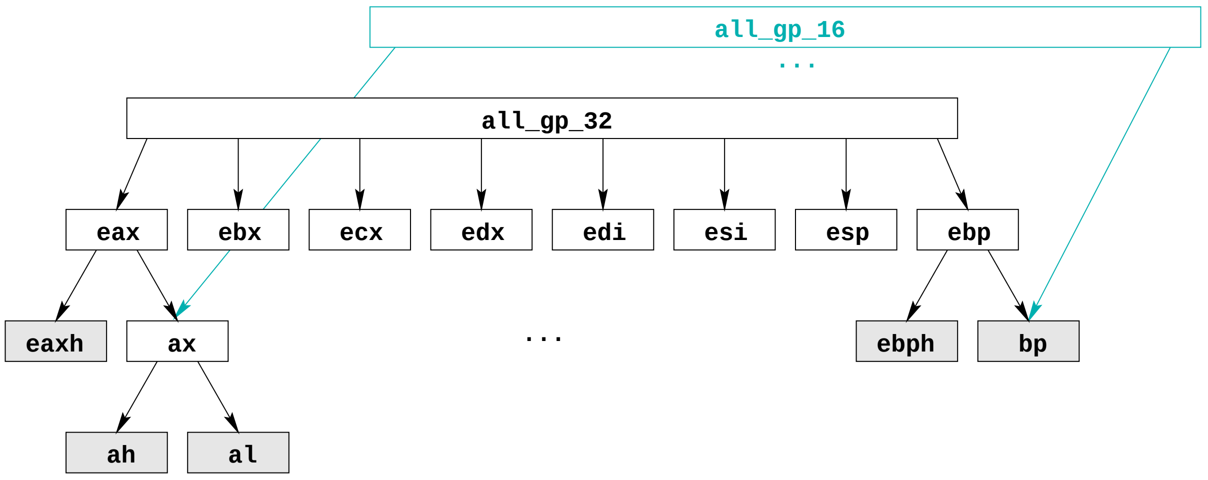


Figure 5.4 Breakdown of a superset register into its constituent subset registers.

Internal to the reoptimizer, each flag is its own bit-register, while at the Mcode level, a handful of superset registers exist, as shown in Table 5.2(a). Note that if the exact combination of flags is not represented by a single superset register, combinations of the individual flags and superset registers can be used to represent the exact combination. Or, a superset register minimally containing the necessary flags can be used; however, the extra flags will create false dependences possibly limiting optimization.

Table 5.2(b) and (c) list all of the registers implemented in the x86 binary reoptimizer's version of Mcode. Memory operands are represented in a unique manner. See Section 3.2.2 in [10] for a complete description. Because complex memory addressing is available in the x86 architecture, two different source registers may be needed to compute an address operand. Since the operand layout in Mcode is tailored to closely match x86 assembly, multiple registers for a particular operand do not map neatly into the Mcode. Therefore, a place-holder macro, "`addr`", is inserted in the proper operand location, and the memory address computation is

Table 5.2 Comprehensive list of available register operands.

oszapc_flag	oszpc_flag	oszap_flag	szapc_flag	osz_flag
oc_flag	zc_flag	os_flag	o_flag	s_flag
a_flag	z_flag	c_flag	d_flag	p_flag

(a) Flags operands.

name	dtype	name	dtype	name	dtype	name	dtype	name	dtype
eax	i	eaxh	sh	ax	sh	ah	c	al	c
ebx	i	ebxh	sh	bx	sh	bh	c	bl	c
ecx	i	ecxh	sh	cx	sh	ch	c	cl	c
edx	i	edxh	sh	dx	sh	dh	c	dl	c

name	dtype	name	dtype	name	dtype	name	dtype	name	dtype	name	dtype
esp	i	esph	sh	sp	sh	ebp	i	ebph	sh	bp	sh
esi	i	esih	sh	si	sh	edi	i	edih	sh	di	sh

name	dtype	name	dtype
all_gp_32	8 * i	all_gp_16	8 * sh

name	dtype	name	dtype	name	dtype
cs	sh	ds	sh	es	sh
fs	sh	gs	sh	ss	sh

(b) General register operands.

name	dtype	name	dtype	name	dtype	name	dtype
st0	f	st1	f	st2	f	st3	f
st4	f	st5	f	st6	f	st7	f
mm0	f	mm1	f	mm2	f	mm3	f
mm4	f	mm5	f	mm6	f	mm7	f
all_fst	8 * f	all_mm	8 * f	fpsw	void	fpcw	void

(c) Floating-point and MMX/3DNow! register operands.

represented in the last four operands of the source operand field: base register, index register, scale integer, and displacement address.

5.4 Exceptions to the Typical Instruction Format

Not all x86 instructions map neatly into a particular Mcode representation. For example, there are a handful of instructions that perform *memory-to-memory moves*, whereas the Mcode format only allows for representation of a single memory address. It is also possible that no functional opcode exists to accurately represent the operation being performed. The following examples present a few of the anomalous representations.

The lack of support for instructions that require two memory operands complicates their representation. Several of the *string instructions* require two memory operands, and their representation is described in Section 3.2.3 of [10]. There are two other *memory-to-memory move* instructions in the architecture, *push contents of a memory location*, "`push [mem]`", and *pop value to a memory location*, "`pop [mem]`", that must also be supported. Rather than extend the maximum number of source operands to accommodate these rare cases and in order to keep all *push* and *pop* instructions consistent, the memory access to the stack inherent to these instructions is implied. Likewise, the memory accesses in the *string instructions* are also implied. These instructions are then handled as special cases inside the dataflow and dependence analysis engines. However, since the *push memory* and *pop memory* instructions are infrequently executed, creating special cases specific to these situations would also be a viable alternative.

A typical *push* instruction is depicted in Figure 5.5. As can be seen in the register transfer language (RTL) description, the stack pointer is both read and modified by this instruction. This situation is correctly represented in the Mcode by the presence of the `esp` operand in both the source and destination fields. Furthermore, the pre-increment of the stack pointer is reflected in the functional opcode, *pre-increment store*, "`st_pre_i`". While the actual address macro and address operands specifying the destination stack memory location are omitted, the correct memory attributes are present. Figure 5.6 depicts a *push memory* instruction where the address fields of the instruction are used for the memory address of the pushed operand.

Assembly Code:
```
push eax
```

RTL:
```
esp ← esp - 4
[esp] ← eax
```

Mcode:
```
(op 324 st_pre_i [(mac $esp i)]
                 [(mac $esp i)(mac $eax i)]
                 <(mem_size (i 4))(mem_write)
                   (gen_opc (l_g_abs push))(popc (i 1632))>)
```

Figure 5.5 Mcode representation of a *push register* instruction.

Assembly Code:
```
push DWORD ptr [eax]
```

RTL:
```
esp ← esp - 4
[esp] ← [eax]
```

Mcode:
```
(op 325 st_pre_i [(mac $esp i)]
                 [(mac $esp i)(mac $addr void)()(mac $eax i)]
                 <(mem_size (i 4))(mem_read_write)
                   (gen_opc (l_g_abs push))(popc (i 1634))>)
```

Figure 5.6 Mcode representation of a *push memory* instruction.

Another complication arises when no functional opcode accurately describes the operation being performed. Consider Figure 5.7 which depicts an instruction that adjusts two unpacked binary-coded-decimal (BCD) digits so that a division operation performed on the result will yield a correct unpacked BCD value. Clearly, no RISC-like instruction set, such as Mcode, will have an opcode for this type of operation. Therefore, an unused Mcode functional opcode "`min`" is used in conjunction with the *L_OPER_PROCESSOR_SPECIFIC* flag, denoted "`<?>`", for such instructions.

Assembly Code:
```
   aad 10d
```

RTL:
```
   al ⟵ (al + (ah * 0x0A)) AND 0xFF
   ah ⟵ 0
   oszapc_flag ⟵ (based on result)
```

Mcode:
```
   (op 55  min <?>  [(mac $ax sh)()()(mac $oszapc_flag void)]
                    [(mac $ax sh)(i 10)]
                    <(gen_opc (l_g_abs aad))(popc (i 96))>)
```

Figure 5.7 Mcode representation of an *ASCII adjust* instruction.

Figure 5.8 depicts an example of a floating-point instruction.[1] Note that the floating-point stack registers are used. Currently in the framework, floating-point instructions are not optimized. There are no critical technical challenges to such optimization other than precision problems inherent to expression reformulation of floating-point computations and dataflow analysis problems associated with the floating-point operand stack. For example, a reference to `st0`, the top of the floating-point operand stack, and a subsequent reference to `st0` may not be the same physical register if a stack *push* occurred in between. Stack models are not automatically handled by our dataflow analysis algorithms. Also, no exact functional opcode is available for this example instruction. While the *floating-point divide*, "`DIV_F`", is essentially correct, the fact that the operands are reversed for the computation is only represented in the processor-specific opcode. The operands themselves cannot be reversed in the Mcode because they represent assembly code operand ordering, not computational operand ordering. Thus, the *L_OPER_PROCESSOR_SPECIFIC* flag is used, and the actual opcode is stored in the processor-specific opcode field, as it is in all instructions.

[1]Some older versions of the Pentium Instruction Set Reference manuals have incorrect descriptions of the "`fdiv`" and "`fdivr`" opcodes.

Assembly Code:

```
   fdivr st(0), st(7)
```

RTL:

```
   st(0) ←  st(7) / st(0)
```

Mcode:

```
   (op 470 div_f <?> [(mac $st0 f)]
                     [(mac $st0 f)(mac $st7 f)]
                     <(gen_opc (l_g_abs fdivr))(popc (i 2928))>)
```

Figure 5.8 Mcode representation of a *floating-point reverse divide* instruction.

CHAPTER 6

CONVERSION OF ASSEMBLY TO MCODE

The x86toM application, the utility that performs the conversion from x86 binary into Mcode, first decodes the binary into an internal assembly-like representation (which will be referred to as the assembly representation) using control flow analysis techniques, as discussed in Section 3.3. During the decoding process, the instructions are grouped into basic blocks, and basic blocks are grouped into functions. Once the process of decoding and grouping is complete, the process of conversion to Mcode begins. Details on the structures and techniques used for decoding are outlined in Chapters 3 and 4. More details are discussed in C. N. George's thesis [4].

6.1 Conversion of Functions and Basic Blocks

The assembly representation to Mcode conversion algorithm for functions and blocks is summarized in Figure 6.1. The decoder provides the conversion algorithm with a linked list of functions, each of which contains a linked list of basic blocks and a linked list of entry-point addresses. Furthermore, a memory map has been constructed by the decoder through which the specific function and block can be determined for any address. If a symbol table is present in the binary, then it can also be accessed with an address to find a symbol name, and vice versa. Because instruction decode and conversion is one of the most expensive portions of the process in terms of both running time and memory usage, a primary goal of the conversion algorithm is to completely process all information in a decoded assembly instruction in a single pass.

In order to convert the instructions in a single pass, certain information must be available prior to the conversion of the first instruction. Consider the conversion of a *call* instruction. One of the primary conversion tasks is to replace the *call*'s target address with a target function name, or label. However, if the target entry point to the function does not have a name, there will be no target label available to add to the instruction. Likewise, if the target has the same name as another function, each will need to be given a unique name so that the label in the *call* instruction targets the correct version.[1]

Thus, the conversion process begins by allocating the Mcode function data structure and by verifying entry-point names for each function. Duplicate names are distinguished by appending a version number to the function name. Normally, functions have a single entry point at the top of the function. In this case, the Mcode function name field in the internal structure is assigned that name, implying entry at the top of the function. In other cases, the function has an entry point that is not at the top of the function, which includes functions with multiple entry points. In these cases, a single function name which implies entry at the top is not accurate. Therefore, these functions are specially tagged to represent the different entry points. Specifically, the Mcode function is assigned a name (usually one of the entry point names) appended with "_me" to indicate the presence of multiple named entry points. Then, each entry point in the entry-point address list is assigned its name. Later, each basic block that is an entry point is tagged with its entry-point name listed in an attribute.

[1]Duplicate function names within a single binary actually occurs rather frequently. For example, the function `_strncnt()` is statically defined within three C Library source files (`a_cmp.c` and `a_map.c`, and an inline version in `tchar.h`). In this situation, the code for the different versions is identical, but in general, this cannot be guaranteed.

```
Global List of FunctionInfoNodes, Memory Map, and Symbol Table

Convert functions to Mcode (){
    For each function in list{
        Create Mcode function shell
        If non-top-of-function entry point exists{
            Name Mcode function with one entry-point name appended with "_me"
            Ensure each entry point has a unique name by assigning alternate name
                to absent or duplicate names
        }
        Else:
            Ensure top entry point has unique name by assigning alternate names
    }
    For each function in list{
        Sort the list of basic blocks within this function by address
        Create Mcode basic block shells
        For each basic block{
            For each instruction{
                Create Mcode instruction shell
                Create explicit operands
                Assign Mcode functional opcode
                Create implicit operands
                Annotate instruction for implicit memory reference
                Create flags operands
                If branch or call instruction{
                    Compute target address
                    If address is relocatable:
                        create symbol operand
                    If address is basic block:
                        create basic block operand and create taken-branch flow arcs
                }
                Encode processor-specific opcode
            }
        }
        Restructure basic blocks for single exit point
        Create jump table and fall-through flow arcs
        Restructure basic blocks for single entry point
    }
}
```

Figure 6.1 Pseudocode algorithm for converting functions, basic blocks, and instructions into Mcode.

The next step is to create the basic blocks within the functions. The list of basic blocks comes to the converter in a depth-first-search order. This list must be sorted by address so that fall-through paths actually fall through to their proper blocks without the addition of unconditional jumps.

In this implementation of the framework, all of the basic block shells are created within a function before any of the instructions are inserted. Consider the conversion of *jump* and *branch* instructions to Mcode. The target addresses of these instructions are determined from the information in the assembly representation and from the current program counter value. When creating the instruction, the actual target basic block must already exist because, internally, the target operand is a pointer to the target basic block computed from the target address. Therefore, to simplify the conversion process, all of the Mcode basic blocks within the function are created before any instructions are converted. Since pointers to the target basic blocks have already been obtained, the flow arcs associated with these instructions are also created at this time. The fall-through arcs and any jump table arcs are created at a later stage. Their creation does not require information from the decoded instruction, but rather can be formed by a quick analysis of the Mcode.

After all the basic block shells are created for a function, the instructions are created block-by-block. This is detailed further in Section 6.2. Once all of the instructions have been added to a function, a single exit point is created to facilitate simpler data-flow analysis. This transformation consists of creating a single *return* instruction in a new block called the *epilogue*. All original *return* instructions are converted into *unconditional jump* instructions to the new epilogue. This restructuring will be reversed before the final binary is written.

Next, all of the fall-through flow arcs are created between the basic blocks. Likewise, all jump table flow arcs are created at this time. As mentioned in Section 4.2, jump tables are often allocated next to each other, so it is difficult to accurately determine the boundary of one table from another. In other words, several jump tables may appear as one contiguous region of basic block pointers. Likewise, multiple *indirect jump* instructions may access the same region. Because the boundaries are not definitive, each block containing a indirect jump instruction using that region will have a flow arc to each block specified by the pointers in that region. This is a conservative analysis but is correct as execution will never actually traverse the fictitious flow arcs and will only cause extra registers to be live-in to the target blocks.

At this point, the appropriate transformations are made to the function to correctly represent entry points that are not at the top of the function. See Section 3.3.2 in [10] for details on the transformation. A *prologue* basic block is created at the top of the function that contains an *indirect jump* to each of the entry points. Actual program flow will never execute this block or the *jump*, but data-flow analysis must operate on a function with a single-function entry point. Then, each entry-point basic block is annotated with an entry-point attribute.

6.2 Conversion of Instructions

Each instruction is generally converted in the following manner, although the order of a few of the steps may be altered for a few special cases. First, the Mcode instruction data structure, or shell, is created. Then each of the explicit operands represented in the assembly is converted, as will be discussed in the next section. Analyzing the explicit operands also adds memory usage attributes for the explicit operands to the Mcode when appropriate. The functional opcode, as defined in Section 5.2, is assigned next based on the assembly opcode and sometimes the explicit operands. The operands are sometimes necessary for functional opcode determination because of instructions such as a "`mov`" which may be a *load*, a *store*, or a *register-to-register move*, all of which have different functional opcodes.

Next, any implicit operands (and implicit memory reference attributes) are added, such as references to the stack pointer (and stack memory) in stack operations, as discussed in Section 5.4. If the instruction reads or writes to the flags registers, the correct flags operand is then chosen and inserted into the Mcode. If the instruction is a *direct branch* or a *call*, the target address is computed from the current instruction's program counter and displacement. That address is located in the memory map from which a pointer to the appropriate block can be obtained. Finally, the instruction is annotated with its processor-specific opcode information.

6.3 Conversion of Operands

Up to three explicit operands can be represented in the assembly representation, usually one destination and two sources. Each is processed by the algorithm in Figure 6.2 in the following manner: if the operand is a register, then the corresponding Mcode register is added to the instruction; if it is an immediate, its corresponding label is added, if one exists; if it is a near address, its corresponding label is added; or if it is a memory reference, the address operand place-holder is inserted and the memory operands are then added. Note that the "`enter`" instruction, as shown in Figure 6.3, contains two immediates. The instruction creates a new stack frame of the size of the first immediate, and pushes nested frame pointers based on the value stored in the second immediate. The second immediate is stored in the displacement field in the internal assembly representation to avoid extending the assembly representation.

```
Create explicit operands (internal assembly instruction, Mcode instruction){
    For each of the three explicit operands in the assembly{
        If general, segment, or floating-point register:
            Add register to Mcode
        If immediate or near address:
            If enter instruction and processing second immediate:
                Process immediate using displacement value
            Else:
                Process immediate
        If memory address{
            Add address macro to Mcode
            Create memory sources
        }
        If flag register:
            Delay insertion until later
        If far address:
            Error: Shouldn't see far calls in Win32 code
    }
}
```

Figure 6.2 Pseudocode algorithm for converting explicit operands into Mcode.

Assembly Code:

```
fdivr st(0), st(7)
```

RTL:

$$\texttt{st(0)} \leftarrow \texttt{st(7) / st(0)}$$

Mcode:

```
(op 470 div_f <?> [(mac $st0 f)]
                  [(mac $st0 f)(mac $st7 f)]
                  <(gen_opc (l_g_abs fdivr))(popc (i 2928))>)
```

Figure 6.3 Mcode representation of an *enter* instruction.

When processing a memory reference, the base register, index register, and scale integer are added to the Mcode. The displacement integer is often a pointer to a memory address table, so that integer must be converted to its corresponding label. Likewise, if an immediate operand is a pointer, then its corresponding label must be found. Note that only 32-bit values can be pointers, and thus are the only values examined for label conversion. The *Process Possible Relocatable* routine checks to see if the value's location is in the base relocation table, as show in Figure 6.4. If so, the value is expected to be a pointer, and the label creation routine is called. If the value is not a relocatable, or no label was found to match the value, then the value is just a constant and is inserted into the Mcode as such. Otherwise, the correct label is inserted. Note that the labels follow the standard formats, as shown in Figure 3.3.

The two algorithms in Figure 6.5 show how addresses are converted into labels. The first algorithm expects the value to point to a jump table, which can be located in either the data or text section. For a jump table in the data section, the address is converted into nontext section label, as shown in Figure 3.3. This address is simply converted by subtracting the base address of the section from the original address and adding that result as the offset from the section label. For jump tables that appear within the text section, the appropriate block must be found. The displacement value must point into a jump table or point immediately before or after a jump table, mirroring the discovery heuristics in the flow decoder. A jump table should be found, again because the same heuristic in the decoder found one, and a label comprising the jump table block name and offset from the beginning is produced.

The second algorithm is used when a jump table is not expected. If the value points into a section other than the text section, a label comprised of the section name plus offset is produced. When the value points into the text section, it must point to the beginning of a block, whether that block is a table or instructions. If the block is an entry point to the function, a function name label will be produced; if it is a basic block, then a basic block label will be produced; otherwise, a generic block label will be produced. Occasionally, the displacement value does not

```
Process Immediate (internal assembly instruction){
    If 32-bit immediate:
        Process possible relocatable as not a jump table
    Else:
        Add immediate constant to Mcode
}

Process Displacement (internal assembly instruction){
    If 32-bit displacement:
        Process possible relocatable as a possible jump table
    Else:
        Add immediate constant to Mcode
}

Process Possible Relocatable (internal assembly instruction,
        address of immediate or displacement, is it a possible jump table?){
    If the value at the address is a relocatable:
        If possible jump table:
            Find possible jump table label
        Else:
            Find label
    If not a relocatable or no label was found:
        Add constant to Mcode
    Else:
        Add label to Mcode
}
```

Figure 6.4 Pseudocode algorithm for converting possible relocatable operands into Mcode.

point into a section at all, such as when the displacement value of a nonzero-based jump table points far from the actual table. This makes identifying the jump table a difficult problem. A simple heuristic is employed which assumes that the range of indexes is nonnegative, and that the jump table itself exists in the next section in the address space.

```
Find possible jump table label (address of immediate or displacement){
    Obtain the value of the immediate or displacement
    If value points into section other than the text section:
        Return label for beginning of section + offset
    Else if value points into text section{
        If value points into a predetermined jump table:
            Return label for beginning of jump table + offset
        Else{
            Search forward and backward one DWORD (same as decoder heuristic)
            If search found a jump table:
                Return label for beginning of jump table + offset
            Else:
                Error: No jump table found.
        }
    }
}

Find label (address of immediate or displacement){
    Obtain the value of the immediate or displacement
    If value points into section other than the text section:
        Return label for beginning of section + offset
    Else if value points into text section{
        If value points to a function entry point:
            Return function name label
        Else if value points to the beginning of a basic block:
            Return basic block label
        Else if value points to the beginning of a jump table or data block:
            Return block label
        Else:
            Error: Must point to the beginning of a block
    }
    Else if value does not point into any section
        Warn and return label for beginning of next sequential section + offset
}
```

Figure 6.5 Pseudocode algorithm for converting label operands into Mcode.

CHAPTER 7

BINARY PROFILING

Traditional optimization of an application consists primarily of compacting the code by removing unnecessary or redundant instructions. This reduces program execution time by reducing the number of instructions required to execute the program, and often by reducing cache misses due to smaller code size. Recent efforts have lead to the development of optimizations that expose fine-grained parallelism, or *instruction-level parallelism* (ILP), allowing for multiple, independent instructions from the execution stream to be executed in parallel. These techniques have been shown to lead to significant performance improvements in scalar code [16]. Other algorithms can optimize code for particular values or for particular control flow paths. However, some of these optimizations actually increase the number of instructions executed. At the same time, the dependence height may be reduced, or second-order factors, such as instruction cache behavior, may be improved. Such optimizations may, however, degrade performance if applied in the wrong situations.

Consider superblock formation and optimization with tail duplication. Superblock formation selects a commonly executed path through the code creating straight-line code. Side entrances into the superblock are not allowed because they create extra dependences into the block that may hinder optimization of the selected frequent path. Furthermore, entrances via side entrances may have different branch behavior patterns than along the main path. If the same code were to be used for both entrances, optimizing for one pattern might adversely affect the other patterns. Thus, the side entrances are removed from the block and redirected to duplicates of the original code that follow the side entrances. Aggressive ILP optimization can be effectively applied to superblocks.

In order to make wise optimization decisions, the important functions, blocks, or paths in the code need to be identified. A *profile*, or sample of program execution behavior, is gathered and used to make such determinations. This profile is assumed to be representative of normal program usage, making it a good indicator of those important functions, blocks, or paths. With the addition of profile information, compilers are able to concentrate optimization efforts on the important regions of the code. Static heuristics have also been developed to estimate which portions of the code are important [17]. The profiling mechanism described in this thesis is designed to be simple and is not necessarily the most efficient profiling mechanism available.

7.1 Profiling Probe Insertion

The current IMPACT infrastructure utilizes *edge-weight* profiles, also called *branch profiles*, to make optimization decisions. With this type of profiling, the execution weights of each possible control path out of a block are gathered. (The terminology refers to a graph representation of control flow where blocks are nodes and control flow transfers are edges.) From the edge weights, the block and instruction weights can also be inferred using Kirchhoff's current law. This law states that the sum of all of the in-edge weights to a block must be equal to the sum of all of the out-edge weights. That sum is also the weight of the block. Since the profiling portion of the binary optimization framework uses basic blocks, that sum is also the weight of all of the instructions within the block. There are three primary types of branches used in x86 32-bit Windows applications: *conditional direct branch*, *unconditional indirect jump* (usually jump-through-memory rather than jump-through-register), and *unconditional direct jump*.

Assembly Code:

```
pushf
inc DWORD ptr [IMPACT_PROFILE_TABLE+36]
popf
```

Mcode:

```
(op 113 st_pre_i [(mac $esp i)]
  [(mac $esp i)(mac $d_flag void)(mac $oszapc_flag void)]
  <(mem_size (i 4))(mem_write)(gen_opc (l_g_abs(l_g_abs pushf))(popc (i 1664))>)
(op 112 add [()()()(mac $oszap_flag void)]
  [(mac $addr void)(i 1)()()()()(l_g_abs IMPACT_PROFILE_TABLE+36)]
  <(mem_size (i 4))(mem_read_write)(gen_opc (l_g_abs inc))(popc (i 660))>)
(op 111 st_post_i [()(mac $esp i)(mac $d_flag void)(mac $oszapc_flag void)]
  [()(mac $esp i)]
  <(mem_size (i 4))(mem_read)(gen_opc (l_g_abs popf))(popc (i 1616))>)
```

Figure 7.1 Mcode probes using the *increment memory* instruction with surrounding save and restore of the flags.

The profile weights are collected using a set of counters that are updated by probes inserted into the code. These counters will count events such as block execution and fall-through branch execution. All of the counters are added as a contiguous block to the beginning of the initialized data section and are initialized to zero.

The profile counters are updated via inserted probes implemented by the *increment memory* instruction, as shown in Figure 7.1. While either the *increment memory* or the *add to memory* instruction could be used to update the counters, *increment memory* modifies one less status flag than the *add*. (The *add* modifies the overflow, sign, zero, auxiliary, carry, and parity flags, while the *increment* does not modify the carry flag.) These flag modifications are side-effects of the counter updates and could potentially destroy needed information stored in those flags. For example, if such a probe were inserted between a *compare* instruction and a *conditional branch* instruction, the flags set by the *compare* and used by the *conditional branch* could be changed, possibly altering the direction of the branch.

The side-effects of the probes are handled in one of two ways. First, when dataflow analysis is performed, information specifying which registers (including the flags registers) contain useful values is generated. This portion of dataflow analysis is called *live variable analysis*. By using this analysis, the probe insertion routines can determine whether or not the flags are in fact live where the probe is to be inserted. If the flags are live, then the flags registers must be saved before the counter increment is performed and restored afterward. Otherwise, the probe destroys no useful data and can be used without saving and restoring the flags. If dataflow analysis is not performed, the flags registers are conservatively assumed to be live and must be saved and restored.

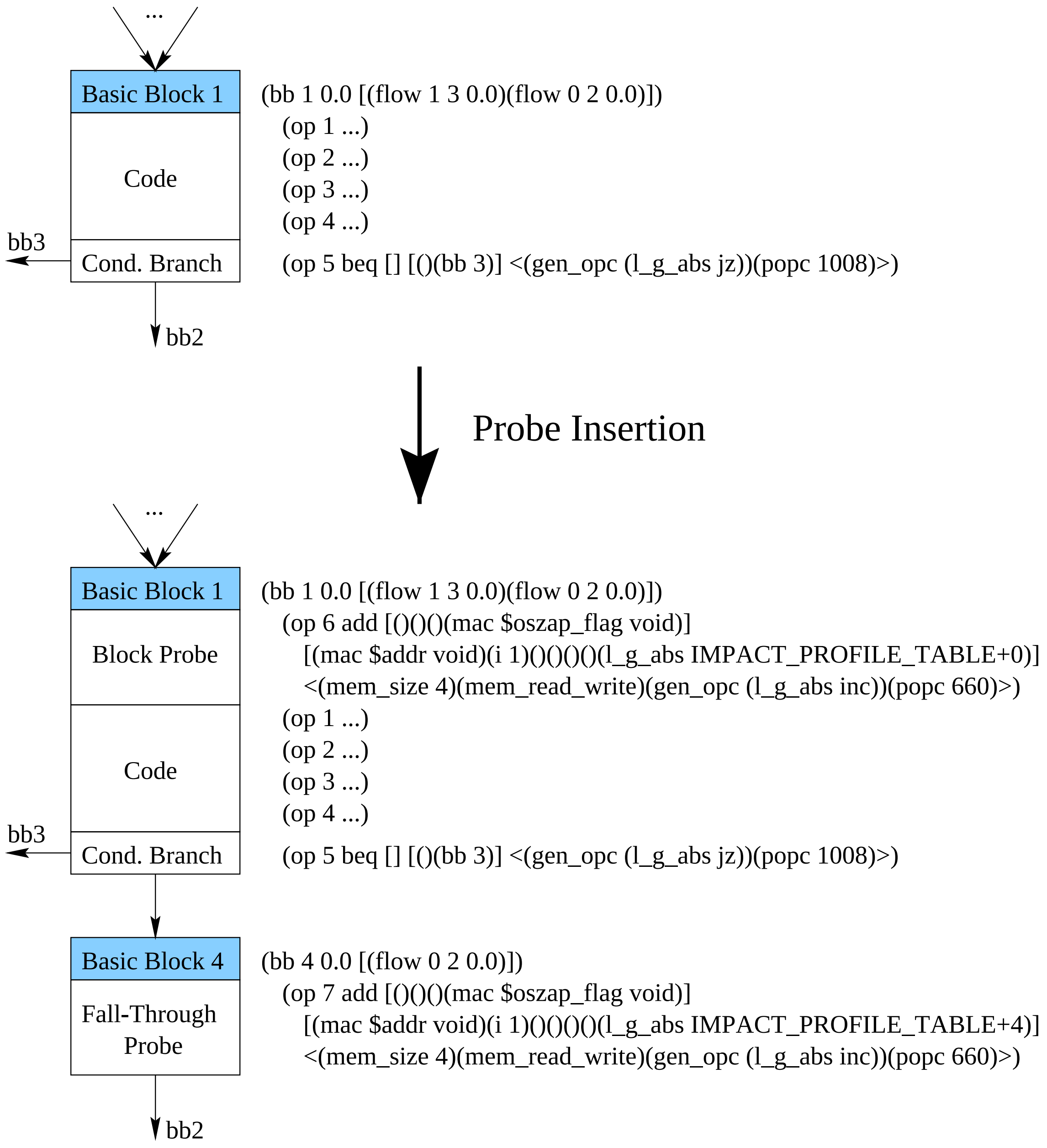
7.1.1 Direct branches and jumps

In order to collect the edge-weight profiles for a *conditional branch*, for example, probes that update counters could be inserted along the taken and fall-through paths. The basic block that contains that *conditional branch* would have the weight of the sum of the two edges. Or, if the basic block weight is known, the taken edge weight could be computed as the difference between the block weight and the fall-through edge weight. This latter scheme was chosen for ease of implementation and is shown in Figure 7.2.

For *unconditional direct jumps*, there is only one edge out of the block. Because there is only a single exit point, the taken jump weight is the same as the block weight, and only the block probe is necessary.

7.1.2 Indirect jumps

There are two primary variations of *indirect jumps* used in x86 programs. The first is a single-target *jump* through a single memory location. Simple profiling is performed on this



```
(bb 1 0.0 [(flow 1 3 0.0)(flow 0 2 0.0)])
  (op 1 ...)
  (op 2 ...)
  (op 3 ...)
  (op 4 ...)
  (op 5 beq [] [()(bb 3)] <(gen_opc (l_g_abs jz))(popc 1008)>)
```

```
(bb 1 0.0 [(flow 1 3 0.0)(flow 0 2 0.0)])
  (op 6 add [()()()(mac $oszap_flag void)]
    [(mac $addr void)(i 1)()()()()(l_g_abs IMPACT_PROFILE_TABLE+0)]
    <(mem_size 4)(mem_read_write)(gen_opc (l_g_abs inc))(popc 660)>)
  (op 1 ...)
  (op 2 ...)
  (op 3 ...)
  (op 4 ...)
  (op 5 beq [] [()(bb 3)] <(gen_opc (l_g_abs jz))(popc 1008)>)

(bb 4 0.0 [(flow 0 2 0.0)])
  (op 7 add [()()()(mac $oszap_flag void)]
    [(mac $addr void)(i 1)()()()()(l_g_abs IMPACT_PROFILE_TABLE+4)]
    <(mem_size 4)(mem_read_write)(gen_opc (l_g_abs inc))(popc 660)>)
```

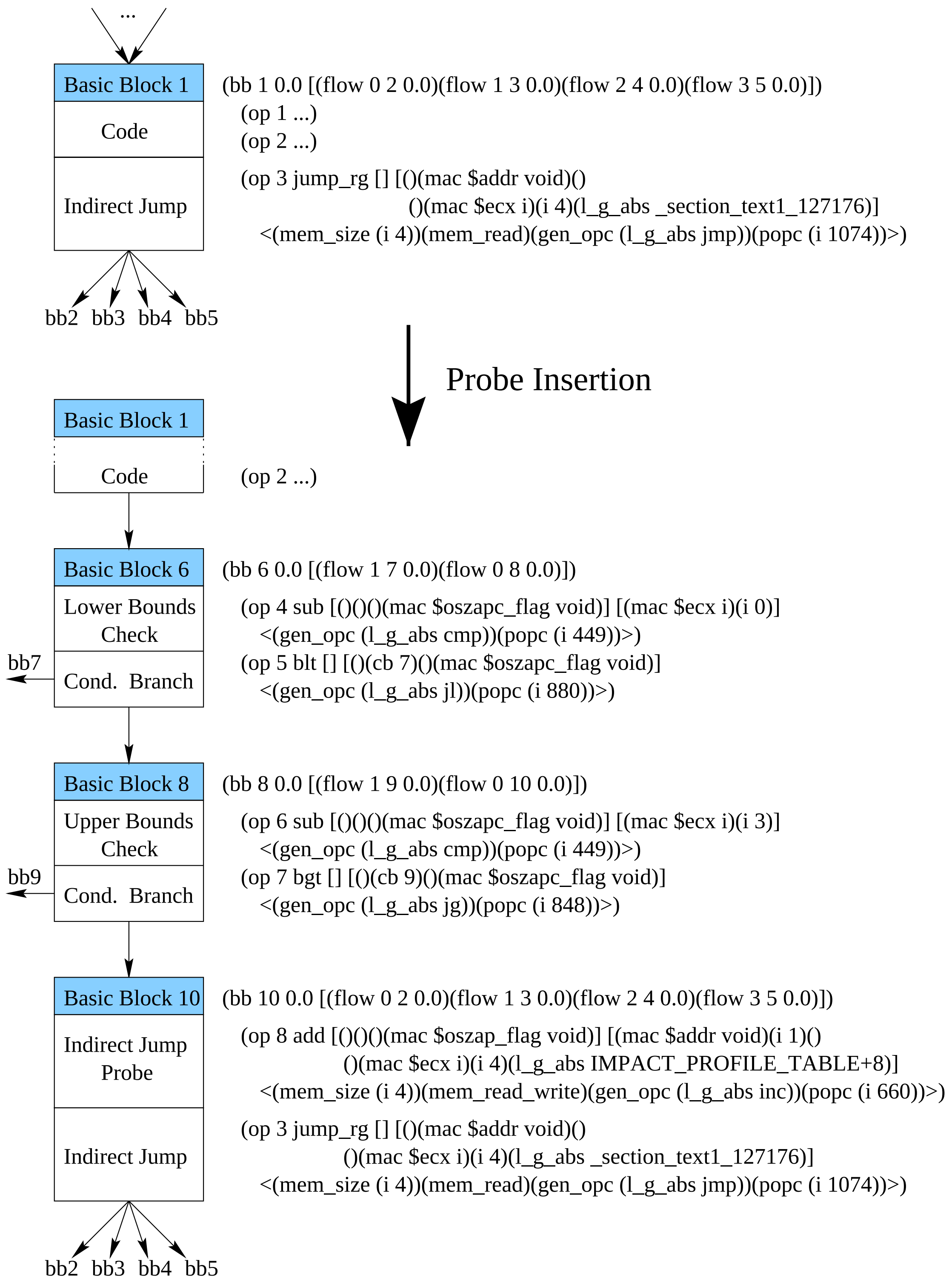
Figure 7.2 Mcode probe insertion for a basic block ending in a *conditional branch*.

jump, just as in *unconditional direct jumps*. More complicated profiling may be performed which tabulates the various targets reached, but such profiling is not implemented in this framework. Most commonly, the instruction contains a displacement address at which that target address is located. These types of *jumps* are particularly difficult for the framework to analyze, in general, because the complete list of possible targets is unknown.

The second and most common variation is an *indirect jump* through a jump table. This type of *jump* is characterized by a displacement containing an offset to or near the jump table and a register containing the index into the table. The memory location that contains the target address is computed through the x86 complex addressing mode that multiplies the index register by four and then adds in the displacement. Note that the displacement address to the jump table does not necessarily point to the beginning of the jump table, as discussed in Chapter 4. Figure 7.3 shows an example of probe insertion for this type of *indirect jump*. By using that same index register as the actual jump, an index into a table of profile counters can be computed. Likewise, the offset of the displacement address in the *jump* instruction from the actual jump table is used to calculate the correct displacement in the probe instruction from the counter table.

Recall the definitions of *zero-based* and *nonzero-based* jump tables as defined in Chapter 4. Consider zero-based jump tables where the first index is zero and the displacement of the *indirect jump* contains the address of the beginning of the table. When creating the probe instruction, the displacement will be simply the beginning of the allocated table of counters. For the nonzero-based jump tables, the address of the first entry is the lowest valid index multiplied by four plus the displacement. Therefore, the displacement in the probe instruction is computed by subtracting the lowest index multiplied by four from the beginning of the counter table.

Because heuristics in the decoder are employed to locate the targets of an *indirect jump*, some targets may not be correctly identified. The heuristics produce an expected range for the value of the index register. As a debugging tool, extra check instructions are inserted before the



```
(bb 1 0.0 [(flow 0 2 0.0)(flow 1 3 0.0)(flow 2 4 0.0)(flow 3 5 0.0)])
  (op 1 ...)
  (op 2 ...)
  (op 3 jump_rg [] [()(mac $addr void)()
                              ()(mac $ecx i)(i 4)(l_g_abs _section_text1_127176)]
    <(mem_size (i 4))(mem_read)(gen_opc (l_g_abs jmp))(popc (i 1074))>)

  (op 2 ...)

(bb 6 0.0 [(flow 1 7 0.0)(flow 0 8 0.0)])
  (op 4 sub [()()()(mac $oszapc_flag void)] [(mac $ecx i)(i 0)]
    <(gen_opc (l_g_abs cmp))(popc (i 449))>)
  (op 5 blt [] [()(cb 7)()(mac $oszapc_flag void)]
    <(gen_opc (l_g_abs jl))(popc (i 880))>)

(bb 8 0.0 [(flow 1 9 0.0)(flow 0 10 0.0)])
  (op 6 sub [()()()(mac $oszapc_flag void)] [(mac $ecx i)(i 3)]
    <(gen_opc (l_g_abs cmp))(popc (i 449))>)
  (op 7 bgt [] [()(cb 9)()(mac $oszapc_flag void)]
    <(gen_opc (l_g_abs jg))(popc (i 848))>)

(bb 10 0.0 [(flow 0 2 0.0)(flow 1 3 0.0)(flow 2 4 0.0)(flow 3 5 0.0)])
  (op 8 add [()()()(mac $oszap_flag void)] [(mac $addr void)(i 1)()
                 ()(mac $ecx i)(i 4)(l_g_abs IMPACT_PROFILE_TABLE+8)]
    <(mem_size (i 4))(mem_read_write)(gen_opc (l_g_abs inc))(popc (i 660))>)
  (op 3 jump_rg [] [()(mac $addr void)()
                 ()(mac $ecx i)(i 4)(l_g_abs _section_text1_127176)]
    <(mem_size (i 4))(mem_read)(gen_opc (l_g_abs jmp))(popc (i 1074))>)
```

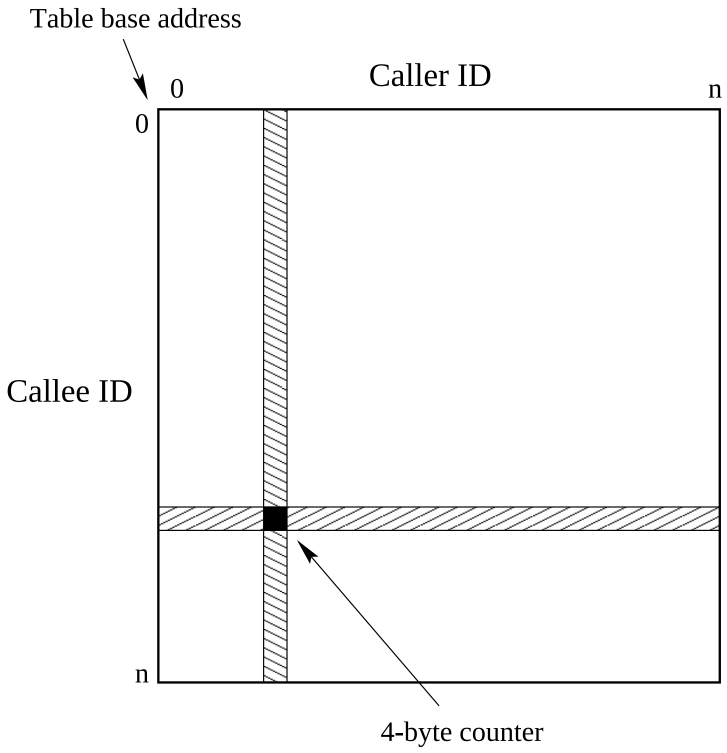
Figure 7.3 Mcode probe insertion for a basic block ending in an *indirect jump*.

indirect jump, which verify that the index register is within the expected bounds. In the current implementation, when a bounds check fails, control flow transfers to a basic block that executes a user breakpoint interrupt. Note that if the targets are not correctly identified by the decoder, the *indirect jump* through an unexpected table entry will eventually cause program failure. This can be expected because the table itself was most likely moved, and the surrounding bytes (unexpected entries) do not contain the correct target addresses. Even if the surrounding bytes contain valid addresses to code (another jump table perhaps), mismatches in register contents would likely cause an invalid data memory reference in the unexpected target code. So, by adding a bounds check, an interrupt is triggered immediately before an invalid table entry would be accessed, pin-pointing the table and invalid index.

7.1.3 Call graph collection

Some optimizations require the use of a *call graph*, which is graph that depicts the calling relationships between functions. An accurate call graph can be generated from an edge-weight profile if all of the *calls* are *direct calls*. However, additional profiling techniques must be employed to gather information on *indirect calls*, those through a register or a memory location.

In order to collect this type of profiling information, a counter must be maintained for each executed caller-callee function combination. This framework takes a conservative approach and maintains a counter for each possible caller-callee function combination using a simple two-dimensional array of counters. The row of the table is selected by the callee function identifier and the column is selected by the caller function identifier. Figure 7.4 depicts this counter table used for collecting the call graph profile. A complete two-dimensional table implementation was chosen for several reasons. First, from standard binary code, it is difficult to



Counter location = Table base address + (Caller ID * 4) + (Callee ID * 4 * n)

Figure 7.4 Call graph profile mechanism.

determine the subset of functions that are reachable from a particular caller function. Thus, the profile gathering mechanism must be able to accurately record weights for every possible caller-callee combination. Second, support C library functions that are required for other profiling mechanisms, such as `_malloc()`, may not be present in the original binary or may not be recognizable due to the lack of symbol information within the binary. Therefore, implementation of other collection approaches, such as a counter hash table, would require the addition in assembly or Mcode of such routines. Furthermore, the added routines must not interfere with the original versions, if they exist. Last, the computation of the exact counter location is very quick in this scheme compared to other approaches. The drawback, of course, is the rather large, sparse table.

The equation used to compute the counter location is also shown in Figure 7.4. Immediately prior to a *call* instruction, the column offset corresponding to the caller function is stored in a global variable. That column offset is the caller function identifier multiplied by the counter size, which is four bytes. Then, at each callee function entry point, the column offset is added to the row offset and the counter table's base address. The row offset is the callee function identifier multiplied by the number of functions and the counter size. This method guarantees a unique counter location for each caller-callee function combination.

7.2 IMPACT Probe Descriptor File

In order to assign the counter values to the correct edges in the Mcode representation once the program has completed execution, a description of the meaning of the counters is necessary. A file called the *IMPACT probe descriptor* (IPD) file is written by Lbx86 Phase 2 as the probes are inserted. This file allows the monitor program that extracts the counter values as the application is exiting to construct the *weight file*. A utility program called *Lget* annotates the intermediate representation of the application with the weights. Details on the weight file and weight annotation can be found in [18]. The call graph profile, however, is written to a separate file. Each call graph arc from a caller to a callee with a weight greater than zero is written to the file in a format for easy call graph construction within the optimizer.

The records in the IPD file are shown in Figure 7.5. Each record corresponds to an entry in the weight file and specifies how to produce the actual weight. Each record begins with a record-type identifier that is followed by a tag that identifies the particular function, block, or

```
IPD FN (FUNCTION) record:
    WORD 0 : FN_ENTRY identifier
    WORD 1 : index into the function name symbol table in the ipd file
    WORD 2 : number of entry points (cb weights will be added together for a fn weight)
    WORD 3 thru WORD (2+(WORD 2)) : counter offsets for entry points

IPD CB (BASIC BLOCK) record:
    WORD 0 : CB_ENTRY identifier
    WORD 1 : cb id
    WORD 2 : counter offset

IPD J (UNCONDITIONAL JUMP) record:
    WORD 0 : J_ENTRY identifier
    WORD 1 : branch id
    WORD 2 : counter offset

IPD B (CONDITIONAL BRANCH) record:
    WORD 0 : B_ENTRY identifier
    WORD 1 : branch id
    WORD 2 : entry counter offset
    WORD 3 : fallthru counter offset

IPD J_RG (JUMP VIA JUMP TABLE) record:
    WORD 0 : J_RG_ENTRY identifier
    WORD 1 : branch id
    WORD 2 : number of targets
    WORD 3 thru WORD (2+2*(WORD 2)) : counter offsets for table targets

IPD CG (CALL GRAPH) record:
    WORD 0 : CG_ENTRY identifier
    WORD 1 : counter offset for caller ID location
    WORD 2 : counter offset for call graph table
    WORD 3 : number of counters
```

Figure 7.5 IMPACT probe descriptor file record formats.

branch. Following those two identifiers is information pertaining to the various counters needed to compute the weight for that type of record.

Each function in the IPD file begins with a function record. This record indicates which basic blocks are the entry points to this function and that the function weight is the sum of those block weights. The flow weights into the entry points are also taken from the weights of those entry-point blocks. The computation of these two sets of weights assumes that the blocks are only executed as entry points into the function and never as internal blocks. Block and unconditional branch entries simply specify which counter contains the weight. *Conditional branches* require the branch entry counter (same as its basic block counter) and its fall through counter allowing for the calculation of the taken weight. Jump table *jumps* require a counter for each possible target, and the corresponding entries specify those counter locations. Finally, the IPD file contains a single call graph record that contains the offset to the entire caller-callee table. Figure 7.6 is a partial example of the IMPACT probe descriptor file for *129.compress.*

7.3 Profile Counter Extraction

The profile weights generated during execution of the probed application must be written to a file prior to termination of the application. The monitor tool is design to run concurrently with the probed application and perform the extraction at the appropriate time. In order for the monitor tool to have access to the counters within the probed application's data space, standard process protection between these applications must be circumvented. This is accomplished by launching the probed application as a child process in debug mode from within the monitor. Debug mode allows the parent monitor program to arbitrarily read or alter various aspects of the child, including values stored in its data space. Furthermore, the parent and child processes operate in an interactive fashion, where the child is suspended and the parent reactivated whenever the child experiences a significant event. When the parent monitor program is finished

Address	ASCII	Hexadecimal	Description
0x00	IS..	49 53 00 00	IMPACT Probe Descriptor (IPD) file header
0x04	comp	63 6F 6D 70	
0x08	ress	72 65 73 73	
0x0C	_nt.	5F 6E 74 2E	Binary associated with this descriptor file
0x10	exe.	65 78 65 2E	
0x14	x.ex	78 2E 65 78	
0x18	e...	60 00 00 00	
0x1C		00 00 40 00	Preferred binary base address (00400000h)
0x20		00 30 41 00	Preferred counter table base address (00413000h)
0x24		73 12 00 00	Number of probes (1273h)
0x28		F9 44 00 00	Number of DWORDS in descriptor array (44F9h)
0x2C		FD 00 00 00	Number of functions (FDh)
0x30		00 00 00 00	Function record
0x34		00 00 00 00	Function name is symbol 0
0x38		01 00 00 00	1 entry-point
0x3C		00 00 00 00	1st entry-point weight is at offset 0
0x40		01 00 00 00	Basic block record
0x44		01 00 00 00	Basic block 1
0x48		00 00 00 00	Basic block weight is at offset 0
0x4C		03 00 00 00	Conditional branch record
0x50		00 00 00 00	Branch 0
0x54		24 00 00 00	Branch execution weight is at offset 24h
0x58		28 00 00 00	Branch fall-through weight is at offset 28h
...		...	

Figure 7.6 Example of an IMPACT probe descriptor file for SPEC95 *129.compress*.

performing any tasks related to the event, it reactivates the child and suspends itself waiting for the next event. Such significant child events include the loading of DLLs and program termination.

Specifically, when the operating system executes on behalf of the child process, the OS sends an `event` message to the monitor program. Among the various events sent to the monitor, the following have obvious significance for our purposes: `CREATE_PROCESS_DEBUG_EVENT`, `EXIT_PROCESS_DEBUG_EVENT`, `LOAD_DLL_DEBUG_EVENT`, and `EXCEPTION_DEBUG_EVENT`. Note that the monitor is notified of `EXIT_PROCESS_DEBUG_EVENT` just prior to the actual termination of the child, allowing the monitor to extract the counters from the child's data space before the child is terminated. Upon handling these and other events, `_ContinueDebugEvent()` is called by the monitor, which resumes execution of the child; and then `_WaitForDebugEvent()` is called, which suspends the monitor until the next event.

An alternative to allowing an outside program to extract the counters is to instrument the application to write out the counters on its own. The C functions `_atexit()` and `_onexit()` install functions which will be called upon termination of the program. Specifically, a routine to write the counter values to a file could be installed in this manner. However, the binary optimizer operates on the assembly level. Such a cleanup function would require the use of a file output function to write the counter values to the file. Because symbols in the original application are not necessarily required, it may be impossible to locate the library output functions. Additional output functions could be added (at the same time as the probes are inserted) that produce the weight file, but the monitor program approach is simpler.

7.4 Speedup of Probed Executables

The addition of the probes and the jump table bounds checks add overhead to the execution. While these probes are not included in any optimized version of the application, it is useful to briefly present the speedup of the probed applications. The probed results are shown in Figure 7.7 and are compared to the original executables as the baseline. Since the functions are not placed in the same order as in the original executable, as will be discussed further in Section 9.1, a comparison is also made to a version of the application with the same alternate

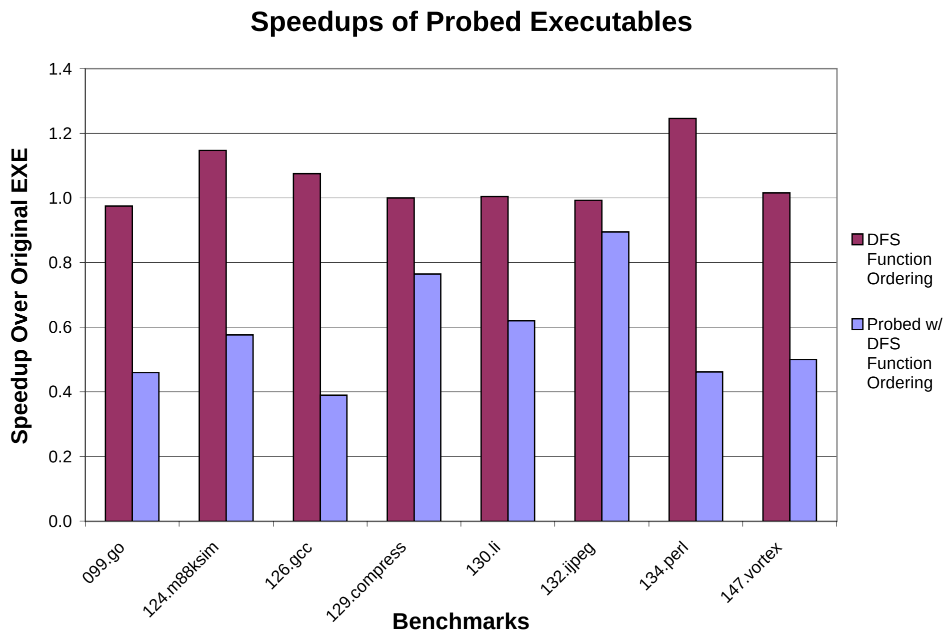


Figure 7.7 Probed application speedups.

function ordering, but without the probes. The probed applications require on average twice as long to execute as the originals.

CHAPTER 8

BASIC SUPPORT FOR MEMORY DISAMBIGUATION

Like register accesses, memory access ordering is also constrained by dependences. A flow dependence exists between the write to a particular memory location and a successive read from it. Output and anti dependences also exist. However, it is often difficult for the compiler to determine what memory location will actually be accessed by a *load* or a *store* instruction. Therefore, in order to maintain correctness, two memory instructions are assumed dependent if there is any chance that they refer to the same memory location. The process of determining whether or not memory accesses refer to the same location is called *memory disambiguation.*

The easiest types of memory references to disambiguate are constant address references, which are most commonly global variables. A simple comparison of the addresses encoded in the instructions determines whether the accesses refer to the same memory location. Another common memory reference is into stack space. An analysis is employed to calculate the change in the stack pointer between the two instructions. By combining that difference with the offsets from the stack pointer in the two instructions, location independence can be determined. Furthermore, it is highly unlikely that a constant address reference would access stack space, and vice versa. Therefore, a constant address reference and a stack reference are always assumed to be independent. Future work will include accurate disambiguation which will eliminate the need for this assumption. Note that because of the limited number of registers, the process stack is heavily used for local variable storage and for parameter passing in the x86 architecture. This makes disambiguation of pointers into stack space particularly important.

Other memory references are based off of register values. It is sometimes possible to trace backward in the program to the origins of the register value. This type of advanced analysis in a compiler can sometimes determine what space (global variable, heap, stack, read-only data, etc.) a memory reference accesses and can sometimes even localize the reference to a variable, list, or array. However, these analyses require label information passed from the front-end of the compiler. Such information is not available to the reoptimizer. Future work on this problem includes a more advanced pointer analysis at the binary level and a feasibility study on encoding some label information into the binary itself.

CHAPTER 9

PROFILE-BASED OPTIMIZATION

Using the information collected from the profiled code, the important functions, blocks, and paths can be more effectively optimized. The following sections describe some initial attempts at profile-based optimization within the framework. However, they have not been highly tuned, as their purpose is to show the types of binary optimizations that might benefit from profile information.

All benchmarks used in this thesis are compiled with Microsoft Visual C++ 5.0 with the *maximize for speed* and the *inline all suitable* settings. Each benchmark is trained on its designated training input (only the jumble input for 134.perl) and executed for performance measurement on its designated reference inputs. The performance measurements were taken on an AMD K6-2 300-MHz processor with 64 MB of main memory running Microsoft Windows 98 for all but 147.vortex, which was run on an AMD K6 200-MHz processor with 64 MB of main memory running Microsoft Windows NT.[1] All performance measurements were taken while running the standard reference inputs. The *time* utility was used to gather these measurements, along with *make* and *bash*, from the *Cygwin32* [19] tool set. Each version of the benchmark was executed three times, and the arithmetic average of the wall clock run times for each version was gathered. Speedups are reported according to the following equation:

$$Speedup = \frac{original\_running\_time}{new\_running\_time}$$

[1]147.vortex compiled with Microsoft Visual C++ 5.0 as specified above memory faults under Microsoft Windows 98 but executes correctly under Windows NT.

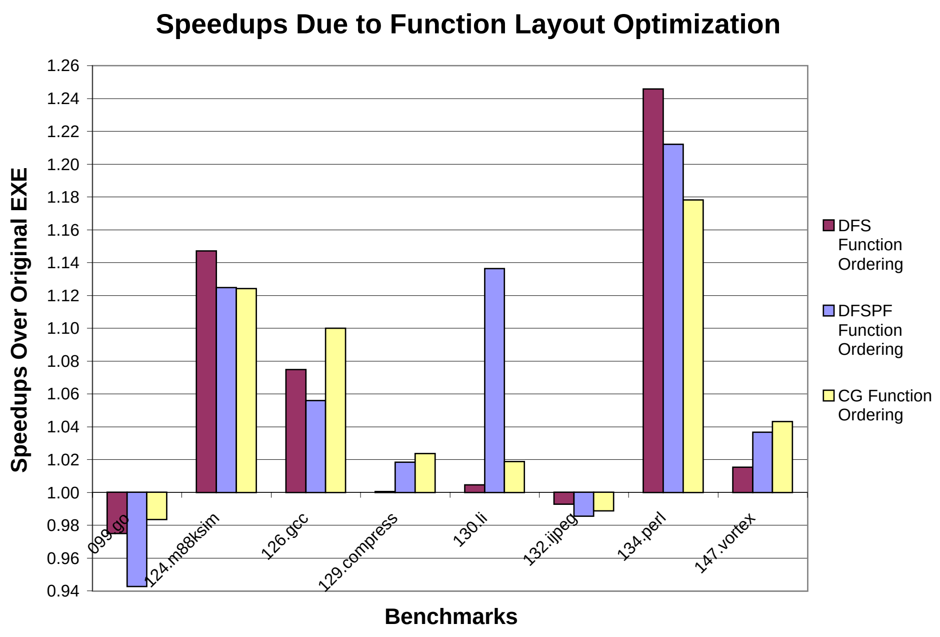
Other optimizations were performed along with the optimizations described in this thesis. Those optimizations serve as the baseline for the reoptimizer to which new optimizations are compared, and will be referred to as the *K6 optimizations.* Specifically, several K6-specific optimizations are employed which were implemented by Thiems [10], and instruction rescheduling is performed using a machine description implemented by George [4]. Scheduling is performed in the IMPACT Schedule Manager [20]. The machine description model is constructed in IMPACT's Machine Description Language [21], and only models the decoders in the K6, which allow decode of two short or one long instruction per cycle.

9.1 Function Layout

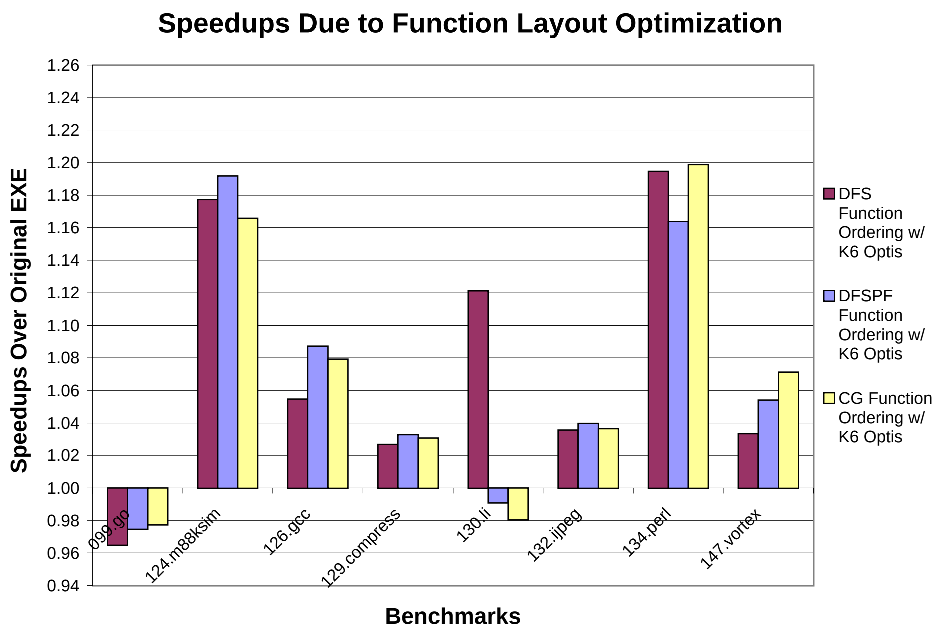
One promising optimization is to intelligently order the functions within the executable, called *function layout.* Three layout schemes are supported in the framework, two of which utilize profile information. The first ordering places functions according to a depth-first search (DFS) of the program call graph (statically constructed call graph) as produced by the decoder. Through an examination of the profile weights of the functions, it was found that the most heavily executed functions were often scattered throughout the DFS layout. This scattering may increase cache conflicts as several heavily executed functions may map into the same cache lines while never executed functions map into others. The second algorithm, then, packs forward all functions executed more than ten times in DFS order followed by the remaining functions in DFS order (DFSPF). A third algorithm implements a method developed by Hwu and Chang [22] and enhanced by Pettis and Hanson [23] which utilizes a call graph (CG). This algorithm places functions that call each other frequently near each other, so that they will not likely conflict with each other in the cache. Because accurate call target information is necessary, including indirect call targets, the call graph gathered during profiling is used to guide function layout.

Figure 9.1(a) shows the speedups of the three function layouts compared to the original applications. Figure 9.1(b) shows the speedups of the three layouts when the K6 optimizations implemented by Thiems [10] and rescheduling (using a machine description that models the two short decoders) implemented by George [4] were also applied. In general, the K6 optimized executables have higher speedups than the non-K6 optimized executables. It might also be expected that the CG layout would perform better than the others as it uses the profiled call graph. However, the training input does not always accurately represent the reference input, and thus layouts may not truly be optimized under reference conditions. Furthermore, cache behavior can vary widely when the positions of key instructions are moved even slightly. None of the algorithms account for cache size, and thus important functions may still conflict in the cache due to the cache's modulo indexing mechanism. Future work includes a more detailed cache simulation of the 130.li benchmark to further understand its varied behavior, and improvements to the CG layout algorithm to account for the cache size. The CG layout algorithm will be used as the baseline layout optimization throughout the remainder of this thesis.

Because of restrictions in the Mcode function representation, it is difficult to marshal the functions and data blocks within the text section back into their original order. First, jump tables are intimately associated with their referring function and are automatically placed immediately after their referring function in the new layout. It would be difficult in the current implementation to place the jump tables elsewhere, and thus it may not be possible to preserve their original program placement. Likewise, functions must remain contiguous in the current



(a) Function layout optimization only.



(b) Function layout optimization with K6 optimizations.

Figure 9.1 Speedups with the three function layout optimizations applied.

framework, thus making it difficult to place multiple entry-point functions, which are often scattered across the original binary, in their original positions. Because of these complexities, there is no compelling reason to attempt such an ordering. Note that the blocks within a function are placed in original program order unless a *block layout* optimization, such as the optimization described in the next section, is applied.

9.2 Superblock Formation

The superblock [24] has been shown to be an effective tool for ILP compilation. Of critical importance is the selection of a frequently executed path to form into a superblock. The basic blocks in the superblock will become straight-line code and have no side entrances due to tail duplication. While, at this time, no superblock-specific optimizations are performed on the superblock itself, it is believed that some benefits may still be reaped. In particular, branch prediction should be more accurate because other paths (side entrances) through the code have been eliminated and all branches are expected to fall through to the next block in the superblock. Furthermore, since all of the blocks are now in a straight line, there should be fewer instruction cache misses.

Because of the limited support within the x86 architecture for exposed ILP, superblocks must be chosen and formed carefully. Inherent to the superblock are features that may negatively impact performance. For example, moving less frequently used paths away from the superblock may replace natural fall-through paths with *unconditional jumps*, adding to the number of instructions executed for these paths. In addition, tail duplication increases code size, which may adversely affect cache performance. This optimization is model specific as the cost of mispredicted branches and cache misses varies from model to model. For example, if the cost

Table 9.1 Superblock formation parameter settings.

Name	Cons. Value	Aggr. Value	Meaning
LB_maximum_code_growth	2.0	2.0	Per function, factor of code growth allowed due to tail duplication
LB_parm_branch_prediction_method	profile	profile	Use static or profile weights to predict branches
LB_minimum_superblock_weight	100	50	Start block must be executed at least this many times
LB_min_branch_ratio	.90	.70	For a conditional branch, the weight of an out going flow arc must be greater than or equal to this factor of the branch weight for superblock inclusion of the successor
LB_trace_min_cb_ratio	.90	.90	For reverse superblock growth, the arc from the heaviest of the predecessor blocks must at least this factor of the superblock weight for inclusion.

of a branch misprediction is extremely high in a particular model, a more aggressive effort to form superblocks along the frequently executed paths may reap significant benefits.

Table 9.1 lists the parameters that guide IMPACT's general superblock formation algorithm. Parameters chosen for both an aggressive and a conservative formation are shown. Future work on superblock formation within the reoptimizer consists of tuning the formation heuristics and parameters more appropriately for x86 architectures and developing some superblock optimizations. The speedups for superblock formation are shown in Figure 9.2. In general, the conservative approach performs better, although once superblock optimizations are applied, the performance of the aggressive approach is likely to increase. Again, more in-depth study of the 130.li benchmark is necessary.

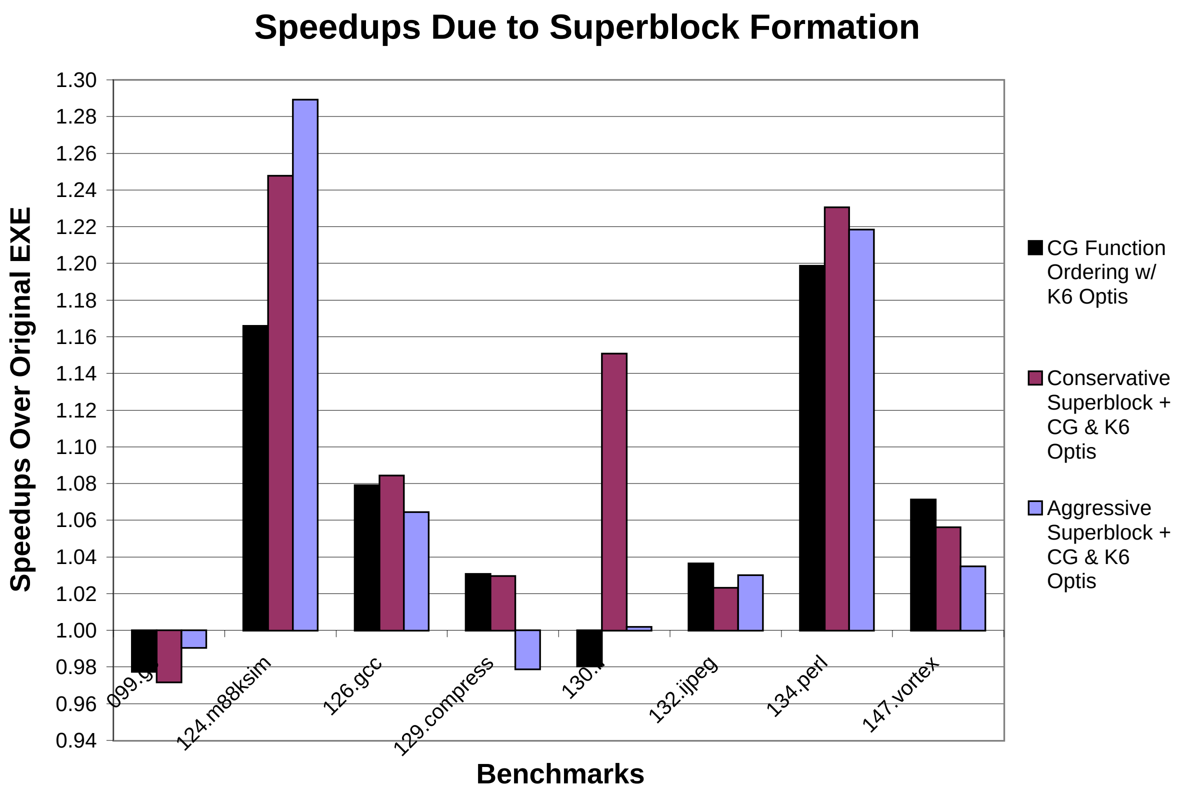


Figure 9.2 Speedups for the superblock formation algorithms in addition to CG function ordering and K6 optimizations.

9.3 Push and Pop Chain Optimizations

The goal of many instruction-level-parallelism optimizations is to break dependences between instructions allowing more of the instructions to be executed in parallel. One such optimization recommended for AMD's K6 processor is the *stack allocation* optimization, or *push chain* optimization [25].

The push chain optimization breaks a chain of stack *push* instructions into a single large stack allocation instruction and a set of *move* instructions into the allocated space, as shown

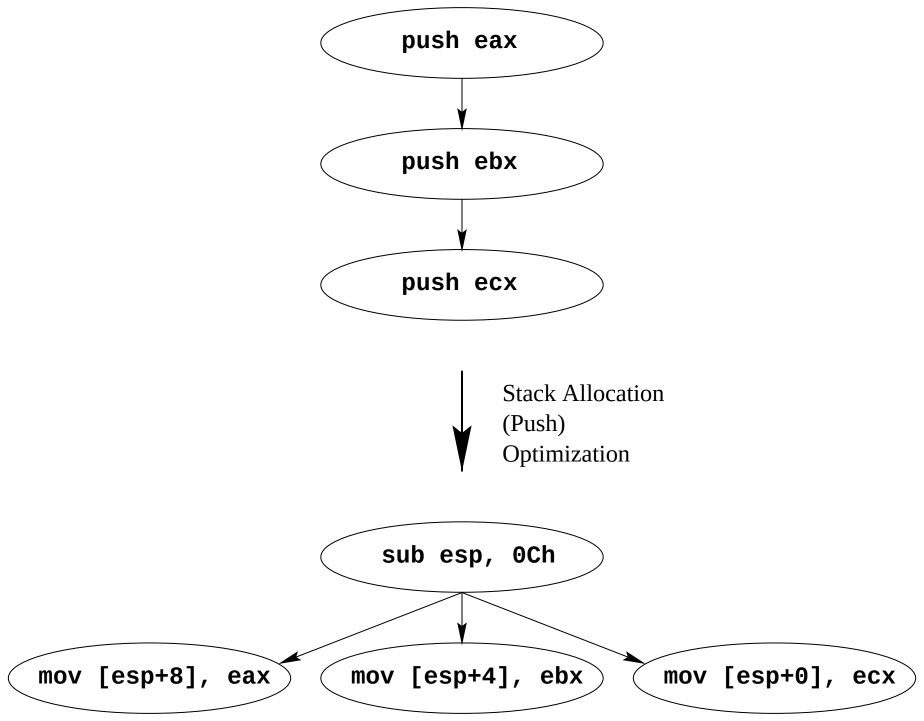


Figure 9.3 Example of the stack allocation (push) optimization.

in Figure 9.3. Between each *push* instruction in the original chain is a flow dependence via the stack pointer because each instruction reads the stack pointer, `esp`, and modifies it to allocate space for the new data element. Thus, the next *push* must wait to read the stack pointer until the previous *push* completes its modification of the stack pointer. These flow dependences cause the instructions to be executed sequentially.

This optimization is model specific because other models may have automatic hardware mechanisms that can execute *push* instructions in parallel. Furthermore, other hardware specifics in models without that automatic hardware, such as the number of store units, may have an effect on the ideal code generated by this optimization.

Pushes are commonly used to temporarily spill register values to memory in order to free up registers and to pass parameters to called functions. In some microarchitectures, *stores* to memory cannot be reordered by the hardware. The push optimization allows for the *pushes* to be converted into a set of independent *stores* which can then be reordered by the reoptimizer's scheduler to allow for greater concurrent execution of instructions.

Much like the push chain optimization, a series of *pops* can also be converted into a series of *loads* followed by a stack deallocation. Also observed are stack allocation instructions intermixed in a series of *pushes* and stack deallocation instructions intermixed in a series of *pops*. Such sequences are also optimized, as the extra allocated space is wrapped in with the new series allocation instruction and all stack pointer offsets are carefully adjusted.

The results of these optimizations are shown in Figure 9.4. For the stack optimizations to be applied, the sequence must consist of at least three *pushes* or three *pops*. Because code size can increase, the optimization is only applied when the sequence executes at least five times in our aggressive approach and at least fifty times in our conservative approach. Unfortunately, this optimization also can increase the total dependence height of the sequence on machines with only a single store unit, such as the K6. Thus, a more conservative approach can be taken with this optimization such that it is only performed when an existing allocation or deallocation instruction exists. With the push optimization, all allocation instructions will be wrapped into a single allocation instruction. Such an optimization will not increase the dependence height and may decrease it if a number of allocation instructions are wrapped into one.

The speedups due to the stack allocation optimizations vary. Note that the number of instructions and the code size may have increased due to the optimization, and thus cache performance may degrade. Further research into the optimal situations to apply these optimizations is necessary along with study of their performance on processor models with more than one store unit.

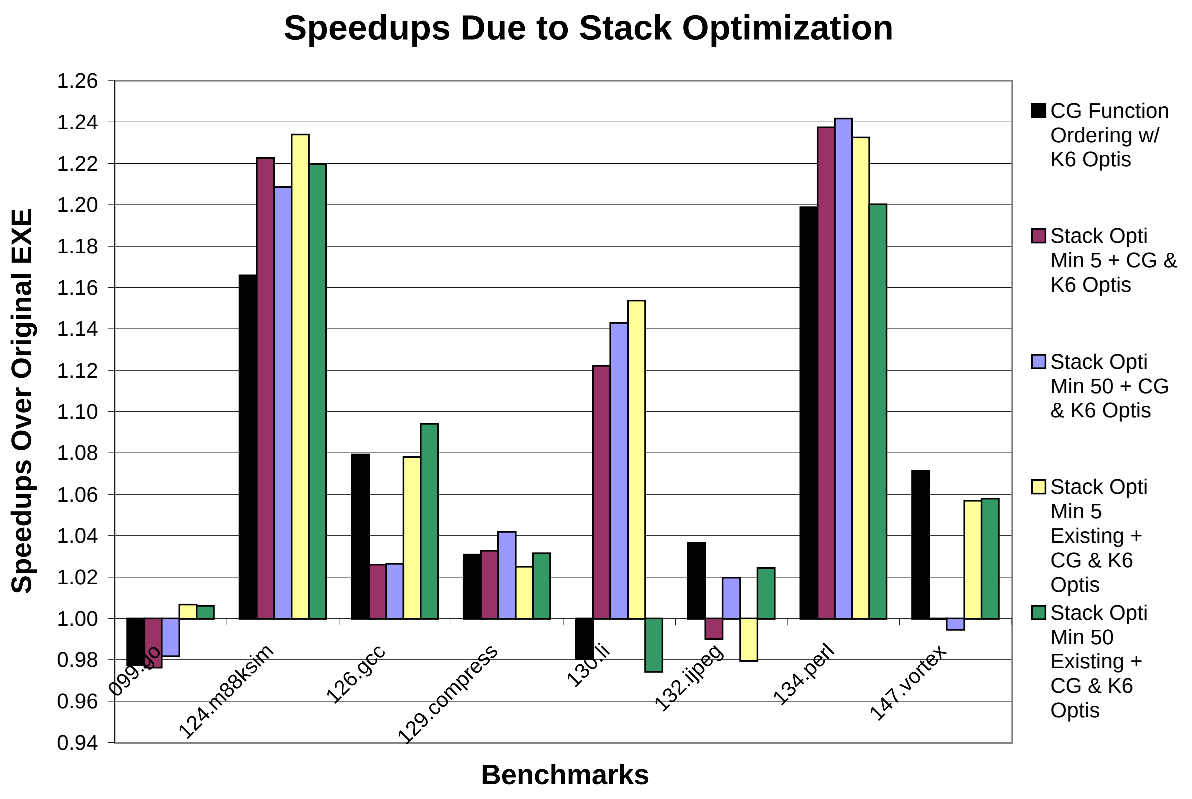


Figure 9.4 Speedups for the stack allocation algorithms in addition to CG function ordering and K6 optimizations.

CHAPTER 10

FUTURE WORK AND CONCLUSION

This thesis describes a powerful mechanism for optimizing binary executables, Dynamic-Link Libraries, and object files using model-specific and profile-based optimizations. Many of the algorithms used within IMPACT's x86 binary reoptimization framework for decoding, conversion, profiling, and reoptimization are discussed in detail. Combined with Thiems [10] and George [4], a complete picture of the framework is presented. The major features of the system include: static binary decoding, complete x86 user instruction set representation in an intermediate representation, probing and profile gathering, model-specific and profile-based optimization, and optimized binary reconstruction.

Specifically, this thesis presents heuristics that allow for stable decoding and reoptimization of traditional C applications. However, the heuristics used are empirically derived and future work will focus on improving the robustness of decoding through both the use of more advanced, provable algorithms, and through the addition of a minimal amount of extra information into the binary that will be inserted by the compiler. A simple edge weight binary profiling mechanism is then described that does not rely on dynamic memory allocation or file manipulation functions internal to the binary, but rather Windows-specific process debugging features for profile collection. Superblock formation is presented as an example of profile-based optimization. Likewise, the stack allocation optimization is presented as an example of model-specific optimization. Promising performance improvements are shown for these two optimizations and the function layout optimizations. Future work includes an investigation into new model-specific optimizations and profile-based optimizations.

The promising results presented in this thesis, and in the other IMPACT x86 binary reoptimization theses, show that significant potential exists for binary reoptimization. Furthermore, this framework shows that a binary can be profiled and reoptimized by the user, rather than just by the vendor, to improve performance. While this framework is specific to x86 32-bit Windows applications, many of the techniques should apply equally well to binary reoptimizers for other architectures.

REFERENCES

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

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

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

[4] C. N. George, "A framework for install-time optimization of binary DLLs," master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1999.

[5] W. Dugal, "Code scheduling and optimization of a superscalar x86 microprocessor," master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[6] B. Sander, "Performance optimization and evaluation for the impact x86 compiler," master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

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

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

[9] M. C. Merten and M. S. Thiems, "An overview of the IMPACT X86 binary reoptimization framework," Tech. Rep. CRHC-98-05, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, June 1998.

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

[11] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 158–168.

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

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

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

[15] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Santa Clara, CA, 1997.

[16] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1995, December 1995.

[17] B. L. Deitrich, "Static program analysis to enhance profile independence in instruction-level parallelism compilation," PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[18] T. L. Johnson, "Automatic annotation of instrictions with profiling information," master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[19] G. J. Noer, "Cygwin32: A free Win32 porting layer for UNIX applications," in *2nd USENIX Windows NT Symposium*, p. 31, August 3–4 1998.

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

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

[22] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989, pp. 242–251.

[23] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[24] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[25] Advanced Micro Devices, "AMD-K6 MMX enhanced processor code optimization application note," Tech. Rep. 21828, Advanced Micro Devices, Sunnyvale, CA, August 1997.