DYNAMIC INSTRUCTION STREAM EDITING

Marc Corliss

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2006

E Christopher Lewis
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

Many people deserve thanks for helping me navigate through my PhD. First and foremost, I must thank my wife, Stephanie, for her loving support without which I certainly would not have succeeded. She is a wonderful companion, and I feel like the luckiest man on the planet to be married to her. I thank her for her patience through my many long work days, and for helping me stay sane through my many deadlines.

My parents, Art and Nancy, were also extremely supportive throughout my six years in graduate school. I greatly appreciated their loving phone calls, emails, and visits. They have always been there for me. I also must thank, my brother, Ryan, my grandmother, Barbara, as well as Stephanie's family. Their encouragement and loving support certainly helped me through my PhD.

My advisor, E Christopher Lewis, is chiefly responsible for my academic and professional development. I have benefitted profusely from his guidance and support. I learned from E what it means to deeply understand a research problem, and to always consider the broader impact of my research. E is also an incredible teacher, breaking the most complicated concepts down into simple manageable pieces. I will try to emulate these skills in my next endeavor as a professor. I am still not entirely sure why he decided to take a chance on a lowly Master's student with no research experience, but I am grateful that he did. I think he would agree that it worked out wonderfully.

I also owe thanks to Amir Roth, my research collaborator and dissertation chair, who worked with E and myself on much of this project. Amir was another important mentor to me that I learned a great deal from. In particular, his guidance in my first three years in graduate school was instrumental to my development as a researcher. He also provided crucial help on this dissertation.

The third faculty member I owe thanks to is Milo Martin. Together, Milo, Amir, and

E make up the faculty members of the Architecture and Compilers Group (ACG) at the University of Pennsylvania, which has been my research home. Although I never worked with Milo on any research projects, I have met and discussed my work and career goals with him on countless occasions. In addition, Milo served on my dissertation committee, and helped me a great deal in writing and completing this disssertation.

The other two members of my dissertation committee, Jonathan Smith and Calvin Lin, were also helpful in writing this dissertation. In particular, their outside perspectives on this work helped shape the end result.

Many students at the University of Pennsylvania made life for me much more pleasant during graduate school. First, I must thank Gary Zhang and Sid Suri, my two closest friends. I hope that we will remain close even as we go off in separate directions. I also must thank Anne Bracy and Vlad Petric, two other close friends, and fellow members of ACG who often helped me out when I was stumped in my work, especially early on. In addition, some other students that I should thank for their friendship and support (this list is by no means exhaustive) include Stanislav Angelov, Colin Blundell, Aaron Evans, Drew Hilton, Andrew McGregor, Tingting Sha, Marcelo Siqueira, and Kilian Weinberger.

Finally, I should mention that this material is based upon work supported by the National Science Foundation under Grant No. 0311199.

ABSTRACT

DYNAMIC INSTRUCTION STREAM EDITING

Marc Corliss

E Christopher Lewis

This dissertation proposes a novel, cooperative hardware/software mechanism, called *DISE* (*dynamic instruction stream editor*), for efficiently transforming programs. DISE transforms programs using programmable instruction macro-expansion. It resides within the processor inspecting every fetched instruction. Based on user-defined rules, it macro-expands some of those instructions into parameterized replacement sequences.

DISE can express a broad range of transformations including transformations for profiling program characteristics, implementing interactive debugging primitives, decompressing compressed programs, and detecting stack and pointer smashing attacks. This dissertation describes the functionality, interface, and system architecture of DISE and proposes one implementation of this architecture. Our evaluation demonstrates that DISE transformation is highly efficient. Unlike transformation mechanisms implemented entirely in software, DISE has no impact on instruction cache performance because it transforms instructions within the processor. Furthermore, the performance cost of macro-expanding instructions is neglible, which is not true of software mechanisms (although some mechanisms transform code statically rather than at runtime). The only significant performance cost of DISE transformation is executing the additional instructions, and this overhead is usually less than 25% for most transformations and benchmarks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing has changed significantly over the last few decades. Previously, the central concern was performance, but recently, concerns have shifted to other metrics such as security, reliability, and program size, to name only a few. Some contributing factors for this shift include the growth of the Internet, the ever increasing complexity of both hardware and software systems, and the emergence of embedded systems. For example, because some programs are now internet accessible, they are more prone to attack. In these programs, security is the primary concern.

Unfortunately, not all users care equally about each metric for any particular program. There are many programs for which some users are primarily concerned with one metric (*e.g.*, performance), while other users are primarily concerned with another metric (*e.g.*, security). As a result, programs are often *customized* for each individual user and program.

1.1 Customizing Programs

Users customize programs in many ways.

Debugging. First, when developing a program, a developer customizes the program for debugging. For instance, the developer might augment the program with code to detect whenever a particular memory value is altered, *i.e.*, a *watchpoint* in an interactive debugger. Watchpoints are important for finding memory corruption bugs that crop up in languages like C and C++ in which memory allocation is managed explicitly by the programmer.

Profiling. Once the program is ready for distribution, the developer might customize it for profiling. For example, the developer might choose to trace memory writes. This information can be used to optimize the program for the context in which it is run (*e.g.*, rescheduling instructions).

Code size. Alternatively, a developer porting the program to an embedded machine, where memory and cache size are limited, customizes for code size. In this case, customization is split into two phases: first, the program is compressed and ported, and second, the program is decompressed at runtime (preferably, in small chunks).

Security. A consumer (*i.e.*, user) might customize the program for security. For example, suppose the program is internet accessible. Malicious users can potentially supply unexpected inputs to subvert the program. In this case, the user might want to insert code into the program to detect such attacks.

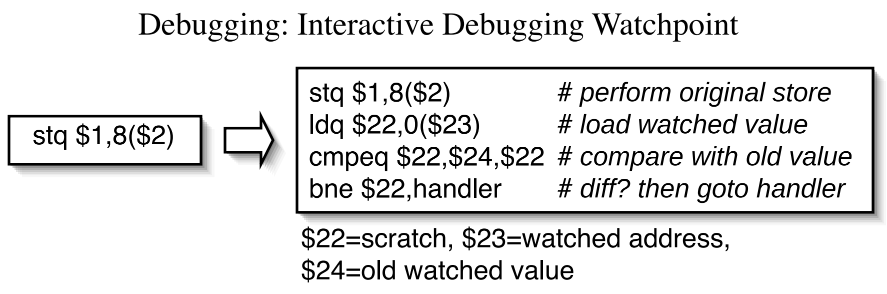
Finally, other consumers may not be willing to tradeoff security for the additional overhead of the detection code. These users may not customize the program at all.

1.2 Customization Mechanisms

There are two existing approaches to building customization mechanisms: building it in software or in hardware.

Software approaches. To customize an application in software, the customization mechanism performs *program transformation*. Each of the customizations described above can be implemented using a program transformation. For example, Figure 1.1 shows the debugging customization that implements a watchpoint in an interactive debugger. Although not a requirement, the transformation is performed at the assembly-level[1]. The transformation replaces all stores (*e.g.*, **stq** and **stl**) with a code sequence that performs the original store, and then reads the value of the watched location to check if it was modified. If it was, then a handler routine is called. In the code sequence, three registers are reserved for

[1]In this dissertation, we use Alpha assembly code [81].


Debugging: Interactive Debugging Watchpoint

```
stq $1,8($2)
```

```
stq $1,8($2)            # perform original store
ldq $22,0($23)          # load watched value
cmpeq $22,$24,$22       # compare with old value
bne $22,handler         # diff? then goto handler
```

$22=scratch, $23=watched address,
$24=old watched value

Figure 1.1: Customizing for debugging: a transformation that implements an interactive debugging watchpoint. Assume registers **$22-23** are available and for use by the transformed code.

the transformed code: **$22** for scratch, **$23** to hold the watched address, and **$24** to hold the previous value of the watchpoint.

These software customizers are called *translation mechanisms*. There are a number of different types of translation mechanisms. One natural translation mechanism is the compiler. However, compilation is usually done by the developer and it is impractical to expect developers to anticipate all of their users' needs. Even if they could, they would have to compile and manage many variants of each application. A more practical approach is to use a client-side facility that transforms the executable.

One proposed client-side facility is a *programmable software translator*, which transforms the static program image [14, 33, 46, 57, 64, 69, 73, 78, 82]. For example, a software translator can perform the transformation in Figure 1.1 that implements a debugging watchpoint. Of course, because the translator is programmable, it can be used for many other transformations such as to enhance security or profile the application.

Hardware approaches. Alternatively, other researchers have proposed using *dedicated hardware widgets* to customize programs within the processor [19, 59, 99]. Hardware widgets do not transform a program, but instead are used to augment or monitor the running program. For example, most processors provide debugging registers for implementing watchpoints. An interactive debugger sets one of the debugger registers with the address of the watchpoint. Whenever that memory value is written to, the processor traps to the operating system, which then transfers control to the interactive debugger.

Software versus hardware. These two approaches have complementary advantages and disadvantages. First, software customizers are programmable and thus, flexible. With software translation mechanism, users can easily specify new customizations or modifying existing ones. Software translators are also general-purpose. They export a rich and expressive API with which users can specify almost any transformation (*i.e.*, customization). But at the same time, software translators are inefficient. The additional code is spliced directly into the program image, degrading instruction cache performance. Furthermore, the transformation cost is often high, and if performed at runtime, is perceived as additional overhead.

On the contrary, hardware widgets are highly efficient. The cost of monitoring the program is minimal and the program footprint is unchanged, so there is no impact on instruction cache performance. However, hardware widgets are dedicated to a particular customization (*e.g.*, interactive debugging watchpoint). So, for example, to profile an application requires a new hardware widget. In addition, hardware widgets are not flexible. Modifying the functionality of a hardware widget means modifying hardware. Finally, hardware widgets are limited in their uses since they do not transform programs.

This work seeks the best of both worlds: a customization mechanism that is both flexible and general-purpose as well as highly efficient. To achieve these goals, we propose a hybrid approach, which we call *Dynamic Instruction Stream Editing (DISE)* [22].

1.3 DISE

DISE is effectively a programmable hardware translator for transforming programs. Because DISE is programmable, transformations (*i.e.*, customizations) are easily added, removed, or modified. In addition, this dissertation will demonstrate the formulation of many different types of transformations in DISE, including transformations for debugging [25], decompression [23, 26], and security [24, 21]. Because DISE is a hardware mechanism, transformation is also efficient. As described later, the transformation cost is negligible (although there is the overhead of executing the transformed code). In addition, the program image is unaltered and consequently instruction cache performance does not suffer.

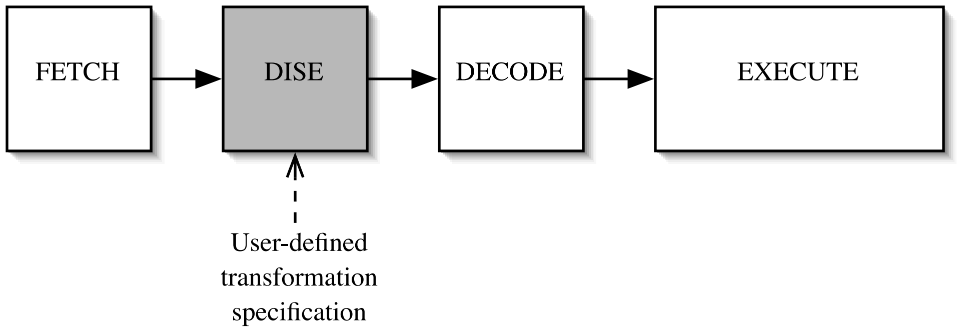


Figure 1.2: A simplified diagram of a DISE processor.

DISE transforms programs using instruction macro-expansion. DISE takes as input individual instructions and outputs instruction sequences based on the inputted instruction. As shown in Figure 1.2, DISE inspects every fetched instruction, macro-expanding on some of the instructions, and the translated instruction stream is passed to the decoder.

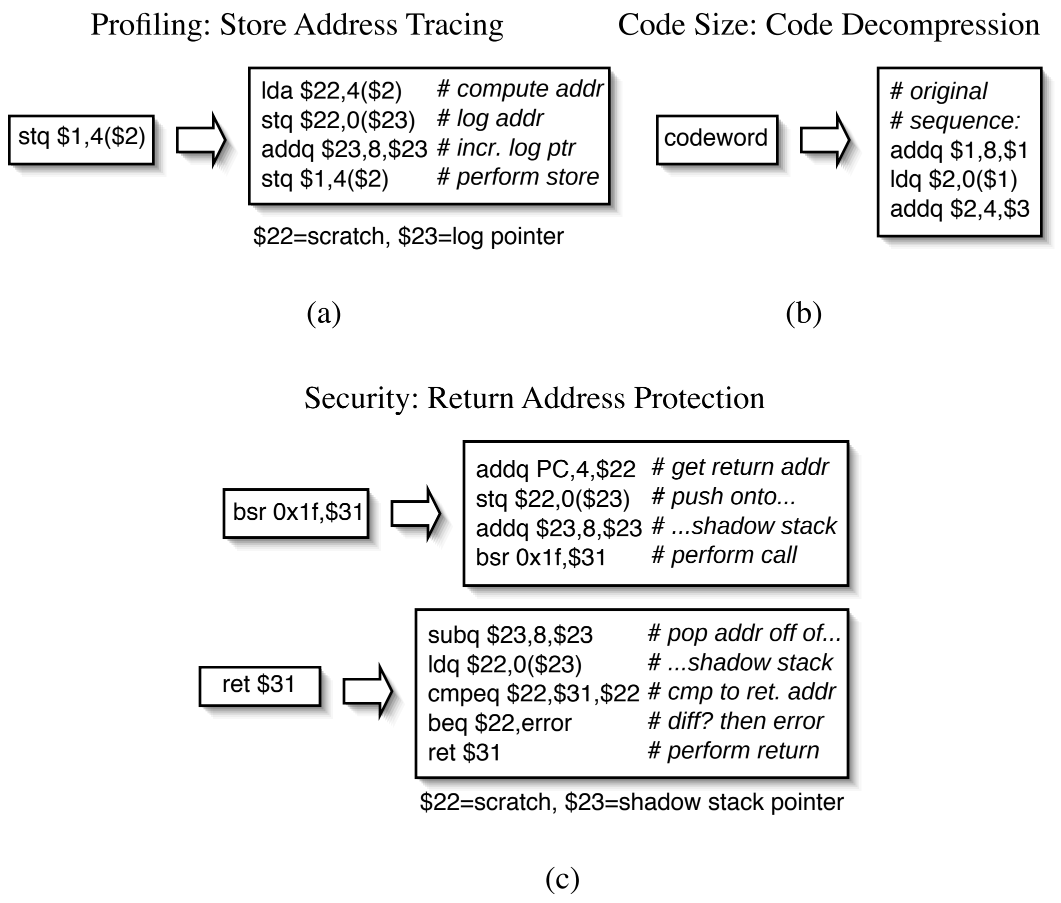
DISE essentially allows users to change the semantics of existing instructions or to define new instructions. These two capabilities allow DISE to express many transformations.

1.4 Example DISE Transformations

Figure 1.1 and Figure 1.3 show four different transformations, all of which can be performed by DISE. First, DISE can implement an interactive debugging watchpoint (*i.e.*, customizing for debugging). DISE changes the semantics of store instructions, macro-expanding each store into a code sequence that performs the original store and then checks if the watched location was modified (as shown Figure 1.1).

DISE can also trace store addresses (*i.e.*, customizing for profiling). Again, DISE changes the semantics of store instructions, macro-expanding stores into a sequence that logs the PC and the memory address before performing the original store (as shown in Figure 1.3(a)).

DISE can also perform dynamic code decompression (*i.e.*, customizing for code size). First, a static binary rewriter replaces frequently-occurring instruction sequences with compressed codewords (instructions that use a reserved opcode). DISE defines these new


Profiling: Store Address Tracing
Code Size: Code Decompression
(a)
(b)
Security: Return Address Protection
(c)

```
stq $1,4($2)  =>  lda $22,4($2)      # compute addr
                  stq $22,0($23)     # log addr
                  addq $23,8,$23     # incr. log ptr
                  stq $1,4($2)       # perform store
                  $22=scratch, $23=log pointer

codeword  =>  # original
              # sequence:
              addq $1,8,$1
              ldq $2,0($1)
              addq $2,4,$3

bsr 0x1f,$31  =>  addq PC,4,$22    # get return addr
                  stq $22,0($23)   # push onto...
                  addq $23,8,$23   # ...shadow stack
                  bsr 0x1f,$31     # perform call

ret $31  =>  subq $23,8,$23       # pop addr off of...
             ldq $22,0($23)       # ...shadow stack
             cmpeq $22,$31,$22    # cmp to ret. addr
             beq $22,error        # diff? then error
             ret $31              # perform return
             $22=scratch, $23=shadow stack pointer
```

Figure 1.3: Three DISE transformations: (a) store address tracing (customizing for profiling), (b) dynamic code decompression (customizing for code size), and (c) return address protection (customizing for security). These transformations assume registers **$22-23** are available and for use by the transformed code.

instructions; it macro-expands them into the decompressed sequences (as shown in Figure 1.3(b)).

Finally, DISE can protect memory-resident return addresses (*i.e.*, customizing for security). This protection is accomplished by changing the semantics of call and return instructions. DISE macro-expands call instructions into a code sequence that saves the return address to a shadow stack before performing the original call (as shown in Figure 1.3(c)). DISE macro-expands return instructions into a code sequence that verifies the actual return address matches the corresponding address on the shadow stack before performing the original return (as shown in Figure 1.3(c)).

Although DISE has many uses, it cannot perform arbitrary transformations. DISE allows users to either change the semantics of existing instructions or to define new ones. DISE is limited to transformations that can be formulated using one of these approaches. Furthermore, it is limited to peephole transformations, transformations of one instruction at a time. DISE cannot perform transformations that require multi-instruction windows such as basic blocks or traces. However, we have not found any uses of multi-instruction matching that would warrant the added complexity and performance overhead of the implementation.

1.5 Contributions

This dissertation makes the following contributions:

- **Designs a programmable, hardware-based translation mechanism that can implement a range of transformations.** We show the utility of DISE in three contexts.
  - Debugging. DISE can implement both breakpoints and watchpoints in an interactive debugger.
  - Code compression. DISE can serve as a dynamic code decompressor.
  - Security. DISE can inject code into an application to detect stack and pointer smashing attacks.
- **Presents a preliminary system-level design and proposed hardware organization for DISE.**
- **Evaluates the performance characteristics of DISE.** Our evaluation shows the following:
  - Overhead of DISE. We show that the overhead of transformation in DISE is low. We demonstrate this result in the context of profiling, security, code decompression, and debugging.

- Performance of DISE versus software translation. In particular, it shows that DISE outperforms software translation and that trends in application workloads (*i.e.*, larger memory footprints) will widen this gap.
- Sensitivity to design parameters. In particular, we evaluate the performance of DISE as we vary microarchitectural characteristics as well as the transformation.

1.6 Differences from Previous DISE Publications

Some of the work presented in this dissertation was published previously [22, 23, 24, 25, 26]. This dissertation extends these earlier documents by explicitly defining the DISE programming interface, describing the compilation process of specifications, and introducing additional DISE transformations (*e.g.*, pointer smashing protection).

This dissertation also makes some simplifications of the DISE mechanism. First, the PT and RT are treated as caches of memory-resident patterns and specifications. Second, the microarchitecture described in this dissertation is targeted for a RISC (Alpha-like) machine (although a CISC implementation may also be possible). Finally, in this dissertation DISE is a user-level tool only. This restriction simplifies the operating system support as well as the implementation of the DISE hardware structures.

1.7 Overview

The rest of this dissertation is organized as follows. Chapter 2 presents related work. In particular, it shows how DISE fits into the context of existing translation mechanisms. It also discusses other hardware-based customization mechanisms. Finally, it also discusses other mechanisms with similar implementations as DISE, although they have much different functionality.

DISE mechanism. Chapters 3 and 4 present the DISE mechanism. Chapter 3 presents the architecture; it discusses the functionality, interface, and system architecture of DISE. In particular, how to specify transformations in DISE and how to program DISE with these specifications.

Chapter 4 looks at the DISE microarchitecture; it describes and evaluates one particular optimized implementation. This implementation has little performance impact on non-transformed programs. In addition, the latency of instruction macro-expansion (excluding the cost of executing the transformed instructions) is negligible. Finally, the evaluation shows the performance characteristics of a DISE processor as we vary several design parameters.

Utility of DISE. Chapters 5-7 show the utility of DISE in three different contexts. Chapter 5 discusses using DISE to implement interactive debugging watchpoints (using a transformation like that in Figure 1.1). In particular, DISE watchpoints are highly-efficient and flexible. DISE-based watchpoints eliminate costly unnecessary transitions to the debugger (*i.e.*, context switches), which in commercial debuggers can result in 40,000 times slowdowns, whereas DISE usually adds less than 25% overhead on most benchmarks. In addition, this chapter shows that DISE can watch arbitrarily complex expressions.

Chapter 6 explores the DISE mechanism as a dynamic code decompressor (using transformations like that in Figure 1.3(b)). It shows the use of DISE as an efficient dynamic code decompressor. It demonstrates code size reductions of over 35% using DISE. It also demonstrates the benefits of parameterized decompression and customized decompression dictionaries, which in both cases can improve compression by as much as 20%. Finally, it shows that DISE decompression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%.

Chapter 7 shows the utility of DISE in enhancing security and, in particular, its use in detecting two common types of attack: stack and pointer smashing. It shows that DISE-based attack detection is efficient with overheads usually less than 25%. DISE also separates the program from the attack detection code, which is simply a transformation specification. This separation makes it possible to customize attack detection as well as potentially adapt to new attacks (assuming DISE can be used to detect these new attacks).

The fact that DISE can effectively implement all of these transformations, demonstrates its utility as a general-purpose tool.

Finally, Chapter 8 gives some conclusions and discusses future directions of this work.

Chapter 2

Related Work

This chapter discusses related work. Section 2.1 discusses translation mechanisms and Section 2.2 discusses ad-hoc hardware widgets both of which can be used for customization. Section 2.3 discusses existing hardware-based macro-expansion mechanisms, the mechanism that DISE uses to transform programs, which have a similar implementation but a much different usage model. Finally, Section 2.4 summarizes these techniques, comparing and contrasting them with DISE.

Note that in Chapters 5-7, where we explore three uses of DISE, we discuss related work to other customization techniques. In Chapter 5 (page 86) we discuss related work in debugging, in Chapter 6 (page 123) we discuss related work in code compression, and in Chapter 7 (page 142) we discuss related work in enhancing software security.

2.1 Translation Mechanisms

There are a number of existing translation mechanisms for program transformation. The mechanisms are characterized by when and where they perform transformation: during compilation or post-compilation, statically or at runtime, in software or in hardware. Figure 2.1 shows a taxonomy of translation mechanisms. As described below, there are advantages and disadvantages to each approach.

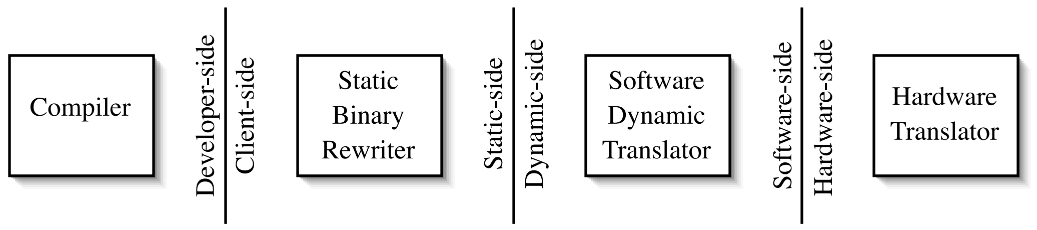


Figure 2.1: Taxonomy of translation mechanisms.

2.1.1 Compiler

The natural place to transform a program is within the compiler. Some existing compiler transformations include array bounds checking in C [48, 75], return address or pointer protection in C [18, 27, 28], and debugging watchpoints [89, 90]. The advantage to transforming within the compiler is that the compiler has access to the source code. However, the major disadvantage is that compilation is often done by the developer not by users and developers cannot anticipate the transformation needs of each user. Furthermore, compilation time is lengthy for some types of applications (*e.g.*, databases). Requiring users to recompile the program every time they want to customize it is not appropriate for many customizations (*e.g.*, adding/removing a debugging watchpoint). In addition, most compilers do not provide an API for customizing an application in any general way. Therefore, the compiler must be modified.

2.1.2 Static Binary Rewriter

Static binary rewriters are a client-side mechanism that take as input an executable program and output a transformed executable. They provide hooks for specifying application transformations. Transforming via a binary rewriter has several similarities with transforming via a compiler. First, both a binary rewriter and a compiler transform programs offline: no transformation is done during the runtime of the transformed program. Because transformation is offline, more elaborate time-consuming transformations are possible. But at the same time, binary rewriters (and compilers) cannot take advantage of runtime information. In addition, they do not transform shared library code nor can they transform JITs (Just-In-Time compiled programs).

The advantage of transforming via a binary rewriter rather than a compiler is that a binary rewriter is programmed by the end user not by the developer. The developer does not have to anticipate each user's transformation needs. Instead, each user can customize their application. Of course, this also means that the binary rewriter does not have access to source code. But many transformations can be performed on the machine code (*e.g.*, profiling, debugging).

Example binary rewriters include Atom [82], Etch [73], and EEL [57]. Demonstrated transformations using a binary rewriter include profiling [10], dynamic race detection [76], shared memory communication [77], debugging breakpoints/watchpoints [51], return address protection [11], and memory fault isolation [91].

2.1.3 Software Dynamic Translator

A third mechanism for customization is software dynamic translation. There are two types of software dynamic translators: *in-place* translators and *code-cache* translators.

In-place translators. In-place techniques, as the name suggests, maintain the original layout of the program, and patch it locally. They are dynamic in that they can transform the application as it runs. Because it is costly to insert a code sequence in the middle of a program at runtime, in-place techniques generally use *trampolining*. The translator replaces one or more instructions with a jump to the appropriate code sequence. When the code sequence finishes executing, it jumps to the next instruction in the original program. In-place techniques usually transform the code *eagerly* rather than *lazily*, meaning they translate all of the code at once rather than when the code is first executed. Code that is never executed is still translated. DynInst [46] is an example of an in-place dynamic translator.

Code-cache translators. In contrast, code-cache dynamic translators fully reconstruct the program layout. They translate the dynamically-executed parts of the program into a code cache, and execute them from there as opposed to the original binary image. As a result, users can formulate more complex transformations in a code cache translator, such as ISA conversions. In addition, code-cache techniques operate lazily. When an untranslated block of code is encountered, the application relinquishes control to the runtime system, which builds the appropriate block of code, loads it into the code cache, and returns control back to the application at the beginning of the newly translated block. Most code-cache techniques translate code at the granularity of dynamic traces. At a trace exit point, control is transferred back to the runtime system, which determines the next trace that needs to be executed, building it if necessary. To avoid this cost, most software dynamic translators link traces within the code cache, *i.e.,* transform trace exits from jumps to the runtime system into jumps to entries in the code cache. With good code reuse, the cost of translation is amortized over the total execution of the program. Under the right circumstances, performance can actually improve over that of the untransformed program; the cost of translation is outweighed by the high performance of the translated code.

Example code-cache dynamic translators include DELI [33], DynamoRIO [14], Strata [78], Valgrind [69] (Valgrind compiles basic blocks rather than traces and does not do any linking), and Pin [64]. Some transformation-specific code-cache translators include FX!32 [88], DAISY [35], and Transmeta's Crusoe [44]. These three translators convert one ISA to another in software. Dynamo [9] uses translation to dynamically optimize programs.

Like other software translators (compilers and binary rewriters), software dynamic translators (both in-place translators and code-cache translators) can perform arbitrary transformations. However, because transformation is performed at runtime, software dynamic translators are often limited to low-overhead transformations that do not require whole-program analyses. In general, the overhead is higher for a software dynamic translator because they transform at runtime (although in some cases, the software dynamic translator can optimize the program and amortize the cost of transformation). An important virtue of software dynamic translators is that they can exploit runtime information. The transformations are also late-binding, meaning the user can decide how to transform the program at runtime, rather than at some earlier point.

2.1.4 Hardware Translator

A final approach for transforming programs is via hardware. Hardware translators have a lower performance cost than software translators; they do not bloat the code nor is there a high fixed transformation cost (the cost of transformation excluding the cost of executing additional instructions). But they sacrifice functionality for performance. Many hardware expanders are application specific, and none are user programmable.

One example application-specific hardware translator is decoder-based, dynamic code decompression [59], which transforms a compressed program into a decompressed program. Tagged instructions in a compressed application are interpreted as dictionary indices and replaced by the corresponding entries.

Programmable microcode [17, 72] was an early choice for transformations like address tracing [2]. Although microcode remains a viable option for implementing complex instructions and programmable microcode stores persist (*e.g.*, Intel supports limited microcode patching to fix bugs in the field [43]), its current use is too sparse and irregular to effectively support transformation. Alpha's PAL [81] exposes hardware internals to privileged software, but is invoked using calls and traps, not matching and replacement, making it unsuitable for many transformations. Neither microcode nor Alpha PAL is user programmable.

2.2 Ad-Hoc Hardware Widget

In addition to translation mechanisms, there are also ad-hoc hardware widgets that can perform customization. These mechanisms do not transform programs but rather monitor or augment the running program within the processor. These are highly efficient, but not programmable and not general-purpose. The profiling processor [99] and instruction path co-processor [19] provide additional functionality—profiling and trace construction, respectively—at virtually no cost to the application using dedicated, potentially programmable, pipeline stages. To minimize performance impact, the dedicated stages are placed post retirement and thus can only monitor programs (*e.g.*, cannot do decompression).

Hardware debugging registers provided by some architectures (*e.g.*, x86, IA-64, PowerPC) are also a form of hardware customization. These registers allow interactive debuggers to implement watchpoints (as well as breakpoints, although more flexible techniques are generally used for breakpoints). The debugger loads these registers with the memory address of each watched variable. If any of these addresses are stored to, then the processor traps and control is transferred to the debugger. However, these registers are not a flexible approach to implementing watchpoints. First, they are limited in number (*e.g.*, 4). They also have no support for conditional watchpoints (*i.e.*, watchpoints that only fire based on a user-specified predicate). In Chapter 5, we show that this inflexibility forces interactive debuggers in many circumstances to resort to more inefficient techniques, which can lead to slowdowns greater than 40,000 times. These registers are also debugging specific.

Finally, many security enhancing hardware customizers has been proposed [29, 55, 49, 65, 83, 87, 95, 97]. These prevent various forms of software attacks such as stack and pointer smashing attacks. Like the mechanisms discussed above, these techniques are all application specific.

2.3 Decoder-Based Macro Expansion

An additional related body of work is in hardware translation mechanisms. IA32 processors [39, 41, 42] dynamically macro-expand each CISC instruction into one or more internal RISC operations, potentially caching the translations [39]. Dynamic Instruction Formatting [66] schedules cached instructions into VLIW groups. Speculative Decode [52] implements microarchitectural execution time optimizations like silent-store elimination using alternate expansions. These facilities resemble DISE mechanically, but differ in two major ways. First, they translate the ISA to a simpler form for the purpose of reducing execution complexity. DISE adds instructions in order to add functionality. Second, they are inaccessible to software, and thus capable only of changing/optimizing representations. To add functionality, DISE has an API.

2.4 Summary

We have discussed a number of related mechanisms to DISE. Here we compare and contrast these mechanisms. As shown in Table 2.1, we focus on six functionality attributes including whether the mechanism can (i) monitor a program, (ii) transform a program, (iii) transform a program using peephole transformation, (iv) transform a program using global

		Compiler	Binary rewriter	Software dynamic translator	Hardware translator	Ad hoc hardware widget	Decoder -based expander	DISE
Functionality attributes	Program monitoring	Yes	Yes	Yes	Yes	Yes	No	Yes
	Program transformation	Yes	Yes	Yes	Yes	No	No	Yes
	Peephole transformation	Yes	Yes	Yes	Yes	No	No	Yes
	Global transformation	Yes	Yes	Yes	No	No	No	No
	Flexible/ programmable	Yes	Yes	Yes	Yes	No	No	Yes
	Programmable by a user	No	Yes	Yes	No	No	No	Yes
Performance attributes	Fixed transformation cost	High*	High*	High	Low	None	Low	Low
	Static instruction footprint overhead	High	High	High	None	None	None	None
	Dynamic instruction footprint overhead	High	High	High	High	None	High	High

Table 2.1: Summary of related work. * Note that although compilers and binary rewriters have a fixed transformation cost, it is performed statically (*i.e.*, offline).

transformation, (v) whether the mechanism is flexible and programmable, and finally, (vi), whether the mechanism is programmable by users (rather than just designers of the mechanism and/or privileged software). We also look at three performance attributes. These attributes make up the total cost of transforming a program. They include (i) the fixed cost of transformation, (ii) the static instruction footprint overhead, and (iii) the dynamic instruction footprint overhead.

Software techniques. Software techniques (compilers, binary rewriters, software dynamic translators) can perform any kind of transformation including both peephole and global transformations. Software techniques can also be used for performance monitoring (via transformation). Software techniques are highly flexible. In terms of programmability, binary rewriters and software dynamic translators are programmable by end users (as well as by designers), while compilers are only programmable by the compiler designer.

A user would have to modify a compiler to perform some arbitrary transformation, which is beyond the capabilities of most users. Furthermore, the source code for many compilers is not available to end users.

On the performance side, software techniques have high transformation overheads. First, there is a high transformation cost (note that compilers and binary rewriters transform statically). Second, software techniques have a high static instruction footprint overhead, *i.e.*, they bloat the code. Finally, software techniques also have a high dynamic instruction footprint overhead, *i.e.*, any inserted instructions must be executed and thus, reduce instruction bandwidth.

Hardware techniques. In contrast to software techniques, hardware techniques have good performance, but less functionality. The only cost of hardware expanders (*e.g.*, microcode, Alpha PAL) is reduced instruction bandwidth, however, they are not programmable by users, but rather by processor vendors (microcode) and privileged software (Alpha PAL). Ad-hoc hardware widgets have no transformation costs, however, these are not programmable at all. Furthermore, they can not transform a program, but only monitor it. Existing decoder-based macro-expanders, which have other uses such as CISC-to-RISC translation, are not appropriate for program customization.

DISE. DISE is a hybrid software and hardware mechanism that combines most of the functionality benefits of software techniques with most of the performance benefits of hardware techniques. DISE is programmable by users and can transform programs using peephole transformation. Because it does not change the program image, it is also a nice program monitor. Like existing hardware expanders, DISE has only one transformation cost: dynamic instruction footprint overhead. Moreover, as we show in later chapters, this overhead is small for most transformations and benchmarks and much less than the overhead of software techniques, which also have a static instruction footprint cost.

Chapter 3

DISE Architecture

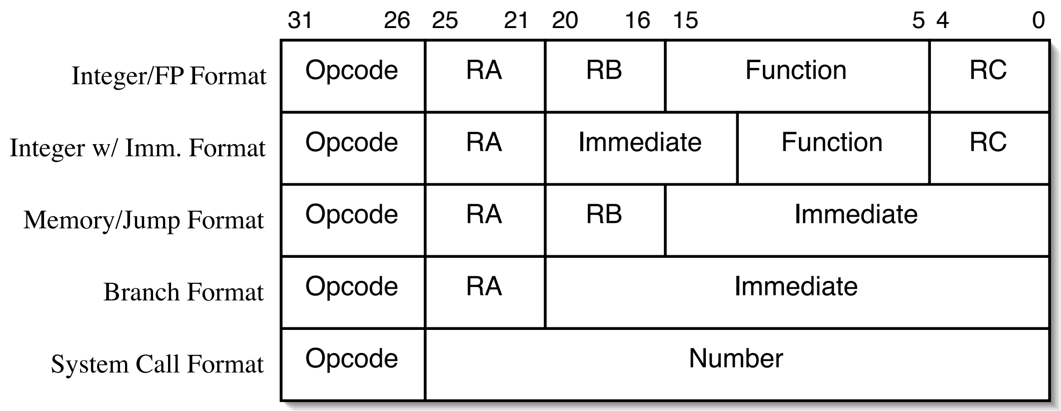
This chapter presents the DISE architecture [22]. Section 3.1 describes the functionality of DISE, and Section 3.2 describes the interface and system architecture.

Although DISE is mainly ISA independent, we describe the implementation of DISE on the Alpha ISA [81]. All assembly code in this dissertation uses the Alpha ISA. Figure 3.1 summarizes aspects of the Alpha ISA that are particularly relevant to this dissertation. Figure 3.1(a) shows the five encoding formats of Alpha instructions. The instructions are divided up into *fields*, *e.g.*, opcode, immediate, and register identifiers (*i.e.*, **RA**, **RB**, **RC**). Figure 3.1(b) describes how each register field (*i.e.*, **RA**, **RB**, and **RC**) is used within various instructions. Figure 3.1(c) lists some commonly-used instructions in the Alpha ISA. Although not shown in Figure 3.1, register names in Alpha are prefixed with '$' (*e.g.*, $2), while immediates have no prefix (*e.g.*, -64).

3.1 Functionality

This section gives a user's perspective of the DISE mechanism. The DISE *engine* for transforming programs operates by performing instruction macro-expansion. It takes as input an individual instruction and outputs an instruction sequence based on the inputted instruction. DISE inspects every dynamic instruction (prior to execution) and macro-expands some of the instructions based on user-specified rules.

Alpha Instruction Encodings



(a)

Alpha Register Field Uses

RA	Source in Integer/FP (w/ or w/o immediate), source or destination in stores and loads, return address in indirect jumps (calls/returns), branch test
RB	Integer/FP source (w/o immediate), base address in stores and loads
RC	Target in Integer/FP (w/ or w/o immediate)

(b)

Commonly-Used Alpha Instructions

Opcode	Description
jsr/ bsr	Call a subroutine
ret	Return from a subroutine
beq/ bne	Branch if test register is (is not) 0
cmpeq	Compare 2 registers, if same put 1 in target, otherwise put 0
addq/ subq	Add (subtract) two quad operands, put result in target
and/ bis	And (or) two quad operands, put result in target
srl	Shift first operand by second operand, put result in target
ldq/ stq	Load (store) quadword from (to) memory
lda	Load target with base address plus immediate

(c)

Figure 3.1: Alpha ISA: (a) instruction encodings, (b) register field uses, and (c) commonly-used instructions.

Instruction macro-expansion consists of two logically-separate components: instruction matching and instruction replacement. The matching component is responsible for identifying instructions that need to be transformed (*i.e.*, macro-expanded). A matched instruction is called a *trigger* because it triggers transformation. The replacement component is responsible for replacing trigger instructions with a user-specified instruction sequence. Likewise, a DISE user-specified rule, called a *transformation specification*, consists of two parts: a *pattern* for matching instructions and a *replacement sequence* that DISE uses to replace the matched instruction. We discuss both instruction matching and instruction replacement below.

3.1.1 Instruction Matching

To match instructions, users specify a pattern, which is defined on the bits of the instruction. It is any combination of opcode, opcode class (*e.g.*, all memory instructions), register names, immediate, or immediate sign. For example, users can specify patterns such as "loads that use the stack pointer as their address register" or "conditional branches with negative offsets."

A pattern cannot refer to any dynamic property of an instruction such as the value in one its register operands (this restriction simplifies the microarchitecture of DISE). Furthermore, the pattern cannot refer to the instruction's program counter (PC). Because we have not found any compelling uses of PC matching, we leave it out of the current design (including it would increase the size of the pattern representation presented in Section 3.2).

To specify a pattern (*i.e.*, *pattern specification*), the user supplies a list of predicates separated by '&&'. The pattern only matches an instruction if all the predicates are true for that particular instruction. A predicate has the form **T.X==Y** where **T** refers to the instruction DISE is matching, **X** is an attribute of **T** (*e.g.*, opcode), and **Y** is a legal value of **X** (*e.g.*, **stq**).

Table 3.1(a) lists the attributes that can appear in a predicate. Attributes are defined on the assembly instruction not the machine instruction. For instance, **T.OPCODE** refers to the assembly opcode of the instruction (*e.g.*, **bis**) not the 6-bit machine instruction opcode (*e.g.*, **010001**).

Figure 3.2 shows a DISE transformation for store address tracing, which we use as a

Instruction Attributes for Matching

Attribute	Description	Examples
T.OPCODE	Instruction's opcode	ldq, jsr
T.OPCLASS	Instruction's opcode class	load, call
T.RA	Instruction's RA register	\$5, \$21
T.RB	Instruction's RB register	\$5, \$21
T.RC	Instruction's RC register	\$5, \$21
T.IMM	Instruction's immediate	64, -48
T.IMMSIGN	Instruction's imm. sign	+, -

(a)

Example Opclasses on Alpha

Classes	Example Opcodes
Memory	ldq, stq, ldbu, stbu
Load	ldq, ldbu
Store	stq, stbu
Control flow	beq, bne, jmp, bsr jsr, ret
Branch	beq, bne
Indirect jump	jmp, jsr, ret
Call/return	bsr, jsr, ret
Call	bsr, jsr
Return	ret
ALU	addq, subq, srl, and, bis
All	addq, srl, jsr, bne, stq

(b)

Table 3.1: Matching attributes: (a) the instruction attributes that DISE uses in matching, and (b) some examples of the opclass attribute on the Alpha.

running example throughout this section. In store address tracing, the memory address of each executed store in the application is logged. Figure 3.2(a) shows a transformation on one particular store. The store is transformed into a code sequence that logs the address and PC before performing the original store. The sequence also increments the log pointer, checks if the log is full, and if so, calls a routine (*e.g.*, **writedisk**) to write the log to disk.

Figure 3.2(b) shows the DISE transformation specification for store address tracing. The format of a transformation specification is as follows: **pattern => replacement sequence** (ignore the replacement sequence for the moment). The pattern in Figure 3.2(b), **T.OPCLASS==store**, matches on all classes of store instructions, which in Alpha includes **stq**, **stl**, **stw**, **stb**, *etc.* There are many opcode classes, including all control flow instructions and all memory instructions. They are architecture dependent and defined by the processor vendor.

Table 3.1(b) lists the opcode classes used in this dissertation, as well as a few other

```
                 lda $d0,16($2)              # compute address
                 stq $d0,0($d1)              # log address
                 stq T.PC,8($d1)             # log PC
                 addq $d1,16,$d1             # increment log pointer
stq $1,16($2) →  cmpeq $d1,$d2,$d0           # is log full?
                 d_beq $d0,1                 # yes, branch
                 d_callne writedisk,$d0      # no, call writedisk
                 stq $1,16($2)               # perform original store
```

(a)

```
# Pattern (match on all stores)
T.OPCLASS == store
=> # Replacement sequence
   lda $d0,T.IMM(T.RB)
   stq $d0,0($d1)
   stq T.PC,8($d1)
   addq $d1,16,$d1
   cmpeq $d1,$d2,$d0
   d_beq $d0,1
   d_callne $d3,$d0
   T.INST
```

(b)

Figure 3.2: DISE transformation for store address tracing: (a) transformation for one particular store and (b) the transformation specification in DISE.

desirable classes. Although not shown in Table 3.1(b), many of the classes could be broken down further based on data size. For example, there could be a class for loads and stores of bytes. DISE processor vendors might also want to include subclasses of the ALU class (*e.g.*, arithmetic instructions, shift instructions) and classes of floating point instructions.

Two usage modes. DISE has two usage modes: *transparent* and *aware*. A transparent transformation operates on unmodified executables using specifications that match "naturally occurring" instructions. In this mode, DISE changes the semantics of existing instructions. Store address tracing is an example of a transparent transformation. An aware transformation operates on modified applications into which specially-crafted DISE codewords (instructions that do not occur naturally) have been planted. An aware application—one with codewords planted in it—will only run on a DISE processor.

Dynamic code decompression (discussed in Chapter 6) is an example of an aware

```
T.OP == res2 &&       # match reserved op
T.IMM == 0            # and immediate of 0
=> addq $4,8,$4       # Compressed
   ldq $5,0($4)       # sequence
   cmpeq $6,$5,$3

T.OP == res2 &&       # match reserved op
T.IMM == 1            # and immediate of 1
=> and $7,8,$7        # Compressed
   cmpeq $5,$7,$6     # sequence
```

Figure 3.3: Aware transformation specifications for decompression.

transformation. In decompression, the program image contains codewords that were used to compress frequently-occurring instruction sequences. DISE macro-expands these codewords back into the original instruction sequence at runtime (*i.e.*, decompressing them). Figure 3.3 shows two specifications for DISE decompression. Both specifications match instructions with a reserved opcode **res2** (*i.e.*, codewords). They match on different immediates, depending on instruction sequence the codeword replaced.

3.1.2 Instruction Replacement

To replace a matched instruction, a user defines a replacement sequence (in addition to a pattern). A replacement sequence is simply a sequence of instructions (with a few differences, discussed below). In store address tracing (Figure 3.2(b)), the replacement sequence has 8 instructions. Replacement instructions have a few differences with conventional instructions, which make it easier to program DISE.

Parameterized fields. First, replacement instructions are *parameterized* with respect to the trigger, *i.e.*, they can use fields from the trigger instruction (see Figure 3.1(a) for a list of Alpha fields). A field in the replacement instruction can reference a field in the trigger instruction using **T.**<**field name**> (*e.g.*, **T.OPCODE**). Table 3.2 lists all the fields in the trigger which replacement instructions can reference, many of which are also used in matching (*e.g.*, **T.RA**).

In store address tracing (Figure 3.2(b)), the first, third, and last instructions in the

Field	Description	Examples
T.OPCODE	Instruction's opcode	stq, jsr
T.RA	Instruction's RA register	$5, $21
T.RB	Instruction's RB register	$5, $21
T.RC	Instruction's RC register	$5, $21
T.IMM	Instruction's immediate	64, -256
T.PC	Instruction's PC	0x0011e4bf
T.INST	Entire instruction	stq $1,0($2)

Table 3.2: Parameter types. The instruction fields which replacement instructions can reference.

replacement sequence are parameterized. The first instruction in the replacement sequence computes the memory address of the matched store, using the sum of the base register and immediate field. In the first instruction in Figure 3.2(b), **T.RB** refers to the trigger instruction's base register and **T.IMM** refers to the trigger instruction's immediate. The third instruction is also parameterized. It references the instruction's program counter (**T.PC**). Finally, the last instruction in the sequence is parameterized (*i.e.*, **T.INST**); it uses all fields from the trigger instruction. The last replacement instruction is simply the trigger instruction.

Essentially, parameterization allows users to make more efficient use of a transformation. Without parameterization, users would have to specify transformations for each instance of the matched instruction (*e.g.*, **stq $1,0($2)**, **stq $1,4($3)**, *etc.*).

DISE registers. DISE also provides 16 dedicated registers (**$d0-$d15**) that are only usable by replacement instructions (there is one exception described below). These registers provide scratch space for replacement instructions; DISE users do not to have scavenge registers from the application. The transformation specification for store address tracing, shown in Figure 3.2(b), uses two DISE registers: **$d0**, **$d1**, and **$d2**.

DISE registers have an additional benefit beyond providing temporary space for saving scratch values (*e.g.*, **$d0**). They allow users to synthesize global behavior. Because the

DISE engine uses instruction macro expansion, DISE is limited to *peephole* transformations. It transforms programs one instruction at a time. However, DISE registers can be used to synthesize global behavior by passing values from one transformation to another. In Figure 3.2(b), **$d1** is used in this manner. Each transformation updates the log pointer; the new value of the log pointer is passed to a subsequent transformation through **$d1**.

Two instructions are available for moving values between the DISE registers and the conventional registers. The instruction **d_mtdr** moves a value from a conventional register to a DISE register and the instruction **d_mfdr** moves a value from a DISE register to a conventional register (these are analogous to **mtc0** and **mfc0** in MIPS [37]).

DISE control flow. Replacement sequences can also contain control flow. This control flow works as one would expect, allowing users to branch within a replacement sequence, jump to other instructions in the application, and call routines (calls have slightly different semantics than other DISE control flow and are discussed later).

Control flow within a replacement sequence is facilitated using an additional DISE state element called the *DISEPC*. The address of an instruction (DISE or non-DISE) is given by the pair $\langle$PC:DISEPC$\rangle$. A (dynamic) replacement instruction has the same PC as the trigger instruction, but its DISEPC is equal to the index within the replacement sequence. Non-replacement-code has a DISEPC of 0.

There are two kinds of control transfers that can appear within a replacement sequence (excluding DISE calls): *intra-instruction* and *inter-instruction*. As the name implies, intra-instruction control flow transfers control to another instruction within the replacement sequence (*i.e.*, within the trigger instruction). Inter-instruction control flow transfers control to another instruction outside of the replacement sequence (*i.e.*, to another instruction in the application). The target instruction must be within the application and not within another replacement sequence. DISE does not support jumps from one replacement sequence to the middle of another replacement sequence. This restriction preserves the abstraction that expansions are self-contained within individual instructions, making it simpler to program and reason about DISE transformations.

Intra-instruction control flow allow users to branch within a replacement sequence. These branches (*i.e.*, **d_beq** and **d_bne**) work similarly to conventional branches, however, the offset is with respect to the DISEPC rather than the PC. Naturally, offsets are not allowed to go past the beginning or the end of the sequence. Figure 3.2(b), the replacement sequence contains an intra-instruction branch (**d_beq**).

Inter-instruction control are simply conventional Alpha control flow instructions. These include both branches and jumps. They allows users to transfer control to another instruction in the program image, *i.e.*, to address PC+offset.

The DISEPC is also used in transfers of control due to exceptions. Precise state is defined at each ⟨PC:DISEPC⟩ boundary. If an instruction is interrupted, the operating system saves both the PC and DISEPC, and uses both of these state elements to restart the program at the appropriate point (which may be within a replacement sequence).

DISE calls. DISE replacement sequences can also contain function calls. A DISE function call at ⟨PC:DISEPC⟩ to ⟨newPC:0⟩ saves the return address ⟨PC:DISEPC+1⟩. The called function is composed of conventional instructions (*i.e.*, non-DISE code), which are fetched from instruction memory. To access values in DISE registers, the called function can use the instructions **d_mtdr** and **d_mfdr**. The function returns to ⟨PC:DISEPC+1⟩, *i.e.*, the replacement instruction following the call. DISE itself is disabled (automatically via the call) within the body of a function called from within a replacement sequence. This again preserves the notion that replacement sequences are self-contained within the application instructions and prevents bottomless recursion of DISE expansions.

A special DISE call instruction, **d_call**, is used to call functions (which simplifies the microarchitecture, as discussed in the next chapter). A conventional call (*e.g.*, **bsr** or **jsr**) will not disable DISE, and the corresponding return will not transfer control to the middle of the replacement sequence. The function, itself, must also use a special DISE return, **d_ret**, to return back to the replacement instruction following the call and to re-enable expansion. DISE returns are only legal within a DISE-called function.

A DISE call (as well as a conditional call) is an indirect jump (similar to **jsr**). The address of the called function is placed in a DISE register (Section 3.2 describes how this register is initialized), which is passed as an operand to the instruction. Because distinct static instructions with different PCs can expand to the same replacement sequence, there are no direct jump DISE calls (*i.e.*, the offset would need to be different at each expansion). Although DISE call instructions are indirect jumps, for clarity, we sometimes use the name of the function in place of the register operand (*e.g.*, **d_call foo** as opposed to **d_call $d5**).

```
T.OPCLASS == store            # match on stores
=> lda $d0,T.IMM(T.RB)        # compute address
   stq $d0,0($d1)             # log address
   stq T.PC,8($d1)            # log PC
   addq $d1,16,$d1            # increment log ptr.
   cmpeq $d1,$d2,$d0          # is log full?
   d_ccallne writedisk,$d0    # yes, write to disk
   T.INST                     # perform store
```

Figure 3.4: DISE transformation with a conditional call.

In Figure 3.2(b), the replacement sequence contains a DISE call (**d_call**) to **writedisk**, which writes the log to disk and empties the log in memory.

DISE conditional calls. As discussed further later, we introduce a conditional call instruction to reduce branch overheads. As shown in Figure 3.4, the instruction **d_ccallne** calls **writedisk** only if **$d0** is not equal to zero. Similarly, there is also a conditional call, **d_ccalleq**, which calls a routine only if its second operand is equal to zero.

3.2 Interface and System Architecture

This section describes the interface and system architecture to DISE. Although users will probably write transformations in the format described in the previous section (*e.g.*, Figure 3.2(b)), DISE, which is a hardware mechanism, is naturally programmed at a lower level. This section first describes that low-level interface. It then discusses how specifications written in the higher-level format are translated into the low-level representation. Finally, it concludes by looking at the ISA extensions and operating system support necessary in a DISE system.

3.2.1 Representing Patterns

Patterns and replacement sequences have a binary representation. Below we discuss the representation of a pattern specification (in the next subsection we look at the representation of a replacement sequence).

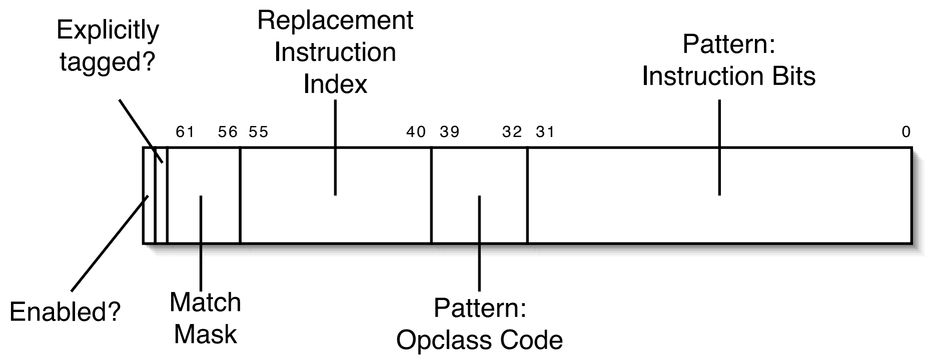


Figure 3.5: The binary representation of a pattern specification.

A pattern specification, shown in Figure 3.5, of course, contains a pattern. The pattern is split into subpatterns, which include opcode, register identifiers (**RA**, **RB**, and **RC**), immediate, sign of the immediate, and opclass. A six bit match mask indicates which subpatterns are to be used in matching instructions. If the **RA** bit in the match mask is set to 1, then the **RA** pattern is used to match instructions. Otherwise, the **RA** field is ignored when matching instructions.

The subpatterns for opcode, register identifiers, immediate, and immediate sign are wrapped into an instruction ("Pattern: Instruction Bits" in Figure 3.5). To match on any of these fields, users construct a dummy instruction with any fields that are to be used for matching set appropriately. The values of the other fields are ignored (hence the term dummy instruction). The reason for using an instruction to represent multiple subpatterns is to reduce the pattern encoding size. A machine instruction is more densely encoded than separately representing every field that can appear in any instruction in the ISA.

The opclass subpattern is represented separately ("Pattern: Opclass Code") because it is not encoded within an (Alpha) instruction. An opclass is represented using a 8-bit code, which is defined by the processor vendor. An opclass code of 0 is illegal (below, we discuss the reason for this).

In addition to the pattern, a pattern specification contains several other fields. First, it contains an enabled flag. It also contains a replacement instruction index, which specifies the replacement sequence that will be expanded on a match (unless, as described below, the index is taken from the trigger instruction). The instructions in a replacement sequence, which are stored in memory, are referenced using a 64-bit address. To avoid storing the entire 64-bit address in every specification, we instead store the base address in a DISE register, **$drsbase**, and a 18-bit offset (the remaining bits in the quadword) in each pattern specification. On a match, DISE computes the address of the first instruction in the replacement sequence by shifting the offset by 3 (since replacement instructions are quad-aligned) and adding it to the base register. **$drsbase** is read and written using the instructions **d_mfdr** and **d_mtdr**.

Storing the base address in a DISE register reduces the size of the pattern specification, but it also limits the total number of replacement sequences. If the index is 18 bits long, the total size of the replacement sequence space is 256K replacement instructions. This size limit is not a problem for the transformations used in this dissertation. Alternatively, instead of using a replacement instruction index, we could have used a replacement sequence index. Indexing by sequence increases the total number of replacement sequences we can reference (unless each sequence contains only one instruction, in which case, the two approaches are equivalent). But because replacement sequences have variable lengths, this approach would require a level of indirection, *i.e.*, a table of replacement sequence pointers or fixed-size replacement sequence entries. The former case would require an additional load to access a replacement sequence. The latter case would suffer from internal fragmentation.

The final field in a specification is a flag for *explicit tagging*. Explicit tagging allows users to encode the replacement sequence index (called a tag) within the trigger instruction. For the microarchitectural implementation presented in Chapter 4 that caches patterns for fast access, too many patterns (*e.g.*, more than 16) results in performance loss. With explicit tagging, a single pattern can map to multiple replacement sequences. Explicit tagging is generally applicable only for aware transformations, where the trigger instructions are planted specially for DISE expansion. In this case, we can encode the tag within the inserted instructions. As discussed in Chapter 6, explicit tagging is important for DISE dynamic code decompression (an aware transformation). In Chapter 6, we present explicit tagging in greater detail.

Specifications table. A memory-resident *specifications table*, shown in Figure 3.6, holds the pattern specifications. The location in memory of this table is stored in a special DISE

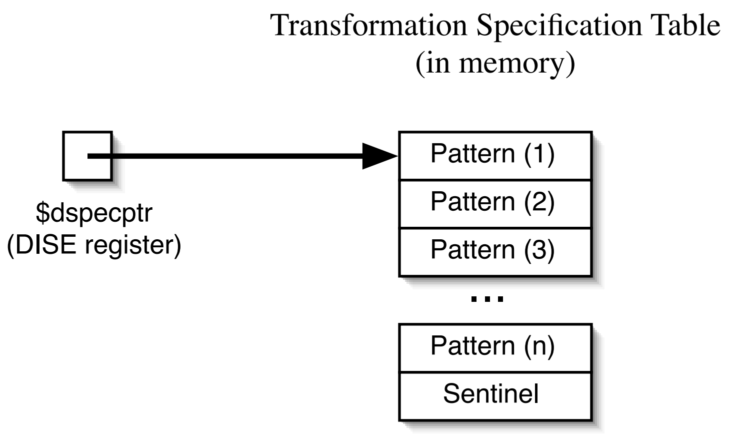


Figure 3.6: The DISE specifications table and special registers.

register, **$dspecptr**. Like **$drsbase**, this register is manipulated using **d_mtdr** and **d_mfdr**. The table must be a contiguous list of patterns, although patterns within the table can be disabled. A sentinel, which is a null entry (a 0 quadword), marks the end of the list. Because the opclass encoding in a pattern specification cannot be 0, a legal pattern can never be confused for the sentinel.

The order of the addresses within the table is not arbitrary. It determines the priority of the patterns. Patterns at lower addresses have higher priority than those at higher addresses. When multiple patterns match on an instruction, the pattern with the higher priority along with its corresponding replacement sequence are selected for expansion. Users must keep the priority of patterns in mind when modifying the specification table.

3.2.2 Representing Replacement Sequences

A replacement sequence is logically a list of *template instructions* and *directives*. A template instruction is an instruction that may contain parameterized fields (*i.e.*, references to fields in the trigger instruction) as well as DISE opcodes or operands (*i.e.*, DISE registers). Each field in a template instruction has a corresponding directive that indicates whether that field is not applicable (*e.g.*, **RC** in a **ldq**), literal (*i.e.*, bits are interpreted literally), a DISE register (only applicable for register fields), a DISE opcode (only applicable for the opcode field), or parameterized. When the directive indicates that the field is a DISE

Transformation Specification

```
T.OPCLASS == store
=> lda $d0,T.IMM(T.RB)
   stq $d0,0($d1)
   stq T.PC,8($d1)
   addq $d1,16,$d1
   cmpeq $d1,$d2,$d0
   d_ccallne $d3,$d0
   T.INST
```

(a)

Replacement Sequence

	Directives				
Template Instructions	Opcode	RA	RB	RC	IMM
lda $d0,T.IMM(T.RB)	Literal	DISE Reg.	Parameter	N/A	Parameter
stq $d0,0($d1)	Literal	DISE Reg.	DISE Reg.	N/A	Literal
stq T.PC,8($d1)	Literal	Parameter	DISE Reg.	N/A	Literal
addq $d1,16,$d1	Literal	DISE Reg.	DISE Reg.	N/A	Literal
cmpeq $d1,$d2,$d0	Literal	DISE Reg.	DISE Reg.	DISE Reg.	N/A
d_ccallne $d3,$d0	DISE Op.	DISE Reg.	DISE Reg.	N/A	N/A
T.OP T.RA,T.IMM(T.RB)	Parameter	Parameter	Parameter	N/A	Parameter

(b)

Figure 3.7: The logical representation of a replacement sequence for store address tracing: (a) high-level format and (b) logical replacement sequence representation.

register or DISE opcode then the corresponding bits in the template instruction are interpreted as a DISE register or DISE opcode, respectively. When the directive indicates that the field is parameterized, then the corresponding bits in the template instruction specify which field in the trigger will replace the field in the template instruction. Notice that only three bits are necessary to indicate the parameterized field (*i.e.*, **T.OPCODE**, **T.RA**, **T.RB**, **T.RC**, **T.IMM**, and **T.PC**), and each field in the template instruction has more than 3 bits (see Figure 3.1(a)).

Example. Figure 3.7 makes this discussion concrete by showing an example for store address tracing. Figure 3.7(a) shows the high-level, assembly-formatted transformation specification and Figure 3.7(b) shows the logical representation of a replacement sequence. In store address tracing, the first, third, and last instructions are parameterized. The first replacement instruction (**lda**) computes the address of the store by adding the store's base register (**RB**) to the store's immediate field (**IMM**). Therefore, the directive for the first replacement instruction's **RB** field specifies that it is parameterized and the **RB** field in the template instruction specifies that is **T.RB**; likewise for the immediate field. In both cases, the parameterized field refers to the same field in the trigger instruction, but it could also refer to a different field in the trigger (*e.g.*, the **RB** field in the replacement instruction parameterized with **T.RA**). The third template instruction is also parameterized. It stores the instruction's address to the log. Therefore, the **RA** directive is parameterized and the bits in the template instruction encode **T.PC**. Finally, the last template instruction is parameterized. It uses all the fields in the trigger instruction including **T.OPCODE**, **T.RA**, **T.RB**, and **T.IMM**. Note that **RC** is not used in a store instruction.

In addition, many of the replacement instructions use DISE registers. For example, the first instruction uses register **$d0** in the **RB** field. In this case, the directive indicates that **RB** is a DISE register and the **RB** field in the template instruction is 0, which is interpreted as **$d0**. The sixth replacement instruction uses the DISE opcode **d_ccallne**. The opcode directive indicates that it uses a DISE opcode and the opcode field within the template instruction is interpreted as a DISE opcode rather than a standard Alpha opcode. Essentially, directives give replacement instructions a wider encoding, allowing them to refer to new opcodes and operands, unlike conventional instructions which are limited due to their dense encoding.

Binary representation. The binary representation of a replacement sequence, shown in Figure 3.8, is a list of replacement instructions. Each replacement instruction has three components (two of which were discussed above): a template instruction, a set of directives, and a last flag indicating whether the replacement instruction is the last instruction in the sequence. The template instruction is simply a 32-bit instruction. The encoding and layout of the directives is shown in Figure 3.9. In total, the directives require 15 bits, but we dedicate 31 bits for them to quad-align replacement instructions (*i.e.*, 16 bits are unused). The directives for the register fields and the opcode field require 2 bits (3 possible

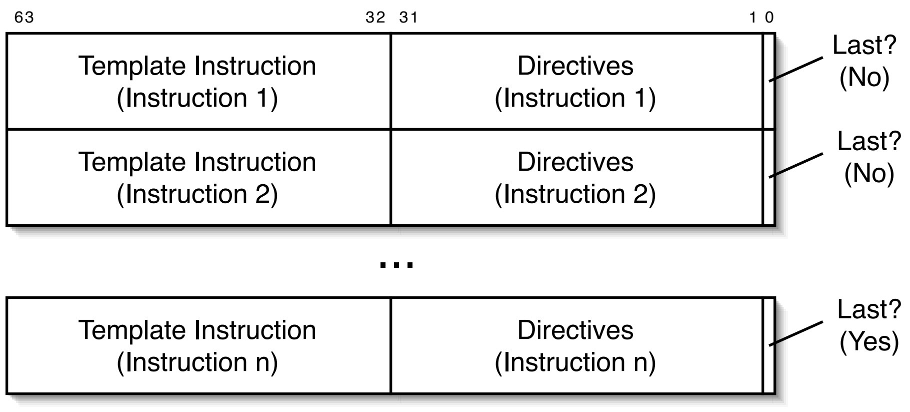

63
32
31
1
0
Template Instruction (Instruction 1)
Directives (Instruction 1)
Last? (No)
Template Instruction (Instruction 2)
Directives (Instruction 2)
Last? (No)
...
Template Instruction (Instruction n)
Directives (Instruction n)
Last? (Yes)

Figure 3.8: The binary representation of a replacement sequence specification.

states). The immediate field requires only 1 bit (2 possible states), however, we dedicate 7 bits for it. In Chapter 6 we discuss some uses of these additional bits in the context of decompression.

3.2.3 Adding/Removing Specifications

To add or remove specifications, a user manipulates the specifications table (described above). To add a specification, the user adds the pattern specification to the specifications table and places the corresponding replacement sequence in the replacement sequence space, updating the pattern's replacement sequence index, accordingly. To remove a specification, the user removes the pattern specification from the specifications table (the replacement sequence specification does not have to be explicitly removed).

The suggested technique for adding or removing specifications is to disable DISE while manipulating the table. DISE is disabled using the instruction **d_toggle** with an immediate value of 0 (non-zero immediate values cause DISE to be enabled). From our experiences with DISE, this approach is usually adequate. In most cases, it is not necessary to transform the code that programs DISE, because this code is usually not considered part of the main application.

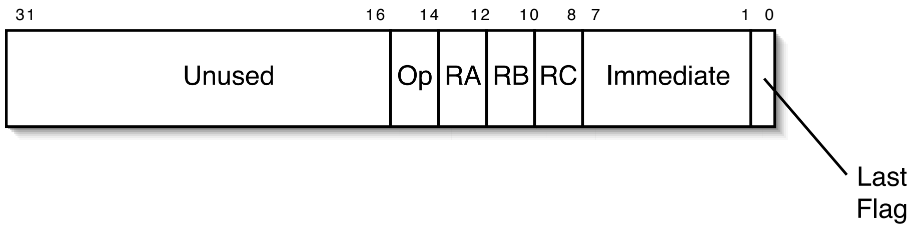
Depending on the implementation of DISE, updating the table in memory may not instantaneously update the DISE engine. The implementation described in Chapter 4,

Directives Encoding

Opcode	RA/RB/RC	Immediate
Literal - 0	Literal - 0	Literal - 0
Parameterized - 1	Parameterized - 1	Parameterized - 1
DISE Opcode - 2	DISE register - 2	

(a)

Directives Layout



(b)

Figure 3.9: Directives: (a) encoding and (b) layout.

which uses caching, is one such example. When a user modifies a specification in memory, which is also cached in a DISE hardware structure, the cached version is not instantaneously updated. Until the cached copy is evicted, the program is transformed using the old specification. To force the DISE hardware to synchronize with the specifications in memory, users can execute the instruction **d_sync**. Users should execute this instruction, before DISE is re-enabled.

If disabling DISE is not desirable, then users should create a new table and update **$dspecptr** rather than modify the table directly. Otherwise, there may be synchronization problems. If the table is modified in place, then the DISE structures may be updated anytime between the execution of a store to the table and a **d_sync**. In some cases, this ambiguity can cause problems. Consider swapping two entries in the table in order to interchange their priorities. This operation cannot be done atomically, but rather in two steps: one to overwrite the first pattern with the second and one to overwrite the second pattern with the first. If the table is modified in place, DISE could update its caches between the first and second step. For a short time (*i.e.*, until a **d_sync** is executed), DISE will not use one pattern specification. To avoid this problem, the user can create a new table with the entries swapped and update **$dspecptr** using **d_mtdr**. The user can then perform a **d_sync** to guarantee that all subsequent code is transformed using the updated specifications. This technique is not expensive, in general, because the specifications table is small. We have found that there are not that many patterns for most transformations (in this dissertation, no more than 10 patterns are used for any transformation).

A worse result can occur if replacement instructions are modified in place while DISE is enabled. It is possible that after modifying part of the replacement sequence, a miss in the replacement sequence cache will result in the partially-modified sequence being brought into the cache. For at least a short time, DISE will use a faulty replacement sequence. To avoid this problem, users should either disable DISE or construct a new set of replacement sequences, "off to the side," update **$drsbase**, and perform a **d_sync**.

3.2.4 Constructing Specifications

We have described the interface to DISE, both at a high level in Section 3.1 and at a low level in this section. Below we discuss how these two levels are related. We also discuss how users initialize DISE registers, and allocate and initialize memory regions used by DISE code. Finally, we discuss some errors that can arise when constructing specifications.

Translating specifications. Users do not need to construct specifications at the low level, but rather can write specification using the high-level format from the previous section (*e.g.*, Figure 3.2(b)). These are translated to the low-level format in a process similar to compiling conventional (non-DISE) assembly programs.

Figure 3.10(a) shows the compilation process for DISE specifications and auxiliary code (*e.g.*, DISE-called routines). First, a specifications translator converts the high-level, assembly specifications into the low-level ones, which it stores in an object (.o) file. Any auxiliary code must also be compiled into object files. This compilation can be done using a conventional assembler. Then these object files are linked into one binary (*i.e.*, *DISE binary*). The code within the DISE binary cannot have any external references, *i.e.*, it can

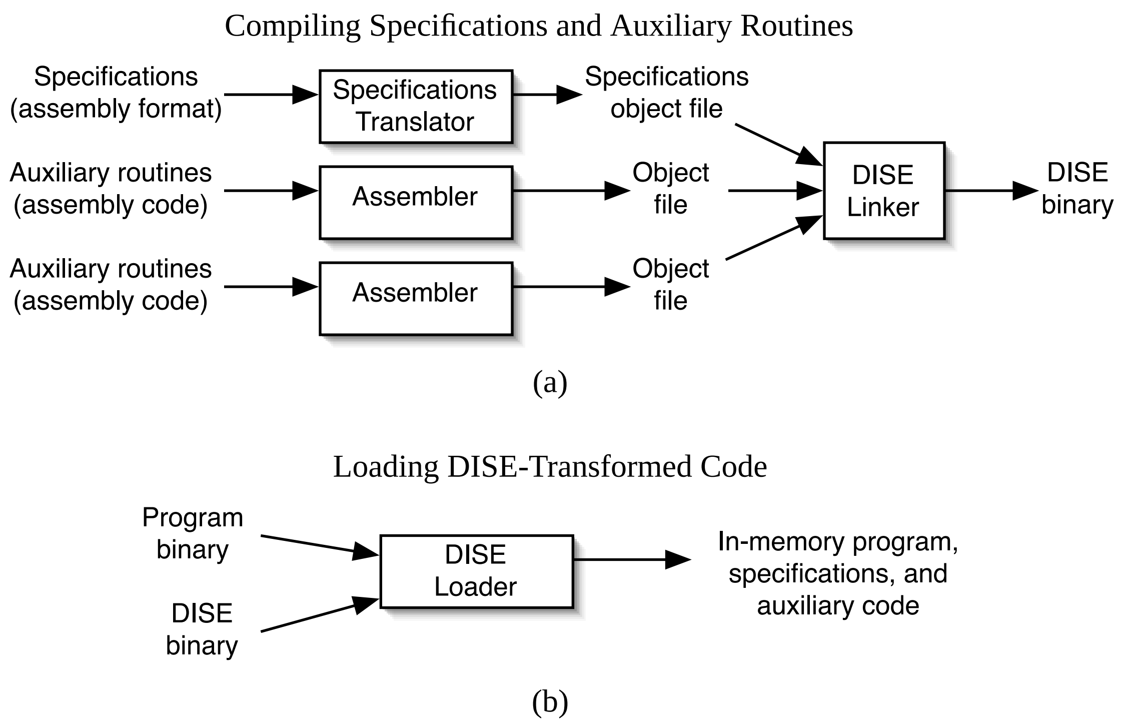

Compiling Specifications and Auxiliary Routines
Specifications (assembly format)
Specifications Translator
Specifications object file
Auxiliary routines (assembly code)
Assembler
Object file
Auxiliary routines (assembly code)
Assembler
Object file
DISE Linker
DISE binary
(a)
Loading DISE-Transformed Code
Program binary
DISE binary
DISE Loader
In-memory program, specifications, and auxiliary code
(b)

Figure 3.10: Compiling DISE specifications and auxiliary routines: (a) compiling and (b) loading.

not refer to symbols (routines or data) in the application. This restriction helps preserve the separation of DISE state from program state. Note the name spaces for DISE code and program code are separate. For example, both the DISE binary and the program binary can contain a routine called **malloc()**.

For an aware transformation, DISE users can store the binary specifications and auxiliary code within the program binary. Combining the binaries is convenient since the user only needs to keep track of one file. However, in some contexts, the user may want to alter the specifications without altering the program (Chapter 7 shows an example in security). When using a single binary, the header indicates that it contains both program code and DISE code. The header also indicates the location within the binary file of the DISE specifications and auxiliary code.

To run a program using DISE, the user sets an environment variable. The user also stores the path to the DISE binary file in a second environment variable. As shown in Figure 3.10(b), the loader takes the program binary (via the command-line) and the DISE binary (via the environment variable) and moves the program, specifications, and auxiliary code to memory. The loader must patch any references in the program or the auxiliary code, however, this works the same as with a conventional (non-DISE) loader (in Alpha, the loader uses a global offset table [81]). To load the specifications, the loader follows the steps outlined in the previous subsection; it moves the patterns and replacement sequences to memory (with DISE disabled) and then initializes **$dspecptr** and **$drsbase**. Before the loader begins executing the C runtime library (or the program), it performs a **d_sync** and enables DISE.

Of course, an aware application can directly program DISE, rather than (or in addition to) programming DISE via the loader. Furthermore, if loader support is unnecessary, then the user does not have to set any environment variables.

Initializing DISE registers and memory. DISE registers and memory regions used in replacement code are initialized within auxiliary code. For instance, DISE users can define a special routine called **dise_start** that is executed by the loader before the program is started. This routine can set any DISE register via the instruction **d_mtdr**. In addition, the auxiliary code can define data segments, which the auxiliary routines (*e.g.*, **dise_start**) can initialize. If replacement instructions need access to a data segment then the address can be placed in a DISE register. In some cases, DISE replacement code or DISE-called functions may need to dynamically allocate memory. This functionality is achieved by linking a memory allocation routine such as **malloc()** into the DISE binary.

Figure 3.11 shows the auxiliary code for store address tracing that initializes the DISE registers and memory. The auxiliary code defines a data segment called **sat_log**, which is used to hold the address log. The auxiliary code also contains a **dise_start** routine. **dise_start** references the data segment using a **lda** (which is patched by the linker and loader). It then uses this address to set the log pointer, **$d1**, and the end address of the log, **$d2**.

Specification errors. Although programming DISE is easier than programming a software translator, programmers will still occasionally write faulty specifications. As with general programming languages, there are both syntactic and semantic errors. Some of these errors can be caught, while others cannot.

A syntactic error occurs when a user (or application) submits a specification or routine

```
# data segment for log
.comm sat_log,16384

# prolog (not shown)
dise_start:
    # save reg. $1 (not shown)

    # set $1 to address of log
    lda $1,sat_log

    # initialize $d1 (log pointer)
    d_mtdr $1,$d1
    # initialize $d2 (end addr.)
    lda $1,16384($1)
    d_mtdr $1,$d2

    # restore reg. $1 (not shown)

    ret
    # epilog (not shown)
```

Figure 3.11: Initialization code for store address tracing.

that uses an incorrect assembly format. For example, the user specifies an instruction with a non-existent opcode or forgets a comma in a replacement instruction. As with other programming languages, all syntactic errors are caught by the specifications translator.

There are also semantic errors in writing DISE specifications. The user might refer to a non-existent field (*e.g.*, **T.FOO**), either in the pattern or the replacement sequence. The user might also refer to a field that does not exist for a particular type of fetched instruction. For example, the user might specify a pattern to match on all loads that use **$1** in the **RC** field, even though loads do not use the **RC** field. These errors are also caught by the translator. A user might also program a DISE call using a **bsr** or **jsr** rather than a **d_call**. Often, when this mistake is made, there is code following the call, which is unreachable (because the function will return to the trigger instruction rather than the middle of the replacement sequence, as discussed in the previous section). If the translator finds any unreachable code, it returns an error.

General ISA instruction extensions

d_toggle	Enable/disable DISE engine
d_sync	Synchronize DISE engine with specifications in memory
d_mtdr / d_mfdr	Move values between DISE registers and conventional registers

(a)

DISE ISA instruction extensions

d_beq / d_bne	DISE intra-instruction branch
d_call	DISE unconditional call
d_ccalleq / d_ccallne	DISE conditional call

(b)

Table 3.3: ISA instruction extensions: (a) extensions to the general ISA and (b) extensions to the DISE ISA.

In addition, if users write specifications using the low-level interface—or if the specifications translator is faulty—then semantic errors can occur with the low-level representation. These errors will not be caught by the translator since it is not invoked. However, the DISE hardware will throw an exception when it discovers a faulty specification (*e.g.,* a replacement instruction that parameterizes a field that does not exist in the trigger instruction).

Finally, as with other programming languages, there are errors that cannot be caught. For example, the user wanted to match on all stores, but instead wrote a pattern that matches on all loads, or the user wrote an incorrect replacement sequence that accidently causes a segmentation violation. Because of the simplicity of the DISE API, the expectation is that these types of errors are rare.

3.2.5 ISA Extensions

Throughout the last two sections, we have discussed some extensions (both instructions and registers) to the ISA. The additional instructions are listed in Table 3.3. Some of these extensions, like the instruction **d_mtdr** can be used anywhere (in application code or DISE replacement sequence code). These represent general ISA changes. Other instructions, like **d_ccallne**, can only be used in a DISE replacement sequence. These extensions don't change the ISA, but rather the *DISE ISA*. Both the ISA support and the DISE ISA support are described below.

ISA support. It is important to note that the ISA changes in a DISE processor are only extensions. The existing instructions are not changed. Furthermore, of the new instructions we have introduced, only **d_toggle**, **d_sync**, **d_mtdr**, and **d_mfdr** represent ISA changes (shown in Table 3.3(a)). The other instructions (*e.g.*, **d_ccallne**) are instead part of the DISE ISA (discussed below). Moreover, we can dedicate one opcode for all of these instructions and use function bits to distinguish between them.

DISE registers can also be considered part of the ISA since they are referenced by the instructions **d_mtdr** and **d_mfdr**. However, they cannot be referenced by any other non-replacement instruction. Therefore, their impact on the general ISA is small.

DISE ISA support. In Section 3.1, the replacement sequences for several transformations included instructions that are not part of the Alpha ISA (shown in Table 3.3(b)). The replacement sequences also used additional operands such as DISE registers and the instruction's PC. All of these extensions are possible because replacement instructions have a wider encoding than conventional instructions, allowing them to encode additional opcodes and operands.

3.2.6 Operating System Support

DISE is a user-level tool only. It is not designed be used to transform kernel-level code. When in kernel mode, DISE will not match on any instructions. There are potential applications of DISE that transform kernel code. For example, DISE could perform transformations like those done in Nooks [84], which fault isolate device drivers. However, restricting DISE to a user-level tool simplifies the design of the operating system and

DISE-enabled status bit
In-DISE-call status bit
DISEPC
General-purpose dedicated registers \$d0-\$d15
Specialized register \$dspecptr
Specialized register \$drsbase
Specialized register \$dra
Pattern specifications in memory
Replacement sequence specifications in memory

Table 3.4: Architected DISE state.

DISE. Without it, system designers would have to prevent users from subverting the OS using DISE.

As a result, only modest operating system support is necessary in a DISE system: the OS must save and restore DISE state on a context switch. Table 3.4 lists the architectural DISE state. Of this state, the specifications are stored in the process's virtual address space, and so, the OS does not have to do anything special with them. The OS must save the DISE-enabled and in-DISE-call status bits (in addition to the other conventional status bits), the DISEPC (in addition to the PC), the general-purpose DISE registers (in addition to the conventional registers), and the special registers **\$dspecptr** (pointer to the specifications table in memory), **\$drsbase** (base pointer to the replacement sequences in memory), and **\$dra** (return address on a DISE call). To save the DISE registers (general-purpose or special), the OS uses the instructions **d_mfdr** and **d_mtdr**. In addition to saving the state of the suspended process, the OS must also synchronize the DISE engine with the resumed process's specifications in memory. This synchronization is done using the instruction **d_sync**.

3.3 Summary

DISE transforms programs using processor-based instruction macro-expansion. It inspects every fetched instruction in the processor, and macro-expands those that match based on user-defined transformation specifications. Instruction macro-expansion consists of two parts: matching and replacement. To program DISE, users define a pattern for matching instructions, and a replacement sequence that replaces the matched instruction. To aid programmers, DISE provides several features including a private register set and control flow that can be used within replacement sequences. DISE can transparently transform unmodified executables, matching and replacing “naturally occurring” instructions. It can also transform aware applications that have specially-crafted instructions embedded within them. In the former case, DISE redefines the semantics of existing instructions, and in the latter case, DISE defines how the new instructions work.

Because DISE is a hardware mechanism, it naturally has a low-level, binary interface. However, users can write specifications at a higher level, using short blocks of assembly code that are easy to construct and reason about. These assembly-formatted specifications are translated into the low-level representation in a process similar to compiling a general assembly program. As we will show in Chapters 5-7, this interface allows us to express a broad range of transformations.

The operating system support necessary in a DISE system is small. DISE is a user-level tool only, and therefore the primary task of the operating system is to save and restore state on a context switch. Furthermore, most DISE state is stored in memory, and thus, the OS does not have to do anything special with it. The OS must save the DISEPC, DISE registers (both general-purpose and specialized), and the status bits. The OS must also execute a synchronization instruction before starting the new (resumed) process.

This chapter presented only the DISE architecture and did not discuss the microarchitecture. In the next chapter, we explore the microarchitecture, and propose and evaluate one highly-optimized implementation of DISE.

Chapter 4

DISE Microarchitecture

This chapter proposes a highly-optimized implementation of the DISE microarchitecture within an Alpha processor. Section 4.1 describes our microarchitectural implementation, which caches specifications to reduce the overhead of macro-expansion. Section 4.2 evaluates the implementation, demonstrating that DISE transformation is highly efficient and evaluating several key design parameters.

4.1 Microarchitectural Implementation

The microarchitectural implementation of DISE has two primary objectives: little to no performance degradation on non-transformed code and transformation overhead that is equal to or less than that of a corresponding software implementation. We propose a microarchitectural implementation that meets these two goals. As shown in Figure 4.1, we implement DISE between the fetch unit and the decoder within a processor's pipeline. Our implementation caches the patterns and replacement sequences so instructions can be efficiently macro-expanded within at most one or two pipeline stages. These additional stages are the only overhead for non-transformed programs, which in general is small (*i.e.*, the primary cost is on a branch misprediction, which is infrequent for most applications). In addition, the changes to the processor are primarily localized to the point where expansion takes place. The rest of the processor requires only modest changes.

Although other microarchitectural implementations of DISE are possible, below we

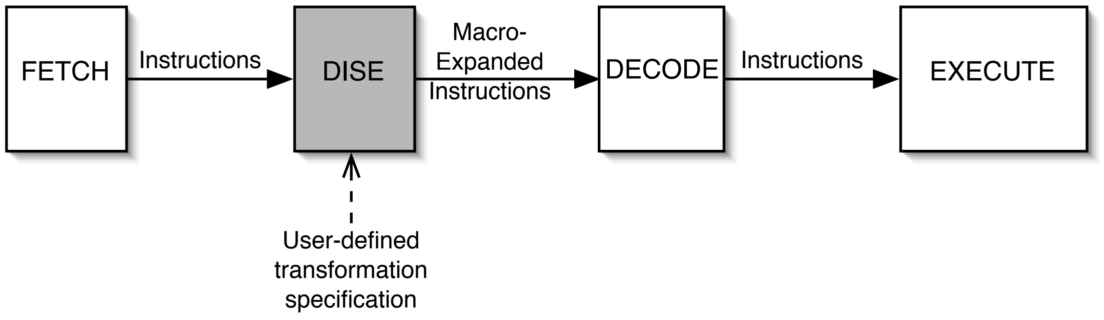

FETCH
Instructions
DISE
Macro-Expanded Instructions
DECODE
Instructions
EXECUTE
User-defined transformation specification

Figure 4.1: A simplified diagram of a DISE processor.

discuss one implementation. We describe the DISE engine hardware structures as well as the pipeline organization. We also discuss DISE control flow prediction. Finally, we look at other microarchitectural changes, beyond the DISE engine.

4.1.1 DISE Engine

Although the usage model is different (as described in Chapter 2), the DISE mechanism for performing instruction macro-expansion is similar in implementation to the instruction-to-microinstruction mechanism in IA32 processors that converts complex instructions to a more regular (three register) internal form [34, 39, 41, 42].

Figure 4.2 shows the abstract structure of the implementation of the instruction macro-expander, which transforms matching instructions into a parameterized replacement sequence. To lower the cost of macro-expansion, we cache patterns in a *pattern table (PT)* and replacement sequences in a *replacement table (RT)*. A third structure, called the *instantiation logic (IL)*, instantiates the replacement instructions on an expansion. We discuss each of these three structures (PT, RT, and IL), below.

PT. The PT houses the pattern specifications (described in Section 3.2) for matching instructions. The PT is a fully-associative structure that matches every instruction to all enabled patterns. It is ordered by priority. Each entry in the PT contains a pattern specification along with some matching logic that determines if the pattern specification in that particular entry matches the inputted instruction (*i.e.*, the matching logic is replicated for each PT entry). PT entries are tagged using their index within the specifications table (described in Chapter 3). Some PT entries may be invalid, for example, if there are fewer

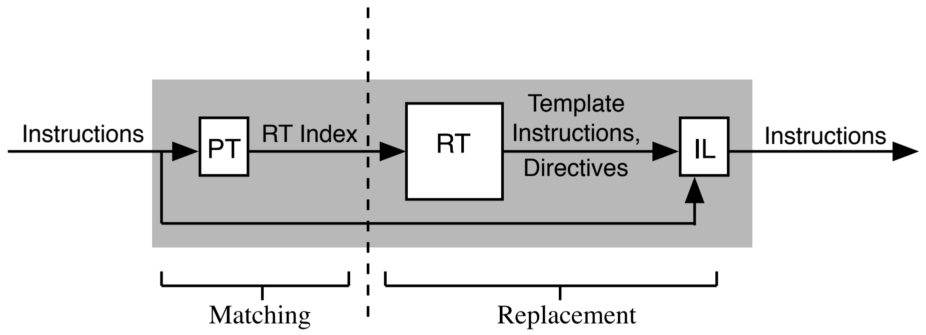


Figure 4.2: Abstract diagram of the DISE engine.

specifications than entries. Because the PT is fully associative and each entry has its own matching logic, it must be small (*e.g.*, 16 or 32 entries).

Matching an instruction on the Alpha is achieved by masking bits in the instruction and comparing to the appropriate field in the PT entry There is one possible exception: matching based on opclass. There are two possible implementations of opclass matching. It might be achieved through bit masking. In fact, it is likely, the Alpha (as well as other processors) do something similar already, since some opclasses are required by later stages in the pipeline. For example, the dispatch stage must know whether an instruction is a store (in Alpha, each store has a distinct 6-bit opcode), so that it can allocate a store queue entry for it. If bit masking will not work, then a second approach is to use a table (implemented in ROM), where the rows represent opcodes and the columns represent opclass codes. Each entry in the table is a single bit that indicates whether the opcode at that particular row belongs to the opclass at that particular column. Using the second approach may require limiting the number of opclasses in order to reduce the size of the table so that it can be accessed in one cycle.

Detecting a miss in the PT is more challenging than in a traditional cache (*e.g.*, data cache), because a PT miss is interpreted as a non-match. To detect PT misses, we track the number of enabled in-memory patterns associated with each opcode in a small direct-mapped table called the *pattern counter table (PCT)*. In addition, the matching logic in the PT computes the number of PT-resident patterns associated with the fetched instruction's opcode. If this value does not match the value in the PCT entry for that particular opcode, then a PT miss is triggered (even though a matching pattern with the highest priority may

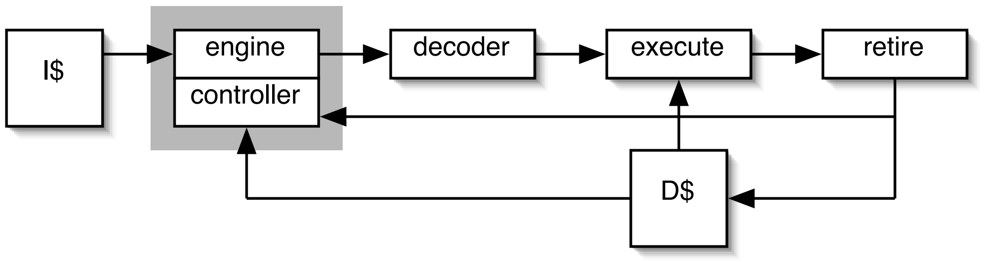


Figure 4.3: DISE controller.

already be in the PT).

A *DISE controller*, shown in Figure 4.3 services PT misses (as well as RT misses, discussed below). A PT miss interrupts the processor via the controller. A software miss handler scans each pattern in memory, going from highest priority to lowest priority. It checks if the pattern could match any instruction with the same opcode as the fetched instruction. The handler routine moves each such pattern into the PT, until the PT contains only these patterns (*i.e.*, patterns that could match on the fetched instruction's opcode). Of course, a pattern may already be in the PT, so it first checks the PT tags of each valid entry. Because the PT is fully-associative, the handler can swap a memory-resident pattern with any PT entry. The controller uses a not-recently-used (NRU) replacement strategy. The controller also must maintain priority in the PT, which will require it to move some patterns to either higher or lower priority entries, before bringing the new pattern into the PT.

Although serviced similarly to software-managed TLB misses, the cost is higher due to the additional cost of scanning patterns in memory and prioritizing patterns in the PT. Assuming there are less than 100 patterns (a reasonable assumption, given that all transformations combined in this dissertation require less than 20 patterns), then servicing the miss could take a few hundred cycles. Therefore, frequent PT misses could significantly degrade performance.

To synchronize the PT with memory-resident patterns, the controller flushes the PT when a **d_sync** is executed. The contents of the PT are subsequently faulted in on misses. In addition to flushing the PT, the controller fills the PCT when a **d_sync** is executed. The DISE controller scans the pattern specifications in memory, counts the number of possible matches for each individual opcode, and updates the PCT. Therefore, a **d_sync** will have a similar cost as a PT miss.

RT. As described in Section 3.2, each pattern specification contains a replacement sequence index. On a match in the PT, the low-order bits of this index are used to access the corresponding replacement sequence in the RT, which caches replacement sequences. For explicitly-tagged specifications, the low-order bits of the codeword tag are used as an index rather than the index in the pattern specification.

Because the RT is not associatively searched, but rather indexed, it can be made much larger than the PT. Possible sizes are 512 entries to 2K entries. RT entries, which do not have to be quad-aligned, are 6 bytes rather than 8 bytes (see Section 3.2). Therefore, a 512-entry RT is 4KB and a 2K-entry RT is 16KB. The RT may be direct-mapped or set-associative. Set associativity may increase the access time and power consumption of the cache, however, it also reduces the conflicts. In general, increasing the RT associativity is a good idea so long as it does not impact the clock rate. As we show in the evaluation in Section 4.2, a 2-way set associative RT results in significantly fewer misses than a direct-mapped RT.

To reduce access time, the RT is split into **n** banks where **n** is the width of the processor. Although the length of the replacement sequence is not known *a priori* on a PT match (*i.e.*, the last flag in the replacement instruction specification determines the end of the sequence), **n** replacement instructions, starting from the replacement sequence index, are extracted (in parallel) from the RT. If the sequence is greater than **n**, the pipeline is stalled and **n** additional instructions are extracted from the RT during the next clock cycle.

Unlike PT miss detection, RT miss detection is similar to miss detection in a conventional cache. Each RT entry is tagged using the replacement instruction index. If the replacement sequence index in the PT, or the codeword tag for explicitly-tagged specifications (which is incremented to access later instructions in the sequence), does not match the RT tag, then a miss is triggered. A RT miss interrupts the processor via the controller. The mechanics of RT miss handling resemble those of software TLB miss handling and have similar costs. The pipeline is flushed and the missing specifications are loaded via the controller. If the RT is set associative, then the controller uses an NRU replacement strategy.

As with the PT, the controller flushes the RT when a **d_sync** is executed.

Instantiation logic. The third structure is the instantiation logic, a combinational circuit that executes instantiation directives to combine replacement literals with trigger fields and produce the actual replacement instructions that are spliced into the application's execution stream.

4.1.2 Pipeline Organization

DISE is situated in the front-end of a processor's pipeline. It logically resides between the fetch and decode stage. To incorporate DISE into the pipeline requires a few modifications from the diagram in Figure 4.2. First, a selector is necessary to select between the original fetched instruction on a non-match and the DISE replacement sequence on a match. The PT, which determines if a fetched instruction matches a pattern specification, drives the selector. In addition, since DISE transforms a single instruction into multiple instructions, a buffer is necessary for any excess instructions. When the buffer is full, the pipeline is stalled. For non-superscalar machines, the additional stalls will hurt performance, but for wider machines these stalls are not significant as Section 4.2 shows.

Even though the PT and RT are small, because they are accessed in series, it is unlikely they can both be read in a single cycle. Given this constraint, there are two strategies for laying out the PT and RT. These approaches are shown in Figure 4.4. In Figure 4.4(a), the PT and RT are positioned within a single stage and a match results results in a 1-cycle pipeline stall. In Figure 4.4(b), the PT and RT are positioned in separate stages.

These two approaches represent a tradeoff. The first approach has less impact on non-DISE code. Instructions that do not macro-expand must pass through only one additional pipeline stage rather than two stages. However, each DISE expansion causes a pipeline stall. In the second approach, DISE expansion does not result in any stalls, however, it adds an extra pipeline stage. The primary cost of an additional pipeline stage is an extra cycle to recover from a branch misprediction. If the number of mispredictions outweighs the number of DISE expansions then the first approach will perform better, and *vice versa*. In the evaluation section (Section 4.2), we show that when DISE is enabled, the second approach (Figure 4.4(b)) results in higher performance since DISE expansion is frequent and branch misprediction is not for most benchmarks.

In some architectures, processor designers can optimize the organization of DISE by

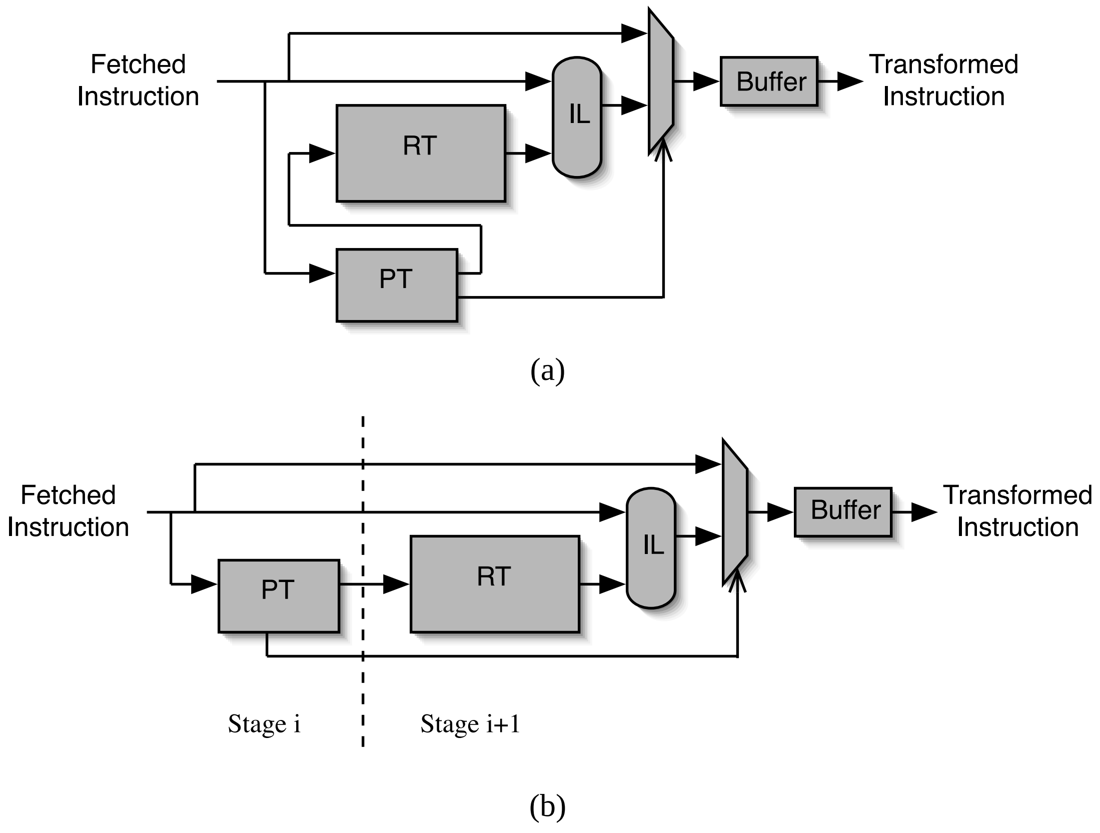


Figure 4.4: DISE engine implementations: (a) one stage with a feedback loop and (b) two stage.

incorporating it within the decoder. Figure 4.5 shows this organization for the one stage implementation of DISE with a feedback loop from the PT to RT. This organization reduces one pipeline stage. However, if the processor has a one-stage decoder as in Figure 4.5, then DISE still requires a stall for every expansion (but not an additional stage) or one additional stage (but not two).

With the unified organization, the RT must contain pre-decoded replacement sequences. Because replacement sequences are programmed at the instruction level, they must be translated (*i.e.*, decoded) before placed in the RT. The DISE controller, which manages the PT and RT, can also perform this translation, and abstract the internal microinstruction formats. By owning the controller, the processor vendor retains the freedom to change internal formats in future products. The controller translates specifications from their external representation—a directive-annotated version of the processor's native ISA—to the internal formats used by the PT/RT. Pre-decoding is done on an RT fill.

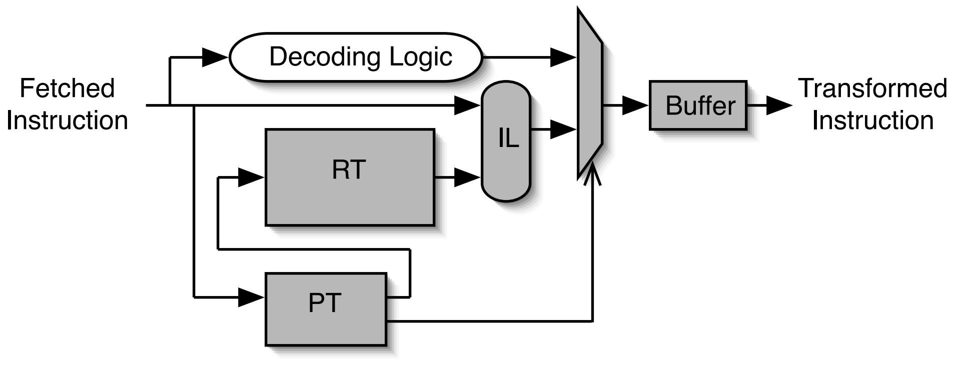


Figure 4.5: A decoder-based DISE engine implemented in one stage using a feedback loop. DISE-specific hardware is shaded.

x86 implementation. This section has described the implementation of DISE within an Alpha processor. Below we discuss implementing DISE in a x86 processor. We do not provide a complete solution, but rather summarize some issues and challenges.

The x86 CISC ISA presents one implementation challenge. Pattern matching for a CISC ISA is more difficult than for a RISC ISA. An instruction can perform many distinct tasks, such as simultaneously call a routine and write the return address to the stack in memory. Implementing the hardware logic to find all memory writes in an x86 architecture is much more challenging than in an Alpha architecture. In addition, the x86 ISA uses an irregular encoding. Finding the individual fields within a x86 instruction requires more complex, and potentially higher latency, circuitry. Therefore, it may be necessary to partially decode and reformat instructions before accessing the PT for more efficient matching. It is likely that x86 processors already contain pre-decode facilities. However, depending on the pipeline organization, pre-decoding might increase the stall time on a DISE match or require an additional stage for matching.

A benefit of implementing DISE in an x86 processor is that, as described above, the DISE mechanism is similar to the CISC-instruction-to-RISC-microinstruction in the x86 decoder [34, 39, 41, 42]. In an x86 implementation, it is possible to unify the two mechanisms. Thus DISE expansion has no impact on both DISE code and non-DISE code. Figure 4.6 shows this organization. CISC-to-internal-RISC translation is performed in two ways. Translation resulting in four or fewer microinstructions is done via combinational logic arrays (CLAs). Translation requiring longer sequences is performed by sequentially

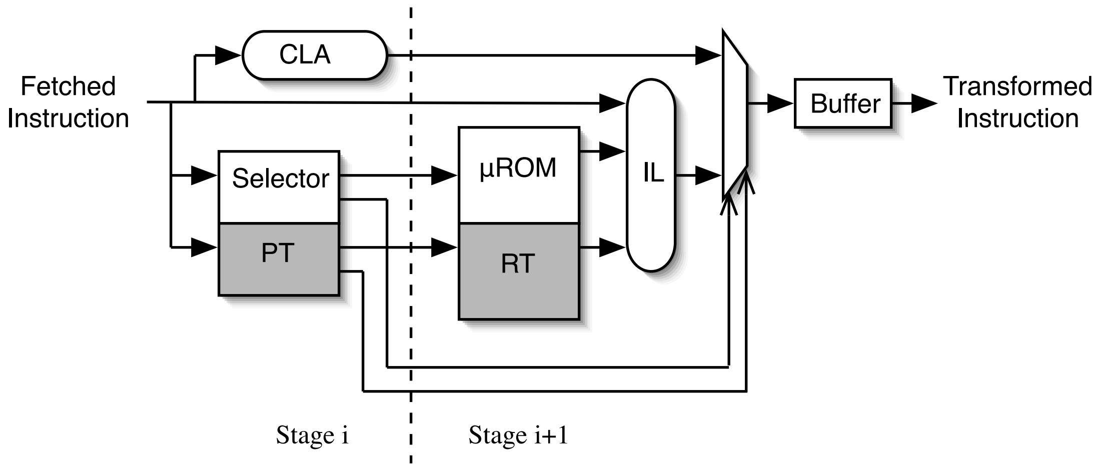


Figure 4.6: DISE engine implementation in a CISC (x86) processor. The DISE-specific hardware is shaded.

instantiating templates from a ROM (μROM). CLA/μROM multiplexing (and μROM indexing) is based on the CISC opcode and performed by a selector, itself either a CLA or ROM. The PT and RT parallel the selector and μROM, respectively, in functionality and positioning. A PT match overrides both the μROM and CLA. The CISC-to-RISC and DISE ILs are physically unified. The RT and μROM may also be unified, although the original μROM contents must be kept immutable and invisible.

At the same time, because the decoder in an x86 is already complicated it may be difficult to incorporate DISE into the decoder without negatively impacting clock rate. This problem is more prevalent for wider machines such as the Pentium 4. For such machines, DISE may be limited to macro-expanding only one instruction at a time. If multiple instructions require simultaneous macro-expansion, the younger instruction(s), and all preceeding instructions, are stalled until the next cycle.

Another challenge in implementing DISE in current x86 processors arises due to the trace cache. The trace cache removes an important benefit of DISE: no code bloat. Because a trace cache houses microinstructions that have been previously decoded, these instructions are already expanded. Therefore, the performance characteristics of a trace cache implementation of DISE is similar to the performance characteristics in a static binary implementation. There are still other benefits to using DISE, such as programming ease and late-binding transformation. It may be possible to position DISE after the trace cache, however, this organization would add significant complexity to both the DISE engine and controller. For example, DISE would need to pattern match on pre-decoded instructions, although users program patterns in terms of machine instructions. The controller would need to translate patterns into the appropriate form.

4.1.3 DISE Control Flow

DISE control transfers are implemented by flushing the pipeline and restarting at ⟨new PC:new DISEPC⟩. For intra-instruction control flow, which transfer control within the replacement sequence, the PC is unchanged and only the DISEPC is updated (*i.e.*, to the index within the replacement sequence of the target instruction). For inter-instruction control flow, which transfer control to another instruction in the application, the PC is set to the target PC and the DISEPC is set to 0 (transfers to the middle of another replacement sequence are not allowed).

DISE calls work similarly to other inter-instruction control transfers except when they are executed, the DISE engine is disabled and the return address is saved in a DISE register **$dra** (a register suffices because there is no recursive expansion). The DISE return uses this address to return to the trigger instruction of the replacement sequence that contained the DISE call.

Restarting within the middle of a replacement sequence (*e.g.*, on an intra-instruction control transfer or after a page fault exception is handled) is facilitated primarily by the DISE engine. The fetch unit ignores the DISEPC, instead fetching the trigger instruction. The DISE engine recognizes the annotation and expands the replacement sequence starting at offset DISEPC.

For simplicity, control flow within a DISE replacement sequence is always predicted not taken. Because DISE expands instructions within the decoder, it does not have easy (low-latency) access to the branch predictor. We could have added a DISE-only branch predictor or added support for statically-predicted control flow. However, for the transformations discussed in this dissertation, we have found that conditional calls and DISE auxiliary functions can eliminate most mispredicted control flow.

Reducing mispredictions. As discussed in Chapter 3, the DISE ISA provides a conditional call (*e.g.*, **d_ccalleq** and **d_ccallne**). As shown in Figure 3.2(b) (page 27), without a conditional call, the replacement sequence for store address tracing results in a misprediction every time it is executed: either the branch or the call will be taken. As shown in Figure 3.4 (page 27), a conditional call can reduce these mispredictions. The new specification has one fewer instruction because the branch and call are fused, but its primary advantage is that it avoids frequent pipeline flushes due to mispredictions. A flush occurs only when the conditional call is taken. Section 4.2 evaluates the performance impact the conditional call, and shows that the conditional call is critical for store address tracing. Furthermore, this scenario is not isolated to store address tracing. As shown in Chapters 5 and 7, it is also important for debugging and security enhancing transformations.

In addition, if users need to use more complex control flow patterns, which contains many taken branches or jumps, then they can place this code in a DISE auxiliary routine. DISE is disabled for routines called from a DISE sequence. Therefore, any control flow within the routine is predicted normally (the call to the routine is still mispredicted).

4.1.4 Other Microarchitectural Changes

Changes to a DISE processor are primarily localized to the front-end stages of the pipeline where expansion takes place. However, there are some other modest changes to the processor, which we outline below.

New instruction support. Some new instructions must be supported (*e.g.*, **d_mtdr**). The instructions **d_mtdr** and **d_mfdr** (discussed in Chapter 3) can be implemented as simple moves. The **d_toggle** instruction sets the DISE-enabled status bit to either 0 or 1. The **d_sync** instruction communicates with the DISE controller, which results in a PT/RT flush. Both **d_toggle** and **d_sync** trigger a pipeline flush. In addition, the processor must have support for conditional calls (*i.e.*, **d_ccalleq**, **d_ccallne**). Finally, DISE calls (*i.e.*, **d_call**, **d_ccalleq**, **d_ccallne**), and returns (**d_ret**) must enable and disable DISE, respectively; set and unset the in-DISE-call status bit, respectively; and trigger a pipeline flush.

DISE registers and status bits. A DISE processor also requires 16 general-purpose, DISE registers and 3 specialized, DISE registers. The size of the physical register file and map table will need to be increased to account for these additional architected registers. Also, two additional status bits are necessary: a DISE-enabled bit and a in-DISE-call bit.

Machine	MIPS R10000-like, out-of-order, speculative
ISA	Alpha
Processor width	4
Pipeline stages	12
Reorder buffer size	128
Reservation stations	80
Instruction cache	32KB, 2-way set associative, 1-cycle access time
Instruction TLB	64-entry, 4-way set associative
Data cache	32KB, 2-way set associative, 1-cycle access time
Data TLB	64-entry, 4-way set associative
Unified L2 cache	1MB, 4-way set associative, 12-cycle access time
Main memory	infinite, 100 cycle access time
Memory bus	32 bytes wide, 1/4 processor frequency
Branch predictor	Hybrid bimodal/gshare, 8K entry
Branch target buffer	256 entry

Table 4.1: Default machine characteristics.

4.2 Evaluation of the Microarchitecture

This section evaluates the microarchitectural implementation of DISE presented in the previous section. It shows the overhead of DISE transformation, comparing DISE with software translation. It also performs a sensitivity analysis on the pipeline organization, DISE structures, and DISE ISA. We evaluate the DISE microarchitecture primarily using store address tracing (one figure evaluates decompression and one table shows the number of PT/RT entries for all transformations in this dissertation). In Chapters 5-7, we evaluate other transformations.

4.2.1 Methodology

We evaluate DISE using simulation tools built on top of the SimpleScalar Alpha instruction set and system call definition modules [16]. We use a similar methodology throughout this dissertation (*i.e.*, in Chapters 4-7). Table 4.1 shows the default machine characteristics. The simulator models a MIPS R10000-like 4-way superscalar processor with a 12 stage pipeline, 128 entry reorder buffer, 80 reservation stations, and aggressive branch and load speculation. We model an on-chip memory hierarchy with 32KB instruction and data caches and a unified 1MB L2. By default, the simulator models a 2-stage, decoder-based implementation (the DISE processor uses 1 more pipeline stage than the baseline

benchmark	static attributes	dynamic attributes		
	code size (insn)	IPC	I$ misses	stores
bzip2	36,013	2.3884	$\sim$0%	16.93%
crafty	82,863	2.0447	.47%	5.46%
eon	150,998	2.1230	.61%	17.41%
gap	172,581	1.5479	.86%	11.18%
gcc	364,429	1.4368	1.58%	12.24%
gzip	38,871	2.3498	$\sim$0%	8.07%
mcf	32,018	1.3803	.01%	14.87%
parser	57,617	1.8122	.03%	11.29%
perlbmk	173,135	1.7513	.61%	14.59%
twolf	88,324	1.9100	.05%	7.53%
vortex	162,613	2.1834	.99%	16.91%
vpr	70,735	1.7034	$\sim$0%	11.61%

Table 4.2: Benchmark summary.

processor). The default configuration uses a PT and RT with 32 entries and 2K entries, respectively. Each PT entry occupies 8 bytes and each RT entry occupies 6 bytes. Therefore, the total size of the PT and RT is 256 bytes and 12KB, respectively. On a PT or RT miss, the simulator flushes the pipeline and stalls for 30 cycles.

Our simulator only models single-process performance, it does not model a full system. When we evaluate store address tracing, below, we do not model the cost of writing the log to disk.

We transform the SPEC2000 integer benchmarks. The benchmarks are compiled for the Alpha EV6 architecture with GCC 3.2.2 using the -O4 optimization flag. Results are reported using complete runs on test inputs. Table 4.2 lists some characteristics of the benchmarks that are useful in interpreting the results below. The instruction cache misses, calls (returns), loads, and stores columns are normalized to the total number of executed instructions.

Our simulation environment extracts all nops from both the dynamic instruction stream and the static program image. They are inserted by the Alpha compiler to optimize two idiosyncratic aspects of the Alpha microarchitecture (cache-line alignment of branch targets and clustered execution engine control). Our simulator does not model these idiosyncrasies, so for us the nops serve no purpose. Furthermore, when we evaluate compression/decompression (in one figure below, and in more detail, in Chapter 6), the presence of these nops may exaggerate the benefits of compression.

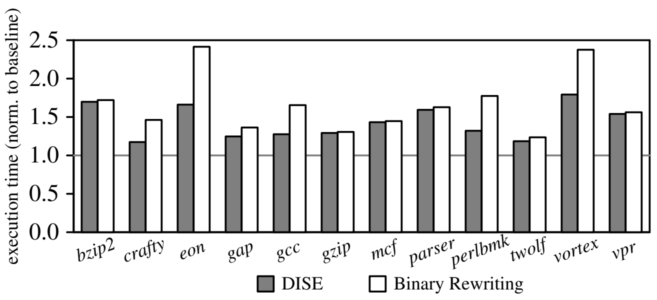


Figure 4.7: The overhead of DISE and binary rewriting for store address tracing.

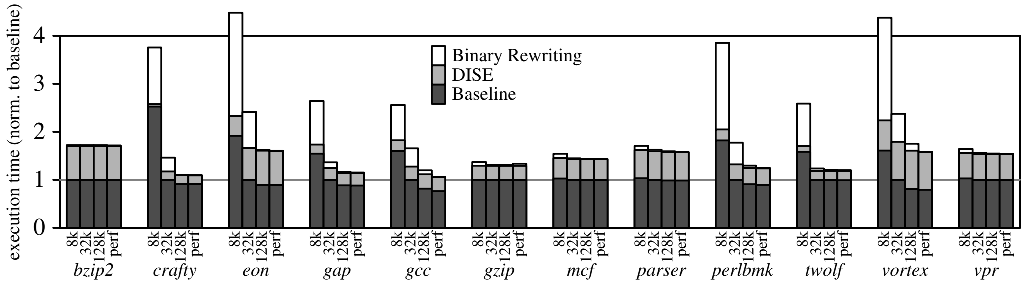
4.2.2 Transformation Overhead

Figure 4.7 shows the overhead of DISE for store address tracing. In general, the overhead of DISE transformation is low. For most benchmarks it is less than 25%. In some cases, overheads over 50% (*e.g.*, *bzip2*, *eon*, *parser*, *vortex*, *vpr*), but the overhead is always less than 80%.

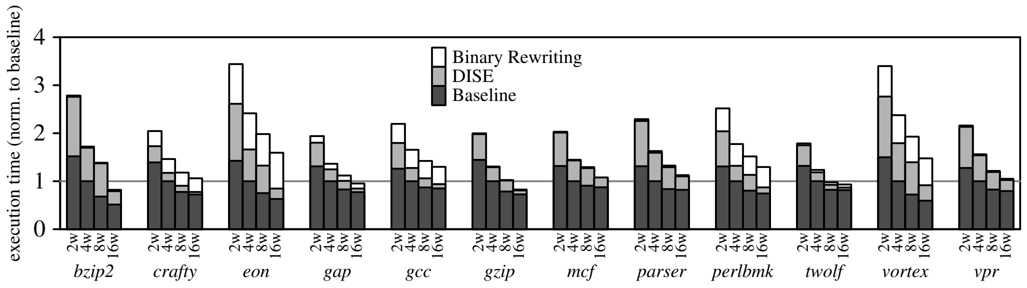
Versus static transformation. Figure 4.7 also compares DISE with a binary rewriting implementation. The binary rewritten code contains exactly the same instructions as the DISE transformed code (after dynamic instrumentation) because the former are not statically optimized. However, because this transformation instruments stores, there is little opportunity for static optimization. For many benchmarks (*e.g.*, *eon*, *gcc*, *perlbmk*, and *vortex*), DISE outperforms binary rewriting. The increased overhead of binary rewriting is a result of additional instruction cache misses. For larger benchmarks or benchmarks with a significant number of instruction cache misses, the difference in overheads is large. On *eon*, the overhead of DISE is 66%, while the overhead of binary rewriting is 141%.

Cache size and processor width. Transformation has two costs. The static cost is decreased effective instruction cache capacity. The dynamic cost is decreased effective pipeline throughput. DISE transformations have only the dynamic cost. Figure 4.8 isolates these costs, by relaxing cache capacity and pipeline bandwidth constraints, respectively.

Figure 4.8(a) shows relative execution times of DISE and binary rewriting on 4-wide processors with instruction caches of varying sizes. As cache size increases, static overhead decreases and the dynamic overhead remains constant. For the binary rewriting implementation, the relative total overhead decreases. The DISE implementation does not



(a)



(b)

Figure 4.8: Varying instruction cache size and processor width. The overhead of store address tracing in DISE and binary rewriting for several different (a) instruction cache sizes and (b) processor widths.

have any static overhead, so the relative dynamic overhead grows as the baseline performance improves with the growing cache size. These trends favor DISE. Physical cache size is limited by access latency while instruction working sets are growing.

Figure 4.8(b) shows relative performance on 32KB instruction-cache processors of different widths. At high widths, data dependences limit parallelism within a fixed reordering window, allowing replacement code to exploit idle resources at little perceived cost. Although this trend is apparent for DISE, the binary rewriting implementation does not improve as rapidly with wider machines. While increased processor width reduces the dynamic cost of transformations, the static cost remains and, in fact, becomes relatively larger. As the absolute cost of the application shrinks, the relative cost of each cache miss grows. This trend also bodes well for DISE: its advantage over binary rewriting will increase as processor performance grows.

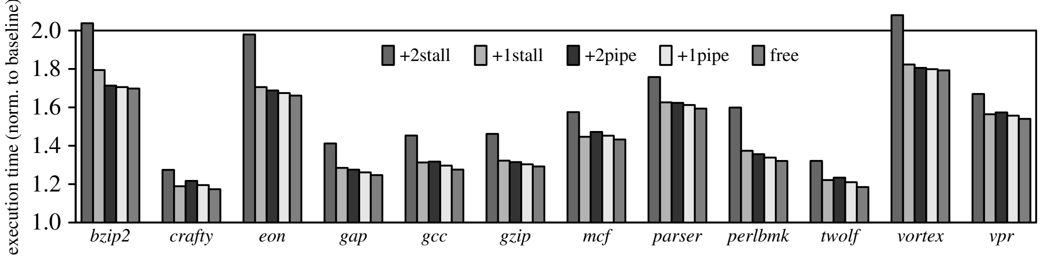


Figure 4.9: Performance impact of the DISE pipeline configuration.

4.2.3 Sensitivity Analysis

Pipeline configuration. Up to now, we have assumed a 2-stage decoder-based implementation of DISE, which has one more front-end pipeline stage than the baseline machine. Here we look at other possible configurations. As discussed in Section 4.1, fitting DISE into a single-cycle decoder (*e.g.*, Alpha decoder) requires either adding another decoding stage or incurring a single cycle stall on every successful DISE expansion. Furthermore, on some architectures it may not be possible to fit DISE within the decoder at all. In this case, DISE will require two stages or a two cycle stall for every expansion. The performance of these various implementations is evaluated in Figure 4.9 for store address tracing. The bars *+2stall* and *+1stall* represent implementations that add 2 or 1 cycles for every expansion, respectively. These implementations do not change the pipeline length. The bars *+2pipe* and *+1pipe* represent implementations with 2 or 1 additional pipeline stages, respectively. Macro-expansion does not cause a stall for these implementations. The *free* bar, shown for comparison purposes, represents a DISE implementation with no macro-expansion cost.

The effects in Figure 4.9 are intuitive: the penalty of an additional decoding stage is proportional to the frequency of mispredicted branches, typically around 5% (10% for 2 additional stages). The penalty of a single-cycle stall per expansion is proportional to the total number of expansions (multiplied by two for 2-cycle stall). The advantage of the stall option is that there is no performance degradation on non-transformed code. Unfortunately, expansion frequency is often much higher than branch misprediction frequency. Store address tracing, for instance, expands about 12% of dynamic instructions. If heavy

Transformation	Number of PT entries	Number of RT entries
Store address tracing (Chapter 3)	1 (match stores)	7
Debugging breakpoint (Chapter 5)	1 (match codewords)	2
Debugging watchpoint (Chapter 5)	1 (match stores)	8
Decompression (Chapter 6)	1 (match codewords)	$>$2K
Return address protection (Chapter 7)	2 (match calls/returns)	12
Pointer protection (Chapter 7)	2 (match pointer loads/stores)	4

Table 4.3: Number of PT and RT entries for each transformation shown in this dissertation.

DISE use is projected, the elongated pipeline is the more sensible choice. For the remainder of the evaluation, we assume this design. We also assume the decoder-based implementation is possible (*+1pipe*), *i.e.*, only one additional pipeline stage.

PT and RT size. Table 4.3 shows the number of PT entries and RT entries for each transformation used in this dissertation (some of these transformations are described in later chapters). Because we have not found a transformation that requires a large number of patterns (none of our transformations require more than 2 patterns) we do not evaluate the impact of PT size on performance.

On the other hand, we have found one transformation, decompression (presented briefly in Chapter 3 on page 22 and presented in greater detail in Chapter 6), that can fill a 2K RT for many benchmarks. Here we evaluate the impact of RT size and configuration on performance for decompression. To model an RT miss, which has a similar cost as a software-managed TLB miss, we flush the pipeline and stall for 30 cycles. We use decompression dictionaries that were selected to minimize code size without regard to RT misses. Although we have not described decompression in great detail (we do in Chapter 6), the details are mostly irrelevant for this study.

Figure 4.10 shows the performance for four RT configurations, 512 and 2K entries, each both direct-mapped and 2-way set-associative. Note that we only show results for half the benchmarks, but these are representative of the entire suite. In Figure 4.10, the 2K, 2-way RT (nearly) matches the perfect RT in all benchmarks. The direct-mapped configuration performs almost as well. The effectiveness of 512-entry RTs depends largely on the code size of the benchmark. For smaller programs (*e.g.*, *gzip*, *mcf*), the performance is good, especially with the set-associative RT. For larger programs, the performance degrades significantly. If RT misses are a significant problem, we can reduce RT misses by

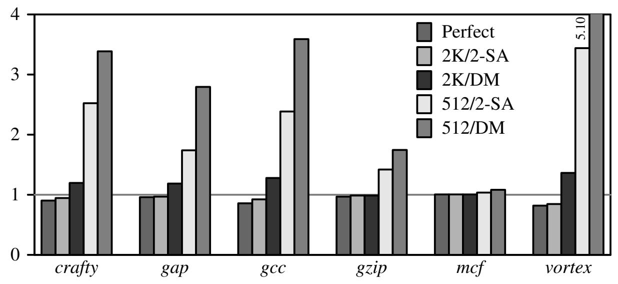


Figure 4.10: Performance impact of the RT configuration.

limiting the size of the dictionary. In this way, we can trade off compression for performance.

DISE ISA. As described in Section 4.1, DISE control flow is predicted not taken, which can lead to frequent mispredictions in some contexts. In store address tracing, the replacement code calls a DISE routine, but only when the log is full (a rare event). In this case, one misprediction is incurred every time the sequence is executed. Either the call is mispredicted (if it is executed) or the branch that jumps over the call is mispredicted. We can optimize this scenario by fusing the call and branch into a conditional call. By using a conditional call, we eliminate one instruction, but more importantly, we eliminate a misprediction when the call is not taken.

In the evaluation above, the DISE ISA included a conditional call. Here we look at the impact on performance when the DISE ISA does not include a conditional call. Figure 4.11 shows the performance with and without a conditional call for store address tracing. With a conditional call, the overhead is usually less than 25% and always less than 80%. Without the conditional call we observe slowdowns greater than 3 times for most benchmarks and sometimes as high as 6 times (*e.g.*, *eon*). Clearly, a conditional call is critical in a DISE system. For architectures that do not support conditional calls, a conditional should be added to the DISE ISA.

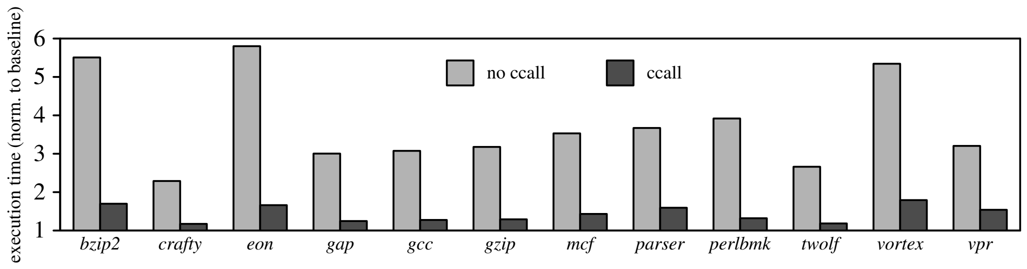


Figure 4.11: Performance impact of a conditional call.

4.3 Summary

In this chapter, we presented one optimized implementation of the DISE microarchitecture that uses caching to reduce the overhead of transformation. With caching, the penalty of macro-expanding instructions is reduced to one extra cycle per branch mispredict (which is generally rare). Although there is still the cost of executing the injected instructions, the transformation overhead is generally low (*e.g.*, less than 25%). The low overhead is in large part because, unlike software translators, DISE has no impact on instruction cache performance.

In addition, DISE requires only modest changes to the processor. Nearly all changes are localized to the two front-end stages where expansion takes place.

The evaluation in this chapter demonstrated that DISE overhead is low, usually less than 25%. Furthermore, the evaluation showed that DISE outperforms software translation and that trends in application workloads (*i.e.*, larger memory footprints) will widen this gap. The evaluation also looked at the sensitivity to several key design parameters including pipeline organization, and RT configuration.

Although we have shown the DISE mechanism, in detail, in both this chapter and the previous chapter, we have not demonstrated the utility of DISE. In subsequent chapters, we will demonstrate DISE's utility in three different contexts: debugging, code compression, and security.

Chapter 5

Debugging with DISE

Programming errors (*bugs*) are an unfortunate but inevitable part of the application development cycle, and debugging, the identification and repair of these errors, is a major enterprise. Although tools exist to identify the sources of certain classes of errors (*e.g.*, memory leaks), in general there is no substitute for interactive debugging. A user employs a *debugger* to observe a bug as it develops in order to trace it to its origin; a debugger allows a user to control the execution of an application and inspect/manipulate its state.

This chapter looks at the applications of DISE in debugging. It focuses on interactive debugging, showing that DISE can aid interactive debuggers in efficiently and non-intrusively monitoring (buggy) applications. In particular, interactive debuggers can use DISE to implement *breakpoints* and *watchpoints*, two important, but often, inefficient, debugging tools. Breakpoints (sometimes called *control breakpoints*) and watchpoints (sometimes called *data breakpoints*) allow users to focus on the application's instructions and data that may be pertinent to a particular bug. A breakpoint transfers control to the user when the application executes a user-specified instruction. A watchpoint transfers control to the user when the value of a user-specified expression changes. Both breakpoints and watchpoints can be *conditional* so that control is only transferred if the breakpoint/watchpoint criterion is met and a user-specified predicate is true. Essentially, breakpoints, watchpoints, and conditionals reduce the frequency of user-application interactions, easing the intellectual burden on users and accelerating the debugging process.

Unfortunately, the natural implementation of breakpoints and watchpoints can be unacceptably inefficient. For safety and simplicity, the debugger and the application it controls typically reside in different processes [74]. The breakpoint and watchpoint logic resides in the debugger, necessitating an application-debugger context switch to determine whether session control should be transferred to the user. If control does ultimately transfer to the user, the overhead of the switch becomes irrelevant. Most application-debugger context switches, however, are not masked by user interaction, and their cost is perceived as additional application latency, resulting in substantial overhead (*e.g.*, as high as 40,000 times slowdown in current commercial and open-source debuggers).

We explore injecting debugging logic into the application itself, obviating the need for unmasked application-debugger context switches. Breakpoints and watchpoints are implemented as program transformations. The transformed code identifies necessary transitions to the user, and in such cases (and only in such cases), traps and initiates a context switch to the debugger. To implement a breakpoint, the break instruction is transformed into a trap. To implement a watchpoint, all store instructions (the only type of instruction that can change a the memory-resident variable that appears in an expression) are transformed into a sequence of code that checks whether any of the watched expressions have changed. If any of the watched expressions has changed, then the transformed code executes a trap. To implement a conditional breakpoint or watchpoint, the transformed code also includes the evaluation of the predicate.

All of these transformations avoid costly unmasked context switches to the debugger. Of course, there is the bandwidth cost of the additional instructions. To be sure, this cost increases with the density of breakpoints and the complexity of conditionals, but it remains comfortably lower (by many orders of magnitude) than the cost of any implementation that involves even a minimal amount of unmasked context switching.

Previous proposals used software translation to inject debugging logic into the application [6, 51, 89, 91]. Unfortunately, this approach has major deficiencies. First, it is cumbersome for the debugger implementor because it requires the debugger to perform register scavenging, register re-allocation, and branch retargeting. It is also inefficient because the transformation process contributes to the perceived latency of the debugging session; and the transformed code is bloated, degrading instruction cache performance (although this overhead is certainly lower than that arising from unmasked context switches). Most importantly, injecting debugger code and data into an application violates the separation of application and debugger, allowing a buggy application to corrupt debugger structures or the debugger to perturb application behavior (*e.g.*, by changing the stack-frame layout), potentially resulting in dreaded “heisenbugs.” Alternatively, the hardware itself may be augmented to perform some subset of the debugger’s duties. The challenge here is in defining support that is both efficient and sufficiently flexible to allow arbitrarily complex and many watchpoints and conditions.

This chapter shows that DISE can be used to implement interactive breakpoints and watchpoints, conditional and otherwise, without the above shortcomings [25]. Because DISE has a declarative, simple-to-use interface, implementing the debugger is much less work. DISE specifications are small, simple pieces of code. DISE transformation is also efficient. The overhead of macro-expanding instructions is negligible and DISE has no impact on instruction cache performance. DISE also provides features (*e.g.*, a private register space) that enforce the separation of application and debugger despite the fact that it dynamically intermingles code from each. Finally, because DISE is programmable, debuggers can use DISE with arbitrarily complex breakpoints and watchpoints.

The focus of this chapter is on interactive debugging and on the breakpoint/watchpoint interface presented to the user by existing interactive debuggers. However, DISE can be used in other areas of debugging besides *interactive* debugging. The same techniques we describe can also efficiently implement other debugging interfaces: non-interactive ones like Purify [45] and Valgrind [80] and programmatic ones like iWatcher [97].

This chapter is organized as follows. Section 5.1 gives background on interactive debugging. Section 5.2 describes the use of DISE in implementing efficient breakpoints and watchpoints. Section 5.3 compares the performance of a DISE-based approach to existing implementations. Finally, Section 5.4 discusses other techniques for implementing watchpoints and compares them with our DISE-based approach.

5.1 Interactive Debugging Background

A debugging session consists of three principals: the *application* to be debugged, the *user*, and the *debugger* which serves as a mediator between the two. The user is the slowest party. Breakpoints, watchpoints, and conditionals reduce the frequency of *user transitions*—transitions from the debugger to the user and back—and can dramatically accelerate the debugging process. Conversely, they increase the frequency of *debugger transitions*—transitions from the application to the debugger and back. Debugger transitions that are not masked by corresponding user transitions are perceived as additional application latency.

When user-transition frequency is low (typically a user's goal), the aggregate latency of debugger transitions can dominate execution time. As a consequence, breakpoint/watchpoint implementations can be evaluated by the number of *spurious* (unmasked) debugger transitions they generate. The more spurious transitions, the greater the perceived overhead. There are three types of spurious transitions. *Spurious address transitions* are transitions to the debugger that occur even though watched data is not written, or equivalently no instruction tagged as a breakpoint is executed. *Spurious value transitions* apply to watchpoints only and occur when a variable in a watched expression is updated but the value of the expression is unchanged. The most common cause for this is a *silent store*, which is a store that overwrites a value with the same value [62]. *Spurious predicate transitions* apply to conditional breakpoints and watchpoints and occur when the associated predicate evaluates to false.

Below we summarize well-established implementation techniques employed by widely-used debuggers. Techniques still under active researched are discussed in Section 5.4.

Single-stepping vs. trap-handling. The naïve breakpoint and watchpoint implementation relies on *single-stepping*. The application transfers control to the debugger after every instruction (or source-level statement), and checks whether any of the currently active breakpoints or watchpoint criteria are satisfied before single-stepping to the next instruction. Single-stepping is terribly inefficient, causing many spurious address transitions. Unfortunately, even debuggers that support superior implementations (see below) often resort to it. For example, Microsoft's Visual Studio 6.0 debugger uses single-stepping when watching global variables.

Trap handling is an attractive alternative that avoids many spurious address transitions. The debugger registers a trap handler with the operating system and configures either the application or the processor to generate a trap when an instruction (datum) at a particular address is executed (written). The fast breakpoint and watchpoint techniques that are implemented in modern debuggers all use this approach. Note that while there are straightforward mechanisms for trapping on address-based events, there are no such mechanisms for trapping on events related to values. As a result, trap handling solutions only reduce spurious address transitions. There are no currently used debugger techniques that eliminate spurious value and predicate transitions.

Breakpoint techniques. The standard trap handling solution for breakpoints uses static binary transformation to temporarily replace intended breakpoint instructions with explicit trapping instructions [74]. This implementation has excellent performance characteristics. It induces no spurious address transitions and it degrades application performance only when the breakpoint is encountered. Alternatively, some architectures (*e.g.*, x86, IA-64, PowerPC) provide breakpoint registers. The debugger loads these registers with the addresses of intended breakpoints; the processor traps when an instruction whose PC matches one of these addresses is about to commit. Breakpoint registers are convenient, but typically there are only a few of them. If the number of breakpoints required is larger than the number of hardware registers, the previous techniques are used for the remainder.

Watchpoint techniques. Hardware registers can also be used to implement watchpoints. The debugger loads these with the addresses of the variables in the watched expression, and the processor traps on a store to any of these addresses. For example, GNU's gdb 5.3.90 supports hardware watchpoint registers on Linux/x86 (notice that when setting some watchpoints, gdb prints the message “Hardware watchpoint 1”). Again, the drawback of hardware watchpoint registers is their limited number. IA-32 has four and these also serve as breakpoint registers, IA-64 also has four, PowerPC has one, and some architectures like SPARC and Alpha have none. While four watchpoint registers may sometimes suffice, the user may wish to watch multiple expressions, multiple distinct pieces of data (*e.g.*, appearing in a complex expression or representing a linked data structure), or a single large piece of data like a structure or an array. IA-64 addresses the latter shortcoming by allowing low-order bits to be ignored during matching, letting a single register watch a larger memory segment. However, this is not a general solution.

If the number of watched addresses exceeds the number of hardware watchpoint registers, the virtual memory system can be harnessed to generate traps on writes to certain addresses [5]. Here, the debugger uses an interface like **mprotect()** to remove the write permissions from the page on which the watched address resides. The virtual memory implementation can be used to watch an unlimited number of addresses, but at the cost spurious address transitions. Spatial data locality makes it likely that frequently written non-watched data resides on the same page as watched data.

Virtual memory and hardware registers can easily implement watchpoints provided that all addresses referenced by the watched expression can be statically calculated by the debugger. Addresses generated by indirection (*e.g.*, pointer dereferences or dynamically indexed array elements) cannot be statically determined. To watch an indirect expression ***p**, the debugger could watch the base address **p** then update the ***p** watch condition whenever the value of **p** changes. However, we know of no commercial debuggers that actually implement this. Instead, they resort to (highly inefficient) single stepping. In gdb, for example, a request to watch a pointer variable **p** elicits the message “Hardware watchpoint.” A similar request to watch ***p** yields the message “Watchpoint.”

5.2 Debugging with DISE

The high cost of watchpoints and conditional breakpoints in conventional debuggers is primarily due to the fact that the application and debugger reside in separate processes. Reducing the overhead of these vital primitives in a significant way requires embedding pieces of the debugger—address matching and condition-testing logic—into the application itself. As obviously beneficial as this approach is, existing debuggers do not use it because it is cumbersome (requiring register scavenging, register re-allocation, and branch retargeting), inefficient (due to code bloat), dangerous (because a buggy application may corrupt debugger state), and intrusive (because the extra debugger code may perturb application behavior, *e.g.*, by changing stack-frame layout). DISE-based implementations realize the benefits of injecting debugger logic into the application, without these problems.

In this discussion, it is important to remember that the DISE specifications are automatically generated by the debugger (using templates) in response to the user's setting of breakpoints and watchpoints. We are not relying on the user to manually program the correct specifications, so the debugging session is vulnerable to specification errors to the same extent that it is vulnerable to errors in any other part of the debugger.

5.2.1 Breakpoints

There is little need to use DISE to implement unconditional breakpoints, because the static binary-transformation implementation is straightforward enough and performs well [74]. Nevertheless, we discuss how this can be done in DISE.

The DISE approach parallels the binary transformation approach. The breakpoint instruction is replaced with a DISE codeword. An aware transformation is defined to match and expand on that codeword. The replacement sequence consists of a trapping instruction followed by the original instruction. This implementation is actually more efficient than the conventional rewriting one, because it does not require a three step procedure—restore original instruction, single-step, re-install trapping instruction—to restart the application.

If DISE is extended to support PC matching, then a second approach is possible, which parallels hardware breakpoint registers. The replacement sequence is unchanged, but the pattern matches the instruction's PC. Unlike breakpoint registers, DISE would have no limit on the number of breakpoints. Unfortunately, PC matching would significantly increase the size of each pattern stored in memory and the PT (by 64 bits), and we have not found a strong enough application of it, to merit adding it to DISE.

5.2.2 Watchpoints

Although far more efficient, the DISE watchpoint implementation parallels single-stepping. In DISE, watchpoint specifications match and replace stores. The replacement sequence varies in length depending on the number and complexity of the watched expressions. A naïve specification for watching a single static (at least within the current scope)

Watchpoint Specification

```
T.OPCLASS == store      # match on stores
=> T.INST               # perform store
    ldq $d0,0($dar)     # load watched val
    cmpeq $d0,$dpv,$d0  # comp. w/ old val
    d_bne $d0,1         # same? branch
    trap                # diff? trap to OS
```

(a)

With Conditional Call

```
T.OPCLASS == store       # match on stores
=> T.INST                # perform store
    ldq $d0,0($dar)      # load watched val
    cmpeq $d0,$dpv,$d0   # comp. w/ old val
    d_ccalleq $dhd, $d0  # diff? call handler
```

(b)

With Address Match Gating

```
T.OPCLASS == store          # match on stores
=> T.INST                   # perform store
    lda $d0,T.IMM(T.RB)     # get address
    bic $d0,0x7,$d0         # align address
    cmpeq $d0,$dar,$d0      # comp. w/ old val
    d_ccalleq $dhd, $d0     # diff? call handler
```

(c)

With Isolation

```
T.OPCLASS == store         # match on stores
=> lda $d0,T.IMM(T.RB)     # get address
    srl $d0,11,$d1         # get segment ID
    cmpeq $d1,             # compare to
          $dseg,$d1        #     segment ID
    d_ccalleq $derr,$d1    # diff? call error
    T.INST                 # perform store
    bic $d0,0x7,$d0        # align address
    cmpeq $d0,$dar,$d0     # comp. w/ old val
    d_ccalleq $dhd, $d0    # diff? call handler
```

(d)

Figure 5.1: Example implementations of a single watchpoint in DISE: (a) Naïve, (b) with conditional call, (c) with address match gating, (d) with isolation.

variable consists of the five instructions appearing in Figure 5.1(a): (i) the original store (**T.INST**), (ii) a load of the watched variable from a statically-calculated address stored in DISE register **$dar**, (iii) a comparison of the previous value stored in DISE register **$dpv** and the current value **$d0**, (iv) a DISE intra-instruction branch that skips one instruction if the values match, and (v) a trap. The sequence branches over the trap if the expression does not change in value.

Optimization I: Conditional call/trap. The preceding specification works correctly but often performs poorly. As discussed in Chapter 4, DISE branches by flushing, refetching the original program instruction and re-expanding starting at a new DISEPC. This implies a pipeline flush on every store that does not change the value of the watched expression. As stores typically make up about 10-15% of the dynamic instruction stream, this is a costly proposition.

As stated in Chapter 4, the DISE ISA supports a conditional call (**d_ccalleq** and

d_ccallne), that calls a routine based on the value of a register (*i.e.*, second operand). We can place the trap in a debugger-generated function (**handler**). Figure 5.1(b) shows the optimized specification. If the watched value has changed (*i.e.*, **$d0** is 0), then the instruction **d_ccalleq** calls **handler** (the address of **handler** is stored in DISE register **$dhd**).

We could also add a conditional trap (**d_ctrapeq** and **d_ctrapne**) to the DISE ISA. With a conditional trap, we would avoid executing the call and the debugger would not need to generate **handler**. However, the conditional trap is not a strictly necessary extension since we can achieve the same functionality using a conditional call and a handler function. For this reason, we do not include a conditional trap in the DISE ISA.

Optimization II: Address match gating. Another source of inefficiency in the naïve specification is the load of the watched variable. Loads are expensive because data cache access is high latency and low bandwidth. Replacing every store with a replacement sequence that includes a load—or multiple loads if multiple expressions or complex expressions are watched—increases load port contention and may degrade performance.

The solution to this problem (Figure 5.1(c)) mirrors the virtual-memory and hardware-register techniques. Rather than always re-evaluating the watched expression, the replacement sequence first examines the store address. The expression is only re-evaluated if the store address matches a watched address. The expression re-evaluation is performed in the handler routine rather than inlined into the replacement sequence. Load contention is reduced and performance improved because an expensive load is replaced by a cheaper address comparison.

Unless care is taken, address matching can miss "partial" read/write overlaps, *e.g.*, a long (4-byte) store to the lower half of a watched quad (8-byte) variable. Therefore, when the sizes of the watched and stored data differ, the address of the smaller must be aligned with that of the larger (via logical-bit-clear **bic** in Figure 5.1(c)). For instance, when watching a byte and storing a quad, the watched address is quad aligned. Conversely, when watching a quad and storing a byte, the store address is quad aligned.

With this formulation, the replacement sequence for a store (Figure 5.1(c)) is: (i) the original store, (ii) an ALU operation that re-constructs the store address from the base address register and immediate, (iii) potentially a logical-bit-clear operation to align either the store address or the watched address, (iv) a comparison of this address to the watched

```
# prolog / save regs $1-$4 (not shown)
...
lda $1,glob_ptr
ldq $2,0($1)      # get watch address
ldq $3,8($1)      # get old value
ldq $2,0($2)      # get current value
cmpeq $3,$2,$3    # change?
bne $3,skip       # no, continue
stq $2,8($1)      # yes, update current value
trap              # and trap to debugger

skip:
# epilog / restore regs (not shown)
...
d_ret
```

Figure 5.2: An example **handler()** routine, which evaluates the watchpoint expression.

address which is stored in DISE register **$dar**, and (v) a conditional call to a debugger-generated function (*i.e.*, **handler**), which is rarely taken.

Debugger-generated function. The final watchpoint implementation (above) requires the debugger to dynamically generate a function and add it into the application's text segment. The debugger encodes the address of this function into a DISE register, which is used by the conditional call (for simplicity, we show the name of the function in the conditional call rather than the DISE register). Figure 5.2 shows the function that accompanies the DISE specifications of Figure 5.1(c).

In addition, the debugger utilizes some DISE allocated memory (described in Chapter 3) where it stores watched addresses and current values (to determine when a watched variable has changed). When evaluating expressions, the debugger-generated function indexes this region of memory to access this data.

Protecting both the debugger's embedded data and DISE state. By adding a temporary copy of some of its own data to the debugged application's virtual address space, the debugger makes this data vulnerable to corruption by a buggy application. In addition, DISE specifications, which are stored in memory, are also vulnerable to corruption. Often this is not a problem, as this data is small and the application itself naturally contains no pointers into it. If write access is unnecessary, then we can use the virtual memory system to protect this data. The write permission is disabled for all pages that contain debugger embedded data or DISE state. A write to this memory will result in a trap to the operating system. This approach could work for the specifications and handler routine shown in Figure 5.1(d) and 5.2 with a few minor changes. When the value of the watchpoint is changed, the debugger must update the value rather than the handler routine. Also, some memory, which is writable, must be available to the handler routine for saving the scratch registers.

If the handler or the replacement sequence requires write access to the debugger's data, then the debugger can program DISE to isolate the debugger and DISE state from an errant application memory write. The same specifications that test store addresses against watched addresses can also test them against the debugger's own data region and the region containing DISE specifications (similar to software-based fault isolation [90]). The specification in Figure 5.1(d) augments that in Figure 5.1(c) to branch to an error handler if the store refers to an address in the debugger's 2KB data segment (the 21 high order bits of which are specified by the DISE register **$dseg**). DISE-based protection monitors all executed code (*e.g.*, dynamically generated code or shared libraries), which is not true for previous approaches that statically transform the debugged application [6, 51, 89, 91]. In general, the watchpoint specification may be combined with any other DISE specifications, allowing, *e.g.*, compressed (see Chapter 6) code to be debugged.

Multithreading DISE function calls. A taken DISE function call requires two pipeline flushes, one on the call itself and one on the return. The conditional call instruction means this cost is incurred only when a watched address is written, but the aggregate cost can be high for frequently written watchpoints. We can eliminate this cost by adapting a technique that was previously proposed to reduce the cost of short exception-handling routines like TLB miss handlers, avoiding the pipeline flushes that implement program/exception-handler/program serialization by executing the exception handler on another thread context in a multithreaded processor [98]. We simply execute the body of a DISE-called function on a separate thread. The mechanics of the technique are quite similar to, and actually simpler than, those of the previously proposed mechanism. Modified retirement logic provides global in-order retirement for the now-segmented main application thread and exception-handler/function-body thread. Unlike the previously proposed scheme, which must support precise application-handler communication via the exception registers, a DISE function body only communicates with the application thread via memory. This means that correct data dependences can be established by a simple extension to the function thread's store queue pointer; register renaming is not modified. iWatcher uses a similar technique to reduce the cost of its function calls [97].

Watching multiple addresses. Our optimized replacement sequence matches the current store's address to the watched variable's address as a preliminary test that avoids the potentially more expensive expression value test. For scalar, single-address expressions, the watched address is stored in a DISE register. In general, a user will set watchpoints on multiple or complex expressions that require comparisons of the current store's address to multiple watched addresses.

There are efficient ways of implementing multiple matches. If there are fewer watched addresses than available DISE registers, serial comparison is used. Otherwise, if the watched addresses are in a small range—for instance, the user may be watching a structure, all the elements in a small array, or several nearby variables—the replacement sequence checks the store's address against the upper and lower bounds of the region rather than individual addresses in it. Finally, if the number of watched addresses is both large and sparse, the debugger sets up a watched address bitmap similar to a Bloom filter [12]—in which zeros indicate definite negatives, and ones indicate only probable positives—in its static data region and hashes each store address into this bitmap. The last technique may trigger some spurious calls to the debugger-generated function, but these should be compensated for by the simplified address checking sequence. In general, because the replacement sequence is a piece of software, it may use any address comparison strategy whatsoever.

Pattern matching optimizations. In addition to replacement sequence optimizations, the debugger may also modify the watchpoint pattern specification to trigger on only a subset of the stores. For example, if all of the watched variables are either in the static data segment or the heap, the debugger can choose not to expand stores to the stack by specifying two patterns: a higher-priority pattern for stores to the stack which expands to the original store, and a lower-priority pattern for stores in general which expands to the watchpoint replacement sequence.

The same technique cannot be used if only stack variables are watched because the stack can be accessed indirectly through registers other than the stack pointer. However, the debugger may choose to activate and deactivate the watchpoint expansion when the program enters or leaves the corresponding function's scope. The debugger can set an efficient hook to the scope entry and exit points by setting breakpoints on the function's first and last instructions.

5.2.3 Conditionals

With support for DISE calls to debugger-generated functions, implementing conditional watchpoints is trivial. The condition itself is compiled into the debugger-generated function and guards the trap. The conditional breakpoint implementation is somewhat trickier than the conditional watchpoint case. For conditional breakpoints—which do not require cheap address tests to bypass the more expensive condition test—it often makes sense to compile the condition into the replacement sequence directly. In this case, one or two DISE registers are used as temporaries to evaluate the conditional expression from auxiliary information in the debugger's static data area.

5.2.4 Discussion

DISE is a less cumbersome, less intrusive, safer, and better performing form of binary transformation. With the help of DISE, the debugger does not need to modify the application binary, except in two well-defined and simple ways, *i.e.*, appending a dynamically-generated function and small data region to the application's text and data segments, respectively. All would-be modifications to existing code are performed via DISE transformations, transparent to the application itself, leaving the application statically unchanged and unbloated. DISE is also far less intrusive than any approaches that transform the static image of the program. It does not change the placement of any code or data, and it does not impact many hardware performance counters. Finally, the DISE mechanism itself can also be used to ensure that debugger structures embedded in the application are protected from buggy applications.

On the other hand, hardware-assisted debugging (*e.g.,* via watchpoint registers) can have nil overhead if there are no spurious value or predicate transitions, that is if all breakpoints and watchpoints are unconditional and if all writes to watched addresses also change values of the corresponding expressions. While DISE's overhead is not zero in this scenario, it is low as we will see in the next section. DISE's advantage over hardware-assisted debugging manifests if writes to watched addresses do not always change values of watched expressions or if conditional breakpoints and watchpoints are used, in which case DISE's small, constant overhead will be smaller than that resulting from expensive spurious transitions.

5.3 Evaluation of DISE Debugging

We use cycle-level simulation to measure the overhead of the DISE implementation of watchpoints and compare it to the overhead of four existing watchpoint implementations: source statement single-stepping, trap handling based on the virtual memory system, trap handling based on hardware watchpoint registers, and static code transformation via binary rewriting. Our experiments focus on conditional and unconditional watchpoints. Unconditional breakpoints have a widely-used "ideal" implementation via static binary transformation. Conditional breakpoints exhibit cross-implementation performance trends relative to unconditional breakpoints that are similar to the trends exhibited by conditional watchpoints relative to unconditional ones. We experiment with several different kinds of watchpoints—scalar, array, and complex expression—and also compare the mechanisms based on their effectiveness at supporting multiple watchpoints. Finally, we perform a sensitivity analysis on the DISE implementation, evaluate the benefit using multithreading to lower the DISE overhead, and measure the cost of using DISE to protect both debugger and DISE state stored in the application's address space.

Simulator. Our general methodology is described in Section 4.2 of Chapter 4. We simulate using SimpleScalar Alpha [16], modeling the machine from Table 4.1 on page 54. We use a 2-stage decoder-based implementation of DISE with a 32-entry pattern table and a 512-entry replacement table (see Chapter 4).

	function	instructions executed	IPC	store density
bzip2	generateMTFValues	1,828,109,152	2.45	19.8%
crafty	InitializeAttackBoards	18,546,482	2.39	10.8%
gcc	regclass	18,016,384	1.90	9.68%
mcf	write_circs	1,847,332	0.33	16.2%
twolf	uloop	2,336,334	1.87	13.7%
vortex	BMT_TraverseSets	205,690,692	2.25	17.6%

Table 5.1: Benchmark summary.

	HOT	WARM1	WARM2	COLD	INDIRECT	RANGE
bzip2	24805.7	193.4	∼0	0	24805.7	193.4
crafty	6531.4	3308.4	6.7	.4	6531.4	72.8
gcc	454.8	223.7	.2	.1	454.8	8197.9
mcf	11229.8	1168.4	215.4	0	11229.8	0
twolf	1467.4	227.5	101.4	80.8	1467.4	250.6
vortex	7290.3	27.6	27.6	∼0	7290.3	.4

Table 5.2: Watchpoint write frequency (per 100K stores).

Benchmarks and watchpoints. We perform our experiments on the SPEC2000 integer benchmarks, which we compiled for the Alpha EV6 using GNU gcc 3.2 with the debugging-appropriate optimization flags *-O0 -g* (although the topic of debugging optimized code is an interesting one, it is beyond the scope of the present work). For each benchmark, we used the GNU *gprof* profiler to identify long-running functions. From this list of functions, we selected one function per benchmark that is also statically large. For each benchmark, Table 5.1 gives properties of the functions we use. We simulate each function in its entirety.

We use a combination of source-code inspection and profiling to select six watchpoints for each benchmark. The first four watchpoints are scalar variables (two heap and two locals) whose written-to frequency ranges from frequently (HOT) to occasionally (WARM1/WARM2) to rarely (COLD). The fifth is a dereference (INDIRECT), and the sixth is a non-scalar, like a structure or an array (RANGE). INDIRECT actually refers to the same storage as HOT, but through a pointer. The use of multiple watchpoints allows us to measure debugging overhead under a range of conditions, but they consume presentation space. To compensate, we show only a representative subset of the SPEC2000 benchmarks. Table 5.2 shows the written-to frequency of each watchpoint, normalized by

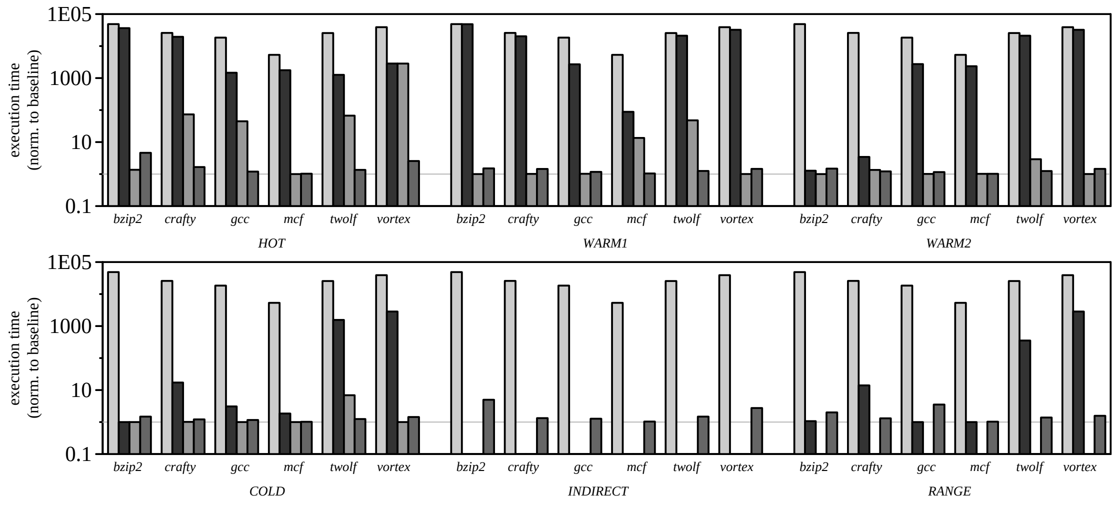


Figure 5.3: Comparison of four unconditional watchpoint implementations.

the total number of stores.

Methodology. Presenting results that can be meaningfully interpreted and easily compared requires that we: (i) simulate the same number of instructions for each experiment, (ii) factor "user latency" out of our measurements, and (iii) realistically model the cost of debugger transitions and the debugger itself even though our simulator executes only user-level code.

We satisfy these requirements by modeling user transitions and their accompanying debugger transitions (*i.e.*, non-spurious debugger transitions) as having zero cost. We model the cost of spurious debugger transitions by flushing the pipeline and stalling for 100,000 cycles. This figure is a conservative estimate of the actual cost as measured in existing debugger implementations. Using the IA-32 cycle-level timer—via the **rdtsc** instruction—we measure the round-trip debugger transition latency for two debuggers: GNU's gdb 5.3.90 under Linux and Microsoft's Visual Studio 6.0 under Windows XP. On a 3 GHz Pentium 4, this latency is 290,000 cycles for gdb and 513,000 cycles for Visual Studio.

5.3.1 Unconditional Watchpoints

Figure 5.3 presents debugging overhead—execution time relative to an undebugged application—for a single unconditional watchpoint.

DISE. The DISE implementation dynamically inserts three or four instructions (depending on the data sizes of the watchpoint and store instruction) after every store, regardless of its address. While these increase the dynamic instruction count by as much as a factor of three, performance overhead rarely exceeds 25%. The added instructions are all ALU instructions and they do not add to the application's critical path. Nevertheless, overhead can be high for frequently written watchpoints (*e.g.*, HOT/*bzip2* and HOT/*vortex*), requiring frequent flushing when the expression-evaluation function is called. HOT/*mcf* is also frequently updated, but its cost is masked by the memory latency which dominates this benchmark.

Single stepping. Single-stepping is clearly the worst performing implementation, producing slowdown factors of 6,000 to 40,000 times in many cases. These figures are consistent with the observed performance behavior of real debuggers.

Virtual memory. The virtual memory implementation has almost no overhead for some watchpoints (*e.g.*, COLD/*bzip2*), but for many others (*e.g.*, COLD/*twolf* and COLD/*vortex*) its overhead can be quite high, sometimes equaling the slowdown of single stepping (*e.g.*, WARM1/*bzip2*). This erratic behavior is due to the coarse (page) granularity of the address-matching mechanism and the frequency with which unwatched addresses that reside on the same page as a watched address are written. If most writes to the page are to the watched address, perceived overhead is low. Conversely, if a watched address shares a page with unwatched, frequently-written addresses, many spurious address transitions will result and overhead will be high. Certainly, page size can impact the number of spurious transitions, with smaller pages producing fewer. Our page size is 4KB, on the small end for real systems. Our experiments (not shown) indicate that reasonable overhead is achieved for these watchpoints only for impractically small page sizes (*e.g.*, 128 bytes).

Finally, notice that there is no virtual memory experiment for the INDIRECT watchpoint. The debugger cannot statically determine what pages to write-protect for a watchpoint expression containing pointer dereferences because the value of the pointer may change during execution. It is possible to watch the pointer itself and dynamically update the page protection for the datum to which it points (in which case the overhead would be similar to the HOT case), but we are aware of no debugger that does this.

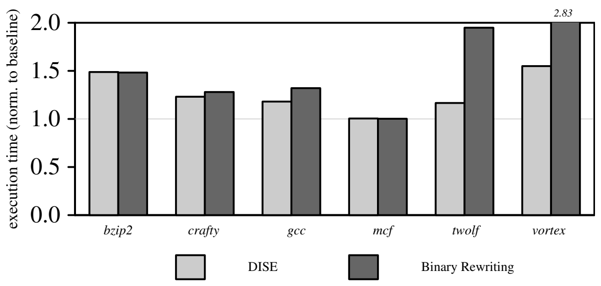


Figure 5.4: Comparison to binary rewriting.

Hardware watchpoints. Unlike virtual memory watchpoints, hardware register watchpoints are quad-granularity and only result in spurious address transitions when a partial quad is watched and a different part of the same quad is written. Unfortunately, hardware watchpoints are still susceptible to spurious value transitions caused by silent stores. If these occur with any frequency, performance can be significantly impacted. For example, in all HOT benchmarks—save *bzip2*—50% or more of all stores to the watched address do not change the data value, resulting in significant perceived slowdowns. This is a realistic scenario, because silent stores are common [62] and watchpoints are appropriate for determining exactly where such data are actually changed.

Like virtual memory, there is no hardware register experiment for the INDIRECT watchpoint, because a debugger cannot statically determine the address to monitor. In contrast with virtual memory, there is also no experiment for the large watchpoint RANGE. Hardware registers are principally used to watch scalars. For non-scalars like structures and arrays, real debuggers resort to using virtual memory or single-stepping. Some hardware watchpoint register implementations (*e.g.*, IA-64) allow larger segments of memory to be monitored by masking low-order bits during address comparison, but this may result in spurious address transitions.

Static transformation. Watchpoints may also be implemented via binary rewriting [89, 91]. In Section 5.2 we argued that this approach is cumbersome, intrusive, and dangerous; but it is also inefficient, both in terms of the startup cost to perform the transformation and the instruction cache cost, which we illustrate here. Figure 5.4 gives the COLD watchpoint overhead of a binary-rewriting-based watchpoint implementation in which the

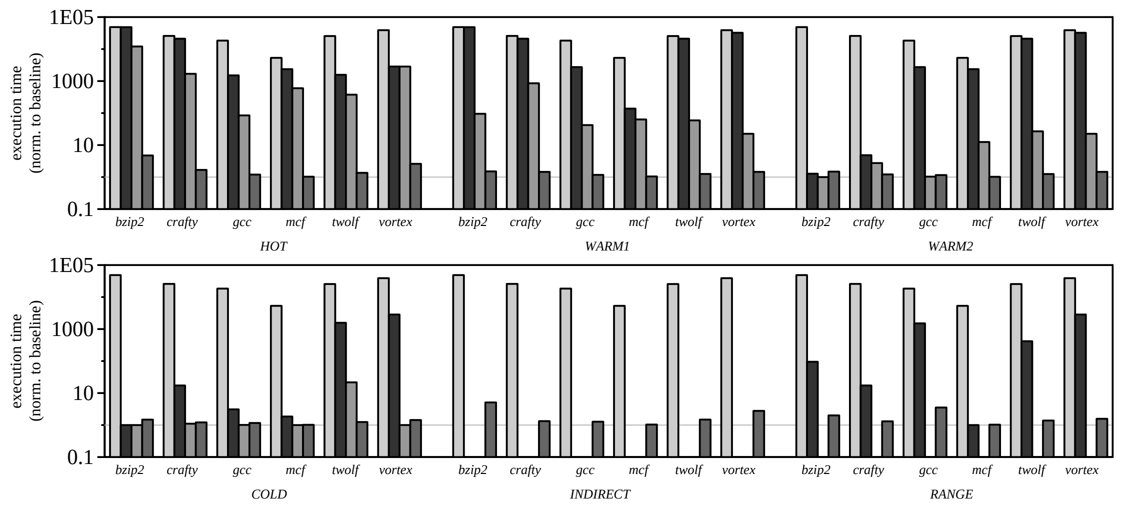


Figure 5.5: Comparison of four conditional watchpoint implementations.

code of Figure 5.1c is statically inlined at every store (*i.e.*, no static optimization is performed). Note that this graph does not include the additional overhead of performing the static transformation. We examine COLD watchpoints because they represent a common usage scenario and highlight the difference between DISE and binary rewriting. Both presented implementations have comparable performance (ignoring static transformation cost) for benchmarks with small instruction memory footprints (*e.g.*, *bzip2*, *crafty*, and *mcf*). For larger programs (*e.g.*, *gcc*, *twolf*, and *vortex*) the additional instructions in the static image degrade instruction cache performance considerably. We exclude figures for binary-rewriting-based implementations from our other graphs because these results are governed by code size and instruction cache performance, *not* watchpoint characteristics.

Summary. For single, unconditional watchpoints, DISE has low overhead, generally 0–25%. It also significantly outperforms virtual memory on all indirect watchpoints and direct ones that share pages with unwatched, frequently-written data. It outperforms hardware debugging registers for large and indirect watchpoints and watchpoints with a non-negligible number of silent stores. It is comparable to a binary rewriting implementation for codes with small instruction working sets and superior otherwise.

5.3.2 Conditional Watchpoints

The performance benefits of DISE are even more pronounced for conditional watchpoints. Of the four implementations, it is the only one that can avoid spurious predicate transitions by evaluating conditions in the application itself. Figure 5.5 compares the overheads of the four implementations on single, conditional watchpoints. Aside from the condition, these are the same watchpoints used in the previous experiment. To model a realistic condition which significantly reduces the number of user transitions, our predicate compares the value of the watched expression to a constant it never matches.

The use of conditionals does not change DISE's relative advantage over single-stepping. Conditional or not, single-stepping incurs a debugger transition on (approximately) every store while DISE incurs transitions only when they lead to user transitions.

DISE's overhead as compared to that of the virtual-memory and hardware-register implementations depends on the frequency with which the watchpoint address is written. DISE adds a small fixed amount of overhead per store, regardless of its address. Virtual memory (modulo false address positives) and hardware registers add a much higher cost, but one that is proportional to the number of writes to the watched address. For infrequently-written watchpoints (*e.g.*, COLD/*bzip2* and COLD/*gcc*), they trigger few spurious predicate transitions and slightly outperform DISE. HOT watchpoints and many WARM watchpoints trigger many spurious predicate transitions, making DISE's constant low overhead seem insignificant by comparison.

We can compute the rough store frequency crossover point from the ratio of the cycle cost of DISE replacement sequence to the cycle cost of a debugger transition. Let us assume that DISE watchpoints add one cycle per store and that a debugger transition costs 100,000 cycles. Hardware registers and virtual memory will have lower overheads on conditional watchpoints whose addresses are written to by fewer than one of every 100,000 stores (less than 1 in Table 5.2). Otherwise, DISE will have the advantage. From Figure 5.5 it is clear that DISE is always competitive with and usually superior to the alternative implementations of conditional watchpoints.

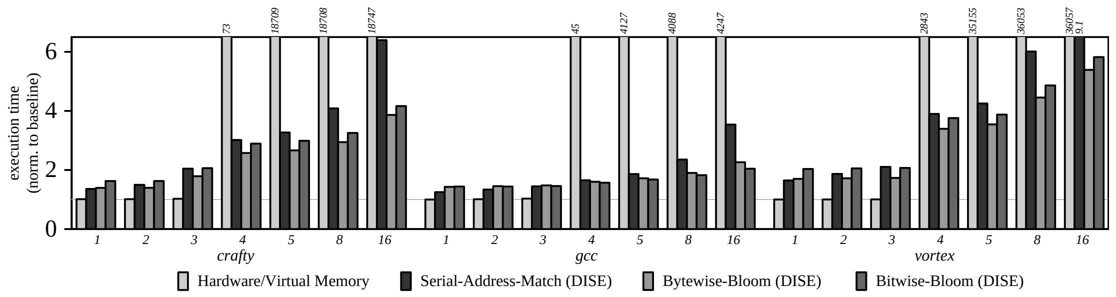


Figure 5.6: Performance impact of the number of watchpoints.

5.3.3 Number of Watchpoints

With respect to performance, DISE faces strong competition in only limited scenarios. For unconditional scalar watchpoints (admittedly the most common kind) it may be outperformed, albeit not significantly so, by a hardware register mechanism. However, even here DISE has an advantage in that it can easily support multiple watchpoints with constant low overhead, while the hardware mechanism is limited by the number of watchpoint registers.

In Figure 5.6, we vary the number of watchpoints for both DISE and a hardware register mechanism. The hardware mechanism uses virtual memory for every watchpoint after the fourth. For DISE, we examine three replacement sequence implementations. *Serial-Address-Match* matches each address serially. *Bytewise-Bloom* hashes store addresses to bytes in a 2KB array, similar to a Bloom filter [12]; a byte value of 1 indicates a probable match and triggers a DISE function call; false positives impact performance but not correctness. *Bitwise-Bloom* hashes quad addresses to bits, increasing effective array size by a factor of eight. This results in fewer false positives, but requires two extra bit-manipulation operations to access the array.

As long as it can use hardware registers and not fall back to virtual memory—*i.e.*, there are four or fewer watchpoints—a hardware mechanism will often slightly outperform any DISE implementation. Again, a large number of silent stores on any of the watchpoints can change this dynamic, as is seen for *vortex* at four watchpoints. Once virtual memory must be used, however, a single watchpoint that occupies the same page as unwatched, frequently-written data will cause spurious address transitions to spike along with overhead. With multiple watchpoints, the probability that such a watchpoint is included in the set is high. For 5, 8 and 16 watchpoints, all three DISE implementations outperform the hardware mechanism by at least three orders of magnitude.

Note that our experimental methodology, which discounts user transitions and their accompanying debugger transitions, results in some anomalous-looking—but not actually anomalous—virtual memory results. When going from five watchpoints to eight on *gcc*, the slowdown drops from 4127 to 4088. One of the three new watchpoints resides on the same page as the fifth watchpoint. With only five watchpoints, writes to this address trigger spurious address transitions. When this address is watched, the transitions triggered by its writes are no longer considered spurious, and they are assigned no cost in our experiment.

Across the DISE implementations, the dominant effect is the efficiency of the replacement sequence. For one or two watchpoints, *Serial-Address-Matching*—which avoids costly loads—is the best approach. However, the length of this replacement sequence increases linearly with the number of watchpoints. Despite the fact that it contains a load, the constant length Bloom filter replacement sequences are more efficient for three or more watchpoints. The bytewise Bloom filter performs better then the bitwise version in almost all cases, as the shorter replacement sequence compensates for the cost of a few additional false positives (each of which incurs two pipeline flushes and the execution of a short function). The exception is *gcc* where the bitwise filter outperforms the bytewise one for three or more watchpoints. Here false positives dominate. For 16 watchpoints, the bytewise filter incurs 30,000 false positives (with 40,000 true hits) as compared to 100 false positives for the bitwise filter. The important point is that for a (relatively) large number of watchpoints *any* DISE approach is superior to a hardware-register/virtual-memory combination. Furthermore, the DISE implementations are much less data dependent (*i.e.*, they have good *and* predictable performance).

5.3.4 Implementation Effects

Below we evaluate the performance impact of several variations of the DISE watchpoint implementation.

Varying ISA support. Our DISE experiments to this point use a replacement sequence that contains a cheap address check and a conditional DISE call. Here we investigate

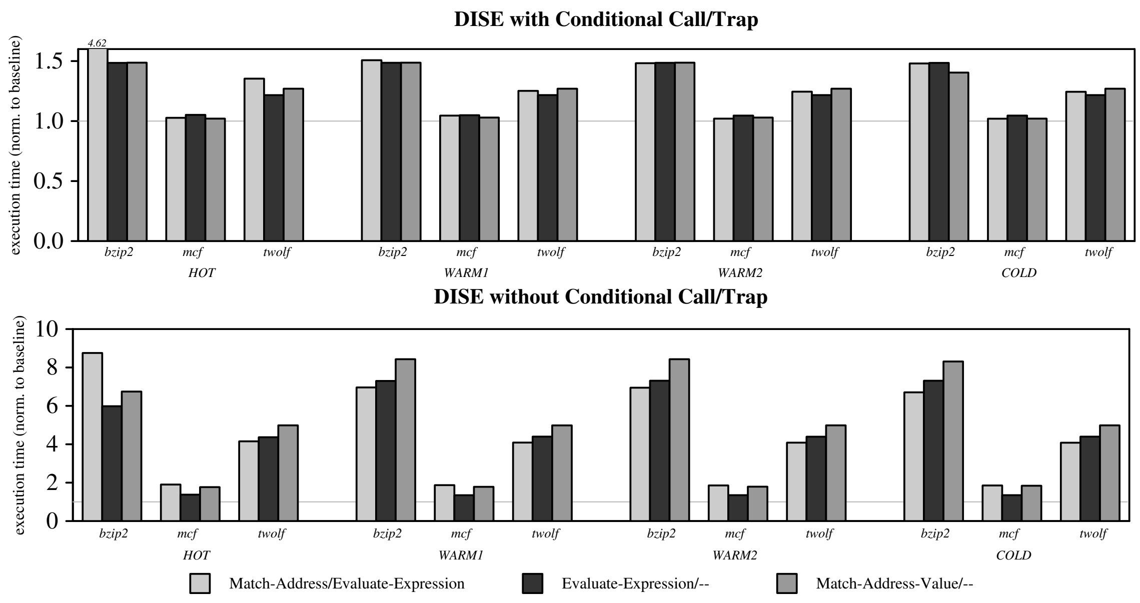


Figure 5.7: Overhead of various DISE watchpoint implementations.

the impact of the conditional call instruction and the call itself in the context of debugging (note in Chapter 4, we evaluated the impact of the conditional call on store address tracing).

Figure 5.7 shows the unconditional watchpoint overheads of six different versions of the DISE replacement sequence/function combination. The six versions are divided into two groups. In the top group, conditional calls and traps are used to avoid common-case pipeline flushes. In the bottom group, these instructions are not available and the same functionality is instead implemented using a combination of conditional branch and unconditional call/trap, which elicits flushes in the common case. Within each group, three alternative implementations are presented. *Match-Address/Evaluate-Expression* matches addresses in the replacement sequence and calls a function to re-evaluate the expression on a match (Figures 5.1c and d). This has been our default. *Evaluate-Expression/–* evaluates the expression in the replacement sequence directly (Figures 5.1a and b), forgoing the address match and obviating the need for a function call. *Match-Address-Value/–* matches the store's address to the watched address and its value to the previous value of the expression. This is tantamount to evaluating the expression without the cost of a load if (i) the watched expression is a scalar, and (ii) the data size of the watched scalar and the store are the same (*e.g.,* both quads or both bytes).

Not surprisingly, the unavailability of conditional calls and traps (bottom graph) results in considerably higher overhead, regardless of the replacement sequence/function organization. This result confirms what we saw in Chapter 4 with store address tracing. The lesson is clear: intra-instruction control transfers (*i.e.*, transfers within the same replacement sequence) should be avoided even at the expense of executing more instructions.

When conditional calls and traps are available (top graph) and the number of pipeline flushes is kept to a minimum, second order effects can be observed. For instance, with frequent flushing, the *Evaluate-Expression/–* implementation often has the highest overhead even though it executes the fewest additional instructions. The key is that one of the added instructions is a load, and load bandwidth is often highly contended. *Match-Address-Value/–* often has the lowest overhead, requiring neither pipeline flushes nor replacement sequence loads. Unfortunately, this implementation can only be used in select cases.

There are exceptions, however, arising from the trade-off between the cost of a load and the cost of an address-match-induced function call. For instance, for frequently-written watchpoints, *Evaluate-Expression/–* (despite its load) can be more efficient than *Match-Address/Evaluate-Expression*. This is the case for HOT/*bzip2*, which under *Match-Address/Evaluate-Expression* triggers a function call on 25% of all stores, resulting in a slowdown factor of 4.62. For watchpoints written this frequently, direct expression evaluation in the replacement sequence is a better alternative.

Except in extreme cases like the one described above, DISE implementations are not particularly sensitive to the frequency with which a variable in a watched expression is updated. Again, DISE is a form of ultra-lightweight single-stepping. Like single-stepping, it has roughly constant overhead. The difference is that this overhead is quite low. It is important to note that the overhead of all of these alternatives (even the worst among them) is orders of magnitude lower than worst-case overhead for each of the other non-DISE implementations.

Exploiting multithreading. A major component of the overhead of DISE comes from the pipeline flushes necessary to call and return from a function from within a replacement sequence. Figure 5.8 shows the benefit to DISE of the multithreading optimization described in Section 5.2. Watchpoints with relatively little overhead (*e.g.,* most of the WARM and

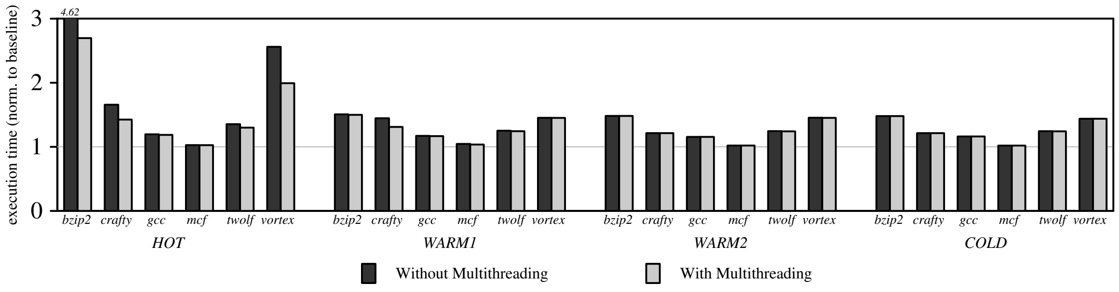


Figure 5.8: Overhead of multithreaded DISE watchpoints.

COLD ones) benefit very little, because the flushing cost is already a minor performance factor. The HOT watchpoints that have many address matches (resulting in many function calls) naturally exhibit a significant reduction in the overall overhead (by nearly a factor of two for *bzip2*).

Protecting debugger structures. A virtue of a DISE-based implementation of watchpoints is that debugger logic is *dynamically* embedded into the running program, preserving the logical separation of the application and debugger. Unfortunately, an errant program can still corrupt the debugger data structures (*e.g.*, the Bloom filter). In addition, the buggy application could corrupt DISE itself, since the specifications are stored in memory. If we the DISE replacement code or handler routines do not need write access to this memory, then we can use the virtual memory system to protect it (in which case there is almost no overhead). If write access to this data is required then we can solve this problem by modifying our DISE transformation to check the legality of addresses referenced by all store instructions (as described in Section 5.2). Figure 5.9 plots the overheads of watching a COLD expression with and without protecting debugger data structures. We evaluate COLD expressions in order to illustrate the maximum additional cost, for the overhead of hotter watchpoints would mask the additional address-checking overhead. Nevertheless, the protection contributes only a modest additional overhead.

5.4 Related Work in Debugging

Several lines of research relate to debugging with DISE.

Embedding debugging logic into the application. The high cost of context switching

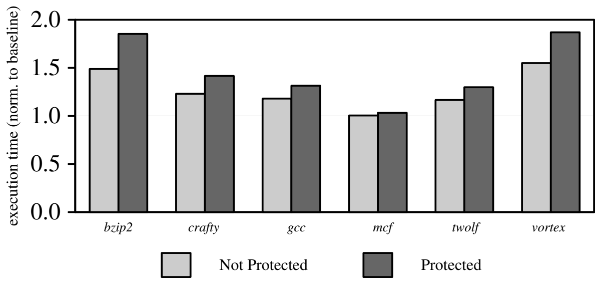


Figure 5.9: Performance impact of protecting debugger and DISE state.

that results from keeping the debugger and debugged application in separate processes has been observed numerous times. Several systems (propose to) move some debugging logic into the debugged application's process to reduce the number of context-switches. In Parasight [6], the debugger shares an address space with a shared-memory parallel application. Kessler moves debugger logic into a serial application to reduce the cost of conditional breakpoints [51]. An intended conditional breakpoint is replaced with a jump to a custom code snippet that evaluates the condition before trapping to the debugger or jumping back to the application's original control path. Wahbe *et al.* extend this work to include watchpoints [89, 91] by replacing stores with calls to an address matching routine. Unlike Kessler's system, which uses simple in-place rewriting, Wahbe's requires—and benefits from—wholesale re-compilation in order to prune unnecessary calls. Re-compilation cost is high for large applications, but individual functions may be re-compiled on demand using just-in-time infrastructures like DELI [33] or Dyninst [15].

Our implementation follows the embedding approach, but has several important advantages. These derive from DISE's advantages as compared to traditional static rewriting. First, DISE is much less intrusive than static rewriting. The presence of DISE registers means that there is no need to scavenge registers from the application, and the fact that instructions are expanded after fetch means that there is no need to retarget application branches around inserted code and that inserted code does not reduce effective instruction cache capacity; rewriting systems that insert code out-of-line using "trampolines" [51] eliminate the need to retarget branches but still require register scavenging and expand the instruction footprint. Similarly, watchpoints and breakpoints can be enabled and disabled quickly by activating and de-activating the proper DISE transformation specifications, without modifying the executable. Non-intrusiveness begets safety. Because they primarily use different register and PC spaces, the application and debugger are less likely to interfere with each other than they would if combined statically. The DISE mechanism can also be used to ensure that application stores do not corrupt debugger structures.

Reducing context-switch cost. An alternative to eliminating context-switches is to reduce their cost. Thekkath and Levy propose hardware modifications that allow traps to vector directly into user code [86].

Valgrind. Valgrind is a popular tool that has been applied to profiling and debugging x86 programs [80]. Valgrind is a basic-block interpreter/dynamic compiler with an instrumentation interface similar to those supplied by static rewriting packages like Atom [82], EEL [57], and Etch [73]. Non-interactive (and we suppose interactive) debugging features can be implemented in Valgrind by registering the appropriate instrumentation functions. Unlike a conventional interactive debugger, Valgrind forces the user to write debugging code. Unlike our DISE implementation, its performance is quite poor [97]. Even without instrumentation it induces slowdown factors of four; basic instrumentation—like instruction counting—can increase this factor to 25. In addition, the Valgrind runtime system perturbs much of the processor state, including registers, caches, and hardware performance monitors.

iWatcher. iWatcher [97] is a recently-proposed hardware-assisted debugger. There are two aspects to iWatcher. The first is a programming interface for registering with the processor pairs of "interesting" memory regions and fixed-interface callback functions; when a program writes to (or reads from) a registered memory region, the processor arranges for the registered function to be called with arguments describing the access supplied by the hardware. The second is hardware support for efficiently executing this interface, including a hierarchal implementation of a memory region tracking table and an adaptation of thread-level speculation for serializing the function call within the execution of the program at low cost (our multithreading technique is a lighter-weight version of this). The work in this chapter relates primarily to the implementation aspect. Here, while iWatcher relies primarily on "hardware," *i.e.*, tables and comparators, DISE provides the same support using what is in effect lightweight software, *i.e.*, injected instructions. We could easily replace the iWatcher implementation with DISE—(almost) anything one can do in hardware can also be done in software—with comparable performance. The iWatcher implementation would have a slight performance advantage for infrequently-modified watched regions as DISE's instruction overhead (while low) may still be noticeable. For more frequently-modified watched regions, the DISE implementation would have an advantage because DISE can prune many spurious value and predicate transitions without making a function call whereas iWatcher cannot. DISE has the additional advantage of not being debugging specific.

5.5 Summary

The conventional implementation of debuggers—as processes separate from the debugged application—makes the implementation of breakpoints and watchpoints costly. The typical debugging session will contain many expensive application-debugger context switches that do not ultimately transfer session control to the user but are necessary to evaluate expressions and predicates in the debugger. These can slow down the application by factors of 40,000 or more.

In this chapter, we propose avoiding expensive and unnecessary context-switching by embedding the debugger's breakpoint, watchpoint, and conditional logic into the application itself. Most debuggers avoid this approach because it is practically cumbersome, has the high initial overhead of analyzing and transforming the application, may introduce "heisenbugs," and is potentially unsafe. The novel aspect of our proposal is that we perform the embedding without these problems using DISE, which (efficiently) transforms an application's dynamic instruction stream rather than its static image. We find that for most watchpoints and all conditional breakpoints and watchpoints, DISE's performance advantage is significant. Its slowdown versus undebugged code is usually less than 25% and is always modest, while that of a conventional debugger can be four orders of magnitude worse.

Chapter 6

Code Compression with DISE

As we have seen in previous chapters, a virtue of DISE transformation is that it does not bloat the instruction cache or memory. Instructions are macro-expanded after they are fetched from memory. A natural application of this design is code compression. Frequently-occurring instruction sequences are stored in DISE replacement sequences and replaced in the program with compressed codewords. At runtime, DISE macro-expands (*i.e.*, decompresses) the codewords back to the original sequences.

There are many benefits of code compression. First, code compression reduces the memory footprint, and in doing so, effectively increases the sizes of memory and the instruction cache. This is particularly important for embedded devices where cache and memory size are limited. Second, because processor designers can use smaller memory structures to achieve the same performance, code compression can also help to reduce power, a growing concern for both embedded and general-purpose systems. Alternatively, code compression reduces instruction cache misses, and can be used as a performance enabler.

This chapter demonstrates a DISE implementation of code compression. We leverage an existing technique called *dictionary-based code compression* [59]. A static binary rewriter compresses a program by replacing all frequently-occurring code sequences with DISE codewords. The original sequences are placed in the *dictionary*, which with our DISE-based approach, is a set of replacement sequences. At runtime, DISE expands (*i.e.*, decompresses) each fetched codeword into the original instruction sequence. Although DISE is used only in decompression, we use the terminology DISE-based compression to refer to static compression via a binary rewriter coupled with dynamic decompression via DISE.

DISE is particularly appropriate for dynamic code decompression because it enables parameterized compression, an extension to conventional compression that allows multiple, similar-but-not-identical compressed sequences to share dictionary entries, improving dictionary space utilization. Parameterization also allows PC-relative branches to be included in compressed instruction sequences. In addition, DISE's programmability allows the dictionary to be customized on a per application basis, further improving compression. Finally, as a general purpose mechanism, DISE can implement many other features and even combine them (dynamically) with decompression.

This chapter shows that DISE-based compression enables code size reductions of over 35% and performance improvements of 5–20%. Parameterized decompression—a feature unique to a DISE implementation of hardware decompression—accounts for a significant portion of total compression. This chapter also shows that dictionary programmability is an important consideration for dynamic decompression, as well as how to reduce the overhead of application-customized dictionaries. Although previous dynamic code decompression proposals do not preclude programmability and may even assume it, none evaluates its importance or provides a mechanism for its implementation (more on other techniques in Section 6.4). Finally, this chapter shows that DISE-based compression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%.

This chapter is organized as follows. Section 6.1 gives background on dynamic code decompression. Section 6.2 presents our implementation of DISE-based compression, and Section 6.3 evaluates this implementation. Section 6.4 discusses some related work in code compression.

6.1 Dynamic Code Decompression Background

Dynamic code decompression techniques are characterized by *when* they perform decompression: between memory (or an L2) and the instruction cache (*i.e.*, *fill-path* decompression) or after instructions are fetched in the processor (*i.e.*, *post-fetch* decompression).

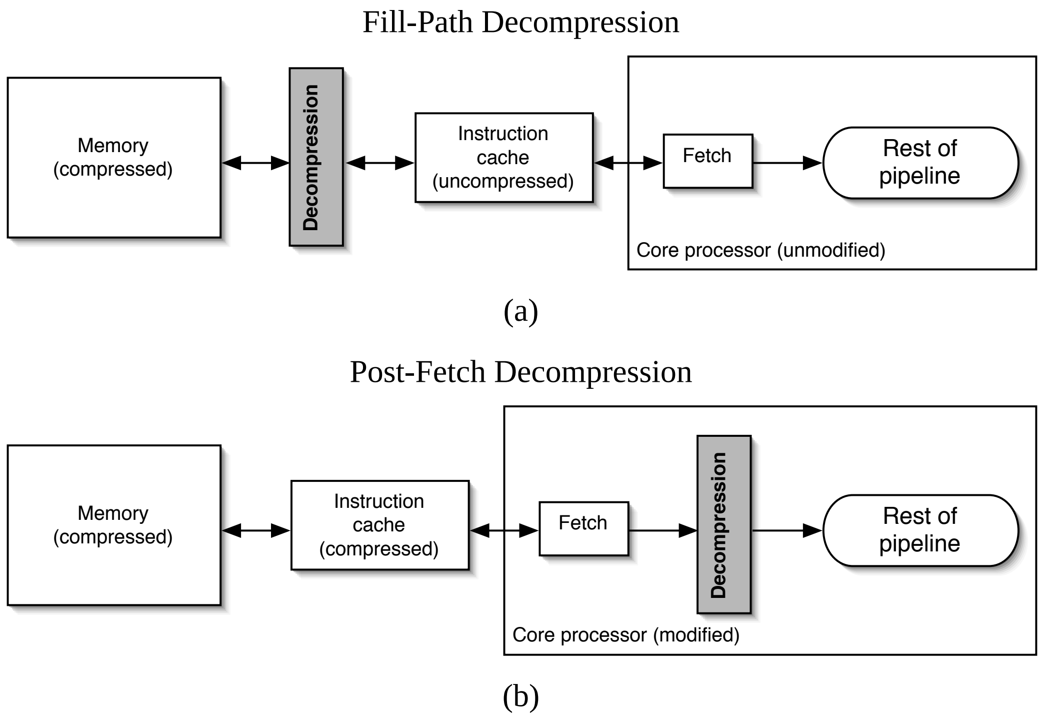


Figure 6.1: Two implementations of dynamic code decompression: (a) fill-path and (b) post-fetch.

Both approaches are illustrated in Figure 6.1.

Fill-path decompression. Several systems integrate decompression into the instruction cache fill path [50, 94]. This approach is illustrated in Figure 6.1(a). The advantages of fill-path decompression are that it allows the use of unmodified cores while incurring the decompression penalty only on instruction cache misses. Because instruction cache misses are rare for most workloads, longer decompression latencies can be tolerated. Therefore, more effective compression algorithms (*i.e.*, Zempel-Liv) with longer decompression latencies can be used. The disadvantages of fill-path decompression are that it stores uncompressed code in the instruction cache and requires a mechanism for translating instruction addresses from the uncompressed image (in the instruction cache and pipeline) to the compressed one (in memory).

Post-fetch decompression. An alternative approach, shown in Figure 6.1(b), decompresses instructions after they are fetched from the cache but before they enter the execution engine [59]. Post-fetch decompression requires a modified processor core and an ultra-efficient decompression implementation, because every fetched instruction must at the very least be inspected for possible decompression. However, it allows the instruction cache to store code in compressed form and eliminates the need for a compressed-to-decompressed address translation mechanism; only a single static version of the code exists, the compressed one.

DISE decompression. DISE, which is implemented in the front-end stages of a processor, is well-suited as a post-fetch decompressor. At runtime, DISE macro-expands compressed codewords into the original instruction sequence. As shown in Chapter 4, the cost of macro-expanding (*i.e.*, decompressing) instructions is negligible, a necessary property of post-fetch decompressors. In the next section, we describe how we achieve decompression via DISE.

6.2 DISE-Based Code Compression

DISE enables an implementation of dictionary-based post-fetch decompression that is functionally similar to a previously described scheme [59]. The DISE implementation is unique among hardware decompression schemes in that it supports parameterized decompression, has a programming interface that allows program-specific dictionaries, and uses hardware that is not decompression-specific. We elaborate on how DISE may be used to perform dynamic decompression and present our compression algorithm.

6.2.1 Dynamic Decompression

A DISE decompression implementation stores each dictionary entry in a separate DISE replacement sequence. Decompression is an aware transformation. A DISE-aware compressor replaces frequently-occurring instruction sequences with DISE codewords, which are recognized by their use of a single reserved opcode. DISE decompression uses explicit tagging; a single pattern matches all decompression codewords via the reserved opcode, and the codeword itself encodes the replacement sequence identifier (*i.e.*, *tag*). This arrangement is basically the same as the one used by a previously described scheme [59]. However, to support parameterized decompression, DISE also uses some non-opcode bits of a codeword to encode register/immediate parameters.

```
T.OP == res2                 # match reserved opcode
=>                           # (uses explicit tagging)
0:  lda $T.P1,T.P2($T.P1)    # Sequence 1
    ldq $T.P3,0($T.P1)       # (when T.TAG == 0)
    addq $T.P3,$7,$T.P3
3:  and $T.P1,T.P2,$T.P1     # Sequence 2
    cmpeq $T.P3,$T.P1,$6     # (when T.TAG == 3)
```

(a)

```
                               and $4,8,$4
res2 3 $4,8,$5  ──────────►    cmpeq $5,$4,$6
```

(b)

Figure 6.2: DISE specification for decompression: (a) specification and (b) example transformation.

Figure 6.2 shows a decompression specification (a) and an example transformation (b). The pattern in Figure 6.2(a) matches on the reserved opcode **res2**. A trigger instruction can expand to one of two possible replacement sequences. Recall from Chapter 3 that each distinct replacement instruction is annotated with its tag (a number followed ":"). In Figure 6.2(a) there are two replacement sequences, one starting at index 0 (*i.e.*, **T.TAG==0**) and the other starting at index 3 (*i.e.*, **T.TAG==3**). In Figure 6.2(b) the tag is 3, so the trigger expands to the second replacement sequence. Notice that both replacement sequences leverage parameterization (for now ignore the '$' that precedes many of the parameter references).

Figure 6.2(b) also shows the assembly format of a DISE codeword. DISE codewords use a reserved opcode (*e.g.*, **res2**). The first operand in a DISE codeword is the tag (*e.g.*, 3), which immediately follows the reserved opcode. The next operands are a sequence of 2-4 register/immediate parameters separated by commas (we discuss how these are encoded later, under the heading "Encoding DISE Codewords"). In Figure 6.2(b), the codeword uses 3 parameters (*e.g.*, **$4**, **8**, **$5**). To reference a parameter, the replacement sequence specifies **T.P1** for the first parameter, **T.P2** for the second parameter, *etc.*. In addition, a '$' in front of the parameter indicates that the parameter refers to a register rather than an immediate, which is not always discernable from its use. It is illegal to use a parameter as both an immediate and a register. In Figure 6.2(b), **$T.P1** becomes **$4** in the transformed code, **T.P2** becomes **8**, and **$T.P3** becomes **$5**.

Parameterized compression. Register/immediate parameters encoded into compressed codewords exploit DISE's parameterized replacement mechanism to allow more sophisticated compression than that supported by dedicated (*i.e.*, dictionary-index only) decompressors. In DISE, a single decompressed code template may yield decompressed sequences with different register names or immediate values when instantiated with different "arguments" from different static locations in the compressed code. In this way, parameterization can be used to make more efficient use of dictionary space.

The use and benefit of parameterized decompression is illustrated in Figure 6.3. Part (a) shows uncompressed static code; the two boxed three-instruction sequences are candidates for compression (the algorithm is presented in Section 6.2.2). Part (b) shows the static code and the dictionary contents for unparameterized compression. Since the sequences differ slightly, they require separate dictionary entries. With parameterized decompression, part (c), the two sequences can share a single parameterized dictionary entry. The entry uses two parameters (shown in bold): **P1** parameterizes the first instruction's input and output registers (referred to as **$T.P1** since it parameterizes a register) and the second instruction's input register, **P2** parameterizes the first instruction's immediate operand. To recover the original uncompressed sequences, the first codeword uses **$3** and **8** as values for the two parameters, while the second uses **$4** and **-8**, respectively.

In addition to allowing more concise dictionaries, parameterization permits the compression of sequences containing PC-relative branches. Conventional mechanisms are incapable of this because compression itself changes PC offsets. Although two static branches may use the same offset before compression, it is likely this will not be true after compression. General solution of this conflict is NP-complete [85]. In DISE, post-compression PC-relative offset changes are no longer a problem. Multiple static branches that share the same dictionary entry prior to compression can continue to do so afterward. With parameterization, even branches that use different *a priori* offsets can share a dictionary entry. The one restriction to incorporating PC-relative branches into dictionary entries is that their offsets must fit within the width of a single parameter. This restriction guarantees that no iterative rewriting will be needed, because compression can only reduce PC-relative offsets. As we show in Section 6.3, the ability to compress PC-relative

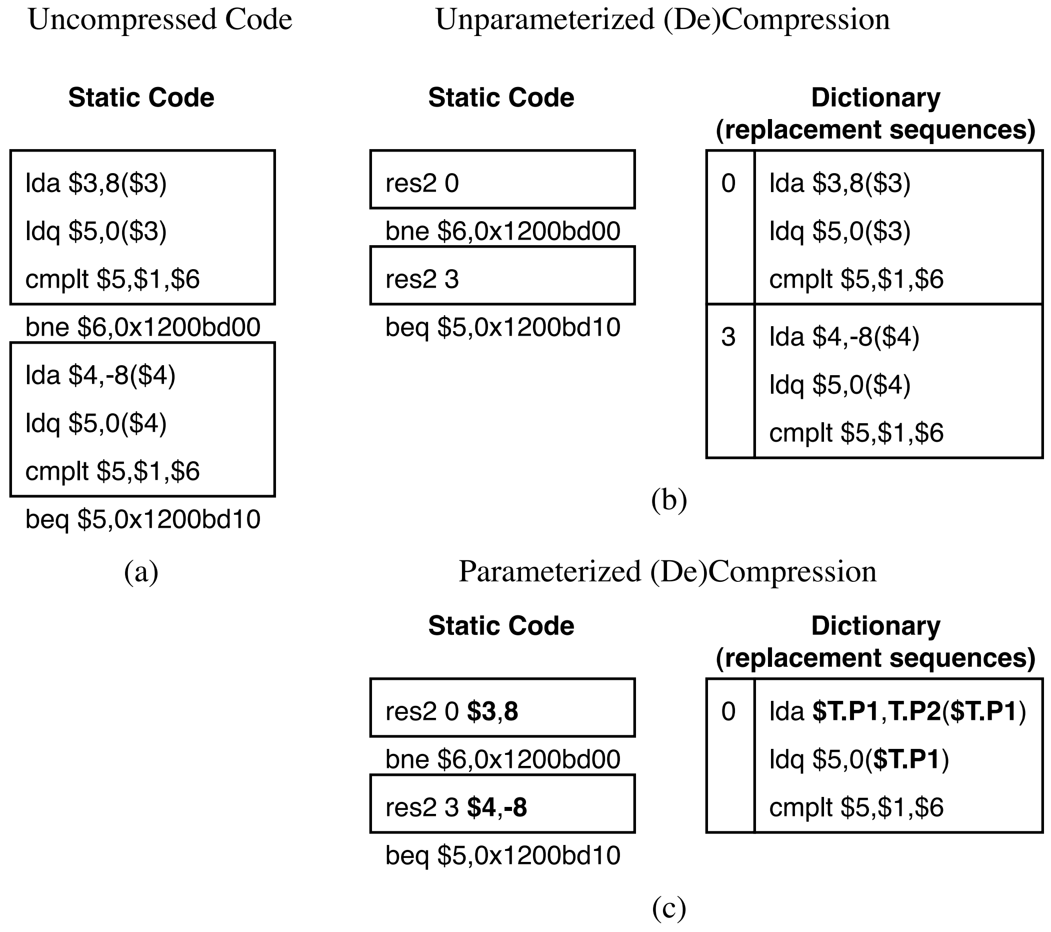
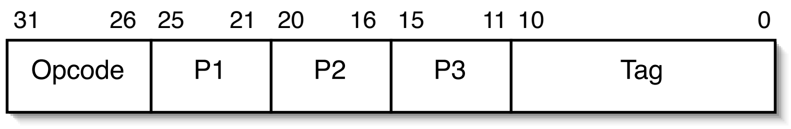


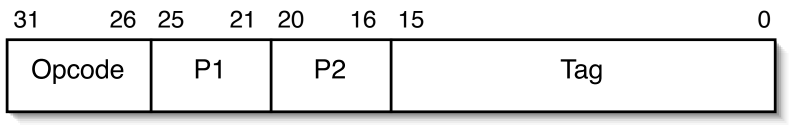
Figure 6.3: Compression examples.

branches gives a significant benefit, because they represent as much as 20% of all instructions.

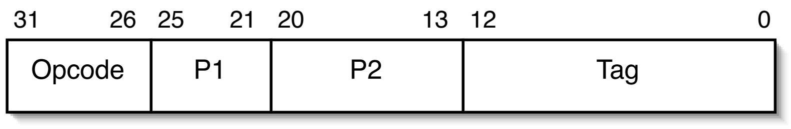
Parameterization is effective because only a few parameters are needed to capture differences between similar sequences. This is due to the local nature of register communication of common programming idioms and the resulting register name repetition. In Figure 6.3, the three-instruction sequence (**lda**, **ldq**, **cmplt**) increments an array pointer, loads the value, and compares it to a second value. The 7 register names used within this sequence represent four distinct values: the array element pointer, the array element value, the compared value and the comparison result. Given four register parameters, we could generalize this sequence completely.



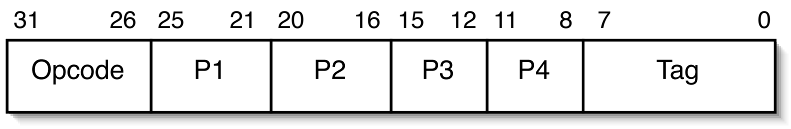
(a)



(b)



(c)



(d)

Figure 6.4: Four DISE codeword encoding formats. The encoding in (a) is our default format.

Encoding DISE codewords. Figure 6.4 shows four DISE codeword encodings. The particular format is determined via a DISE register **$dcfr** (DISE codeword format). It is read and written using the instructions **d_mfdr** and **d_mtdr**. Because an individual register determines the codeword format, two formats cannot be used simultaneously.

The four codewords vary in the size of the tag as well as the number and size of the parameters. Because the tag is used to index a replacement instruction, the size of the tag determines the size of the dictionary (*i.e.*, the size of the replacement sequence space). The implied dictionary sizes of the formats in Figure 6.4 are 2K instructions (a), 65K instructions (b), 8K instructions (c), and 256 instructions (d). Benchmarks with larger footprints may require a format with a larger-sized tag since a larger dictionary naturally improves the compression effectiveness. But at the same time, these formats have

Immediate Interpretation

Code	Interpretation
0	Literal
1	Multiple of 2
2	Multiple of 4
3	Multiple of 8

Table 6.1: Interpretations of 5-bit immediate parameters.

less bits for parameterization, which can hurt compression since similar-but-not-identical sequences may not be able to share the same dictionary entry. We have found that the encoding in Figure 6.4(a) works well in practice (an 11-bit tag with 3 5-bit parameters), and we use it exclusively in our evaluation of DISE-based compression.

A careful reader has noticed that there is a potential problem with the encodings in Figure 6.4. Although the width of the parameter fields is almost always less than or equal to five bits (except for parameter **P2** in Figure 6.4(c), which has eight bits), most immediate fields are wider than five bits. The key here is that in the static uses of a given decompression entry only a few immediates will be used and this small set can be compactly represented in a small number of bits. Within the directives of each replacement instruction we specify how the immediate field, if used, should be interpreted. For example, we could interpret the bits literally, *i.e.*, interpret **00011** as 3. In a 64-bit machine like Alpha, loads and stores often use immediates that are multiples of 8 (quadword size). To capture many of these constants, we could define the interpretation of the immediate to be the 3-bit left-shift of the literal (*i.e.*, multiplied by 8 so that immediates are quad-aligned). The DISE engine performs this logic when instantiating the immediate field in the replacement instruction. Figure 6.1 lists the four immediate interpretations we use in this chapter. Although not shown in Figure 6.1, we also can interpret an immediate as signed or unsigned.

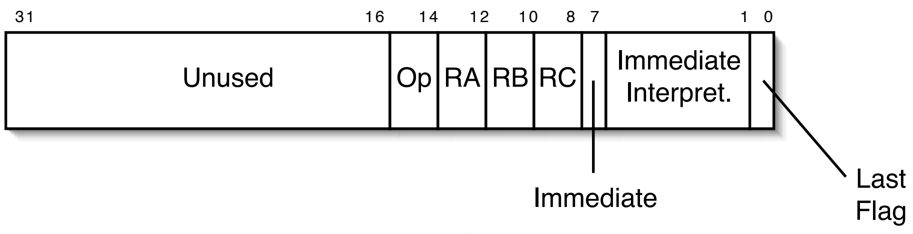
In addition, notice that the fourth encoding (Figure 6.4(d)) has 2 4-bit parameters (along with 2 5-bit parameters). Like a 5-bit parameter, a 4-bit parameter can represent an immediate, although it can capture only half as many immediates. It can also hold a

Directives Encoding

Opcode	RA/RB/RC	Immediate
Literal - 0	Literal - 0	Literal - 0
Parameterized - 1	Parameterized - 1	Parameterized - 1
DISE Opcode - 2	DISE register - 2	

(a)

Directives Layout



(b)

Figure 6.5: Directives: (a) encoding and (b) layout.

register identifier from \$0 to \$15.

Encoding replacement instructions. The encoding of replacement instructions is nearly the same as described in Chapter 3 (page 30). For clarity, however, we omitted a few details in the discussion of directives in Chapter 3 that we discuss here. First, a field in the replacement instruction (at least a register or an immediate field) can be parameterized with the **P1**, **P2**, **P3**, or **P4** fields of the trigger (*i.e.*, codeword). Recall from Chapter 3 that the directive indicates that the field is parameterized and the type of the parameter (*e.g.*, **P1**) is specified in the corresponding field in the template instruction. This field must be able to also encode **P1-P4** in addition to other types of parameters (*e.g.*, **RA**). Because there are less than 16 ways to parameterize any particular replacement instruction field and because each field in the template instruction has more than 4 bits, this is not a problem.

In addition, one other field is encoded within the directives. Figure 6.5 shows the directives encoding and memory layout. In Figure 6.5, unlike in Figure 3.9 from Chapter 3, we dedicate six bits for the interpretation of the immediates, allowing us to encode immediates in the compressed codewords using only five bits.

PT and RT. Up to now, our implementation of decompression has been independent of the implementation of DISE. However, for performance and power reasons, it is important to consider the DISE implementation when using DISE for decompression. First, consider performance. As long as the cost of transformation is small, then the overhead of DISE decompression is negligible (in fact, we show in Section 6.3 that DISE decompression can actually improve performance). Recall, to reduce the overhead of transformation (*i.e.*, macro-expansion), DISE caches frequently-used patterns in a pattern table (PT) and replacement sequences in a replacement table (RT). In the absence of PT and RT misses, the overhead of macro-expansion is small; one additional pipeline stage, which adds a cycle on every mispredicted branch (a rare event). This overhead increases dramatically if there are a significant number of PT and RT misses. Servicing each PT/RT miss can take as much as 30 cycles (more if the specification is not in the data cache). If DISE decompression results in a significant number of PT or RT misses, then performance will suffer.

Fortunately, decompression has little impact on PT misses since it requires only a single pattern. But it can impact RT misses since it generally uses many replacement instructions. Depending on the exact implementation, the RT can hold 512 to 2K replacement instructions. For benchmarks with smaller footprints, which use smaller dictionaries, the RT size is adequate since the entire dictionary fits into the RT. For larger benchmarks, the dictionary will not fit into the RT and the result is additional RT misses, which could significantly degrade performance. If performance is a concern, we can limit the dictionary size to the size of the RT, avoiding costly RT misses at the expense of some loss in compression. In Section 6.3, we evaluate this tradeoff.

If power is a concern, we also need to consider the implementation of DISE. In this case, we may want to modify the implementation slightly to help reduce power. In particular, reducing the sizes of PT and RT entries results in a significant reduction in power consumption. In Figure 6.5, 16 bits are unused in the directives, allowing us to use a smaller RT entry (the replacement instructions in memory and in the executable are 8 bytes for quad alignment). In addition, if the replacement sequence space is larger than RT size, then we also have to tag the RT with the upper bits of the replacement sequence identifier. For a 64K instruction replacement sequence space (*i.e.*, 16-bit identifier) and a 2K instruction RT, the RT tag adds 5 bits to each entry. If saving power is critical, then it is probably advantageous to make the replacement sequence space equal to the RT size. This approach limits the size of our dictionary, and hence hurts compression. It also hurts specification portability, since specifications are designed for a particular RT size, which may change in the future (and have already changed in the past). But, as we show in Section 6.3, it can help in reducing power.

6.2.2 Compression Algorithm

Code compression for DISE consists of three steps. First, a *compression profile* is gathered from one or more applications. Next, an iterative algorithm uses the compression profile to build a *decompression dictionary* (*i.e.*, the set of replacement sequences). Finally, the static executable is compressed using the dictionary (in reverse) to replace compressible sequences (*i.e.*, those that match dictionary entries) with appropriate DISE codewords. We elaborate on each step, below.

Gathering a compression profile. A *compression profile* is a set of weighted instruction sequences extracted from one or more applications. The weight of each sequence represents its static or dynamic frequency. If customized per-program dictionaries are supported, the compression profile for a given program is mined from its own text. If the dictionary is fixed (*i.e.*, a single dictionary is used for multiple programs), a profile that represents multiple applications may be more useful.

A compression profile may contain a redundant and exhaustive representation of instruction subsequences in a program. For instance, the sequence $<$1,2,3,4$>$ may be represented by up to six sequences in the profile: $<$1,2$>$, $<$2,3$>$, $<$3,4$>$, $<$1,2,3$>$, $<$2,3,4$>$, and $<$1,2,3,4$>$. This exhaustive representation is not required, but it gives the dictionary construction algorithm (below) maximum flexibility, improving resultant dictionary quality. We limit the maximum length of these subsequences to some small k (the minimum length of a useful sequence is two instructions), and we do not allow the sequences to span basic blocks. The latter constraint is shared by all existing post-fetch decompression mechanisms and is necessary for correctness because DISE does not permit control to be transfered to the middle of a replacement sequence. Both constraints limit the size of compression profiles and instruction sequence lengths, which are naturally not very long (see Section 6.3).

A weight is associated with each instruction sequence in a profile in order to estimate the potential benefit of compressing it. We compute the benefit of sequence p via the formula: benefit(p) = weight(p) $\times$ (length(p) $-$ 1). The latter factor represents the number of instructions eliminated if an instance of p is compressed to a single codeword. Weight may be based on a static measure (*i.e.*, the number of times the sequence appears in the static executable(s)), a dynamic measure (*i.e.*, the number of times the sequence appears in some dynamic trace or traces), or some combination of the two, allowing the algorithm to target compression for static code size, reduced fetch consumption—a feature that can be used to reduce instruction cache energy consumption (see Section 6.3.6)—or both. For best results, the weights in a profile should match the overlap relationships among the instruction sequences. In particular, the weight of a sequence should never exceed the weight associated with one of its proper subsequences, since the appearance of the subsequence must be at least as frequent as the appearance of the supersequence.

Building the dictionary. A compression/decompression dictionary is built from the instruction sequences in a compression profile using the iterative procedure outlined in Figure 6.6. At each iterative step, the instruction sequence with the greatest estimated compression benefit (minus its cost in terms of space consumed in the dictionary) is identified and added to the dictionary. In environments where the dictionary is fixed and need not be encoded into the application binary, we set the cost of all sequences to zero. In this case, it may be useful to cap the size of the dictionary to prevent it from growing too large. Otherwise, the iterative process continues until no instruction sequences have a benefit that exceeds their cost.

When a sequence is added to the dictionary, corrections must be made to the benefits of all remaining sequences that fully or partially overlap it to account for the fact that these sequences may no longer be compressed. Since DISE only expands the fetch stream and does not re-expand the expanded stream, a sequence that contains a decompression codeword cannot itself be compressed. We recompute the benefit of each sequence (using

```
Initialize dictionary $D$
$P \leftarrow$ GenerateCompressionProfile($\{$programs$\}$)
while $\exists p \in P$ s.t. benefit($p$) $>$ cost($p$)
    select $p \in P$ with largest benefit($p$)$-$cost($p$)
    $P \leftarrow P - \{p\}$
    UpdateDictionary($D$, $p$)    { unify $p$ with existing
                                    entries of $D$ if possible }
    foreach $q \in P$
        benefit($q$) $\leftarrow$ RecalculateBenefit($D$, $q$)
return $D$
```

Figure 6.6: Dictionary construction algorithm.

RecalculateBenefit()) given the sequences that are currently in the dictionary and information encoded in the profile.

Benefit correction monotonically reduces the benefit of a sequence, and may drive it to zero. For example, from our group of six sequences, if sequence <1,2,3> is selected first, the benefit of the sequence <1,2,3,4> goes to zero. Once <1,2,3> is compressed, no sequence <1,2,3,4> will remain. If <1,2,3,4> is selected first, the benefit of sequence <1,2,3> will be reduced, but perhaps not to zero. Once <1,2,3,4> is compressed, instances of <1,2,3> may still be found in other contexts.

Parameterized compression. The dictionary building algorithm is easily extended to support parameterized compression. At each step, before adding the selected sequence to the end of the dictionary, we attempt to *unify* it via parameterization with an existing entry. Two sequences may be unified if they differ by at most p distinct register specifiers or immediate values, where p is the maximum number of parameter values that can be accommodated within a given instruction (a 32-bit instruction can realistically accommodate 3). For instance, assuming p is 1 (our implementation actually supports 3), the sequence <**addq r2,r2,8; ldq r3,0(r2)**> can be unified with the existing sequence <**addq r4,r4,8; ldq r3,0(r4)**> by the decompression entry <**addq P1,P1,8; ldq r3,0(P1)**>. The sequence <**addq r2,r2,16; ldq r3,0(r2)**> cannot be unified with the existing sequence using only a single parameter. We do not attempt opcode parameterization. If unification is possible, the sequence is effectively added to the dictionary for free, *i.e.*, without occupying any additional dictionary space. If unification with multiple entries is possible—a rare occurrence since it implies that two nearly identical entries were not already unified with each other—the one that necessitates the fewest number of parameters is chosen.

In environments where the dictionary size is capped (*i.e.*, to reduce RT misses), parameterization allows us to continue to add sequences to the dictionary so long as they can be unified with existing entries. In other words, the algorithm adds sequences whose cost exceeds their benefit if they may be unified with existing dictionary entries (*i.e.*, they have effectively no cost).

Custom, fixed, and hybrid dictionaries. When the compression profile used by this algorithm is derived from a single program, a dictionary will be generated that is customized to its characteristics, resulting in very effective compression for that program. Unfortunately, the dictionary itself must be encoded in the program binary, for it should only be used to decompress that particular program. If a compression profile is derived from a large collection of programs, a dictionary of more general utility is produced, and it may be used to compress a greater variety of programs. Although the effectiveness of this (fixed) dictionary is likely to be inferior to that of a custom dictionary, the dictionary itself need not be encoded in the program binary, because the system (*e.g.*, OS or hardware vendor) can provide this dictionary to be shared by all programs. It may also be valuable to build a hybrid dictionary that includes fixed and customized components, only the latter of which needs to be embedded in the program binary. This hybrid dictionary offers the promise of achieving the best of both customized (good compression of the program itself) and fixed (little overhead) dictionaries. We evaluate all three approaches in Section 6.3.4.

Rather than compute hybrid custom/fixed dictionaries directly, we combine a custom and fixed dictionary after they have been produced by the algorithm in Figure 6.6. Assuming a fixed-size dictionary, we allocate a portion of this structure to contain fixed entries (*i.e.*, entries derived from profiling a large class of applications) and the remainder is devoted to custom entries (*i.e.*, entries derived from the particular application being compressed). The former will be shared by all compressed programs, so it need not be encoded in the binary. The latter is unique to each compressed program, so it must be encoded in the program binary. The fixed and custom portions are computed as described above, except that dictionary entries appearing in the custom portion that are *subsumed* by entries appearing in the fixed portion are removed. One entry subsumes another when both are identical except for fields in the subsuming sequence that are parameterized where the subsumed sequence was literal. A (perhaps) superior approach would be to detect and eliminate subsumed entries in the algorithm in Figure 6.6, but we have found the naïvesolution to be adequate.

Compressing the program. Given a decompression dictionary—a set of decompression specifications and their replacement sequence identifiers (*i.e.*, tags)—compressing a program is straightforward. The executable is statically analyzed and instruction sequences that match dictionary entries are replaced by the corresponding DISE codewords. The search-and-replace procedure is performed in dictionary order. In other words, for each dictionary entry, we scan the entire binary, and compress all instances of that entry before attempting to compress instances of the next entry. This compression order matches the order implicitly assumed by our dictionary selection algorithm. When compression is finished, branch and jump targets—including those in jump tables and PC-relative offsets in codewords—are recomputed.

Complexity. Dictionary construction dominates the computational complexity of compression. Because sequences are limited to a maximum constant length (k, above), there are $O(n)$ instruction sequences in the compression profile associated with a program containing (before compression) n instructions. Dictionary construction is quadratic in the number of sequences in the compression profile, so it is quadratic in the size of the uncompressed program. No effort has been applied to optimizing the complexity or performance of the compression algorithm. Nevertheless, for most of our benchmarks compression takes less than 30 seconds on 2 GHz Pentium 4.

6.3 Evaluation of DISE Decompression

DISE is an effective mechanism for implementing dynamic decompression in both general purpose and embedded processors. We demonstrate this using custom tools that implement DISE-based compression. Our primary metric is *compression ratio*, the ratio of compressed to uncompressed program sizes. Section 6.3.2 shows the effectiveness of DISE-based compression versus a dedicated-hardware approach. Section 6.3.3 explores the sensitivity of compression to factors such as dictionary size and number of available parameters. Section 6.3.4 assesses both program-specific and fixed-dictionary compression as well as a hybrid of the two. Sections 6.3.5 and 6.3.6 use cycle-level simulation to evaluate the performance and energy implications of executing compressed code.

The experimental data presented in this section serves three purposes. First, it demonstrates that DISE-based decompression is effective and evaluates the impact of DISE-specific features (*e.g.*, the impact of parameters on compression ratio and the impact on performance of demand loading the decompression dictionary into the RT). Second, it compares DISE-based decompression with a dedicated-hardware approach. Finally, some of this data (*e.g.*, impact of dictionary size, impact on energy, *etc.*) is, in fact, DISE-neutral, so our results are equally relevant to dedicated-hardware implementations.

6.3.1 Methodology

Simulator. Our general methodology is described in Section 4.2 of Chapter 4. We simulate using SimpleScalar Alpha [16], modeling the machine from Table 4.1 on page 54. We also model an embedded machine. Table 6.2 shows the characteristics of this machine.

Our DISE default configuration uses a 2-stage decoder-based implementation of DISE with a 32-entry pattern table (although decompression requires only 1 PT entry) and a 2K-entry, 2-way set associative replacement table (see Chapter 4). Each PT entry occupies 8 bytes while each RT entry occupies 6 bytes so the total sizes of the two structures are 512 bytes and 12KB, respectively. The sizes of both structures are chosen to be as large as possible without impacting the clock frequency. In some architectures (besides Alpha), a 12KB RT may be too large to access in 1-cycle. For this reason, we also show results for a 3KB (512-entry) and a .75KB (128-entry) RT.

For the general purpose configuration, we assume a 2-stage decoder, so DISE expansion introduces no overhead. For the embedded configuration, we assume an *a priori* 1-stage decoder. The DISE interface and its cost do not impact the use of DISE for code decompression, so they are not explicitly modeled. We model the DISE miss handler by flushing the pipeline and stalling for 30 cycles.

The simulator models power consumption using the Wattch framework [13], a widely-used research tool for architecture power analysis, and CACTI-3 [92], a cache area, access and cycle time and power consumption estimation tool. Our power estimates are

Machine	MIPS-like, in-order
ISA	Alpha
Processor width	2
Pipeline stages	5
Reorder buffer size	128
Reservation stations	80
Instruction cache	8KB, 2-way set associative, 1-cycle access time
Instruction TLB	none
Data cache	16KB, 2-way set associative, 1-cycle access time
Data TLB	none
Unified L2 cache	none
Main memory	infinite, 50 cycle access time
Memory bus	32 bytes wide, 1/4 processor frequency
Branch predictor	Hybrid bimodal/gshare, 1K entry
Branch target buffer	128 entry

Table 6.2: Default embedded machine characteristics.

for 0.13μm technology. The structures were configured carefully to minimize power consumption and roughly mirror the per-structure power distributions of actual processors. For a given logical configuration, CACTI-3 employs both squarification and horizontal/vertical sub-banking to minimize some combination of delay, power consumption and area. We configure both the instruction cache and RT as two-way interleaved, single-ported (read/write) structures that are accessed at most once per cycle.

Benchmarks. We perform our experiments on the SPEC2000 integer and MediaBench [58] benchmarks. The SPEC benchmarks run on the general purpose processor configuration, while the MediaBench codes run on the embedded configuration. All programs are compiled for the Alpha EV6 architecture with the native Digital Unix C compiler with optimization flags *-O4 -fast*. When execution times are reported for SPEC,

SPEC

benchmark	code size (insn)	IPC	I$ misses
bzip2	36,013	2.46	∼0%
crafty	82,863	2.08	.51%
eon	150,998	2.13	.64%
gap	172,581	1.70	.58%
gcc	364,429	1.53	1.25%
gzip	38,871	2.15	∼0%
mcf	32,018	0.45	.01%
parser	57,617	1.51	∼0%
perlbmk	173,135	1.55	1.1%
twolf	88,324	1.60	.11%
vortex	162,613	2.30	.75%
vpr	70,735	1.24	∼0%

(a)

MediaBench

benchmark	code size (insn)	IPC	I$ misses
adpcm.caudio	26,189	0.98	∼0%
epic.epic	45,211	1.33	∼0%
epic.unepic	34,190	1.15	.02%
g721.dec	23,875	1.19	∼0%
ghostscr.gs	329,371	0.99	1.47%
gsm.toast	38,810	1.60	.14%
jpeg.cjpeg	48,733	1.02	.03%
jpeg.djpeg	52,484	1.06	.09%
mesa.mip	189,309	1.23	∼0%
mpeg2.dec	59,416	1.16	.03%
mpeg2.enc	47,087	1.37	∼0%
pegwit.enc	43,961	0.94	.01%

(b)

Table 6.3: Benchmark summary for (a) SPEC Int 2000 and (b) MediaBench.

they come from complete runs sampled at 10% (100M instructions per sample) using the train input. MediaBench results are for complete runs using the inputs provided [58]; no sampling is used. Table 6.3 lists some characteristics of the benchmarks that are useful in interpreting the results below. Note the instruction cache misses are normalized to the total number of executed instructions.

Dictionaries. Compression profiles are constructed by static binary analysis (except in Section 6.3.6). The compression tool generates a set of decompression specifications (the dictionary) via the algorithm presented in Section 6.2. Our default compression parameters are a maximum dictionary entry length of 8 instructions and no more than 3 register/immediate parameters per entry. Except for the experiments in Section 6.3.4, a custom dictionary is used for each benchmark. Except for the experiment in Section 6.3.6, each dictionary is constructed using a compression profile where weights encode static instruction sequence frequency.

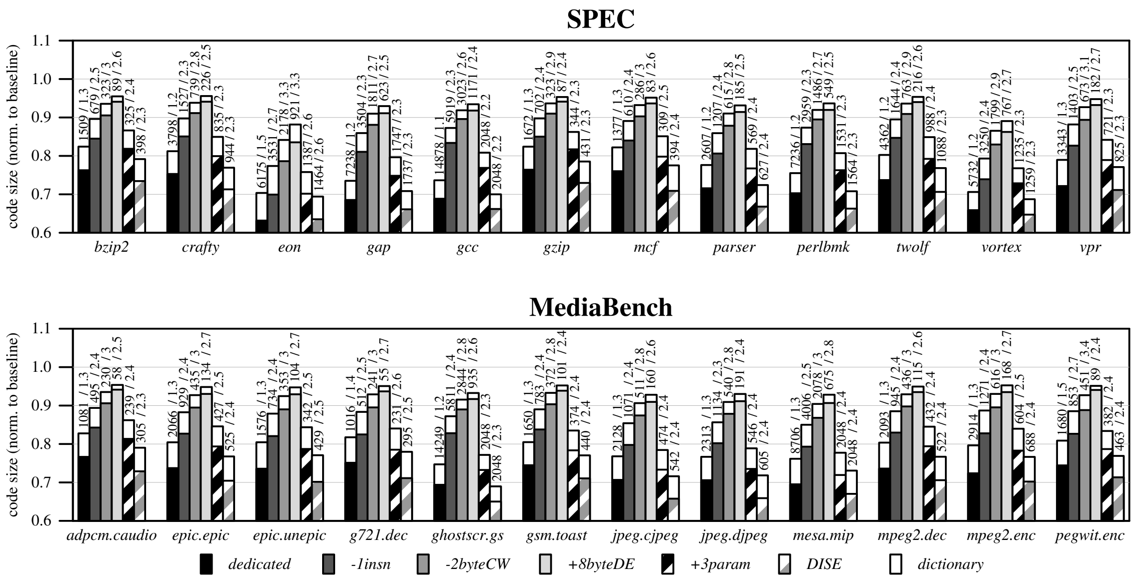


Figure 6.7: Dedicated and DISE-based feature impact on compression.

6.3.2 Compression Effectiveness

We begin with a comparison of the compression efficacy of DISE to that of a previously proposed system that exploits dedicated decompression-specific hardware [59]. The dedicated approach does not support parameterized replacement. As a result, it cannot compress PC-relative branches or share dictionary entries in certain situations, but it does have smaller dictionary entries (no directives) and smaller codewords (no parameters), and so it can profitably compress single instruction sequences.

We separate the impact of these differences in Figure 6.7. Bars represent static compression ratio broken down into two components. The first (bottom, shaded portion of each stack) is the (normalized) compressed size of the original program text. The second (top white portion) is the size of the dictionary as a fraction of original program text size. The combination of these two bars represents the total amount of data required to represent a compressed program. The two numbers written on top of each bar are the total number of dictionary entries, and the average number of instructions per entry, respectively. Each bar gives the compression of a decompressor with a slightly different feature set.

Dedicated decompression features. The first bar (*dedicated*) corresponds to a dedicated hardware decompressor, complete with 2-byte codewords and single-instruction compression [59]. The compression ratios achieved—about 70–75% of original text size, dictionary not included (note the scale of the graph)—are comparable to those previously published [59]. In the next two bars, we progressively eliminate the dedicated decompressor's two advantages: single-instruction compression (*-1insn*) and the use of 2-byte codewords (*-2byteCW*). Eliminating these features reduces compression effectiveness to approximately 85%.

DISE decompression features. With dedicated-decompression-specific features removed, the next three bars add DISE-specific features. The use of parameterized replacement requires four additional bytes per dictionary entry to hold the instantiation directives (*+8byteDE*). Without parameterization, larger dictionary entries require more static instances to be considered profitable. As a result, fewer of them are selected and compression ratios degrade to 90% and above. Shown in the fifth bar, parameterization (*+3param*, we allow three parameters per dictionary entry) more than compensates for the increased cost of each dictionary entry by allowing sequences with small differences to share entries; it improves compression ratios dramatically (back down to 75–80%). The final bar (*DISE*)—corresponding to the full-featured DISE implementation—adds the compression of PC-relative branches. The high static frequency of PC-relative branches enables compression ratios of 65%, appreciably better than those achieved with the dedicated hardware scheme.

The numbers on top of the bars—number of dictionary entries and average number of instructions per entry—point to interesting differences in dictionary-space usage between the dedicated and DISE schemes. While the two schemes use roughly the same amount of total dictionary storage (see the portion of each bar), recall that DISE requires twice the storage per instruction, meaning the DISE dictionaries contain roughly half the number of instructions as the dedicated ones. Beyond that, dedicated dictionaries typically consist of a large number of small entries, including many single-instruction entries. DISE dictionaries typically consist of a smaller number of longer entries. The difference is due to the absence of single-instruction compression—which means that the average compression sequence length must be at least two—and the use of 4-byte codewords which require longer compressed sequences to be profitable. Parameterized replacement does not increase the average entry size, it just makes more entries profitable since they can be shared among more static locations.

Notice, the total number of dictionary entries for the DISE schemes cannot exceed 2K, since parameterized DISE codewords contain only 11 bits for the tag (*i.e.*, replacement sequence).

6.3.3 Sensitivity Analysis

The results of the previous section demonstrate that unconstrained decompression is effective. Below, we investigate the impact of dictionary entry size (in terms of instructions), total dictionary size, and the number of register/immediate parameters per dictionary entry.

Dictionary entry size. Post-fetch decompression restricts compressed sequences to reside fully within basic blocks. Although basic block size is small in the benchmarks we consider, there may be benefit to restricting dictionary entry size even beyond this natural limit. Small sequences may admit more efficient RT organizations and tagging schemes and can reduce the running time of the compressor itself. Our experiments (not graphed here) show that 4-instruction sequences allow better compression (up to 8%) than 2-instruction sequences, 8-instruction sequences occasionally result in slightly better compression still, and 16-instruction sequences offer virtually no advantage over those. Our algorithm simply never selects long instruction sequences for compression because similar long sequences do not appear frequently in the codes we studied.

Dictionary size. Although DISE virtualization allows the dictionary to be larger than the physical RT, a dictionary whose working set exceeds RT capacity will degrade performance via expensive RT miss handling. To avoid RT misses, it is often useful to limit the size of the dictionary, but this naturally degrades compression effectiveness. Figure 6.8 shows the impact of dictionary size on compression ratio. Note, we define dictionary size as the total number of instructions, *not* the number of entries (*i.e.*, instruction sequences).

Non-trivial compression, reductions of 1–5% in code size are possible with dictionaries as small as 8 total instructions, and 12% reductions are possible with 32-instruction dictionaries (*e.g.*, *vortex*). 512-instruction dictionaries achieve excellent compression,

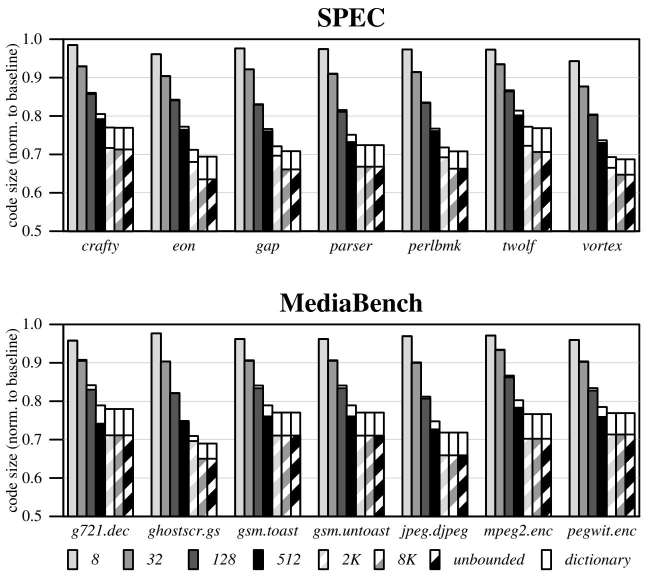


Figure 6.8: Impact of dictionary size on compression.

70–80% of original program code size on all programs. Increasing dictionary size to 2K instructions yields small benefits. Only the larger benchmarks (*i.e., eon, perlbmk,* and *ghostscript*) reap additional benefit from an 8K instruction dictionary. The remaining benchmarks are unable to exploit the additional capacity.

Number of parameters. Parameterized decompression allows for smaller, more effective dictionaries, because similar-but-not-identical sequences can share a single dictionary entry as in Figure 6.3. This is a feature unique to DISE among hardware decompression schemes; Figure 6.9 shows its impact.

Compression ratios improve steadily as the number of parameters is increased from zero to three; the difference between zero and three parameters is about 15% in absolute terms. Compression improves even further if more than three parameters are used, but there is little benefit to allowing more than six parameters (assuming we could encode six parameters in a single codeword). This diminishing return follows directly from our dictionary entry size results. Each instruction contains no more than three registers (or two registers and one immediate). Since most dictionary entries are 2–4 instructions long, they cannot possibly contain more than 12 distinct register names or immediate values. Of course, in practice the number of distinct names is much smaller. Contiguous instructions tend to be data-dependent and these dependences are expressed by shared register names.

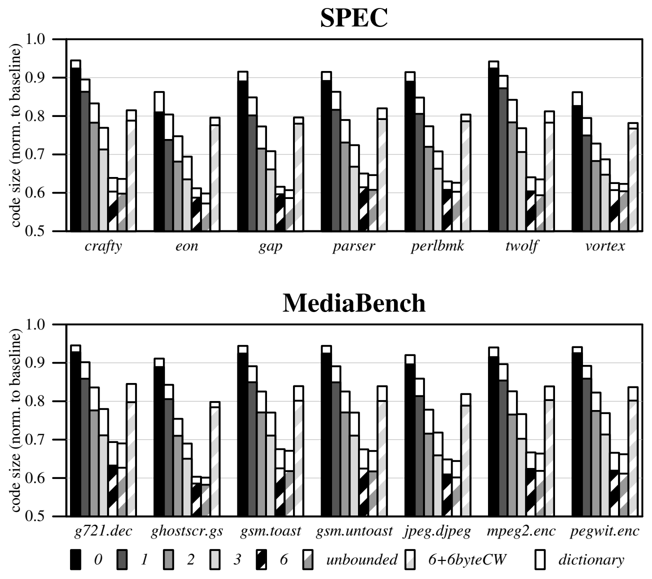


Figure 6.9: Impact of parameters on compression.

Parameterized replacement therefore has the nice property that a few parameters capture a significant portion of the benefit. The final bar (*6+6byteCW*) repeats the 6-parameter experiment, but uses longer—6 rather than 4 byte—codewords to realistically represent the overhead of encoding additional parameters. The use of longer codewords makes the compression of shorter sequences less profitable, completely overwhelming the benefit achieved by the additional three parameters. Three parameters—the maximum number that can fit within a 32-bit codeword and still maintain a reasonably sized replacement sequence identifier—yields the best compression ratios.

6.3.4 Dictionary Programmability

One advantage of DISE-based compression is dictionary programmability, the ability to use a per-application dictionary. Although previous proposals for post-fetch decompression [59] did not explicitly preclude programmability, a programming mechanism was never proposed and the impact of programmability was never evaluated. In DISE, dynamic dictionary manipulation is possible via the controller.

Custom versus fixed dictionaries. We consider the impact of programmability by comparing three compression usage scenarios. In *application* we create a custom dictionary

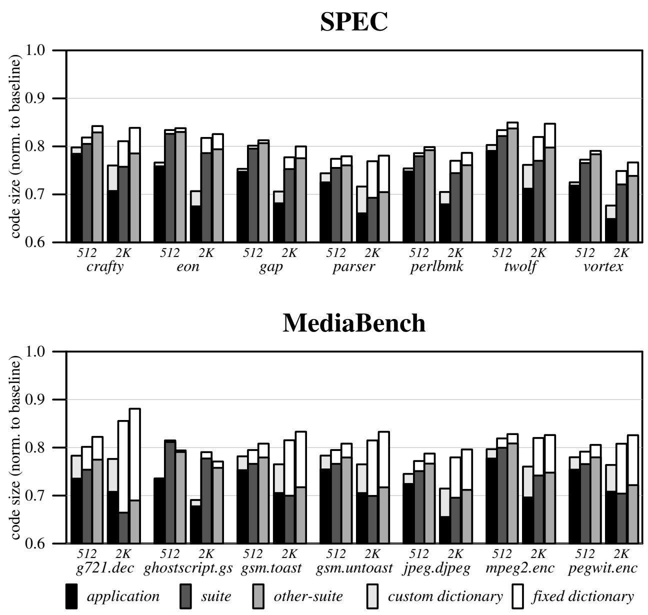


Figure 6.10: Impact of dictionary customization on compression.

for each application and encode it into the executable. All the data presented thus far assumes this scenario. The other two scenarios assume a fixed, system-supplied dictionary, that either resides in kernel memory or is perhaps hardwired into the RT. In these scenarios, the system provides the compression utility. The first of these, *suite*, models a system with a limited but well-understood application domain. Here, we build a dictionary using static profile data collected from the other applications in the benchmark suite. The second (*other-suite*) models a system with little or no *a priori* knowledge of the application domain. Here, dictionaries are built using profile data from programs in the other benchmark suite. One advantage of system-provided (*i.e.*, fixed) dictionaries is that they do not consume space in the compressed application's executable.

Figure 6.10 shows the impact of each usage scenario on compression ratio. We actually show the results of two experiments, limiting dictionary size to 512 and 2K total instructions. Not surprisingly, at small dictionary sizes, an application-specific dictionary (*application*) out-compresses a fixed dictionary (*suite* and *other-suite*), even when considering that dictionary space is part of the compressed executable in this scenario and not the other two scenarios. Being restricted to relatively few compression sequences while limiting the overall cost of the dictionary to the application places a premium on careful selection and gives the *application* scenario an advantage. As dictionary size is increased, however, careful selection of sequences becomes less important while the fact that entries in fixed dictionaries are "free" to the application increases in importance. With a 2K instruction dictionary, "inversions" in which an application-agnostic dictionary outperforms the application-specific one are observed (*e.g.*, *g721*, *gsm*, *pegwit*). Of course, these are achieved using very large fixed dictionaries which would not be used if the application were forced to include the dictionary in its own binary.

The *suite* scenario often out-compresses *other-suite*, implying that there is idiomatic similarity within a particular application domain. For instance, a few of the MediaBench programs have many floating-point operations whose compression idioms will not be generated by the integer SPEC benchmark suite. The one exception to this rule is *ghostscript*, which arguably looks more like an integer program—it's call-intensive in addition to loop-intensive—than an embedded media program.

Hybrid custom/fixed dictionaries. From the data in Figure 6.10, it is apparent that there are unique virtues to both customized (*application*) and fixed (*suite* and *other-suite*) approaches to building and using dictionaries. Customized dictionaries allow for the best compression of the program, but the dictionary itself must be encoded in the program binary, sometimes negating the benefit of customization (*e.g., g721.dec*). Fixed dictionaries have the benefit that they need not be represented in the program binary, but they usually result in poorer compression of the program itself (although this is not always true for reasons described above). A *hybrid* approach for dictionary construction attempts to achieve the best of both worlds.

Figure 6.11 presents the impact of hybridization. We partition both 512 and 2K entry RTs (the RT must house *both* the custom and fixed part of the dictionary) in six incrementally different ways. The percent under each bar indicates the portion of the total dictionary devoted to custom entries. 100% is completely custom and 0% is entirely fixed. The white (top) portion of each bar represents the custom portion of the dictionary that must be encoded in the binary. So that all benchmarks share exactly the same fixed portion, our fixed entries are derived from all of the benchmarks within each benchmark suite (including the benchmark being compressed). As a result, the 0% figures differ slightly from the *suite* bars (which exclude the benchmark being compressed) in Figure 6.10.

Ignoring custom dictionary overhead for the moment (top white portion of each bar),

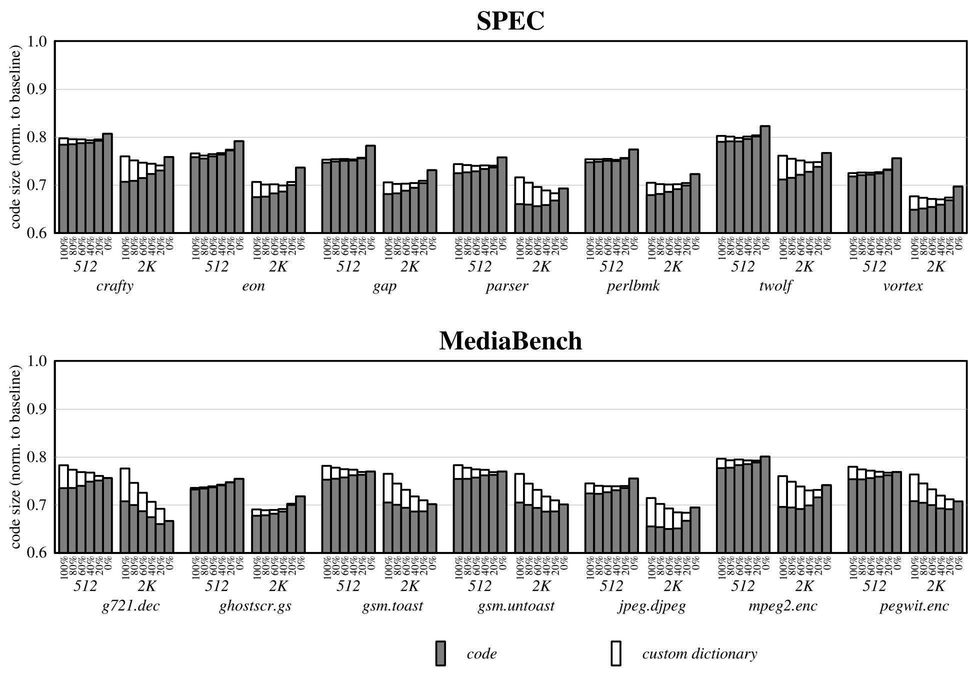


Figure 6.11: Impact of hybrid custom/fixed dictionary on compression.

each set of bars exhibits one of three basic shapes: (1) compression ratio (gray bar only) increases as less of the dictionary is dedicated to custom entries, (2) compression ratio decreases, or (3) compression ratio decreases, then increases. The first case (*e.g.*, *crafty*, 2K-entry RT) is most natural and common. As the total dictionary becomes less customized to the application being compressed, the compression ratio will naturally get worse (*i.e.*, increase). We see this in almost all SPEC benchmarks and most of the MediaBench codes with small (*i.e.*, 512-entry) RTs.

The second case (compression ratio actually improves when more of the dictionary is devoted to fixed entries) is, at first, unintuitive. The origin of this odd occurrence is that an entry is only added to the custom dictionary if the compression benefit (in the program) exceeds the cost of adding the entry to the dictionary. As a result, it is often the case (particularly for large RTs) that the custom portion of the dictionary is not full, and converting custom entries to fixed entries does not in fact reduce the number of custom entries but it does increase the number of fixed entries. The end result is that there are actually more total entries in the dictionary resulting in better compression. This most naturally occurs for small programs and large dictionaries, so we see it for a number of MediaBench codes with 2K RTs.

The third case (compression ratio improves, then degrades, *e.g., g721.dec*/2K) is the natural combination of the first two cases. The ratio improves at first because fixed-dictionary entries are being added without impacting the custom entries, but at some point, the fixed dictionary cuts into valuable custom entries, degrading compression ratios.

Now we consider the overhead of the custom dictionary (top white portion of each bar). Naturally, this decreases as more of the dictionary is devoted to fixed entries. In some cases, the reduced custom dictionary overhead is overshadowed by the degraded compression of the binary (*e.g., ghostscript.gs*, 512-entry RT), but most codes actually benefit from dedicating at least some of the dictionary to fixed entries. *Crafty*/2K is a nice example; it achieves the best total compression when only 20% of the dictionary is customized. Most of the codes exhibit a similar valley around 20% or 40%. Although sometimes the best compression is achieved with a completely fixed dictionary (*e.g., g721.dec*), there is usually a significant jump from the 20% bar to the 0% bar, suggesting that some amount of customization is useful.

6.3.5 Performance Impact

The performance of a system that uses DISE decompression depends on the average access times of two caches: the instruction cache and the RT, which acts as a cache for the dictionary. Since each is accessed in an in-order front-end stage, penalties are taken in series and translate directly into end latency.

The next two sections of the evaluation focus on performance and energy, variations in which are due to trade-offs between the instruction cache and RT.

Instruction cache performance. Figure 6.12 isolates instruction cache performance by simulating an ideal DISE engine, an infinite RT with no penalty per expansion. The figure shows the relative performance of fifteen instruction-cache/dictionary configurations: each of three cache sizes used in conjunction with each of five dictionary sizes—0 (no decompression), 128 entries, 512, 2K, and an unbounded dictionary. We show performance (IPC; higher bars are better) normalized to that of a configuration with a 32KB instruction cache and no decompression. Performance, naturally, decreases with cache size, in some cases significantly (*e.g., crafty* suffers five times the number of misses with a 16KB cache versus a 32KB cache). Of the three components of average access time—hit time, miss

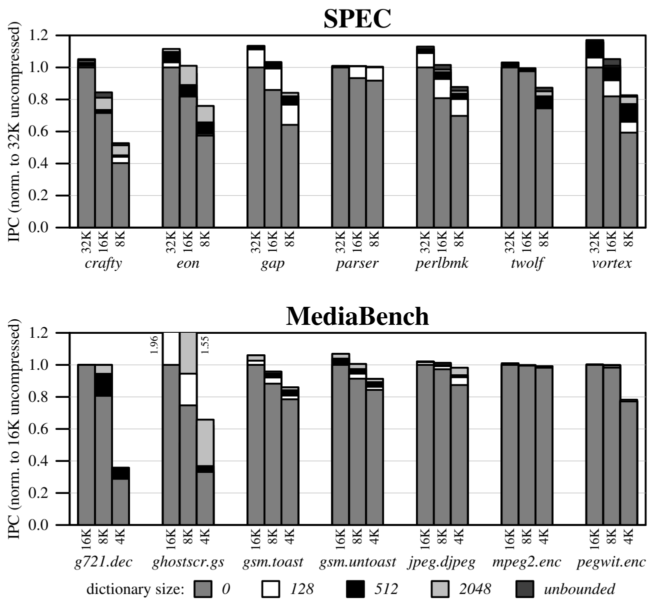


Figure 6.12: Performance impact of instruction cache and dictionary.

rate, and miss penalty—only the miss rate is impacted by DISE; we fix the miss penalty and ignore the possibility that smaller caches could be accessed in fewer pipeline stages.

While larger dictionaries can improve static compression ratios, small ones suffice from a performance standpoint. For many programs, much of the static text compressed by larger dictionaries is not part of the dynamic working set, and its compression does not influence effective cache capacity. About half of the programs (*e.g.*, *gap*, *parser*, and *perlbmk*) benefit little from dictionaries larger than 128 total instructions, and only *crafty* and *vortex* show significant improvement when dictionary size is increased beyond 2K instructions.

Counter-intuitively, compression may hurt cache performance by producing pathological cache conflicts that did not exist in uncompressed (or less aggressively compressed) code. This effect is more likely to occur at small cache sizes. A prime example is *ghostscript*. Although not immediately evident from the figure, on the 8KB and 4KB caches, the 512 instruction dictionary actually underperforms the 128 instruction dictionary. The pathological conflict—actually there are two clustered groups of conflicts each involving 4–5 sets—disappears when the larger, 2K instruction dictionary is used. We have verified that this artifact disappears at higher associativities (*e.g.*, 8-way). The same effect occurs, but to a far lesser degree, in *gap* and *twolf*. The presence of such artifacts argues for the utility of programmable compression.

DISE engine performance. In contrast with the preceding, here we are concerned with all aspects of RT performance. As discussed in Chapter 4, RT hit time is determined by the DISE engine pipeline organization. We add an additional stage to the decoder (which was originally a 1-stage decoder), which results in a one-cycle penalty on every mispredicted branch. Because mispredicted branches are uncommon, the cost is quite small (*e.g.*, 0.5–1%).

The other components of RT access time are miss rate and the cost of servicing a miss. The miss rate was also briefly evaluated in Chapter 4, but here we re-evaluate it in more detail. The RT miss rate is a function of virtual dictionary working set size and the physical RT configuration, primarily the capacity. RT misses are quite expensive. We model the RT miss penalty by flushing the pipeline and stalling for 30 cycles. Figure 6.13 shows the performance (*i.e.*, IPC) of systems with several virtual dictionary sizes (128, 512, 2K instructions) on RTs of several different configurations (128, 512, and 2K instruction specification slots arranged in four instruction blocks, both direct mapped and 2-way set-associative). Performance is normalized to the "large instruction cache" (32K or 16K) DISE-free configuration, while the DISE experiments all use smaller caches. For this reason, slowdowns—normalized performance of less than 1—are sometimes observed, especially for the small physical RT configurations. Since the RT miss penalty is fixed, performance differences are a strict function of the RT miss rate.

As the figure shows, a large virtual dictionary on a small physical RT produces an abundance of expensive RT misses which cause frequent execution serializations. A 2K-instruction dictionary executing on a 128 entry RT can degrade performance by a factor of 5 to 10 (*e.g.*, *vortex*). Although RT virtualization guarantees correct execution, to preserve performance, dictionaries should not exceed the physical size of the RT. The instruction conflict pathology described in the previous section is again evident in *twolf*. On a 2K-instruction RT, the 512-instruction dictionary outperforms the 2K-instruction dictionary, even though neither generates RT misses.

The MediaBench programs typically require smaller dictionaries and are more loop oriented than their SPEC counterparts. 2K-instruction dictionaries are rare even when no limit is placed on dictionary size, and dictionaries tend to exhibit better RT locality. As a

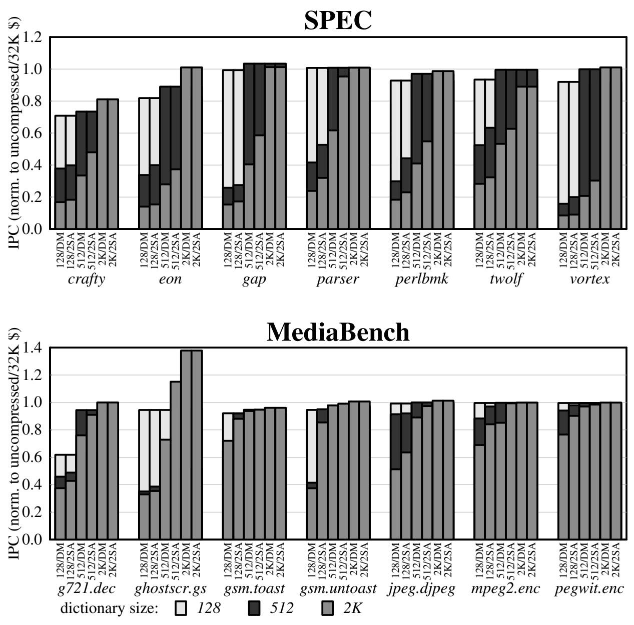


Figure 6.13: Performance impact of RT misses.

result, larger dictionaries perform relatively better on small RTs than they do in SPEC.

6.3.6 Energy Implications

In a typical general purpose processor, instruction cache access accounts for as much as 20% of total processor energy consumption. Other structures, like the data cache and L2 cache, may be as large or larger than the instruction cache, but are accessed less frequently (the instruction cache is accessed nearly every cycle) and typically one bank at a time (all instruction-cache banks are accessed on each cache access cycle). In an embedded processor, which may contain neither an L2 nor a complex execution engine, this ratio may be even higher.

Post-fetch decompression can be used to reduce energy consumption, both in the instruction cache and in total. Energy reduction can come from two sources: (i) reduced execution times due to compressed instruction footprints and fewer instruction cache misses, and/or (ii) the use of smaller, lower-power caches. However, there are two complementary sources of energy consumption increase. First, the DISE structures themselves consume energy. Second, the use of a smaller instruction cache may decrease effective instruction capacity beyond compression's ability to compensate for it, increasing instruction cache

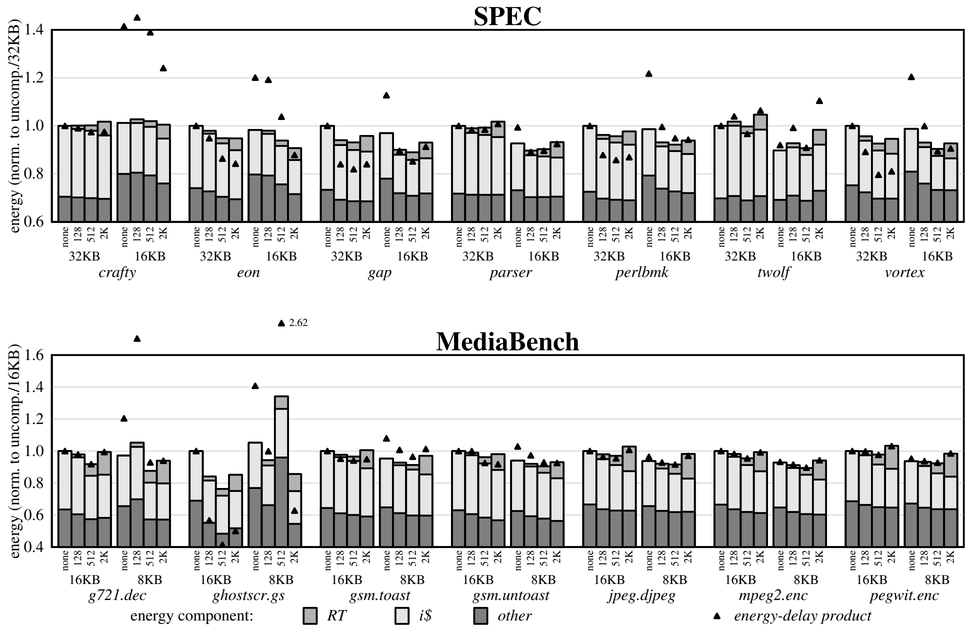


Figure 6.14: Impact of compression on energy.

misses and execution time. These effects must be balanced against one another. The potential exists for doing so on a per-application basis by selectively powering down cache ways [3] or sets [96]. A similar strategy can be used for the RT. Below, we assume that the replacement sequence space is the same size at the RT. No RT tag is necessary, reducing the size of each RT entry, and thus, the RT's total power consumption.

Energy and EDP. Figure 6.14 shows the relative energy consumptions and energy-delay products (EDP, a metric that considers both energy and execution performance) of several DISE-free and DISE decompression configurations. Energy bars are normalized to total energy consumption of the DISE-free system with the larger (32KB or 16KB) instruction cache, respectively. Each bar shows three energy components: instruction cache (light), DISE structures (medium), and all other resources (dark). Notice, instruction cache energy is about 15-25% of total energy in a general purpose processor and 35-45% in an embedded processor. The EDP for each configuration is shown as a triangle. There are eight total configurations, uncompressed and compressed with three RT sizes for each of two instruction cache sizes. Since RT misses have a high performance penalty and thus energy cost, we use virtual dictionaries that are of the same size as the physical RTs.

DISE-based compression can reduce total energy and EDP even though the trade-off between cache and RT instruction capacity highly favors the cache. In the first place, accessing two 16KB structures consumes more energy than accessing a single 32KB structure. Although wordline and bitline power grows roughly linearly with the number of RAM cells in an array, the power consumed by supporting structures—wordline decoders, sense-amplifiers and output drivers—is largely independent of array size. Multiple structures also consume more tag power. Our simulations show that a 32KB single-ported cache consumes only slightly over 40% more energy per access than a single-ported 16KB cache, not 100% more. Beyond that, however, an RT is less space efficient than an instruction cache because it must store per-instruction instantiation directives as well. When we combine these factors, we see that in order to save energy over a 32KB configuration, we must replace 16KB of cache (storage for 4K instructions) with a 3KB RT (storage for 512 replacement instruction specifications). Fortunately, the use of parameterized replacement enables even small dictionaries to cover large static instruction spaces, making this organization profitable.

For most benchmarks, the lowest energy (or EDP) configuration combines an instruction cache with an appropriately sized dictionary and RT. Note, the lowest energy and the lowest EDP are often achieved using different configurations. In general, DISE is more effective at reducing EDP than energy, as it trades instruction cache energy for RT energy. Typical energy reductions are 2-5%, although reductions of 18% are sometimes observed (*e.g.*, *ghostscript* with 16KB instruction cache and 512-instruction dictionary). Without RT misses (recall virtual dictionaries are sized to eliminate misses), performance improvements due to instruction cache miss reductions account for EDP reductions which often exceed 10% (*e.g.*, *eon*, *gap*, *perlbmk*, *vortex*) and sometimes reach 60% (*e.g.*, *ghostscript*).

Targeting compression to reduce cache accesses. A third way to reduce instruction cache energy—and thus total energy and EDP—is to reduce the number of instruction cache *accesses*. To this point, our compression profiles have been based on static instruction sequence frequency. As a result the statically most frequently occurring sequences are compressed. Alternatively, compression profiles may encode dynamic sequence frequency, allowing us to compress sequences that appear frequently in the dynamic execution stream. Our compression algorithm easily builds compression dictionaries for this

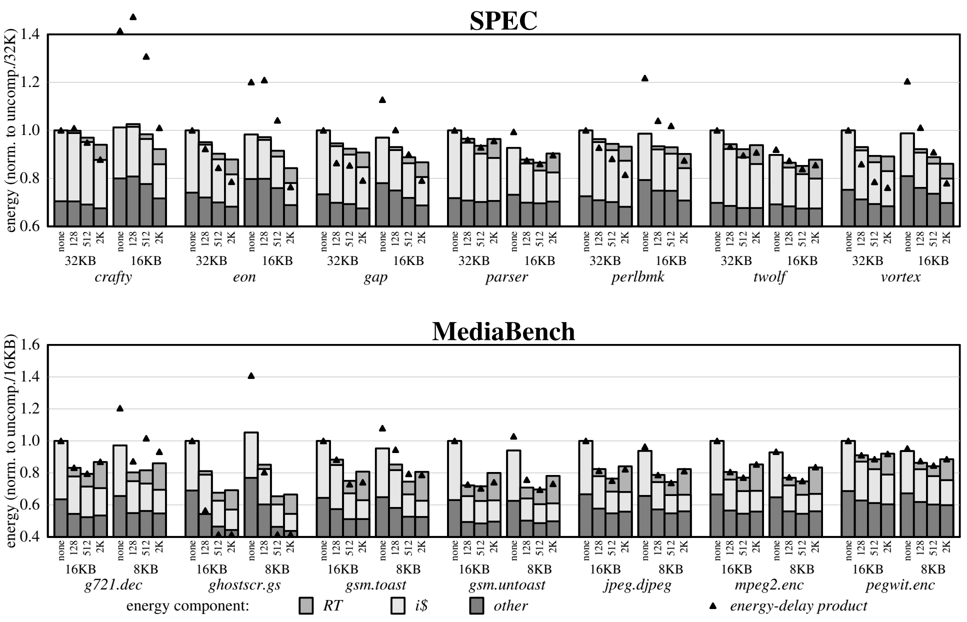


Figure 6.15: Impact of profile-based code compression on energy.

scenario. It simply weighs each instruction sequence in a compression profile by an incidence frequency taken from some dynamic execution profile. Although this will likely not reduce code size by as much as the static alternative, compression in this manner will further reduce cache energy.

In Figure 6.15, we repeat our experiment using dynamic-profile-based dictionaries (note that we have chosen the inputs to these benchmarks to be different from those used in the dynamic profiling step). By greatly reducing instruction cache power, especially for larger dictionaries, dynamic decompression provides more significant reductions in both energy and EDP. 10% energy reductions are common (*e.g.*, *eon*, *gap*, *vortex*, *ghostscript*, *gsm*) as are 20% EDP reductions. Note, the additional EDP reduction comes from the corresponding energy reduction, not from a further reduction in execution time.

6.4 Related Work in Code Compression

The large body of work on code compression speaks to the importance of the technique. In summary, the principal contribution of this chapter the demonstration that a general purpose-dynamic code transformation mechanism (*i.e.*, DISE) can be used to effectively and efficiently implement dynamic code decompression.

Software-based approaches. Traditional static optimizations intended to accelerate execution (*e.g.,* dead-code elimination, common sub-expression elimination, register allocation, *etc.*) often have the side effect of reducing code size [32]. We use an already optimized uncompressed baseline in our experiments. *Code factoring* replaces common instruction sequences with calls to procedures containing these sequences. Factoring reduces code size at the expense of increased execution time due to function call overhead [20, 32]. ISA extensions have been proposed to reduce this overhead [63].

Non-executable compressed formats permit more aggressive compression but require an explicit and expensive decompression step before execution. Systems have been proposed for decompressing code at the procedure [54] and cache-line granularities [60]. Although effective in reducing code size, the performance of these systems degrades significantly. An interesting extension to these works builds on the observation that decompressing frequently executed code slows execution; therefore by compressing only infrequently executed code, code size is reduced yet execution time is minimally degraded (say, 4% [31]). In contrast, hardware post-fetch decompression implementations like DISE, can actually reduce execution time and energy by specifically concentrating on compressing frequently executed code.

Ernst *et al.* describe a compressed code format (byte-code RISC or BRISC) that may be directly interpreted (*i.e.*, it does not require a separate decompression step) [36]. The representation includes a form of operand parameterization. Although highly effective in an interpreted environment, the approach is too expensive for hardware implementation.

ISA extensions. Certain ISAs (*e.g.*, ARM's Thumb [1] and MIPS16 [56]) support compact code via short-form versions of commonly used instructions. Although there is no significant overhead in decompression itself, performance suffers because the short formats provide a limited register and opcode menu, increasing the number of instructions in short format regions (mode switches are required between short and 32-bit code regions). A recent ISA extension, Thumb-2, better balances the compression/performance trade-off, approaching the compression levels of Thumb without as significant a performance degradation [70]. Nevertheless, dense instruction encodings do not exploit repetition of code sequences like coarse-grained (*i.e.*, multiple instruction) compression schemes. Dense encodings and coarse-grained compression mechanisms are orthogonal and can be used in conjunction.

Hardware-based approaches. Fill-path decompression is a hardware technique in which compressed code in memory is decompressed by the instruction cache fill unit on a miss. Examples of fill-path decompression include the Compressed Code RISC Processor (CCRP) [94] and IBM's CodePack [50]. Fill-path decompression schemes necessitate no processor core modifications and incur decompression cost only on instruction cache misses. Although in rare cases they may improve performance (*e.g.*, CodePack implements a form of prefetching), they often use sequential and computationally expensive compression techniques (*e.g.*, Huffman), resulting in significant runtime overhead. In addition, they store uncompressed code in the instruction cache, so the cache does not benefit from a compressed image and the hardware must map uncompressed addresses to compressed ones. Finally, the unit of compression is limited to the cache line, so individual instructions (or bytes) are compressed rather than instruction sequences.

DISE performs post-fetch decompression, allowing the instruction cache to store compressed code while maintaining a single static (compressed) image which does not require address translation structures. Implementations of post-fetch dictionary decompression using custom hardware has been previously proposed. One such system [59] uses a very large dictionary (up to 8K entries, each consisting of one or more instructions) and 16-bit codewords (which admit the compression of single instructions) to achieve impressive code size reductions on PowerPC binaries. Another post-fetch decompression system [61] uses variable length codewords and dictionary-based compression of common instructions (not instruction sequences). Our implementation uses general-purpose hardware, a small dictionary, and supports both parameterized and programmable decompression. Although not a fundamental limitation of DISE, our scheme currently uses only 32-bit word-aligned codewords.

Operand factorization [7] extends post-fetch decompression. Building on the observation that compressing whole instructions—*i.e.*, opcodes and operands together—limits the efficacy of a compression algorithm, operand factorization compresses opcodes (tree patterns) and operands (operand patterns) separately. After fetch, tree and operand patterns are decompressed and reassembled to form machine instructions. Operand factorization is effective for very large dictionaries. Via register/immediate parameterization, DISE supports a limited form of operand factoring within the framework of an existing mechanism.

Nam *et al.* propose a VLIW post-fetch decompression system that supports a variant of operand factorization [67]. Individual VLIW instruction words are compressed to indices in opcode and operand dictionaries. The number of instructions encoded by a single compressed instruction is a function of the number of operations that appear in each VLIW instruction (*i.e.*, longer or shorter encodings are not possible). Nam *et al.* also find that compressing similar but not identical sequences (via *instruction isomorphism*) dramatically improves compression effectiveness.

6.5 Summary

Code compression/decompression is an important tool for architects of both embedded and general purpose microprocessors. In this chapter, we present and evaluate an implementation of dynamic code decompression based on DISE. A DISE implementation of decompression has many advantages. It implements post-fetch decompression, allowing the instruction cache to benefit from a compressed program image and removing the need for mechanisms for translating uncompressed addresses to compressed ones. DISE's matching and parameterized replacement functionality supports parameterized compression, enabling better dictionary space utilization. DISE's programmability also allows individual applications to exploit custom dictionaries. Perhaps, DISE's most compelling advantage, however, is that it has many other applications besides code decompression, making its inclusion in system design easier to justify than other proposed decompression mechanisms.

This chapter includes an extensive experimental evaluation in which we not only measure code compression itself, but also evaluate its impact on dynamic characteristics such as performance and energy. We show that DISE enables code size reductions of 25% to 35% and often results in better compression than previously proposed custom compression hardware. We measure the impact of DISE-specific features or attributes, such as parameterization, branch compression, and demand-loading the dictionary. We find that the most unique DISE feature (versus other hardware approaches to dynamic decompression), parameterization, dramatically improves its ability compress (by up to 20%) and allows PC-relative branches to be compressed. We also evaluate a number of issues that have implications for any post-fetch decompression mechanism. For example, we find that application-customized dictionaries enable better compression than fixed dictionaries, and a hybrid of the two is better still. We find that very large dictionaries are unnecessary. We also quantify the impact of compression on performance (in some case a 20% improvement) and energy (in some cases a 10% reduction).

Chapter 7

Security with DISE

Computer attacks that exploit software security vulnerabilities are a significant source of lost time, lost data, lost revenue. These vulnerabilities arise from the use of unsafe programming languages (C and C++, in particular) despite the availability of type-safe programming languages (*e.g.*, Java) as well as type-safe variants of unsafe languages [8, 47, 68]. The continued use and deployment of code written in unsafe languages is a result of practical concerns, legacy codes, and inertia. Moreover, removing the vulnerabilities from the static program is a costly endeavor in C [48, 75].

This chapter demonstrates the use of DISE in dynamically detecting two common forms of attack: *stack smashing* and *pointer smashing* [24, 26]. Both attacks can severely compromise the security of a system. In these attacks, the adversary exploits either a bounds-unchecked array copy (called *buffer overflow*) or an improper use of format strings to corrupt a memory-resident return address or pointer. By corrupting a return address or pointer, the attacker can cause the program to transfer control to either attacker-defined code (called *code injection*) or non-attacker-defined code such as a libc routine (called *arc injection*).

We use two existing techniques for preventing these two attacks. We prevent stack smashing by dynamically verifying that each return address stored on the stack is not corrupted during the lifetime of the function with which it is associated (as done in Lib-verify [11], Return Address Defender [18], and StackGhost [38]). We prevent pointer smashing by storing pointers in memory in encoded form (as done in PointGuard [28]). Attackers cannot give pointers *particular* values because the attacker does not know the encoding key. DISE can protect return addresses without compiler support (*i.e.*, it can protect legacy code), while pointer protection in DISE requires recompilation to identify loads and stores of pointers. Notice that our techniques do not prevent all types of attacks, but rather prevent only stack and pointer smashing attacks.

Although both of these techniques have previously been implemented in software [11, 18, 27, 38, 28], a DISE-based attack detector has two virtues. First, as discussed in previous chapters, DISE does not perturb the instruction cache, and thus, efficiently provides attack detection. In addition, DISE separates the applications from the detection code, the latter of which is simply a transformation specification. With this separation, DISE attack detection is highly flexible. If a new attack is discovered in which DISE can be used to thwart, then only the separate attack detection code needs to be updated. A related benefit is that system administrators or users may make per-system or per-application choices concerning the level of protection (if any) they require without recompilation. A DISE-based attack detector retains the above virtues even in instances when compiler support is necessary (*i.e.*, to identify pointer loads and stores for pointer protection), because the compiler is used to identify and annotate potential vulnerabilities rather than embed the actual detection mechanism.

Alternatively, researchers have proposed using hardware widgets to detect stack and pointer smashing attacks [29, 49, 55, 65, 83, 87, 95, 97]. But these are more inflexible than software approaches. To modify or augment the attack detection techniques requires new hardware. Furthermore, unlike DISE, these hardware widgets are dedicated to attack detection.

The outline for this chapter is as follows. Sections 7.1 and 7.2 describe and evaluate a DISE-based implementation for protecting memory-resident return addresses and pointers. Section 7.3 discusses some related work in software security.

7.1 DISE-Based Attack Detection

DISE can be used to protect stack-resident return addresses and memory-resident pointers.

7.1.1 Return Address Protection

We describe a DISE implementation of a mechanism for protecting return addresses from stack-resident buffer-overflow attacks [24]. We use a previously-proposed shadow stack approach [11, 18, 38]. The basic functional design is simple. We maintain a heap-based shadow stack that mirrors the return addresses stored in the call stack. At each function return, we check that the actual return address matches the address on top of the shadow stack and alert the OS on a mismatch.

Our implementation requires two specifications for transforming function calls, and returns. It also uses two auxiliary DISE routines and a region of memory for storing the shadow stack (see Chapter 3). Finally, our implementation uses several DISE registers, which we refer to mnemonically as **$dscr** (scratch), **$dssb** (shadow stack base), **$dssp** (shadow stack pointer), and **$darp**, which points to the top of the currently allocated shadow stack region.

Maintaining the shadow stack. Shadow-stack management is performed by transformation on call (**jsr** and **bsr**) and return (**ret**) instructions. The replacement sequence in the call specification (top of Figure 7.1) computes the return address using the trigger's own program counter, pushes it onto the shadow stack (along with the stack pointer, which is discussed below), checks for shadow-stack overflow (calling **expand()** if necessary), and performs the original call (**T.INST**). If **expand()** is called, it will allocate a larger stack region (similar to **malloc()**), copy the old stack into the new buffer, and update the DISE registers to reflect the new location. The return replacement sequence (bottom) pops the shadow stack and performs the original return (again, **T.INST**).

Verifying return addresses. In addition to popping the shadow stack, the return instruction's replacement code verifies that the intended or actual return address matches the address at the top of the shadow stack. The replacement sequence compares the popped address to the address specified by the return's source register (**T.RB**). On a match—this is the common case—the original return instruction is executed. On a mismatch, the **addrcheck()** function is called.

Normally, **addrcheck()** will terminate the program because address mismatch indicates tampering. However, there are circumstances in which return address mismatches are

```
T.OPCLASS == call              # match on calls
=> addq T.PC,4,$dscr           # compute return addr
   stq $dscr,0($dssp)          # push return addr...
   stq $sp,8($dssp)            # ...and stack pointer
   addq $dssp,16,$dssp         # ...on shadow stack
   cmpeq $dssp,$darp,$dscr     # stack full?
   d_ccallne expand,$dscr      # yes? then call expand
   T.INST                      # perform call

T.OPCLASS == return            # match on returns
=> subq $dssp,16,$dssp         # pop address off...
   ldq $dscr,0($dssp)          # ...of shadow stack
   cmpeq $dscr,T.RB,$dscr      # comp. to return addr
   d_ccalleq addrcheck,$dscr   # diff? then call error
   T.INST                      # perform return
```

Figure 7.1: DISE transformation specifications for return address protection.

legal. The use of non-local returns (*e.g.*, exceptions or **setjmp()**/**longjmp()**) will cause the system to falsely report a corrupted return address. Previous systems [18, 38] handled these situations by repeatedly popping the shadow stack until an address match is obtained, terminating the program only when the shadow stack underflows. This solution has two drawbacks. First, it allows the shadow and runtime stacks to get out of sync when multiple instances of the same call site are active. Second, it does not prevent an attacker from diverting control to arbitrary locations in the call chain. We solve this problem by pushing the current stack pointer (**$sp**) along with the return address onto the shadow stack. On a return address mismatch, we repeatedly pop shadow stack entries (within **addrcheck()**) until the return address and stack pointer *both* match. We depend on the fact that the stack pointer itself is not stored in memory and can be reliably used to identify the calling context of a function and thus distinguish benign non-local returns from malicious ones. If **addrcheck()** recognizes a non-local return and returns to the replacement sequence without terminating the program, the actual return instruction (**T.INST**) is executed and program execution continues.

Figure 7.2 shows the **addrcheck()** routine. The routine starts by saving some registers to memory. It then uses **d_mfdr** to move values from DISE registers to conventional registers. The core of **addrcheck()** is a loop that scans the shadow stack looking for an entry with a matching return address and stack pointer. It starts by popping the top of the shadow

```
# save registers $1-$4 to DISE memory region (not shown)

# move state from DISE registers to conventional registers:
d_mfdr $1,$d0       # get the problem return address
d_mfdr $2,$dssp     # get the shadow stack pointer
d_mfdr $3,$dssb     # get the shadow stack base address

loop:               # start of loop
subq $2,16,$2       # pop top of shadow stack
cmpeq $2,$3,$4      # compare shadow stack pointer to base address
bne $4,no_match     # same? then branch to no_match
ldq $4,0($2)        # else get address on top of shadow stack
cmpeq $4,$1,$4      # compare address to problem return address
beq $4,loop         # diff? then goto next loop iteration (skip sp check)
ldq $4,8($2)        # (otherwise check sp) get sp off of top of shadow stack
cmpeq $sp,$4,$4     # compare stack sp to current sp
bne $4,match        # same? then goto match
br loop             # else continue looping (end of loop)

no_match:           # no match found
bsr error_handler   # call error handler (doesn't return)

match:              # match found
subq $2,16,$2       # pop top of shadow stack

# restore registers $1-$4 (not shown)

d_ret               # DISE return
```

Figure 7.2: The **addrcheck()** routine.

stack (since this entry was checked within the replacement sequence) and checking if the next entry has a matching return address and stack pointer. If it does, then **addrcheck()** breaks from the loop, pops the top of the shadow stack, restores the used registers (not shown), and returns using a DISE return (**d_ret**). Otherwise, **addrcheck()** continues to the next iteration of the loop. Although the overhead from executing this code is expensive, **addrcheck()** will be called only in rare cases (*e.g.*, on a **setjmp()**/**longjmp()**). Furthermore, if **addrcheck()** finds that the return address was corrupted, then the overhead is irrelevant because the program will be terminated.

Compiler support. Like the Alpha, most ISAs have instructions for function calls and returns, so they require no compiler support to protect return addresses via DISE. Those that do not, require the compiler to instrument the program with DISE codewords at function call and return. These will expand into instruction sequences to manipulate the shadow stack.

Protecting the shadow stack itself. In addition to detecting return address corruption, we also detect corruption of the shadow stack, itself. We use a previously-proposed technique [11, 18] that exploits virtual memory protection. We sandwich the shadow stack between two unused, write-protected pages (*e.g.*, via **mprotect()**), thus preventing any buffer from overflowing into it.

Benefits of DISE. In a DISE implementation of return address protection, the application and attack detection mechanism are separate, making the implementation conceptually simpler, more efficient, and more flexible. Unlike static transformation approaches, DISE return address protection can operate on legacy binaries, dynamically-linked code, and even dynamically-generated code. Unlike hardware approaches, a DISE-based attack detector is far more general, able to detect other attacks (*e.g.*, pointer corruption) or implement non-security-related transformations (*e.g.*, for profiling, code decompression, or debugging). Finally, a DISE implementation has a software-distribution advantage. A distributed patch, which is simply the new DISE specifications, can be applied transparently to all applications. The equivalent software patches must be distributed and applied on an application or dynamically-linked-library basis.

7.1.2 Pointer Protection

Protecting return addresses prevents stack smashing, but there are other ways to subvert a running program. If an attacker can corrupt a pointer (function or data) then the attacker may also be able to achieve a code or arc injection attack [71]. To prevent such attacks, we use a previously proposed technique [28] that encodes pointers (including array pointers) while in memory. Attackers may still corrupt pointers, but they can not give them particular values because they do not know the encoding key. As a result, corruption will likely cause a program crash rather than subversion.

```
T.OPCODE == res1           # match "store" codewords
=> xor T.RA,$dxr,$dscr     # encode pointer
   stq $dscr,T.IMM(T.RB)   # perform the (patched) store

T.OPCODE == res2           # match "load" codewords
=> ldq $dscr,T.IMM(T.RB)   # perform the (patched) store
   xor $dscr,$dxr,T.RA     # decode pointer
```

Figure 7.3: DISE transformation specifications for pointer encoding/decoding.

We achieve pointer protection in DISE by replacing pointer-manipulating load and store instructions with DISE-aware codewords during compilation. The DISE mechanism is programmed with corresponding specifications that expand to loads or stores along with the appropriate encoding and decoding logic. Although recompilation is necessary, replacement sequence selection (which may happen as late as load time) determines what encoding mechanism (if any) is to be used.

Encoding/decoding. Any reversible encoding technique may be used, but performance concerns exclude true encryption in most contexts. In most cases, it is sufficient to use computationally efficient encodings via **xor**, bit-wise permutation, bit-wise rotation, arithmetic operations (*e.g.*, add and subtract), or combinations of these.[1] **Xor**-based store and load specifications are shown in Figure 7.3. The store specification matches on codewords using reserved opcode **res1** and the load specification matches on codewords using reserved opcode **res2**. The encoding key, which is determined at application startup, is kept in DISE register **$dxr**.

Strong encryption/decryption. As Tuck *et al.* described [87], an attacker can exploit a *read buffer overflow* vulnerability to uncover encoded pointer values. A read buffer overflow attack is similar to a conventional buffer overflow attack except the vulnerable code reads from an array rather than writing to one. The attacker may also know the unencoded pointer value, and using cryptanalysis may be able to determine the key given a weak encoding scheme (*e.g.*, **xor**). This attack, however, must be executed while the program is running since a different key is generated for each running process. Even still, in

[1]Naturally, arithmetic, rotational, and permuted encodings should not be used alone; their value comes in using them with other transformations.

some circumstances, this attack may be a realistic threat. In these cases, strong encryption/decryption techniques should be employed. To make this efficient, encryption and decryption needs to be implemented in hardware. For example, Tuck *et al.* [87] propose adding two new instructions to the ISA: **e-store** (store encrypted) and **d-load** (load decrypted). There are many ways to implement these instructions. Tuck *et al.* propose using either a random permutation table or for higher security, hardware-implemented AES [30]. In processors that support these instructions, DISE can easily take advantage of them. The instructions in the two replacement sequences in Figure 7.3 are simply replaced with an **e-store** and a **d-load**, respectively. Because the application and protection mechanism are separate, the encoding or encryption technique can be decided by the end user. If higher security is necessitated and a processor has hardware support for encryption/decryption, then encryption can be utilized without recompiling or statically transforming the application.

Compiler support. The compiler must statically identify loads and stores that manipulate pointers and replace them with corresponding DISE codewords. We have not implemented this compiler support ourselves, however, this support has been incorporated into GNU gcc by Cowan *et al.* in their implementation of PointGuard [28]. Instead, in our evaluation of pointer protection in Section 7.2, we use a dynamic analysis tool to conservatively approximate pointer loads and stores.

Unfortunately, compiler support limits application portability. Statically-inserted codewords will not execute on machines without DISE. Fortunately, we can still achieve portability by utilizing trap handling. When a codeword is executed, a non-DISE processor will trap to the operating system. The application can run normally (and with protection) by registering a trap handler that executes the appropriate replacement sequence. The semantics of the program and the protection mechanism will be preserved although at a greater performance cost. However, as shown in the evaluation in Section 7.2, pointer loads and stores are infrequently executed, making the performance degradation reasonable.

Separating DISE code and application code. As described in Chapter 3 (page 36), when compiling an aware application users have the option of storing DISE code (specifications and auxiliary code) within the program binary. In the context of security, users should instead leave the binaries separate so that the DISE code can be recompiled without requiring the application to be recompiled. When new attacks arise, only the DISE code requires recompilation. Furthermore, one DISE binary can be used to transform many applications.

Benefits of DISE. The separation of application and attack protection mechanism allows users and system administrators to stay ahead of attackers without recompilation. Once a program has been recompiled once to identify loads and stores of pointers, we may protect those pointers using any mechanism we like via replacement sequence selection. When attackers have subverted simple **xor**s [87], we may use encodings also involving bit-wise rotation and permutation. We can even use strong encryption if the context warrants it. By customizing the encoding mechanism per system or even per process, attack is made more difficult. Customization in static pointer-protection schemes would likely require frequent compilation. In addition, a DISE-based pointer protector benefits from DISE's private register space. The running program is not permitted to see DISE registers, and therefore, it is more difficult for attackers to learn encoding keys directly. Conversely, a static implementation is likely to spill the encoding key (residing in a standard, non-DISE register) to memory on function calls, exposing it to read buffer overflow attacks [87].

7.2 Evaluation of DISE-Based Attack Detection

We use cycle-level simulation to evaluate DISE-based return address and pointer smashing attack detection, both in terms of effectiveness and performance overhead.

7.2.1 Methodology

Simulator. Our general methodology is described in Section 4.2 of Chapter 4. We simulate using SimpleScalar Alpha [16], modeling the machine from Table 4.1 on page 54. We use a 2-stage decoder-based implementation of DISE with a 32-entry pattern table and a 512-entry replacement table (see Chapter 4).

Benchmarks. Our benchmarks are selected from SPEC, MiBench [40], and CommBench [93]. Other than the SPEC benchmarks, we choose codes that would likely be

suite	benchmark	static attributes	dynamic attributes					
		code size (insn)	IPC	calls	loads	stores	ptr. loads	ptr. stores
SPECint 2000	bzip2	36,013	2.37	.51%	24.45%	16.93%	2.96%	1.27%
	eon	150,998	2.13	1.94%	25.14%	17.41%	5.80%	2.24%
	mcf	32,018	1.36	2.54%	21.89%	14.87%	9.05%	2.57%
	twolf	88,324	1.84	1.07%	23.77%	7.53%	7.54%	.78%
MiBench	bfish.enc	15,923	2.86	.87%	22.33%	10.67%	5.56%	4.04%
	crc	27,447	3.83	∼0%	21.74%	13.04%	4.35%	4.35%
	patricia	36,811	1.82	2.04%	22.24%	9.79%	7.98%	2.60%
	sha	22,057	3.00	.03%	13.47%	3.63%	.01%	∼0%
Comm-bench	cast.enc	15,301	2.47	.60%	21.56%	9.42%	4.21%	2.90%
	drr	27,486	2.16	∼0%	35.48%	9.39%	24.57%	8.48%
	reed.dec	13,582	2.59	∼0%	12.24%	5.69%	.01%	∼0%
	rtr	38,522	2.59	1.65%	25.47%	6.77%	9.10%	3.27%

Table 7.1: Benchmark summary.

sensitive to attacks (*e.g.*, those running with root permissions or in a trusted piece of hardware) and thus would benefit from protection. For brevity, we show only a subset of the three benchmark suites. However, the chosen benchmarks are representative of all the benchmarks in all three suites. Table 7.1 summarizes benchmark characteristics useful in interpreting the presented results.

Compiler support. Detecting corruption of memory-resident pointers requires compiler support. Rather than build or extend a compiler, we have built binary analysis tools to approximate the necessary static analysis and transformation. We use simple static analysis and dynamic profile information to conservatively identify memory operations that manipulate pointers. This solution is not satisfactory in general, but it is representative of true static transformations. It is used only in estimating performance overhead.

Binary rewriter. We compare DISE-based attack detection with a binary rewriter implementation. The binary rewritten code contains exactly the same instructions as the DISE transformed code (after dynamic instrumentation) because the former are not statically optimized. However, because these transformations instrument loads, stores, and control flow instructions, there is little opportunity for static optimization.

7.2.2 Protection Effectiveness

Here we show that using DISE, we can detect real attacks on vulnerable code. Furthermore, for the benchmarks and inputs that we used, our techniques detected all attacks and did not signal an attack when there was none.

Protecting return addresses. None of our benchmarks are vulnerable to return address corruption, so we identify three other vulnerable programs: *overflow1*, *gzip-1.2.4*, and *sendmail-8.7.5*. The first was presented in a hacker's tutorial on buffer-overflow attacks [4] and represents a prototypical vulnerability. The others are vulnerable versions of well-known codes. In all three cases, our DISE implementation successfully detects an input attack and terminates the program. At the same time, non-malicious inputs do not spuriously signal an attack for these three programs or any of the benchmarks used to evaluate performance.

Protecting pointers. The effectiveness of pointer encoding has been demonstrated elsewhere [28, 87], but we confirm these results with a synthetic fault injector. Our fault injector changes the value read by a random pointer load to a random value, simulating the corruption of a single pointer in memory. We use this fault injector to "attack" each benchmark ten times, one fault per attack. In the majority of cases, the program crashes. On a few occasions corruption does not result in a crash because the corrupted value is not used after it is loaded. In most cases, the program crashes very soon after the pointer is corrupted. In addition, we find no false positives, *i.e.*, programs crashing when there is no attack.

7.2.3 Performance Overhead

The overhead of the two protection mechanisms in DISE is shown in Figure 7.4 (shaded bar). Because function calls, returns, and pointer memory operations are relatively infrequent (see Table 7.1) the overhead of both return address protection (*RAP*) and pointer protection (*PP*) is usually small (*i.e.*, less than 10%). Benchmarks with higher function call density suffer a higher return address protection overhead (*e.g.*, *eon*, *mcf*, and *rtr*). Similarly, benchmarks with relatively more pointer loads and stores suffer higher overhead for pointer protection (*e.g.*, *drr*, *rtr*, and *twolf*). Nevertheless, in all cases, the overhead

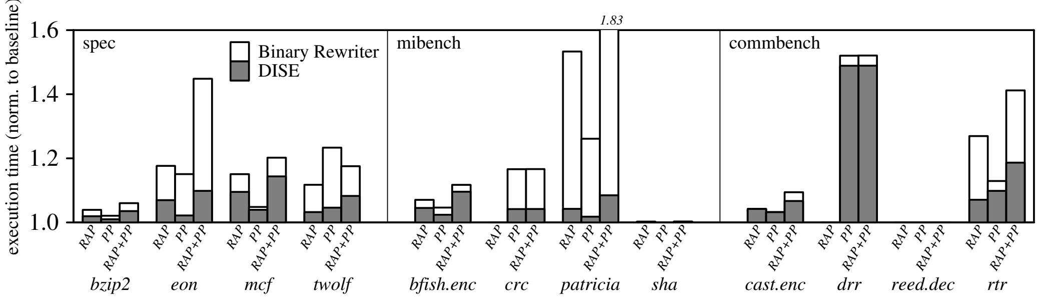


Figure 7.4: Overhead of return address protection (RAP), pointer protection (PP), and the combination of both techniques (RAP+PP) using DISE and binary rewriting.

for protecting return addresses and pointers is less than 50%.

Combining the techniques. Figure 7.4 also shows protection overhead when combining these techniques (*RAP+PP*). The overhead of combining the two techniques is naturally additive because each injects code at different places in the program (calls/returns versus memory operations). As with protecting return addresses or pointers separately, the overhead is always less than 50% and usually less than 20%. A virtue of using DISE is that system administrators can choose the protection mechanisms. For systems that require higher performance, administrators can tradeoff protection for overhead (*e.g.*, turning off pointer protection for workloads such as *drr*).

Versus static transformation. Although software-only static transformation sacrifices the benefits of DISE, it is still a natural approach to dynamic attack detection. Figure 7.4 also shows the overhead of a static binary rewriting implementation. In some cases, the performance is very similar (*e.g.*, *bzip2*, *bfish.enc*, and *cast.enc*), while in other cases the DISE-based approach performs much better (*e.g.*, *eon*, *patricia*, *rtr*). The differences arise due to poor instruction cache performance in the static case. The benchmarks with the larger code sizes (from Table 7.1) generally have larger differences in performance. The overhead of dynamic software translation (translation at load/runtime) would be worse then the overhead of the compiler-based approach in Figure 7.4, due to the additional cost of the translation.

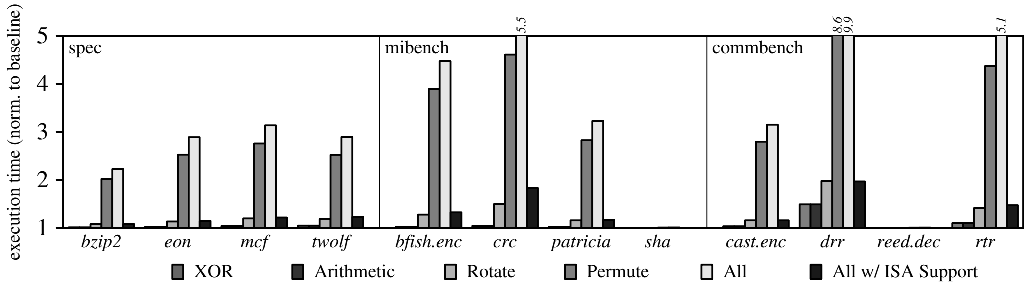


Figure 7.5: Performance impact of pointer encoding.

7.2.4 Customizing Attack Detection

As described in the previous section, DISE attack detection is customizable. Below we evaluate the performance impact of customizing the pointer-encoding technique.

Customizing the encoding. DISE is flexible in that any set of protection mechanisms may be applied to a single program. This allows users to select an appropriate protection level, but it also contributes to the strength of security itself. For example, consider pointer protection. Encoding pointers via **xor** is efficient but not particularly strong. Tuck *et al.* describe an attack that exploits this weakness [87]. With DISE, one may change the encoding algorithm on a per-system or per-application basis. Subversion is made more difficult because the attacker does not know the encoding key nor the algorithm used. Figure 7.5, shows five example encodings: **xor**, arithmetic (*e.g.*, add and subtract), bit-wise rotate, byte-wise permute, and all five together. Rotate and permute may require multiple instructions to implement (*e.g.*, as in the Alpha ISA), so they are much more expensive then **xor** or arithmetic. Rotation can be achieved using 4 Alpha instructions so it generally has a reasonable overhead (less than 50%). Permute, on the other hand, requires 31 instructions, so it is expensive. Note that the overheads of *sha* and *reed.dec* are negligible since they have very few loads and stores of pointers.

When all four encodings are combined (second-to-last bar in Figure 7.5), performance is limited by the very expensive permute. If an architecture provides bit-wise permute and rotate instructions, the overhead is much lower (the final bar).

Strong encryption/decryption. For some systems, strong encryption may be more desirable than weaker encoding schemes like those shown in Figure 7.5. To achieve acceptable

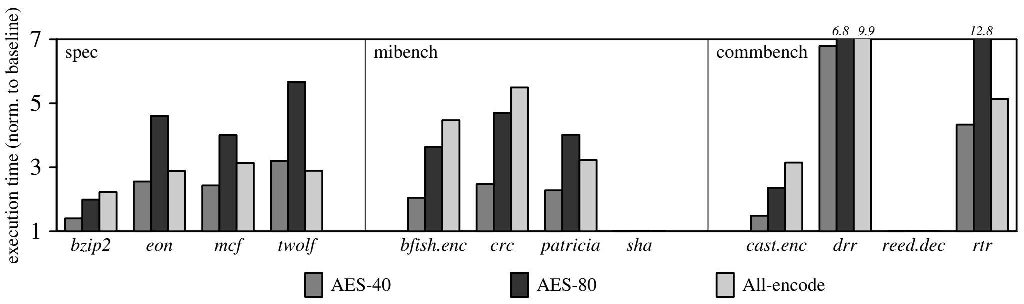


Figure 7.6: Performance impact of hardware-supported pointer encryption.

performance, encryption must be implemented in hardware. Because of the flexibility of DISE, utilizing hardware support for encryption can be done without recompiling the application.

Figure 7.6 shows DISE pointer protection utilizing hardware support for AES. *AES-40* and *AES-80* represent the performance with a 40-cycle, encryption/decryption latency and a 80-cycle, encryption/decryption latency, respectively. For comparison purposes, we show the *All* bar (without ISA support) from Figure 7.5 (in Figure 7.6 called *all-encode*). As we would expect, the overhead when using hardware-implemented AES is high for benchmarks with a significant number of pointer loads and stores and low for the others.

For many benchmarks, the overhead of *AES-40* is similar to that of *All-encode*. In *All-encode* we added approximately 40 instructions per pointer memory instruction, while in *AES-40* the overhead of a pointer memory instruction is 40 cycles. Often, these two transformations are equivalent. However, *All-encode*, unlike *AES-40*, reduces machine resources due to the additional instructions. For benchmarks with higher IPCs (*e.g.*, *bfish.enc* and *crc*), *All-encode* does significantly worse than *AES-40* since they are more sensitive to the loss in resources. In summary, *AES-40* outperforms *All-encode* and provides higher security. If machines support a 40-cycle encryption operation in hardware, it should be utilized over the more expensive encoding operations from Figure 7.5 (*e.g.*, permutation).

Of course, the overhead of *AES-80* is always higher than *AES-40*. In addition, the overhead of *AES-80* is sometimes lower than *All-encode* (*e.g.*, *drr*), but usually significantly higher (*e.g.*, *eon*, *twolf*, and *rtr*). The replacement code in *AES-80* does not use as many machine resources, however, the latency (at least, 80 cycles) is often higher than the latency of the *All-encode* replacement sequence. These two bars represent a tradeoff between security and performance. When machines support a 80-cycle encryption instruction in hardware (and not a 40-cycle instruction), users concerned primarily with security should use *AES-80*, while users concerned more with performance should use *All-encode* (or some other encoding scheme that has even lower overhead). Because of the flexibility of DISE, switching between any of the encryption or encoding techniques shown in Figures 7.5 and 7.6 can be done without recompiling the application.

7.3 Related Work in Software Security

Both software and hardware techniques have been proposed to protect software from security vulnerabilities.

Software techniques. Bounds checking can be added by the compiler to C programs [48, 75] to prevent buffer overflow attacks. This provides a higher level of security, then dynamic attack detection, albeit at a higher overhead. In addition, this technique will not prevent attacks that exploit improper uses of format strings and in some circumstances results in false positives. Libsafe [11] performs bounds checking on stack-resident buffers referenced by C standard library routines (*e.g.*, **strcpy()**) via stack inspection. This approach is more efficient than general bounds checking, although more limited in scope.

Some techniques, like those discussed in this chapter, forgo detecting vulnerabilities (*e.g.*, unchecked buffer access) and instead detect the subsequent attack. A host of compilation techniques (*e.g.*, StackGuard [27], Libverify [11], Return Address Defender (RAD) [18], and StackGhost [38]) have been proposed to detect return address corruption. In this chapter, we implemented return address protection via a shadow stack as described in three of these works [11, 18, 38]. Some software techniques leverage dynamic software translation (*e.g.*, program shepherding [53], Strata [79]). Like DISE, these provide the flexibility of altering or modifying the attack detection mechanism, but with a higher performance overhead due to the additional cost of translation.

Cowan *et al.* thwart a broader class of code and arc injection attacks that rely on pointer corruption by encoding pointers (and only pointers) when they are written to memory [28].

Since attackers do not know the application-specific encoding key, they are unable to corrupt pointers in memory with particular values. In this chapter, we showed an implementation of this technique in DISE.

Some recent proposals have explored type-safe variants of the C programming language [8, 47, 68]. These proposals provide higher security than the techniques in this chapter, however, they require some changes to the C programming language.

Hardware techniques. Return address protection has also been implemented in hardware [65, 95, 97]. Kirovski *et al.* [55] propose hardware and software installation support to encrypt programs based on a key hidden in the hardware. Kc *et al.* [49] proposed hardware support for instruction-set randomization. Both techniques prevent code injection attacks, but neither prevents arc injection attacks. Suh *et al.* [83] and Crandall and Chong [29] propose implementing data-flow monitoring in hardware to track spurious (*i.e.*, untrusted) data identified by the operating system. Whenever spurious data is used in an insecure way (*e.g.*, as the target of a jump), the program is killed. These techniques detect a broader class of attacks, at the expense of false positives and non-trivial, security-specific processor modifications. Tuck *et al.* [87] propose hardware support for PointGuard [28] with stronger encryption techniques to prevent some forms of read buffer overflow attacks.

7.4 Summary

DISE is an effective mechanism for dynamically detecting stack and pointer smashing attacks. As discussed in previous chapters, DISE attack detection is efficient because instruction macro-expansion latency is negligible and because DISE has no impact on instruction cache performance. In addition, a DISE-based attack detector separates application and detection code, dynamically injecting the latter as the former runs. The flexibility of this structure allows a single program (once transformed to identify vulnerabilities) to be protected in many different ways based on the needs of the user or system administrator. It also allows one to apply new protection techniques to programs without rebuilding them. Finally, DISE is a general purpose mechanism that can not only be used to detect stack and pointer smashing attacks, but also to profile (Chapters 3 and 4), decompress a program (Chapter 6), or to watch a memory location in an interactive debugging environment (Chapter 5). In summary, we find that stack and pointer smashing attack detection via DISE is effective, flexible, and efficient.

Chapter 8

Conclusions

In this chapter, we summarize our findings in this dissertation and describe some future directions of this work.

8.1 DISE Summary

The field of computing is no longer performance-dominated like it once was. Today's users are concerned with many other characteristics including security, reliability, and program size. Unfortunately, not all users care equally about each characteristic, and therefore, we need to customize programs for each individual user and program.

Customization can be achieved through program transformation. Many various transformation mechanisms exist, all implemented in software (*e.g.*, binary rewriters, software dynamic translators). Unfortunately, a software translator splices the additional code into the program, degrading instruction cache performance. The transformation cost is also high, and if translation occurs at runtime is perceived as additional overhead.

To address these performance shortcomings, many researchers have proposed hardware widgets to customize the program (*e.g.*, monitoring execution) within the processor. Hardware widgets have low overhead, but they are not programmable and are generally dedicated to one particular type of customization (*e.g.*, profiling).

In this dissertation, we propose a hybrid approach: a programmable hardware mechanism. We call this facility a *dynamic instruction stream editor* or *DISE,* for short. DISE transforms programs using instruction macro-expansion. It takes as input individual instructions and outputs instruction sequences based on the inputted instruction. It inspects every fetched instruction, macro-expanding on some of the instructions, and the translated instruction stream is passed to the decoder.

Because DISE is programmable, transformations are easily added, removed, or modified. In addition, this dissertation demonstrates the formulation of many different types of transformations in DISE, including transformations for profiling (Chapter 3), debugging (Chapter 5), decompression (Chapter 6), and security (Chapter 7). Because DISE is a hardware mechanism, transformation is also efficient. The cost of transforming instructions is negligible (although there is the overhead of executing the transformed code). In addition, the program image is unaltered and consequently instruction cache performance does not suffer. In summary, we find that DISE is a flexible and efficient mechanism for transforming (and customizing) programs.

8.2 Future Directions

Below we present three future directions of this work.

System-level transformation. In this dissertation, we proposed a user-level tool (*i.e.*, DISE) for transforming applications. In future work, we will explore a system-level DISE. For example, if DISE could transform kernel code, then it could potentially be used to fault isolate device drivers (as done in Nooks [84]). However, we would need to prevent users from subverting the operating system via DISE. For instance, DISE should not be allowed to match on programming instructions such at **d_toggle** or **d_sync**. Otherwise, a user-level application could prevent the OS from disabling DISE. In this dissertation, because DISE cannot match on kernel-level code, matching on programming instructions is not a problem. In addition, when the machine is booted, DISE should be disabled.

In addition to transforming kernel code, we will explore system-controlled DISE transformation. If system administrators could transform user-level applications in arbitrary ways, it would allow them to perform system wide transformations (*e.g.*, profiling all applications) or to enforce security transformations (*e.g.*, stack and pointer smashing) in certain classes of applications (*e.g.*, network accessible applications). There are many challenges in the implementation of system-controlled DISE transformation. First, administrator-defined transformations should not be subvertable by the user or application. Furthermore, the system should be able to compose transformations submitted by multiple parties for the same application. In such cases, the higher-privileged transformations (*i.e.*, administrator's) should not be undone by lower-privileged transformations (*i.e.*, user's).

More advanced pattern matching. Another area of future work is exploring more advanced pattern matching in DISE. For example, allowing DISE to match on a window of instructions rather than only one instruction at a time. Furthermore, allowing DISE to match on dynamic properties of an instruction, such as the value in one of its register operands. These two extensions would potentially allow users to perform optimizations such as silent store elimination [62]. The main challenge is in the implementation. Can this be efficiently implemented, preferably with as little impact as possible on non-transformed programs? What is the impact on microarchitecture design? Furthermore, do the additional uses of DISE justify the implementation costs?

Abstract interface. As discussed in Chapter 3, DISE is programmed (at the lowest level) using a binary representation of patterns and replacement sequence. Of course, if this interface changes in the future then previously-written specifications will no longer be compatible. Furthermore, the low-level interface encodes machine instructions into pattern and replacement instruction representations. Therefore, if the ISA changes, the DISE interface will also have to change. In future work, we will explore an abstract interface of DISE. We will add library support for translating the abstract interface into the binary interface. These library routines, which will translate specifications at load time, will need to be efficient, otherwise, this will add to the startup cost of the running program.

Bibliography

[1] Advanced RISC Machines Ltd. *An Introduction to Thumb*, March 1995.

[2] A. Agarwal, R. Sites, and M. Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proc. 13th International Symposium on Computer Architecture*, pages 119–127, May 1986.

[3] D. Albonesi. Selective cache ways: On demand cache resource allocation. In *Proceedings 32nd International Symposium on Microarchitecture*, pages 248–259, November 1999.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.

[5] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Apr. 1991.

[6] Z. Aral, I. Gertner, and G. Schaffer. Efficient debugging primitives for multiprocessors. In *Proc. of 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 87–95, Apr. 1989.

[7] G. Araujo, P. Centoducatte, and M. Cortes. Code compression based on operand factorization. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 194–201, December 1998.

[8] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[9] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[10] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[11] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. of USENIX Annual Technical Conference*, Jun. 2000.

[12] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, Jul. 1970.

[13] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[14] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of Intl. Symp. on Code Generation and Optimization*, pages 265–275, Mar. 2003.

[15] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Intl. J. of High Performance Computing Applications*, 14(4):317–329, 2000.

[16] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin–Madison Computer Sciences Department, 1997.

[17] R. E. Calcagni and W. Sherwood. Patchable control store for reduced microcode risk in a VLSI VAX microcomputer. In *Proc. of the 17th Microprogramming Workshop*, pages 70–76, 1984.

[18] Tzi-Ckr Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Intl. Conf. on Distributed Computing Systems*, Apr. 2001.

[19] Yuan Chou, Jason Fung, and John Paul Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, 1999.

[20] Keith Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[21] M. L. Corliss and E. C. Lewis. A DISE framework for securing software. Technical Report MS-CIS-05-13, University of Pennsylvania, Apr. 2005.

[22] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. of 30th Intl. Symp. on Computer Architecture*, Jun. 2003.

[23] M. L. Corliss, E. C. Lewis, and A. Roth. A DISE implementation of dynamic code decompression. In *Proc. of Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 232–243, Jun. 2003.

[24] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. In *Proc. of Workshop on Architectural Support for Security and Anti-Virus*, pages 65–72, Oct. 2004.

[25] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with DISE. In *Proc. of 11th Intl. Symp. on High-Performance Computer Architecture*, pages 303–314, Feb. 2005.

[26] M. L. Corliss, E. C. Lewis, and A. Roth. The Implementation and Evaluation of Dynamic Code Decompression Using DISE. *ACM Transactions on Embedded Computing Systems*, 4(1), Feb. 2005.

[27] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention buffer overflow attacks. In *Proc. of 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of 12th USENIX Security Symposium*, pages 91–104, Aug. 2003.

[29] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of 37th Intl. Symp. on Microarchitecture*, pages 221–232, Dec. 2004.

[30] J. Daemen and V. Rijmen. The rijndael block cipher: AES proposal. URL: *http://csrc.nist.gov/encryption/aes/*, 2001.

[31] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 95–105, June 2002.

[32] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compression. *ACM Transactions on Programming Languages and Operating Systems*, 22(2):378–415, March 2000.

[33] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In *Proc. of 35th Intl. Symp. on Microarchitecture*, pages 257–268, Nov. 2002.

[34] K. Diefendorf. K7 challenges Intel. *Microprocessor Report*, 12(14), Nov. 1998.

[35] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proc. 24th International Symposium on Computer Architecture*, pages 26–38, Jun. 1997.

[36] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 358–365, June 1997.

[37] Erin Farquhar and Philip J. Bunce. *The Mips Programmer's Handbook*. Morgan Kaufmann, 1994.

[38] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. of 10th USENIX Security Symposium*, pages 55–66, Aug. 2001.

[39] P. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, 14(8), Aug. 2000.

[40] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of 4th Annual IEEE Workshop on Workload Characterization*, Dec. 2001.

[41] L. Gwenapp. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), February 1995.

[42] L. Gwenapp. Nx686 Goes Toe-to-Toe with Pentium Pro. *Microprocessor Report*, 14(9), Oct. 1995.

[43] L. Gwenapp. P6 Microcode can be Patched. *Microprocessor Report*, 11(12), Sept. 1997.

[44] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, Feb. 2000.

[45] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors in C and C++ programs. In *Proc. of Winter 1992 USENIX Conf.*, pages 125–138, Jan. 1992.

[46] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. of Scalable High Performance Computing Conf.*, pages 841–850, May 1994.

[47] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. of USENIX Annual Technical Conference*, pages 275–288, Jun. 2002.

[48] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of Intl. Workshop on Automatic Debugging*, pages 13–26, May 1997.

[49] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of 10th ACM Conf. on Computer and Communications Security*, pages 272–280, Oct. 2003.

[50] T. M. Kemp, R. K. Montoye, D. J. Auerback, J. D. Harper, and J. D. Palmer. A decompression core for PowerPC. *IBM Systems Journal*, 42(6):807–812, November 1998.

[51] P. B. Kessler. Fast breakpoints: Design and implementation. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 78–84, Jun. 1990.

[52] I. Kim and M. Lipasti. Implementing Optimizations at Decode Time. In *Proc. 29th International Symposium on Computer Architecture*, pages 221–232, May 2002.

[53] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proc. of 11th USENIX Security Symposium*, pages 191–206, Aug. 2002.

[54] D. Kirovski, J. Kin, and W. Mangione-Smith. Procedure based program compression. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 204–213, December 1997.

[55] Darko Kirovski, Milenko Drini, and Miodrag Potkonjak. Enabling trusted software integrity. In *Proc. of 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Oct. 2002.

[56] K. Kissell. *MIPS16: High-Density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[57] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 291–300, Jun. 1995.

[58] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings 30th International Symposium on Microarchitecture*, pages 330–335, December 1997.

[59] C. Lefurgy, P. Bird, I.-C. Cheng, and T. Mudge. Improving Code Density Using Compression Techniques. In *Proc. 30th International Symposium on Microarchitecture*, pages 194–203, Dec. 1997.

[60] C. Lefurgy, E. Piccininni, and Trevor Mudge. Reducing code size with run-time decompression. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 218–227, January 2000.

[61] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *Proceedings 36th Design Automation Conference*, pages 294–299, June 2000.

[62] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proc. 27th Intl. Symp. on Computer Architecture*, pages 182–191, Jun. 2000.

[63] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electrical Systems*, 4(1):12–38, January 1999.

[64] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of Conf. on Programming Language Design and Implementation*, Jun. 2005.

[65] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A processor architecture defense against buffer overflow attacks. In *Proc. of IEEE Intl. Conf. on Information Technology: Research and Education*, pages 243–250, Aug. 2003.

[66] R. Nair and M. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc. 24th International Symposium on Computer Architecture*, pages 13–25, Jun. 1997.

[67] Sang-Joon Nam, In-Cheol Park, and Chong-Min Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Transactions on Fundamentals*, E82-A(11):2318–2324, November 1999.

[68] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of 29th Symp. on Principles of Programming Languages*, pages 128–139, Jan. 2002.

[69] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proc. of Workshop on Runtime Verification*, Jul. 2003.

[70] Richard Phelan. Improving ARM code density and performance. Technical report, Advanced RISC Machines Ltd., June 2003.

[71] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, Jul./Aug. 2004.

[72] T. Rauscher and A. Argawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, 1978.

[73] Ted Romer, Geoff Voelker Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proc. of USENIX Windows NT Workshop*, pages 1–8, Aug. 1997.

[74] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architectures*. John Wiley and Sons, 1996.

[75] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. of 11th Network and Distributed Systems Security Symposium*, Feb. 2004.

[76] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4), November 1997.

[77] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[78] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proc. of Workshop on Binary Translation*, Jul. 2001.

[79] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proc. of Annual Computer Security Application Conf.*, pages 209–218, Dec. 2002.

[80] J. Seward. Valgrind. URL: *http://valgrind.kde.org/*.

[81] Richard L. Sites, editor. *Alpha architecture reference manual*. Digital Press, 1992.

[82] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 196–205, Jun. 1994.

[83] G. Edward Suh, Jae W. Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Oct. 2004.

[84] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. of 19th ACM Symp. on Operating Systems Principles*, Oct. 2003.

[85] T. Szymanski. Assembling code for machines with span dependent instructions. *Communications of the ACM*, 21(4):300–308, April 1978.

[86] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Proc. of 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, Oct. 1994.

[87] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proc. of 37th Intl. Symp. on Microarchitecture*, pages 202–220, Dec. 2004.

[88] J. Turley. Alpha Runs X86 Code with FX!32. *Microprocessor Report*, 10(3), Mar. 1996.

[89] R. Wahbe. Efficient data breakpoints. In *Proc. of 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, Oct. 1992.

[90] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 1–12, Jun. 1993.

[91] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of 14th ACM Symp. on Operating Systems Principles*, Dec. 1993.

[92] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, DEC Western Research Laboratory, 1994.

[93] Tilman Wolf and Mark Franklin. CommBench – A telecommunications benchmark for network processors. In *Proc. of IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Apr. 2000.

[94] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 81–91, December 1992.

[95] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Proc. of EASY-2 Workshop*, Oct. 2002.

[96] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings*

8th International Symposium on High Performance Computer Architecture, January 2002.

[97] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proc. 31st Intl. Symp. on Computer Architecture*, pages 224–235, Jun. 2004.

[98] C.B. Zilles, J.S. Emer, and G.S. Sohi. The use of multithreading for exception handling. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 219–229, Nov. 1999.

[99] C.B. Zilles and G.S. Sohi. A programmable co-processor for profiling. In *Proc. 7th International Symposium on High Performance Computer Architecture*, 2001.