Efficient, Transparent, and Comprehensive Runtime Code Manipulation

by

Derek L. Bruening

Bachelor of Science, Computer Science and Engineering
Massachusetts Institute of Technology, 1998

Master of Engineering, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author ______________________________

Department of Electrical Engineering and Computer Science
September 24, 2004

Certified by ______________________________

Saman Amarasinghe
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Efficient, Transparent, and Comprehensive Runtime Code Manipulation

by

Derek L. Bruening

Submitted to the Department of Electrical Engineering and Computer Science on September 24, 2004 in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

This thesis addresses the challenges of building a software system for general-purpose runtime code manipulation. Modern applications, with dynamically-loaded modules and dynamically-generated code, are assembled at runtime. While it was once feasible at compile time to observe and manipulate every instruction — which is critical for program analysis, instrumentation, trace gathering, optimization, and similar tools — it can now only be done at runtime. Existing runtime tools are successful at inserting instrumentation calls, but no general framework has been developed for fine-grained and comprehensive code observation and modification without high overheads.

This thesis demonstrates the feasibility of building such a system in software. We present *DynamoRIO*, a fully-implemented runtime code manipulation system that supports code transformations on any part of a program, *while it executes*. DynamoRIO uses code caching technology to provide efficient, transparent, and comprehensive manipulation of an unmodified application running on a stock operating system and commodity hardware. DynamoRIO executes large, complex, modern applications with dynamically-loaded, generated, or even modified code. Despite the formidable obstacles inherent in the IA-32 architecture, DynamoRIO provides these capabilities efficiently, with zero to thirty percent time and memory overhead on both Windows and Linux.

DynamoRIO exports an interface for building custom runtime code manipulation tools of all types. It has been used by many researchers, with several hundred downloads of our public release, and is being commercialized in a product for protection against remote security exploits, one of numerous applications of runtime code manipulation.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I thank my advisor, Professor Saman Amarasinghe, for his boundless energy, immense time commitment to his students, accessibility, and advice. Thanks also go to my readers, Professors Martin Rinard, Frans Kaashoek, and Arvind, for their time and insightful feedback.

Several students at MIT contributed significantly to the DynamoRIO project. I would like to thank Vladimir Kiriansky for his fresh insights and competent approach. He was the primary inventor of program shepherding. He also came up with and implemented our open-address hashtable, and proposed our current method of performing arithmetic operations without modifying the condition codes. Thank-yous go also to Timothy Garnett for his contributions to our dynamic optimizations and to the interpreter project, as well as his help with the DynamoRIO release; to Iris Baron and Greg Sullivan, who spearheaded the interpreter project; and to Josh Jacobs, for implementing hardware performance counter profiling for DynamoRIO.

I would like to thank the researchers at Hewlett-Packard's former Cambridge laboratory. DynamoRIO's precursor was the Dynamo dynamic optimization system [Bala et al. 2000], developed at Hewlett-Packard. Vas Bala and Mike Smith built an initial dynamic optimization framework on IA-32, while Giuseppe Desoli gave valuable early advice on Windows details. I would like to especially thank Evelyn Duesterwald and Josh Fisher for facilitating the source code contract between MIT and Hewlett-Packard that made the DynamoRIO project possible. Evelyn gave support throughout and was instrumental in enabling us to release our work to the public as part of the Hewlett-Packard-MIT Alliance.

I also extend thanks to Sandy Wilbourn and Nand Mulchandani, for understanding and aiding my return to school to finish my thesis.

Last but not least, special thanks to my wife, Barb, for her innumerable suggestions, invaluable help, homemade fruit pies, and moral support.

This research was supported in part by Defense Advanced Research Projects Agency awards DABT63-96-C-0036, N66001-99-2-891702, and F29601-01-2-0166, and by a grant from LCS Project Oxygen.

Contents

List of Figures and Tables

Chapter 1

Introduction

As modern applications become larger, more complex, and more dynamic, building tools to manipulate these programs becomes increasingly difficult. At the same time the need for tools to manage application complexity grows. We need information-gathering tools for program analysis, introspection, instrumentation, and trace gathering, to aid in software development, testing, debugging, and simulation. We also need tools that modify programs for optimization, translation, compatibility, sandboxing, etc.

Modern applications are assembled and defined at runtime, making use of shared libraries, virtual functions, plugins, dynamically-generated code, and other dynamic mechanisms. The amount of program information available statically is shrinking. Static tools have necessarily turned to feedback from profiling runs, but these give only an estimate of program behavior. The complete picture of a program's runtime behavior is only available at runtime.

Consider an important modern application, the web server. Figure 1.1 shows the components of a running server, highlighting which parts can be seen by the compiler, linker, and loader. Today's web servers are built for extension by third-party code, in the form of dynamically-loaded modules (e.g., Internet Server Application Programming Interface (ISAPI) components used to provide dynamic data and capabilities for web sites). Even the designers of the web server program cannot anticipate all of the third-party code that will be executed when the web server is in actual use. Tools for operating on applications like this must have a runtime presence.

A runtime tool has many advantages beyond naturally handling dynamic program behavior. Operating at runtime allows the tool to focus on only the code that is executed, rather than wasting

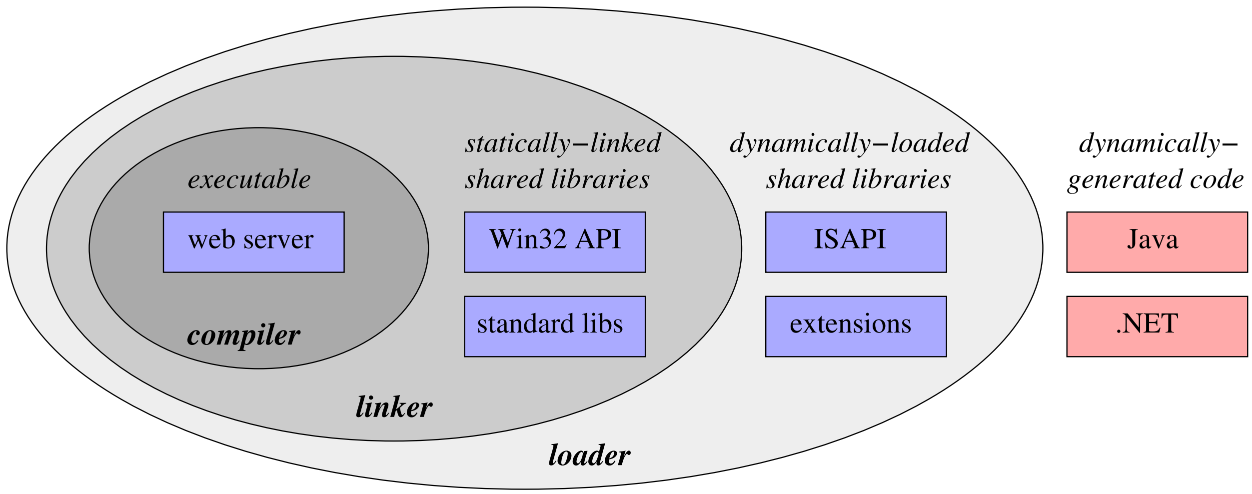

Figure 1.1: The components of a modern web server, and which can be seen by the compiler, linker, and loader. The only components that are known statically, and thus viewable by the compiler or linker, are the executable itself and the shared libraries that it imports. Neither tool knows about custom extension libraries that are loaded in dynamically. The loader can see these, but even the loader has no knowledge of dynamically-generated code for languages like Java and .NET. In modern web servers, extension modules and generated code are prevalent. In addition to missing dynamic behavior, the linker and loader have difficulty seeing inside modules: code discovery and indirect branch target resolution are persistent problems.

analysis resources (which may not matter statically but do matter if operating at load time) on never-seen code. This natural focus on executed code also avoids the code discovery problems that plague link-time and load-time tools. With a runtime view of the program, module boundaries disappear and the entire application can be treated uniformly. Additionally, runtime tools need not require the target application's source code, re-compilation, or re-linking, although they can be coupled with static components to obtain extra information (from the compiler, for example).

1.1 Goals

The goal of this thesis is to create a runtime tool platform for fine-grained code manipulation. We would like a *comprehensive* tool platform that systematically interposes itself between *every* instruction executed by a running application and the underlying hardware, as shown in Figure 1.2. Custom tools can then be embedded in this flexible software layer. In order for this layer to be

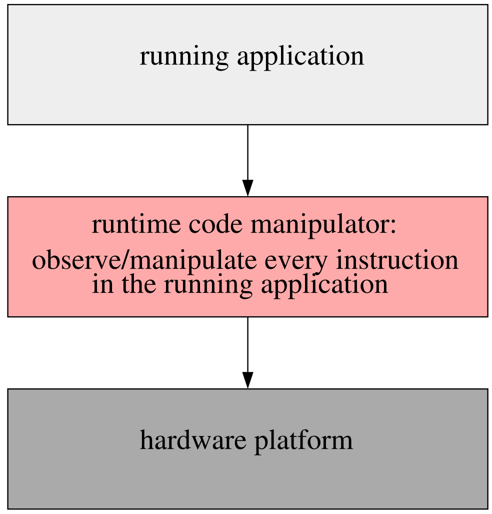

Figure 1.2: Our goal was to build a flexible software layer that comprehensively interposes itself between a running application and the underlying platform. The layer acts as a runtime control point, allowing custom tools to be embedded inside it.

maximally usable, it should be:

- **Deployable**

  The layer should be easily inserted underneath any particular application on a production system. Our target tools operate on and dynamically modify applications *in actual use*; they are not limited to studying emulated application behavior. Examples include secure execution environments, dynamic patching for security or compatibility, on-the-fly decompression, and dynamic optimization. This goal drives all of the other ones.

- **Efficient**

  The layer should amortize its overhead to avoid excessive slowdowns. Poor performance is always a deterrent to tool use, and near-native performance is required for deployment in production environments.

- **Transparent**

  The layer should operate on unmodified programs and should not inadvertently alter the behavior of any program. Transparency is critical when targeting and modifying applications in actual use, where unintended changes in behavior can have serious consequences. Even a seemingly innocuous imposition can cause incorrect behavior in applications with subtle dependences.

- **Comprehensive**

  The layer must be able to observe and modify any and all executed instructions to do more than periodic information gathering. Tools such as secure execution environments require interposition between every instruction.

- **Practical**

  To be useful, the layer must work on existing, relevant, unmodified, commodity hardware and operating system platforms.

- **Universal**

  The layer should be robust, capable of operating on every application, including hand-crafted machine code and large, complex, multi-threaded, commercial products. Operating at runtime allows us to target applications for which source code is unavailable.

- **Customizable**

  The layer should be extensible for construction of custom runtime tools.

These goals shape the design of our code manipulation layer. Some are complementary: universal and transparent work together to operate on as many applications as possible. Other goals conflict, such as being comprehensive and practical while maintaining efficiency. This thesis is about optimally satisfying the combination of these goals.

1.2 DynamoRIO

We present *DynamoRIO*, a fully-implemented runtime code manipulation system that allows code transformations on any part of a program, *while it executes*. DynamoRIO extends existing code caching technology to allow efficient, transparent, and comprehensive manipulation of an individual, unmodified application, running on a stock operating system and commodity hardware.

Figure 1.3 illustrates the high-level design of DynamoRIO. DynamoRIO executes a target application by copying the application code into a code cache, one basic block at a time. The code cache is entered via a context switch from DynamoRIO's dispatch state to that of the application.

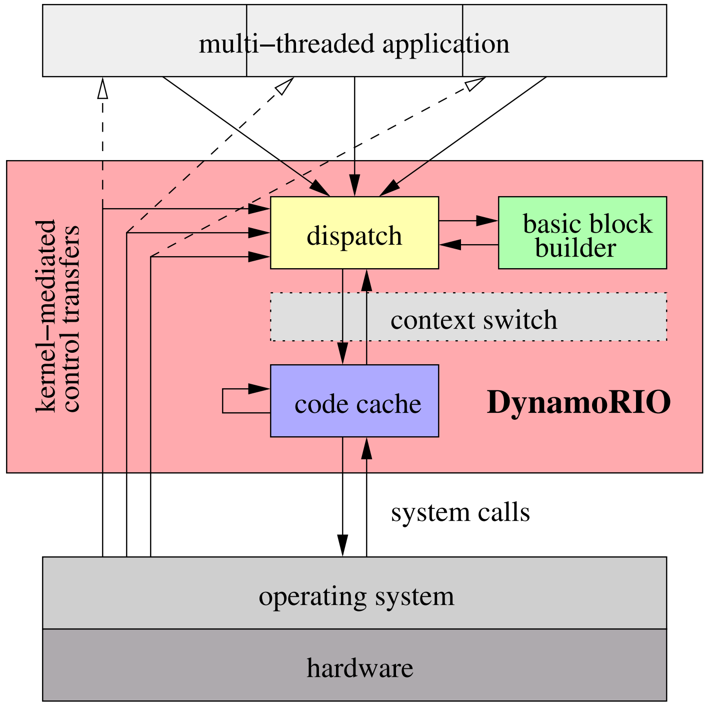

Figure 1.3: The DynamoRIO runtime code manipulation layer. DynamoRIO interposes itself between an application and the underlying operating system and hardware. It executes a copy of the application's code out of a *code cache* to avoid emulation overhead. Key challenges include managing multiple threads, intercepting direct transfers of control from the kernel, monitoring code modification to maintain cache consistency, and bounding the size of the code cache.

The cached code can then be executed natively, avoiding emulation overhead. However, shifting execution into a cache that occupies the application's own address space complicates transparency. One of our most significant lessons is that DynamoRIO cannot run large, complex, modern applications unless it is fully transparent: it must take every precaution to avoid affecting the behavior of the program it is executing.

To reach the widest possible set of applications (to be universal and practical), DynamoRIO targets the most common architecture, IA-32 (a.k.a. x86), and the most popular operating systems on that architecture, Windows and Linux. The efficiency of a runtime code manipulation system depends on the characteristics of the underlying hardware, and the Complex Instruction Set Computer (CISC) design of IA-32 requires a significant effort to achieve efficiency. To be universal, DynamoRIO must handle dynamically-loaded, generated, and even modified code. Unfortunately, since any store to memory could legitimately modify code on IA-32, maintaining cache consistency is challenging. Every write to application code must be detected, and system calls that load or unload shared libraries must be monitored. Further challenges arise because DynamoRIO resides on top of the operating system: multiple threads complicate its cache management and other operations, and comprehensiveness requires intercepting kernel-mediated control transfers (e.g., signal or callback delivery) and related system calls. Finally, DynamoRIO must dynamically bound its code cache size to be deployable on production systems without disturbing other programs on the same machine by exhausting memory resources.

DynamoRIO has met all of these challenges, and is capable of executing multi-threaded commercial desktop and server applications with minimal overhead that averages from zero to thirty percent. When aggressive optimizations are performed DynamoRIO is capable of surpassing native performance on some benchmarks by as much as forty percent. DynamoRIO is available to the public in binary form [MIT and Hewlett-Packard 2002] and has been used by many researchers for customized runtime applications via its interface, which supports the development of a wide range of custom runtime tools. Furthermore, DynamoRIO is being commercialized in a security product.

1.3 Contributions

Runtime code manipulation and code caching are mature fields of research. Many systems with different goals and designs have utilized these technologies, including emulators, simulators, virtual machines, dynamic optimizers, and dynamic translators. Chapter 10 compares and contrasts the differences in the goals and technologies of these systems with DynamoRIO. We extend runtime interposition technology in a number of different directions, the combination of which is required to comprehensively execute inside the process of a modern application:

- **Transparency** (Chapter 3)

  We show how to achieve transparency when executing from a code cache inside of the application's own process, and we classify the types of transparency that are required.

- **Architectural challenges** (Chapter 4)

  We contribute several novel schemes for coping with the CISC IA-32 architecture: an adaptive level-of-detail instruction representation to reduce decoding and encoding costs, efficient condition code preservation, and reduction of indirect branch performance bottlenecks.

- **Operating system challenges** (Chapter 5)

  We show how to handle thread complications in everything from application synchronization to cache management to obtaining scratch space. Another contribution is handling kernel transfers whose suspended context is kept in kernel mode, invisible to a user-mode runtime system, and causing havoc on cache management and continuation. These problematic transfers are ubiquitous in Windows applications. We also give a systematic treatment of state-handling options across kernel transfers, show how to operate at the system-call level on Windows, and enumerate the system calls that must be monitored to retain control.

- **Code cache management** (Chapter 6)

  We present a novel algorithm for efficient cache consistency in the face of multiple threads and self-modifying code, and extend the prior art with an incremental, runtime algorithm for adapting the cache size to match the application's working set size.

- **Validation and evaluation on real-world programs** (Chapter 7)

  We show that it is possible to build a runtime interposition point in software that can achieve zero to thirty percent overhead while executing large, complex, real-world programs with dynamic behavior and multiple threads.

- **Runtime client interface** (Chapter 8)

  We present our interface for building custom runtime code manipulation tools, which abstracts away the details of the underlying system and allows a tool designer to focus on manipulating the application's runtime code stream. Our interface provides support to the tool builder for maintaining transparency and allows efficient self-replacement of code in our code cache, facilitating adaptive tools.

Case studies of several applications of DynamoRIO are presented in Chapter 9. Related work is described in Chapter 10, and conclusions and future work are discussed in Chapter 11. To provide background for the subsequent chapters, the next chapter (Chapter 2) describes how DynamoRIO incorporates the standard code caching techniques of linking and trace building, including novel twists on trace starting conditions and basic block building across unconditional control transfers.

Chapter 2

Code Cache

DynamoRIO is able to observe and manipulate every application instruction prior to its execution by building upon known techniques of code caching, linking, and trace building. This chapter describes our implementation of these techniques, but delays discussing a number of important and novel aspects of DynamoRIO to subsequent chapters: transparency (Chapter 3), architectural challenges such as instruction representation and branch prediction problems (Chapter 4), challenges of interacting with the operating system (Chapter 5), and code cache management and consistency (Chapter 6).

Figure 2.1 shows the components of DynamoRIO and the flow of operation between them. The figure concentrates on the flow of control in and out of the *code cache*, which is the bottom portion of the figure. The cached application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure, and which must be modified to ensure that DynamoRIO retains control. This chapter describes each component in the figure: how we populate our code cache one *basic block* at a time (Section 2.1) and then *link* the blocks together (Section 2.2). The code cache enables native execution to replace emulation, bringing performance down from a several hundred times slowdown for pure emulation to an order of magnitude (Table 2.2). Linking of direct branches reduces slowdown further, to around three times native performance. Adding in indirect branch linking, by using a fast lookup of the variable indirect branch target, pushes that performance further, down under two times. Our novel twist on linking is to separate the stubs of code required for the unlinked case from the code for the block itself. We achieve further performance gains by building *traces* (Section 2.3) in a slightly different

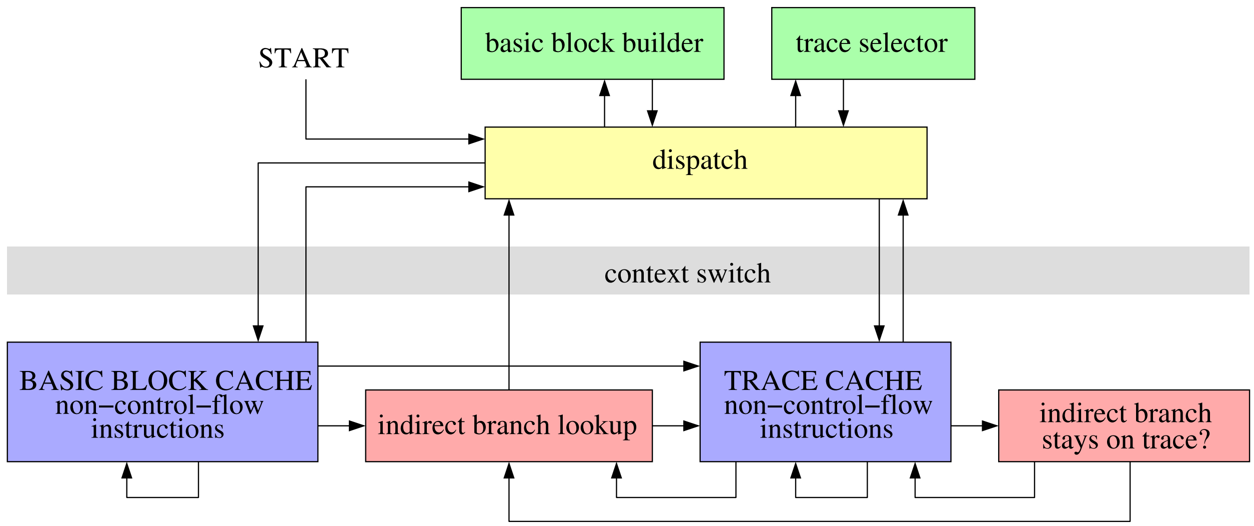

Figure 2.1: Flow chart of DynamoRIO. A context switch separates the code cache from DynamoRIO code (though it all executes in the same process and address space). Application code is copied into the two caches, with control transfers (shown by arrows in the figure) modified in order to retain control.

	Average slowdown	
System Components	SPECFP	SPECINT
Emulation	~300x	~300x
Basic block cache	3.54x	17.16x
+ Link direct branches	1.32x	3.04x
+ Link indirect branches	1.05x	1.44x
+ Traces	1.02x	1.17x
+ Optimizations	0.88x	1.13x

Table 2.2: Performance summary of the fundamental components of DynamoRIO described in this chapter: a basic block cache, linking of direct and indirect branches, and building traces. Average numbers for both the floating-point (SPECFP) and integer (SPECINT) benchmarks from the SPEC CPU2000 suite are given (our benchmarks are described in Section 7.1). We overcame numerous architectural challenges (Chapter 4) to bring each component to the performance level listed here. The final entry in the table shows the best performance we have achieved with DynamoRIO, using aggressive optimizations to surpass native performance for some benchmarks (see Section 9.2).

manner from other systems, and by our novel scheme of *eliding unconditional control transfers* when building basic blocks (Section 2.4).

```
original: add %eax, %ecx
          cmp $4, %eax
          jle 0x40106f
```

Figure 2.3: An example basic block consisting of three IA-32 instructions: an add, a compare, and a conditional direct branch.

2.1 Basic Blocks

DynamoRIO copies application code into its code cache in units of *basic blocks*, sequences of instructions ending with a single control transfer instruction. Figure 2.3 shows an example basic block from an application. DynamoRIO's basic blocks are different from the traditional static analysis notion of basic blocks. DynamoRIO considers each entry point to begin a new basic block, and follows it until a control transfer is reached, even if it duplicates the tail of an existing basic block. This is for simplicity of code discovery. Unlike static analyzers, DynamoRIO does not have the luxury of examining an entire code unit such as a procedure. At runtime such information may not be available, nor is there time to spend analyzing it.

The application's code is executed by transferring control to corresponding basic blocks in the code cache. At the end of each block, the application's machine state is saved and control returned to DynamoRIO (a *context switch*) to copy the next basic block. Figure 2.4 shows what the example block looks like inside of DynamoRIO's code cache. Before the targets of its exits have materialized in the cache, they point to two *exit stubs*. Each stub records a pointer to a stub-specific data structure so DynamoRIO can determine which exit was taken. At first glance, putting the second stub first seems like an optimization to remove the jump targeting it, but as Section 2.2 will show, we use that jump for linking, and it is not worth optimizing for the rare unlinked case.

Table 2.5 shows statistics on the sizes of basic blocks in our benchmark suite. A typical basic block consists of six or seven instructions taking up twenty or thirty bytes, although some blocks can be quite large, in the thousands of bytes.

Figure 2.6 shows the performance of a basic block cache system. Pure emulation slows down execution by about 300 times compared to native; directly executing the non-control flow instructions in a basic block cache, and only emulating the branches, brings that slowdown down to about six times on average. Each successive addition of linking and trace building brings that perfor-

```
fragment7: add %eax, %ecx
           cmp $4, %eax
           jle stub0
           jmp stub1
    stub0: mov %eax, eax-slot
           mov &dstub0, %eax
           jmp context_switch
    stub1: mov %eax, eax-slot
           mov &dstub1, %eax
           jmp context_switch
```

Figure 2.4: The example basic block from Figure 2.3 copied into DynamoRIO's code cache. Each exit stub records a pointer to its own data structure (*dstub0* or *dstub1*) before transferring control to the context switch, so that DynamoRIO can figure out which branch was taken. The pointer is stored in a register that first needs to be spilled because this two-instruction combination is more efficient than a ten-byte (slowly-decoded) store of the pointer directly to memory.

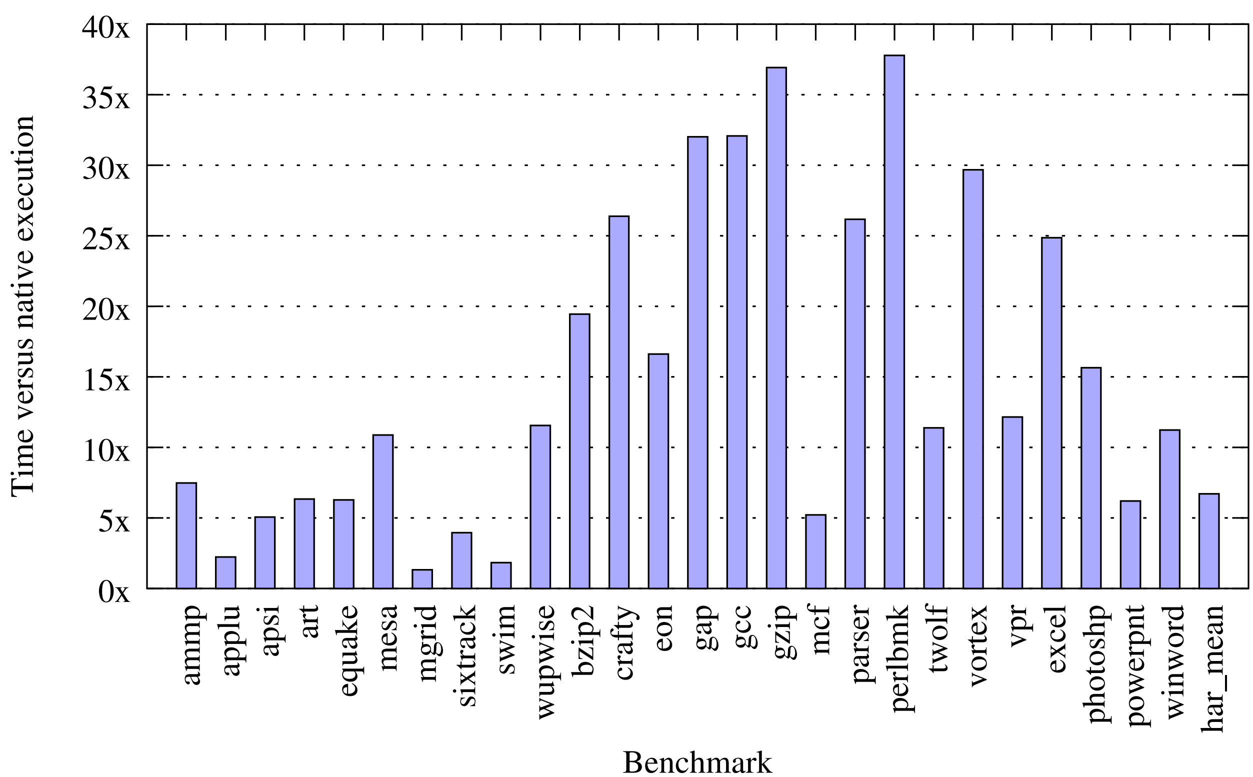

Figure 2.6: Performance of a basic block cache system versus native execution. This graph shows time, so smaller numbers are better.

Benchmark	# blocks	Max bytes	Ave bytes	Max instrs	Ave instrs
ammp	2351	2293	22.59	520	6.70
applu	2687	33360	72.99	7570	16.84
apsi	4470	3763	34.88	771	9.73
art	1395	211	17.47	57	5.47
equake	1940	372	21.22	118	6.04
mesa	2884	1743	22.52	364	6.31
mgrid	2321	3975	25.79	812	7.05
sixtrack	9270	3039	29.62	908	7.58
swim	2342	1332	18.32	310	5.50
wupwise	2665	4805	22.30	1023	6.71
bzip2	1693	193	19.20	35	5.61
crafty	6306	834	23.60	163	6.55
eon	6002	1247	40.33	206	8.65
gap	8645	1002	16.03	103	5.19
gcc	36494	748	13.97	102	4.51
gzip	1600	193	17.33	29	5.20
mcf	1661	313	15.84	87	5.01
parser	6538	194	14.18	56	4.73
perlbmk	14695	1673	15.07	583	4.80
twolf	5781	280	19.31	68	5.86
vortex	12461	532	17.85	81	5.96
vpr	3799	298	17.52	68	5.52
excel	92043	1129	13.04	458	4.39
photoshp	206094	4023	16.35	834	5.13
powerpnt	153984	1206	12.61	458	4.34
winword	111570	2794	13.42	1009	4.52
average	26988	2752	22.05	646	6.30

Table 2.5: Sizes of basic blocks measured in both bytes and instructions (since IA-32 instructions are variable-sized).

mance down still further (Table 2.2 summarizes the numbers).

```
fragment7: add %eax, %ecx
           cmp $4, %eax
           jle fragment42
           jmp fragment8
    stub0: mov %eax, eax-slot
           mov &dstub0, %eax
           jmp context_switch
    stub1: mov %eax, eax-slot
           mov &dstub1, %eax
           jmp context_switch
```

Figure 2.7: The example basic block from Figure 2.4 with both the taken branch and the fall-through linked to other fragments in the code cache.

2.2 Linking

Copying each basic block into a code cache and executing it natively reduces the performance hit of interpretation enormously. However, we are still interpreting each control transfer by going back to DynamoRIO to find the target. If the target is already present in the code cache, and is targeted via a direct branch, DynamoRIO can *link* the two blocks together with a direct jump, avoiding the cost of a subsequent context switch. Figure 2.7 shows how the exit stubs of our example block are bypassed completely after linking. The performance improvement of linking direct control transfers is dramatic (Figure 2.8), as expensive context switches are replaced with single jumps.

Linking may be done either *proactively*, when a fragment is created, or *lazily*, when an exit is taken. Section 4.5.1 explains why proactive linking is a better choice for IA-32. In either case, data structures must be kept to record the outgoing links of each fragment. The incoming links must also be kept, in order to efficiently delete a single fragment: otherwise, all other fragments must be searched to make sure all links to the dead fragment are removed, or alternatively space must be wasted with a placeholder in the dead fragment's place. Single-fragment deletion is essential for cache consistency (see Section 6.2). Incoming link records are also required to quickly shift links from one fragment to another for things like trace head status changes (Section 2.3.2) or replacing a fragment with a new version of itself (Section 8.2.3). Incoming links to non-existent fragments must be stored as well, for which we use a *future fragment* data structure as a placeholder. Once an actual fragment at that target is built, it replaces the future fragment and takes over its incoming link list. Future fragments can also be used to keep persistent state across fragment deletions and

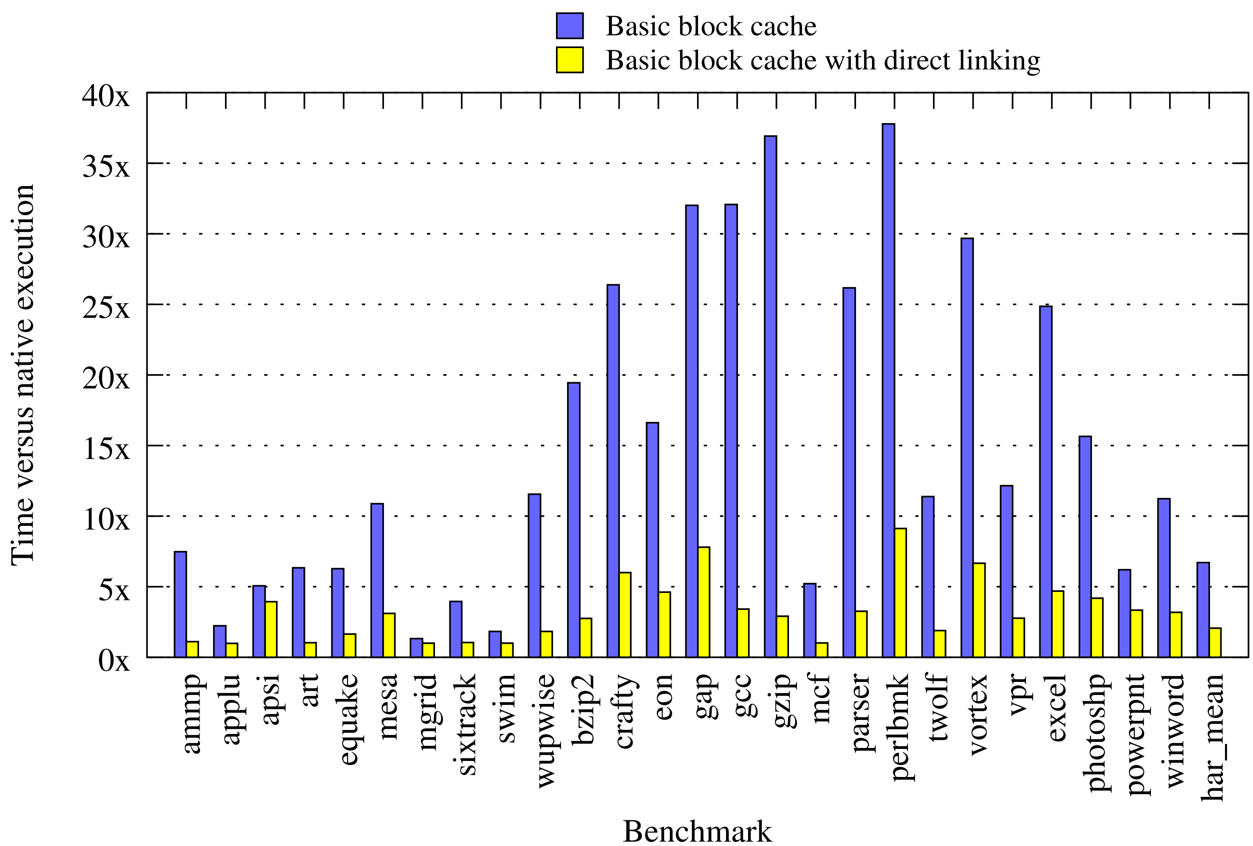

Figure 2.8: Performance impact of linking direct control transfers, compared to the performance of a basic block cache with no linking, versus native execution time.

re-creations, such as for cache capacity (Section 6.3.3) and trace head counters (Section 2.3.2).

We must be able to undo linking on demand, for building traces (Section 2.3), bounding time delay of delivering signals (Section 5.3.5), fragment replacement (Section 8.2.3), and when deleting a fragment. Unlinking requires either incoming link information or using a prefix on each fragment. DynamoRIO uses incoming link information, as it is already needed for proactive linking and other features.

The actual process of linking and unlinking boils down to modifying the exits of a fragment. Examining Figure 2.7 and its unlinked version Figure 2.4 shows that each branch exiting a fragment either points to its corresponding exit stub (the *unlinked* state) or points to its actual fragment target (the *linked* state). Switching from one state to the other takes a single 32-bit store, which, if the targets do not straddle cache lines or if the `lock` prefix is used, is atomic on all recent IA-32 processors [Intel Corporation 2001, vol. 3] and thus can be performed in the presence of multiple threads without synchronization.

Fortunately, on IA-32 we do not have reachability problems that systems on other architectures faced [Bala et al. 1999]. The variable-length instruction set allows for full 32-bit addresses as immediate operands, allowing a single branch to target any location in memory. A few specific branch types take only eight-bit immediates, but we are able to transform these to equivalent 32-bit-immediate branches (see Section 4.1.3).

Once an exit from a basic block is linked, the corresponding exit stub is not needed again unless the exit is later unlinked. By locating the exit stubs in a separate cache from the basic block body, we can delete and re-create exit stubs on demand as they are needed. This both compacts the cache, reducing the working set size of the program, and reduces overall memory usage by deleting stubs no longer needed. The performance impact of separating direct exit stubs is shown in Figure 2.9. The resulting reduced instruction cache pressure helps benchmarks with larger code sizes, such as `photoshp` and `gcc` in our suite. Memory savings are given in Section 6.3.5. About one-half of all stubs are not needed at any given time (when not using them for profiling as in Section 7.3.3). The other half are mainly exits whose targets have not yet been reached during execution (and may never be reached).

Indirect branches cannot be linked in the same way as direct branches because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls — see Section 3.3.3). These addresses must be translated to their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup inside the code cache (avoiding a context switch back to DynamoRIO). Figure 2.10 shows the performance improvement of linking indirect control transfers. Benchmarks with more indirect branches, such as `perlbmk` and `gap`, are more affected by optimizing indirect branch performance than applications with few indirect branches, like `swim` (see Table 7.4 for the indirect branch statistics of our benchmark suite).

The translation of indirect branches is the single largest source of overhead in DynamoRIO. Why this is so, and our attempts to reduce the cost by both optimizing our hashtable lookup and eliminating the translation altogether, are discussed in Section 4.2 and Section 4.3.

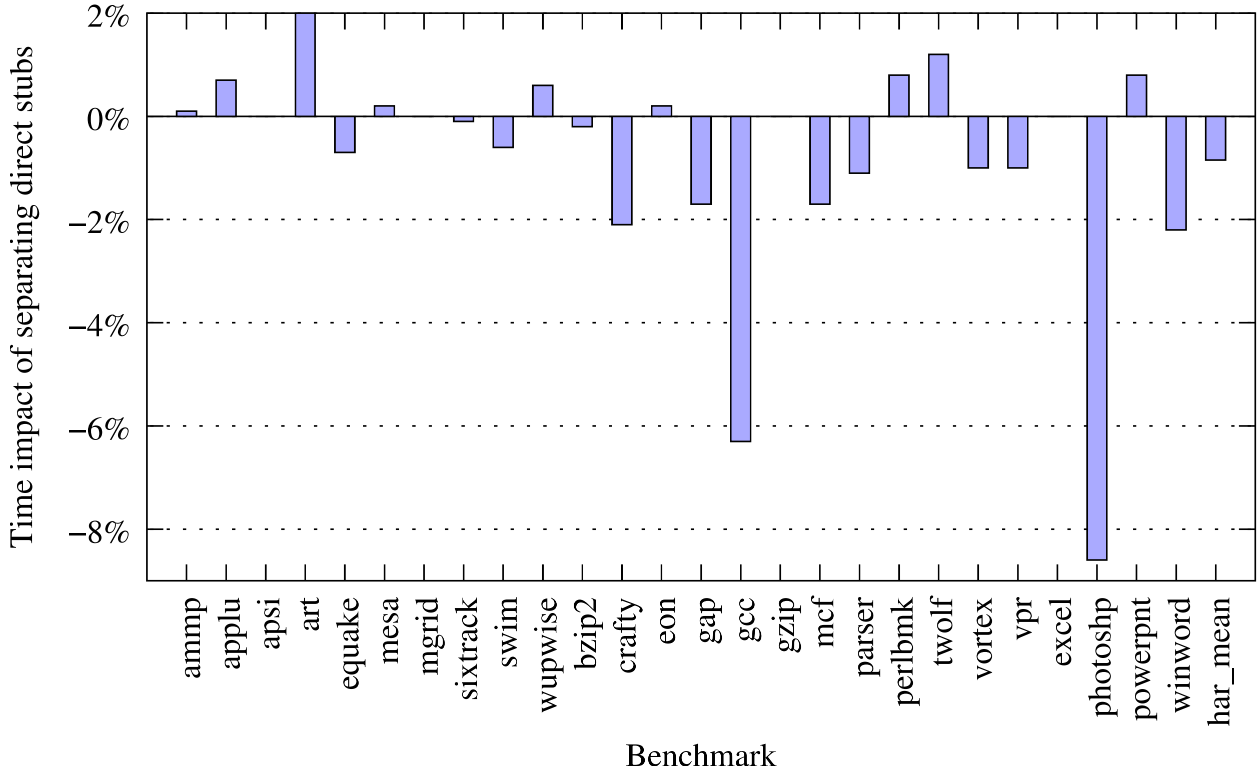

Figure 2.9: Performance impact of separating direct exit stubs. Relative time impact is shown compared to base DynamoRIO performance, so smaller numbers are better. As in all of our performance measurements, noise produces an impact of up to one or even two percent (see Section 7.1.1).

2.3 Traces

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. The superior code layout and inter-block branch elimination in traces provide a significant performance boost, as shown in Figure 2.11. Benchmarks whose hot loops consist of single basic blocks, such as `mgrid` and `swim`, are not improved by traces; fortunately, such benchmarks already perform well under DynamoRIO. One of the biggest benefits of traces is in avoiding indirect branch lookups by inlining a popular target of an indirect branch into a trace (with a check to ensure that the actual target stays on the trace, falling back on the full lookup when the check fails). This explains why their biggest impact is often on benchmarks with many indirect branches.

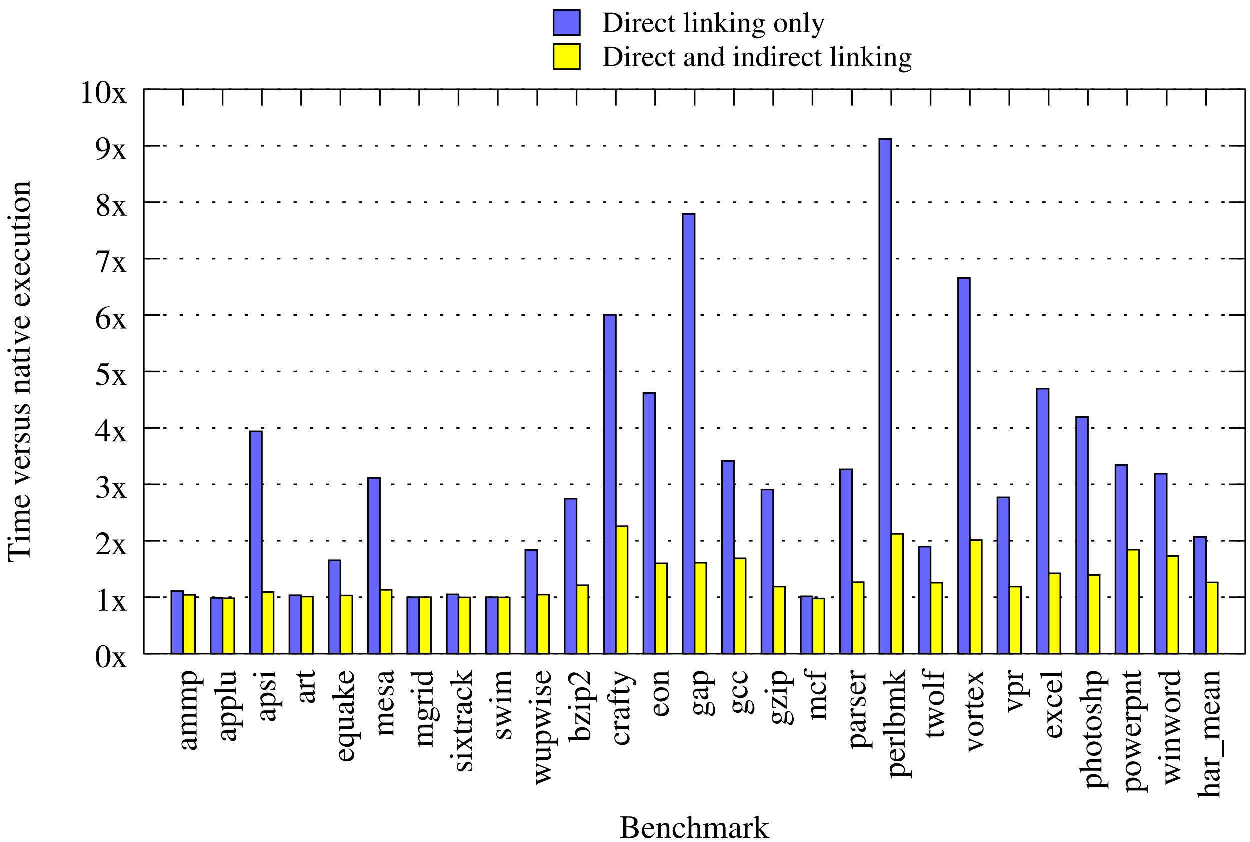

Figure 2.10: Performance impact of linking indirect control transfers, compared to only linking direct control transfers, versus native execution time.

Trace building is also used as a hardware instruction fetch optimization [Rotenberg et al. 1996], and the Pentium 4 contains a hardware trace cache. Although the Pentium 4 hardware trace cache stitches together IA-32 micro-operations, it is targeting branch removal just like a software trace cache, and there is some competition between the two. The hardware cache has a smaller window of operation, but its effects are noticeable. In Figure 2.11 the average overall speedup is 11% on the Pentium 3 as opposed to just over 7% for the Pentium 4. The differences for individual benchmarks are sometimes reversed (e.g., `powerpnt` and `winword`) for reasons we have not tracked down, perhaps due to other differences in the underlying machines.

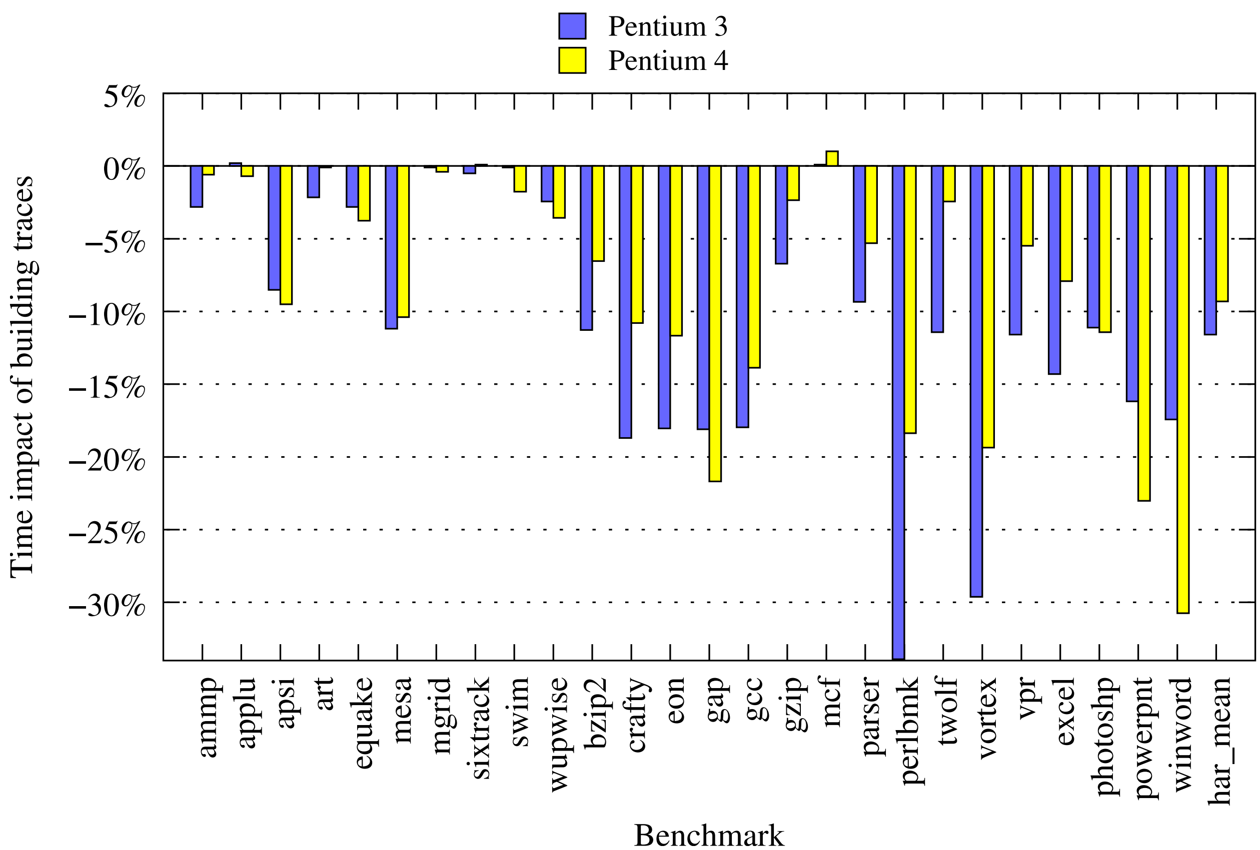

Figure 2.11: Performance impact of traces on both a Pentium 3 and a Pentium 4, versus DynamoRIO performance without traces.

2.3.1 Trace Shape

DynamoRIO's traces are based on the Next Executing Tail (NET) scheme [Duesterwald and Bala 2000]. Figure 2.12 shows two example traces created from sequences of basic blocks. As Duesterwald and Bala [2000] show, a runtime system has very different profiling needs than a static system. For static or offline processing, path profiling [Ball and Larus 1996] works well. However, its overheads are too high to be used online, especially in terms of missed opportunities while determining hot paths. Another problem with many path profiling algorithms is a preparatory static analysis phase that requires access to complete source code. These algorithms can only be used in a runtime system by coordinating with a compiler [Feigin 1999]. General runtime profiling must be done incrementally, as code is discovered — all the code to be profiled is not known beforehand. Some path profiling algorithms can operate online, such as bit tracing [Duesterwald and Bala 2000], but

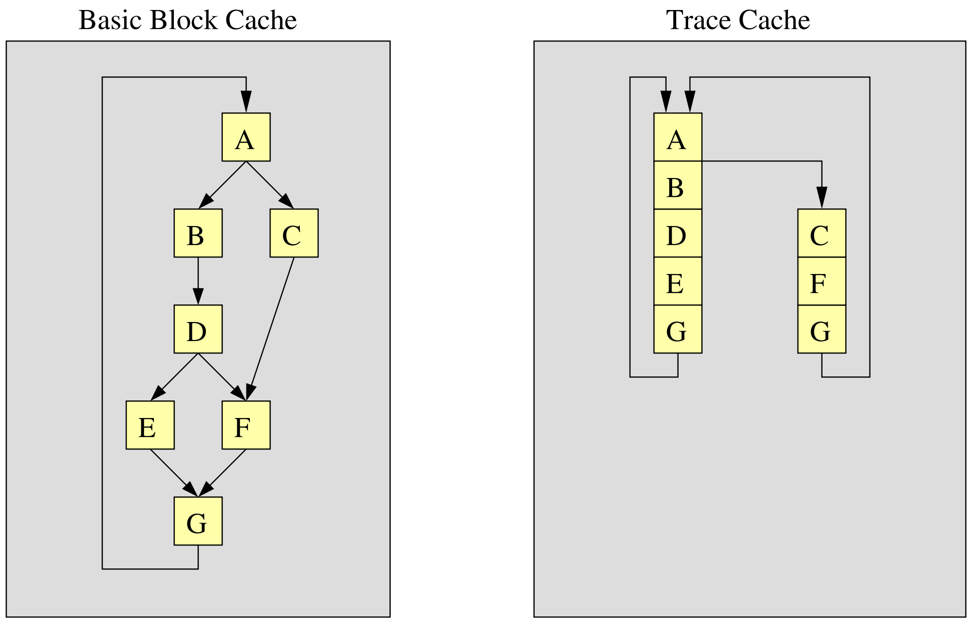

Figure 2.12: Building traces from basic blocks. Block A, as a target of a backward branch, is a *trace head* with an associated execution counter. Once its counter exceeds a threshold, the *next executing tail* is used to build the trace headed by A. In this example, the tail is BDEG. Block C, as an exit from a (newly created) trace, becomes a *secondary trace head*. If it becomes hot, the secondary trace shown will be created.

none identify hot paths quickly enough.

The NET trace creation scheme is specifically designed for low-overhead, incremental use. Despite its simplicity, it has been shown to identify traces with comparable quality to more sophisticated schemes [Duesterwald and Bala 2000]. NET operates by associating a counter with each *trace head*. A trace head is either the target of a backward branch (targeting loops) or an exit from an existing trace (called a *secondary trace head*). The counter is incremented on each execution of the trace head. Once the counter exceeds a threshold (usually a small number such as fifty), trace creation mode is entered. The next executing tail (NET) is taken to be the hot path. This means that the next sequence of basic blocks that is executed after the trace head becomes hot is concatenated together to become a new trace. The trace is terminated when it reaches a backward branch or another trace or trace head.

DynamoRIO modifies NET to not consider a backward *indirect* branch target to be a trace head. Consequently, where NET would stop trace creation at a backward indirect branch, we continue. This has both an advantage and a disadvantage. The advantage is that more indirect branches

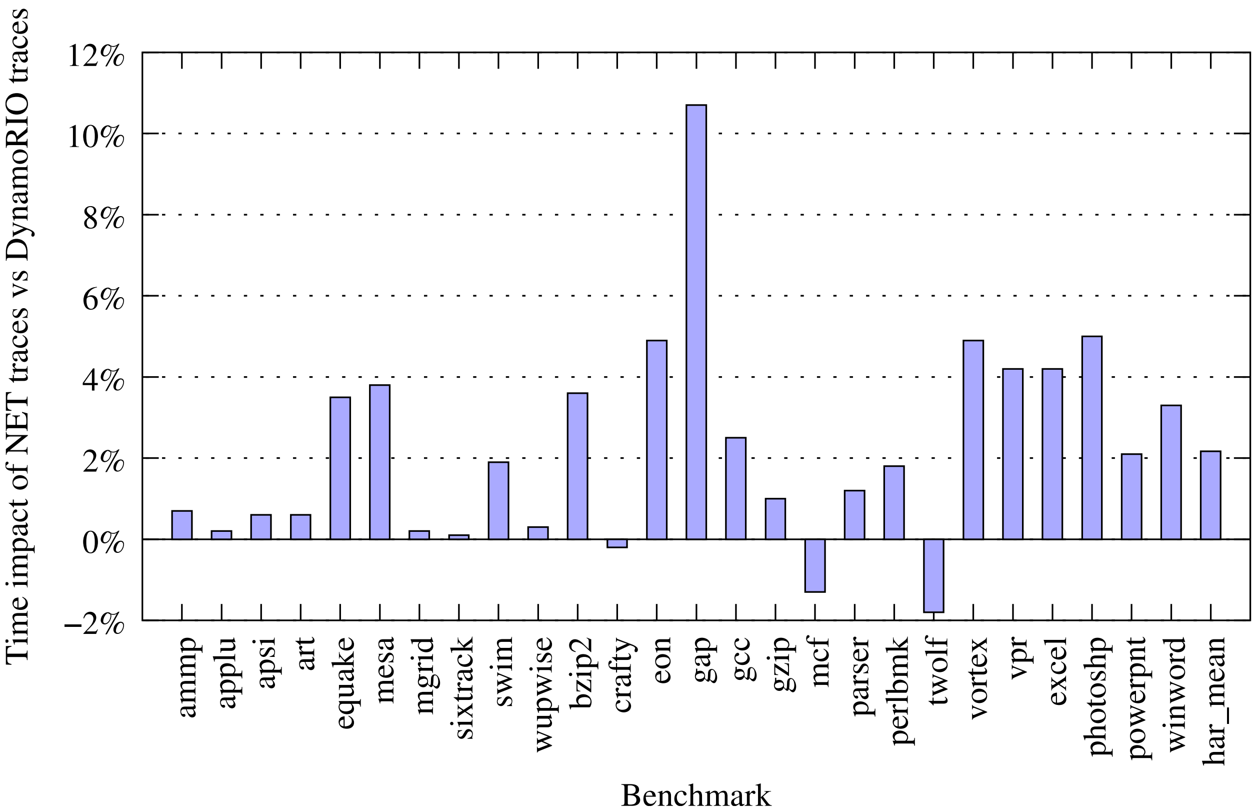

Figure 2.13: Performance impact of using the NET trace building scheme versus DynamoRIO's trace building scheme. NET treats indirect and direct branches the same for trace head purposes, while DynamoRIO does *not* treat a backward indirect branch target as a trace head. NET traces perform worse than DynamoRIO's traces on nearly all of our benchmarks.

will be inlined into traces, where with the NET scheme, half of the time a trace will stop at an indirect branch. The disadvantage is that in pathological situations (e.g., a recursive loop where the recursive call is indirect) unlimited loop unrolling can occur. We feel that the advantage is worth the extra unrolling, and use a maximum trace size to limit code bloat. Figure 2.13, showing performance, and Table 2.14, showing size, back up our choice: the average size increase is under eight percent, while the performance improvement is as much as ten percent. We have not tracked down the exact indirect branches in `gap` and the other benchmarks that are responsible for the difference in trace performance.

The key insight is that more trace heads do not result in better traces. Since trace creation stops upon reaching a trace head (to avoid code duplication), more trace heads can result in many tiny

Benchmark	Trace cache
ammp	7.2%
applu	2.1%
apsi	5.8%
art	7.1%
equake	11.8%
mesa	5.9%
mgrid	8.0%
sixtrack	7.1%
swim	7.1%
wupwise	1.0%
bzip2	0.0%
crafty	5.4%
eon	17.3%
gap	8.2%
gcc	6.1%
gzip	4.6%
mcf	14.3%
parser	3.4%
perlbmk	4.5%
twolf	22.9%
vortex	4.0%
vpr	14.3%
excel	9.5%
photoshp	8.0%
powerpnt	13.4%
winword	5.8%
average	7.9%

Table 2.14: Trace cache size increase from DynamoRIO's changes to the NET trace building scheme. The average size increase is under eight percent, which is a reasonable cost for achieving performance improvements as high as ten percent (Figure 2.13).

traces. By selectively eliminating trace heads that are targets of indirect branches, we try to build traces across those branches.

However, DynamoRIO's trace building scheme does do poorly in some extreme cases. An example is a threaded interpreter, such as Objective Caml [Leroy 2003], where indirect branches are used almost exclusively, causing DynamoRIO to build no traces. This is not a catastrophic situation; we will simply not get the performance boost of traces.

To understand the shape of our traces, see Table 2.15. An average trace consists of four basic blocks, about 29 instructions. More than one in two traces contains an inlined indirect branch, one of the goals of trace building. Traces reduce DynamoRIO's indirect branch translation overhead significantly.

Table 2.16 shows the coverage and completion rates of our traces. We gathered these using our *exit counter profiling*, which is discussed in Section 7.3.3. On average, only five traces are needed to cover a full one-half of a benchmark's execution time. Ten traces cover nearly two-thirds, and fifty approaches seven-eighths. For completion, on average a trace is only executed all the way to the end one-third of the time. However, execution reaches at least the half-way point in a trace 90% of the time.

2.3.2 Trace Implementation

To increment the counter associated with each trace head, the simplest solution is to never link any fragment to a trace head, and perform the increment inside DynamoRIO (the first method in Figure 2.17). As there will never be more than a small number of increments before the head is turned into a trace, this is not much of a performance hit. We tried two different strategies for incrementing without the context switch back to DynamoRIO. One strategy is to place the increment inside the trace head fragment itself (the second method in Figure 2.17). However, this requires replacing the old fragment code once the fragment is discovered to be a trace head (which often happens after the fragment is already in the code cache, when a later backward branch is found to target it). The cost of replacing the fragment overwhelms the performance improvement from having the increment inlined (remember, the increment only occurs a small number of times — DynamoRIO's default is fifty).

A different strategy is to use a shared routine inside the cache to perform the increment (the

Benchmark	Basic blocks Max	Basic blocks Ave	Instructions Max	Instructions Ave	Bytes Max	Bytes Ave	Inlined ind. br. Max	Inlined ind. br. Ave
ammp	50	4.3	531	32	2355	108	12	0.5
applu	40	3.3	4096	63	18068	278	13	0.3
apsi	41	4.6	774	44	3778	152	13	0.6
art	34	3.6	200	21	924	69	12	0.4
equake	38	4.4	198	25	824	92	14	0.6
mesa	45	5.4	669	39	2270	136	16	0.8
mgrid	40	4.3	815	37	3984	137	13	0.7
sixtrack	53	4.3	926	31	3113	111	17	0.5
swim	40	4.3	313	27	1341	92	13	0.7
wupwise	36	5.4	1034	45	4827	146	12	0.6
bzip2	54	3.0	303	20	936	71	5	0.2
crafty	69	3.4	530	23	2006	83	14	0.3
eon	49	5.5	570	40	2271	130	13	1.1
gap	66	3.7	292	19	752	53	22	0.6
gcc	48	3.7	190	18	750	55	13	0.3
gzip	26	3.2	149	18	612	63	9	0.3
mcf	35	4.7	215	23	732	73	11	0.7
parser	48	2.9	210	15	601	45	15	0.2
perlbmk	57	4.0	604	21	1712	63	13	0.4
twolf	52	5.0	270	27	853	93	14	0.7
vortex	82	5.8	402	47	1244	137	16	0.5
vpr	38	4.6	211	26	699	78	12	0.6
excel	121	4.6	438	22	1262	67	23	0.8
photoshp	62	4.1	394	37	1210	109	24	0.8
powerpnt	21	4.3	95	20	303	59	6	0.8
winword	321	4.4	1994	22	5880	66	99	0.7
average	60	4.3	632	29	2435	98	17	0.6

Table 2.15: Trace shape statistics. The numbers for each benchmark are an average over all of that benchmark's traces. (See Table 7.3 for trace counts for each benchmark.) The maximum and the arithmetic mean are shown for each of four categories: number of basic blocks composing each trace, number of application instructions in each trace, number of bytes in those instructions (i.e., the sizes given are for the original application basic blocks that are stitched together, not the resulting trace size in the code cache, which would include exit stubs, prefixes, and indirect branch comparison code), and number of indirect branches inlined into each trace.

Benchmark	Coverage Top 5	Coverage Top 10	Coverage Top 50	Completion End	Completion Half
ammp	80.4%	92.6%	99.5%	23.7%	93.6%
applu	58.3%	76.6%	99.4%	17.7%	96.8%
apsi	46.1%	63.2%	94.3%	33.1%	93.7%
art	71.6%	84.5%	100.0%	24.6%	90.8%
equake	69.8%	85.4%	99.3%	25.8%	91.3%
mesa	44.9%	65.3%	98.9%	34.6%	92.5%
mgrid	94.9%	98.9%	100.0%	29.4%	93.9%
sixtrack	82.7%	98.7%	99.9%	41.4%	94.0%
swim	99.9%	99.9%	100.0%	36.4%	95.9%
wupwise	72.2%	83.6%	94.7%	24.6%	92.4%
bzip2	36.7%	50.6%	90.6%	25.6%	90.0%
crafty	15.9%	24.5%	53.5%	33.8%	85.4%
eon	22.8%	35.7%	76.2%	30.4%	87.5%
gap	31.3%	47.5%	81.5%	44.7%	89.5%
gcc	27.0%	33.2%	47.7%	41.9%	90.3%
gzip	36.2%	58.3%	97.5%	26.8%	88.5%
mcf	54.2%	78.9%	97.6%	31.9%	89.8%
parser	16.3%	24.0%	54.5%	36.4%	89.8%
perlbmk	65.9%	76.0%	91.6%	38.2%	89.0%
twolf	24.3%	40.7%	81.4%	27.1%	91.0%
vortex	42.6%	56.2%	81.9%	48.0%	92.5%
vpr	33.5%	53.9%	96.8%	34.9%	86.8%
excel	60.2%	86.5%	98.0%	55.3%	84.7%
photoshp	29.0%	36.9%	57.7%	23.0%	90.7%
powerpnt	66.9%	87.1%	99.8%	40.2%	88.2%
winword	17.8%	27.1%	49.9%	62.1%	86.5%
average	50.1%	64.1%	86.2%	34.3%	90.6%

Table 2.16: Trace coverage and completion statistics. The numbers for each benchmark are an average over all of that benchmark's traces. For coverage, the percentages of total trace execution time spent in the top five, ten, and fifty traces are shown in the first three columns, respectively. (Trace execution time is very close to total execution time for nearly all of our benchmarks, as shown in Table 7.8.) The fourth column shows how frequently execution makes it to the end of the trace (without exiting early). The final column shows the percentage of the time that execution makes it to the second half (defined in terms of exits) of the trace.

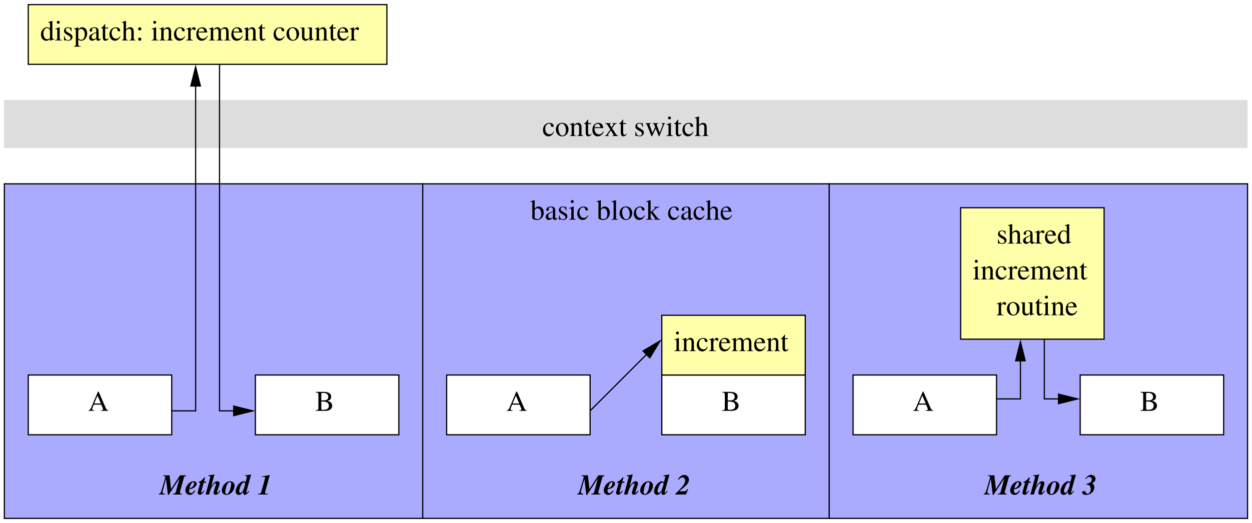

Figure 2.17: Three methods of incrementing trace head counters: exiting the cache to perform the increment in DynamoRIO code, re-writing the trace head to increment its counter inline, and using a shared increment routine inside the code cache.

third method in Figure 2.17). When discovering that a fragment is a trace head, all fragments pointing to it can be changed to instead link to the increment routine. This link change is most easily done when incoming links are recorded (see Section 2.2). The increment routine increments the counter for the target trace head and then performs an indirect branch to the trace head's code cache entry point. Since a register must be spilled to transfer information between the calling fragment and the increment routine, the routine needs to restore that register, while keeping the indirect branch target available. Only two options allow both: storing the indirect branch target in memory, or adding a prefix to all potential trace heads (all basic blocks, unless blocks are replaced once they are marked as trace heads, which as mentioned earlier is expensive) that will restore the register containing the target to its application value. We chose to store the target in memory, though this has ramifications for self-protection (see Section 9.4.5).

Incrementing the counter without leaving the code cache drastically reduces the number of exits from the cache (Table 2.18). Surprisingly, the performance difference (Figure 2.19) is no more than noise for nearly all of our benchmarks. The explanation is that code cache exits are not a major source of overhead because the number of them is already small. The benchmarks that it does make a difference on are those that execute large amounts of code with little re-use, our

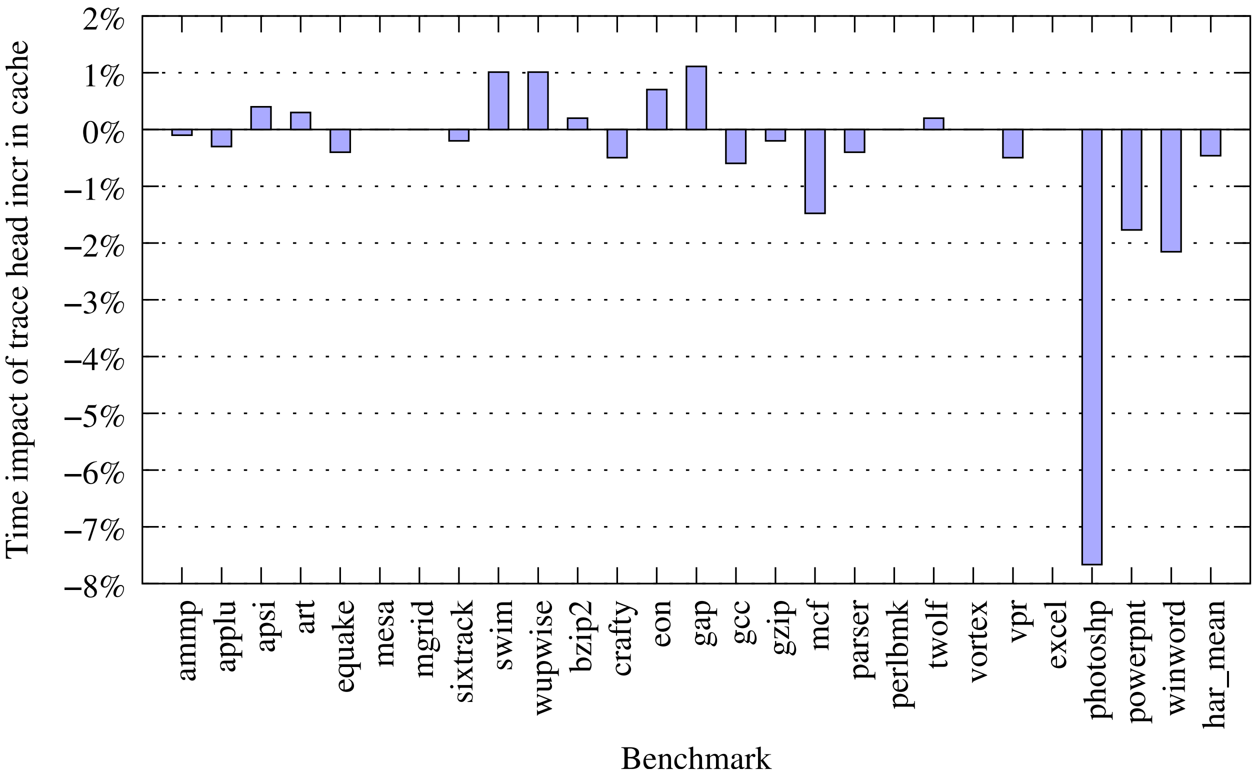

Figure 2.19: Performance impact of incrementing trace head counters inside the code cache, versus exiting the cache to perform increments inside DynamoRIO.

desktop benchmarks, and are spending noticeable time entering and exiting the cache.

Indirect branches targeting trace heads present some complications. For the first increment method of not linking to trace heads, the hashtable(s) used for indirect branches must not contain trace heads at all, to avoid directly targeting a trace head and skipping its counter increment. The most straightforward way is to use two separate hashtables, one for basic blocks and one for traces, with only the trace hashtable being consulted when resolving an indirect branch. However, this can result in terrible performance on programs with pathological trace building problems, such as the threaded interpreters mentioned above, since basic blocks will never be indirectly linked to other basic blocks. One solution is to use a different lookup routine for basic blocks that looks in both the basic block and trace hashtables, but that requires support for fragments to exist in multiple hashtables simultaneously. A simpler solution that preserves a one-hashtable-per-fragment invariant (which has advantages for a traditional chained hashtable, as explained in Section 4.3.3) is

Benchmark	Code cache exits		% Reduction	Instrs between exits
	Exit to incr	Incr in cache		
ammp	37491	18940	49%	10040410
applu	40721	16570	59%	98173295
apsi	91341	55624	39%	35475064
art	20803	8609	59%	5969804
equake	35974	17490	51%	4729822
mesa	33257	17830	46%	9500127
mgrid	36878	20641	44%	173472495
sixtrack	178403	118093	34%	11307727
swim	31539	18197	42%	41273912
wupwise	33436	21265	36%	26797111
bzip2	27197	7471	73%	3364326
crafty	151004	65280	57%	1415142
eon	101240	67845	33%	785384
gap	208695	122664	41%	1175068
gcc	995130	537771	46%	75366
gzip	26011	10167	61%	2887328
mcf	24198	12421	49%	2051516
parser	192279	79012	59%	1572238
perlbmk	315179	195251	38%	360501
twolf	159911	91408	43%	1852202
vortex	190478	137046	28%	650459
vpr	64522	34318	47%	1810481
excel	1919330	1735839	10%	N/A
photoshp	10804377	9986015	8%	N/A
powerpnt	19475416	19070649	2%	N/A
winword	1151742	866214	25%	N/A
average			42%	

Table 2.18: The number of code cache exits. The first column shows the number of exits when we must exit the cache to increment a trace head counter. Column two shows the number when we perform increments in the cache itself. The third column gives the resulting reduction in exits. To give an idea of how infrequent exits are, the final column divides the total instructions executed by the first column, resulting in an average number of instructions executed between code cache exits. This numbers in the millions for most benchmarks.

to have two disjoint hashtables: one that contains trace heads and one that contains all non-trace heads, both traces and basic blocks. For the second increment method, the indirect branch lookup routine must be modified to check whether its target is a trace head. If so, it should transfer control to the shared increment routine and pass it a pointer to the target fragment.

To avoid losing the trace head count due to eviction of the trace head from the cache for capacity reasons (see Section 6.3), it is best to use *persistent trace head counters*. When a trace head is deleted, its count can be stored in the *future fragment* data structure used to store incoming links for a deleted or not-yet-created fragment (see Section 2.2). Once the trace head is re-created, the existing count can be transferred so that it does not start at zero. Persistent trace head counters are important for maintaining trace building progress, and thus performance, when the basic block cache size is limited (see Section 6.3.2).

Once a trace head's counter exceeds the trace threshold, a new trace is built by executing basic blocks one at a time. Each block's outgoing exits are unlinked, so that after execution it will come back to DynamoRIO in order to have the subsequent block added to the trace. Each block is marked as un-deletable as well, to avoid a capacity miss that happens to evict this particular block from ruining the trace being built. After being copied into the trace-in-progress and being executed to find the subsequent basic block, the current block is re-linked and marked as deletable again. Then the next block is unlinked and the process repeats. Once the subsequent block is known, if the just-executed block ends in a conditional branch or indirect branch, that branch is inlined into the trace. For a conditional branch, the condition is reversed if necessary to have the fall-through branch direction keep control on the trace, as shown in Figure 2.20. The taken branch exits the trace. For an indirect branch, a check is inserted comparing the actual target of the branch with the target that will keep it on the trace. If the check fails, the trace is exited.

Once a trace is built, all basic blocks targeted by its outgoing exits automatically become secondary trace heads. This ensures that multiple hot tails of a trace head will all become traces. The trace head that caused trace creation is removed from the code cache, as its execution is replaced by the new trace.

The shape of basic blocks has a large impact on trace creation because it changes the trace heads. Section 2.4 discusses one variant on basic block shape and how it affects traces.

Traces and basic blocks are treated in the same manner once they are copied to the cache.

```
block1:
  0x08069905   cmp    (%eax), %edx
  0x08069907   jnb    $0x8069a02 <block3>
block2:
  0x0806990d   mov    $4, %esi
  ...
block3:
  0x08069a02   mov    0x810f46c, %edx
  ...

trace:
  0x4c3f584e   cmp    (%eax), %edx
  0x4c3f5850   jb     <fragment for block2>
  0x4c3f5856   mov    0x810f46c, %edx
```

Figure 2.20: An example of reversing the direction of a conditional branch in a trace. Three basic blocks from the application are shown. The first block ends in a conditional branch, whose fall-through target is the second block. During trace creation, the conditional branch is taken, and so the first and third blocks are placed in the trace. The direction of the conditional branch is reversed to make the fall-through target the third block and stay on the trace.

We use the term *fragment* to refer to either a basic block or a trace in the code cache. Both types of fragment are single-entry, multiple-exit, linear sequences of instructions. As discussed in Section 4.1.1, these features facilitate optimizations and other code transformations.

2.3.3 Alternative Trace Designs

We considered several alternative trace designs.

Supertraces

DynamoRIO profiles basic blocks to build hot sequences of blocks, or traces. We attempted to add another generation, profiling traces to build hot sequences of traces, or *supertraces*, by taking the results of exit counter profiling (Section 7.3.3) and connecting traces joined by direct exits into larger, self-contained units. Figure 2.21 shows some example supertraces determined for three benchmarks. For a benchmark like `mgrid` where traces are already capturing the hot code well, the supertraces are identical to the traces. The supertraces are more interesting for other benchmarks, where they combine several traces. However, for some benchmarks the supertraces we came up

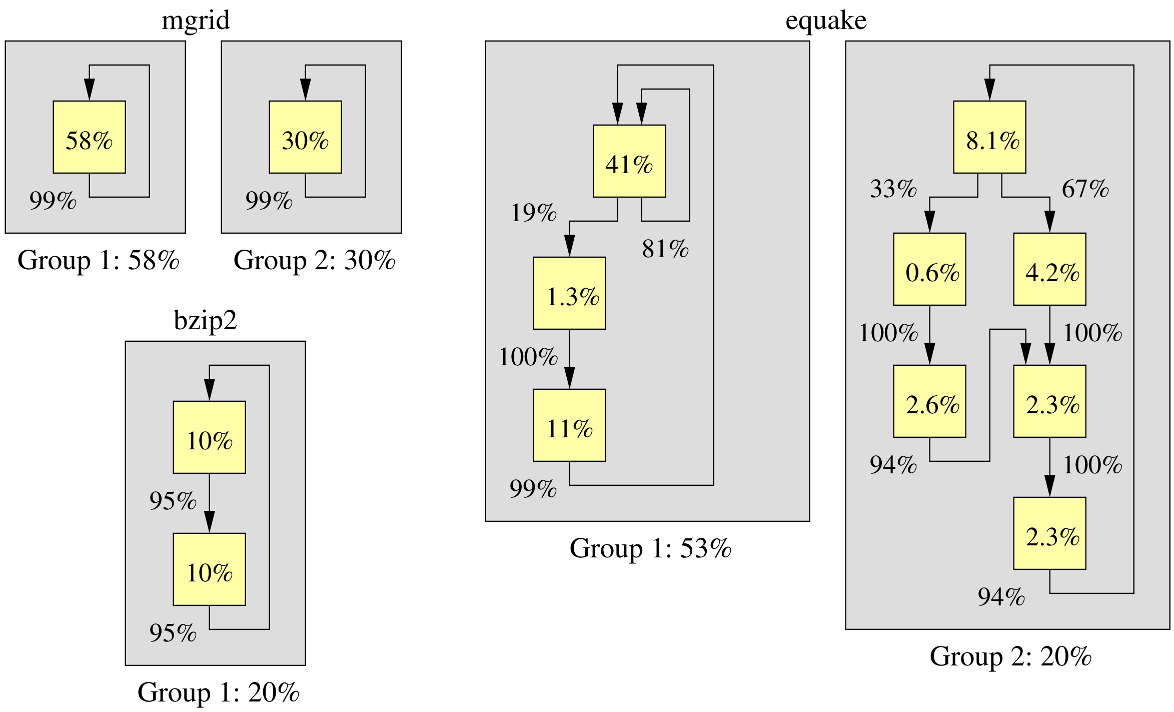

Figure 2.21: Example supertraces from analysis of exit counter profiling results. Each square is one trace, labeled with the percentage of total application execution time it covers. The main exits from each trace are labeled with their frequency and joined to their targets to produce closed groups, whose total execution time is shown. In the top example, from `mgrid`, traces already capture the hot code and the supertraces are identical to the traces. The rest of the examples show how combinations of traces were found to turn into supertraces.

with were too large to be practical (there was no small set of traces that formed a closed set).

Building larger traces by stitching together direct exits is not as important as inlining indirect exits into traces. Furthermore, very large traces may be detrimental to performance, just as too much loop unrolling is bad. Traces are better improved by targeting them to the performance bottlenecks of the application, rather than making them longer.

Higher-Level Traces

When operating on a layered system such as an interpreter with a higher-level application executing on a lower-level program, our trace building will be blind to the higher-level code and will try to find frequently executed code streams in the lower-level interpreter. This often leads to traces that capture a central dispatch loop, rather than specific sequences through that loop. In Section 9.3 we describe how to build *logical traces* rather than lower-level native traces to capture hot code in a higher-level application.

Local Contexts

The problem with longer traces is code duplication. Without internal control flow, we must unroll loops in order to build a trace that reaches our target. One alternative is to add internal control flow to traces. We can, however, maintain the attractive linear control flow properties of all our code fragments by using an idea we call *local contexts*. Each trace, in essence, has its own private code cache consisting of basic block fragments for each constituent element of the trace. These fragments are private to the trace and can only be accessed after entering the top of the trace. Exits from the trace return to the regular code cache. This local context is useful for any situation where a single trace must reach from one code point to another, such as from a call point to a corresponding return in order to inline the return. We have also proposed local context code duplication to aid in function pointer analysis for building a secure execution environment [Kiriansky et al. 2003].

Traces in Other Systems

NET was used for building traces in the Dynamo [Bala et al. 2000] system. Mojo [Chen et al. 2000] also used NET, with modified trace termination rules, though they never specified what those modifications were. The rePLay system [Fahs et al. 2001, Patel and Lumetta 1999] uses hardware to build traces based on branch correlation graphs. The Sable Java virtual machine [Berndl and Hendren 2003] combines the simple profiling of NET with branch correlation to create traces with higher completion rates than ours.

In an earlier study on compilation unit shapes for just-in-time compilers [Bruening and Duesterwald 2000], we found that inlining small methods is critical for Java performance. The same applies to IA-32, where there is a large discrepancy between the performance of a return instruction and the general indirect jump it must be transformed into inside the code cache (Section 4.2). In Section 9.2.4 we present a variation on our trace building that actively tries to inline entire procedure calls into traces, which is successful at improving performance on a number of benchmarks. Again, tailoring traces toward indirect branch inlining (in this case returns) is where we have found performance improvements.

2.4 Eliding Unconditional Control Transfers

A simple optimization may be performed when an unconditional jump or call instruction is encountered while building a basic block. Instead of stopping the block at the control transfer, it can be *elided* and the block continued at its target, which is statically known. This is an initial step toward building traces, which are described in Section 2.3.

Eliding unconditional control transfers provides a code layout benefit. However, it leads to duplicated code if the unconditional target is also targeted by other branches, since those other targets will build a separate basic block. If there are few such duplications, however, eliding unconditionals can result in less memory use because there are fewer basic blocks and therefore fewer corresponding data structures. We found that the performance and memory impact of eliding unconditionals varies significantly by application.

Figure 2.22 gives the performance impact of eliding unconditionals. Numbers both with and without traces are given, since eliding changes trace creation, as we discuss later. The `apsi` benchmark improves significantly, entirely due to unconditional jumps. The more minor improvements for the other benchmarks are mostly from direct calls. But a few benchmarks actually slow down, and the harmonic mean is just a slight improvement well within the noise.

Table 2.23 shows the code cache size impact. In some cases, more memory is used when eliding, due to duplicated code. However, for many applications, especially large Windows applications, there is a significant memory savings when eliding conditionals, because many of the unconditional targets are not targets of other branches, and so eliding ends up reducing the number of exit stubs (as well as data structures in the heap). Because of this memory benefit on these applications, DynamoRIO elides unconditional control transfers by default.

Table 2.24 shows the effect on individual basic block size when eliding unconditionals. The number of basic blocks drops since one block is now doing the work of two in all cases where the unconditional target is not reached through other branches. The average size of a basic block rises by about one-half. The maximum size does not change much — it seems that the extremely large blocks in these benchmarks only rarely contain unconditional transfers.

Care must be taken to maintain application transparency when eliding unconditionals. If the target is invalid memory, or results in an infinite loop, we do not want our basic block builder to

Benchmark	Basic block cache	Trace cache	No traces
ammp	-5.9%	10.8%	-3.6%
applu	-41.2%	-111.6%	-7.4%
apsi	-6.9%	-7.4%	1.7%
art	-11.1%	8.9%	-6.3%
equake	-2.6%	13.7%	-2.7%
mesa	-6.9%	5.9%	-5.0%
mgrid	-25.0%	-81.2%	-5.5%
sixtrack	0.2%	-1.2%	4.9%
swim	-12.6%	-10.6%	-6.3%
wupwise	-10.1%	-10.5%	-3.2%
bzip2	-6.9%	0.0%	-2.5%
crafty	2.5%	12.3%	6.0%
eon	8.8%	18.4%	7.0%
gap	1.3%	9.3%	5.7%
gcc	9.6%	16.4%	9.9%
gzip	-9.9%	7.7%	-4.1%
mcf	-9.9%	14.3%	-7.7%
parser	4.1%	9.9%	4.9%
perlbmk	9.0%	14.6%	8.4%
twolf	0.9%	37.0%	6.7%
vortex	17.8%	24.3%	16.8%
vpr	4.0%	19.9%	4.6%
excel	-3.1%	-2.0%	0.7%
photoshp	-13.7%	-16.1%	-9.4%
powerpnt	-6.2%	-12.2%	-2.1%
winword	-3.4%	-7.2%	-0.2%
average	-4.5%	2.9%	0.7%

Table 2.23: Cache size increase of eliding unconditionals. The first two columns give the basic block and trace cache increases, respectively. The final column gives the basic block increase when traces are disabled.

Benchmark	# blocks	Max bytes	Ave bytes	Max instrs	Ave instrs
ammp	-16.7%	1.4%	29.8%	1.2%	31.9%
applu	-28.6%	0.0%	35.5%	0.0%	38.2%
apsi	-23.7%	0.4%	61.6%	0.4%	69.8%
art	-18.1%	0.0%	29.8%	0.0%	29.8%
equake	-16.1%	22.3%	28.2%	5.1%	28.5%
mesa	-17.5%	2.1%	28.6%	0.0%	29.6%
mgrid	-20.8%	0.2%	48.2%	0.4%	52.2%
sixtrack	-19.8%	0.5%	70.8%	0.3%	85.6%
swim	-21.2%	0.7%	44.2%	1.0%	46.0%
wupwise	-21.9%	0.2%	59.8%	0.3%	63.2%
bzip2	-13.5%	0.0%	26.5%	45.7%	29.9%
crafty	-10.0%	0.0%	35.3%	0.0%	40.3%
eon	-16.0%	4.3%	39.2%	4.4%	56.9%
gap	-12.0%	1.8%	45.2%	7.8%	54.7%
gcc	-5.8%	0.7%	50.0%	1.0%	59.9%
gzip	-17.6%	0.0%	37.6%	58.6%	38.7%
mcf	-20.0%	0.0%	32.3%	0.0%	31.9%
parser	-10.0%	21.1%	47.7%	30.4%	49.9%
perlbmk	-10.0%	1.1%	52.1%	1.5%	60.6%
twolf	-11.8%	1.4%	34.4%	1.5%	36.9%
vortex	-8.4%	4.3%	102.0%	101.2%	105.2%
vpr	-14.1%	59.7%	51.7%	57.4%	55.4%
excel	-13.0%	5.4%	42.5%	4.6%	44.9%
photoshp	-9.1%	0.1%	86.1%	0.0%	90.4%
powerpnt	-12.5%	0.0%	51.0%	4.6%	53.9%
winword	-12.5%	0.0%	45.5%	0.0%	49.1%
average	-15.4%	4.9%	46.8%	12.6%	51.3%

Table 2.24: Effect on basic block sizes when eliding unconditionals, measured in both bytes and instructions (since IA-32 instructions are variable-sized). Each number is the percentage increase when eliding versus not eliding (the base numbers of not eliding are in Table 2.5).

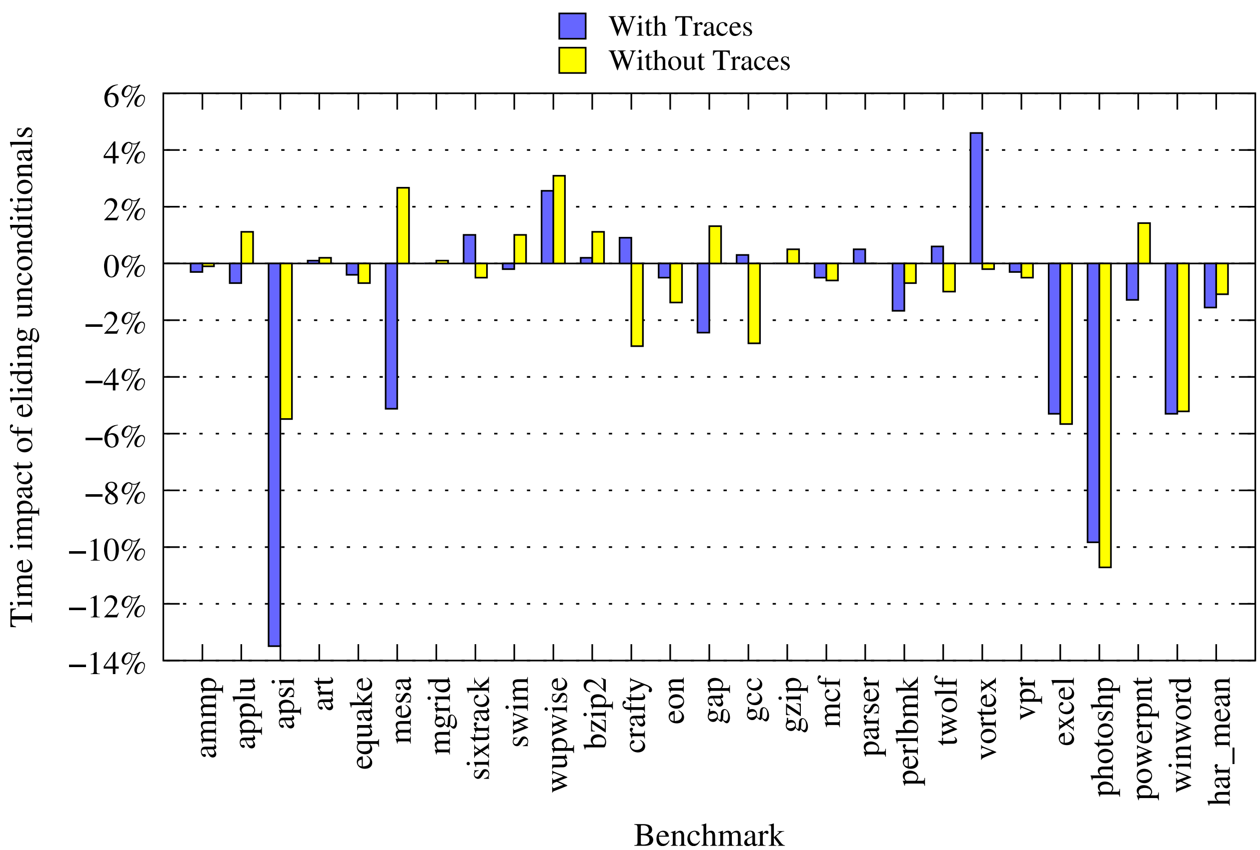

Figure 2.22: Performance impact of eliding unconditional control transfers when building basic blocks.

prematurely trigger that condition (this is *error transparency* — see Section 3.3.5). We check the target of the branch to see if it will result in a read fault (at the same time that we check its memory region for cache consistency purposes (Section 6.2.4)). To handle the infinite loop problem of blocks like that shown in Figure 2.25, our implementation uses a maximum basic block size.

Eliding unconditionals impacts trace building, since eliding *backward* unconditionals changes which blocks will become trace heads. As Figure 2.22 shows, `apsi` is particularly sensitive to eliding. It has basic blocks that are joined by backward unconditional jumps. If we do not elide such a jump, the second block will be its own trace head, and we will never end up placing the two blocks adjacent to each other, since traces always stop upon meeting other traces or trace heads. If we do elide the jump, the second block will not be a trace head, but we will have achieved superior code layout. Not considering a backward unconditional jump to mark trace heads could make a

```
loop: mov $0, %eax
      int $0x80
      jmp loop
```

Figure 2.25: An example of a troublesome basic block to decode when following unconditional control transfers. This block is for Linux kernels that use interrupt 80 as the system call gateway. System call zero is the exit system call. When this block is executed natively, the process exits prior to reaching the jump instruction.

difference, but the second block is often also targeted by a backward conditional jump. Eliding has an additional impact on building traces at call sites. When not eliding, a single basic block will represent the entrance of a callee. This makes it more difficult to create call-site-specific traces that cross into the callee. Eliding can enable the creation of more specific traces by ensuring that a trace that reaches the call site also reaches into the callee. The performance impact of eliding, independent of traces (with traces turned off), is shown as the second dataset in Figure 2.22. The improvement is less than it is when including traces (though still slightly positive on average), showing that eliding is complementary to, rather than competing with, trace building.

2.4.1 Alternative Super-block Designs

As eliding unconditionals proved, building larger units than classical basic blocks often reduces memory usage (since the data structures required to manage a small basic block are often larger than the block itself) and improves code layout. An area of future work is to allow internal control flow and build blocks that follow both sides of a conditional branch. Incorporating related work on increasing block sizes [Patel et al. 2000] could also be investigated.

2.5 Chapter Summary

This chapter introduced the fundamental components of DynamoRIO. Beginning with executing the application one basic block at a time out of a code cache, the crucial performance additions of direct linking, indirect linking via hashtable lookup, and traces bring code cache execution close to native speed. With this chapter as background, subsequent chapters turn to more novel contributions of this thesis, beginning with transparency in Chapter 3.

Chapter 3

Transparency

DynamoRIO must avoid interfering with the semantics of a program while it executes. Full transparency is exceedingly difficult for an in-process system that redirects all execution to a code cache. DynamoRIO must have its hands everywhere to maintain control, yet it must have such a delicate touch that the application cannot tell it is there.

The further we push transparency, the more difficult it is to implement, while at the same time fewer applications require it. It is challenging and costly to handle all of the corner cases, and many can be ignored if we only want to execute simple programs like the SPEC CPU [Standard Performance Evaluation Corporation 2000] benchmarks. Yet, for nearly every corner case, there exists an application that depends on it. For example, most programs do not use self-modifying code. But Adobe Premiere does. Another example is using code cache return addresses (see Section 3.3.3), which improve performance on our SPEC benchmarks but violate transparency enough to prevent execution of our desktop benchmarks. We found that every shortcut like this violates some program's dependencies. One of our most significant lessons from building DynamoRIO is that *to run large applications, DynamoRIO must be absolutely transparent*.

To achieve transparency, we cannot make any assumptions about a program's stack usage, heap usage, or any of its dependences on the instruction set architecture or operating system. We can only assume the bare minimum architecture and operating system interfaces. We classify aspects of transparency under three *rules of transparency*: avoid resource usage conflicts (Section 3.1), leave the application unchanged when possible (Section 3.2), and pretend the application is unchanged when it is not (Section 3.3).

3.1 Resource Usage Conflicts

Ideally, DynamoRIO's resources should be completely disjoint from the application's. That is not possible when executing inside the same process, but DynamoRIO must do its best to avoid conflicts in the usage of libraries, heap, input/output, and locks.

3.1.1 Library Transparency

Sharing libraries with the application can cause problems with re-entrancy and corruption of persistent state like error codes (see Section 5.2.2). DynamoRIO's dispatch code can execute at arbitrary points in the middle of application code. If both the application and DynamoRIO use the same non-re-entrant library routine, DynamoRIO might call the routine while the application is inside it, causing incorrect behavior. We have learned this lesson the hard way, having run into it several times. The solution is for DynamoRIO's external resources to come only from system calls and never from user libraries. This is straightforward to accomplish on Linux, and most operating systems, where the system call interface is a standard mechanism for requesting services (Figure 3.1a). However, on Windows, the documented method of interacting with the operating system is not via system calls but instead through an application programming interface (the *Win32 API*) built with user libraries on top of the system call interface (Figure 3.1b). If DynamoRIO uses this interface, re-entrancy and other resource usage conflicts can, and will, occur. To achieve full transparency on Windows, the system call interface (Figure 3.1c) must be used, rather than the API layer. (Other reasons to avoid the API layer are simplicity and robustness in watching application requests of the operating system (Section 5.4) and in intercepting callbacks (Section 5.3.1).) Unfortunately, this binds DynamoRIO to an undocumented interface that may change without notice in future versions of Windows.

Our initial implementation of DynamoRIO on Windows used the Win32 API. However, as we tried to run larger applications than just the SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmarks, we ran into numerous transparency issues. We then began replacing all Win32 API usage with the corresponding Native API [Nebbett 2000] system calls. DynamoRIO can get away with using some stateless C library routines (e.g., string manipulation), although early injection requires no library dependences (other than `ntdll.dll` — see

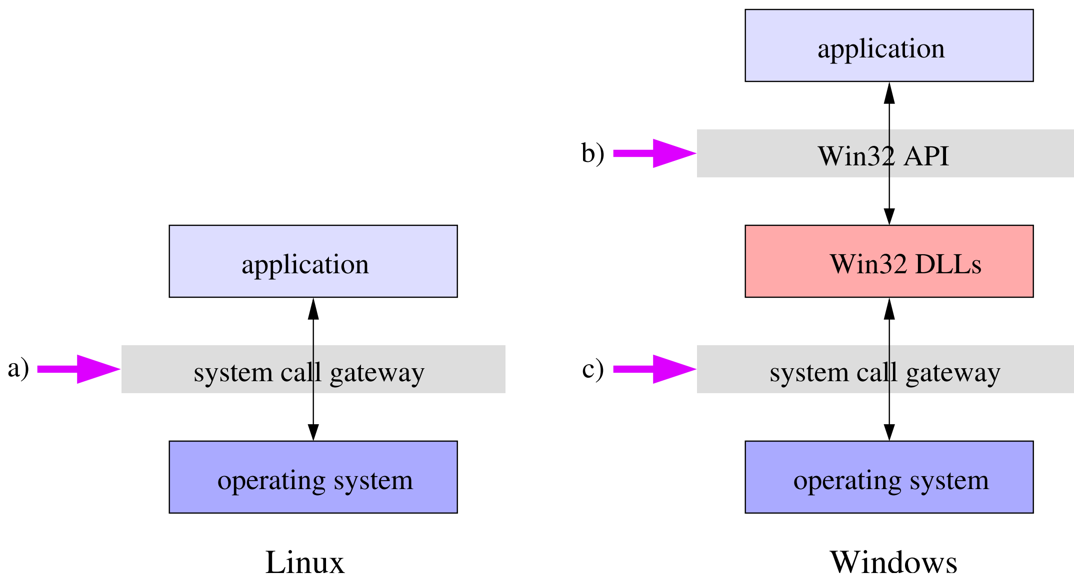

Figure 3.1: On the left is the usual relationship between an application and the operating system: the application invokes operating system services via a system call gateway. DynamoRIO can avoid application resource conflicts by operating at this system call layer, only requesting external services via system calls (*a*). On Windows, however, there is a layer of user-mode libraries that intervene, as shown on the right. The documented method for an application to request operating system services is through the Win32 application programming interface (Win32 API). This API is implemented in a number of user libraries, which themselves use the system call gateway to communicate with the operating system. Transparency problems result if DynamoRIO also operates through the Win32 API (*b*). The solution is to operate at the undocumented system call layer (*c*).

Section 5.5). The Native API is not officially documented or supported, but we have little choice but to use it.

On Linux we had some problems with using the C library and LinuxThreads. Our solution (Section 3.2.1) of using the `__libc_` routines raised more issues, however. The GNU C library [GNU C Library] changed enough between versions 2.2 and 2.3 with respect to binding to these routines that DynamoRIO built against one version is not binary compatible with the other version. We also ran into problems dealing with signal data structures at the system call level and via the C library simultaneously. Some of these structures have different layouts in the kernel than in `glibc`. Were we completely independent of the C library we could solely use the kernel version; in our implementation we must translate between the two. It is future work to become completely independent of all user libraries.

3.1.2 Heap Transparency

Memory allocated by DynamoRIO must be separate from that used by the application. First, sharing heap allocation routines with the application violates library transparency (Section 3.1.1) — and most heap allocation routines are not re-entrant (they are thread-safe, but not re-entrant). Additionally, DynamoRIO should not interfere with the data layout of the application (data transparency, Section 3.2.3) or with application memory bugs (error transparency, Section 3.3.5). DynamoRIO obtains its memory directly from system calls and parcels it out internally with a custom memory manager (see Section 6.4). DynamoRIO also provides explicit support in its customization interface to ensure that runtime tools maintain heap transparency, by opening up its own heap allocation routines to tools (see Section 8.2.9).

3.1.3 Input/Output Transparency

DynamoRIO uses its own input/output routines to avoid interfering with the application's buffering. As with heap transparency, DynamoRIO exports its input/output routines to tools to ensure that transparency is not violated (Section 8.2.9).

3.1.4 Synchronization Transparency

Shared locks can cause many problems. Concurrency is hard enough in a single body of code where protocols can be agreed upon and code changed to match them. When dealing with an arbitrary application, the only viable solution is to avoid acquiring locks that the application also acquires. This can be difficult for locks on needed data structures, like the `LoaderLock` on Windows, which protects the loader's list of modules. DynamoRIO's solution is to try to acquire the lock, and if unsuccessful to walk the target data structure without holding the lock and being careful to avoid either de-referencing invalid memory or entering an infinite loop (we use a maximum iteration count). Fortunately DynamoRIO does not need to write to these problematic data structures, only read from them. A better solution (future work) is for DynamoRIO to keep its own module list.

The same problem holds in reverse. DynamoRIO must worry about application threads sharing DynamoRIO routines and locks, and cannot allow an application thread to suspend another thread that is inside a non-re-entrant DynamoRIO routine or holding a DynamoRIO lock. Such synchronization problems are further discussed in Section 5.2.3.

3.2 Leaving The Application Unchanged When Possible

As many aspects of the application as possible should be left unchanged. Some cannot, such as shifting code into the code cache. But the original binary, application data, and the number of threads can be left unmodified.

3.2.1 Thread Transparency

DynamoRIO does not create any threads of its own, to avoid interfering with applications that monitor all threads in the process. DynamoRIO code is executed by application threads, with a context switch to separate DynamoRIO state from application state. Each application thread has its own DynamoRIO context (see Section 5.2). Using a dedicated DynamoRIO thread per application thread can be more transparent by truly separating contexts, and can solve other problems like thread-local state transparency (Section 5.2.2) and LinuxThreads transparency (see below). However, it can also cause performance problems in applications with hundreds or thousands of threads by doubling the number of threads, and other solutions exist to these separate problems that are more efficient as well as more transparent. Using a single DynamoRIO thread would be prohibitively expensive, as application threads would have to wait their turn for use of that one thread every time each wanted to enter DynamoRIO code.

The thread library on Linux, LinuxThreads [Leroy], does not have thread-local registers and locates thread-local memory by dispatching on the stack pointer. The threading library itself overrides weak symbols [Levine 1999] in the C library in order to create thread-aware routines. If DynamoRIO calls the normal C routines, it confuses the threading library since DynamoRIO's stack is not known to it (DynamoRIO uses a separate stack for stack transparency, Section 3.2.4). The cleanest solution (as stated in Section 3.1.1) is to not use the C library at all. For a short-term solution, however, since we did not want to modify the thread library to know about our stack, or use a separate thread paired up with each application thread (as mentioned above), we linked directly to lower-level C library routines that are exported for use in such situations (e.g., when the threading library itself needs to bypass its own thread-dispatching routines). These are mostly basic input and output routines like `__libc_open` and `__libc_read`. We also had to make our own version of `vsnprintf`. These problems with LinuxThreads have plagued other systems [Seward 2002]. The next generation of Linux threads [Drepper and Molnar] is cleaner and should obviate the need for these C library tricks. On Windows, the user libraries are built to support threads at all times and do not have such issues.

3.2.2 Executable Transparency

No special preparation of a program should be necessary for use with DynamoRIO. The program binary should be unmodified, and it should not matter what compiler was used, or whether a compiler was used. No source code or annotations should be required. Any binary that will execute on the processor natively should be able to execute under DynamoRIO.

3.2.3 Data Transparency

DynamoRIO leaves application data unmodified, removing a potentially enormous class of transparency problems. Preserving data layout requires heap transparency (Section 3.1.2).

3.2.4 Stack Transparency

The application stack must look exactly like it does natively. It is tempting to use the application stack for scratch space, but we have seen applications like Microsoft Office access data beyond the top of the stack (i.e., the application stores data on the top of the stack, moves the stack pointer to the previous location, and then accesses the data). Using the application stack for scratch space would clobber such data. Additionally, hand-crafted code might use the stack pointer as a general-purpose register. Other and better options for temporary space are available (see Section 5.2.1).

For its own code, DynamoRIO uses a private stack for each thread, and never assumes even that the application stack is valid. The problem with the code cache return address idea mentioned earlier (and further discussed in Section 3.3.3) is that many applications examine their stack and may not work properly if something is slightly different than expected. Another aspect of stack transparency overlaps with error transparency (Section 3.3.5): application stack overflows should not be triggered by the runtime system when they would not occur natively.

3.3 Pretending The Application Is Unchanged When It Is Not

For changes that are necessary (such as executing out of a code cache), DynamoRIO must warp events like interrupts, signals, and exceptions such that they appear to have occurred natively.

3.3.1 Cache Consistency

DynamoRIO must keep its cached copies of the application code consistent with the actual copy in memory. If the application unloads a shared library and loads a new one in its place, or modifies its own code, DynamoRIO must change its code cache to reflect those changes to avoid incorrectly executing stale code. This challenge is an important and difficult one when the underlying hardware keeps its instruction cache consistent and does not require explicit application work to modify code. Our algorithm for cache consistency is presented in Section 6.2.

3.3.2 Address Space Transparency

DynamoRIO must pretend that it is not perturbing the application's address space. An application bug that writes to invalid memory and generates an exception should do the same thing under DynamoRIO, even if we have allocated memory at that location that would natively have been invalid. This requires protecting all DynamoRIO memory from inadvertent (or malicious) writes by the application. Our solution of using page protection is discussed in detail in Section 9.4.5 in the context of building a secure execution environment, but the same technique is required to achieve address space transparency.

Furthermore, DynamoRIO must hide itself from introspection. For example, on Windows, some applications iterate over all loaded shared libraries using the `NtQueryVirtualMemory` [Nebbett 2000] system call to traverse each region of memory and the Windows API routine `GetModuleFileName` [Microsoft Developer Network Library] to find out if a library is present in that region. DynamoRIO detects such queries to its addresses and modifies the returned data to make the application think that there is no library there. This trick is required to correctly run certain applications, such as the `Mozilla` web browser, which install hooks in loaded libraries.

3.3.3 Application Address Transparency

Although the application's code is moved into a cache, every address manipulated by the application must remain an original application address. DynamoRIO must translate indirect branch targets from application addresses to code cache addresses, and conversely if a code cache address is ever exposed to the application, DynamoRIO must translate it back to its original application address. The latter occurs when the operating system hands a machine context to a signal or exception handler. In that case both the faulting or interrupted address and the complete register state must be made to look like the signal or exception occurred natively, rather than inside the code cache where it actually occurred (see Section 3.3.4 for how we translate in this direction).

As mentioned earlier, using code cache addresses as return addresses allows DynamoRIO to use return instructions directly and avoid any translation cost on returns. However, doing so requires that DynamoRIO catch every application access of the return address and translate it back to the application address. For example, position-independent code obtains the current program counter by making a call to the next instruction and then popping the return address. DynamoRIO can do pattern matching on this and other common ways the return address is read in order to get many applications to work, but even some of the SPEC CPU benchmarks [Standard Performance Evaluation Corporation 2000], like `perlbmk`, read the return address in too many different ways to detect easily. If DynamoRIO misses even one, the application usually crashes. The only general solution is to watch every memory load, which of course reverses the performance boost. See Section 4.2.1 and in particular Figure 4.7 for more information on this problem.

3.3.4 Context Translation

DynamoRIO must translate every machine context that the operating system hands to the application, to pretend that the context was originally saved in the application code rather than the code cache. This happens in exception and signal handlers, as mentioned above, and DynamoRIO's stateless handling of exceptions and signals (see Section 5.3) demands perfect context translation. Additionally, Windows provides a `GetThreadContext` Win32 API routine, and a corresponding system call, that enables one thread to obtain the context of another thread. DynamoRIO intercepts this call and translates the context so that the target thread appears to be executing natively instead of in the code cache.

Context translation takes several steps, each bringing the code cache context closer to the state it would contain natively. The first step is translating the program counter from the code cache to its corresponding application address. One option is to store a mapping table for each fragment. DynamoRIO's approach, to save memory, is to re-create the fragment from application code, keeping track of the original address of each instruction, and then correlate the code cache address to the address pointed at in the reconstruction at the same point in the fragment. Since the original application code cannot have changed since we built a fragment (see Section 6.2), we only need to store the starting address of a basic block, and the starting addresses of each block making up a trace. We then rebuild the fragment as though we were encountering new code, making sure to store the original address of each instruction. If this is a trace, we rebuild each constituent block. We re-apply any optimizations (our approach here only works for deterministic optimizations). Finally, we walk through the reproduction and the code cache fragment in lockstep, until we reach the target point in the code cache fragment. The application address pointed at by the corresponding instruction in the reconstructed fragment is the program counter translation.

The second step is ensuring that the registers contain the proper values. As Section 5.3 discusses, context translation can be limited to only controlled points outside of the code cache, and points inside where a fault can arise. In the absence of optimizations and other code transformations, only inserted code for indirect branches causes problems here (the load of the indirect branch target could fail). In this case several registers must have their application values restored to complete the translation (see Figure 4.19). DynamoRIO does not currently restore register values in the presence of optimizations.

Full translation for DynamoRIO is simpler than for systems that are interrupted at arbitrary times with events that cannot be delayed. These systems must be built to roll back or forward to a clean state from any location, not just at the few code transformation points of our base system (without optimizations).

3.3.5 Error Transparency

Application errors under DynamoRIO must occur as they would natively. An illegal instruction or a jump to invalid memory should not cause the DynamoRIO's decoder to crash — rather, the error must be propagated back to the application. However, since the decoder reads ahead in the instruction stream, the application executing natively may never have reached that point. Consider pathological cases like Figure 2.25. The best solution is to have the decoder suppress the exception and stop the basic block construction prior to the faulting instruction. Only if a new basic block is requested whose first instruction faults should it be delivered to the application. This also makes it easier to pass the proper machine context for the exception to the application, since the start of a basic block is a clean checkpoint of the application state. To implement error handling for decoding, checking every memory reference prior to accessing it is too expensive. A fault-handling solution is best, with a flag set to indicate whether the fault happened while decoding a basic block.

When an error is passed to the application, it needs to be made to look like it occurred natively. On Windows we encountered some unexpected complications in forging exceptions. Several IA-32 faults are split into finer-grained categories by the Windows kernel. For example, executing a privileged instruction in user mode results in a *general protection fault* from the processor, which is interrupt 13, indistinguishable from a memory access error. The Windows kernel figures out whether this was caused by a memory access or a privileged instruction, and issues different exception codes for each. Another example is an invalid lock prefix. The processor generates an *invalid opcode fault*, which is interrupt 6, just as it does for any undefined opcode. But Windows distinguishes the invalid lock prefix case from other undefined instructions, and uses different exception codes. DynamoRIO must emulate the Windows kernel behavior for full error transparency. See also Section 5.3.5 on how signal delivery requires kernel emulation.

Supporting precise synchronous interrupts in the presence of code modification is challenging. DynamoRIO currently does not do this in every case. As an example, we transform a call into a push and a jump. If the native call targets an invalid memory address, the push of the return address will be undone by the processor prior to signaling the exception. To faithfully emulate this, we must explicitly undo the push when we see an exception on the jump. If there is an exception on the push, we need do nothing special (in the absence of instruction-reordering optimizations, which DynamoRIO does not currently use, and which would require recovery code). Systems that perform aggressive optimizations often require hardware support to provide precise interrupts efficiently [Ebcioglu and Altman 1997, Klaiber 2000].

Error transparency overlaps with heap transparency (Section 3.1.2), stack transparency (stack overflows and underflows, Section 3.2.4), address space transparency (application writes targeting DynamoRIO data, Section 3.3.2), and context translation (translating contexts presented to application error handlers, Section 3.3.4).

We have seen actual cases of applications that access invalid memory natively, handle the exception, and carry on. Without error transparency such applications would not work properly under DynamoRIO.

3.3.6 Timing Transparency

We would like to make it impossible for the application to determine whether it is executing inside of DynamoRIO. However, this may not be attainable for some aspects of execution, such as the exact timing of certain operations. This brings efficiency into the transparency equation.

Changing the timing of multi-threaded applications can uncover behavior that does not normally happen natively. We have encountered race conditions while executing under DynamoRIO that are difficult to reproduce outside of our system. An example is Microsoft's Removable Storage service, in which under certain timing circumstances one thread unloads a shared library while another thread returns control to the library *after* it is unloaded. This is not strictly speaking a transparency violation, as the error *could* have occurred without us. Some of these timing violations might occur natively if the underlying processor were changed, or some other modification altered the timing.

3.3.7 Debugging Transparency

A debugger should be able to attach to a process under DynamoRIO's control just like it would natively. Previously discussed transparency issues overlap with debugging transparency. For example, many debuggers inject a thread into the debuggee process in order to efficiently access its address space. DynamoRIO would need to identify this thread as a debugger thread, and let it run natively, for full debugging transparency. Our implementation does not currently do this, but most debuggers work fine with DynamoRIO, including `gdb` [GDB] and the Debugging Tools for Windows [Microsoft Debugging Tools for Windows]. An exception is the Visual Studio debugger [Microsoft Visual Studio], which sometimes hangs or fails when attaching to a process under

DynamoRIO control.

We chose not to use debugging interfaces to control target applications primarily because of their coarse-grained and inefficient nature, but there is a transparency impact as well: only one debugger can be attached to a process at a time. If DynamoRIO used the debugger interface it would rule out attachment of any other debugger.

Our second transparency rule serves us well when interacting with a debugger. Stack transparency and data transparency make debugging the application nearly identical to debugging it when running natively, including call stacks. The main difference is, of course, that the program counter and sometimes register values are different. One solution is that taken by Chaperon, a runtime memory error detector that ships with Insure++ [Parasoft]. Chaperon includes a modified version of the `gdb` debugger that automatically translates the machine context (or at least the program counter) from the code cache to the original application code location.

3.4 Chapter Summary

Transparency is critical for allowing a runtime code manipulation system to faithfully execute arbitrary applications. In building DynamoRIO, we learned many transparency lessons the hard way, by first trying to cut corners, and only when one application or other no longer worked, solving the general case. This chapter tries to pass on the lessons we learned. DynamoRIO also helps runtime tools built on top of it to maintain transparency (see Section 8.2.9).

Chapter 4

Architectural Challenges

A software code manipulator faces many constraints imposed by the hardware, and even small differences between architectures can make huge differences in the performance and transparency of a code caching system. A problematic aspect of modern processors is optimization for common application patterns. An application executing under control of a runtime code manipulation system may end up with different patterns of usage than when executing natively, and if these do not match what the hardware has been optimized for, performance can suffer. DynamoRIO cannot match native behavior with respect to indirect branches (Section 4.2), data cache pressure (Section 4.3.3), or code modification frequency (Section 4.6). This chapter discusses these problems and other features of a Complex Instruction Set Computer (CISC) architecture, and IA-32 in particular, that gave us headaches. DynamoRIO's most significant architectural challenges include CISC's namesake, the complexity of the instruction set (Section 4.1); return instruction branch prediction discrepancies (Section 4.2); hashtable lookup optimization (Section 4.3); condition code preservation, with implicit and pervasive condition code dependences throughout the instruction set (Section 4.4); the processor's instruction cache consistency (Section 4.5) and trace cache (Section 4.6); and efficient machine context switches (Section 4.7).

4.1 Complex Instruction Set

The CISC IA-32 architecture has a complex instruction set, with instructions that vary in length from one to seventeen bytes. While it is difficult to decode instruction boundaries and opcodes, encoding is even harder, due to the specialized instruction templates that vary depending on the operand values themselves [Intel Corporation 2001, vol. 2]. The instruction set is challenging enough that DynamoRIO, unlike most code caching systems on RISC architectures, does not contain an emulator and only makes forward progress by copying code into a code cache to allow the processor to interpret it. To minimize decoding and encoding costs, we use a novel adaptive level-of-detail instruction representation, described below.

This section also discusses handling segments (Section 4.1.2), a CISC feature that can be both useful and painful, and issues with branch reachability (Section 4.1.3), which is not limited to CISC and in fact is more pronounced on RISC architectures whose immediate operand sizes are limited. Issues with transparently handling decoding problems, such as passing illegal instruction and invalid memory faults on to the application, are described elsewhere (Section 3.3.5).

Another CISC difficulty run into when manipulating IA-32 code is the paucity of software-exposed registers. This results in a lack of scratch space, making it more difficult to perform code transformations and optimizations than on most RISC architectures. A related problem is the high percentage of IA-32 instructions that reference memory. Memory references seriously complicate code analysis, further increasing the difficulty of correctly transforming code.

4.1.1 Adaptive Level-of-Detail Instruction Representation

In any system designed to manipulate machine instructions, the instruction representation is key. Ease and flexibility of use have been traditional concerns for compiler writers. Runtime systems add an additional concern: performance. DynamoRIO uses two key features to achieve efficiency. The first is to simplify code sequences by only allowing *linear control flow*. Recall from Chapter 2 that DynamoRIO operates on two kinds of code sequences: basic blocks and traces. Both have linear control flow, with a single entrance at the top and potentially multiple exits but no entrances in the middle. The single-entry multiple-exit format greatly simplifies many aspects of the core system. Linear control flow also simplifies code analysis algorithms in tools built with DynamoRIO, reducing tool overheads as well.

Since each of DynamoRIO's code sequences is linear, we can represent it as a linked list of instructions (not an array because we must support efficient insertion and deletion during code manipulation). We use a data structure called `InstrList` to represent one code sequence. An `InstrList` is composed of a linked list of `Instr` data structures. An `Instr` can represent a single instruction or a group of bundled un-decoded instructions, depending on its level of detail, which we discuss next.

Many runtime systems resort to internal, low-level, often difficult-to-use, instruction representations in the interest of efficiency, since decoding and encoding times add to the overhead of every code manipulation. Existing exported IA-32 code representations, such as VCODE [Engler 1996], focus on a RISC subset of the instruction set. Some code caching systems translate IA-32 to RISC-like internal representations, as well [Seward 2002]. These translations sacrifice transparency and performance to gain simplicity. Preserving the original application instructions is important for duplicating native behavior, especially for tools that would like to study the instruction makeup of applications. Our solution provides the full IA-32 instruction set but at an acceptable performance level. We have developed an adaptive level-of-detail representation, where an instruction is only decoded and encoded as much as is needed. Since code analyses are often only interested in detailed information for a subset of instructions, this detail-on-demand can provide significant savings.

Our adaptive level-of-detail instruction representation uses five different levels, which are illustrated in Figure 4.1:

Level 0 – At its lowest level of detail, the instruction data structure `Instr` holds the raw instruction bytes of a series of instructions and only records the final instruction boundary.

Level 1 – At the next level, an `Instr` again stores only un-decoded raw bits, but it must represent a single machine instruction.

Level 2 – At Level 2, the raw bits are decoded enough to determine the instruction's opcode and effect on the `eflags` register (which contains condition codes and status flags) for quick determination of whether the `eflags` must be saved or restored around inserted instructions. Many IA-32 instructions modify the `eflags` register, making them an important factor to consider in any code transformation.

Level 3 – A Level 3 instruction is fully-decoded, but its raw bits are still valid. It uses `Instr`'s fields for opcode, prefixes, and `eflags` effects, plus two dynamically-allocated arrays of operands, one for sources and one for destinations. The first source is stored statically, Level 0

raw bits
`8d 34 01 8b 46 0c 2b 46 1c 0f b7 4e 08 c1 e1 07 3b c1 0f 8d a2 0a 00 00`

Level 1

raw bits
`8d 34 01`
`8b 46 0c`
`2b 46 1c`
`0f b7 4e 08`
`c1 e1 07`
`3b c1`
`0f 8d a2 0a 00 00`

Level 2

raw bits	opcode	eflags
`8d 34 01`	`lea`	
`8b 46 0c`	`mov`	
`2b 46 1c`	`sub`	`WCPAZSO`
`0f b7 4e 08`	`movzx`	
`c1 e1 07`	`shl`	`WCPAZSO`
`3b c1`	`cmp`	`WCPAZSO`
`0f 8d a2 0a 00 00`	`jnl`	`RSO`

Level 3

raw bits	opcode	operands	eflags
`8d 34 01`	`lea`	`(%ecx,%eax,1) -> %esi`	
`8b 46 0c`	`mov`	`0xc(%esi) -> %eax`	
`2b 46 1c`	`sub`	`0x1c(%esi) %eax -> %eax`	`WCPAZSO`
`0f b7 4e 08`	`movzx`	`0x8(%esi) -> %ecx`	
`c1 e1 07`	`shl`	`$0x07 %ecx -> %ecx`	`WCPAZSO`
`3b c1`	`cmp`	`%eax %ecx`	`WCPAZSO`
`0f 8d a2 0a 00 00`	`jnl`	`$0x77f52269`	`RSO`

Level 4

raw bits	opcode	operands	eflags
	`lea`	`(%edx,%eax,1) -> %esi`	
`8b 46 0c`	`mov`	`0xc(%esi) -> %eax`	
`2b 46 1c`	`sub`	`0x1c(%esi) %eax -> %eax`	`WCPAZSO`
	`movzx`	`0x8(%esi) -> %edx`	
	`shl`	`$0x07 %edx -> %edx`	`WCPAZSO`
	`cmp`	`%eax %edx`	`WCPAZSO`
`0f 8d a2 0a 00 00`	`jnl`	`$0x77f52269`	`RSO`

Figure 4.1: Example sequence of instructions at each of the five levels of representation. Level 0 knows only the raw bits; Level 1 breaks the bits into a linked list of individual instructions; Level 2 decodes the opcode and `eflags` (condition codes) effects; Level 3 is fully decoded, with operands; and Level 4 is reached if an instruction's opcode or operands are modified (in this example, we replaced `ecx` with `edx`), invalidating the original raw bits. Operands are listed in `sources -> destinations` format.

targeting the common case in DynamoRIO of a control transfer instruction. The rest of the operands are dynamically allocated because IA-32 instructions may contain between zero and eight sources and destinations. This level combines quick encoding (simply copy the raw bits) with high-level information.

Level 4 – The final level is a fully-decoded instruction that has been modified (or newly created) and does not have a valid copy of raw instruction bits. This is the only level at which instructions must be fully encoded (or re-encoded) to obtain the machine representation.

To support the multiple `Instr` levels, multiple decoding strategies are employed. The lowest level simply finds instruction boundaries (even this is non-trivial for IA-32). Although the instruction boundaries need to be determined for both Level 0 and Level 1, the boundary information may not be needed later. Level 0 avoids storing that information, and further simplifies encoding by allowing a single memory copy rather than an iteration over multiple boundaries. Level 2 decodes just enough to determine the opcode and the instruction's effect on the `eflags`. Finally, for Level 3 and Level 4, a full decode determines all of the operands. The initial level of an `Instr` is determined by the decoding routine that is used to build the instruction. Later operations can change an instruction's level, either implicitly or explicitly. For example, asking for the opcode of a Level 1 instruction will cause it to be raised to Level 2. Modifying an operand of a Level 3 instruction will cause the raw bits to become invalid, moving it up to Level 4. This automatic adjustment makes it easy for an optimization to use the lowest cost representation possible, with further details available on demand. Switching incrementally between levels costs no more than a single switch spanning multiple levels. (Section 8.2.1 further discusses DynamoRIO's decoding, encoding, and other instruction manipulation routines, which are exported to tool builders.)

When encoding an `Instr` whose raw bits are valid, the encoder simply copies the bits. If the raw bits are invalid (Level 4), the instruction must be fully encoded from its operands. Encoding an IA-32 instruction is costly, as many instructions have special forms when the operands have certain values. The encoder must walk through every operand and find an instruction template that matches. DynamoRIO avoids this by copying raw bits whenever possible.

As an example of the use of various levels of instruction representation, consider the creation of a basic block fragment. DynamoRIO only cares about the control flow instruction terminating the block. Accordingly, the `InstrList` for a basic block typically contains only two `Instr`s, one at Level 0 that points to the raw bits of an arbitrarily long sequence of non-control flow instructions, and one at Level 3 that holds the fully decoded state for the block-ending control flow instruction, ready for modification. This combination of Level 3 for control flow instructions and Level 0 for all others is common enough to be explicitly supported by DynamoRIO (and in our exported API, as presented in Section 8.2.1). We call it the Control Transfer Instruction (CTI) Level. Another example of level-of-detail use is a tool performing optimizations. The tool might fully decode (Level 3) many instructions in order to analyze operands, but may only modify a few of them. All unmodified instructions will remain at Level 3 and can be quickly encoded by copying their raw bits.

For a quantitative evaluation of the different levels of instruction representation, we measured the resources required to decode and encode instructions at each level of detail. Table 4.2 gives the average time and memory to decode and then encode individual instructions and entire basic blocks, across all of the basic blocks in the SPEC CPU2000 benchmarks [Standard Performance Evaluation Corporation 2000]. As expected, Level 0 outperforms the others, with a significant jump in memory usage when moving to individual data structures for each instruction in a basic block. Another jump occurs at Level 4, for which encoding is much more expensive. The combination that we use, Level 3 for control transfer instructions but Level 0 for all others, gives performance intermediate between Level 0 and Level 1.

To show the impact of our adaptive representation on DynamoRIO, Figure 4.3 gives the performance improvement of only decoding instructions to the level required by DynamoRIO (the combination of 0 and 3), compared to fully decoding all instructions to Level 4. The performance difference is significant only for benchmarks that execute large amounts of code, such as `gcc` and our desktop benchmarks, but it reaches as much as nine percent on these applications. Adaptive decoding becomes even more important when a tool is in operation, performing extensive code transformations beyond simply executing the application.

4.1.2 Segments

The IA-32 architecture supports not only paging but segmentation as well. IA-32 uses several segment selectors:

Level	Individual instruction Cycles	Entire basic block Time (μs)	Memory (bytes)
0	92	2.37	64.0
1	531	10.82	579.0
2	615	12.74	579.0
3	882	18.07	730.1
4	2908	59.78	730.1
CTI	272	5.81	158.6

Table 4.2: Performance of decoding and then encoding the basic blocks of the SPEC CPU2000 benchmarks [Standard Performance Evaluation Corporation 2000], on a Pentium 4, at each level of detail. The first column shows the average number of cycles for an individual instruction, while the final two columns give the average time and memory to decode and then encode an entire basic block. The mixture of Level 3 for control transfer instructions and Level 0 for all others, what we call Level CTI, is intermediate between 0 and 1 in performance.

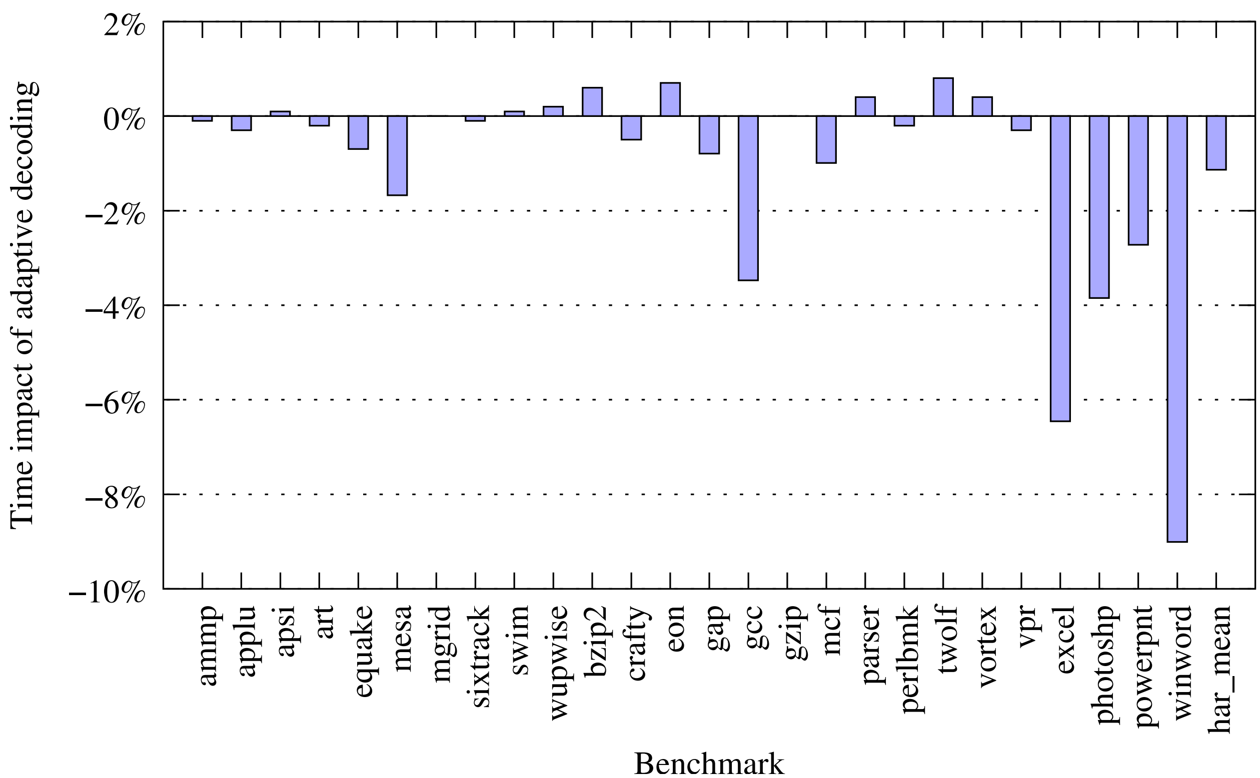

Figure 4.3: Performance impact of decoding instructions as little as possible, using Level 3 for control transfer instructions but Level 0 for all others.

- CS: the segment used for code
- SS: the segment used for stack references
- DS: the segment used for data access, unless overridden by an instruction prefix
- ES: the segment used for data access for string instructions (along with DS)
- FS and GS: available for use as alternate data segments

As can be imagined, it is changes to CS that present the most difficulties, but supporting segment use of any kind by applications is a tricky task. Every code transformation and every address manipulation must be carefully arranged to work with segment prefixes on instructions. Each stored address must either include the segment or be translated to a linear address. This is not feasible in operating systems like Windows that do not give access to the segment descriptor tables to user-mode code. This makes it too difficult to test for identity of segment descriptors based on selectors, and too difficult as well to convert to a linear address. Fortunately, operating systems like Windows that do not give access to descriptor information typically do not allow an application to create its own descriptors.

If the descriptors can be accessed and segment-based addresses can be manipulated, application segment use in general can be dealt with. Complications include saving and restoring the segment selectors along with the general-purpose registers on context switches to and from the code cache. Far (i.e., cross-segment) indirect transfers need their own hashtable lookup routine that ends in an indirect jump with the proper far target. We have not implemented such support on Linux. DynamoRIO does not allow CS to change (thus disallowing far jumps, calls, and returns). DynamoRIO disallows changes to and requires a flat address space for all segments except for FS and GS, which we do support custom uses of. This is reasonable, as 32-bit Windows applications cannot use other segments (other than the FS set up by the operating system, as discussed in Section 5.2.1), and very few Linux applications do. The only Linux cases we have seen are WINE [WINE], which uses FS to emulate Windows' use of it, and the new Linux threading library [Drepper and Molnar], which uses GS to provide thread-local storage.

```
application code:
                 jecxz foo
is transformed into:
                 jecxz ecx_zero
                 jmp ecx_nonzero
       ecx_zero: jmp foo
    ecx_nonzero:
```

Figure 4.4: How we transform eight-bit branches that have no 32-bit counterpart into a semantically equivalent branch with a 32-bit reach.

4.1.3 Reachability

CISC does have some advantages for a code manipulation system. It allows absolute addressing of the entire address space, which avoids messy reachability issues and landing pads that other architectures require [Bala et al. 1999]. Reachability problems do exist with the set of IA-32 branches that are limited to eight-bit relative offsets. We transform these into 32-bit versions to avoid problems in reaching link targets, which could end up much further away in the code cache than in the native code. Most of the eight-bit branch instruction have direct 32-bit counterparts, although a few (`jecxz`, `loop`, `loope`, and `loopne` [Intel Corporation 2001, vol. 2]) have no alternative version. We transform these into the sequence shown in Figure 4.4.

4.2 Return Instruction Branch Prediction

Branch prediction can be a significant factor in the performance of modern out-of-order superscalar processors. A mispredicted branch can result in tens of cycles lost. Modern IA-32 processors (from the Pentium Pro onward) contain three types of branch predictors [Intel Corporation 1999]:

1. Static branch prediction

   The processor uses static rules to predict whether a branch will be taken. The rules state that a branch will be taken if it is unconditional or backward, but that a forward conditional branch will not be taken.

2. Dynamic branch prediction

   A Branch Target Buffer (BTB), indexed by instruction address, is used to determine whether a conditional branch will be taken (it tracks the last four branch directions for each branch entry) and to predict the target of an indirect branch.

3. Return address prediction
   A Return Stack Buffer (RSB) is used to predict the target of return instructions.

Without the RSB, returns would nearly always cause mispredictions, since a single return instruction typically targets a number of different call sites, while the BTB only contains one predicted address. The RSB enables extremely good prediction for returns. This results in a discrepancy in branch prediction for returns versus indirect jumps and indirect calls. This discrepancy is the single largest obstacle toward achieving comparable-to-native performance on IA-32. Since return addresses in DynamoRIO must be translated using an indirect branch lookup (for transparency – see Section 3.3.3), the return instruction cannot be used in a straightforward manner. Instead, an indirect jump is used, which results in poor branch prediction compared to the native version of the code. Figure 4.5 shows the performance difference when return instructions are replaced by a semantically equivalent sequence that manually increments the stack pointer and targets the return address with an indirect jump. The resulting slowdown can reach twenty percent, with a harmonic mean of nearly five percent. In micro-benchmarks, using an indirect jump can be twice as slow as using a return.

Kim and Smith [2003a] found that the RSB limited performance of a dynamic translation system so much that a dual-address RSB was critical to approaching native performance. Kim and Smith [2003b] recently proposed methods similar to our hashtable lookup inlining (see Section 4.3.1) to mitigate this problem in software, but focus mainly on hardware solutions. The best solution for future IA-32 processors would be to expose the RSB to software. If DynamoRIO could supply an RSB hint it could achieve an immediate performance boost. Intel's next generation of the Pentium 4, Prescott, promises to sport a better indirect jump predictor than the BTB, which should help alleviate DynamoRIO's branch prediction performance problems.

4.2.1 Code Cache Return Addresses

We came up with numerous indirect branch lookup designs that tried to make use of the RSB. The obvious one is to violate transparency and place code cache return addresses, rather than original

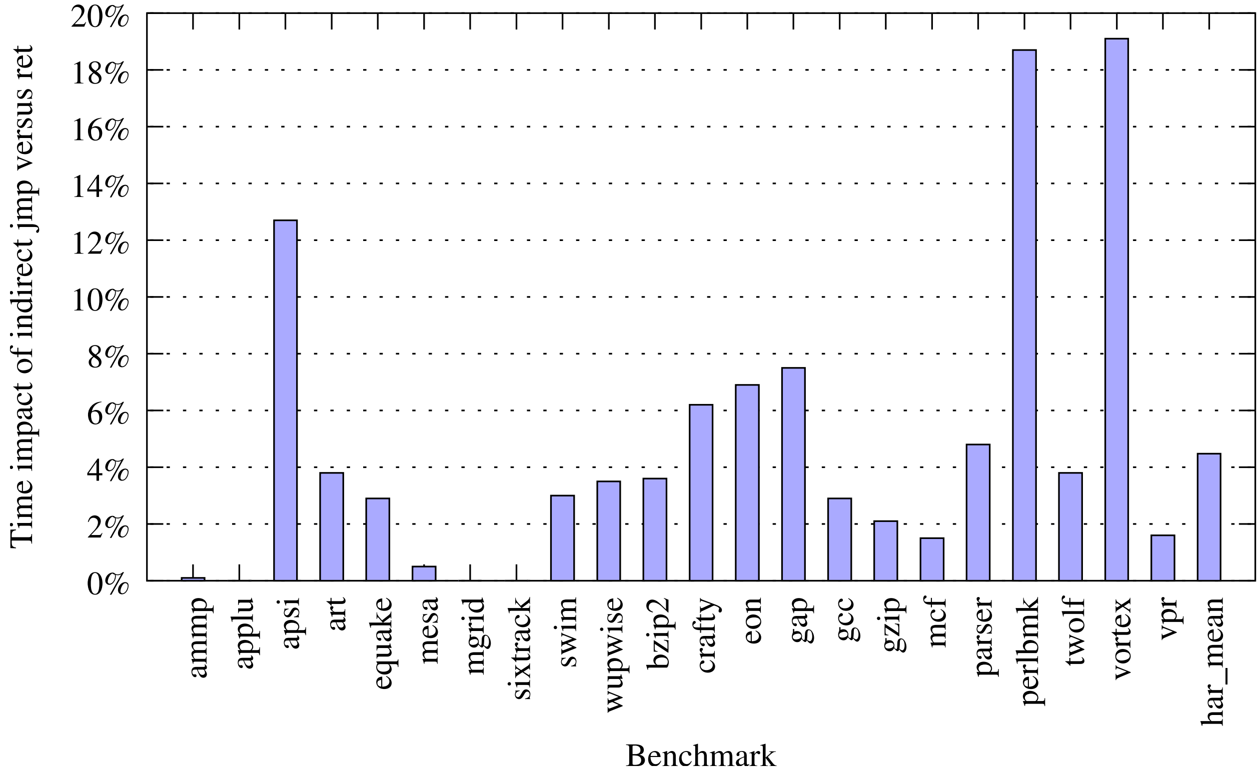

Figure 4.5: Performance impact of replacing return instructions with indirect jumps on our Linux benchmarks. Only the benchmark code itself was modified; the C and C++ libraries were left untouched.

application addresses, on the stack. Then a return instruction can be issued instead of doing a hashtable lookup. Figure 4.6 shows the code transformations that must be used to get this to work with independent basic blocks. Since the RSB is not exposed to software, the only way to get an address into it is to issue a call instruction. The code shown elides unconditional calls (Section 2.4).

Figure 4.7 lists the numerous transparency complications of using this approach, all of which we encountered. The only efficient solution is to pattern-match for situations like these. Zovi [2002] handles calls with no matching return, and vice versa, that are used for dynamic linking by ignoring calls or returns that both originate in and target the runtime linker code section.

This scheme has several other problems besides application transparency. One is that if we do not take over the application at the very beginning, we may see returns for which we never saw the call. One solution is a call depth counter and a check on every return to see if the counter will go

```
call   becomes     call skip
                   jmp app_return_address
                 skip:
                   continue in callee

call*  becomes     call skip
                   jmp app_return_address
                 skip:
                   jmp indirect_branch_lookup (via exit stub)

ret    becomes     ret
```

Figure 4.6: Call and return transformations to enable use of the return instruction. This code assumes that no calls have occurred prior to taking control of the application (otherwise "bottoming out" of the return stack needs to be checked for on a return).

```
1) call with no matching return, not uncommon, used to
   find the current instruction address:
              call next_instr
  next_instr: pop

2) return with no matching call
   example is Linux PLT binding:
   dl_runtime_resolve_addr calls fixup(), which places
   the destination address in eax, and then:
              xchg   (%esp) %eax
              ret    $8

3) PIC code grabbing base address from return address
              call PIC_base
              ...
    PIC_base: mov (%esp), %ebx
              ret

4) other manipulation of return address, sometimes
   in later basic blocks, even across indirect jumps!
              call foo
              ...
         foo: ...
              add (%esp), %eax
              ...
```

Figure 4.7: Transparency problems with using non-application return addresses.

negative. Of course, as discussed in Section 4.4, incrementing a counter in a transparent manner is an expensive proposition on IA-32.

Another problem is that the target of a return address may not still be in the cache when the return is executed. That is yet more overhead that must be added to the return transformation of Figure 4.6. This cannot be optimized by only touching up the return stack on a fragment deletion, because there is no way to know where else return addresses might be copied or stored without making compiler and application assumptions.

Unlinking a return is another issue that needs to be addressed. Since we must be able to unlink any fragment (Section 2.2), we either need more logic in front of the return instruction or we must overwrite the return with a jump to an exit stub.

Finally, DynamoRIO must decide whether a trace ends at a return with a native return instruction or whether the return target should be inlined into the trace.

We hardcoded solutions for all of these problems to see the performance impact, which exceeds ten and even fifteen percent on several integer benchmarks (Figure 4.8). We were able to execute our SPEC benchmarks by pattern-matching all of the problematic scenarios, but we gave up on executing our larger desktop benchmarks, which contain too many accesses to the return address. A general solution to these problems, in particular problem 4 of Figure 4.7, is not feasible, since every memory instruction would need to be watched to make sure every read from the stack was transformed into a read of the application return address instead of the code cache return address. Using code cache addresses on the stack is not an acceptable solution for our goals of transparency and universality.

4.2.2 Software Return Stack

Since it is not feasible to use code cache return addresses, other options include a normal hashtable lookup followed by pushing the result and then performing a return, or using a return stack in software instead of the hashtable lookup. We implemented a software return stack as efficiently as we could. To avoid the cost of transparently juggling two stacks at once, we store a pair of the application return address and the code cache return address as an element on our shadow stack. The application stack remains transparent, storing the application return address only, to avoid the problems of using a code cache address on the application stack (Figure 4.7). Our private shadow

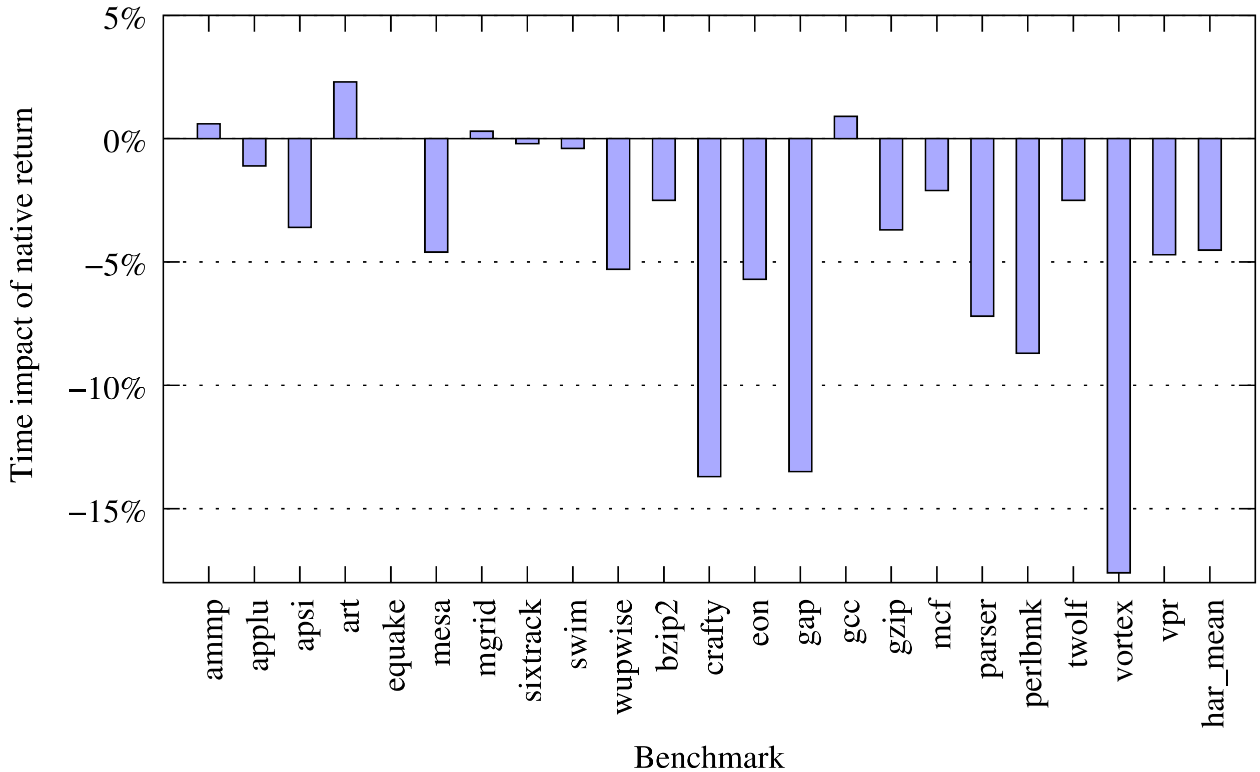

Figure 4.8: Performance impact of using native return instructions by using code cache addresses as return addresses on the stack, on our Linux benchmarks (the transparency problems listed in Figure 4.7 prevented us from executing our desktop applications with this scheme), versus a hashtable lookup with an indirect jump.

stack also allows us to perform cache management, since we can locate all fragments that are represented there, unlike code cache addresses being handed to the application.

In order for the native return instruction to match up in the hardware RSB, the call must be an actual call instruction. This can be arranged in several different ways, but they all require leaving a return slot immediately after the call. The sequence used when eliding direct calls (Section 2.4) is shown in Figure 4.9. It switches to the shadow stack to make the call, which pushes the code cache return address, and explicitly pushes the other half of the pair, the application return address. This sequence is easy to unlink and to include in a trace.

An alternative is to not elide and have a call that, when linked, targets the callee fragment. The call is made while using the shadow stack, and there are two options on where to restore the application stack: to use a prefix in the callee that restores the stack, in which case the prefix must

```
    push <application return address>
    <swap to shadow stack>
    push <application return address>
    call cleanup_stack
    jmp <after call fragment>
cleanup_stack:
    <swap to application stack>
    <continue in callee>
```

Figure 4.9: Instruction sequence to get the code cache return address on the RSB, when eliding direct calls. A shadow return stack is used that contains <application return address, code cache return address> pairs.

```
    push <application return address>
    <swap to shadow stack>
    push <application return address>
    call cleanup_stack
    jmp ret_point
cleanup_stack:
    <swap to application stack>
    jmp <callee fragment>
ret_point:
    <code after call>
```

Figure 4.10: Instruction sequence to get the code cache return address on the RSB, when not eliding direct calls and not using a prefix.

be stripped off while building a trace, or to target the callee with a later jump rather than the call, allowing restoration of the application stack prior to transferring control to the callee. The latter is shown in Figure 4.10. Code after the call can be placed in the same fragment, as our code shows, but this makes trace building difficult, so it is better to simply end the fragment with the transfer to the callee.

In either case, on a return, we swap to our shadow stack and pop the pair of application and code cache addresses. We compare the stored application address to the real one on the application stack. If they match, we replace the real address with the code cache address and perform a return instruction; if not, we do a hashtable lookup.

Although maintaining application stack transparency avoids most of the problems of Figure 4.7, we still must deal with non-paired calls and returns and violations of the calling convention, like

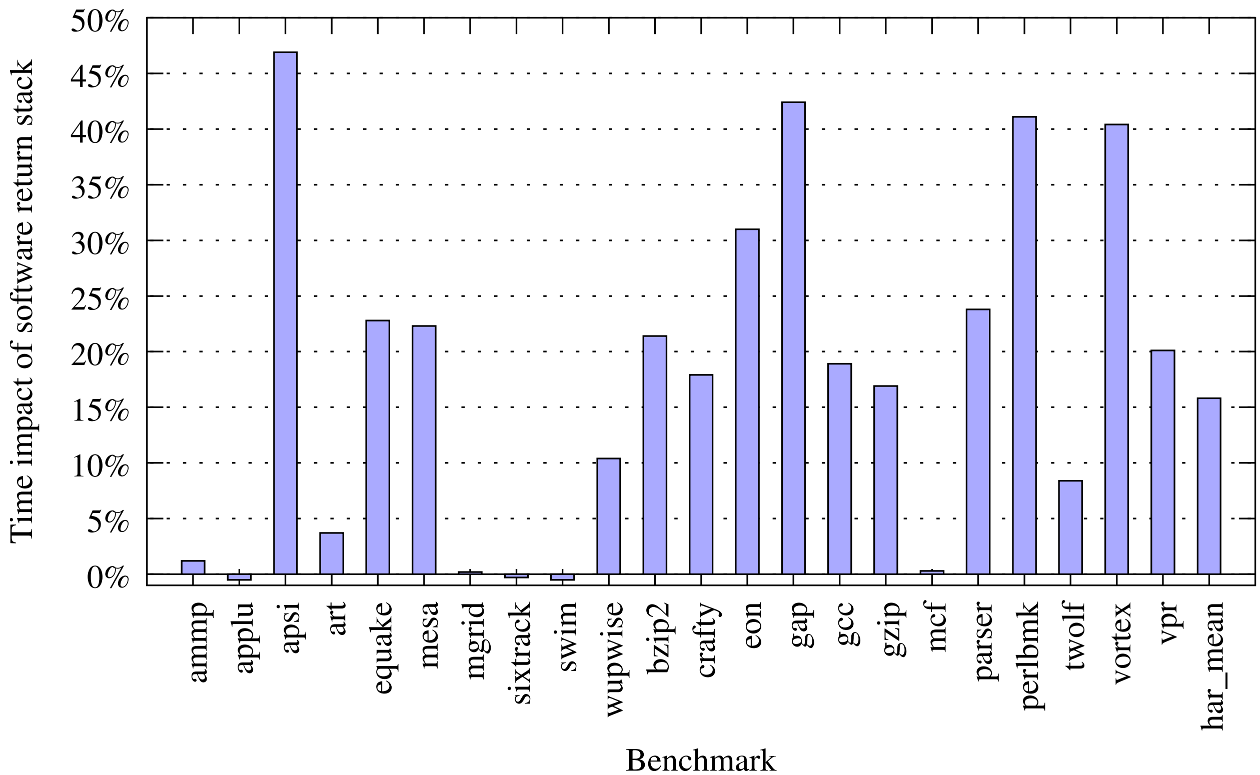

Figure 4.11: Performance impact of using a software return stack on our Linux benchmarks, versus a hashtable lookup.

`longjmp`. Fortunately, we fail gracefully, simply falling back on our hashtable lookup, and will not crash in any of these situations. One heuristic we could use is to keep popping the shadow stack on a miss until a hit is found (or the stack bottoms out) [Prasad and Chiueh 2003]. This ensures that any return to an ancestor, such as `longjmp`, will result in the proper stack unwinding.

The result of all this work is application return instructions that are actually using the RSB inside of our code cache. However, the performance is disappointing: as Figure 4.11 shows, using our shadow return stack is actually slower than our base system, which performs a hashtable lookup and an indirect jump. The numbers shown are a comparison against our best hashtable lookup implementation, using an inlined lookup routine that helps branch prediction (see Section 4.3.1). Against a non-inlined version, the average slowdown is slightly better but still bad, at thirteen percent. Even though we have improved branch prediction, we have added too many memory operations on every call and return. Applications with many calls suffer the most, and extra data cache pressure overwhelms the branch prediction gains.

We also tried using a one-element return stack, since in many programs leaf calls are frequent and a one-element stack will result in a high percentage of hits. The performance was much better than the full return stack, but its improvement over the hashtable lookup was less than noise.

The conclusion is that hardware optimizations are geared toward applications, not systems like DynamoRIO, and in order to morph our operations to look like what the hardware expects we must jump through hoops that outweigh any performance gain.

4.3 Hashtable Lookup

Giving up on using an actual return instruction, we now focus on improving the performance of a hashtable lookup that uses an indirect jump.

4.3.1 Indirect Jump Branch Prediction

Indirect jumps are predicted using the IA-32 Branch Target Buffer (BTB). A first observation is that using a single, shared lookup routine will result in only one BTB entry for the lookup's indirect jump that is shared by every indirect branch in the application. If we can shift the indirect jump to each application indirect branch site, we can improve the target prediction significantly with separate BTB entries for each site. One method is to call the lookup routine, return back to the call site with a return value of the target, and then perform an indirect jump to the target. Figure 4.12 gives the performance improvement of calling the lookup routine transparently by using a separate stack as well as the impact of a non-transparent call that uses the application stack. Both are compared against a base implementation that uses a shared lookup routine. The results show that switching to a safe stack and back again ends up negating any performance improvement. Yet, using the application stack violates transparency and will not work for arbitrary programs.

Another way to bring the indirect jump to each lookup site is to inline the lookup routine itself at the site. Figure 4.13 shows the performance improvement of inlining the lookup routine into the exit stub at each indirect branch site. Inlining achieves better performance than calling the lookup routine, even when using a non-transparent call. In addition to improving branch prediction, inlining also shrinks the hit path of the lookup routine by a few instructions. The obvious

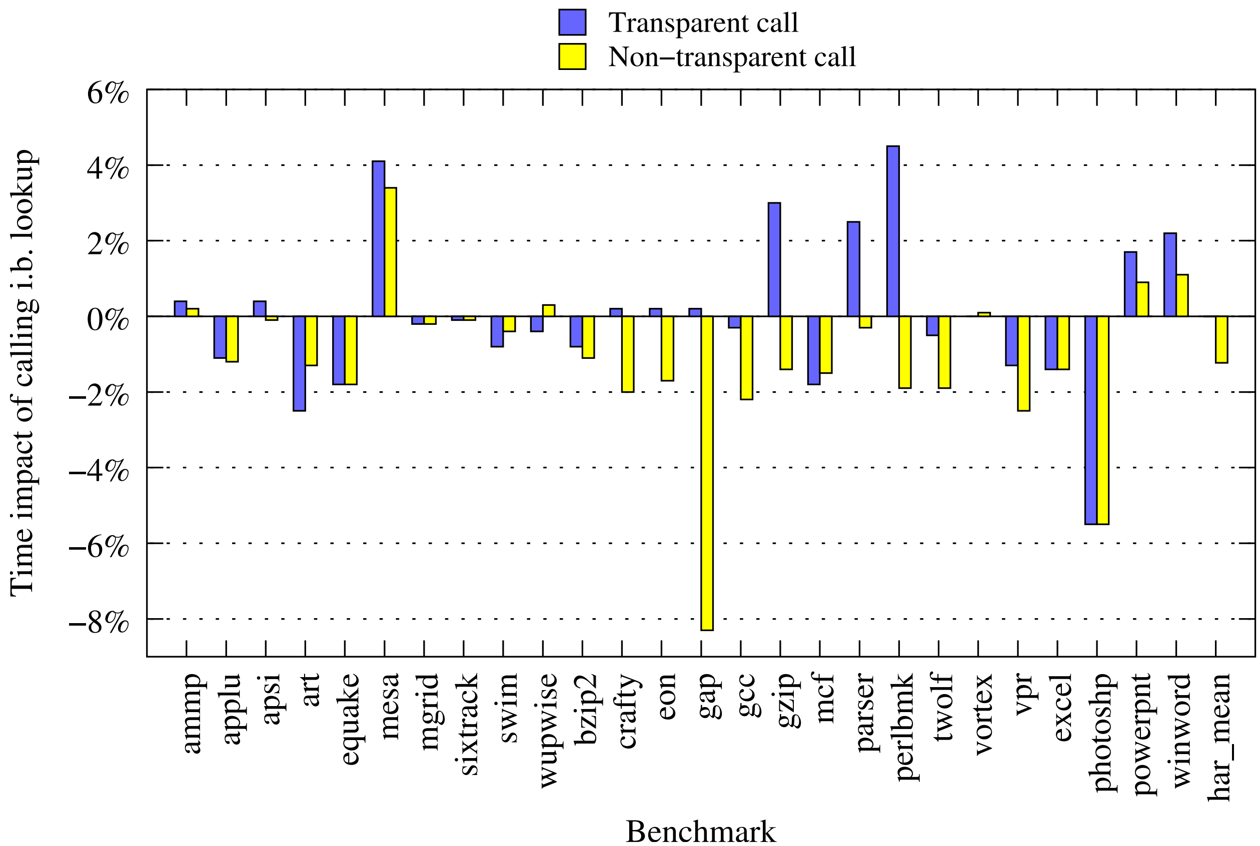

Figure 4.12: Performance impact of calling the indirect branch lookup routine, both transparently (by switching to a safe stack) and non-transparently (by using the application stack), versus jumping to the routine.

disadvantage to inlining is the space increase, which is shown in Table 4.14. Since inlining in the basic block cache takes up more room with less performance improvement (since basic blocks are not performance-critical), one optimization is to only inline the indirect branch lookup into the exit stubs of traces. The second bar of Figure 4.13 shows that the performance difference of this strategy as compared to inlining in all fragments is in the noise for most benchmarks, and a little more than noise for only a few, most likely due to coincidental improved alignment. DynamoRIO inlines only the lookup of the first member of the hashtable collision chain at each return site. A miss there moves on to a shared lookup routine that continues the collision chain walk. Figure 4.15 shows statistics on miss rates at each step of this process. See Table 7.4 for how prevalent indirect branches are for each benchmark, as well as a breakdown of the types of indirect branch

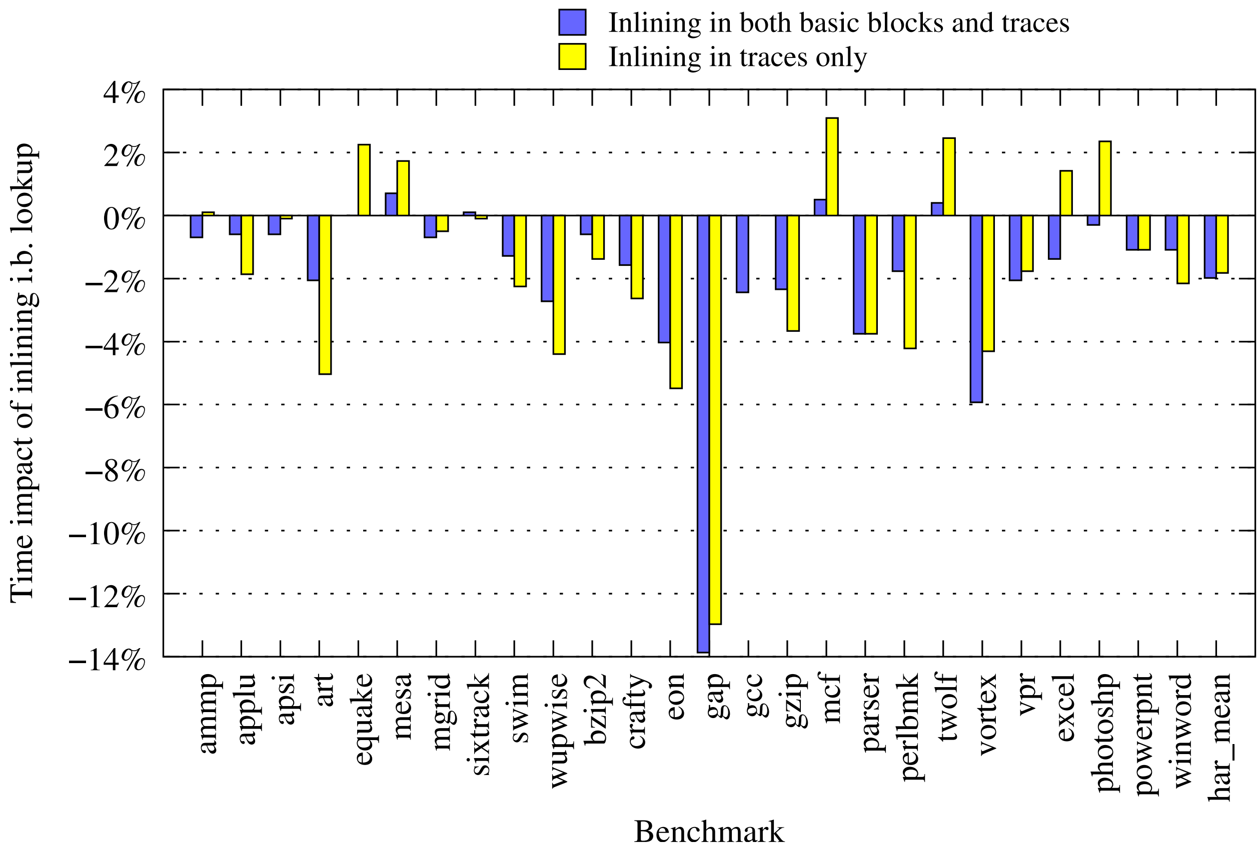

Figure 4.13: Performance impact of inlining the indirect branch lookup routine, in both traces and basic blocks, and in traces only; the two have comparable performance.

instructions.

Instead of a hashtable lookup with an indirect jump, a series of compares and direct jumps can achieve better performance in some cases. We implemented this as a code transformation using our client interface. This optimization is described in Section 9.2.3 and evaluated quantitatively in Section 9.2.5.

4.3.2 Lookup Routine Optimization

The indirect branch lookup routine must be as hand-optimized as possible: scratch registers should be kept to a minimum to reduce spill and restore code, and corner cases should be dealt with in ways that do not require conditional logic in the critical path. For example, the hashtable should contain only potential targets; having the lookup routine check a flag to see if a target is valid is

Benchmark	Basic block cache	Trace cache
ammp	10.5%	9.0%
applu	10.1%	4.3%
apsi	9.6%	9.9%
art	12.3%	10.7%
equake	12.0%	11.8%
mesa	10.4%	11.8%
mgrid	14.7%	11.6%
sixtrack	6.6%	10.0%
swim	14.0%	12.9%
wupwise	15.4%	10.5%
bzip2	8.8%	5.1%
crafty	7.9%	5.9%
eon	13.3%	16.1%
gap	10.7%	14.2%
gcc	7.3%	8.2%
gzip	12.1%	6.2%
mcf	13.9%	14.3%
parser	7.5%	6.5%
perlbmk	8.1%	9.3%
twolf	9.4%	12.5%
vortex	6.0%	7.5%
vpr	13.8%	13.0%
excel	18.6%	19.6%
photoshp	15.2%	21.1%
powerpnt	20.6%	25.8%
winword	16.8%	19.9%
average	11.8%	11.8%

Table 4.14: Cache size increase from inlining the indirect branch lookup routine.

expensive. Also, rather than having a NULL pointer indicate an empty slot or the end of a collision chain, a pointer to a special "NULL" data structure should be used, whose tag will always fail the comparison that must be done to see if the hash matches. This avoids having to both check for empty and for a match. Other optimizations involve efficiently dealing with `eflags` (discussed in Section 4.4) and shifting cleanup code to the target. By placing a prefix that restores a scratch

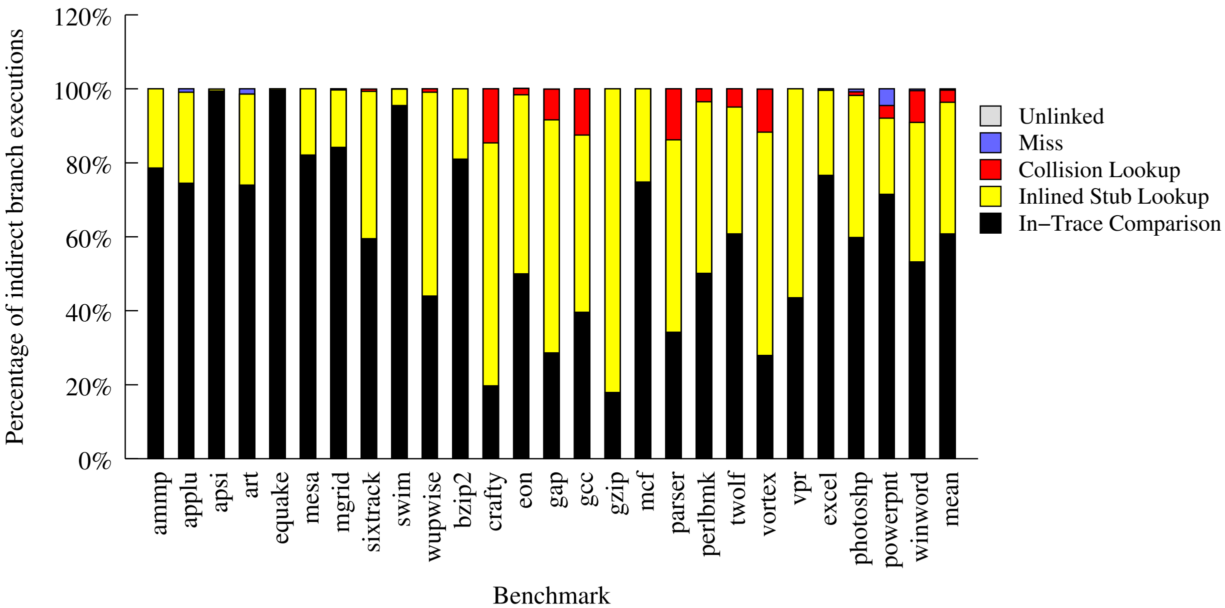

Figure 4.15: Hashtable hit statistics. Each indirect branch is placed into one of the five categories shown. The first category is the fraction of indirect branches that end up staying on a trace. Those that exit a trace require a hashtable lookup. If the inlined lookup in the exit stub hits (meaning the target is first in its collision chain), it falls into the second category. A miss there moves to the full lookup routine that can handle collision chains. The third category is for a hit in this collision lookup. The fourth is for all misses that end up context switching back to DynamoRIO. A final possibility is that this trace has been unlinked, in which case no in-cache lookup will be performed.

register on every fragment in the code cache, the hashtable hit path can be reduced by not requiring a store into memory of the hit address for the final transfer of control. (This is good not only for performance but for security as well, where control transfer targets should not be kept in writable memory — see Section 9.4.5.) The prefix can be optimized for its fragment; if the scratch register is dead in that fragment, the restore can be skipped. The restore of `eflags` can also be moved to the prefix, where parts of it can be skipped based on which flags are dead in the fragment (see Figure 4.22 for an example and Figure 4.24 for the resulting performance). This is all at the expense of extra cache space for these prefixes, so DynamoRIO only uses them for traces.

Our hash function is a simple mask that takes the low-order bits of the address in question. Since instructions are not aligned on IA-32, no shift is desired, just a bit mask. This can be applied with a single IA-32 `and` instruction. The only disadvantage is that `and` modifies the condition codes — see Section 4.4 for a discussion of hash function implementations that do not affect the condition codes.

Hashtable load factors should also be watched, as a table that is too small can contribute to performance degradation. Like all of our data structures, hashtables must be dynamically resized on demand in order to work well with applications of all sizes. DynamoRIO's hashtables start out small (256 or 512 entries), and a check is performed on each addition to see if the target load factor has been surpassed. If it has, the hashtable is doubled in size.

4.3.3 Data Cache Pressure

Other aspects of the indirect branch lookup routine besides branch prediction are critical, such as the amount of memory traffic required to perform lookups. We found that the type of hashtable used makes a big difference: using chaining to resolve collisions is inefficient in terms of the memory hierarchy, while an open-address hashtable with linear probing [Cormen et al. 1990] reduces cache accesses to a minimum and provides a significant performance boost for our indirect-branch-critical benchmarks, as shown in Figure 4.16. Hashtable data cache impact is greater than branch misprediction impact for some benchmarks (Section 7.3.2).

This difference in hashtable types is a good example of a discrepancy between theory and practice. The theoretical limits of hashing have already been reached with static [Fredman et al. 1984] and dynamic [Dietzfelbinger et al. 1994] perfect hashing. However, perfect hashing has a large space constant, and is oblivious to the memory hierarchy of modern computers. The performance of traversing a linked list with elements scattered through memory can be orders of magnitude worse than traversing the sequential storage of an array. This is why open-address hashing, though considered a poor theoretical choice, works well in practice. Other systems have also recognized this [Small 1997].

Open-address hashing also avoids the need for a next pointer, saving memory. With traditional chaining, some memory can be saved by storing the next pointer in the fragment data structure itself and not using a container structure, but that only works well when a fragment cannot be in multiple hashtables simultaneously, which is violated for security applications (Section 9.4.2) and for separate indirect branch lookups for traces and basic blocks (Section 2.3.2).

Our open-address implementation uses a few tricks for speed. Instead of having the lookup

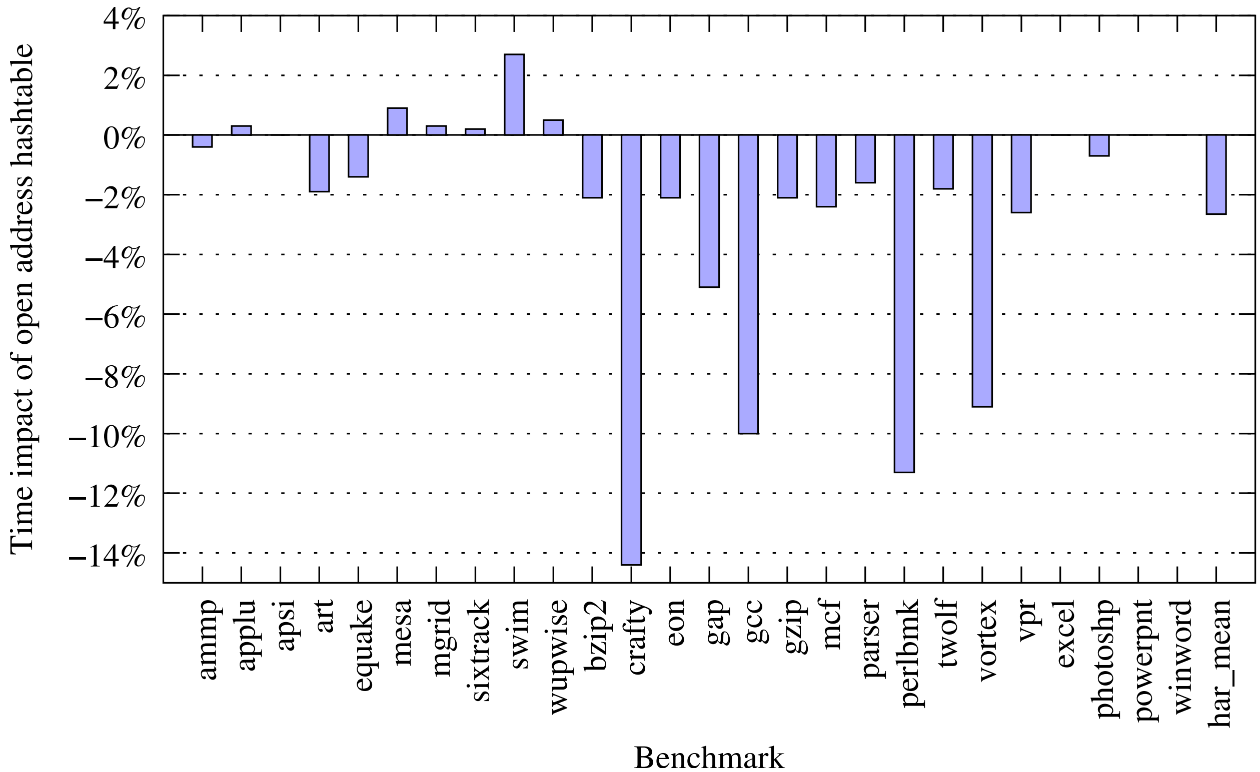

Figure 4.16: Performance impact of using an open-address hashtable for indirect branch lookups, versus a hashtable with a linked list collision chain.

routine perform a `mod` operation or check for wrap-around, we use a sentinel at the end of the table that is identical to an empty slot. We end up with a few misses that are really hits, but we find that out once we are back in DynamoRIO code, and those few misses are outweighed by the increased performance on the rest of the lookups. We also avoid using deletion markers. Instead we use Algorithm R from Knuth [1998] Section 6.4, in which elements are shifted so that there is no gap on a deletion. This adds complexity on deletions, but makes the performance-critical lookups more efficient.

4.4 Condition Codes

The IA-32 condition codes are kept in the `eflags` register [Intel Corporation 2001, vol. 1]. Many instructions implicitly modify `eflags`, but the register takes many machine cycles to save and restore. This makes flags preservation a significant barrier to optimization and code manipulation on general on IA-32. For example, nearly all arithmetic operations modify `eflags`, making a transparent counter increment very difficult to do efficiently. Performing a test and a conditional branch is similarly difficult, so inserting the compare to detect whether an indirect branch target equals the inlined target in a trace (Section 2.3.2) requires saving and restoring `eflags` if done in a straightforward manner. Fortunately, there are some ways to compare values without modifying `eflags`, by using the load-effective-address instruction `lea` for arithmetic and the special-purpose conditional branch `jecxz` [Intel Corporation 2001, vol. 2]. These types of tricks are required for efficient IA-32 code manipulation. They have their limits, however, as `lea` can only perform certain addition and multiplication operations, and `jecxz` requires use of the `ecx` register and has an eight-bit target distance restriction.

This section performs a quantitative analysis of `eflags` handling options. The `pushf` instruction is the only single-instruction method of saving all of the flags modified by a `cmp` comparison instruction. `pushf`'s counterpart for restoring the flags, `popf`, is very expensive, as it restores the entire `eflags` register, which contains many control codes whose modification requires flushing the pipeline. `pushf` and `popf` also require a valid stack, something that is not guaranteed at arbitrary points in application code (Section 3.2.4). Figure 4.17 shows the performance impact of using a stack swap to a safe stack and a `pushf` to save the flags, and a corresponding `popf` and application stack restoration to restore the flags, around each trace indirect branch comparison and indirect branch lookup. In a trace, the flags are saved, the comparison is done, and on a hit, the flags are restored and execution continues on the trace. On a miss, control goes to the indirect branch lookup routine, which assumes the flags have already been saved, does the lookup, and restores the flags afterward on both the hit and miss paths. As the average slowdown approaches twenty percent, `pushf` is clearly too expensive.

The only other multiple-flag-preservation instructions are `lahf` and `sahf`, which save and restore only five of the six flags that `cmp` modifies (the *arithmetic flags*) and require the `eax` register. The unsaved flag is the overflow flag, which can be preserved on its own. We can set a byte in memory to the value of this flag with the `seto` instruction, and we can restore that value by performing an add of `0x7F000000` and the word whose most significant byte was set by `seto` (to either zero or one, depending on the overflow flag value). By doing the add followed by a `sahf`

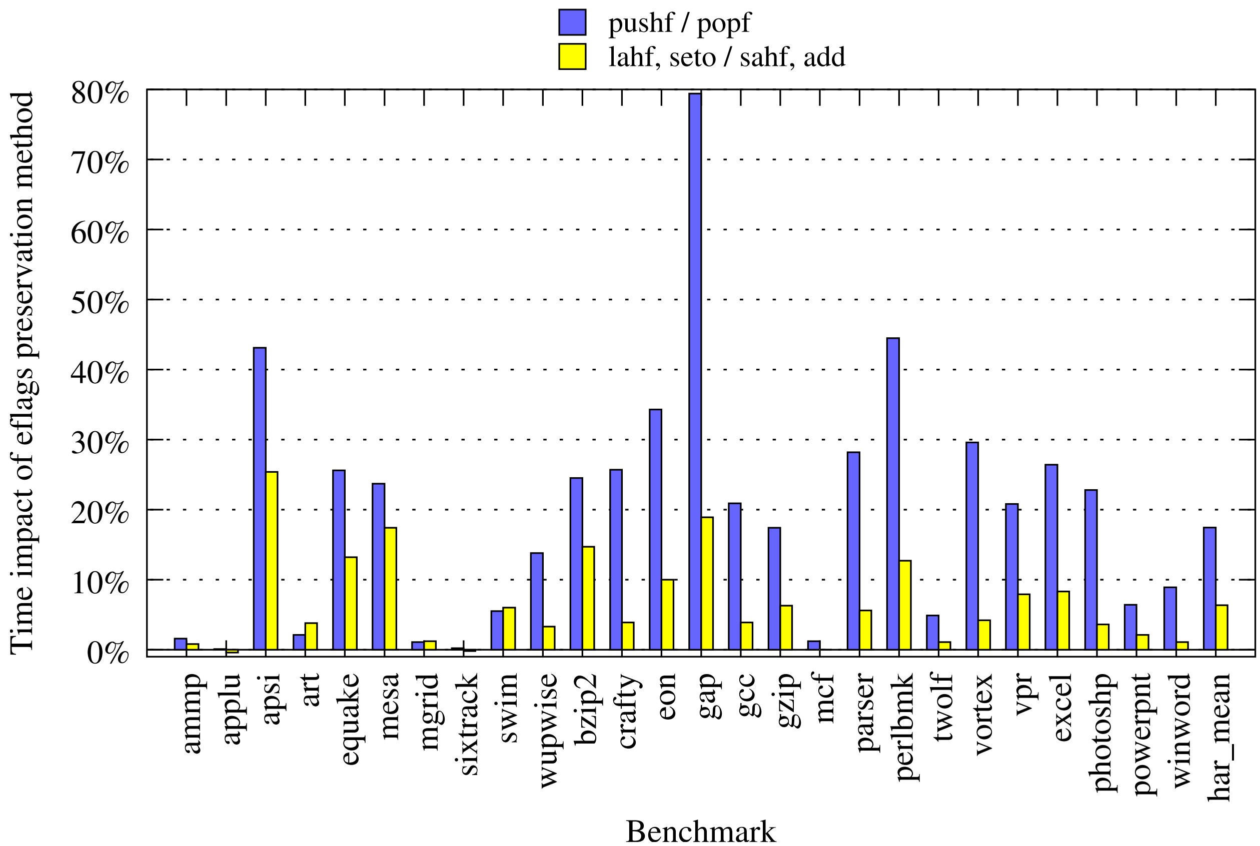

Figure 4.17: Performance impact of using a full eflags save (`pushf`) and restore (`popf`) for in-trace indirect branches and the indirect branch lookup routine. A more efficient save (`lahf` and `seto`) and restore (`sahf` and `add`) can reduce performance considerably, though DynamoRIO's scheme (`lea` and `jecxz`) beats them both (it is the base being compared against in this graph).

we can properly restore all six flags. Unfortunately, when `lahf` is followed by a read of `eax` it causes a partial register stall, costing up to ten cycles. Specific patterns can be used to eliminate the partial register stall, such as using `xor` or `sub` to set `eax` to zero prior to the `lahf`, but all of those patterns modify the flags! Our solution is to keep the flags in `eax` until the restoration point, meaning that an extra scratch register is needed.

The performance impact of using these instruction sequences is the second bar in Figure 4.17. The `lahf` and `seto` combination is much better than `pushf`, with an average slowdown of under ten percent. However, we can do better by using the afore-mentioned `lea` and `jecxz` instructions to do the in-trace comparison without modifying the flags at all. Common comparison patterns replaced with these two instructions are shown in Figure 4.18, and an example of a transformed

```
instead of:
      cmp <register>, <constant>
      je  match
avoid eflags modification via:
      lea -<constant>(<register>), %ecx
      jecxz match

instead of:
      cmp <register>, <constant>
      jne  nomatch
avoid eflags modification via:
      lea -<constant>(<register>), %ecx
      jecxz match
      jmp nomatch
    match:

instead of:
      cmp <register1>, <register2>
      je match
avoid eflags modification via:
      # use "not register1 + 1" for -register1
      not <register1>
      lea 1(<register1>, <register2>, 1), %ecx
      jecxz match
```

Figure 4.18: Instruction sequences for comparisons that do not modify the flags. In order to target addresses more than 128 bytes away, a landing pad must be used, as `jecxz` only takes an eight-bit signed offset. Comparing two registers requires some trickery to do an addition of non-constants without modifying the flags.

return instruction inside of a trace is given in Figure 4.19. DynamoRIO uses this scheme, and its performance is the base for comparison in Figure 4.17.

We can avoid modifying `eflags` for a simple comparison, but the hashtable lookup routine is more complicated as it must perform a number of arithmetic operations, depending on the hash function. As mentioned in Section 4.3.2, our hash function is very simple, a mask that pulls out the low-order bits. However, this is quite difficult to do in the general case without modifying the flags. Figure 4.20 shows some flag-avoiding instruction sequences we came up with for bit masks of various sizes. We could not come up with a general routine that could be efficiently parametrized by the mask size, and ended up hardcoding different routines for different sizes. Figure 4.21 shows

```
original:
    0x08069b53   ret
target of return:
    0x0806a18d   mov    %eax, 0xfffffffc(%ebp)

inside trace:
    0x4c3f592e   mov    %ecx, ecx-slot
    0x4c3f5934   pop    %ecx
    0x4c3f5935   lea    0xf7f95e73(%ecx), %ecx
    0x4c3f593b   jecxz  $0x4c3f5948
    0x4c3f593d   lea    0x806a18d(%ecx), %ecx
    0x4c3f5943   jmp    $0x4c3f5b02 <exit stub 5>
    0x4c3f5948   mov    ecx-slot, %ecx
    0x4c3f594e   mov    %eax, 0xfffffffc(%ebp)
```

Figure 4.19: An example indirect branch (in this case a return) inlined into a trace. A register must be spilled to provide space to hold the return target. The target is then compared to the value that will keep it on the trace (`0x0806a18d`), using the `lea` and `jecxz` combination from Figure 4.18.

the performance of a hardcoded 15-bit mask sequence, where the hashtable routine was statically fixed at 15 bits, versus our base implementation (which uses `lahf` and `seto`), also fixed at 15 bits for a better comparison. The flag-avoiding lookup routine does not perform favorably enough for us; combined with its difficulty in generalizing, we abandoned it and stuck with our use of `and`.

The prefix used to optimize the indirect branch lookup routine (Section 4.3.2) can be used to shift the `eflags` restoration from the lookup routine to the target, allowing flag restoration to be skipped for fragments that write to the flags (Figure 4.22). The frequency of such frequency is shown in Table 4.23: only seven percent of fragments, on average, need an `eflags` restoration. Figure 4.24 shows the resulting performance gain, which is significant, nearly twenty percent for `gap`. This gain is certainly worthwhile for traces, but for less performance-critical basic blocks it might be best to keep the `eflags` restore in the indirect branch lookup to save space.

In conclusion, we were able to limit `eflags` saves and restores to the indirect branch lookup routine and keep them out of our in-trace comparisons. However, we still see a performance impact in our indirect branch lookup routine, even though we are using the quickest available method (`lahf` and `seto`) to preserve the flags. Figure 4.25 shows that the performance improvement if we violate transparency and simply assume we do not need to preserve eflags across indirect branches exceeds five percent on several integer benchmarks. The `gcc` compiler never produces code that

```
14-bit:
  and $0x00003fff %edx -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx
becomes
  lea (,%edx,4) -> %edx
  movzx %dx -> %edx
  mov 0x401f7e64(,%edx,1) -> %edx

15-bit:
  and $0x00007fff %edx -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx
becomes
  lea (,%edx,2) -> %edx
  movzx %dx -> %edx
  mov 0x401f7e64(,%edx,2) -> %edx

16-bit:
  and $0x0000ffff %edx -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx
becomes
  movzx %dx -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx

17-bit:
  and $0x0001ffff %edx -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx
becomes
  movzx %dx -> %ebx
  bswap
  lea (,%edx,8) -> %edx
  lea (,%edx,8) -> %edx
  lea (,%edx,2) -> %edx
  movb $0 -> %dl
  movzx %dx -> %ebx
  lea (%ebx,%edx,2) -> %edx
  mov 0x401f7e64(,%edx,4) -> %edx
```

Figure 4.20: Bitwise `and` instruction sequences that do not modify the flags, for sizes 14 through 17. An efficient general routine is difficult to manufacture.

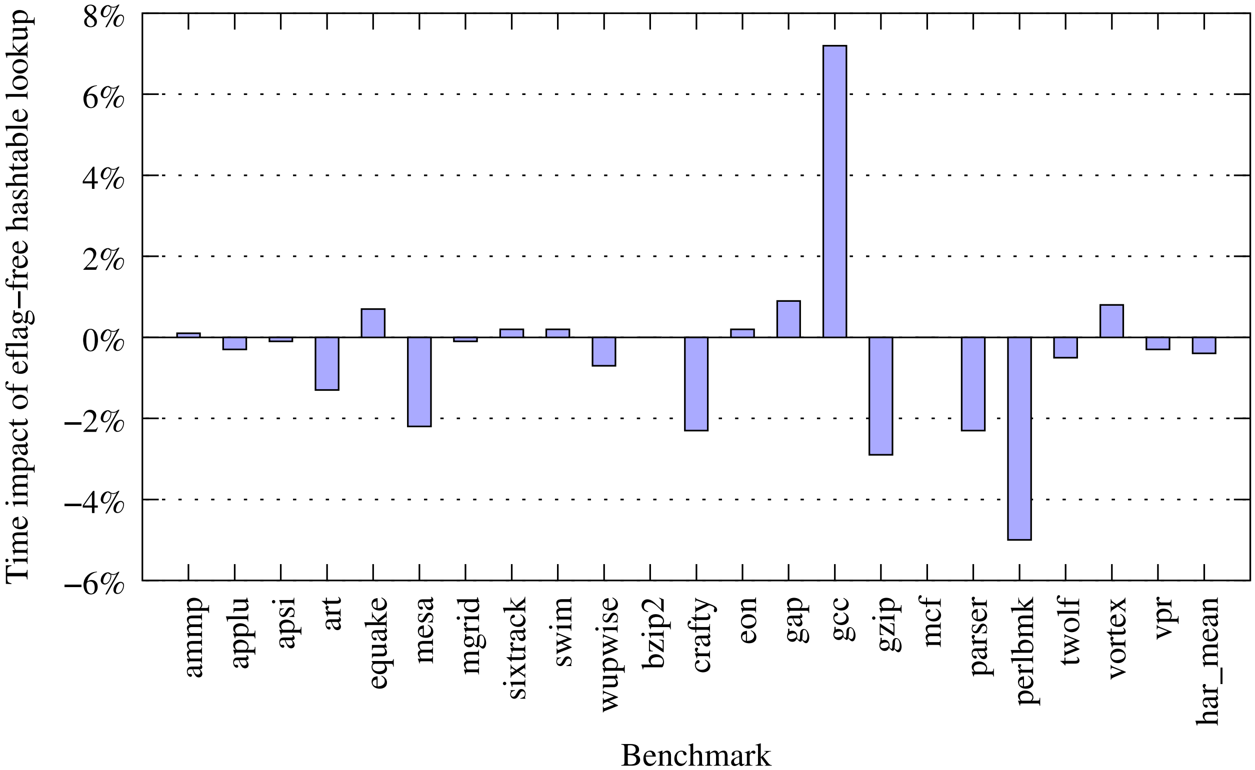

Figure 4.21: Performance impact of an `eflags`-free hashtable lookup routine, versus DynamoRIO's `lea` and `jecxz` scheme.

```
indirect branch target entry:
    if fragment doesn't write overflow flag:
        mov    eflags-overflow-slot, %ecx
        add    $0x7f000000, %ecx
    if fragment doesn't write other 5 arith flags:
        sahf
    if fragment doesn't write eax:
        mov    eax-slot, %eax
    if fragment doesn't write ecx:
        mov    ecx-slot, %ecx
main entry:
        ...
```

Figure 4.22: Our implementation of a prefix that restores both arithmetic flags and scratch registers only if necessary.

Benchmark	Prefixes requiring `eflags` restoration	Prefix executions
ammp	6.9%	0.2%
applu	7.5%	0.0%
apsi	8.6%	1.0%
art	3.8%	0.0%
equake	5.1%	25.3%
mesa	11.4%	22.9%
mgrid	9.9%	0.0%
sixtrack	5.9%	0.1%
swim	9.2%	0.0%
wupwise	9.2%	5.9%
bzip2	4.9%	0.4%
crafty	4.1%	7.8%
eon	11.0%	10.0%
gap	11.9%	5.8%
gcc	9.1%	6.0%
gzip	5.5%	5.8%
mcf	7.1%	0.3%
parser	6.5%	5.3%
perlbmk	8.7%	3.6%
twolf	5.2%	5.2%
vortex	2.2%	1.0%
vpr	6.7%	4.3%
excel	16.6%	19.1%
photoshp	5.6%	0.9%
powerpnt	16.7%	27.9%
winword	15.4%	10.2%
average	8.3%	6.5%

Table 4.23: Frequency of trace prefixes that require restoration of `eflags`. The first column is the static percentage of all trace prefixes, while the second is the dynamic execution percentage.

depends on eflags settings across an indirect branch. The Microsoft C++ compiler, however, does produce such code, so we are not able to measure the impact on Windows programs, and as it breaks those programs we cannot make this assumption in general.

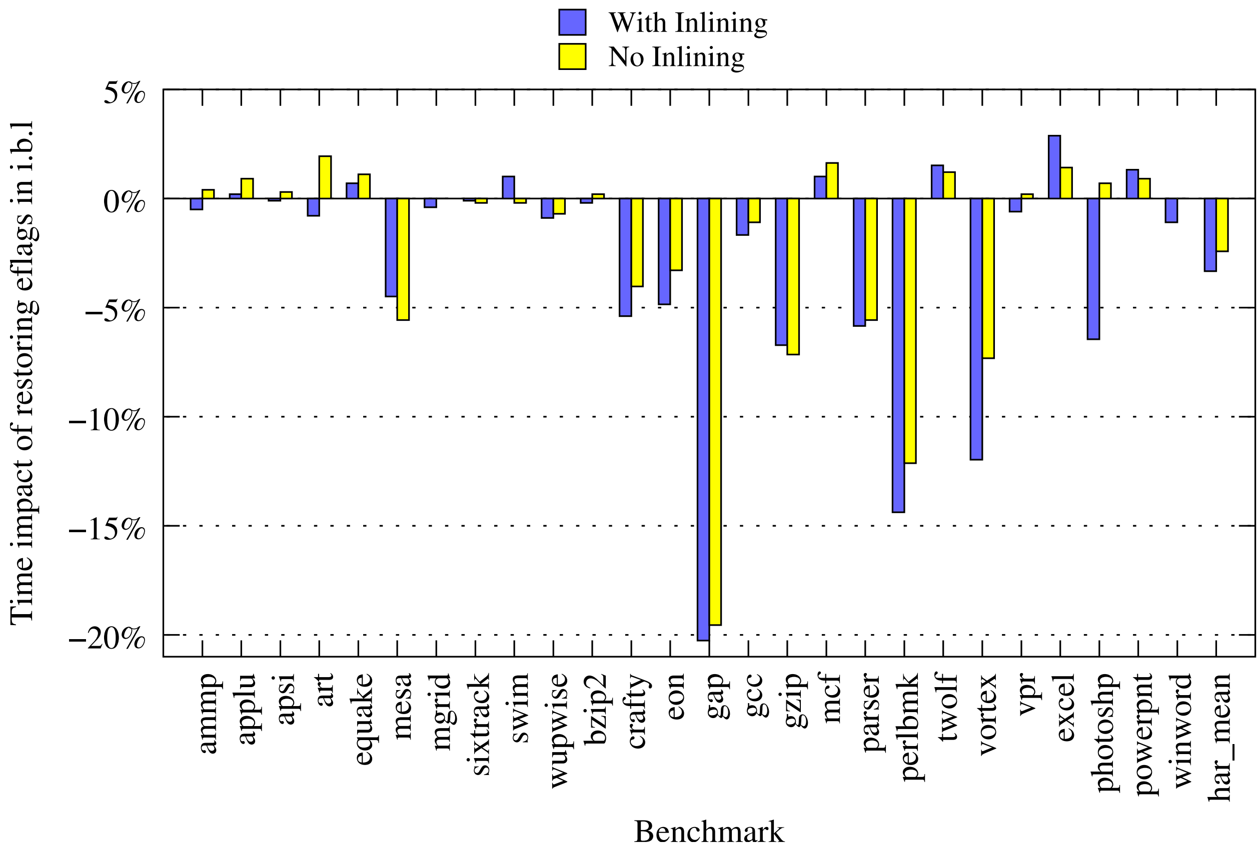

Figure 4.24: Performance impact of shifting the `eflags` restore from the indirect branch lookup routine to the fragment prefixes, where it can be omitted for fragments that write to `eflags` before they read it.

4.5 Instruction Cache Consistency

Another troublesome aspect of the IA-32 architecture is that the instruction cache is kept consistent with the data cache in hardware. Software that modifies or generates code dynamically does not need to issue an explicit flush of the instruction cache in order to ensure that the correct code generated as data makes its way into the processor. This makes it very difficult to detect modifications to code. On architectures that require an explicit flush [Keppel 1991], a code manipulation system knows when to update its code cache by watching for flush instructions. On IA-32, however, extreme measures must be employed to prevent stale instructions from persisting in the code cache once their original sources are dynamically modified. This is discussed in detail in Section 6.2.

Instruction modification is expensive on modern processors with deep pipelines, and IA-32 is

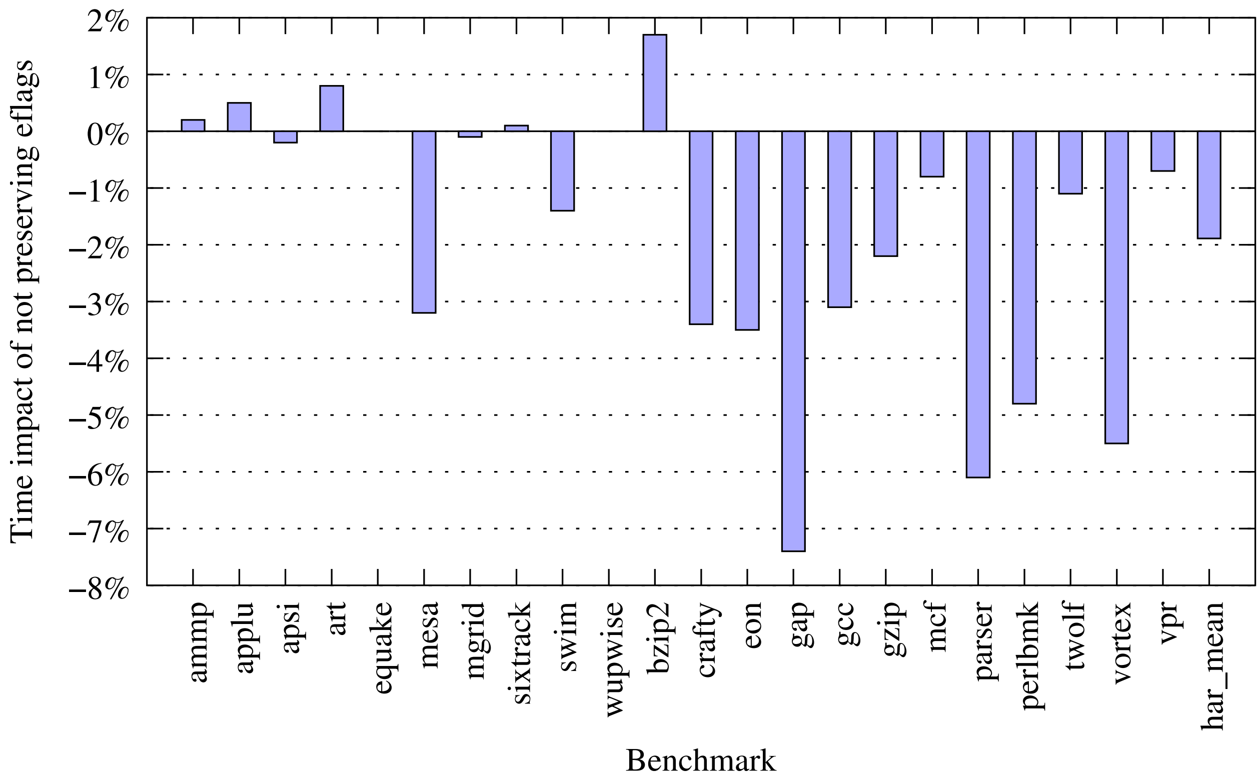

Figure 4.25: Performance impact of not preserving `eflags` across indirect branches on our Linux benchmarks (this violates transparency but illustrates the cost of flag preservation).

no exception. Self-modifying code incurs significant performance penalties, as a write to code invalidates the entire prefetch queue (or the trace cache on the Pentium 4) [Intel Corporation 2001, vol. 3]. This is another case where DynamoRIO's behavior looks different from a typical application's, for which the process has been optimized. DynamoRIO must modify code for linking and unlinking and for cache capacity (replacing one fragment with another in the cache). Fortunately, these costs are usually amortized by spending the majority of the time inside traces and not inside DynamoRIO code.

4.5.1 Proactive Linking

Since every link or unlink results in an immediate instruction cache flush, performing all linking as early as possible and batching it together into bursts was found to be the best policy. This *proactive linking* is in contrast to *lazy linking*, where linking is only performed when a transition

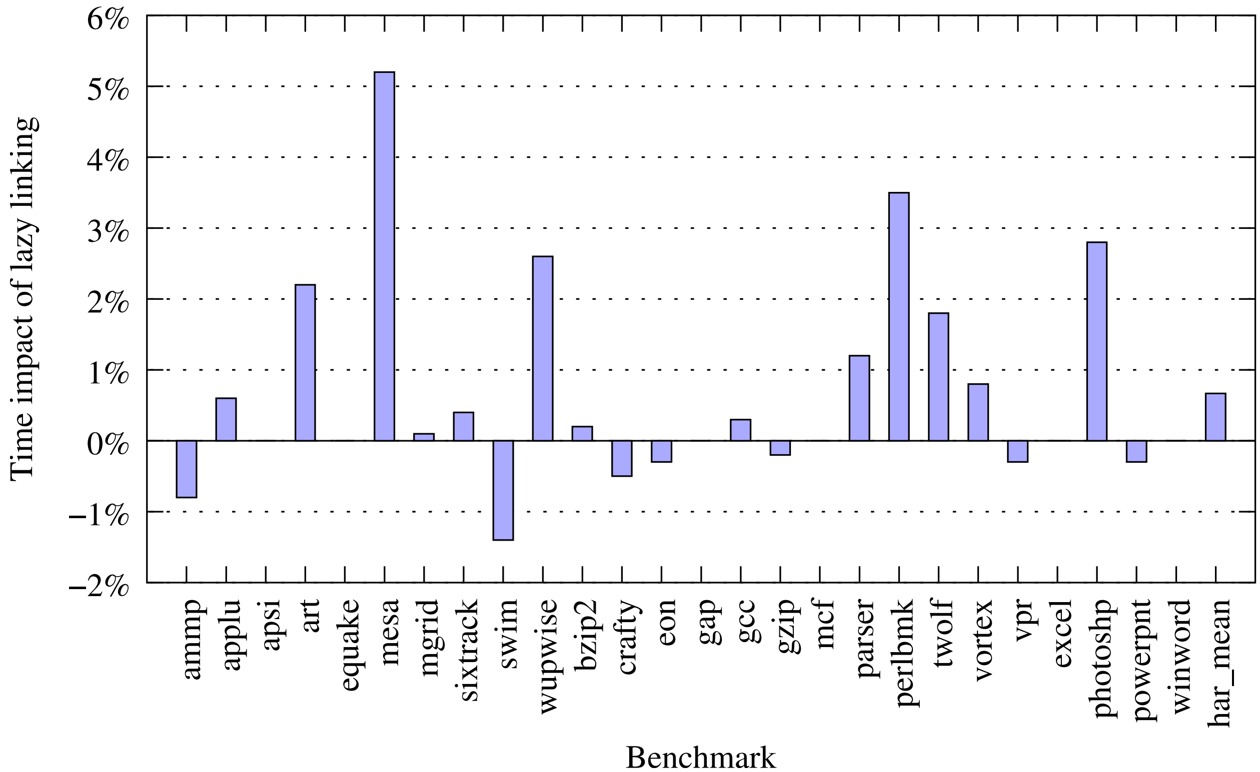

Figure 4.26: Performance impact of lazy linking, as opposed to proactive linking.

is traversed (see also Section 2.2). Although lazy linking performs fewer links overall, its spread-out instruction cache flushing ends up disrupting execution more often than proactive linking, outweighing the smaller amount of work. Figure 4.26 shows that lazy linking does not improve performance beyond the noise, and in fact slows down a few benchmarks. Furthermore, the added data structures for tracking incoming links for proactive linking are integral to many other critical operations, including single-fragment deletion (see Section 2.2).

4.6 Hardware Trace Cache

The Pentium 4 processor contains a *hardware trace cache*. Such caches allow dynamic optimization of the instruction stream off of the critical path [Rotenberg et al. 1996]. The Pentium's cache builds traces out of IA-32 micro-operations (each IA-32 complex instruction is broken down into micro-operations [Intel Corporation 1999]). Despite working at sub-instruction levels, this hardware trace cache competes with our software trace cache, since both achieve performance gains from code layout optimization (removing direct branches, hot path optimization, etc.). Furthermore, the hardware trace cache is not built to expect as many code modifications as DynamoRIO performs, as discussed in Section 4.5. Section 2.3 presents quantitative evidence of the impact of the trace cache by comparing our Pentium 3 performance to our Pentium 4 performance. It is an area of future work to study how our software trace building might be modified in order to focus on complementary optimization to the hardware trace cache.

4.7 Context Switch

The context switch to and from DynamoRIO's code cache is performance-critical. We optimize it to save and restore only the general-purpose registers (since DynamoRIO does not use floating-point operations), the condition codes (the `eflags` register), and any operating system-dependent state (see Section 5.2.2). Keep in mind that the state preservation works both ways. For example, preserving the `eflags` value of the application is not sufficient. Switching back to DynamoRIO mode and using the application's `eflags` can cause erroneous behavior, since the `eflags` register stores a number of critical flags that control fundamental operations, such as the direction of string operations. DynamoRIO's `eflags` must be saved and restored as well as the application's for correct behavior.

4.8 Re-targetability

Some code caching systems have been specially designed to be re-targetable [Scott et al. 2003, Cifuentes et al. 2002, Robinson 2001]. DynamoRIO also sports a modular design, with architectural and operating system-specific elements separated from the main control logic of caching and linking. The only assumption throughout DynamoRIO is that the target hardware uses 32-bit addresses. Other than that, porting to another operating system or architecture would be just as easy in DynamoRIO as in other systems.

However, being able to run large, complex applications on other architectures or operating systems would take a great deal of work. The challenges of efficiency, transparency, and comprehensiveness have components that are very specific to the underlying platform. Some ports would be easier than others — for example, a flavor of UNIX whose signal handling is similar to that found in Linux would require less work in that area. Even so, executing modern, dynamic applications on other platforms is a world away from implementing the minimal support necessary to handle small, static, single-threaded programs.

4.9 Chapter Summary

Many significant challenges to building a successful runtime code manipulation system are specific to the underlying architecture. This chapter discussed the issues relevant to the CISC IA-32 platform. We addressed its complex instruction set with an adaptive level-of-detail instruction representation and its pervasive condition code dependences with a novel scheme to preserve only the arithmetic flags. We were not able to completely solve some problems, such as the performance discrepancy of indirect branches, which may require hardware support to avoid (see Section 11.2.5), although that may come with the indirect branch predictor in the Prescott version of the Pentium 4. In addition to architectural challenges, the underlying platform brings up many issues specific to the operating system, which are discussed in the next chapter.

Chapter 5

Operating System Challenges

This chapter discusses aspects of a runtime code manipulation system that depend on the underlying operating system. In particular, it presents the issues faced by DynamoRIO on Windows and Linux, although the Linux challenges generalize to any UNIX-like operating system. (Some of the Windows challenges have been previously described by the author [Bruening et al. 2001].) We chose these two operating systems for their popularity. DynamoRIO is a tool platform, and to be usable by the widest audience, we needed to work with existing operating systems in widespread use. The majority of computers worldwide run Windows.

DynamoRIO runs in user mode on top of the operating system, to be deployable (Section 1.1). Furthermore, DynamoRIO occupies the same address space as the application, operating inside the application process rather than using debugging interfaces or inter-process communication, which are too coarse-grained to provide comprehensive interposition. We rejected using kernel-mode components to avoid spoiling our goal of operating on commodity operating systems. Additionally, it is difficult from kernel mode to efficiently and safely act as the application would act. To the operating system kernel, an application is a black box that makes system calls, but a code manipulation system needs to interpose itself on much more fine-grained operations than just system calls. Operating in kernel mode would also make memory management more difficult. We also rejected executing underneath the operating system. A whole-system emulator like VMWare [Bugnion et al. 1997] or SimOS [Rosenblum et al. 1995], which runs an operating system and all of its processes, has a hard time seeing inside those processes (e.g., distinguishing threads) without extensive knowledge of operating system internals, a difficult task for closed systems like Windows.

Running in user mode has its disadvantages, as DynamoRIO has no control over the operating system that its target applications are running on. It must handle the multiple threads of control that the operating system provides to applications (Section 5.2); it must go to great lengths to ensure that it does not lose control when the kernel directly transfers control to the application (Section 5.3); it must monitor application requests of the operating system, in order to keep its code cache consistent and, again, monitor kernel-mediated control transfers (Section 5.4); and, it must have a mechanism to take over control of a process in the first place (Section 5.5). Beyond all of that, DynamoRIO must ensure that its operations remain transparent with respect to the application's interactions with the same operating system (Chapter 3). This chapter shows how we accomplish all of this, transparently inserting a layer between the application and the operating system. For discussion of how related systems have tackled these problems see Section 10.2.4.

5.1 Target Operating Systems

As background for the rest of this chapter, this section describes our target operating systems, Windows and Linux.

5.1.1 Windows

Microsoft Windows is defined in terms of the Application Programming Interface known as the Win32 API [Microsoft Developer Network Library, Richter 1999]. Similarly to how IEEE POSIX defines how UNIX operating systems should behave, the Win32 API defines how an application interacts with the Windows operating system. While neither POSIX nor the Win32 API specify the structure of the operating system kernel, all UNIX implementations provide POSIX support mainly through direct interaction between user mode code and the kernel in the form of system calls. Windows, however, adds an additional layer between the Win32 API and the kernel, to support multiple *environment subsystems*. Windows 2000 contains three such subsystems: Win32, POSIX, and OS/2. POSIX support is rudimentary (Windows only supports POSIX.1, to meet government procurement requirements), and OS/2 is similarly limited [Solomon and Russinovich 2000]. Essentially all modern Windows applications use the Win32 subsystem.

Applications interact with their subsystem, which then interacts with the kernel through system

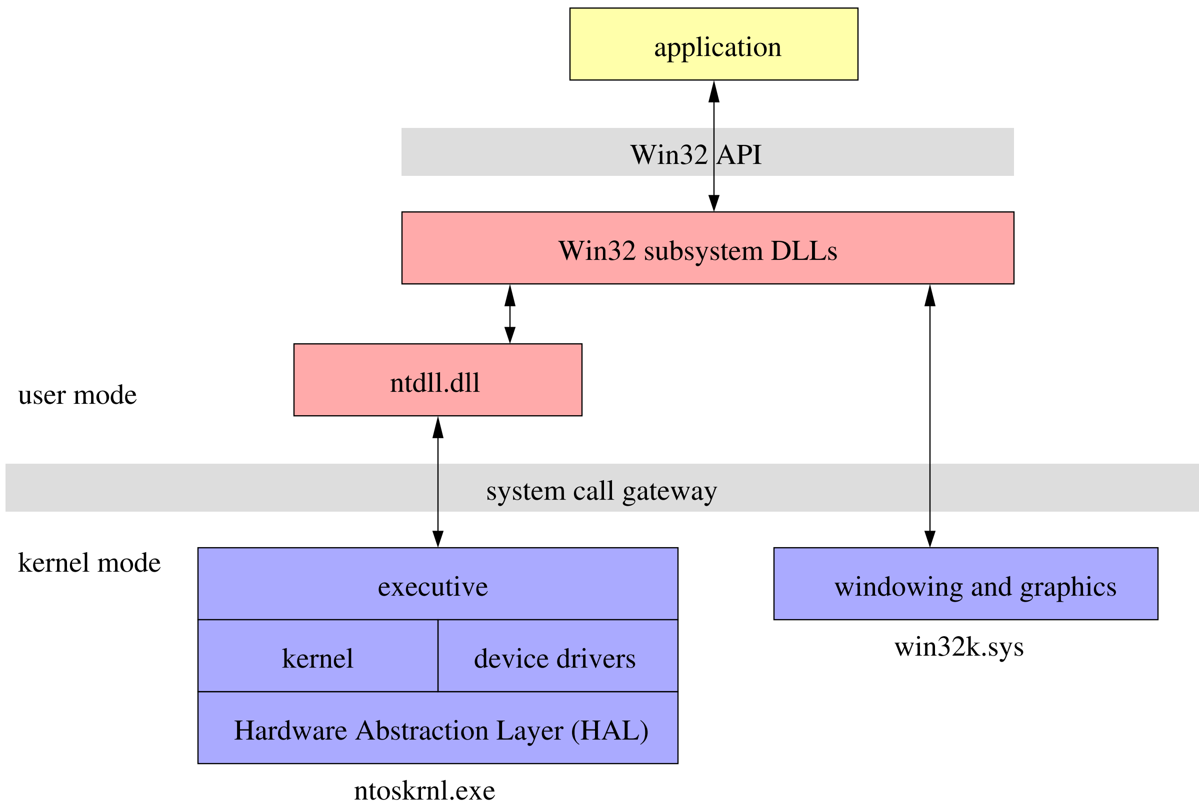

Figure 5.1: An application interacts with the Win32 subsystem through the Win32 API, which is implemented as a set of user-mode libraries. These libraries communicate with two separate components running in kernel mode: `ntoskrnl.exe` on the left, which contains the core of the kernel, and `win32k.sys`, which contains windowing and graphics support that was moved into the kernel for performance reasons. System calls that are processed by `ntoskrnl.exe` are all routed through the user-mode library `ntdll.dll`, while the Win32 subsystem also makes system calls directly to its kernel-mode component, `win32k.sys`.

calls, as shown in Figure 5.1. The system call interface is undocumented; only the subsystem interface is documented. For the Win32 subsystem, the Win32 API is its interface. The API is implemented by a number of user-mode libraries (such as `kernel32.dll`, `user32.dll`, `gdi32.dll`, and `advapi32.dll`) and a kernel-mode component called `win32k.sys`, which was moved into the kernel for performance reasons. The rest of the kernel is contained in `ntoskrnl.exe`. System calls bound for it are all routed through the user-mode library `ntdll.dll`, which exports the system calls and some higher-level routines as the Native API [Nebbett 2000]. This library is used both by the Win32 subsystem and by other subsystems and critical system processes, which operate independently of any subsystem. In order to support these other processes, and for even more important transparency reasons (Section 3.1.1) and interface complexity reasons (Section 5.4), we operate at the system call interface and not at the Win32 API level.

Because the underlying Win32 API implementations are so different in aspects critical to a runtime code manipulation system, our support for Windows versions depends on the kernel. DynamoRIO targets the Microsoft Windows NT family of operating systems, which includes Windows NT, Windows 2000, Windows XP, and Windows 2003. We do not support Windows 95, Windows 98, or Windows ME, which have different mechanisms of kernel control flow that we have not studied.

Although there has been research targeting the Windows operating system, researchers often shy away from working with Windows, because it is proprietary and the lack of information about its internals makes it hard to deal with at a low level. As mentioned above, we chose to target Windows due to its popularity. Windows is currently running on far more computers than any UNIX variant.

We obtained much of our information about Windows from both official sources [Solomon and Russinovich 2000] and unofficial sources [Nebbett 2000, Pietrek 1997, Pietrek 1996]; the latter used techniques such as reverse engineering to obtain their information. These references were not complete enough, however, and we had to resort to experimentation ourselves in many cases to figure out what we needed to know.

5.1.2 Linux

Linux is a UNIX-style operating system, initially developed by Linus Torvalds in 1991 for the IA-32 architecture. In this thesis we focus on the 2.4 version of the Linux kernel [Bovet and Cesati 2002] and the contemporary standard POSIX threads library for Linux, LinuxThreads [Leroy]. The relevant features of this platform for DynamoRIO are its IEEE POSIX compliance with respect to threads and signals, which we will discuss throughout this chapter.

Since Linux was developed under the GNU General Public License [Free Software Foundation], the source code is freely available. This makes understanding the details necessary for a runtime code manipulation system much easier. DynamoRIO needs to duplicate the kernel's actions for passing signals on to the application (see Section 5.3.5), and understanding the kernel's exact signal behavior is a must.

Paradoxically, Linux is more of a moving target and has been harder to support across versions than the proprietary, commercial Microsoft Windows. This is both because the Linux kernel is still under development in key areas relevant to DynamoRIO, such as providing full POSIX support for signals and threads, and because its release process is not as focused on backward compatibility.

5.2 Threads

A key feature of any modern operating system is its support for threads. The presence of threads has far-reaching consequences for DynamoRIO, which must be built from the ground up to handle multiple threads of execution sharing a single address space. Fortunately, user-mode thread libraries, such as Windows *fibers* [Richter 1999], do not present any of the problems that real threads do to a runtime system. User-mode threads can essentially be ignored, as all of their operations are explicit user-mode actions, unlike kernel threads, whose context switches are not observable by DynamoRIO.

The first problem with threads is deciding how application threads correlate with DynamoRIO threads. Transparency issues (Section 3.2.1) require that DynamoRIO create no new threads, so each application thread is also a DynamoRIO thread, with a context switch to save and restore the application state as it exits and enters the code cache, and there are no DynamoRIO-only threads.

Cache and data structure management in the presence of multiple threads is discussed extensively in Chapter 6. Several other issues must be confronted when supporting multiple threads, which we discuss below: how to provide thread-private scratch space, how to preserve thread-local state, and how to cope with synchronization among threads.

5.2.1 Scratch Space

Scratch space is essential for code manipulation. Transparent scratch space needs to be accessible at arbitrary points in the middle of application code without disturbing the application state. It is needed for performing any transparent operation inside of application code, such as an increment of an instrumentation counter, a compare of an indirect branch target to see if control should stay on a trace, an indirect branch hashtable lookup, or a full context switch from application code to DynamoRIO code. Scratch space is used mainly to spill registers. The application register value is copied to the scratch space, freeing up the register for use by DynamoRIO; once finished, the application value is copied back to the register.

When accessing scratch space, there can be no bootstrapping. For example, a function call cannot be used, since the process of making that call and handling its results requires its own scratch space. Scratch space access needs to require no resources that cannot be set up statically by instruction insertion prior to placing code in the code cache, and it needs to be as efficient as possible, since this is direct overhead being added on top of the application code itself.

Scratch space possibilities depend on the underlying instruction set. If absolute addressing is possible (such as on variable-length CISC architectures like IA-32, where a full 32-bit instruction immediate is allowed), accessing scratch space becomes a simple memory access of an absolute address. On architectures where this addressing mode is not available, the typical trick is to steal a register to point to the base of the scratch space. The actual application value of the register is then kept in a slot in the scratch space itself. This solution was employed in Dynamo [Bala et al. 2000].

The presence of threads complicates this picture. Scratch space now needs to be thread-private, which creates problems for absolute addresses. Since neither Windows nor Linux provides user-mode hooks on thread context switches, we cannot use tricks like shifting per-thread structures in and out of a single piece of global memory. Thus, an absolute address can only be used if multiple threads cannot execute in the same code at once. Registers, however, are thread-private, and the operating system saves and restores the stolen register appropriately on each thread context switch. Unfortunately, on an architecture with few registers like IA-32, stealing a register incurs a performance penalty that is much steeper than on register-rich platforms. We measured the performance of stealing a register, choosing `edi` as it is not used for any special purposes (many IA-32 registers have special meanings to certain instructions) except by the string instructions. The overhead is high, with more than a 20% slowdown on several benchmarks, as Figure 5.2 shows for Linux and Figure 5.3 shows for Windows. The difference between the figures highlights the dependence of register stealing's effects on the compiler. The Windows numbers, especially, show an interesting phenomenon: for unoptimized code, stealing a register does not have nearly the effect it does on optimized code. This makes sense, since optimized code more fully utilizes all available registers. Although our register stealing implementation could have been further optimized, we rejected register stealing as an option on IA-32.

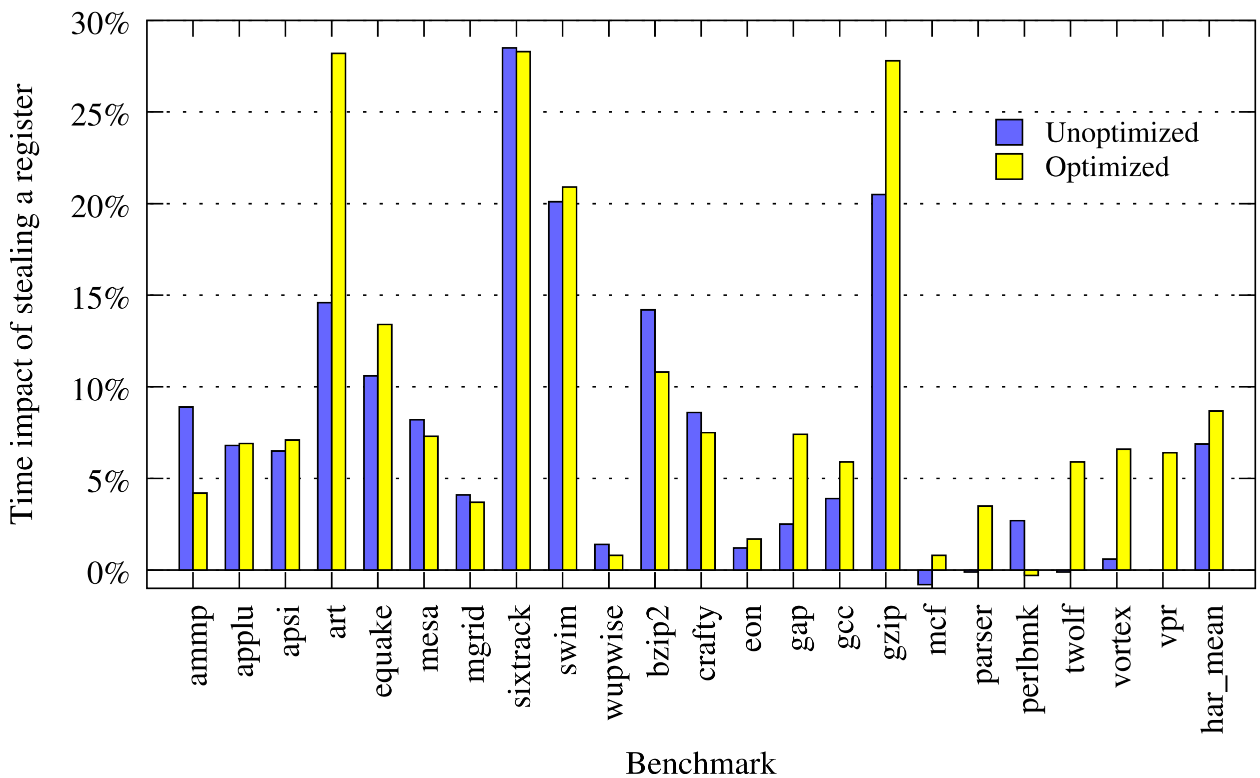

Figure 5.2: Performance impact of stealing `edi` on the SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmarks. The performance is compared to using an absolute address for scratch space. Numbers for the benchmarks compiled unoptimized (`gcc -O0`) and optimized (`gcc -O3`) are given, as performance depends on the compiler's register allocation.

Another source for scratch space is the application stack. We also rejected this idea, because it assumes that the stack pointer is always valid, which goes against our universality goal. We have observed cases in commercial applications where values beyond the top of the stack are live; hand-crafted code does not always obey stack conventions; and for error transparency we want stack overflows to happen at the same place and in the application's own code, not in our code (Section 3.2.4).

Revisiting absolutely-addressed scratch space, recall that it can be used if only one thread will execute any particular code cache fragment at a time. As Chapter 6 discusses, separating the code completely into thread-private pieces makes cache management much simpler. It also makes memory allocation and data structure access more efficient. For these reasons DynamoRIO prefers thread-private caches (see Section 6.1.2). We can hardcode a thread-private absolute address for

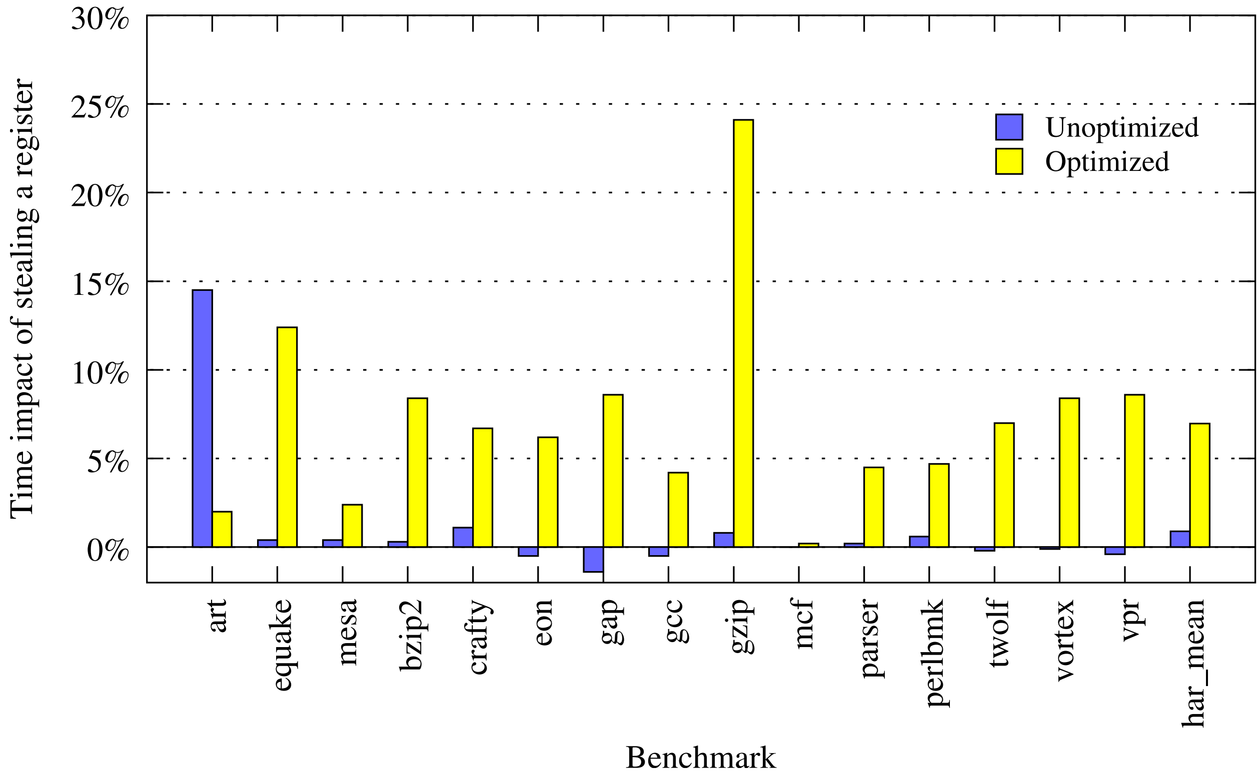

Figure 5.3: Performance impact of stealing `edi` on the non-FORTRAN SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmarks on Windows. The performance is compared to using an absolute address for scratch space. Numbers for the benchmarks compiled unoptimized (`cl /O0`) and optimized (`cl /Ox`) are given, as performance depends on the compiler's register allocation.

scratch space into all inserted code in each of these caches.

A final scratch space alternative is to use IA-32 segments. On Windows segments are used to provide thread-local storage, with the `fs` segment register set up to point to a thread's *Thread Environment Block* [Solomon and Russinovich 2000, Pietrek 1996], or TEB (it is also known as the TIB, I for *Information*). The TEB stores context information for the loader, user libraries, and exception dispatcher. It is also where application thread-local storage is kept. This means that we do not have to steal the register itself; we only need to steal some thread-local storage slots from the application. We use these slots for storing our own thread-private context data structure (Section 5.2.2) as well as for scratch space in thread-shared caches (Section 6.5), but we continue to use absolute addressing for thread-private scratch space.

Thread-local storage on Windows is divided into dynamic and static. Most applications use static storage, which is kept in a separate array that is pointed to by the TEB. Dynamic storage is limited in size to 64 slots kept in the TEB itself, plus (on Windows 2000 and later) 1024 slots kept in the same array as the static slots. Dynamic slots are only used by shared libraries, who do not know what other libraries may be loaded at the same time, and will typically take just one slot that points to a structure containing all the thread-local data they need. Our stealing of one or two slots is unlikely to be noticed. We steal from the end of the TEB slots, to avoid disrupting the dynamic sequence of slot allocation as much as possible.

Segments can also be used on Linux. For our version of Linux we must create the segment descriptor ourselves and load the segment selector. We have not yet implemented this stealing transparently to avoid conflicts with Linux applications that try to use the same segment register (Section 4.1.2). The next generation of Linux threads [Drepper and Molnar] uses `gs` in much the same way as Windows uses `fs`, which eliminates the need to create a custom segment descriptor.

5.2.2 Thread-Local State

DynamoRIO needs to save the application context when switching from the code cache to DynamoRIO code, and must restore it when switching back. This context needs to include all state that DynamoRIO might modify. As described in Section 4.7, this mainly means the general-purpose registers and the condition codes. Since we use some user library routines on both Windows and Linux (see Section 3.1.1), we also must preserve an important piece of persistent state used by those routines: the error code. Using dedicated system threads instead of the application threads would also solve this problem, but for transparency and performance reasons we avoid that solution (see Section 3.2.1). DynamoRIO keeps a data structure for each thread that is accessible via thread-local storage slots for DynamoRIO code and absolute addressing (Section 5.2.1) for code cache code. This structure holds the register spill scratch space, which doubles as the application context storage space while in DynamoRIO code. A slot is allocated for the application error code as well.

```
suspend_thread(thread_id tid)
  suspend_thread_system_call(tid)
  if tid == my_tid
    exit # since now resumed
  cxt = get_context(tid)
  if in_DynamoRIO_code_but_not_at_wait_point_below(cxt)
    tid_wait_at_safe_point_at_DynamoRIO_code_exit = true
    resume_thread_system_call(tid)
    wait(tid_at_safe_point)
```

Figure 5.4: Pseudocode for handling one thread suspending another, to avoid suspending a thread inside of a DynamoRIO routine (while holding a lock or in the middle of persistent state changes). The wait point itself must be considered safe, to handle two threads suspending each other.

5.2.3 Synchronization

Concurrency is difficult to get right. Complex software projects must carefully plan out how multiple threads will be synchronized. Given how hard it is when the developers have control over the code and the threads involved, imagine the problems when the code and threads can be arbitrary. This is the problem of synchronization transparency (Section 3.1.4). Our synchronization challenges are akin to those of an operating system in that it must deal with arbitrary actions by threads. An operating system has to worry about hardware interrupts, though, which make its job harder than ours.

The most obvious problem is one thread suspending another. We cannot allow this to happen while the target thread is in a critical DynamoRIO routine, as that routine may not be re-entrant, and the target thread may own some critical locks. We intercept each thread suspension system call, after the call has gone through, and check where the target thread is. If it is not at a safe point, we set a flag, resume the thread, and wait for it to reach a safe point in our central dispatch routine that reads the flag. It is guaranteed to get there before entering the code cache, and so it will reach there without an arbitrary wait. We must consider the suspending thread's wait point to be a safe point to ensure that two threads suspending each other will not get stuck. Figure 5.4 gives pseudocode for this scheme.

Other problems involve race conditions in DynamoRIO code. During early development of DynamoRIO we often thought that we could ignore a corner case race condition because it would only happen with a bad race condition in the application itself. Then later we would encounter a case where it actually happened in a real application! The problem is that of transparency: we must make it look like an application race condition rather than DynamoRIO crashing or doing something wrong. Many of these are related to decoding and memory unmapping. For decoding, a solution like that proposed in Section 3.3.5 of enabling our exception handler to distinguish an application error could work here as well. For memory unmapping, our cache consistency algorithm tries to handle all synchronization cases (see Section 6.2).

5.3 Kernel-Mediated Control Transfers

Our primary goal is comprehensiveness with respect to application code. No original application code should ever be executed — instead, a copy in our code cache should execute in its place. This requires intercepting all transfers of control. Explicit user-mode control transfers through direct and indirect branch instructions will never leave the code cache. However, there are kernel-mediated control transfers that must be specially intercepted to ensure comprehensiveness. These include Linux signals and Windows callbacks and exceptions. The bulk of the code executed in a typical Windows graphical application is in callback routines, which would be missed if only normal control transfers were followed. In addition to not losing control, the fact that the interrupted stream of execution could be returned to requires managing DynamoRIO state in a careful manner.

Interestingly, a common example of abnormal control flow, the C library routines `setjmp` and `longjmp`, do not require any special handling. The unwinding of the application stack and the final setting of the program counter are all performed in user mode. DynamoRIO simply sees an indirect branch.

A key challenge on Windows is that the exact mechanisms used by the kernel for these events are not officially documented. The Windows source code has not been examined by any of the authors. As such, other methods than those we present here for handling these challenges may exist. All of our information was obtained from observation and from a few books and articles [Solomon and Russinovich 2000, Nebbett 2000, Pietrek 1997, Pietrek 1996]. Mojo [Chen et al. 2000] from Microsoft Research was able to intercept some types of Windows events. However, lack of detail in their sole publication has made it difficult to duplicate their results. Furthermore, they only intercepted a subset of the possible kernel transfers, and did not address detecting the completion

Name	Handler completed with	Delivery points	Old & new context visible?	DynamoRIO interception	DynamoRIO continuation
callback	`NtCallback-Return`, `int 0x2B`	in any interruptible system call	no	`ntdll.dll` trampoline	context stack
asynchronous procedure call (APC)	`NtContinue`	in any interruptible system call	yes	`ntdll.dll` trampoline	stateless
exception	`NtContinue`	synchronous	yes	`ntdll.dll` trampoline	stateless
`NtSetContext-Thread`		synchronous	yes	modify system call parameters	stateless
signal	`sigreturn`, `rt_sigreturn`	at any time	yes	replace handler, delay if in DynamoRIO	stateless

Table 5.5: Summary of kernel-mediated control transfer types on Windows and Linux and how DynamoRIO handles them. The transfer-ending system calls `NtContinue`, `sigreturn`, and `rt_sigreturn` must be watched on their own as well, as they can technically be used independently of their main transfer type. Thread and process creation and `execve` also must be monitored (Section 5.3.6). See the appropriate sections for details on DynamoRIO handling.

of an event handler nor its ramifications on resumption of runtime system state.

Table 5.5 summarizes the kernel-mediated control transfers on Windows and Linux, which the following sections describe and show how to intercept.

5.3.1 Callbacks

The Windows operating system implements many features through message passing. The system delivers events to threads by adding messages to the threads' *message queues*. A thread processes its messages asynchronously via callback routines. Different types of messages are placed in each message queue, but the mechanism for processing all messages is essentially the same. At certain points during program execution, the operating system checks for pending messages. If there are any, the thread's current state is saved – although where it is saved, a critical detail to us, differs among message types. The kernel then sets up the thread to execute whatever callback routine is registered to handle the message. Once the handler is finished, the saved state is restored, and the

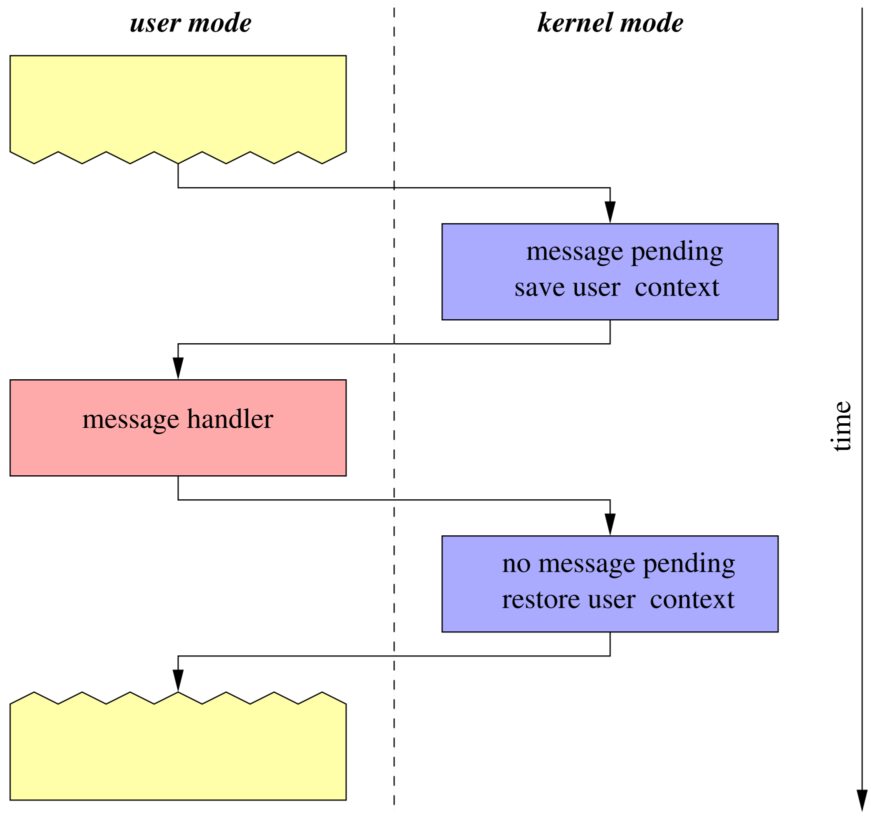

Figure 5.6: Control flow of Windows message delivery, used for callbacks and asynchronous procedure calls. When an application thread is in a certain state, after entering the kernel, prior to returning control to user mode the kernel checks for pending messages on the thread's message queues. If there are any, it saves the current state of the thread and enlists it to execute the registered handler for the target message. Once the handler finishes, if there are no more messages pending, the saved state is resumed.

original execution continues as though it were never interrupted. This whole sequence is illustrated in Figure 5.6.

Callbacks are used extensively in graphical applications to receive user input events. In fact, many Windows applications spend more time in callbacks than in non-callback code. Callbacks can be nested — that is, another callback can be triggered if during execution of a callback the kernel is entered and there are pending messages. In this case the handler will be suspended while the new message is handled. Nesting can be multiple layers deep.

Intercepting Callbacks

One method of intercepting callbacks is to watch for registration of all callback routines and replace the registration with a routine in our own system. However, this requires detailed knowledge of all possible callback routines within the entire Win32 API, which is very large. FX!32 [Chernoff et al. 1998] did in fact wrap every API routine, and recorded which ones took callback handlers as parameters, modifying them in order to retain control when the handler was called. However, this was an immense effort (at the time there were over 12,000 routines that needed to be wrapped), and not straightforward: there are implicit or default callbacks that are hard to discover. Furthermore, this methodology is fragile, targeting a specific version of the API, and requiring significant work with each addition to the API, which is continuously being added to. We rejected this solution for these reasons.

One fortunate fact makes interception feasible. After the kernel sets up a thread to run a callback it re-enters user mode through an exported routine in `ntdll.dll` called `KiUserCallbackDispatcher`. This is a perfect hook for intercepting callbacks. DynamoRIO inserts a trampoline (a jump instruction) at the top of `KiUserCallbackDispatcher` that targets its own routine, which starts up execution of the `KiUserCallbackDispatcher` code inside of DynamoRIO's code cache. This trampoline causes the copy-on-write mechanism to create a private copy of the page containing the top of `KiUserCallbackDispatcher`; the rest of `ntdll.dll` continues to be shared across processes. `ntdll` trampolines are the only cases where DynamoRIO modifies application code. This `ntdll.dll` entry point only exists on Windows NT and its derivatives, not the Windows 95 family. Different techniques may be required to maintain control in these other versions of Windows, which we have not studied.

Suspension Complications

The saved application state that is suspended while the same thread executes other code is problematic. We must not delete the fragment containing the suspension point — otherwise, when the callback handler finishes and the state is restored the thread's resumption point will have been clobbered. Unfortunately, the saved state is kept in kernel mode. We believe it is in the data structure `ETHREAD->KTHREAD->CallbackStack` [Solomon and Russinovich 2000], although there is no documentation on it. DynamoRIO cannot determine from user mode which instruction a thread was executing when the kernel interrupted it to handle a callback.

In which situations the kernel interrupts a thread to deliver a callback is also not documented anywhere (that we could find). From observation we concluded that callback messages are only delivered when a thread is in an *alertable wait state*. A thread is only in this state if it waits on a kernel object handle or it directly tests whether it has pending messages. (This is the same state that triggers user mode asynchronous procedure calls [Solomon and Russinovich 2000] — see Section 5.3.2.) The upshot is that a thread will only be interrupted for callback delivery during certain system calls (we call them *interruptible* system calls) that it makes — never during thread context switches or any other reason it might be in the kernel. The consequences of this are huge, and greatly simplify handling callbacks. We do not have to worry about callbacks coming in at arbitrary times, which causes headaches for Linux signal handling (Section 5.3.5). Although we have no proof that callbacks are only delivered to alertable threads, we have yet to see our conjecture violated in several years of experimentation. We do not know the full list of interruptible system calls, but it seems that quite a few of the `win32k.sys` system calls are interruptible — perhaps all of them explicitly test for callbacks? As for the `ntoskrnl.exe` system calls, only those with the words "Alert" or "Wait" in their names or their parameter names are interruptible.

To solve the suspension problem, DynamoRIO never performs interruptible system calls inside of fragments. Instead we route them all through a single system call location (per thread). This means that DynamoRIO does not have to worry about deleting a fragment from underneath a suspended callback point. However, the many-to-one use of this shared system call means that DynamoRIO must store a return address somewhere. This extra state complicates callback returns, as we will see, but it is less painful than having un-deletable system-call-containing fragments.

Callback Returns

When callback routines finish, they almost never return all the way back to `KiUserCallbackDispatcher`. Instead, they indicate that they are finished by either calling the `NtCallbackReturn` [Nebbett 2000] system call or by executing `int 0x2B` [Solomon and Russinovich 2000], both of which map to the same kernel handler routine. If a callback does return, `KiUserCallbackDispatcher` will invoke interrupt `0x2B` on behalf of the handler, returning the status the handler returned.

These return mechanisms cause the thread to re-enter the kernel. If there are no more messages pending, the kernel restores the saved user context and upon returning to user mode the thread continues with its original execution. If DynamoRIO had no extra state across the callback, it could ignore the callback return and let the kernel resume it. However, DynamoRIO routes all interruptible system calls through a central location, requiring that return address for where to go after the system call. Since the machine context must match the native application state when entering the kernel, this return address must be stored in memory, where it will not be preserved on the `CallbackStack` by the kernel. And since callbacks can be nested, we must have a stack of return addresses. With a stack we must intercept both the callback entry point (to push) and the callback return point (to pop). If we were stealing a register we would have the same state problem. DynamoRIO uses a *callback stack* of thread context data structures (see Section 5.2.2), making it easy to add other state for preservation across callbacks. We also have certain fields of the thread context that are shared across callbacks. On entry to a callback, we push the current context and build a new one (or re-use an old one if we have already been this deep in the callback stack before). On a callback return, we pop the saved context off the stack and restore it as the new current context. Since DynamoRIO uses absolute addressing for scratch space (Section 5.2.1), it must always keep the current context at the same address, and shift the others around in the callback "stack".

Not knowing where a callback will return to means that if we take over control of an application in the middle of execution of a callback handler, we will lose control after the callback return, since we never saw the corresponding callback entry and have no way to intercept the target of the callback return. This prevents any sound method of attaching to an already-running process (see Section 5.5).

Another problem with having to keep state across callbacks is that we will leak one slot on our context stack if a callback does not return. We have never seen this happen, and it is unlikely to happen since the callback return mechanism is underneath the Win32 API layer.

Another problematic issue is handling callbacks received while executing DynamoRIO code. The optimal solution is to avoid such callbacks if at all possible, by never making interruptible system calls. If a system must do so, it would need to queue up any messages received while out of the code cache for delivery once at a stable point. This would not be easy to do transparently due to lack of access to the saved context. One method would be to save the callback handler context and execute a callback return right away. On the next interruptible system call executed by the application, DynamoRIO would emulate the kernel and set the thread up to execute the saved handler context. The next callback return would not be passed to the kernel, but would trigger a return to the system call. The task of emulating the kernel's message delivery is one best avoided. Section 5.3.5 discusses problems with emulating Linux signal delivery.

5.3.2 Asynchronous Procedure Calls

Windows also uses message queues for its Asynchronous Procedure Call (APC) API, which allows threads to communicate with each other by posting messages to their respective queues. The mechanism of APC delivery looks very similar to that of callback delivery (Figure 5.6). APC delivery begins with the kernel transferring control to an entry point in `ntdll.dll` called `KiUserApcDispatcher`, and we intercept this entry point just like we intercept the callback dispatcher.

User mode APCs, again like callbacks, are only delivered when the thread is in an alertable wait state [Solomon and Russinovich 2000]. Conversely, *kernel mode APCs* can be delivered at any time, including during thread context switches. Fortunately these are only used on user threads for applications that target the POSIX subsystem, which are rare, and which we do not support for this reason. Since APCs are triggered by the same interruptible system calls as callbacks, we can use the same shared system call mechanism to ensure there are no suspension points inside fragments. (It is possible that the `win32k.sys` system calls are only interruptible for callbacks and not APCs; we have not confirmed this.)

Yet, the location of the state saved for APC delivery is quite different from that for a callback: it is stored on the user stack. In this respect an APC looks more like a Linux signal (Section 5.3.5). An APC is returned from by executing the `NtContinue` [Nebbett 2000] system call. APCs, unlike callbacks, typically return back to the dispatcher, who executes the `NtContinue`. This system call takes a machine context as an argument, the suspended context that the kernel placed on the stack. This user-mode-accessible context means that we can handle APCs in a stateless manner. When our trampoline in `KiUserApcDispatcher` is invoked, we translate the stored context (using the method of Section 3.3.4) to its native equivalent (instead of the real suspension point, inside of our shared system call routine). This makes us more transparent if the application decides to examine the context. Then, on the `NtContinue`, we change the context's target program counter to our own routine, and store the application target in our thread context.

This stateless handling is cleaner than the callback stack of contexts we must use for handling callbacks. The only complication occurs if we need to perform post-processing (see Section 5.4) on the interruptible system call that triggered the APC. Since we are not returning to our shared system call routine, we must have our `NtContinue` continuation routine perform the post-processing. Being stateless means we do not care if an APC handler does not return at all to the suspended state. Like callback returns, we have never seen this, since the APC return mechanism lies below the Win32 API.

Unlike `NtCallbackReturn`, which due to its use of the kernel-maintained `CallbackStack` can only be used for callbacks, `NtContinue` can be used on its own, and in fact is also used with exceptions (see Section 5.3.3). At the `NtContinue` point itself we cannot tell whether it is being used to return from an APC or not. This would pose problems if we kept state and used our context stack — although we could ignore stack underflow, we would do the wrong thing if in the middle of nested APCs an independent `NtContinue` was executed (one mitigating factor is that we have yet to see an independent `NtContinue`, as it is part of the Native API and not the Win32 API). This is another reason a stateless scheme is superior.

5.3.3 Exceptions

A third type of kernel-mediated control flow on Windows is the exception. Windows calls its exception mechanism *Structured Exception Handling*. Programming language exceptions, such as C++ exceptions, are built on top of Structured Exception Handling.

A fair amount of information is available on the user-mode details of how Windows handles exceptions [Pietrek 1997]. Figure 5.7 shows the control flow, which is similar to that of an APC or callback. A faulting instruction causes a trap to the kernel, which saves the user context and then sets the thread up to enter user mode at the `KiUserExceptionDispatcher` entry point in `ntdll.dll`. This exception dispatcher walks the stack of exception handlers, asking each whether it wants to handle this exception. If it does, it can either re-execute the instruction, or instead

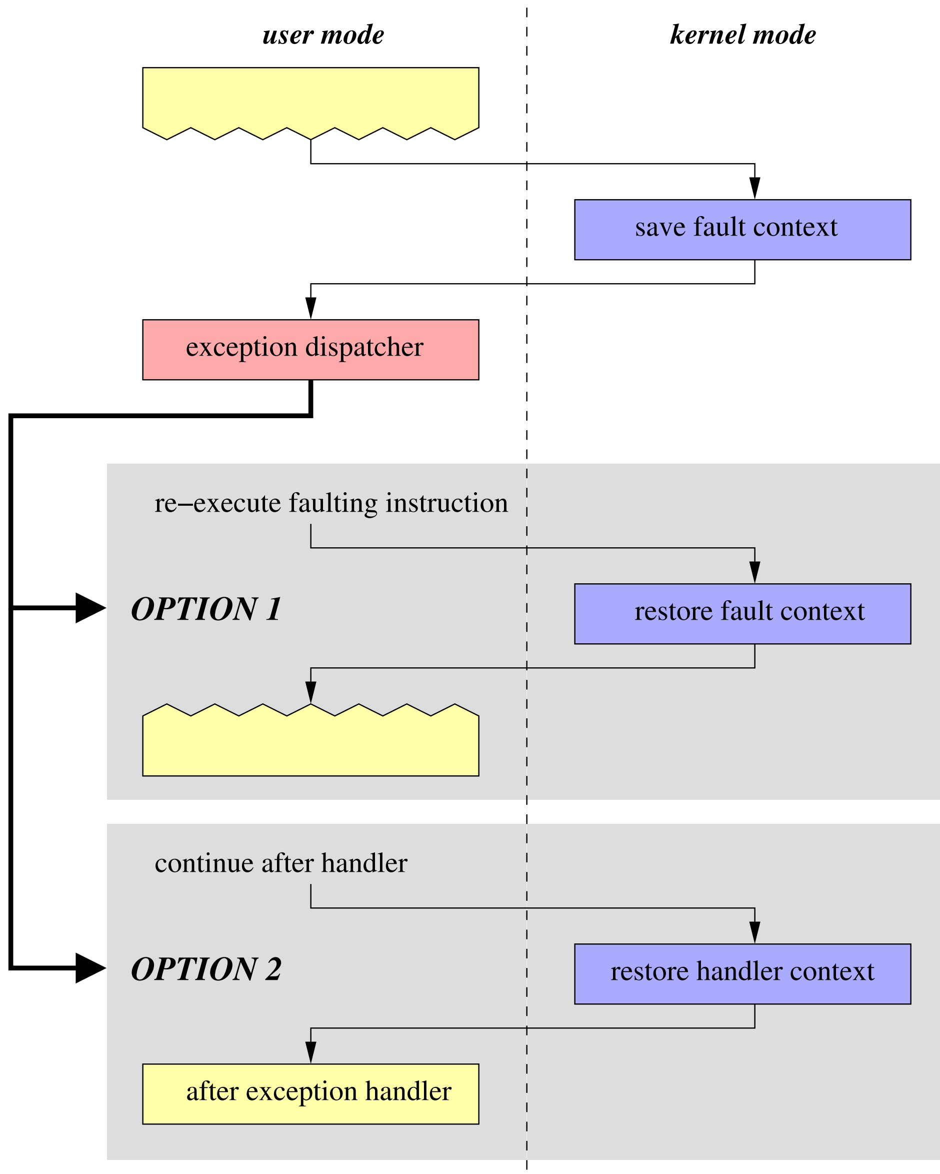

Figure 5.7: Control flow of a Windows exception. A faulting instruction causes a trap to the kernel, which saves the user context and then causes the thread to re-enter user mode in a special exception dispatcher routine that searches for an exception handler. If a handler accepts this fault, it can either re-execute the faulting instruction or continue execution after the handler and abandon the saved context.

continue execution after the handler and abandon the saved context of the faulting instruction.

Just like for an APC, the saved user context is stored on the stack and is accessible to user mode code. When the dispatcher routine is entered, just like for APCs we translate the stored context from the code cache location where the fault actually occurred to the corresponding native location. Context translation is more critical for exceptions than for APCs: exception filters frequently access the context, while APC handlers do not, but more importantly, APCs only happen during interruptible system calls, while exceptions can occur anywhere in the code cache and even in code we insert, such as the test to see if an indirect branch stays on a trace. This makes it more difficult to deal with exceptions in a stateless manner, since we must get the entire machine context just right in order for the continuation after the exception to be correct. If we kept state across exceptions, we would only worry about satisfying the exception handler's examination of the context, likely limited to the faulting instruction address.

DynamoRIO uses a stateless scheme for exceptions, just like for APCs. Both exception outcomes use `NtContinue`, which we handle just like we do for APCs. It would be difficult to distinguish different uses of `NtContinue`, and making all uses stateless is cleanest. If the exception handler chooses to abandon the faulting context, the exception dispatcher changes the context on the stack from the faulting instruction to that of the exception handler. If the instruction is re-executed instead, our stateless handling will build a new fragment beginning with the faulting instruction. We decided that this code duplication, even combined with more critical context translation, is cleaner than keeping state and marking the faulting instruction's fragment as undeletable. Being stateless also removes worry about an exception not returning via `NtContinue`. See Section 5.3.5 below for more arguments for stateless handling of faults.

An exception generated through the `RaiseException` Win32 API routine builds a synthetic context from user mode. Since we are transparent, it builds a context that points to native state, not our code cache. When our exception dispatcher trampoline goes to translate the context and finds a native address, it assumes it came from `RaiseException` or some other user mode context creator, and leaves it alone.

Our exception dispatcher trampoline also detects whether a fault occurred because we made a code region read-only for cache consistency purposes (see Section 6.2). But, surprisingly, Windows provides no mechanism for an alternate exception handling stack, making it impossible to handle a faulty stack pointer. (Windows does provide the notion of a `guard page` [Richter 1999] to enable specialized detection of stack overflows.) Given so many interruptions of program flow (callbacks, APCs, exceptions) that require the application stack, the lack of an alternate stack feature is a poor design. It causes potentially fatal problems with our cache consistency algorithm for which the only workaround is to monitor changes in the stack pointer and try to detect when it is being used for other purposes than to point to a valid stack.

5.3.4 Other Windows Transfers

Other Windows kernel-mediated control transfers exist beyond callbacks, APCs, and exceptions. As mentioned above, the `NtContinue` system call could be used independently of APCs and exceptions. Another system call we must watch is `NtSetContextThread` [Nebbett 2000], which can modify another thread's program counter. When we see it we change the context it passes to the kernel to instead point to our own interception routine, and we store the target program address in our thread context, so that once the kernel sets the state of the target thread we will be set up to control execution from the correct point. The corresponding `NtGetContextThread` must be intercepted as well, and its context translated to a native value, for transparency (Section 3.3.4). Additionally, DynamoRIO intercepts another `ntdll.dll` entry point, `KiRaiseUserExceptionDispatcher`, though it simply calls the Native API routine `RtlRaiseException` and dives right back into the kernel.

Table 5.8 shows how frequently the main transfer types of callbacks, asynchronous procedure calls, and exceptions occur in our benchmarks. We have only seen `NtSetContextThread` in a handful of applications, and have never seen `KiRaiseUserExceptionDispatcher`. `NtContinue` is used all the time for APCs and exceptions, but we have never seen it used independently.

5.3.5 Signals

We now turn our attention to Linux. The only kernel-mediated control transfers on Linux are signals. Figure 5.6, showing the control flow of Windows message delivery, is also an accurate picture of Linux signal delivery. However, there are important differences in the details.

The biggest difference between Linux signals and Windows events is that signals can be delivered at any time. The kernel checks for pending signals every time it is about to re-enter user mode, including for thread context switches, which can occur at arbitrary times, presenting a significant challenge for signal handling in a runtime code manipulation system. Fortunately, like APCs and exceptions, the interrupted user context is stored in user mode. Otherwise signal handling would

Benchmark	Callbacks	APCs	Exceptions
batch excel	744	3	0
batch photoshp	168441	9	60
batch powerpnt	227485	4	0
batch winword	612	3	0
interactive excel	14211	9	0
interactive photoshp	56694	6	42
interactive powerpnt	20642	6	0
interactive winword	30733	16	10
IIS inetinfo	18	55	2
IIS dllhost	8	9	0
apache	0	256	0
sqlservr	2	286	11

Table 5.8: Event counts for the three main types of kernel-mediated control flow in Windows: callbacks, asynchronous procedure calls (APCs), and exceptions, for our desktop and server benchmarks. The batch scenarios are our benchmark workloads, while the interactive are for using our desktop applications manually in an interactive setting (the same as in Table 6.1).

be an intractable problem.

Another difference is that Linux signal handlers often do not return. A common error handling technique is to `longjmp` from the handler to a prior point on the call stack. This adds complexity to any scheme that keeps state across signals. As mentioned in Section 5.3.1, we will leak some memory on a callback that does not return, but it is extremely unlikely (and has never been observed) to happen. With signals such a leak may be frequent enough to be problematic, a significant drawback to keeping state.

The following subsections break down the challenges of handling Linux signals, and discuss our implementation and alternative solutions.

Interception

Signal interception is quite different from the way we intercept Windows events. The number of signal types is limited, and an application must register a signal handler for each type it wishes to receive. If the application does not register a handler, the default action for that signal type is taken, which is generally either nothing or process termination, depending on the signal. In the 2.4 Linux kernel, there are 64 signal types: 32 standard signals and 32 *real-time signals*. The

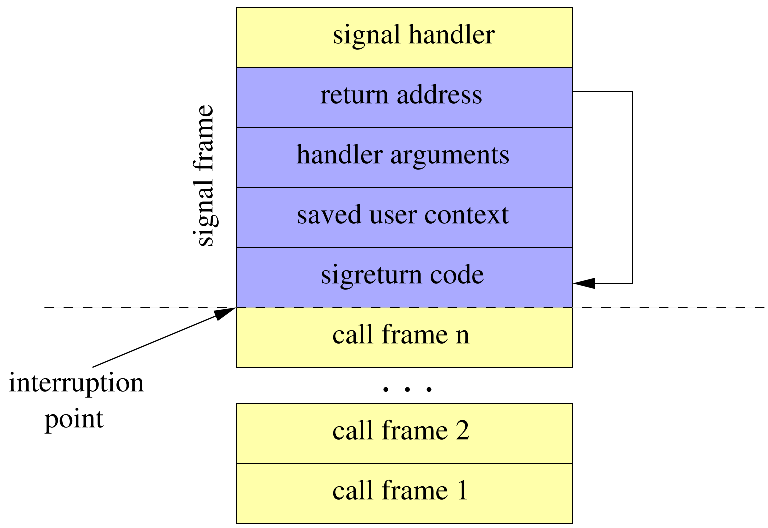

Figure 5.9: Stack layout of a signal frame. The frame is laid out such that the signal handler's return address points to code that executes the `sigreturn` system call at the bottom of the frame. The arguments to the handler point into the saved user context, which is used to resume execution when the `sigreturn` is invoked.

main distinction of real-time signals is that each type can have multiple pending instances at once, while standard signals can only have one pending instance of each type at a time. To deliver a signal, the kernel sends control directly to the registered application handler. To intercept the handler execution, DynamoRIO registers its own handler in place of each of the application's by modifying the arguments to all `signal` or `sigaction` system calls. DynamoRIO uses a single master handler for all signal types.

When a signal is received, the suspended application context, which is stored on the user stack, must be translated to make it seem to have occurred while in application code rather than the code cache. Figure 5.9 shows the actual layout of the *signal frame* constructed by the kernel at the interruption point on the user stack. The signal handler is invoked such that its arguments are on the top of the signal frame and point into the context deeper in the frame. Context translation is not as simple as for Windows events because signals can occur at any time, which we discuss below.

A signal handler returns to the suspended context by issuing the `sigreturn` system call (or the similar `rt_sigreturn` for real-time signals), which is similar to Windows' `NtContinue`. Figure 5.9 shows how a standard signal frame sets up the return address of the handler to point to a short sequence of code that invokes `sigreturn` with the stored context as an argument. Signal handlers can also specify a separate piece of code to be used for handler return, which is the default when a handler is installed via the C library signal support routines. And, as mentioned earlier, signal handlers frequently do not return at all, instead issuing a `longjmp` call to roll back to a prior point on the call stack. Like `NtContinue`, `sigreturn` could be used independently of signals for control transfer, though as it only affects the current thread it is an inefficient choice.

Arbitrary Arrival Times

The kernel delivers pending signals at arbitrary points in a thread's execution. They could come while outside of the code cache, while in the context switch code, or anywhere in DynamoRIO code. We break signals into two groups: those that can always be delayed and those that may must be delivered immediately. We must be conservative here. For example, a memory access fault — signal `SIGSEGV` — usually must be delivered immediately, since the execution stream cannot continue without correcting the fault. However, one thread could pass a `SIGSEGV` to another thread (threads can pass arbitrary signals to each other), having nothing to do with a fault. We do not try to tell the difference and assume that `SIGSEGV` cannot be delayed, unless it is received while out of the code cache. A signal meant for the application (as opposed to a bug in DynamoRIO, or perhaps if DynamoRIO is profiling using signals as described in Section 7.3.1) that arrives while in DynamoRIO code will always be delayable (by assuming that normally non-delayable signals arriving then must be pathological cases like the non-fault `SIGSEGV` mentioned above). Delayable signals can be queued up for delivery at a convenient time. Bounding the delay makes for better transparency, but anything reasonable will do for signals other than timer signals. For signals received while in DynamoRIO code, waiting until the code cache is about to be entered is acceptable.

Delayable signals received while in the code cache are also delayed, to simplify context translation. We unlink the current fragment that the thread is inside, and ensure it does not execute any system calls in that fragment (by exiting prior to the system call — see Section 5.4), to bound the delay time before the thread returns to DynamoRIO code and the signal can be delivered with a clean context. The context of this cache exit can be used, since the application could have received this asynchronous signal anywhere within the immediate vicinity in its execution stream.

To delay signals, they must be stored in a queue, requiring DynamoRIO's signal handler to allocate memory. Memory allocation is a non-local, multi-instruction state change, rendering the signal handler non-re-entrant. This, in turn, means that we must block all signals while in our handler. This is not a problem for real-time signals, since they will be queued up by the kernel and delivered once we leave the handler and unblock signals again, and we are not completely transparent with respect to timing anyway (see Section 3.3.6). Blocking while in our handler might result in missing standard (non-real-time) signals, but there is nothing we can do about it. Since a signal could interrupt our memory allocation routines, we use a special allocator dedicated to our signal handler that hands out pieces of memory the size of a signal frame without any synchronization. We must have a static bound on the size of our signal queue since we cannot re-size this special allocation unit at arbitrary times.

Our delayed signal queue is simply a linked list of signal frames. We copy the frame that the kernel placed on the stack (Figure 5.9) to the queue. The kernel uses two different types of frames, real-time frames and standard frames. Counter-intuitively, real-time frames are not only used for real-time signals, they are also used for any signal whose handler requests more than one argument. The information contained in a real-time frame is a superset of that in a standard frame. Our master signal handler requests a real-time frame, and our queue consists of real-time frames. If we must deliver a signal to a standard handler, we translate to a standard frame. Delivering signals ourselves requires that we duplicate the kernel's behavior, which we discuss below.

State Options

Non-delayable signals must be delivered immediately. These signals are synchronous and correspond to Windows exceptions. We have the same choices here as we did for exceptions: stateless versus stateful handling. Stateless handling requires perfect context translation (Section 3.3.4) and abandonment of any ongoing trace, so periodic signals could result in no forward progress in trace building. Keeping state, however, requires a stack of contexts, just like for Windows callbacks. And since signal handlers do not always return, contexts on the stack might be leaked. Schemes for watching for `longjmp` could mitigate this, but a leak will always be a possibility. Using state also requires un-deletable fragments (now at arbitrary places in the cache) that, again because the signal might not return, might never be marked deletable again. DynamoRIO uses stateless signal handling, which we feel is superior to keeping state across signals, just like for APCs and exceptions.

Kernel Emulation

By delaying signals and delivering them ourselves, we must faithfully emulate the kernel's behavior in copying a signal frame to the application stack and invoking the application signal handler. For example, we must record whether the application asked for an alternate signal handling stack. We must ensure that our signal data structures exactly match the kernel's, which differ in key ways from the C library's signal data structures (see Section 3.1.1), causing us a few headaches in translating between the two. By delaying delivery, we introduce the possibility of changes in the application's blocked signal set since the kernel delivered the signal. Furthermore, since we must keep state about the application's signal handling requests, we must make sure we inherit it properly in child processes. Needless to say, this makes our code quite dependent on the exact signal behavior of the kernel, making DynamoRIO less portable.

Not only must we emulate the kernel in delivering signals to application handlers, but we also must emulate the default signal behavior for signals that DynamoRIO wishes to intercept but the application does not. For example, we trap `SIGSEGV` to provide error handling when our system has a memory access violation and for our cache consistency algorithm (Section 6.2). But if the `SIGSEGV` is not ours, we want to pass it on to the application. If the application has a handler, we simply invoke it in our code cache with the same signal frame. If not, we must perform the default action, and some of the kernel's default actions cannot be easily duplicated from user mode, such as dumping core. One method of invoking a signal's default action is to remove its handler and then send the signal to oneself. This is problematic in general, however, as there is now a race condition if another signal comes along while our handler is not installed. Fortunately, DynamoRIO only needs to issue default actions of ignore, for which we can do nothing, or process termination on a synchronous repeatable fault, for which removing our handler and re-executing the instruction is sufficient.

Memory Access Violations

When we intercept a memory access violation, we want to know not only the faulting instruction that performed the read or write but also its bad target address, for cache consistency (Section 6.2) and protecting DynamoRIO from inadvertent or malicious writes (Section 9.4.5). Windows collects this information and hands it to the exception handler. On Linux, however, we must decode the faulting instruction ourselves in order to calculate the target address, as the kernel only tells the signal handler the address of the faulting instruction.

5.3.6 Thread and Process Creation

For comprehensiveness and transparency, DynamoRIO must control all threads in the address space, as any thread can modify the global address space and affect all other threads. This is usually also desirable by the user of the system, to apply a tool to an entire application. If we assume that we begin in control of the initial thread, we simply need to watch for creation of new threads. On Windows, rather than watching the `NtCreateThread` system call, we wait for the initialization APC of the thread, which is always its first target [Solomon and Russinovich 2000]. This way we also catch threads injected by other processes (although this may not always be desirable in the case of a debugger injecting a thread — see Section 3.3.7).

On Linux, we must watch the system calls `clone` (when called with the `CLONE_VM` flag) and `vfork`, both of which create a new process that shares the parent's address space. We insert code after the system call that checks the return value to identify the child. For the child, the `eax` register is 0. We take advantage of this and use `eax` as scratch space to hold the starting address for the child (the address after the system call) while we jump to a thread initialization routine. Without the dead register we would have problems handling a new thread on Linux, since our scratch space requires being set up beforehand for each thread (see Section 5.2.1), and all other registers are in use for these system calls. Just one register is enough to bootstrap the process of swapping to a safe stack (for transparency) and starting up the new thread. We must use a global stack here protected by a mutex since this thread has no private memory set up yet.

A complication is that some system call numbers and parameters are not set statically, so we may not be able to identify all `clone` calls and even for `clone` calls those being passed the `CLONE_VM` flag. To handle these dynamically parametrized system calls, we insert a jump immediately following the system call instruction that can be changed dynamically to either skip the subsequent clone-handling instructions or target them (Figure 5.10). At system call execution we examine the register values and determine whether this is indeed a new thread being created, and

```
    <pre system call handling>
    system call instruction
    jmp <ignore or xchg> # dynamically modifiable
xchg:
    xchg eax,ecx
    jecxz child
    jmp parent
child:
    mov <app pc after system call>,ecx
    jmp new_thread_take_over
parent:
    xchg eax,ecx
ignore:
    <post system call handling>
```

Figure 5.10: Code for handling potential thread creation in dynamically parametrizable system calls whose system call number and parameters are not known until execution. A jump is inserted that can be modified by the pre-system call handler to either execute clone-handling code or skip it. If the system call does turn out to create a new thread, `eax` containing zero is taken advantage of to store the target start address for the thread and control is transfered to a DynamoRIO routine to initialize the new thread.

modify the jump appropriately, restoring it after handling the call. Since our code is thread-private this does not result in a race condition.

When a new process is created, we may or may not want to take control of it, depending on the user's goals. To take control on Windows, we wait for the `NtCreateThread` call, rather than `NtCreateProcess`, since we cannot take over until the initial thread is created. We can distinguish the initial thread from later threads because its initialization routine and argument differ [Solomon and Russinovich 2000], and its containing process is not completely initialized yet (for example, the process identifier is still 0 on Windows 2000). The mechanism for taking control is dependent on the injection method being used, which is discussed in Section 5.5. On Linux, we must watch the `fork`, `clone` (without the `CLONE_VM` flag), and `execve` system calls. The former two do not require injection, as they begin execution with the same state as the parent, and only require explicit duplication of any kernel objects we have created that these system calls do not duplicate, such as open files or shared memory objects. The `execve` system call, however, does require explicit injection support. Section 5.5 explains how DynamoRIO handles it.

5.3.7 Inter-Process Communication

A runtime code manipulation system needs to keep tabs on various application activities (control flow changes, memory allocation changes) in order to maintain correctness and transparency. One serious worry is manipulation by another process. This is one challenge that is unsolved for us.

The Win32 API provides direct access to another process' address space: reading and writing memory, changing page protection, even thread creation. We will notice a new thread, as it will be scheduled with an initialization APC and will trigger our APC entry point trampoline (see Section 5.3.2). However, we have no way to intercept or notice memory manipulation from afar. The risk is of another process marking application code in our process as writable and then modifying it, causing our cache to contain stale code (see Section 6.2 for information on our cache consistency algorithm).

This problem is not limited to Windows. The same scenario can happen on Linux, or any operating system that supports shared virtual memory. Since each process has its own page table, different processes can have different privileges for the same physical memory. Code that we think is read-only may be writable from another process, and again we have a potential cache consistency problem. We know of no efficient solution to this problem from user mode. A kernel mode component could watch for these cases and provide hints to DynamoRIO.

5.3.8 Systematic Code Discovery

An important research question is whether there is a systematic solution to the problem of intercepting these disparate kernel-mediated control transfers. One method is to mark all pages except our code cache and DynamoRIO's own code as non-executable. Then whenever control switches to any application code, we trap the execution fault and redirect control to DynamoRIO. We leave the pages as non-executable, since future executions should all be within our code cache, and thus catch subsequent executions of the original page as well. Unfortunately, IA-32 makes no distinction between read privileges and execute privileges, making this solution infeasible, since keeping all application pages unreadable and emulating all reads is prohibitively slow. This solution could be used on other architectures, however.

5.4 System Calls

To maintain correctness and transparency, a runtime code manipulation system needs to monitor two types of system calls: all calls that affect the address space, such as allocating memory or mapping in a file (e.g., loading a shared library), as well as all calls that affect the flow of control, such as registering a signal handler. System calls are easily intercepted, as they are explicit application requests that are simple to discover during basic block formation. This is an important advantage of operating at the system call level on Windows, instead of at the documented Win32 API level (see Figure 3.1). Despite being the supported and documented interface between an application and the operating system, the Win32 API is very wide and changes more frequently than the Native API system calls. Additionally, the Win32 API can be and is bypassed by hand-crafted code as well as key system processes that do not rely on the Win32 subsystem (e.g., the session manager, the service control manager, `lsass`, and `winlogon` [Solomon and Russinovich 2000]). Finally, it is much simpler to monitor system calls, which use unique instructions to trap control to the operating system, than to watch for user-mode calls to library routines.

Of course, the downside to operating at the system call level is that it is undocumented and may change without notice in future versions of Windows. On Linux, the system call numbers are well documented and do not change, as would be expected with the primary operating system interface. On Windows, however, where the Win32 API is the documented interface, the system call numbers do change from version to version. In fact, the system call numbers are generated automatically by the kernel build process [Solomon and Russinovich 2000]. New system calls are invariably added for a new version of Windows, and the build alphabetizes them before numbering consecutively. Thus the same system call has a different number on different versions of Windows. To handle this, we have a different table of numbers for each Windows version we support (NT, 2000, XP, and 2003). When DynamoRIO initializes it identifies the version and points to the right table. We also decode the system call wrappers in the `ntdll.dll` library to double-check that our numbers are correct.

Several methods exist for performing a system call on the IA-32 platform. Older Linux implementations and Windows NT and 2000 use the `int` instruction: `int 0x80` on Linux and `int 0x2e` on Windows. Intel's Pentium II added a faster method of performing a system call: the `sysenter` and `sysexit` instructions [Intel Corporation 2001, vol. 2]. AMD similarly added `syscall` and `sysret` [Advanced Micro Devices, Inc. 1998]. Later Linux kernels and Windows XP and 2003 use these more efficient instructions if running on an appropriate processor. DynamoRIO contains code to handle all of these types of system calls: to recognize them for interception, and to perform them, since we redirect interruptible system calls on Windows to a shared routine (see Section 5.3). We dynamically determine which type is being used by waiting until we see the application issue its first system call.

The system call number is placed in the `eax` register on both operating systems. Most system call instructions are dedicated to a specific system call (a constant is placed in `eax`). This means that when we find a basic block containing a system call, that call will either always need to be intercepted or can execute without instrumentation in the code cache. When we encounter a system call whose number is either not statically known or on our list of system calls to intercept, we insert instrumentation to call both a pre-system call and a post-system call handling routine around the system call itself. Some system calls must be acted upon before they happen (such as `exit`) while others can only be acted on after the kernel performs the requested operation (such as `mmap`). We must be careful to undo our pre-system call operations if the system call ends up failing, which we can only determine in the post-system call handler.

On Windows, we also must decide if a system call is interruptible or not (see Section 5.3). If it is, we remove the system call instruction from the basic block and insert a jump to a shared system call routine that will perform the system call using the parameters set up by the basic block.

System calls that we do not need to intercept nor redirect to a shared location can execute unchanged. We can even build traces across them, treating them as non-control-flow instructions. However, there is one caveat: for signal handling, we must be able to exit a fragment from an arbitrary point without executing a system call, to bound signal delivery time (see Section 5.3.5). To do this we simply insert an exit from the fragment prior to the system call, with a jump that hops over it by default (Figure 5.11). We can dynamically modify that jump to instead target the fragment exit if we ever receive a signal in that fragment.

A final complication occurs when Windows uses the `sysenter` system call method. The kernel returns control to user mode at a hardcoded location (the address `0x7ffe0304`). The instruction at that location is a return (this is a shared system call routine that is called by every system call

```
    jmp <syscall or bail>  # dynamically modifiable
bail:
    jmp <fragment exit stub>
syscall:
    <system call instruction>
```

Figure 5.11: Code inserted prior to each system call in the code cache to allow bypassing the system call on signal reception, bounding the delay until the code cache is exited and the signal can be delivered.

wrapper). In order to regain control, we replace the application return address on the stack with the address after our own `sysenter` (either in a fragment for an ignorable system call or in the shared system call location). This is a lack of transparency we have to live with, as our only alternative is to insert a trampoline at the hardcoded location, which is not wise since the trampoline, being longer than the return instruction, would overwrite subsequent memory whose use is unknown.

5.5 Injection

This section discusses how DynamoRIO comes to occupy the address space of a target process (called *injection*) and take control of execution in that process. DynamoRIO is built as a shared library that can be dynamically loaded; we just need a trigger to load it at the appropriate point.

As part of our customization interface (see Chapter 8), we export routines allowing an application to start and stop its own execution under DynamoRIO. This requires application source code access. The application is linked with DynamoRIO's shared library, which is loaded when the application starts. The application can then invoke start and stop routines in our library. More information on this interface is in Section 8.1.1. This start/stop interface is for experimentation with special applications. For general-purpose use, we must be able to execute legacy code without source code access. Recall that our transparency goals require us to operate on unmodified binaries, meaning that modifying the executable's initialization routine to load DynamoRIO is not an option.

5.5.1 Windows

A number of strategies are possible for injecting a library into a process on Windows [Richter 1999]. For targeted injection, we create a process for the target executable that begins in a suspended state. We then insert code into the new process' address space that will load DynamoRIO's library and call its initialization and startup routines. We change the suspended initial thread's program counter to point to this injected code, resume the process, and off we go. The target executable is now running under the control of DynamoRIO. We use this targeted injection method for explicitly starting up a process under our control, as well as for following child processes created by a process under our control (see Section 5.3.6).

Our targeted injection does not take over until the image entry point, and it misses the entire initialization APC (this APC is the first thing executed by a new process [Solomon and Russinovich 2000], even before the executable's entry point is reached). We would like to execute that APC ourselves, but to do so requires being independent of all user libraries except `ntdll.dll`, as it is the only library in the address space from the beginning (it is put in place by the kernel), and also being independent of the loader, as it is not initialized yet. Achieving library independence and early injection in DynamoRIO is future work.

The act of taking control of the execution after a call to our initialization or other routine involves never returning from that routine. Instead, we start copying the basic block beginning at the routine's return address. We execute that block inside of our code cache, and go from there.

We would also like to take over processes not explicitly created by the user. Our current implementation on Windows uses a feature provided by `user32.dll` [Richter 1999]: when `user32.dll` initializes itself, it reads the value of a certain registry key (`HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Windows/AppInit_DLLs`) and loads any libraries named there. By placing our library in that key, and having our initialization routine take control, we can be automatically injected into every process that loads `user32.dll`, which includes nearly every interactive Windows program but not all console applications or services.

As mentioned in Section 5.3.1, we have a serious problem with taking over in the middle of a callback on Windows: we lose control when the callback returns. When using the `user32.dll` injection method, we end up taking control partway through the initialization APC of the first thread in the process. We often lose control before that routine finishes, as it is frequently interrupted for a callback or two. We use a trick to regain control: we place a trampoline at the image entry point of the executable for the process. When the initialization APC finishes, the image entry point is executed (actually, a special startup routine in `kernel32.dll` is executed, which then targets the real image entry point). If we want to try to restore control sooner, we can place trampolines at other points, such as Win32 API routines for loading libraries. These problems also complicate attaching to an already running process, which would be a nice mechanism of targeting an application, rather than requiring that it be started up under DynamoRIO control.

We do not use the operating system's debugging interface to inject ourselves into a process because, on Windows NT and Windows 2000, there is no support for detaching a debugger without killing the debugee process. We would like to be transparent even to debuggers (Section 3.3.7), and there can only be one debugger attached to a process.

5.5.2 Linux

On Linux, we use the `LD_PRELOAD` feature of the loader to have our library injected into any ELF [Tool Interface Standards Committee 1995] binary that is executed. This feature cannot be used for setuid binaries without modifying the `/etc/ld.so.preload` file, which requires superuser privileges. For truly transparent injection, the user loader must be modified. As we mentioned above, we would like to be transparent to debuggers, which is why we do not use the `ptrace` debugging interface on Linux to inject ourselves. Since our Linux injection relies on an environment variable, we must be sure to propagate that variable across an `execve` system call. We check whether the `execve` keeps the previous environment assignments, and if not, we add the `LD_PRELOAD` variable to its environment block, causing DynamoRIO to be loaded into the process' new address space.

5.6 Chapter Summary

This chapter has shown how to transparently interpose a runtime system between the operating system and the application. This involves ensuring that multiple threads are properly handled, kernel-mediated control transfers are intercepted to avoid losing control, and that system calls are monitored as necessary. The final piece of operating system interaction is *injection*, obtaining control of a target process in the first place. The next chapter turns to the issue of memory management, both of the code cache itself and of the data structures required to manage it.

Chapter 6

Memory Management

Many of DynamoRIO's memory management strategies are heavily influenced by the wide range of large applications we support, and would have gone in different directions if we only focused on typical benchmarks. In fact, if we were only interested in running a single SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmark at a time, we could concern ourselves with only performance and not space, since these benchmarks are single-threaded with either small code footprints or far larger data sizes than code sizes. When running large applications, however, memory expansion is a problem, as DynamoRIO makes a copy of all code executed and needs to maintain data structures for each code fragment. Modern applications also require effort to keep DynamoRIO's code cache consistent as libraries are unloaded and code regions are modified.

This chapter first analyzes memory usage requirements in general and with respect to threads (Section 6.1) before presenting our solution to the difficult problem of cache consistency (Section 6.2). Next we show how to use cache capacity to limit memory usage (Section 6.3), and how we manage the data structures in our heap (Section 6.4). Finally, we evaluate DynamoRIO's total memory usage (Section 6.5).

6.1 Storage Requirements

DynamoRIO must allocate memory both for its code cache and for associated data structures needed to manage the cache, so total memory usage scales with the amount of code in the application. The code cache alone, with the additions required to maintain control (Section 2.1 and 2.2), is larger than the original application code, and associated data structures require even more memory. However, not every data structure need be kept permanently. For every piece of information DynamoRIO needs at some point during operation, there is a tradeoff between the space to store it and the time to re-calculate it on demand. For example, we must decode the application's code to copy it into our code cache. The high-level instruction representation we build for that purpose is only needed temporarily; thus, we do not store it, but throw it away as soon as it has been encoded into machine code in our cache. If we later need to find the corresponding application instruction for a given code cache instruction, we re-decode the application code (see Section 3.3.4).

On the other hand, we do store the target of each direct branch, rather than re-decoding the application code to determine it, as it is used often enough to justify the space. We also store each code fragment's original application address, code cache entry point, size, and the offset of each exit within the fragment. Furthermore, we need a data structure for each fragment exit, to determine from where we exited the code cache (Section 2.1). Large applications execute large amounts of code (e.g., see Table 7.3), resulting in hundreds of thousands of each of these data structures. Limiting memory usage requires limiting the code cache size, which determines the number of data structures. However, managing the code cache is difficult.

6.1.1 Cache Management Challenges

Managing fragments in a code cache is more challenging than managing other memory requests, such as a custom heap, for several reasons. First, fragment sizes vary more than heap allocation sizes, which fall into a few categories based on common data structures. Second, while the heap must keep growing with allocation requests, the cache can and must be bounded to limit memory expansion. This requires policies for capacity management — deciding which fragments to remove to make room for new fragments — which is more complex than the simple free list that the heap uses. Third, deleting these variable-sized fragments leads to variable-sized holes, causing fragmentation. Deletions can occur at any point for reasons of cache consistency, not just capacity, as well as during trace building when the trace head is deleted. Fourth, fragments must not be deleted during critical times such as trace building or when using stateful exception handling (Section 5.3.3), complicating cache management. And finally, cache management decisions have more of a performance impact than heap policies, since the cache contents directly dictate execution.

6.1.2 Thread-Private Versus Shared

One challenge that exists for both the heap and the code cache is supporting multiple threads. Our code cache and data structures correspond to application code, which is shared by every thread in the address space. We must choose whether our memory will be similarly shared, or whether each thread will have its own private code cache and corresponding data.

Thread-private caches have a number of attractive advantages over thread-shared caches, including simple and efficient cache management, no synchronization, and absolute addresses as thread-local scratch space (see Section 5.2.1). The only disadvantage is the space and time of duplicating fragments that are shared by multiple threads, although once fragments are duplicated they can be specialized per thread, facilitating thread-specific optimization or instrumentation.

Thread-shared caches have many disadvantages in efficiency and complexity. Deleting a fragment from the cache, either to make room for other fragments or because it is stale due to its source library being unloaded, requires ensuring that no threads are executing inside that fragment. The brute force approach of suspending all threads will ensure there are no race conditions, but is costly. Another key operation, resizing the indirect branch lookup hashtable, requires adding synchronization to the in-cache lookup routine, whose performance is critical (see Section 4.3). Even building traces needs extra synchronization or private copies of each component block in a trace-in-progress to ensure correctness.

To quantify the comparison between the two types of cache, we must know how much code is shared among threads. Naturally, it depends on the application: in a web server, many threads run the same code, while in a desktop application, threads typically perform distinct tasks. Table 6.1 shows the percentage of both basic block fragments and trace fragments that are used by more than one thread, in our desktop and server benchmarks, with an additional set of interactive workloads for our desktop benchmarks to try and drive the sharing as high as possible. Even interactively there is a remarkable lack of shared code in our desktop applications, which matches previous results [Lee et al. 1998] where desktop applications were found to have few instructions executed in any thread other than the primary thread. These results drove our design decision to use thread-private caches exclusively in DynamoRIO. Even for server applications, which have significantly more sharing, thread-private caches outperform thread-shared, and only have problematic memory

Benchmark	Threads	Basic blocks shared	Basic blocks threads/fragment	Traces shared	Traces threads/fragment
batch excel	4	0.8%	2.9	0.2%	2.1
batch photoshp	10	1.2%	4.3	1.0%	3.8
batch powerpnt	5	1.0%	2.8	0.1%	2.2
batch winword	4	0.9%	2.8	0.1%	2.1
interactive excel	10	8.4%	3.5	2.5%	3.2
interactive photoshp	17	4.5%	2.3	1.7%	2.1
interactive powerpnt	7	8.0%	2.2	10.9%	2.0
interactive winword	7	9.8%	4.1	3.3%	2.9
average desktop	8	4.3%	3.1	2.5%	2.6
IIS inetinfo	56	41.2%	11.3	67.1%	14.6
IIS dllhost	19	45.7%	4.6	27.4%	3.6
apache	257	41.8%	16.1	67.2%	7.9
sqlservr	287	44.8%	25.2	71.1%	32.6
average server	155	43.4%	14.3	58.2%	14.7

Table 6.1: Fragment sharing across threads for our desktop and server benchmarks. The thread number is the number of threads ever created, different from the simultaneous counts in Table 7.3. The batch scenarios are our benchmark workloads, while the interactive are for using the application manually in an interactive setting, which creates more threads. For each benchmark, two numbers for basic blocks and for traces are shown. The first is the percentage of fragments that are executed by more than one thread, while the second is threads per fragment, the average number of threads executing each shared fragment.

usage when running multiple applications under DynamoRIO at once. The performance results throughout this thesis are for thread-private caches, although most of the memory management solutions in this chapter, as well as DynamoRIO design decisions in the rest of the thesis, apply to both thread-private and thread-shared caches. The exception is our capacity management, which takes advantage of the efficiency of thread-private single-fragment deletion. We leave cache capacity in thread-shared caches as future work (Section 6.5).

6.2 Code Cache Consistency

Any system that caches copies of application code must ensure that each copy is consistent with the original version in application memory. The original copy might change due to dynamic mod-

Benchmark	Number of memory unmappings
excel	144
photoshp	1168
powerpnt	367
winword	345
IIS inetinfo	3688
IIS dllhost	10
apache	58577
sqlservr	26

Table 6.2: The number of memory unmappings in our desktop and server benchmarks.

ification of the code or de-allocation of memory, e.g., the unmapping of a file containing the code, such as a shared library. Unmapping of files is relatively frequent in large Windows applications, which load and unload shared libraries with surprising frequency, as Table 6.2 shows. If we only had to worry about these memory unmaps, we could partition the code cache by source region, simplifying modified fragment identification and fragment invalidation. Unfortunately we must deal with potential modification of any piece of code at any time. On most architectures, software must issue explicit requests to clear the instruction cache when modifying code [Keppel 1991], facilitating the tracking of application code changes. In contrast, IA-32 keeps the instruction cache consistent in hardware (Section 4.5), making every write to memory a potential code modification. While applications that dynamically modify code are rare, on Windows the loader modifies code sections for rebinding and rebasing (Windows shared libraries do not use position-independent code [Levine 1999]). Furthermore, re-use of the same memory region for repeated dynamic generation of code must be treated as code modification. Finally, actual self-modifying code is seen in a few applications, such as Adobe Premiere and games like Doom.

6.2.1 Memory Unmapping

Memory unmapping that affects code is nearly always unloading of shared libraries, but any file unmap or heap de-allocation can contain code. Unmapping is a relatively simple problem to solve, as it, like instruction cache consistency on other architectures, involves explicit requests to the kernel. We need only watch for the system calls that unmap files or free areas of the address space. On Linux, these are `munmap` and `mremap`; on Windows, `NtUnmapViewOfSection`, `NtFreeVirtualMemory`, and `NtFreeUserPhysicalPages`. When we see such a call, we must flush all fragments that contain pieces of code from that region. We use the same flushing scheme as for responding to memory modification.

6.2.2 Memory Modification

Unlike memory unmapping, the application does not need to issue an explicit request when writing to code. Therefore, we must monitor all memory writes to detect those that affect code. This can be done by instrumenting each write or by using hardware page protection. Page protection provides better performance since there is no cost in the common case of no memory modifications, in contrast to the always-present overhead of instrumentation, and we use it when we can, although we must use instrumentation to handle self-modifying code (Section 6.2.3).

DynamoRIO's cache consistency invariant is this: *to avoid executing stale code, every application region that is represented in the code cache must either be read-only or have its code cache fragments sandboxed to check for modifications*. DynamoRIO keeps an *executable list* of all memory regions that have been marked read-only or sandboxed and are thus allowed to be executed. The list is initially populated with memory regions marked executable *but not writable* when DynamoRIO takes control. Both the Windows and Linux executable formats (PE [Microsoft Corporation 1999] and ELF [Tool Interface Standards Committee 1995], respectively) mark code pages as read-only, so for the common case all code begins on our executable list. The list is updated as regions are allocated and de-allocated through system calls (we do not track intra-process memory allocations through calls like `malloc` – see Section 5.4 for reasons to operate at the system call layer).

When execution reaches a region not on the executable list, the region is added, but if it is not already read-only, DynamoRIO marks it read-only. If a read-only region is written to, we trap the fault, flush the code for that region from the code cache (flushing is discussed later, in Section 6.2.8), remove the region from the executable list, mark the region as writable, and then re-execute the faulting write. If the writing instruction and its target are in the same region, no forward progress will be made with this strategy. Our solution for this *self-modifying code* is discussed in the next section. We do not attempt the alternative strategy of emulating the write instruction, rather than natively executing it. While a simple matter for a `mov` instruction, there are many complex IA-32 instructions that write to memory, and we did not want to build a general emulation engine.

For error transparency (Section 3.3.5) we must distinguish write faults due to our page protection changes from those that would occur natively. When we receive a write fault targeting an area of memory that the application thinks is writable, that fault is guaranteed to belong to us, but all other faults must be routed to the application. Additionally, we must intercept Windows' `QueryVirtualMemory` system call and modify the information it returns to pretend that appropriate areas are writable. If the application changes the protection on a region we have marked read-only, we must update our information so that a later write fault will properly go to the application.

We considered alternatives to making all code pages read-only. One is to have a list of *uncacheable* memory regions. Any fragment that comes from those regions is never linked and only lives in the code cache for one execution. We rejected this solution because it cannot handle self-modifying code in which instructions later in a basic block are modified, unless all basic blocks are restricted to single instructions. Furthermore, the overhead of single-use fragments is prohibitive (Section 2.2 shows how important linking is for performance). On Windows, we considered using several unique operating system features to implement cache consistency. The `NtGetWriteWatch` system call detects written pages in a memory region labeled with a write watch flag. But it cannot be used for memory regions that have already been allocated, and it requires modifying flags the application passes to the allocation routine. Another feature is the `PAGE_GUARD` page protection flag. When a guarded page is accessed the first time, an exception is thrown and the page is made accessible. If a similar feature existed for detecting the initial write, it would reduce overhead by avoiding a separate system call to mark the page writable.

Memory modification occurs in our benchmarks that use just-in-time (JIT) compiled code (Table 6.3). These modifications are mainly to data in the same region as the generated code (*false sharing*), generation of additional code in the same region as previously generated code, or new generated code that is replacing old code at the same address. DynamoRIO cannot systematically detect what the JIT compiler is doing or whether it has invalidated code and is re-using addresses, however, and all of these situations look like modified code to us.

Benchmark	Generated code regions	Code region modifications
excel	21	20
photoshp	40	0
powerpnt	28	33
winword	20	6

Table 6.3: The number of data regions (i.e., regions not initially marked executable by the loader) that contain executed code, and the number of code region modifications, for our desktop benchmarks. The memory modifications are primarily to data stored in the same region as code.

Memory modification also occurs with trampolines used for nested function closures [GNU Compiler Connection Internals], which are often placed on the stack. As the stack is unwound and re-wound, the same address may be used for a different trampoline later in the program. DynamoRIO invalidates the first trampoline when it is written to, whether by subsequent use of the stack for data or generation of a later trampoline. Additionally, the Windows loader directly modifies code in shared libraries for rebasing [Levine 1999]. The loader also modifies the Import Address Table [Pietrek 2002] for rebinding a shared library, and this table is often kept in the first page of the code section. This means that modifications of the table look like code modifications if the entire section is treated as one region. It is difficult to determine whether a perceived code modification is being performed by the loader or not without knowing the internal data structures of the loader itself.

6.2.3 Self-Modifying Code

Read-only code pages do not work when the writing instruction and the target are on the same page (or same region, if regions are larger than a page). These situations may involve actual self-modifying code (such as in Adobe Premiere) or false sharing (writes to data near code, or generation of code near existing code). Marking code pages as read-only also fails when the code is on the Windows stack, for reasons explained below.

To make forward progress when the writer and the target are in the same region, we mark the region as writable and turn to sandboxing. Our strategy is for each fragment from a writable region to verify only that its own code is not stale, by storing a copy of its source application code. At the top of the fragment a check is inserted comparing the current application code with the stored copy, which must be done one byte at a time — comparing a hash is not good enough as a code modification could end up not changing the hash. If the code copy is different, the fragment is exited and immediately flushed. If the check passes, the body of the fragment is executed, but with an added check after each memory write to detect whether code later in the fragment is being modified. If any of these checks fails, we again exit the fragment and immediately flush it. In either flushing case we remove only the fragment in question from the cache. This technique incurs a sizable space penalty for sandboxed fragments, as they store a copy of the original application code and instrumentation code at the beginning and after each write. Even though IA-32 processors from the Pentium onward correctly handle modifying the next instruction, Intel strongly recommends executing a branch or serializing instruction prior to executing newly modified code [Intel Corporation 2001, vol. 3]. If all applications followed this, it would obviate the need to check for code modification after each memory write, but DynamoRIO avoids making any assumptions.

This strategy will not detect one thread modifying code while another is already inside a fragment corresponding to that code — the code modification will not be detected until the next time the target fragment is entered. This consistency violation is identical to that present in our consistency model relaxation for optimizing fragment invalidation (Section 6.2.6), and is discussed in full in Section 6.2.7.

Unlike UNIX operating systems, Windows does not support an alternate exception handling stack. If an exception occurs while the stack pointer is pointing to invalid or unwritable memory, the process is silently killed. Control does not reach user mode at all, as the kernel kills the process when it fails to store arguments for the exception dispatcher on the user stack, and the application has no means of recovery. Thus, we cannot mark any stack region as read-only, as a resulting write fault will kill the process. When we add a code region on the stack to the executable list, instead of marking it read-only we mark it for sandboxing. To identify the stack, we consider both the current stack pointer and the thread's initial assigned stack, although the stack pointer could change at any time, spelling disaster if it later points to memory we made read-only. This is a pathological case, the intersection of two rare events: stack pointer region changes and writable code regions. Future work could address this by watching writes to the stack pointer (optimizing checks for the common writes of stack pushes and pops), which should have a relatively low overhead.

Figure 6.4 shows the performance impact of our sandboxing implementation when it is applied

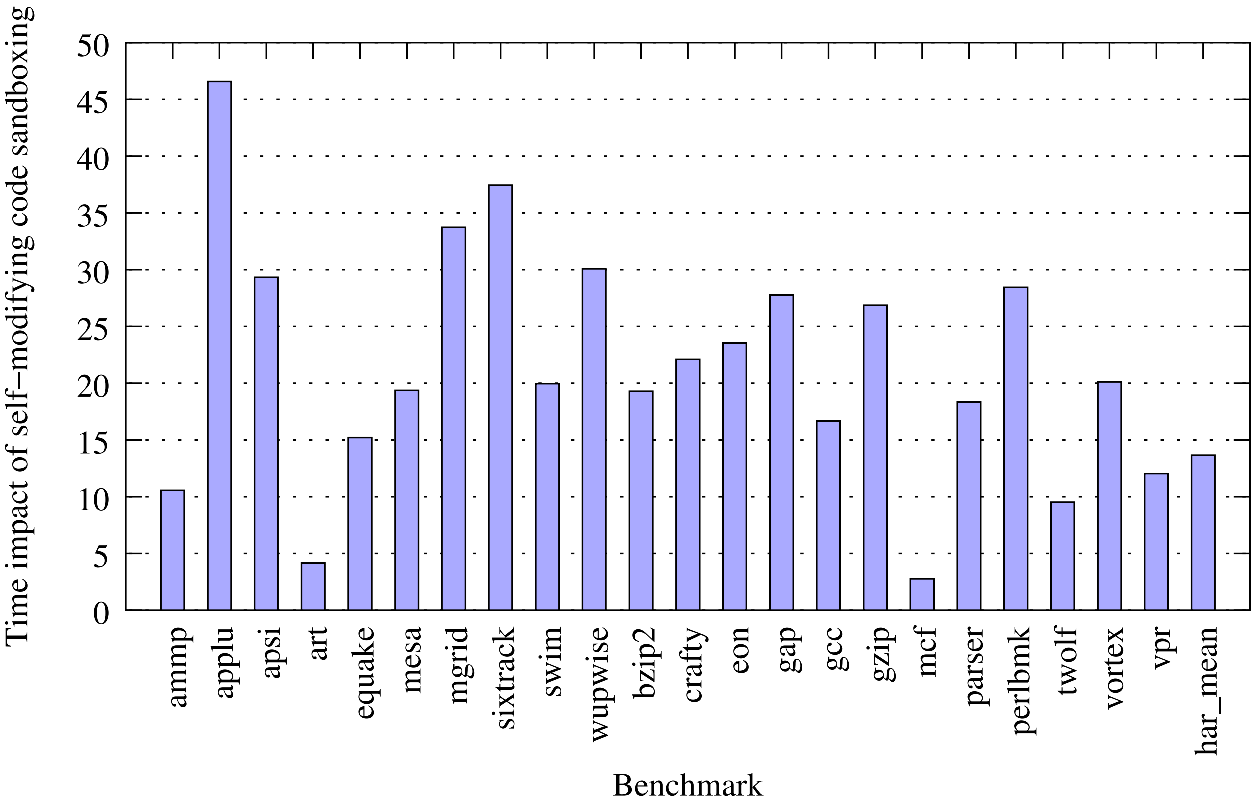

Figure 6.4: Performance impact of self-modifying code sandboxing applied to all code in the SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmarks. Fortunately, self-modifying code is rare, though we must also use it for cases of false sharing due to our page protection granularity.

to every basic block. Not surprisingly, it slows the program down by an order of magnitude, although we have not optimized our sandboxing instrumentation (due to the rarity of use).

Sandboxing may be a better general choice than making pages read-only for cases of false sharing, where many writes to data on the same page can be more expensive with page protection than the cost of sandboxing the code, depending on how frequently executed the code is. The next section further discusses performance optimizations.

6.2.4 Memory Regions

The previous sections talk about *memory regions*. For utilizing page protection, regions must be at least as large as pages, though they can be smaller for sandboxing. If regions are too large, a single code modification will flush many fragments, which is expensive. On the other hand, small regions create a longer executable list and potentially many more protection system calls to mark code as read-only. Large regions work well when code is not being modified, but small regions are more flexible when small pieces of scattered code are being occasionally modified. When regions are frequently modified, sandboxing may be best choice. Another consideration is the pattern of code modification. If code modification and subsequent execution are two separate phases, large regions are best. But, if code is modified and immediately executed, repeatedly, small regions are good for separating the writer from the target and avoiding unnecessary flushing.

DynamoRIO uses an adaptive region granularity to fit regions to the current pattern of code modification. DynamoRIO's initial region definition is a maximal contiguous sequence of pages that have equivalent protection attributes. Since nearly all code regions are read-only to begin with and are never written to, these large regions work well. On a write to a read-only region containing code, we split that region into three pieces: the page being written (which has its fragments flushed and is marked writable and removed from our executable list), and the regions on either side of that page, which stay read-only and executable. If the writing instruction is on the same page as the target, we mark the page as self-modifying. DynamoRIO's executable list merges adjacent regions with the same properties (the same protection privileges, and whether self-modifying), resulting in an adaptive split-and-merge strategy that maintains large regions where little code is being modified and small regions in heavily written-to areas of the address space.

We could also mark a page as self-modifying if it is written to more times than executed from. As mentioned in the previous section, self-modifying code is the best choice for a page primarily used for data that has a few pieces of rarely-executed code on it. Adapting our algorithm to switch to self-modifying code once a threshold of writes is reached is future work.

6.2.5 Mapping Regions to Fragments

Whatever region sizes we use, we must be able to map a region to a list of fragments in the code cache containing code from that region. Since we elide unconditional control transfers (see Section 2.4), even a basic block might contain code from several widely separated regions.

Before mapping a region to fragments, a check that the region actually contains code that has been executed saves unnecessary work. Since we have to worry about code being removed on any unmapping, many potential flushes are only a data file being unmapped. We test for any overlap between the unmap region and the list of executable regions. Another optimization, for thread-private caches, is to store a list of executed-from memory regions for each thread, which can be quickly consulted to determine whether a thread needs to have any of its fragments flushed.

Once these initial region overlap checks indicate that there are fragments to flush, we must identify the fragments in the target region. The solution we tried initially was similar to our context translation solution (see Section 3.3.4): since the original code is read-only, we do not need to store anything and can instead re-create each fragment to determine which regions it touches. One problem with this is self-modifying code, whose application region is not read-only. For self-modifying code we chose to never follow unconditional control transfers, never build traces, and never extend a basic block off the edge of a self-modifying region into an adjacent non-self-modifying region, allowing us to test only the starting address of a self-modifying fragment. Surprisingly, the overhead from re-creating every fragment was prohibitive. Even though flushing is a relatively rare event, the overhead shows up on our desktop benchmarks, as Figure 6.5 (the *Recreate All* dataset) shows. We tried optimizing this scheme by recording, for each basic block built, whether it occupies more than one page. By testing this flag we could avoid the re-creation for all fragments that only touch one page, as we could check only their starting addresses. However, a full one-fifth of basic blocks and one-half of traces occupy more than one page, and Figure 6.5's *Recreate Multi-Page Fragments* dataset shows that the overhead was still too high. Next, we tried not eliding unconditionals and never walking across page boundaries, allowing us to use only the starting address for each fragment and never having to recreate. As Figure 6.5's *Check Start Address Only* dataset shows, even this scheme shows some slowdown. This is because it, like the previous techniques, must consider every single fragment — hundreds of thousands of them — in order to find the handful

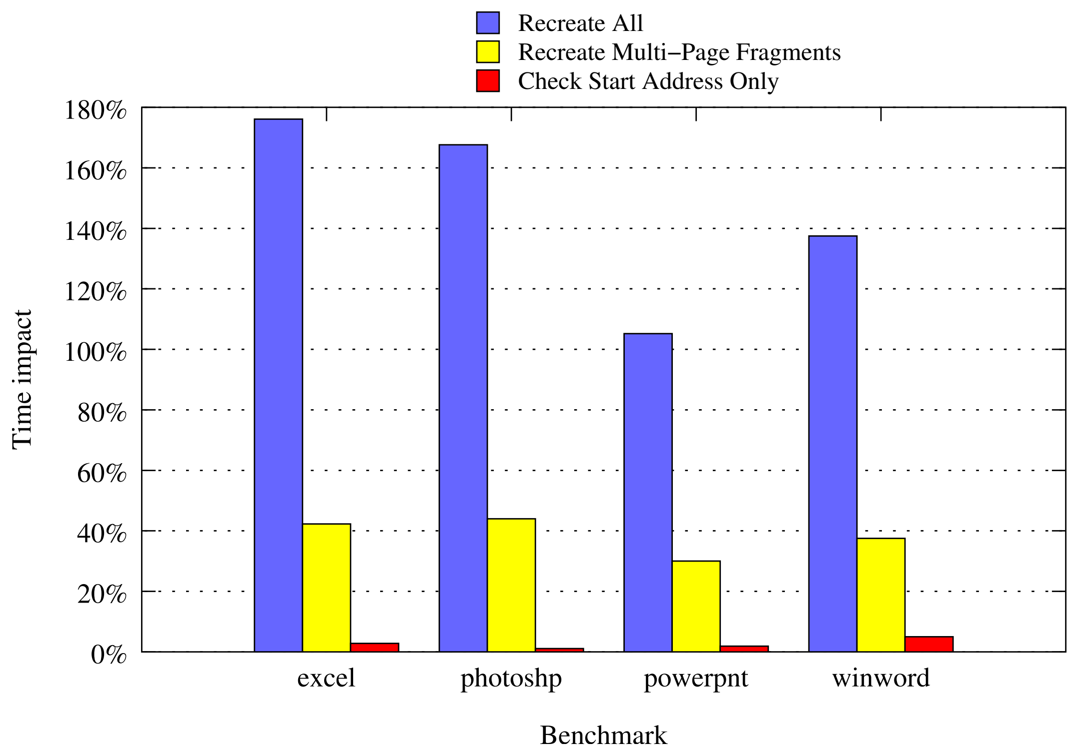

Figure 6.5: Performance impact of schemes for mapping regions to fragments on our desktop benchmarks, versus our chosen solution of per-region fragment lists.

that are in the target region.

The solution adopted by DynamoRIO is to store a list of fragments with each executable list region entry (for thread-private caches, with the thread-private executable list entries). To save memory we embed linked list pointers (see Section 6.4.2) in the fragment data structure itself and use it as the entry in the first region that a fragment touches. Separate dedicated data structures called `MultiEntry` are placed in the fragment list for each additional region the fragment occupies, with all entries for a single fragment chained together in a separate linked list that crosses the region lists. These lists are set up when a basic block or a trace is created, with each new page encountered, either through eliding an unconditional or simply walking off the edge of the previous page, triggering a potential addition of a new region. With these per-region fragment lists, flushing simply walks the list of fragments that must be flushed, and ignores all other fragments. This ties flushing to the region granularity on the executable list, as we must flush an entire region at a time. Still, this is an improvement over most other systems (Section 10.2.5), which flush their entire caches on any cache consistency event.

6.2.6 Invalidating Fragments

Even when using thread-private code caches, a memory unmapping or code modification affects all threads' caches, since they share the same address space. This is the one operation on thread-private caches that requires synchronization.

The actual invalidation of modified code in each thread's code cache must satisfy the memory consistency model in effect. In contrast to systems like software distributed shared memory [Brian N. Bershad and Sawdon 1993, Carter et al. 1991, Li and Hudak 1989], a runtime code manipulation system cannot relax the consistency model by changing the programming model for its target applications. DynamoRIO aims to operate on arbitrary application binaries, which have already been built assuming the underlying hardware's consistency model. Any significant relaxation DynamoRIO implements may break target applications and needs to be considered carefully.

On IA-32, to support all applications, we must follow sequential consistency [Lamport 1979]. To do so requires immediate invalidation of all affected fragments from the code cache of every thread. Otherwise, stale code could be executed. Because any code could be modified at any time, and there is no efficient mechanism to identify where a thread is inside the code cache to the granularity of a fragment, the only way to do this is to use a brute-force approach: suspend all threads and forcibly move those that are executing inside of to-be-invalidated code. Threads may have legitimate reasons to be executing inside of a to-be-deleted region, as that region may contain data that was written to instead of code (false sharing). No thread can be resumed until the target code is not reachable inside the code cache. If writes to code regions are frequent, suspending all threads is too heavyweight of a solution.

We have come up with a relaxation of the consistency model that allows a more efficient invalidation algorithm, which we call *non-precise flushing*. Our consistency model is similar to weak consistency [Dubois et al. 1986] and release consistency [Gharachorloo et al. 1990] in that it takes advantage of synchronization properties of the application. However, we distinguish between code and data, as we only need to worry about consistency of code. The key observation is that ensuring that no thread enters a stale fragment can be separated from the actual removal of the fragment from the cache. The first step can be done atomically with respect to threads in the code cache by unlinking the target fragments and removing them from the indirect branch lookup table(s). The actual deletion of the fragments can be delayed until a safe point when all threads in question have left the code cache on their own. This prevents any new execution of stale code, leaving only the problem of handling a thread currently inside of a stale fragment. Here we turn to our relaxed consistency model. If the application is properly synchronized, and every application synchronization operation terminates its containing fragment, then we can always let a thread finish executing a to-be-deleted fragment without actually executing stale code in a manner that could not occur natively. For example, if thread A modifies some code, then thread B cannot legitimately execute the newly modified code until it has synchronized with A, which requires exiting its current fragment. If all stale fragments are unlinked, then B will not be able to enter or re-enter any stale fragment after the synchronization operation. This consistency model is essentially sequential consistency when only considering data or only considering code, but weak consistency when considering all of memory. Code writes will never be seen out of order, and of course data writes are not affected at all. The only re-ordering with respect to sequential consistency that might occur is between a data write and a code write.

6.2.7 Consistency Violations

This consistency relaxation matches the limitations of our self-modifying sandboxing (Section 6.2.3), which employs a check at the top of each fragment, rather than unlinking, to bound the stale code window to a single fragment body. If we could identify all application synchronization operations and never build fragments across them, neither our consistency model relaxation nor our sandboxing method would break any application in a way that could not occur natively. However, we cannot efficiently identify all possible synchronization operations. For example, an implicitly atomic single-word operation can be used as a condition variable, and we cannot afford to break fragments on every memory access on the chance that it might be a condition variable. Fortunately, for synchronizing more than two threads, an explicitly atomic operation that locks the memory bus (using the `lock` prefix or the `xchg` instruction) is required. Thus, if we break our fragments at such instructions, we should be safe for all but certain two-thread-only synchronization code.

The cases that do break our model are pathological, involving one thread waiting for another to write to code before executing it. Given that Intel discourages executing modified code without a branch or serializing instruction first [Intel Corporation 2001, vol. 3], we could relax our implementation further, only breaking fragments on loops and system calls, and still catch the vast majority of synchronization cases since synchronization is usually separate enough from any transition to modified code that it should be in a separate fragment. The only violating case is a trace (since it must inline a conditional branch) that reads a condition variable prior to jumping to some target code, with another thread setting that condition variable after modifying the code. Not building traces that bridge compiled code modules and generated code regions further narrows the window in which stale code can be executed.

The most disturbing aspect of consistency violations is that there is no way to detect when stale code is executed. Even if the method were extremely costly when stale code is encountered, if it cost little in the common case of no modified code being executed it would not have a significant performance impact. Unfortunately there is no such method, and DynamoRIO cannot tell whether its cache consistency assumptions have been violated.

6.2.8 Non-Precise Flushing

To implement this *non-precise flushing* that allows a delay between the flush and the actual deletion of the flushed fragments, we must accomplish only one thing at the time of the flush: prevent any new executions of the targeted fragments. This requires unlinking and removing them from the indirect branch lookup table. We then add the region being flushed to a queue of to-be-deleted fragments, for deletion when the code cache is free of threads. With this unlinking strategy, atomic unlinking is required even with thread-private code caches. Our linking is designed to be a single write, which can be made atomic by aligning it to not straddle a cache line boundary, or by using the `lock` prefix (see Section 2.2). The hashtable removal must also be safe to be done while another thread is examining the table from the code cache, which may incur a performance impact for thread-shared caches.

Even for the unlinking stage, we must synchronize with each thread. Our synchronization model centers around whether a thread might be reading or modifying linking information, memory region information, or trace information for the fragments in question, or not. For the most part this boils down to whether the thread is in the code cache or in DynamoRIO code, but there are some exceptions, such as most system call handlers, which consist of DynamoRIO code but do not access linking information.

The thread that is performing the flush sets a flag to prevent new threads from being created or old threads from dying and then marches through the thread list, checking whether each thread is accessing link information or not. The majority of threads are in the code cache, and thus not accessing link information, but if one is, the flusher must set a flag and wait for the thread to reach a non-linking state. For thread-shared caches, all threads must by synchronized with simultaneously before acting on the target fragments, while thread-private caches require only one thread at a time. Once the target thread(s) are at a safe point, the flusher checks whether they have any fragments in the flush region (using the technique of Section 6.2.5), and if so, it unlinks them and removes them from the hashtable, adding them to a queue of to-be-deleted fragments. As each thread in the code cache (only one for thread-private, of course) exits, it checks the queue and if it is the last thread out performs the actual deletion of the fragments. Thread-shared caches can use a barrier preventing re-entry to bound the time until all threads exit the cache, or periodically suspend all threads (with a low frequency this technique can perform well — it is when forced to use it on every consistency event that suspension performance is problematic).

Other caching systems either do not support threads or use the brute-force suspend-all-threads algorithm for any fragment deletion (see Section 10.2.5 for discussion of cache consistency in related systems). These systems often do not fully handle cache consistency, and so only perform deletions on rarer cache capacity events. Consistency events are much more common in programs that use dynamically-generated code, and a more efficient solution, like ours, is needed.

6.2.9 Impact on Cache Capacity

Cache consistency has a significant impact on general cache management. Arbitrary fragments can be invalidated at any time, leading to holes in the cache, which complicate multi-fragment deletion. Deleting in batches is ordinarily more efficient, since a contiguous group can be deleted at once, and if memory unmappings were the only type of consistency event this would work well, as batch deletion groups could be organized to match code regions. But memory modification events result in fine-grained fragment invalidation, and a fragment invalidation that occurs in the middle of a batch region requires either evicting the entire batch or splitting it up. The existence of numerous memory modification events in modern, dynamic applications makes single-fragment deletion the best choice for thread-private caches, for which it can be efficiently performed.

Consistency holes in the code cache are often scattered, causing fragmentation. If no capacity policy or fragmentation solution is in place to fill in these holes rather than adding to the cache, repeated cache consistency events can end up causing unlimited growth of the code cache. The next section describes our cache capacity management, including how we address this problem by filling in holes created by consistency invalidations whenever we can.

6.3 Code Cache Capacity

For executing a single application in isolation, there may be no reason to limit the code cache size. However, when executing many programs under DynamoRIO simultaneously, memory usage can become problematic, and we can reduce it significantly by imposing a bound on the code cache size. Additionally, as mentioned in Section 6.2.9, cache consistency fragment invalidations can cause unbounded cache growth in the absence of a fragmentation solution. Of course, cache bounds come with a performance cost, and the trick is to pick the bound with the best space and time tradeoff. Two problems must be solved: how to set an upper limit on the cache size, and how to choose which fragments to evict when that limit is reached. Unlike a hardware cache, a software code cache can be variable-sized. This flexibility makes it possible to tailor a different upper limit for each application, and for that limit to change as the application moves through different phases.

Nearly every system with a software code cache uses a hardcoded size limit, and when it is reached, the entire cache is flushed. The limit is set generously, and it is assumed that it will rarely be reached (see Section 10.2.6 for a complete discussion of cache capacity in related systems). This may work when executing a benchmark suite like SPEC CPU [Standard Performance Evaluation Corporation 2000], but when targeting disparate applications like desktop programs, the value of a cache adaptively sized for the application at hand is apparent. Different programs run vastly different amounts of code, and a single program's code cache needs may change during its execution.

This section first discusses fragment eviction policies for use with whatever size limit is chosen (Section 6.3.1). We then examine how performance is affected when the code cache is limited to a fraction of the size used when no limit is in place, to drive home the importance of adaptive sizing (Section 6.3.2). Next we present our novel algorithm for adaptively sizing the cache

(Section 6.3.3), followed by how we subdivide the cache into units in order to support flexible cache sizes (Section 6.3.4). Finally, we present a simple method for compacting the code cache (Section 6.3.5).

6.3.1 Eviction Policy

Whatever limit is placed on the size of the code cache, a policy is needed to decide which fragments to evict to make room for new fragments once the size limit is reached. Hardware caches typically use a least-recently-used (LRU) eviction policy, but even the minimal profiling needed to calculate the LRU metric is too expensive to use in software. DynamoRIO uses a least-recently-created, or first-in-first-out (FIFO), eviction policy, which allows it to treat the code cache as a circular buffer and avoid any profiling overhead from trying to identify infrequently-used fragments. Furthermore, a FIFO policy has been shown to be comparable to other policies such as LRU or even least-frequently-used (LFU) in terms of miss rate in cache simulation studies of software code caches [Hazelwood and Smith 2002].

Figure 6.6 illustrates DynamoRIO's FIFO replacement. To make room for a new fragment when the cache is full, one or more contiguous fragments at the current point in the FIFO are deleted. This requires single-fragment deletion, which we already must support for cache consistency. If un-deletable fragments are encountered (for example, from trace building, as discussed in Section 2.3.2), the current FIFO point skips over them and the process repeats with a new target victim until enough contiguous space is found for the fragment being added. Hazelwood and Smith [2003] refer to our implementation as "pseudo-circular" because of this re-setting of the FIFO upon encountering an un-deletable fragment. If there is empty space after deleting fragments to make room for a new fragment (due to differences in fragment size), that space will be used when the next fragment is added — that is, the FIFO pointer points at the start of the empty space. By deleting adjacent fragments and moving in a sequential, FIFO order, fragmentation of the cache from capacity eviction is avoided.

Two other sources of cache fragmentation are deletion of trace heads as each trace is built (Section 2.3.2) and cache consistency evictions (Section 6.2), which are surprisingly prevalent as every shared library that is unloaded results in many fragments being deleted from the cache. To combat these types of fragmentation, we use *empty slot promotion*. When a fragment is deleted

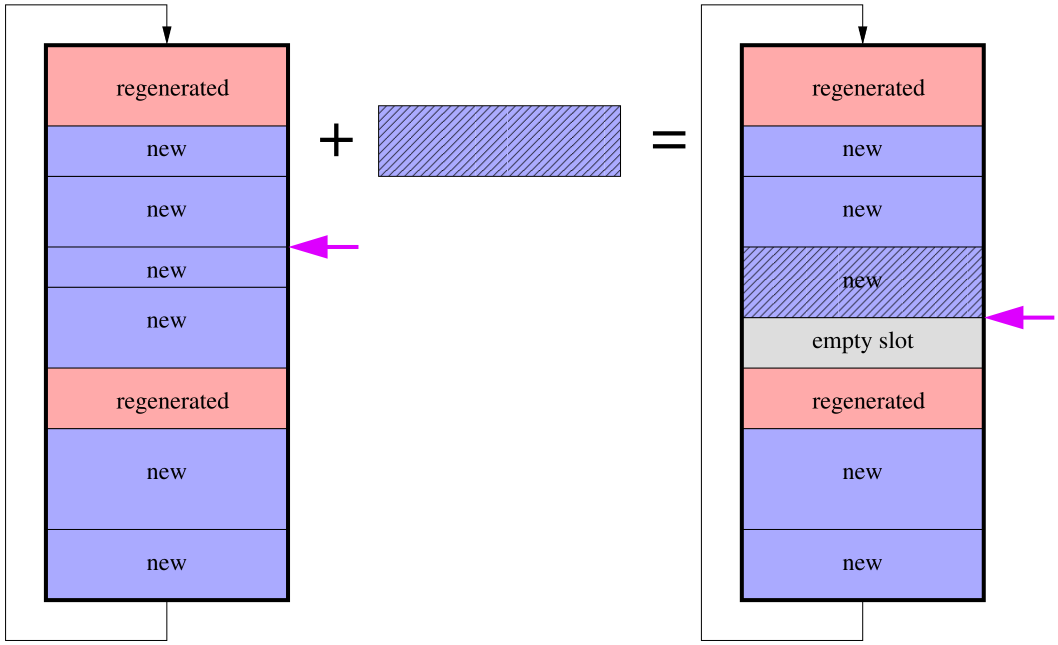

Figure 6.6: Our FIFO fragment eviction policy. The cache can be thought of as a circular buffer, with a new fragment added at the current head, displacing enough old fragments to make room for it. The figure also shows fragments being marked as either regenerated or new, which drives our adaptive working-set-size algorithm described in Section 6.3.3.

from the cache for a non-capacity reason, the resulting empty slot is *promoted* to the front of the FIFO list and will be filled with the next fragment added to the cache. To support empty slot promotion we must use a logical separation of the FIFO from the actual cache address order, as described in Section 6.3.4. Logical separation is also a must for treating multiple memory allocations as a single cache. Hazelwood and Smith [2003] propose to ignore empty slots and simply wait until wrapping around in the FIFO to fill the hole. The problem with this is that it does not work without a cache size limit. We want to support unlimited cache sizes, which offer the best performance when running a single application. By supporting empty slot promotion we allow flexibility in code cache sizing policies.

With empty slots promoted to the front of the FIFO, we must decide whether a fragment being added should only be placed in an empty slot if there is room or whether adjacent fragments in the cache should be removed to make space, just like with an actual fragment at the front of the FIFO. The argument for not removing empty slot neighbors is that it is a disruption of the FIFO order. However, FIFO is a heuristic, not an eviction oracle that should never be violated.

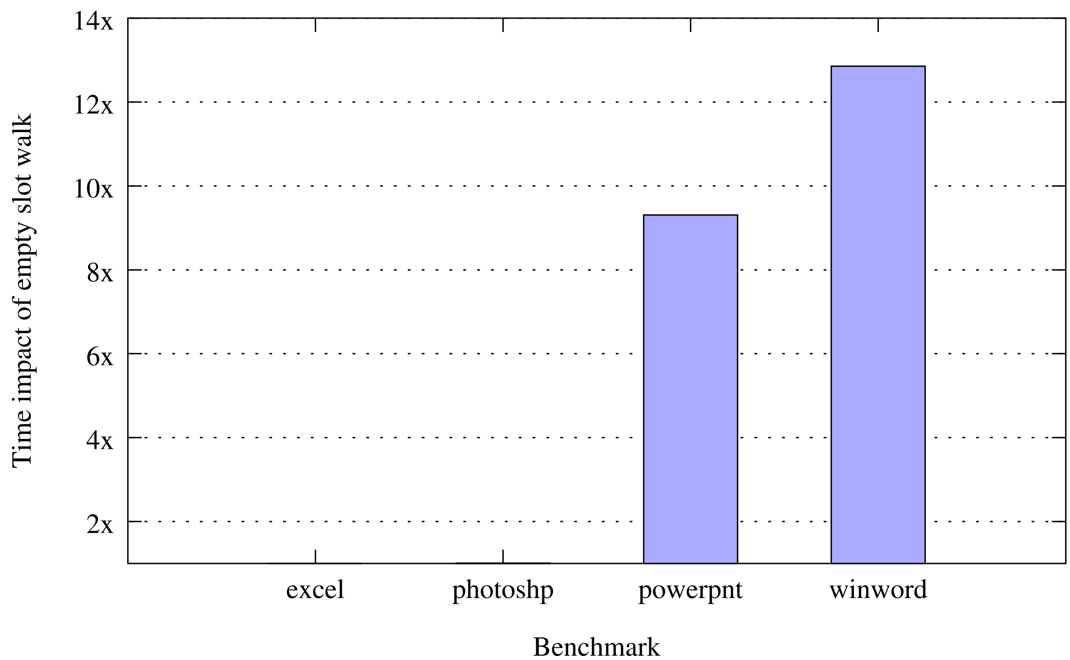

Figure 6.7: Performance impact of walking the empty slot list when the trace cache is limited to one-sixteenth its unlimited size for our desktop benchmarks. Each benchmark has a trace cache size threshold below which the empty walk causes significant performance degradation.

Furthermore, not forcing fragments into empty slots can result in a long chain of small empty slots that must be checked every time a fragment is added, causing performance degradation. We observe prohibitive slowdowns from this phenomenon when the trace cache is limited and traces are repeatedly regenerated. The holes from deleting trace heads accumulate until a long empty slot list perpetually slows down fragment addition (Figure 6.7). We tried breaking the empty slot list into several lists organized by size to mitigate the linear walk slowdown, but it did not improve performance appreciably. To avoid this problem DynamoRIO forces fragments into empty slots.

Independent of other factors, deleting groups of fragments all at once for cache capacity has better performance than single-fragment deletion [Hazelwood and Smith 2004]. However, cache consistency events on modern applications are frequent enough that only supporting large deletions would empty the cache. Furthermore, using single-fragment deletions for consistency thwarts any batch flushing used for capacity, as batch flushing requires groups of fragments to form single allocation and de-allocation units with no individual members deleted separately, and any fragment may be invalidated at any time for consistency reasons.

Benchmark	Basic block cache (KB)	Trace cache (KB)
ammp	50	77
applu	85	195
apsi	84	241
art	26	36
equake	41	72
mesa	82	76
mgrid	61	84
sixtrack	230	355
swim	52	61
wupwise	62	78
bzip2	34	47
crafty	151	278
eon	275	199
gap	153	370
gcc	684	1575
gzip	30	40
mcf	35	40
parser	101	278
perlbmk	323	502
twolf	109	308
vortex	392	459
vpr	96	111
excel	2618	608
photoshp	5521	3371
powerpnt	4242	2259
winword	3135	1126
average	718	494

Table 6.8: Code cache space with unlimited cache size, with indirect branch lookup inlining turned on for traces but not for basic blocks (see Section 4.3.1). The sizes of the main thread's caches are given for each desktop benchmark (the other threads' caches are quite small), and of the longest run for each SPEC CPU2000 benchmark.

6.3.2 Cache Size Effects

The code cache space used by our benchmarks, when the cache size is unlimited, is shown in Table 6.8. To study the performance effects of limited cache size, we imposed a hard upper bound equal to a fraction of the unlimited cache space used by each benchmark. Figure 6.9's resulting per-

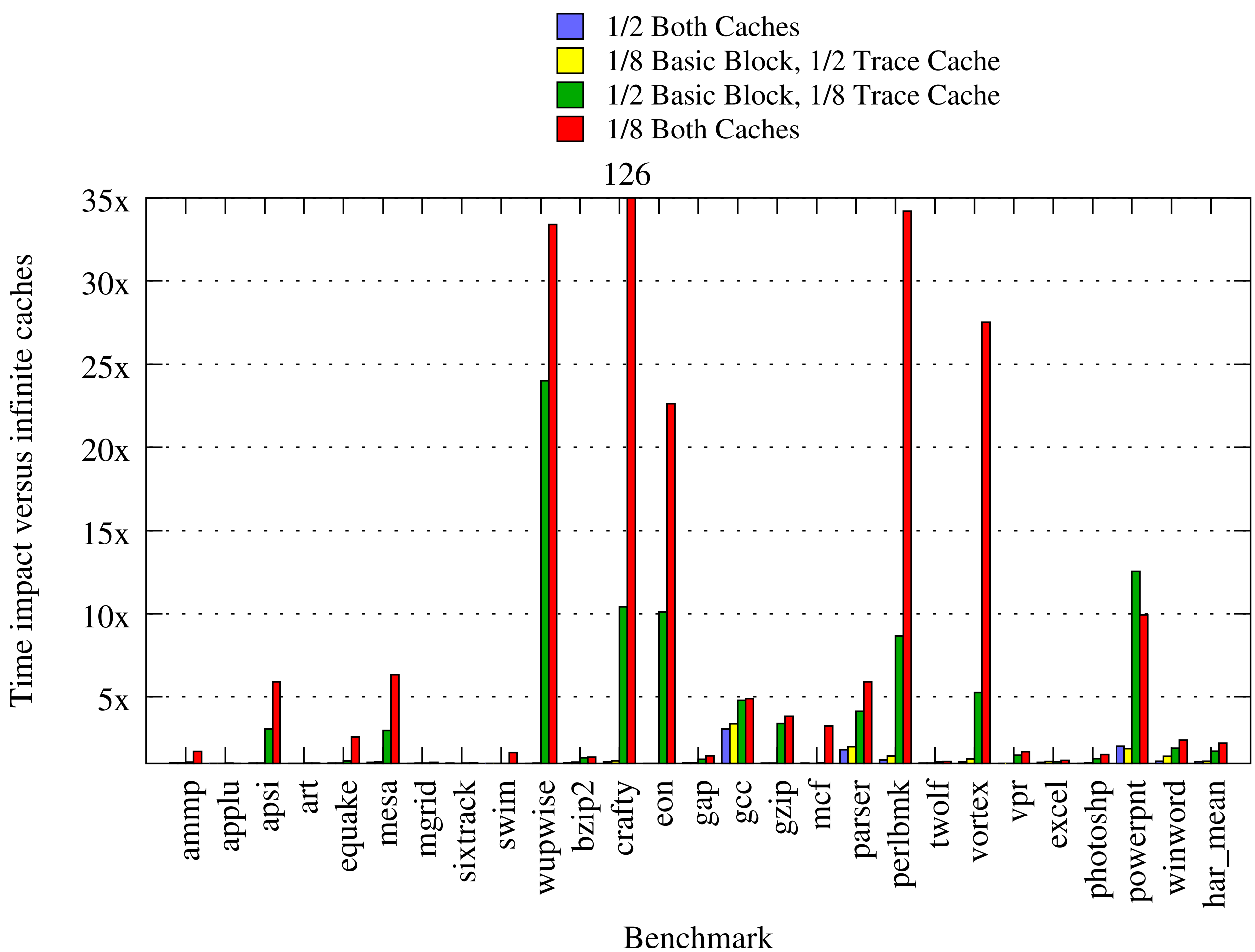

Figure 6.9: Performance impact of shrinking the basic block cache and the trace cache to one-half and one-eighth of their largest sizes, versus unlimited cache sizes.

formance numbers show that performance can be affected in extreme ways, with many slowdowns of an order of magnitude or more. Our first-in-first-out fragment replacement policy (Section 6.3.1) was used to choose which fragment to evict to make room for a new fragment. While a policy that uses profiling (ignoring the overhead of such profiling that makes it unsuitable for runtime use), such as least-recently-used (LRU) or least-frequently-used (LFU), may perform slightly better by keeping valuable fragments longer, the extreme slowdowns exhibited at low cache sizes will be present regardless of the replacement policy due to capacity misses from not fitting the working set of the application in the cache.

The results indicate that the trace cache is much more important than the basic block cache, as

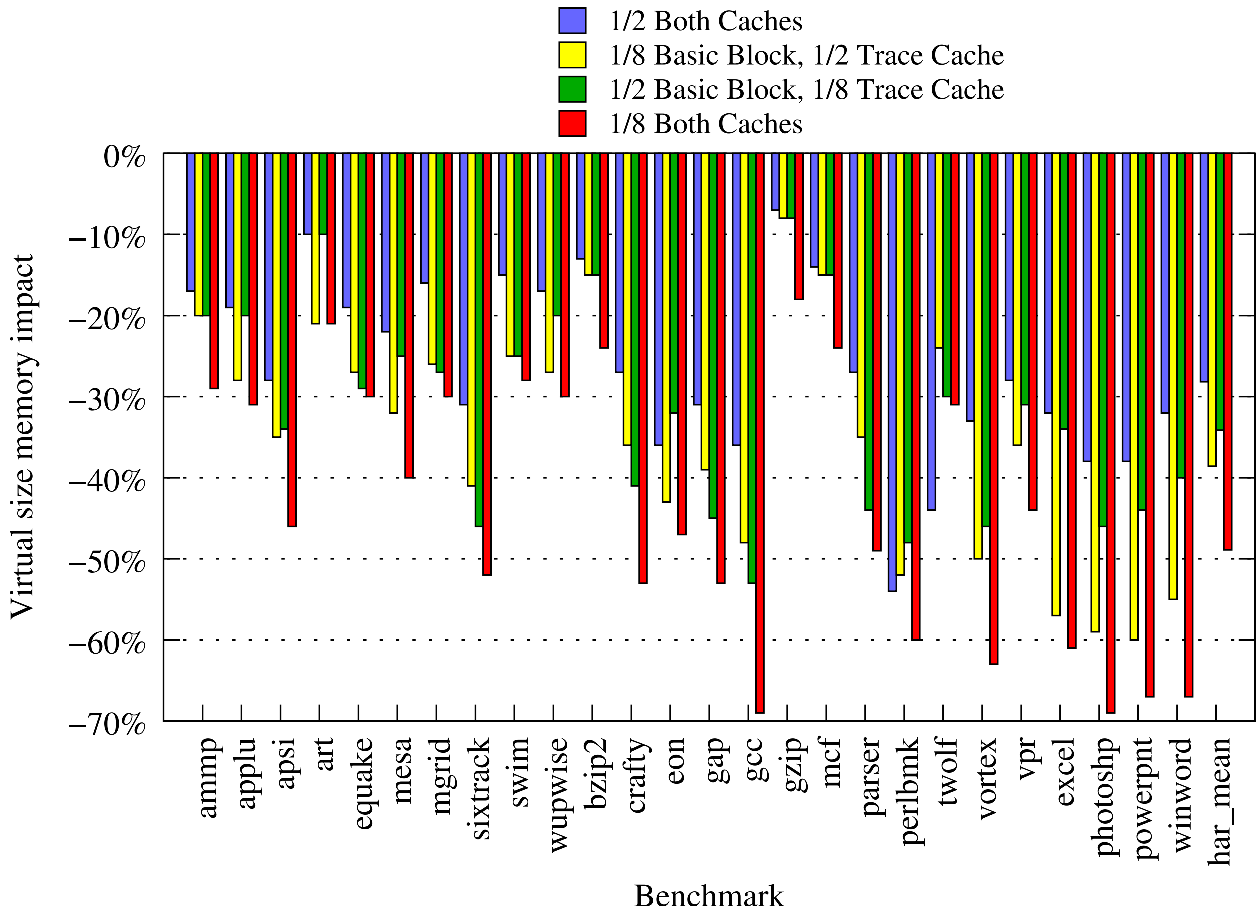

Figure 6.10: Virtual size memory impact of shrinking the basic block cache and the trace cache to one-half and one-eighth of their largest sizes. The graph shows the difference in the *additional* memory that DynamoRIO uses beyond the application's native use, not the difference in total memory used by the process.

expected. Losing a trace is a more serious matter than having a basic block deleted from the cache, as it takes much longer to rebuild the trace. Restricting both caches to one-eighth of their natural sizes results in prohibitive slowdowns for several of the benchmarks, due to thrashing. Shrinking the caches affects each application differently because of differing native behavior. Some of these applications execute little code beyond the performance-critical kernel of the benchmark, and cannot handle space constraints as much as benchmarks that contain much initialization code or other code that adds to the unlimited cache usage but is not performance-critical. Thus, unlimited cache usage is not a good metric to use for sizing the cache.

Figure 6.10 shows the overall (DynamoRIO's code cache and heap combined) memory reduc-

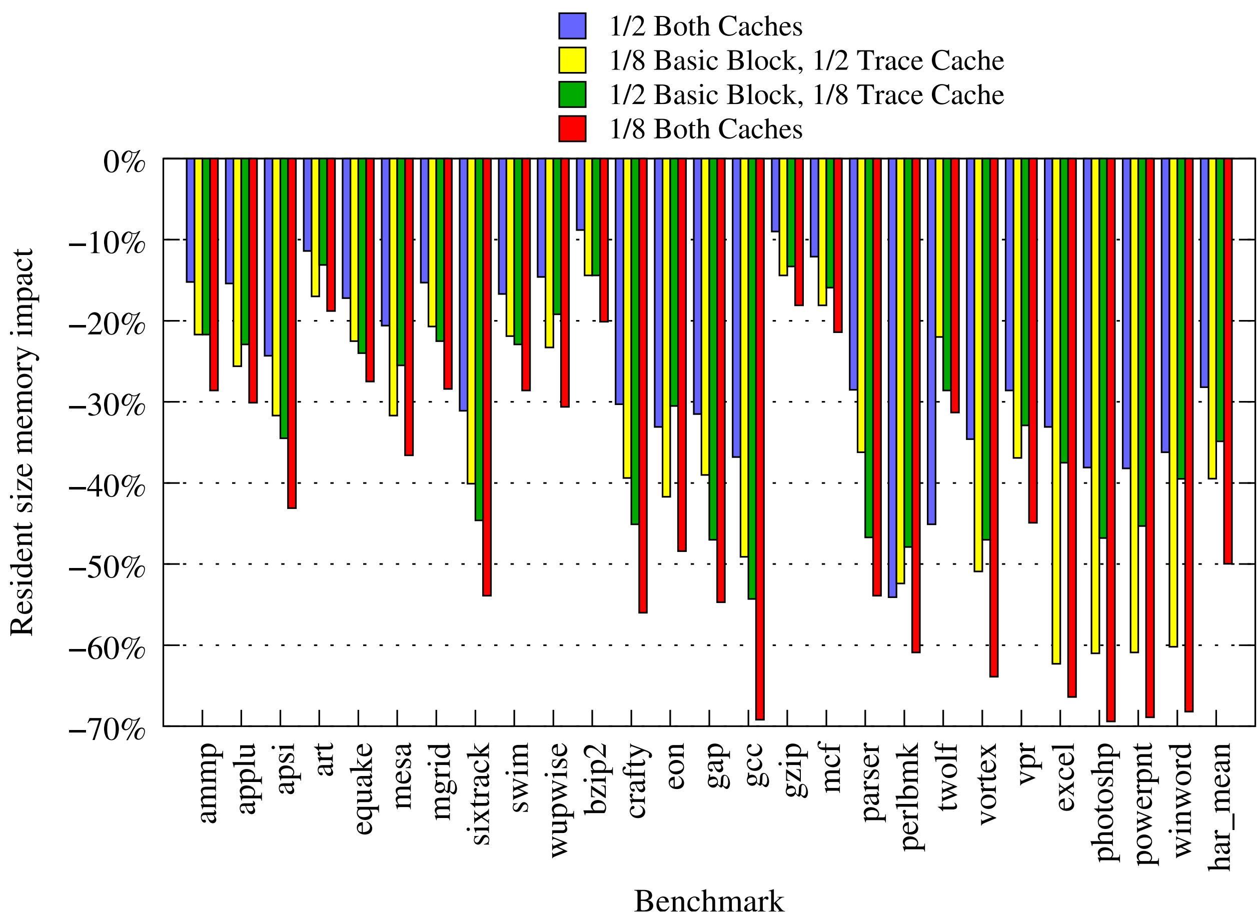

Figure 6.11: Resident size memory impact of shrinking the basic block cache and the trace cache to one-half and one-eighth of their largest sizes. The graph shows the difference in the *additional* memory that DynamoRIO uses beyond the application's native use, not the difference in total memory used by the process.

tion achieved when limiting the code cache size, in terms of total virtual size, while Figure 6.11 shows the reduction in working set size (the amount resident in physical memory). The reduction in both is about the same, reaching as high as two-thirds. However, the performance impact required to reach such drastic cuts via hard cache limits is not acceptable. Section 6.3.3 explores how to reduce memory without such a loss in performance.

When the basic block cache is restricted, trace building can be affected if trace heads are deleted before they become hot enough to trigger trace creation. As mentioned in Section 2.3.2, we use *persistent trace head counters*, storing the count for a deleted trace head so that when it is re-created it can pick up where it left off. Figure 6.12 shows the performance impact of persistent trace head

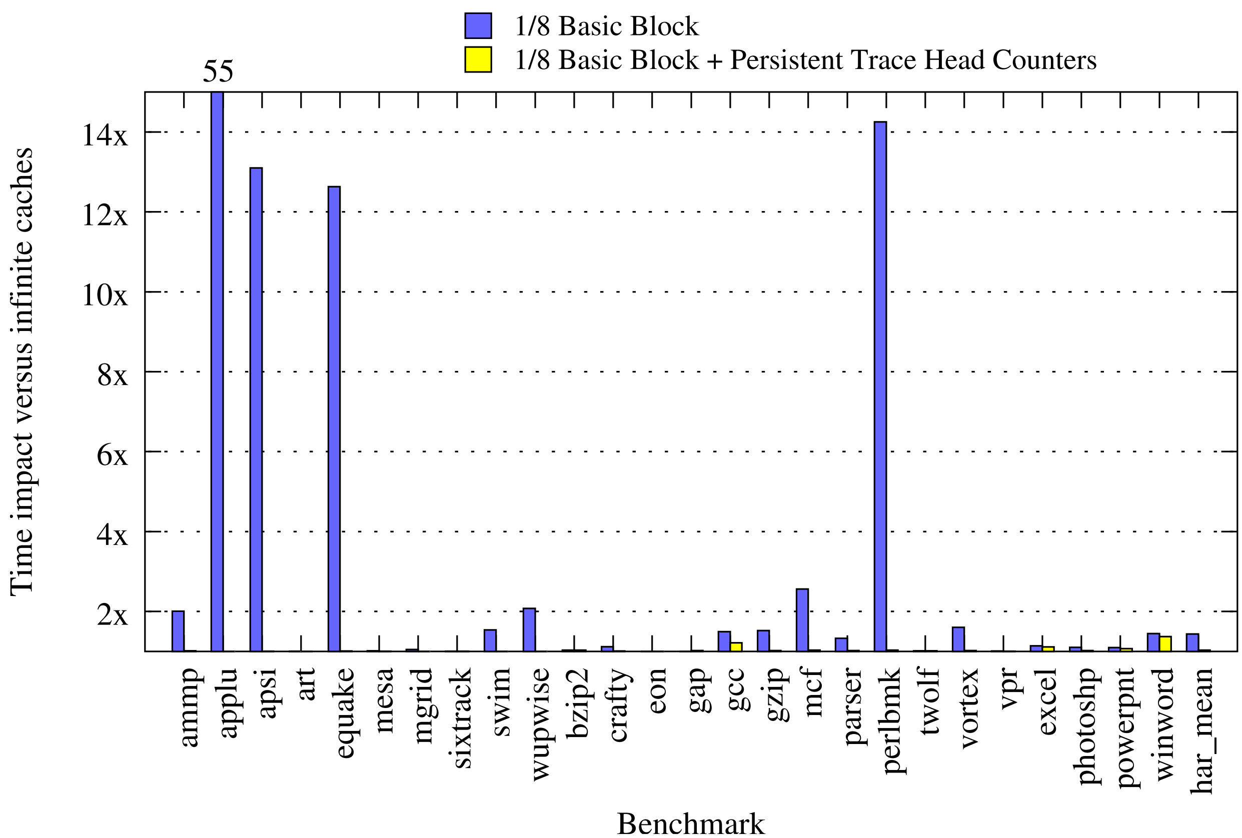

Figure 6.12: Performance impact of persistent trace head counters when the basic block cache is one-eighth of its largest size. Without persistence, performance is prohibitively poor for several benchmarks, even worse than when limiting the trace cache as well as the the basic block cache, as a comparison with Figure 6.9 shows. With persistence, shrinking the basic block cache only has less than a two times slowdown for all benchmarks.

counters when the basic block cache is restricted to one-eighth its unlimited size (the trace cache is left unlimited here to eliminate effects of deleted traces). Not using persistent counters causes drastic performance loss when the basic block cache is limited in a number of benchmarks.

6.3.3 Adaptive Working-Set-Size Detection

We have developed a novel scheme for automatically keeping the code cache an appropriate size for the current working set of the application, to reduce memory usage while avoiding thrashing. In addition to removing requirements for user input to set cache sizes, our dynamically adjusted limit allows for applications with phased behavior that will not work well with any hardcoded limit.

Section 6.3.2 showed that unlimited cache usage is not a good metric to use for sizing the cache, because it is skewed by non-performance-critical code such as initialization sequences. The insight of our algorithm is that it is fine to throw such code away, since it may only be used once. Operating at runtime, we do not have the luxury of examining future application behavior or of performing extensive profiling — we require an incremental, low-overhead, reactive algorithm. Our solution is a simple method for determining when to resize a cache, and could be applied to a simple one-cache setup or to each cache in a generational cache system [Hazelwood and Smith 2003]. Generational caches move frequently-used fragments to successively later caches while earlier generations are replaced by new code. While they may be useful for separating valuable code by adding more layers than DynamoRIO's basic blocks and traces, they require continuous profiling that can be detrimental in a runtime system and do not solve the working set sizing problem as they still require a sizing scheme for each cache.

Our sizing technique revolves around measuring the ratio of *regenerated* fragments to *replaced* fragments. We begin with a small cache, and once it fills up, we incorporate new fragments by removing old fragments using an eviction policy (ours is a first-in, first-out policy that avoids expensive profiling and utilizes our single-fragment deletion power, as described in Section 6.3.1). The number of fragments removed is the *replaced* portion of the ratio. We record every fragment that we remove from the cache by setting a flag in the data structure used for proactive linking (which contains information on all fragments, whether currently in the cache or not): the `FutureFragment` data structure. When we add a new fragment, we check to see whether it was previously in the cache (a capacity miss, as opposed to a cold miss). If so, we increment our count of *regenerated* fragments. Figure 6.6 illustrates the marking of fragments as new or regenerated. If a significant portion of new fragments are regenerated, the cache should be larger than it is, so if the ratio is over a certain threshold we allow the cache to be resized but otherwise force the cache to remain at its present size. The periodic ratio checks allow us to adapt to program behavior only when it changes — our checks are in DynamoRIO code and incur no cost while execution is in the code cache. As the working set changes, we will replace the old fragments with new fragments.

We found that fifty is a good value to use for the replaced fragment count: we check the regenerated count once every fifty fragments that are replaced in the cache. Checking too frequently is too easily influenced by temporary spikes, and too rarely is not reactive enough — we would like

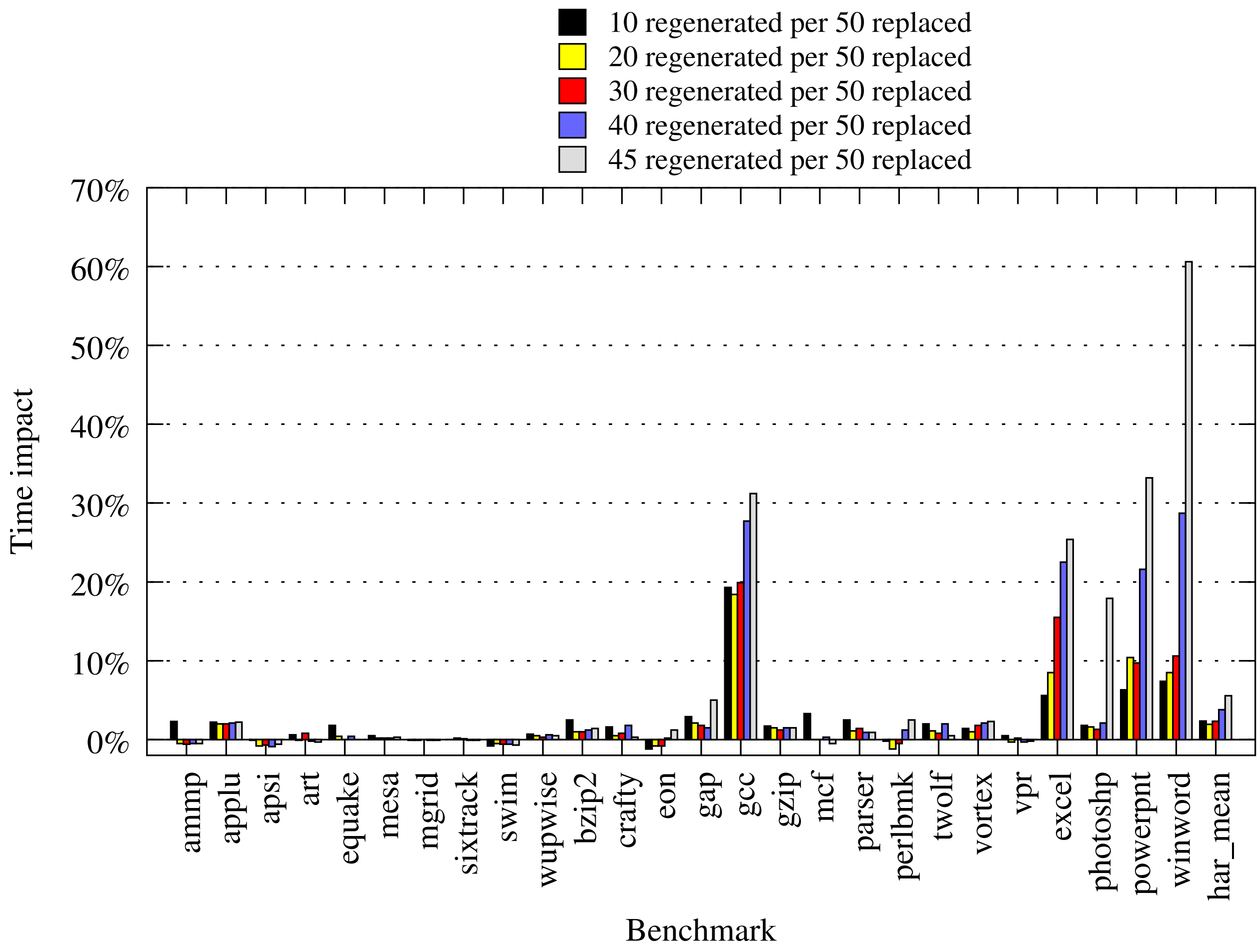

Figure 6.13: Performance impact of adaptive working set with parameters 10 regenerated / 50 replaced, 20/50, 30/50, 40/50, and 45/50.

to average things out a bit but not be too sluggish in resizing. The performance overhead of all of these checks and fragment replacements is low, as shown in Figure 6.13. The figure shows the performance impact of using regenerated counts of ten, twenty, thirty, forty, and forty-five fragments per fifty replaced. Only applications that execute large amounts of code with little reuse show appreciable overhead: `gcc` and our desktop benchmarks.

The resulting cache sizes from these parameters are shown in Table 6.14. For many of the SPEC CPU2000 benchmarks, the cache size quickly reaches the core working set of the application and does not shrink with successively higher regeneration thresholds. This is the goal of our algorithm: to identify the proper cache size to hold the working set of the application. The desktop

Benchmark	Basic block cache (KB)						Trace cache (KB)					
	∞	10	20	30	40	45	∞	10	20	30	40	45
ammp	50	50	50	50	50	50	77	75	75	75	75	75
applu	85	63	63	63	64	64	195	154	154	154	155	154
apsi	84	88	88	75	64	71	241	204	204	205	206	192
art	26	26	26	26	26	26	36	36	36	36	36	36
equake	41	41	41	41	41	41	72	64	64	64	64	64
mesa	82	64	64	64	64	64	76	64	64	64	64	64
mgrid	61	61	61	61	61	61	84	82	82	82	82	82
sixtrack	230	172	160	160	96	96	355	287	288	287	286	256
swim	52	52	52	52	52	52	61	61	61	61	61	61
wupwise	62	62	62	62	62	62	78	66	66	66	66	66
bzip2	34	34	34	34	34	34	47	47	47	47	47	47
crafty	151	96	100	98	96	94	278	243	246	243	240	237
eon	275	103	64	64	64	64	199	101	100	100	96	96
gap	153	107	110	102	64	64	370	313	311	316	310	294
gcc	684	608	594	560	547	522	1575	1503	1504	1504	1510	1520
gzip	30	30	30	30	30	30	40	40	40	40	40	40
mcf	35	35	35	35	35	35	40	40	40	40	40	40
parser	101	92	80	70	64	64	278	242	242	240	244	243
perlbmk	323	298	300	299	292	232	502	480	482	476	451	416
twolf	109	107	100	64	64	64	308	198	200	200	160	128
vortex	392	288	288	224	144	70	459	377	386	382	383	382
vpr	96	96	64	64	64	64	111	84	86	86	86	64
excel	2618	1632	1312	1024	64	64	608	430	416	404	366	347
photoshp	5521	3360	1856	1472	1184	1216	3371	2336	2336	2304	2114	1992
powerpnt	4242	2592	1632	992	736	480	2259	1520	1536	1504	1487	1440
winword	3135	2240	1664	1248	1024	64	1126	915	909	896	832	640
average	718	476	343	270	195	144	494	383	383	379	365	345

Table 6.14: Code cache sizes when using the adaptive working set algorithm with regeneration thresholds of 10, 20, 30, 40, and 45, per 50 replaced fragments. The sizes of the main thread's caches are given for the Windows benchmarks, and of the longest run for each SPEC CPU2000 benchmark.

benchmarks' caches do keep shrinking as the regeneration threshold is raised, especially the basic block cache, and performance begins to suffer as well, simply due to the work of regenerating so many basic blocks. Even when they are not part of the core working set, many are executed more than once. Here there is a tradeoff between memory and performance, and the lower thresholds should typically be chosen, since even with a threshold of ten the cache reduction is significant.

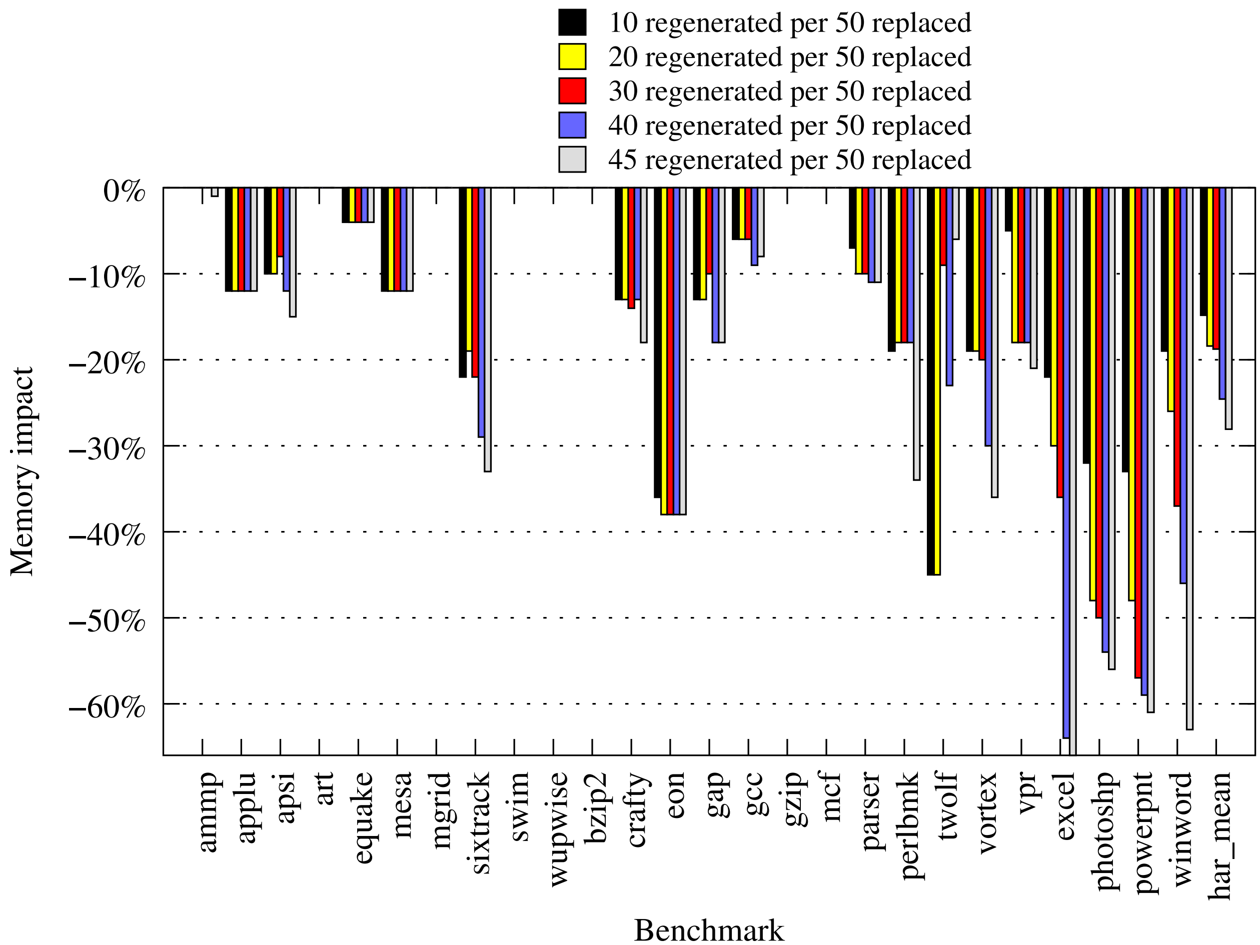

Figure 6.15: Virtual size memory impact of adaptive working set with parameters 10 regenerated / 50 replaced, 20/50, 30/50, 40/50, and 45/50. The graph shows the difference in the *additional* memory that DynamoRIO uses beyond the application's native use, not the difference in total memory used by the process.

The reduction in total memory resulting from the cache reductions of Table 6.14 is shown in Figure 6.15, while the working set reduction is shown in Figure 6.16. These track the code cache reductions, with up to seventy percent memory savings for higher regeneration thresholds, and ten to thirty percent for lower thresholds.

Future work for our algorithm is to shrink the cache when the working set shrinks, which is much more difficult to detect than when it grows. Size increases are driven by application requests, while size decreases must be driven by DynamoRIO via some type of periodic interrupt in order to guarantee that the cache will shrink for a now-idle thread. Such interrupts are problematic on

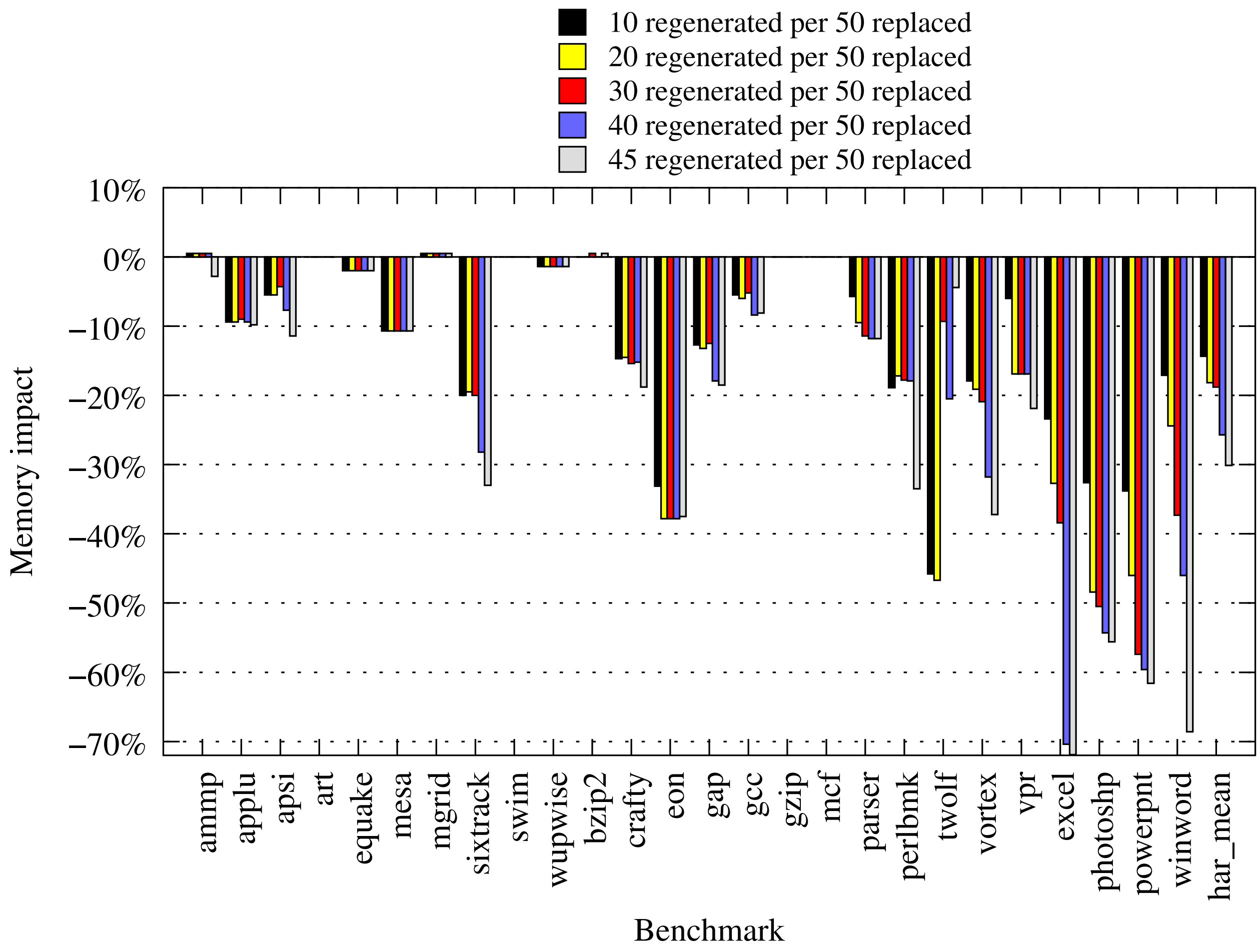

Figure 6.16: Resident size memory impact of adaptive working set with parameters 10 regenerated / 50 replaced, 20/50, 30/50, 40/50, and 45/50. The graph shows the difference in the *additional* memory that DynamoRIO uses beyond the application's native use, not the difference in total memory used by the process.

Windows without either a dedicated DynamoRIO thread or a DynamoRIO component that lives in kernel space. Explicit application actions like unloading libraries that imply reductions in code could also be used to drive cache shrinkage.

We considered whether VMWare's novel memory management techniques [Waldspurger 2002], such as *ballooning*, could be applied to DynamoRIO. However, ballooning requires the target to have its own adaptive memory management, which, while true for the operating system targets of VMWare, is not true of most applications. Typical applications never measure available memory, never tailor their allocation behavior to their environment, and may not even be robust in limited memory environments.

Another idea that we explored for sizing the basic block cache is *trace head distance*. For a given discovered trace head, this is the number of fragments created since the trace head was itself created. The philosophy is that the basic block cache only needs to be big enough to keep trace heads around long enough to turn into traces. The idea did not pan out well, however, as our measurements showed it to be a poor predictor of whether the cache should be made larger during typical cache resize points.

6.3.4 Code Cache Layout

Resizing the cache by allocating a larger region and re-locating the existing one is expensive, as it requires updating all control transfers that exit the cache (direct branches are program-counter-relative on IA-32). To provide more efficient and more flexible cache scalability, we subdivide our cache into units, each of which can be a different size. Asking for more space allocates a new unit, leaving existing units alone. Each unit is allocated directly from the operating system using the `mmap` system call on Linux and `NtAllocateVirtualMemory` [Nebbett 2000] on Windows. Cache units are separate from memory parceled out by the heap manager (Section 6.4) because of their large size.

DynamoRIO uses thread-private code caches, where each thread has its own private basic block cache and trace cache, which are each composed of separate units. Since these units are thread-private, no synchronization is required when accessing them. Freed units (e.g., on thread death) are either placed on a free list for use by future threads or released back to the operating system, according to a heuristic that keeps the free list at a size proportional to the number of threads (we keep at most $\max(5, \frac{\text{num\_threads}}{4})$ free units at any one time).

Logical Code Cache

Adding a level of indirection between the list of fragments in the cache and the actual layout of the cache units is important for keeping the cache manageable. We have two methods of iterating over fragments in the cache, one by physical order within each cache unit and the other by the logical order used for cache management (FIFO order). This separate logical list uses its level of indirection to build a higher abstraction than cache units and physical placements, facilitating the use

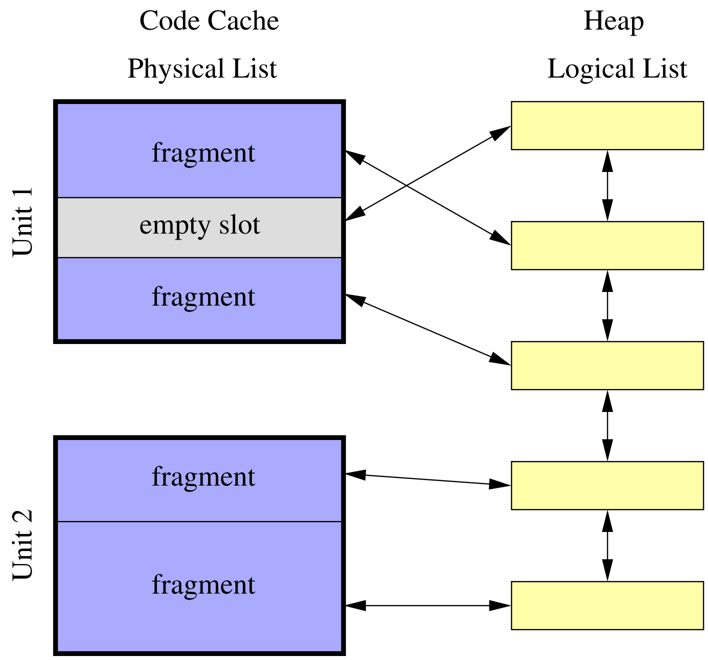

Figure 6.17: The separation of the logical FIFO list from the physical layout of fragments and empty slots in cache units. This indirection allows abstract eviction policies as well as simplifying support for multiple variable-sized cache units. Placing empty slots on the *front* of the list, as illustrated here, is known as *empty slot promotion*.

of multiple cache units with different sizes to represent a single logical code cache (Figure 6.17), as well as allowing cache management orders different from the strict cache address order (e.g., empty slot promotion).

The physical ordering is only required for freeing contiguous space in the cache. A four-byte header at the top of each fragment slot (Figure 6.18) is used to point to the `Fragment` data structure (Section 6.4.2) corresponding to the fragment slot. To walk forward on the physical list, the total fragment size is added to the current header location to produce the location of the next header. For the logical list, next and previous pointers in the `Fragment` data structure are used to chain fragments into a doubly-linked list. Each empty slot in the cache (these occur when a fragment is deleted from the middle of the cache) lives on the logical list as an `EmptySlot` structure, pointed to by the empty slot's cache header.

Fragment Layout

Figure 6.18 illustrates the layout of an individual fragment in a code cache unit:

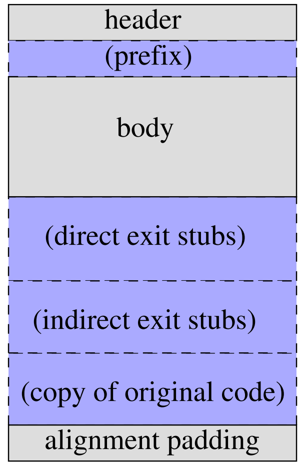

Figure 6.18: The layout of a fragment in the code cache. Dashed lines and blue indicate optional components, which are not always present.

- **Header** — Four bytes used by the cache manager to point to the `Fragment` structure corresponding to the fragment in that cache slot, for traversing the physical order of fragments in the cache. For an empty fragment slot, the header points to an `EmptySlot` data structure, and the subsequent fields are absent.

- **Fragment prefix** — The prefix code for the fragment, used to optimize transfer of control from DynamoRIO's indirect branch lookup routine (see Section 4.3.2 and Section 4.4) by shifting state restoration to the target, where registers and condition codes may not need to be restored if they are not live.

- **Fragment body** — The code for the body of the fragment.

- **Direct exit stubs** — The code for any direct exit stubs. It is best to relocate these and combine them all in a separate area (see Section 2.2 and Section 6.3.5), but they can also be located immediately after the fragment body.

- **Indirect exit stubs** — The code for any indirect exit stubs.

- **Copy of original application code** — This is only used for handling self-modifying code (see Section 6.2.3).

- **Alignment space —** Padding added to a fragment slot to achieve better cache line and word alignment. Padding is added to the end of a fragment, so the beginning becomes aligned only due to the padding added to the previous fragment. When using sub-cache-line alignment, this does make it harder to perform sophisticated alignment strategies that differ for each fragment depending on where in the cache line they fall. However, when a fragment is added it needs to consume empty space too small to become an empty slot (this empty space comes from the previous fragment that used to occupy that slot, which likely was a different size). Thus, all fragments must be able to have padding after the fragment body itself, so also allowing padding before the body would require extra bookkeeping, which we decided was not worth the memory cost.

- **Not-yet-used capacity —** Space at the end of the unit that has not been claimed yet.

The breakdown of cache space for basic blocks is given in Figure 6.19, and for traces in Figure 6.20. DynamoRIO separates direct exit stubs from the cache (see Section 6.3.5) and so they do not show up in these breakdowns. DynamoRIO uses prefixes (see Section 4.4) only on traces, reasoning that the performance advantage is not as needed for off-the-critical-path but basic blocks, where space is more important. Space taken up by the code copy used for self-modifying code is not present in these figures since none of our benchmarks contain self-modifying code. For applications with smaller code caches, the percentage of unused capacity can be relatively significant, since our unit allocation size becomes a factor, but for benchmarks with larger code caches the unused capacity shrinks to a negligible amount relative to the total cache size.

6.3.5 Compacting the Working Set

The code cache breakdowns given in Figure 6.19 and Figure 6.20 do not contain any direct exit stubs, because they have been separated from the cache and placed in a separate memory area. The motivation for this can be seen by looking at the same breakdowns when the stubs are included in the cache, which we show for basic blocks in Figure 6.21 and for traces in Figure 6.22. Direct exit stubs take up a significant portion of the cache and are not part of any critical path, since they are bypassed once direct links are set up.

By allocating the direct stubs in a separate location, we can compact the rest of the cache.

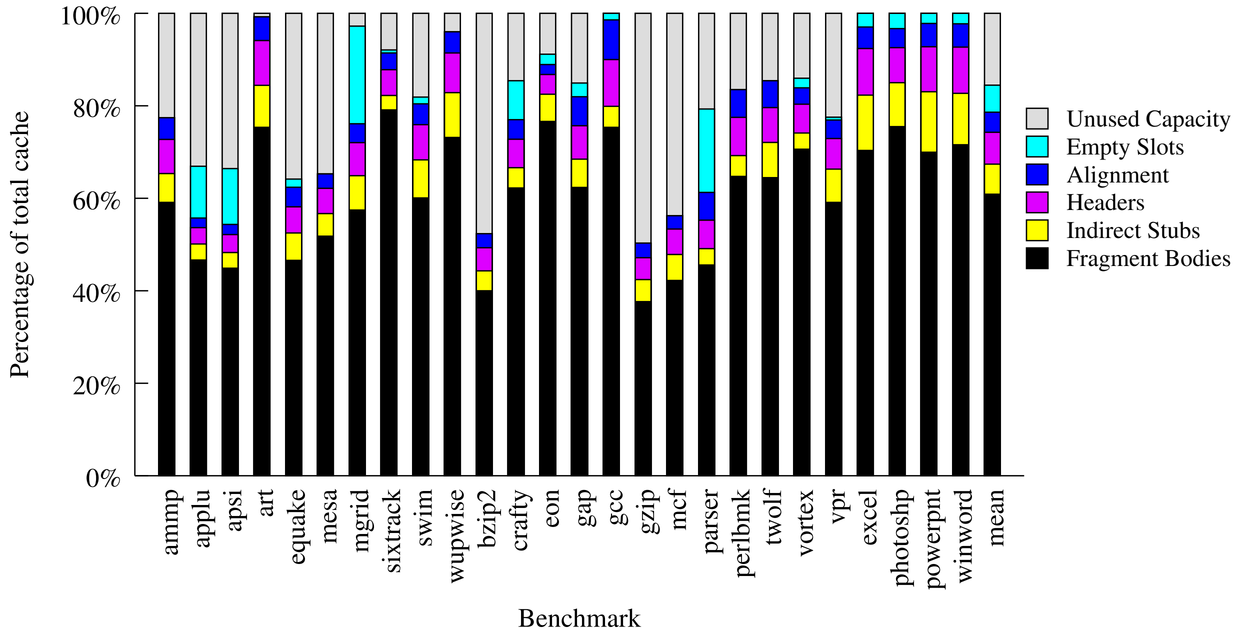

Figure 6.19: A breakdown of the basic block code cache with direct exit stubs stored outside of the cache for compactness and the ability to delete them when no longer needed (see Section 6.3.5).

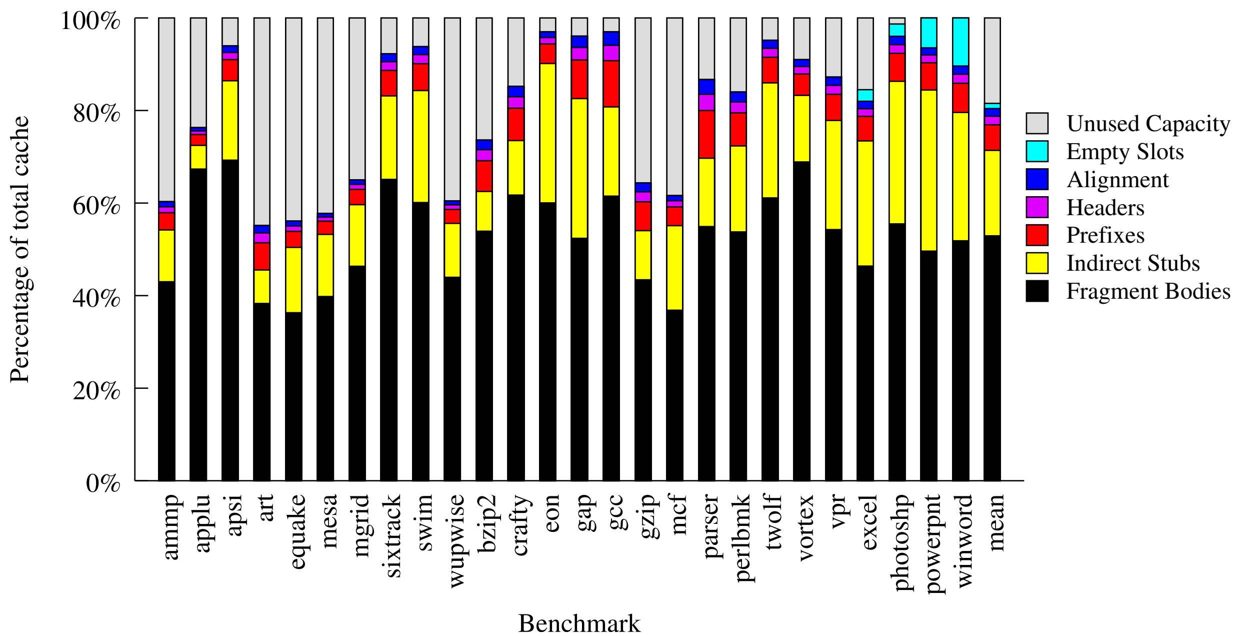

Figure 6.20: A breakdown of the trace code cache with direct exit stubs stored outside of the cache for compactness and the ability to delete them when no longer needed (see Section 6.3.5).

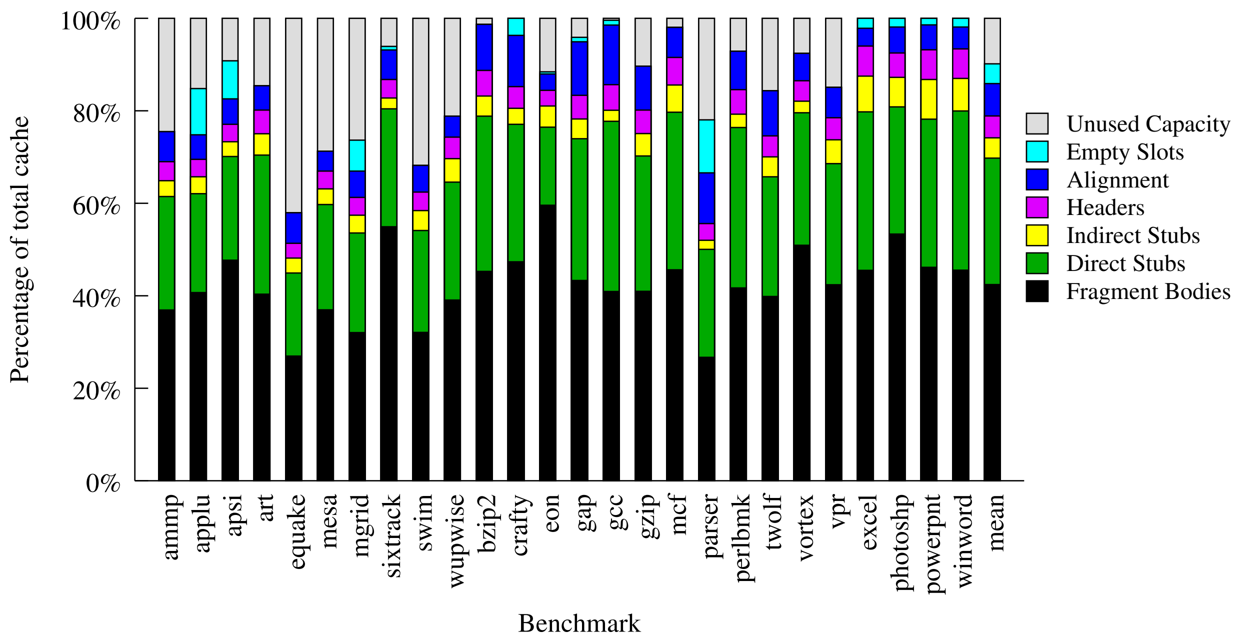

Figure 6.21: A breakdown of the basic block code cache with direct stubs in the cache. Direct stubs take up significant space, and if separated over half can be deleted, as shown in Table 6.23.

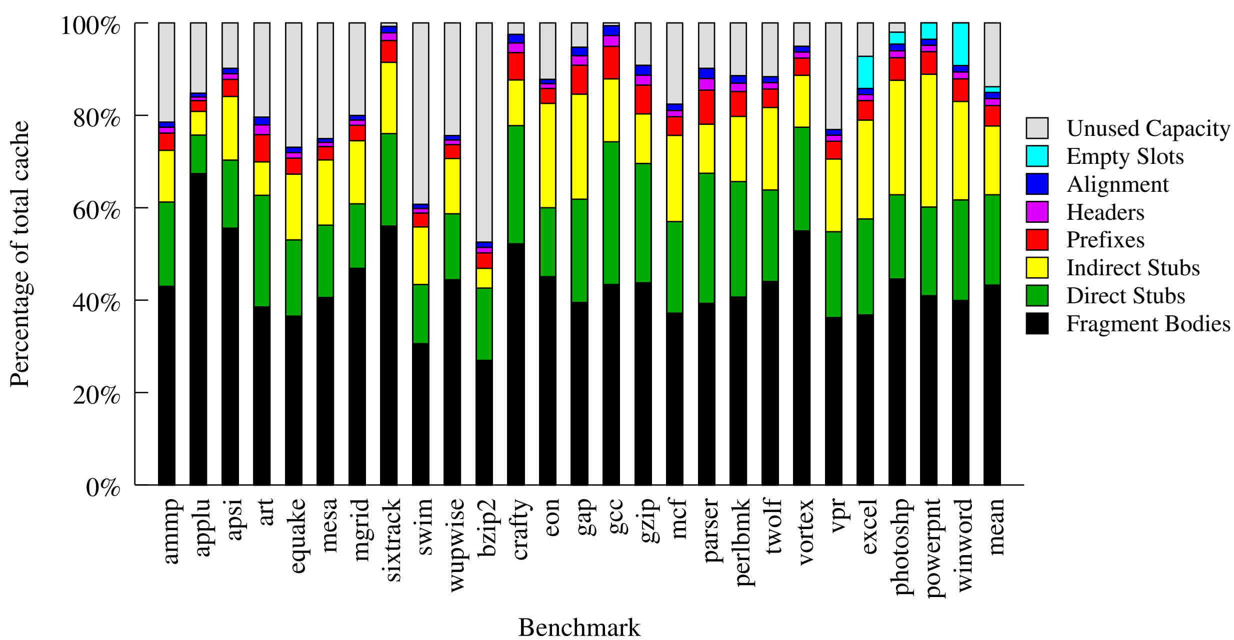

Figure 6.22: A breakdown of the trace code cache with direct stubs in the cache. Direct stubs take up significant space, and if separated over half can be deleted, as shown in Table 6.23.

Furthermore, once a direct exit is linked up to its target, the stub can be deleted, since it is not needed. If that exit becomes unlinked later, a new stub can be allocated on demand. The stub needs to be kept around for certain cases, such as incrementing a target trace head counter without leaving the cache (see Section 2.3.2) or for certain types of profiling (see Section 7.3.3). DynamoRIO does use stubs to increment trace head counters, but can still delete about half of all direct exit stubs. The resulting memory savings are shown in Figure 6.23. By compacting the working set of the code cache, we also achieve a performance improvement (presented in Figure 2.9).

Indirect stubs are always needed and can never be deleted. They could be separated, but since they are much rarer the working set compaction would be less, and, more importantly, the critical indirect branch lookup performance might suffer.

6.4 Heap Management

DynamoRIO must track information about all code in its cache, storing several data structures for each code fragment. This section discusses the data structures that we need and how we manage them in our heap.

6.4.1 Internal Allocation

We cannot use the application's heap for allocating our own data structures for transparency reasons (Section 3.1.1) — we must avoid both using the same allocation routines as the application and changing the application's internal memory layout. We ask for *chunks* of memory straight from the operating system using `mmap` on Linux and `NtAllocateVirtualMemory` [Nebbett 2000] on Windows. We then parcel out pieces of these chunks of memory in response to internal allocation requests. Our chunks are sized at 64 kilobytes, big enough to not cause a performance problem due to frequent allocation system calls, but small enough to not waste space in unused capacity. This is also the granularity of virtual address regions handed out by Windows (i.e., if we ask for less than 64 kilobytes, the rest of the 64 kilobyte address space region will be allocated to us but not be usable).

Since DynamoRIO uses thread-private caches, most of our data structures are thread-private. Each thread has its own private chunks used only for that thread's allocations, making it easy to free

Benchmark	In-cache stub size	Separated stub size	% Reduction
ammp	56070	24576	-56.2%
applu	49920	21504	-56.9%
apsi	92295	32768	-64.5%
art	27630	12288	-55.5%
equake	45165	21504	-52.4%
mesa	65310	32768	-49.8%
mgrid	46530	21504	-53.8%
sixtrack	208620	88064	-57.8%
swim	45660	22528	-50.7%
wupwise	52095	26624	-48.9%
bzip2	42495	15360	-63.9%
crafty	178395	48128	-73.0%
eon	116625	48128	-58.7%
gap	217545	70656	-67.5%
gcc	1185300	364544	-69.2%
gzip	36075	12288	-65.9%
mcf	35310	16384	-53.6%
parser	190530	47104	-75.3%
perlbmk	358275	145408	-59.4%
twolf	158775	64512	-59.4%
vortex	353415	177152	-49.9%
vpr	87990	36864	-58.1%
excel	1691595	1013760	-40.1%
photoshp	3317085	1454080	-56.2%
powerpnt	2807535	1613824	-42.5%
winword	2190765	1305600	-40.4%
average	525269	259151	-56.9%

Table 6.23: Memory reduction from separating direct exit stubs, which allows us to delete stubs no longer in use.

memory when a thread exits by simply freeing all the thread's chunks at once. Freeing can either mean placing the chunks on a free list for use by other threads in the future, or actually freeing the memory back to the operating system. We employ a heuristic to keep the free list at a size proportional to the number of threads (the same as that used for cache units — see Section 6.3.4), to avoid wasting memory by holding onto chunks that will never be used. No synchronization is needed for thread-private memory allocation. We also have a set of global chunks that are used for process-wide allocation needs, which do require synchronization.

Our scheme for parceling out pieces of memory is to use *buckets* of fixed size for common small allocation sizes, for which no header is needed, along with variable-sized larger allocations that use a four-byte header to store their size. The bucket sizes are chosen based on the exact sizes of data structures used in DynamoRIO (see Section 6.4.2). A free list is kept per bucket size, using the first four bytes of a freed allocation to store the pointer to the next item in the free list for that bucket size. An allocation request smaller than the largest bucket size is given the first free entry on the list for the smallest bucket size that will hold the request. A request larger than the largest bucket size uses a custom-sized allocation. The free list for variable-sized allocations also steals four bytes from each freed allocation (after the size header) to link entries together.

Since most of our memory use is comprised of large numbers of only a few different data structures, we save a significant amount of memory by using this fixed-size bucket scheme. Table 6.24 shows the breakdown of allocations into buckets and variable-sized, and shows that the average overhead is less than half a byte per allocation. Counter-intuitively, we use more memory on our heap than we do on our code cache, so saving heap memory is critical.

To aid in detecting and resolving memory errors in DynamoRIO itself, in our debug build we initialize memory on allocation to a certain pattern of bytes, and on freeing to a different pattern of bytes, both invalid addresses. A memory access fault targeting one of these patterns indicates uninitialized memory or overlapping allocations, and a memory bug is much more likely to show up earlier than when not using these patterns.

Thread-private memory allocation has been used elsewhere to avoid synchronization costs [Domani et al. 2002]. Our combination of individual de-allocation with fast freeing of entire regions is similar to the *reaps* introduced by Berger et al. [2002]. Berger et al. [2002] also claim that custom memory allocators rarely improve performance and so are not usually worth their implementation cost; region freeing is the exception.

6.4.2 Data Structures

Table 6.25 lists the important data structures in our system. Many of the structures contain next and previous fields for chaining into linked lists. Embedding the next and previous pointers in

Benchmark	Allocations			Overhead (Bytes)	
	Bucket	Variable	% Buckets	Total	Per Allocation
ammp	42649	23	99.95	20025	0.47
applu	39601	25	99.94	15034	0.38
apsi	87838	31	99.96	32624	0.37
art	18414	15	99.92	6670	0.36
equake	36113	22	99.94	16009	0.44
mesa	46445	30	99.94	18012	0.39
mgrid	40018	22	99.95	14986	0.37
sixtrack	177429	38	99.98	75856	0.43
swim	38437	21	99.95	14303	0.37
wupwise	44167	22	99.95	15339	0.35
bzip2	31600	18	99.94	14107	0.45
crafty	138283	39	99.97	60541	0.44
eon	110404	39	99.96	34785	0.32
gap	184204	41	99.98	89565	0.49
gcc	831441	139	99.98	417042	0.50
gzip	30573	15	99.95	11597	0.38
mcf	27647	24	99.91	9143	0.33
parser	150318	38	99.97	75097	0.50
perlbmk	245753	41	99.98	110664	0.45
twolf	127859	54	99.96	67929	0.53
vortex	250802	60	99.98	119875	0.48
vpr	72009	37	99.95	28769	0.40
excel	1371857	130	99.99	201532	0.15
photoshp	4208704	369	99.99	764094	0.18
powerpnt	2573335	274	99.99	574225	0.22
winword	1736310	174	99.99	351414	0.20
average	487008	66	99.96	121509	0.38

Table 6.24: Statistics on heap allocation requests. Only a handful of allocations are larger than our set of standard bucket sizes and require our variable-sized allocation handling. The overhead from wasted space, from both allocations smaller than their closest bucket size and the four-byte header required for variable-sized allocations, is given in the final columns, as the total bytes and the average bytes per allocation.

Name	Size	Represents	Key fields
Fragment	48	basic block in cache	code (prefix, body, stubs), flush list, cache list
FutureFragment	12	fragment not in cache	proactive linking, adaptive working set
Trace	56+	trace in cache	Fragment + component bb info
Linkstub	24	exit stub	target, exit pc, in & out lists
InstrList	12	linear Instr sequence	first, last
Instr	64	single instruction	opcode, operands
Opnd	12	single operand	type, value, data size
EmptySlot	24	empty slot in cache	location, size, cache list
MultiEntry	24	multi-region fragment	fragment, flush list

Table 6.25: Salient data structures in our system. Size is the non-debug-build size in bytes. More information on instruction representation (InstrList, Instr, Opnd) can be found in Section 4.1.

the data structure itself avoids the need for a container structure for each element of a linked list, saving memory and time, although it only works when each instance can only be at one location on one list at a time.

DynamoRIO's core data structure, called `Fragment`, contains pointers to the code for a basic block in the cache, including the prefix, exit stubs, and the block body itself. It also has next and previous pointers for the cache capacity management list (see Section 6.3.1) and the cache consistency region list (see Section 6.2.5). Traces require storing extra information about their constituent basic blocks, for which we extend `Fragment` to create the `Trace` data structure. For proactive linking and single-fragment deletion (see Section 2.2), we store the incoming branches for potential future fragments in the `FutureFragment` structure. If we never delete future fragments (which we could do when they no longer have any branches targeting them), we can also use them for record keeping to implement our adaptive working set cache management policy (Section 6.3.3). Future fragments are also a convenient location to store persistent trace head counters (Section 2.3.2 and Section 6.3.2). For each exit from a fragment, we must know the target address and the address of the exit branch itself for linking. The `Linkstub` data structure stores this information, along with next fields for chaining into a list of incoming branches for proactive linking. Another list could be created for the outgoing exits from a fragment, but we can save memory by instead allocating outgoing `Linkstubs` in an array immediately after their owning `Fragment`.

Three key data structures are used for our instruction representation: `InstrList`, `Instr`, and

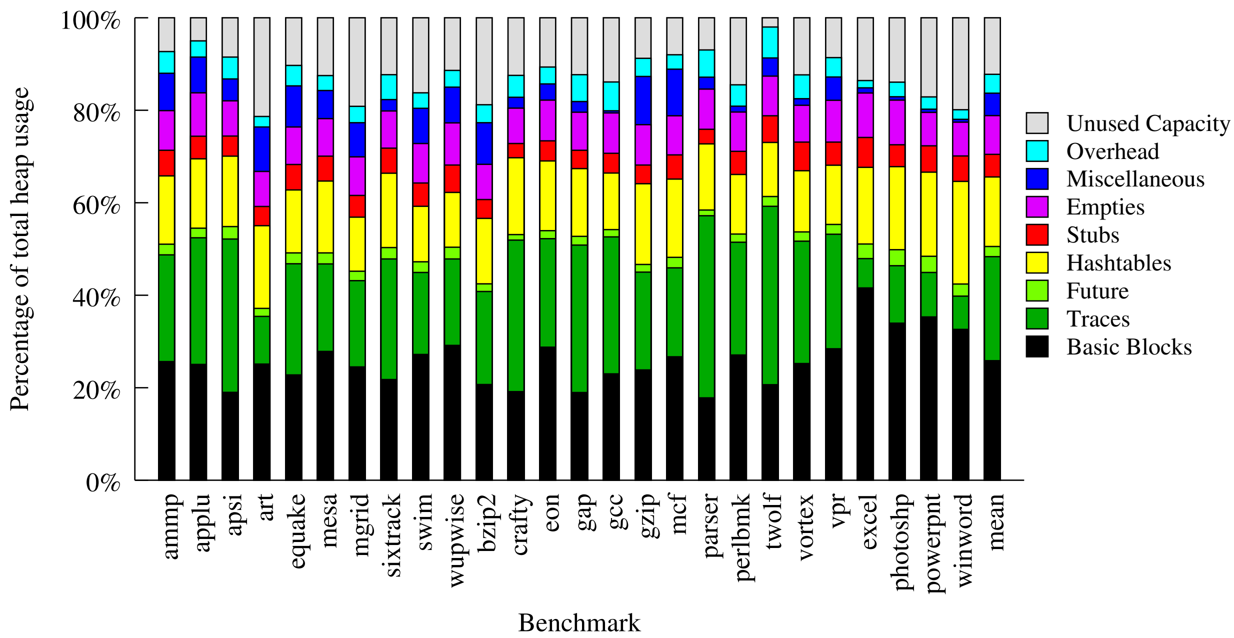

Figure 6.26: A breakdown of the heap used by DynamoRIO.

`Opnd`. They are described in more detail in Section 4.1. Additional data structures are needed for both the cache capacity management list (see Section 6.3.1) and the cache consistency flush list (see Section 6.2.5). For capacity, an extra structure pointing to an empty slot in the code cache, called `EmptySlot`, is used. For consistency, another structure is needed for fragments that contain pieces of code from multiple memory regions: `MultiEntry`.

Larger data structures (not shown in the table) include our fragment hashtables, which use open-address collision resolution (Section 4.3.3) for performance reasons, and each thread's private context structure, which stores the application machine state while in DynamoRIO code (see Section 5.2.2) along with pointers to all thread-local data, such as heap chunks and fragment hashtables. Other data structures used by DynamoRIO include a master thread table, synchronization primitives, memory region lists, and trace building information.

Figure 6.26 shows the heap breakdown for our benchmark suite. Most of our memory usage is taken up by the data structures for fragments in the code cache, `Fragment` and `Trace`. Since the `Linkstub` data structure is allocated with `Fragment` and `Trace` to save space, it shows up under their categories in our heap statistics. Our hashtables come next in terms of total space.

Header overhead from our allocation schemes is quite small, though wasted space from yet-unused capacity does show up.

6.5 Evaluation

Having described both code cache and heap management, we now take a step back and evaluate our total memory usage and re-evaluate thread-shared versus thread-private caches. Figure 6.27 shows the memory that the DynamoRIO system uses, as a function of the amount of code executed by the application. (The memory usage of our server benchmarks is presented separately, later in this section.) Both working set size (the amount present in physical memory) and total virtual size (address space reserved) are given. The code cache expansion combined with associated data structures results in an average of ten times as much memory as the amount of code that is natively executed. How this memory compares to the total memory usage of the application depends, of course, on how the native code executed compares to the amount of data manipulated by the application. An application that uses a large amount of data memory and a small amount of code shows little increase from DynamoRIO, while one that has little data memory shows a large increase. Multiple threads will also result in a larger increase when using thread-private caches. Figure 6.28 shows that the increase in total memory usage when DynamoRIO is controlling an application, compared to native usage, ranges from negligible to nearly twice as much memory, with an average of about a ten percent increase. The absolute memory usage of DynamoRIO is given in Figure 6.29. For large applications like our desktop benchmarks, we use more than ten megabytes of memory. Figure 6.30 breaks our memory usage down into three categories: the private stack used for each thread, the code cache space allocated, and the heap allocated for data structures. Counter-intuitively, heap usage dominates, as the data structures needed to manage a single code fragment often use more space than the code contained in the fragment itself.

For executing a single application at a time, memory usage is seldom a concern. But when running many applications at once, or when targeting applications with many threads, memory usage can become an important issue. DynamoRIO reduces its usage by tightening data structures and deleting unused exit stubs (Section 6.3.5). For applications with many threads, however, a shared code cache is the only truly scalable solution for memory reduction, and a choice must be

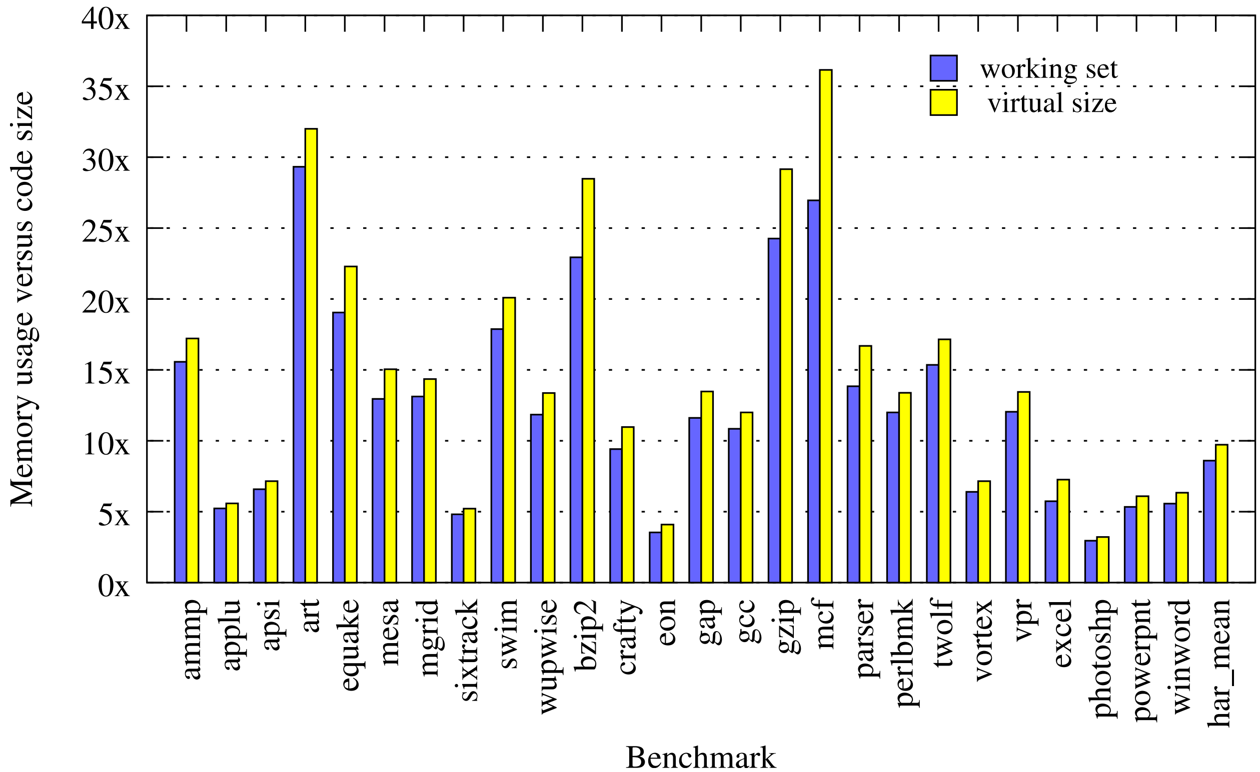

Figure 6.27: The memory used by DynamoRIO using our adaptive working set cache sizing algorithm (with 30 regenerated fragments per 50 replaced — see Section 6.3.3), relative to the amount of native code executed.

made between performance and space.

A hybrid option is to share basic blocks but keep traces thread-private, saving memory because the majority of most applications' memory usage is from basic blocks while retaining the performance advantage of thread-private traces. We performed a preliminary study of the memory savings of such a hybrid scheme, which requires flexibility in all DynamoRIO modules to support both shared and private fragments simultaneously. For scratch space we use segment registers to point to thread-local storage (see Section 5.2.1). For trace building, we share trace-headness, but use private trace head counters, and create a private copy of each shared basic block with which a private trace is being extended. Basic block building uses a single, monolithic mutual exclusion region, which fortunately does not impact performance. Cache consistency uses the scheme described in Section 6.2, which works for both private and shared caches. We do not have a solution to shared cache capacity better than suspending all threads, however, and we leave it as future work,

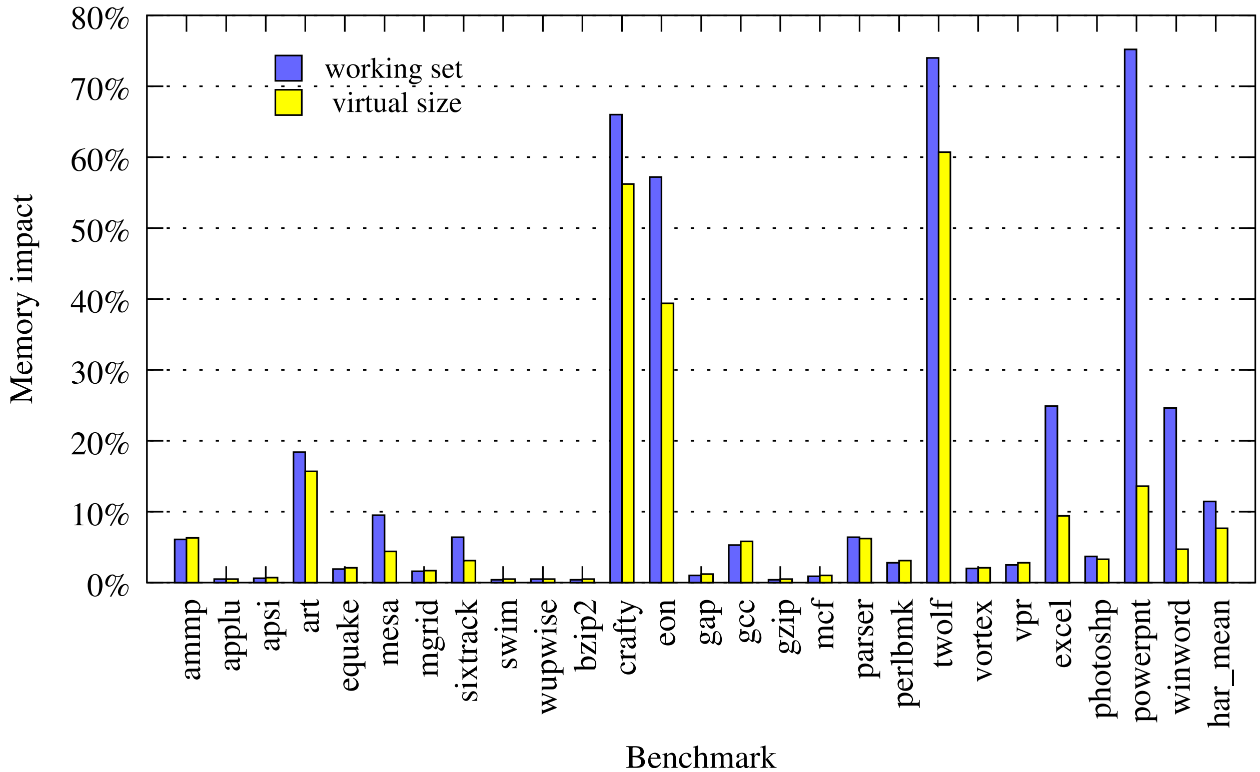

Figure 6.28: The combined memory used by the application and DynamoRIO, when using our adaptive working set cache sizing algorithm (with 30 regenerated fragments per 50 replaced — see Section 6.3.3).

along with how to adaptively choose the proper cache sharing configuration for each application.

Figure 6.31 shows the memory usage of our server benchmarks compared to application code executed, for thread-private trace caches but comparing thread-private and thread-shared basic block caches. Using all thread-private caches, server memory usage dwarfs our other benchmarks, but does not impact performance (see Section 7.2) and is only an issue when running multiple servers at once. Figure 6.32 shows the memory usage impact on the server benchmarks as a whole, which again is higher than our other benchmarks. Sharing basic blocks saves from twenty to forty percent of total memory usage.

Extending this study to full thread-shared caches is future work. The single-fragment deletion taken advantage of in thread-private empty slot filling (Section 6.3.1) and adaptive working set sizing (Section 6.3.3) will not work when every deletion is expensive — these schemes must be modified to reduce the frequency of deletions for thread-shared caches. Single-fragment deletion is

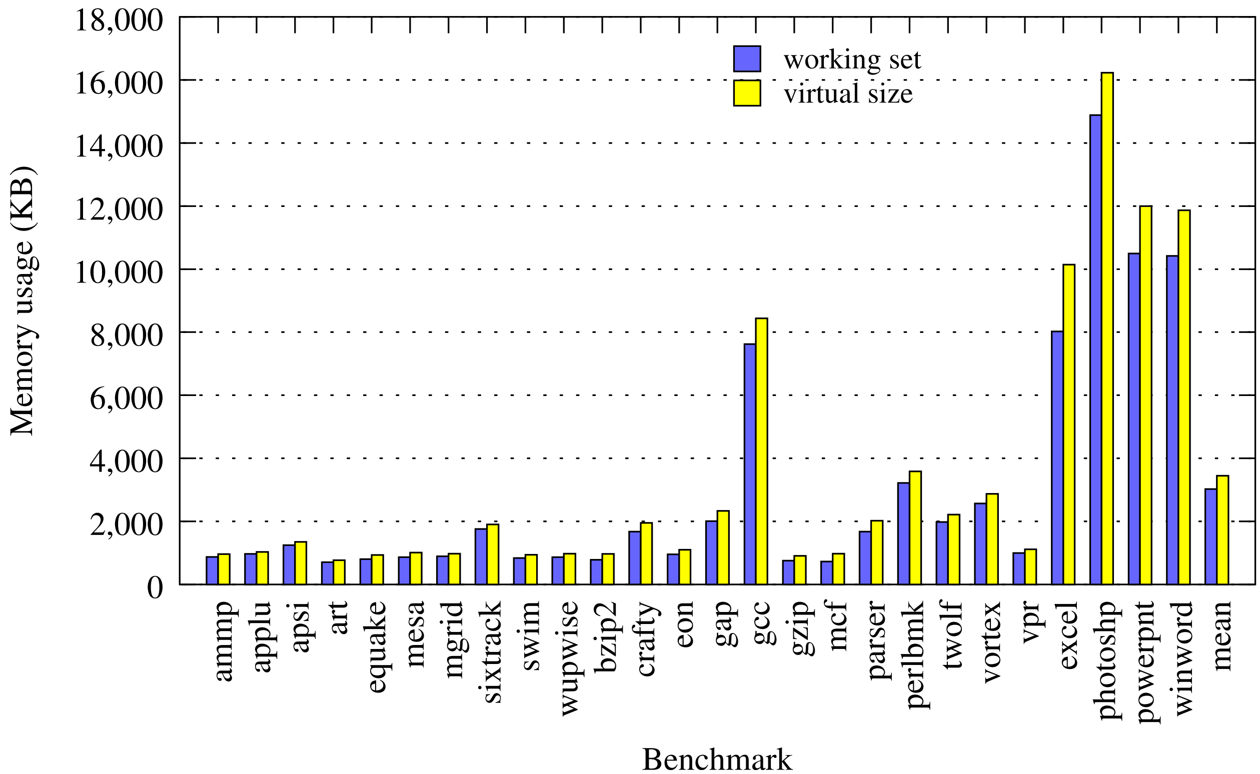

Figure 6.29: Memory usage in KB of DynamoRIO using our adaptive working set cache sizing algorithm (with 30 regenerated fragments per 50 replaced — see Section 6.3.3).

still required for cache consistency, which avoids thread-shared performance problems by delaying actual deletion into later batches. Unfortunately cache capacity cannot be similarly delayed when the cache is full. One possible capacity scheme is to divide the cache into segments and keep track of how many threads are in each segment. Threads enter a synchronization point before entering a new segment (adding a little overhead on all inter-segment links). When room is needed in the cache, the system picks a thread-free segment and prevents other threads from entering while it deletes some or all of the fragments in that segment. If no segment can be found that has no threads in it, all threads in the least-populated segment are suspended.

6.6 Chapter Summary

This chapter discussed strategies for managing both the heap and the cache of a runtime code manipulation system. We presented a novel scheme for sizing the code cache called *adaptive*

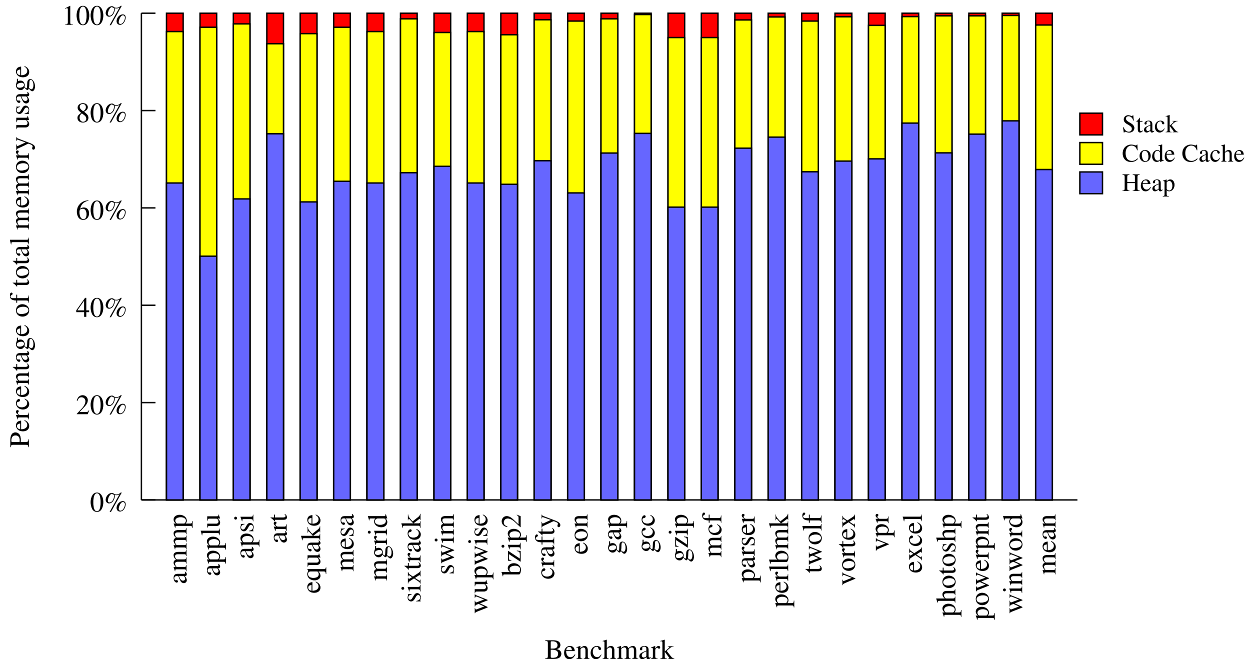

Figure 6.30: A breakdown of the memory used by DynamoRIO. Memory is placed into three categories: the private stacks used for each thread, the code cache, and the heap for allocating data structures used to manage the cache.

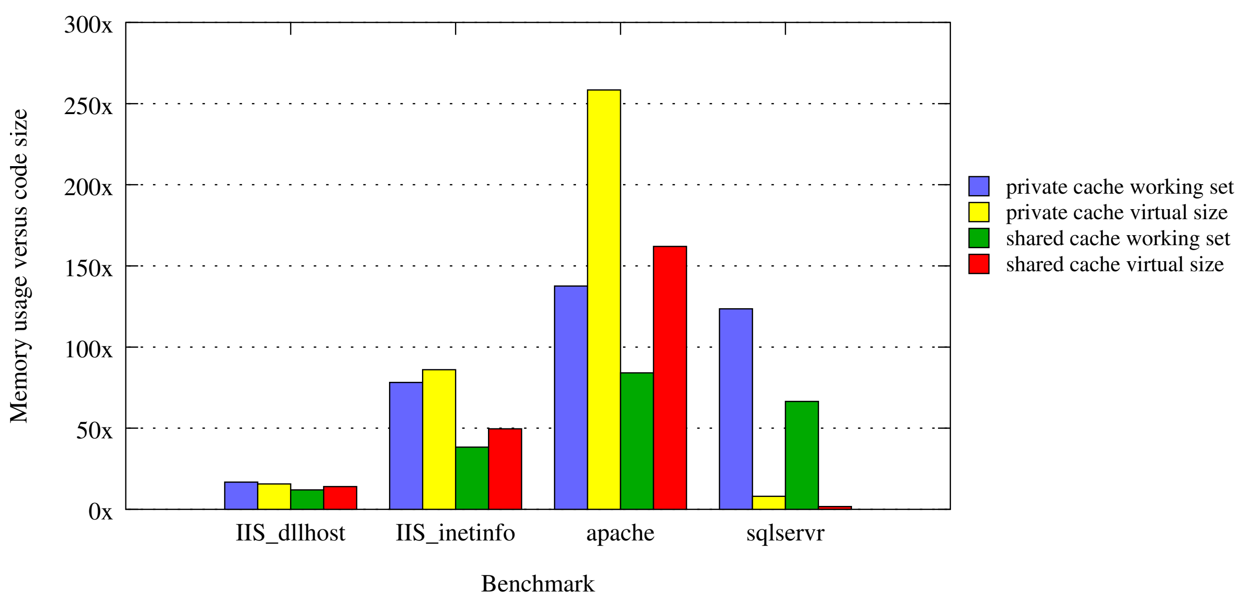

Figure 6.31: The memory used by DynamoRIO on our server benchmarks, relative to the amount of native code executed, for both caches thread-private and for the basic block cache thread-shared.

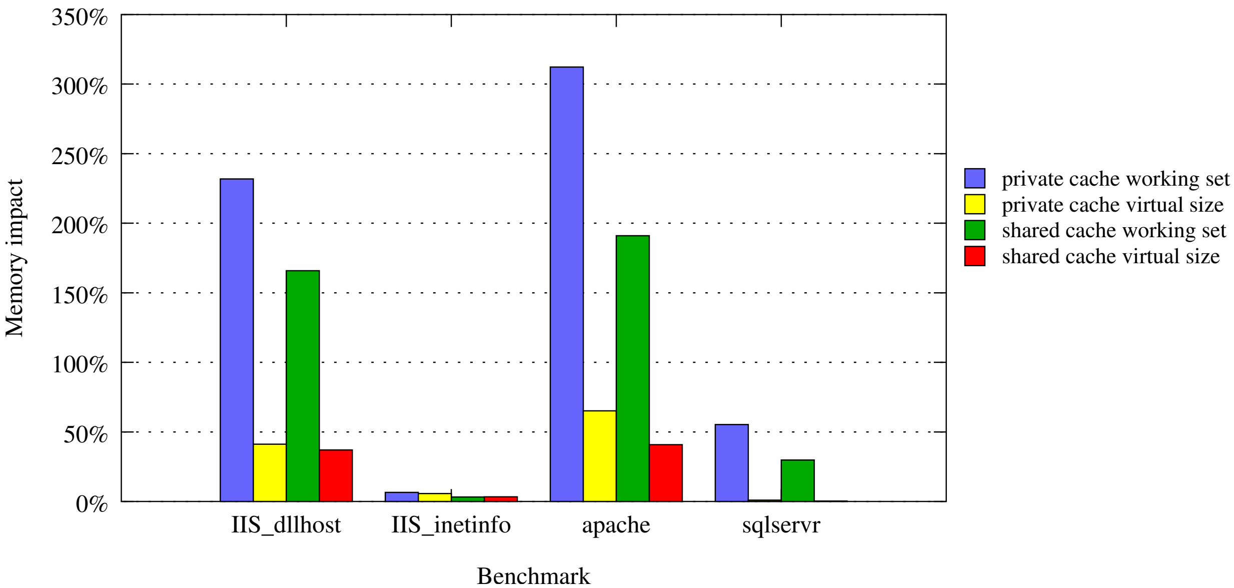

Figure 6.32: The combined memory used by the application and DynamoRIO on our server benchmarks, for both caches thread-private and for the basic block cache thread-shared.

working-set-size detection, and a novel algorithm for maintaining cache consistency efficiently by delaying fragment deletion: *non-precise flushing*. DynamoRIO uses thread-private caches where possible, even when running server applications, for optimal performance. When executing many servers at once under DynamoRIO, however, memory usage can be problematic. We leave support for full thread-shared caches with efficient capacity limits as future work.

Chapter 7

Performance

In this chapter we present our benchmark suite and measurement methodology (Section 7.1) and evaluate DynamoRIO's impact on execution time (Section 7.2). We then describe the profiling tools we use in analyzing the performance both of DynamoRIO and of the application itself (Section 7.3). This chapter focuses on overall system performance; achieving that performance by carefully designing each component of DynamoRIO is presented in Chapter 2 and Chapter 4.

7.1 Benchmark Suite

Our benchmark set consists of the SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] benchmarks on Linux and seven large desktop and server applications on Windows. Table 7.1 and Table 7.2 give brief descriptions of each of these programs. We compiled the SPEC CPU2000 benchmarks using `gcc -O3`. Since `gcc` does not have a FORTRAN 90 front end, we omit the four FORTRAN 90 benchmarks (`facerec`, `fma3d`, `galgel`, and `lucas`) from our SPECFP set.

While our server benchmarks are standard workloads run against commercial web servers and a database server, we created our desktop benchmarks ourselves due to lack of standardized suites of large Windows desktop programs. Since these programs are usually interactive, creating automated benchmarks is challenging. One standard suite is Winstone [VeriTest], which we considered using. However, Winstone reports only one number and does not split up performance over its set of applications. The other benchmark suites for Windows that we found measure the performance of the underlying hardware rather than application performance. Our desktop benchmarks model

Group	Program	Benchmark description
SPECFP: Floating Point	ammp	Modeling large systems of biological molecules
	applu	Computational fluid dynamics
	apsi	Weather prediction
	art	Image recognition via neural networks
	equake	Simulation of seismic wave propagation in large basins
	mesa	3-D graphics library
	mgrid	Computational fluid dynamics
	sixtrack	High energy nuclear physics accelerator design
	swim	Weather prediction
	wupwise	Lattice gauge theory (quantum chromodynamics)
SPECINT: Integer	bzip2	Data compression (bzip2)
	crafty	Chess playing program
	eon	Probabilistic ray tracer
	gap	Group theory interpreter
	gcc	C language optimizing compiler
	gzip	Data compression (LZ77)
	mcf	Combinatorial optimization (single-depot vehicle scheduling)
	parser	Word processing
	perlbmk	Perl scripting language
	twolf	Computer aided design global routing
	vortex	Object-oriented database transactions
	vpr	Integrated circuit computer-aided design program
Desktop	excel	Microsoft Excel 9.0: loads a 2.4MB spreadsheet full of interdependent formulae and modifies one cell, triggering extensive re-calculations
	photoshp	Adobe Photoshop 6.0: the PS6bench [psbench@yahoo.com] Action without the Stop actions and with only one iteration of each sequence
	powerpnt	Microsoft PowerPoint 9.0: loads an 84-slide (455KB) presentation and adds text to the title of every slide
	winword	Microsoft Word 9.0: loads a 1.6MB document, replaces 'a' with 'o', then "selects all" and changes the font type and size

Table 7.1: Our benchmark suite can be broken into four groups. The first two, the SPECFP floating-point and SPECINT integer benchmarks, together comprise the SPEC CPU2000 benchmark suite [Standard Performance Evaluation Corporation 2000] (minus the four FORTRAN 90 benchmarks). The third group is a set of our own desktop benchmarks. The fourth group, our server benchmarks, is shown in Table 7.2.

Group	Program	Benchmark description
Servers	IIS	Microsoft Internet Information Services 5.0: The SPECweb99 [Standard Performance Evaluation Corporation 1999] benchmark was run against IIS, with DynamoRIO executing both `inetinfo.exe` and `dllhost.exe`, with 400 simultaneous client connections. For performance we report the worst result among throughput and response time.
	apache	Apache HTTP Server 2.0.49.0: The WebBench [VeriTest 2002] benchmark with two client machines was run against Apache.
	sqlservr	Microsoft SQL Server 2000: The Database Hammer stress test from the Microsoft SQL Server 2000 Resource Kit [Microsoft Corporation 2001]. Our client was configured to use 60 threads, each making 6000 requests, with 50ms between each request.

Table 7.2: Continuing Table 7.1, our fourth benchmark group is the server applications. All were run on Windows 2000 Advanced Server.

long-running batch computations, not highly interactive use. More interactive scenarios are hard to measure and mask our slowdowns, giving misleading results. We built our benchmarks using the internal scripting available in these applications, as we found that recording macros based on window positions and timing was error-prone. We did use such macros (Macro Express [Insight Software Solutions, Inc.]) to launch the internal scripts for those applications that could not be set up to automatically launch the script on startup (`photoshp` and `powerpnt`).

Table 7.3 gives vital statistics for our benchmark suite, including the size of each executable and its shared libraries, the amount of code executed during the run of the benchmark, the number of threads, and the number of basic blocks and traces that DynamoRIO builds when executing the benchmark. Although the entire SPEC CPU2000 suite is single-threaded, it worked well for measuring the performance impact of most of our architectural challenges (Chapter 4). DynamoRIO also targets real-world, multi-threaded applications like those in our desktop and server suites. These applications are an order of magnitude larger than the SPEC CPU2000 programs, and had we focused only on SPEC CPU many of our design decisions would have gone in different directions, particularly our memory management choices (Chapter 6).

Since indirect branches are a key performance problem for DynamoRIO (see Section 4.2), Table 7.4 shows a breakdown of indirect branches for each benchmark. The floating-point benchmarks have the fewest branches, as expected. Perhaps surprisingly, the desktop and especially the

Program	Exec Size	Libraries	Total Size	Code Seen	Threads	Blocks	Traces
ammp	338 KB	2	6543 KB	51 KB	1	2351	472
applu	355 KB	2	6561 KB	191 KB	1	2687	659
apsi	516 KB	2	6722 KB	152 KB	1	4470	1316
art	62 KB	2	6267 KB	23 KB	1	1395	295
equake	65 KB	2	6271 KB	40 KB	1	1940	471
mesa	1378 KB	2	7583 KB	63 KB	1	2884	377
mgrid	200 KB	2	6405 KB	58 KB	1	2321	480
sixtrack	2526 KB	2	8732 KB	268 KB	1	9270	2282
swim	171 KB	2	6376 KB	41 KB	1	2342	394
wupwise	227 KB	2	6433 KB	58 KB	1	2665	430
SPECFP average	*583 KB*	*2*	*6789 KB*	*94 KB*	*1*	*3232*	*717*
bzip2	127 KB	1	5730 KB	31 KB	1	1693	396
crafty	487 KB	1	6090 KB	145 KB	1	6306	2088
eon	2626 KB	3	10036 KB	236 KB	1	6002	918
gap	1032 KB	1	6635 KB	135 KB	1	8645	2825
gcc	3185 KB	1	8788 KB	498 KB	1	36494	14055
gzip	169 KB	1	5772 KB	27 KB	1	1600	370
mcf	72 KB	1	5675 KB	25 KB	1	1661	274
parser	395 KB	1	5998 KB	90 KB	1	6538	3039
perlbmk	1396 KB	2	7602 KB	216 KB	1	14695	4003
twolf	534 KB	2	6739 KB	109 KB	1	5781	1707
vortex	1385 KB	2	7590 KB	217 KB	1	12461	2377
vpr	388 KB	2	6594 KB	64 KB	1	3799	730
SPECINT average	*983 KB*	*1*	*6937 KB*	*149 KB*	*1*	*8806*	*2731*
excel	6984 KB	22	26056 KB	1397 KB	4	77174	2712
photoshp	13516 KB	52	38164 KB	5043 KB	8	171962	19985
powerpnt	4224 KB	32	25224 KB	1967 KB	5	105054	9413
winword	8592 KB	28	28932 KB	1871 KB	4	98358	6328
desktop average	*8329 KB*	*33*	*29594 KB*	*2569 KB*	*5*	*113137*	*9609*
IIS inetinfo	15 KB	97	19912 KB	1022 KB	44	54554	4488
IIS dllhost	6 KB	35	10464 KB	677 KB	14	34849	2037
apache	21 KB	39	7884 KB	214 KB	253	12004	1692
sqlservr	7345 KB	71	23616 KB	1728 KB	276	80389	4656
server average	*1846 KB*	*60*	*15469 KB*	*910 KB*	*146*	*45449*	*3218*

Table 7.3: Statistics for our benchmarks: the static size of the executable, the number of shared libraries loaded, the sum of the sizes of the executable and all shared libraries loaded by the program, the amount of code actually executed under DynamoRIO, the number of threads, the number of unique basic blocks executed, and the number of unique traces created. For SPEC CPU2000 benchmarks with multiple datasets, the longest run is shown (other runs are similar). For applications with varying numbers of threads, the peak number of simultaneously live threads is given.

Benchmark	% branches	% indirect branches	Indirect Branch Types		
			Returns	Ind jmps	Ind calls
ammp	8.30%	0.07%	70.8%	29.2%	0.0%
applu	1.36%	0.00%	62.5%	17.2%	20.3%
apsi	4.81%	1.10%	99.2%	0.8%	0.0%
art	15.89%	0.00%	73.3%	25.8%	0.9%
equake	4.39%	0.73%	92.8%	7.2%	0.0%
mesa	8.65%	1.00%	77.0%	5.4%	17.6%
mgrid	0.34%	0.00%	58.5%	26.9%	14.7%
sixtrack	1.99%	0.01%	27.9%	68.0%	4.2%
swim	0.89%	0.00%	54.9%	36.6%	8.5%
wupwise	6.86%	0.29%	99.9%	0.1%	0.0%
SPECFP average	*5.35%*	*0.32%*	*71.7%*	*21.7%*	*6.6%*
bzip2	14.33%	0.92%	100.0%	0.0%	0.0%
crafty	13.32%	1.27%	84.6%	15.4%	0.0%
eon	11.83%	1.69%	74.6%	0.8%	24.7%
gap	15.00%	2.69%	57.1%	0.0%	42.9%
gcc	19.76%	1.28%	71.5%	27.0%	1.6%
gzip	18.87%	1.07%	97.7%	2.3%	0.0%
mcf	24.26%	0.12%	86.0%	14.0%	0.1%
parser	18.76%	1.37%	97.9%	2.1%	0.0%
perlbmk	21.10%	2.24%	51.6%	40.1%	8.4%
twolf	16.28%	0.72%	99.6%	0.4%	0.0%
vortex	17.82%	1.84%	96.8%	3.2%	0.0%
vpr	15.66%	1.02%	94.9%	5.1%	0.0%
SPECINT average	*17.25%*	*1.35%*	*84.4%*	*9.2%*	*6.5%*
excel	27.30%	2.77%	61.8%	22.7%	15.5%
photoshp	12.64%	1.57%	67.3%	28.7%	3.8%
powerpnt	22.93%	1.51%	71.4%	1.5%	25.9%
winword	22.96%	2.48%	94.3%	2.8%	2.9%
desktop average	*21.46%*	*2.08%*	*73.7%*	*13.9%*	*12.0%*
IIS inetinfo	26.20%	3.72%	62.5%	4.6%	32.9%
IIS dllhost	18.20%	3.16%	64.4%	2.8%	32.8%
apache	20.80%	2.62%	67.2%	2.6%	30.2%
sqlservr	20.02%	3.71%	75.2%	8.4%	16.4%
server average	*21.30%*	*3.30%*	*67.3%*	*4.6%*	*28.1%*

Table 7.4: Indirect branch statistics. The first column gives the percentage of both direct and indirect branches out of the total dynamic instruction count. The second column gives the percentage of just the indirect branches. The final three columns break down these indirect branches into their three instruction types.

server benchmarks have more branches and indirect branches than the SPECINT benchmarks. The indirect branch type breakdowns show the Windows applications' extensive use of shared libraries in increased indirect call frequencies. However, return instructions are the most common type across all benchmark groups.

7.1.1 Measurement Methodology

For all of our reported performance numbers, the median time from an odd number of runs (from three to seven) is used. Since we are not running in a simulator or an emulator, but on a real-world, complex processor, we see a noticeable amount of noise in our runs. Performing multiple runs and taking a median helps remove some of the noise, but keep in mind that discrepancies of one or even two percent are to be expected and cannot be taken to be statistically significant.

All of our Pentium 3 runs were performed on a Dell PowerEdge 6400 machine with four Intel Xeon 700 MHz processors and 2 GB RAM. Our Pentium 4 numbers were gathered on a SuperMicro SuperServer 6012P-6 machine with two Intel Xeon 2.2 GHz processors and 2 GB RAM. We found wall-clock time on an unloaded machine to be the best measurement method. Unless otherwise stated, all numbers given throughout this thesis were gathered on a Pentium 4 for our Linux numbers, a Pentium 3 for our desktop numbers, and a Pentium 4 for our server numbers. Our operating systems were RedHat Linux 7.2, Windows 2000 Professional, and Windows 2000 Advanced Server.

Where averages of time ratios are given (as in all performance graphs), the harmonic mean is used, as it is best suited to averaging ratios. For averages of other numbers (in our tables), the arithmetic mean is used. When statistics on particular runs are shown, the longest run for each SPEC CPU benchmark with multiple datasets is chosen.

Our server benchmarks were not used for all of our performance evaluations throughout this thesis. Time constraints are the major factor, as the server benchmarks are more complicated to run. They are also the least computationally intensive, making them less relevant for some design studies. We spent more time analyzing those applications on which we perform poorly than on the server applications.

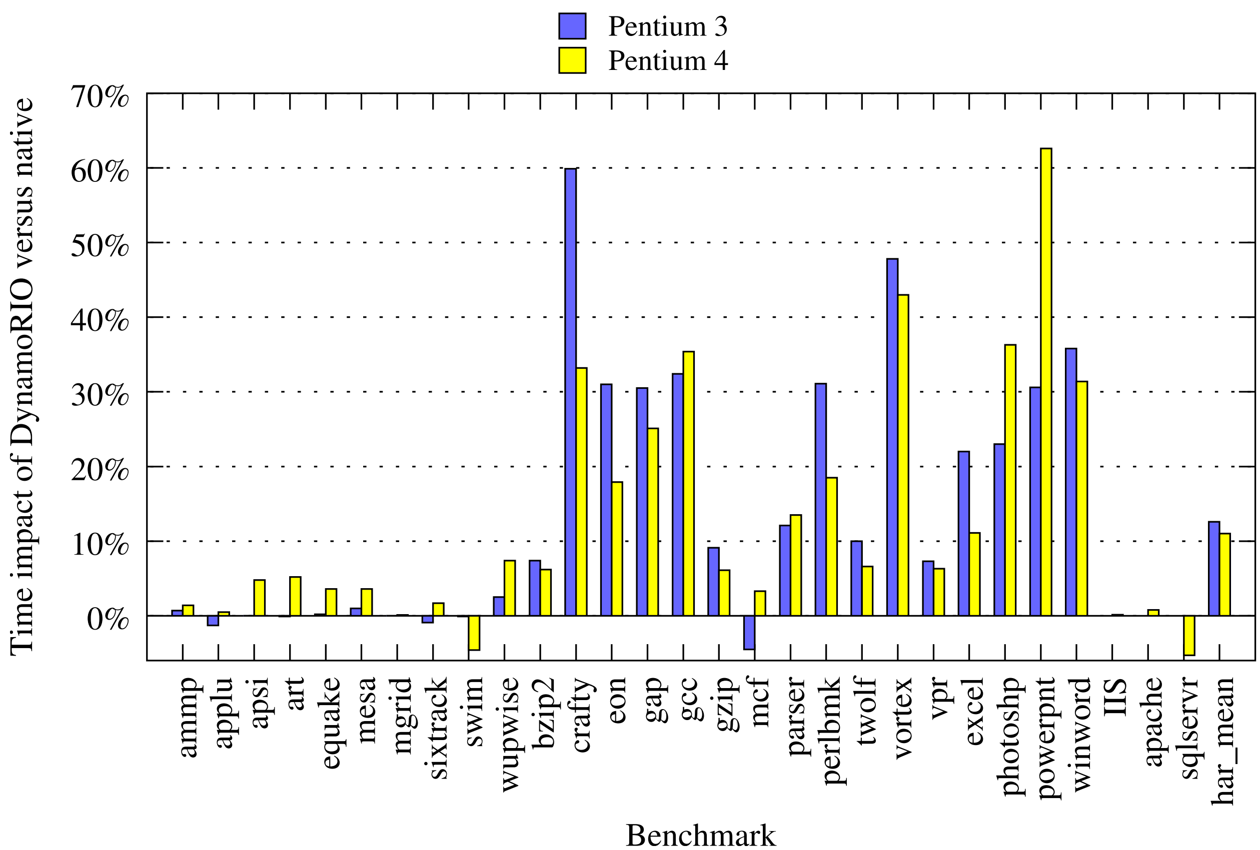

Figure 7.5: Base performance of DynamoRIO on on both the Pentium 3 and Pentium 4. (Our server benchmarks are more complicated to set up and run, so we only have their results for the Pentium 4.) No application code optimizations beyond code layout in building traces are performed in the base DynamoRIO system. We have used our tool interface to apply aggressive optimizations to the SPEC CPU2000 benchmarks, masking DynamoRIO's overhead on a few integer benchmarks and even enabling the floating-point benchmarks to surpass native speed (see Section 9.2).

7.2 Performance Evaluation

Figure 7.5 shows the base performance of DynamoRIO on both the Pentium 3 and Pentium 4, with the average overhead for each group of benchmarks displayed in Table 7.6. (Memory usage is discussed in Chapter 6.) On floating-point applications we have very low overheads, less than three percent on average, due to the fact that these benchmarks spend all of their time in a small number of loops. These loops quickly materialize in our trace cache and we then execute at or faster than native speed (faster because of the superior code layout of traces). Integer benchmarks are more problematic for DynamoRIO, with average overheads around twenty percent. Not only

Benchmark suite	Average slowdown Pentium 3	Pentium 4
SPECFP	0.2%	2.3%
SPECINT	20.3%	16.6%
desktop	27.6%	32.9%
server	N/A	-1.5%

Table 7.6: DynamoRIO's average overhead on each of our four benchmark suites, on both the Pentium 3 and the Pentium 4.

do they execute a wide range of code, rather than a few hot loops, but they usually include a higher percentage of indirect branches (Table 7.4). Sections 4.2 and 4.3 discuss why indirect branch performance is such a problem for DynamoRIO and our work on addressing the issue. Our desktop benchmarks present problems similar to those of the integer programs, and have our worst performance, averaging thirty percent overhead. These desktop applications use mostly integer arithmetic and, even more than SPECINT, have few hot spots that dominate their execution. DynamoRIO does well on server benchmarks, meeting native performance, mainly because they are not as CPU-intensive as the other benchmarks. In general, DynamoRIO adds performance overhead in the zero to thirty percent range for computationally-intensive applications. Improving this further is future work. Optimizing application code can mask DynamoRIO overhead, as we show with aggressive optimizations using our tool interface in Section 9.2. These optimizations even exceed native performance for our floating-point benchmarks, with an average twelve percent speedup and a forty percent speedup on `mgrid`.

7.2.1 Breakdown of Overheads

We used program counter sampling (Section 7.3.1) to break down where time is spent in DynamoRIO. Figure 7.7 gives the average breakdown across our Linux benchmarks (since we do not have program counter sampling implemented on Windows). Counter-intuitively, what at first glance are the heavyweight portions of DynamoRIO, such as the basic block builder, barely show up as overhead (they fall into the Other category, which is broken down in Table 7.9). Once the working set of the application is in the code cache, little time is spent outside of the cache. In the cache, traces are quite effective at capturing hot code, as the amount of time spent in basic blocks

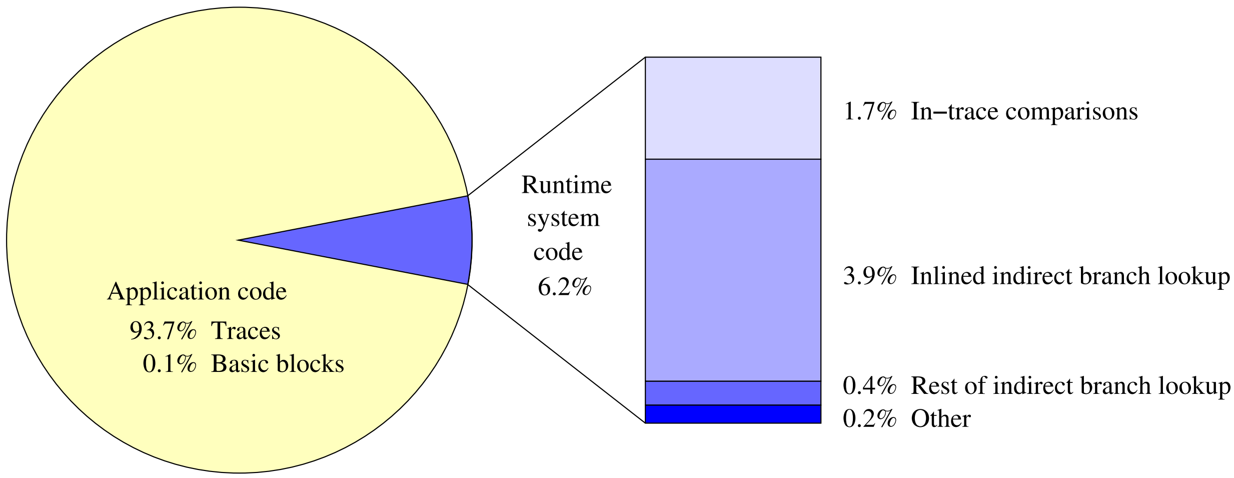

Figure 7.7: Breakdown of where time is spent, divided into two categories, application code and DynamoRIO code plus all code inserted into fragments in the code cache that would not be executed in a native run of the application. These are the average numbers across our Linux benchmarks. For the specific numbers for each benchmark, see Table 7.8; for a breakdown of the Other category, see Table 7.9.

instead of traces is negligible.

The majority of direct overhead added by DynamoRIO is from handling indirect branches, as much more work must be done than when a native indirect branch executes (see Section 4.2). Even when inlined into a trace, a comparison must be performed to ensure that the dynamic target of the branch stays on the trace (Section 2.3.2), contributing an average of nearly two percent of execution time. Otherwise, a hashtable lookup is required to identify the target. DynamoRIO optimizes this lookup by partially inlining it (Section 4.3.1). The inlined portion accounts for about four percent of average execution time. The rest of the lookup routine, and the entire rest of DynamoRIO, together form less than one percent of execution time.

The most striking conclusion from this data is that we have achieved extraordinary success at spending time in application code rather than DynamoRIO code. This is the primary performance goal of a code caching system, to maximize time spent in the code cache. However, adding up the time spent in DynamoRIO-added code does not explain the entire wall-clock slowdowns of these benchmarks. Application code must be executing more slowly in our code cache than it does natively. Performance counters (Section 7.3.2) reveal that extra data cache pressure from

DynamoRIO's own data structures (primarily our indirect branch lookup table) is the root of the problem, although it is surprisingly difficult to get all overhead measurements to add up to the exact observed wall-clock slowdowns. Our relative slowdown is less on machines with larger caches, further supporting the data cache explanation. One area of future work is in further minimizing DynamoRIO's data cache pressure. Open-address hashing (see Section 4.3.3) is one step in that direction that we have implemented.

Table 7.8 shows the detailed program counter sampling results for each Linux benchmark, in the same categories as Figure 7.7. The first two columns give the percentage of samples received in code that would have been executed natively: the application code copied into traces and basic blocks. The final four columns break down the time spent in instructions that DynamoRIO adds beyond native execution. The breakdown of the Other category is given in Table 7.9, which shows that most of the time is spent building basic blocks and traces, as expected.

7.2.2 Impact of System Components

Table 7.10 summarizes the performance impact of each major design decision we faced in building DynamoRIO. The table divides the designs up based on whether we chose to use or discard each feature. The basic features (Chapter 2) bring the most dramatic performance improvements beyond a simple interpreter: a basic block cache, direct linking, indirect linking, and traces. Eliding unconditionals (Section 2.4), various hashtable optimizations (Section 4.3), and condition code preservation choices (Section 4.4) also have significant impact. There were several design choices that resulted in performance improvements but that we had to discard due to transparency problems.

7.3 Profiling Tools

This section discusses the tools we have built both for profiling DynamoRIO itself (Section 7.3.1 and Section 7.3.2) and for using DynamoRIO as a tool for profiling applications (Section 7.3.3).

Benchmark	Traces	Blocks	Cmp-in-trace	Inlined IBL	Rest of IBL	Rest of system
ammp	99.66%	0.00%	0.27%	0.05%	0.00%	0.02%
applu	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%
apsi	97.97%	0.00%	1.91%	0.11%	0.00%	0.01%
art	99.98%	0.00%	0.00%	0.00%	0.00%	0.02%
equake	99.05%	0.00%	0.91%	0.01%	0.00%	0.04%
mesa	97.91%	0.05%	1.40%	0.60%	0.00%	0.04%
mgrid	99.99%	0.00%	0.00%	0.00%	0.00%	0.00%
sixtrack	99.82%	0.00%	0.08%	0.06%	0.00%	0.04%
swim	99.99%	0.00%	0.00%	0.00%	0.00%	0.01%
wupwise	97.87%	0.00%	0.88%	1.20%	0.02%	0.04%
bzip2	97.70%	0.32%	1.30%	0.65%	0.00%	0.03%
crafty	81.52%	0.00%	3.07%	13.17%	2.01%	0.23%
eon	90.37%	0.00%	3.74%	5.52%	0.13%	0.25%
gap	79.89%	0.00%	6.92%	11.79%	1.02%	0.38%
gcc	79.45%	0.22%	3.15%	13.22%	1.80%	2.16%
gzip	90.66%	1.69%	2.06%	5.51%	0.00%	0.09%
mcf	99.89%	0.00%	0.05%	0.04%	0.00%	0.02%
parser	91.57%	0.01%	2.66%	4.84%	0.73%	0.19%
perlbmk	78.98%	0.07%	5.28%	13.83%	1.18%	0.66%
twolf	97.11%	0.00%	0.88%	1.81%	0.16%	0.04%
vortex	83.75%	0.02%	2.83%	11.76%	1.40%	0.24%
vpr	97.39%	0.01%	0.66%	1.43%	0.42%	0.09%
average	93.66%	0.11%	1.73%	3.89%	0.40%	0.21%

Table 7.8: A breakdown of where time is spent on our Linux benchmarks on the Pentium 4, obtained via program counter sampling (Section 7.3.1). The first two columns show time spent in application code that has been copied into either a trace or a basic block. The rest of the columns indicate overhead, time spent in portions of the system that would not occur if the application were executed natively. The *cmp-in-trace* column gives the time spent in comparisons inserted when indirect branches occur in the middle of traces (Section 2.3.2). The next two columns show a breakdown of the indirect branch lookup (IBL) routine (Section 4.3): the portion inlined into the exit stubs of traces and the shared tail of the lookup if the inlined portion misses. The final column gives the time spent in the entire rest of DynamoRIO (copying code, cache management, etc. — *everything* else). For a breakdown of this final column time into the components of DynamoRIO, see Table 7.9. These results show that we are very successful at spending time in application code rather than DynamoRIO code. However, the numbers for the final four columns do not add up to the overall slowdown, meaning that application code in our code cache executs more slowly than it does natively. This is due to data cache misses (because our hashtable adds data cache pressure), as far as we can tell.

Benchmark	Basic block builder	Dispatch	Trace builder	Context switch
ammp	30.00%	0.00%	70.00%	0.00%
applu	25.00%	8.33%	66.67%	0.00%
apsi	34.78%	4.35%	60.87%	0.00%
art	62.50%	12.50%	25.00%	0.00%
equake	50.00%	16.67%	33.33%	0.00%
mesa	38.46%	7.69%	53.85%	0.00%
mgrid	25.00%	8.33%	66.67%	0.00%
sixtrack	35.90%	10.26%	51.28%	2.56%
swim	33.33%	11.11%	55.56%	0.00%
wupwise	35.29%	11.76%	47.06%	5.88%
bzip2	66.67%	0.00%	33.33%	0.00%
crafty	12.07%	5.17%	53.45%	29.31%
eon	68.18%	9.09%	22.73%	0.00%
gap	17.72%	6.33%	40.51%	35.44%
gcc	29.44%	8.88%	56.54%	5.14%
gzip	80.00%	0.00%	20.00%	0.00%
mcf	33.33%	33.33%	33.33%	0.00%
parser	15.38%	4.62%	35.38%	44.62%
perlbmk	35.29%	10.29%	48.53%	5.88%
twolf	27.59%	13.79%	44.83%	13.79%
vortex	45.83%	2.08%	29.17%	22.92%
vpr	23.53%	0.00%	52.94%	23.53%
average	37.51%	8.39%	45.50%	8.59%

Table 7.9: A breakdown of where time is spent when inside of DynamoRIO on our Linux benchmarks, obtained via program counter profiling. This is a breakdown of the small amount of time spent in DynamoRIO itself, rather than the code cache (see Table 7.8 for the overall time breakdown). The small numbers of sampling hits show up as artifacts in the integral percentages in some categories.

7.3.1 Program Counter Sampling

To analyze where time is spent in DynamoRIO, we use program counter sampling. We have only implemented this on Linux, as there is no way to do so on Windows without writing a kernel driver (Windows' asynchronous message-passing style does not allow for precise periodic interruption of oneself) or using Native API [Nebbett 2000] system calls that only work on checked builds on Windows 2000 (though they do work on Windows XP and Windows 2003). On Linux, program

Fate	Feature	Time Impact				Section
		Harmonic Mean			Abs	
		SPECFP	SPECINT	Desktop	Max	
Used	Basic block cache (base perf)	3.5x	17.2x	11.3x	37.8x	2.1
	Link direct branches	-81.8%	-94.7%	-91.5%	-97.2%	2.2
	Link indirect branches	-40.7%	-70.1%	-65.6%	-84.8%	2.2
	Separate direct stubs	-0.4%	-0.9%	-2.6%	-8.6%	2.2
	Traces	-3.2%	-13.6%	-17.1%	-33.9%	2.3
	Trace head incr in cache	0.2%	-0.1%	-3.0%	-7.7%	2.3.2
	Elide unconditional branches	-0.9%	-0.2%	-5.4%	-13.5%	2.4
	Adaptive instruction rep.	-0.3%	-0.3%	-5.6%	-9.0%	4.1
	Inlined hashtable lookup	-0.8%	-3.2%	-1.0%	-13.9%	4.3.1
	Open-address hashtable	0.1%	-5.6%	-0.2%	-14.4%	4.3.3
	`lea/jecxz + lahf/seto`	-10.1%	-25.1%	-15.5%	-79.4%	4.4
Rejected	NET traces	1.2%	2.5%	3.6%	10.7%	2.2
	Code cache return addresses	-1.3%	-7.1%	N/A	-17.6%	4.2.1
	Software return stack	8.9%	22.3%	N/A	46.9%	4.2.1
	Call hashtable lookup routine	-0.2%	0.5%	-1.1%	-6.4%	4.3.1
	`pushf` for trace compare	10.1%	25.1%	15.5%	79.4%	4.4
	`sahf` for trace compare	6.5%	7.2%	3.7%	25.4%	4.4
	Avoid `eflags` in lookup	-0.3%	-0.4%	N/A	7.2%	4.4
	Ignore `eflags` in lookup	-0.3%	-3.1%	N/A	-7.4%	4.4
	Lazy linking	0.9%	0.5%	0.6%	5.2%	4.5.1
Profiling	Exit count profiling	8.8%	39.2%	18.2%	94.4%	7.3.3
	Program counter sampling	0.2%	1.0%	N/A	9.7%	7.3.1

Table 7.10: Performance summary of the major design decisions in building DynamoRIO. The top group lists those designs we chose to implement. The second group contains features that we discarded, either due to poor resulting performance, or because of transparency problems (see the indicated section for details). The final group contains our profiling mechanisms. For each decision we give the harmonic mean of its relative performance impact on three groups in our benchmark suite (see Section 7.1), as well as the maximum impact (by absolute value) on any single benchmark. The first row gives the absolute performance with respect to native performance, rather than relative impact, while the subsequent design decisions reflect performance impact relative to a system that has implemented the previously chosen decisions. The `lea/jecxz + lahf/seto` and `pushf` decisions are two sides of the same coin, with performance measured against each other.

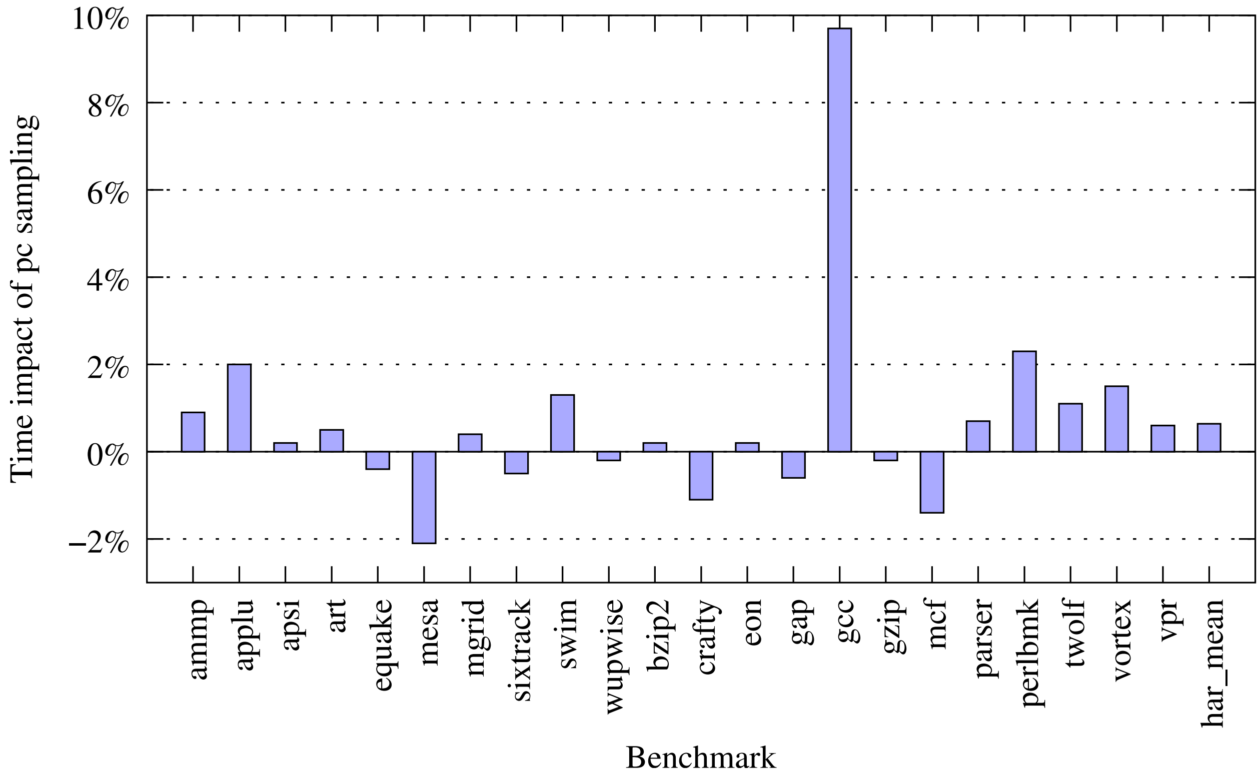

Figure 7.11: Performance impact of program counter sampling on our Linux benchmarks.

counter sampling is simple to implement using the interval timers provided by the kernel. We use the `ITIMER_VIRTUAL` timer, which counts down only when the process is executing and delivers a signal when it expires. Our handler for that signal records the program counter of the process at the time the signal was delivered. We sample the program counter every ten milliseconds. This is a low-overhead profiling method, with slowdowns that are mostly in the noise, as shown in Figure 7.11. The slowdown can be alleviated further by reducing the sampling frequency (e.g., sampling every twenty milliseconds brings `gcc`'s overhead down to five percent).

Example output from our program counter sampling is shown in Figure 7.12. The samples are divided into categories and a breakdown is given. Then a detailed listing of the different fragments shows how many samples each received. Table 7.8 in Section 7.2.1 shows the results of using program counter sampling to determine where time is spent in major portions of DynamoRIO for our Linux benchmarks.

We tried using VTune [Intel VTune Performance Analyzer] to obtain program counter sampling

```
ITIMER distribution (20120):
    0.0% of time in INTERPRETER (7)
    0.0% of time in DISPATCH (1)
    0.0% of time in MONITOR (6)
    0.2% of time in CONTEXT SWITCH (43)
    2.7% of time in INDIRECT BRANCH LOOKUP (535)
   97.1% of time in FRAGMENT CACHE (19527)
    0.0% of time in UNKNOWN (1)

BREAKDOWN:
    1 hit(s) =  0.0% for trace 0x08053f50
    1 hit(s) =  0.0% for trace 0x0806005b
    1 hit(s) =  0.0% for trace 0x0805f707
  ...
  535 hit(s) =  2.7% for DynamoRIO indirect_branch_lookup
  556 hit(s) =  2.8% for trace 0x08050ef0
  556 hit(s) =  2.8% for trace 0x08050b2f
  614 hit(s) =  3.1% for trace 0x08052634
  696 hit(s) =  3.5% for trace 0x40097b22
  743 hit(s) =  3.7% for trace 0x08050d30
  858 hit(s) =  4.3% for trace 0x08051210
  956 hit(s) =  4.8% for trace 0x0805141f
  991 hit(s) =  4.9% for trace 0x08054080
 1002 hit(s) =  5.0% for trace 0x08051294
 1125 hit(s) =  5.6% for trace 0x08052310
 1142 hit(s) =  5.7% for trace 0x08051224
 3313 hit(s) = 16.5% for trace 0x08050d65
```

Figure 7.12: Program counter sampling output for a sample benchmark run. There were 20,120 total program counter samples collected, nearly all in traces.

data on Windows. However, we had problems exporting its data for automated analysis of time spent in the code cache, which is a black box to VTune. VTune showed that the overhead of basic block building shows up more on our desktop Windows benchmarks, because they have less code re-use and spend more time executing new code.

7.3.2 Hardware Performance Counters

Intel processors have hardware performance counters [Intel Corporation 2001, vol. 3] that enable detailed profile information gathering. Figure 7.13 shows the results of comparing performance

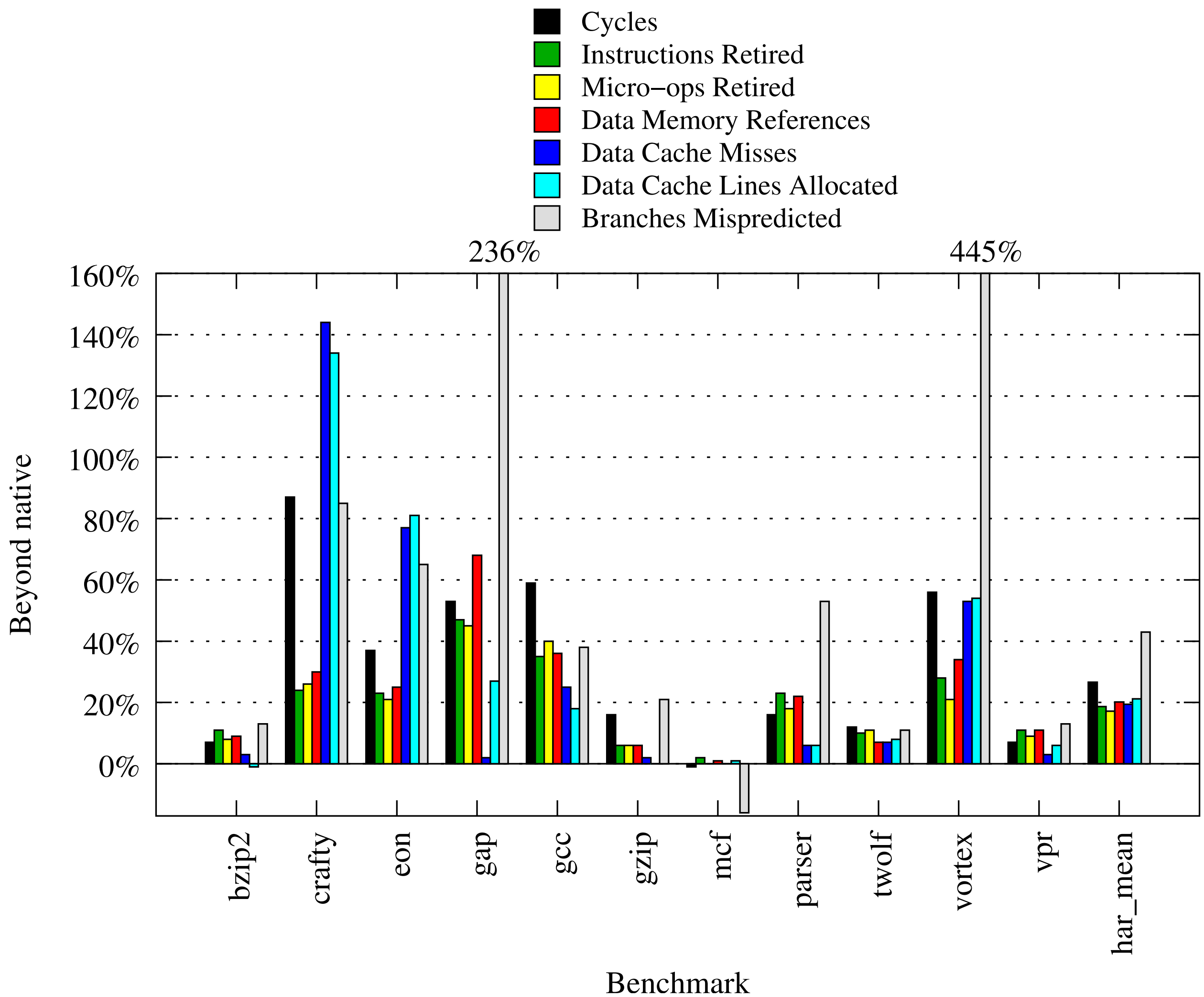

Figure 7.13: Hardware performance counter profiling data on the SPECINT benchmarks. The extra counts for each of seven events when the benchmark is run under DynamoRIO are shown.

counter counts for the integer benchmarks from SPEC CPU2000 [Standard Performance Evaluation Corporation 2000] running under DynamoRIO versus running natively. The floating point benchmarks are not included in this graph in order to highlight the interesting cases, as the floating point applications have little added counts beyond their native runs.

The figure shows that we are adding to the total number of instructions executed by about twenty percent on average. However, for some benchmarks our execution overhead goes well beyond the added instructions. Consider `crafty`, where we are executing 87% more cycles than a native run (we have improved DynamoRIO since these numbers were gathered, as a comparison with Figure 7.5 shows) but only 24% more instructions and 26% more micro-operations. The difference must be coming from pipeline stalls due to cache misses or branch mispredictions. The data cache misses are more than double in the DynamoRIO run compared to the native run, due to the hashtable lookup on each indirect branch. On `gap`, data cache pressure is not as high (a number of extra cache lines are brought in without impacting the application), but branch mispredictions are very high. Other performance counters indicate that these are indirect branch mispredictions.

Hardware performance counters are essential for gathering information on micro-architectural performance bottlenecks. The results from our use of them has born out our analyses throughout this thesis, that our performance is hindered by indirect branch prediction (Section 4.2) and by our hashtable lookup's data cache impact (Section 4.3.3). However, it has been surprisingly difficult to explain the exact slowdown for every benchmark. For example, `gcc` has an approximately 30% slowdown in Figure 7.5. Table 7.8 indicates that 21% of the time is spent outside of `gcc` code. Yet the performance counter data here shows that data cache and branch prediction misses are comparable to instructions added. Is the other 9% explainable by data caches and branch mispredictions? We do not have satisfying quantitative proof.

7.3.3 Trace Profiling

We developed several methods of profiling traces in order to understand the efficacy of our trace design. These same methods can be used to understand where time is spent in the application, as a general profiling tool.

Below we discuss the two methods we fully implemented: directly measuring time in traces and incrementing counters on trace exits. We also toyed with recording patterns of paths and using online compression to store sequences. Related work includes compression of whole-program paths [Larus 1999].

Time Stamp Counter

One technique we tried was to record the number of cycles spent in each trace. We used the IA-32 instruction `rdtsc` [Intel Corporation 2001, vol. 2] to read the processor's time stamp counter at the top of each trace. We then took the difference since the last reading, at the top of the previous trace, and credited that amount of time to the previous trace. We also kept an execution count for each trace.

This profiling method is heavyweight. Using the `rdtsc` instruction transparently requires saving and restoring two registers, and the execution counter increment and time computations require preserving the `eflags` register. Additionally, the `rdtsc` instruction itself is expensive, taking approximately thirty cycles on a Pentium 3 and in excess of seventy cycles on a Pentium 4. We tried computing the typical amount of time spent in the profiling instructions themselves and subtracting that amount per execution of each trace in order to arrive at the actual time spent in the trace. However, this was never accurate enough for absolute values.

We were able to use time stamp profiling results as relative measures, which worked well for identifying hot traces. However, the profiling overhead was much too high, as high as a four or five times slowdown, and it said nothing about how the traces connected to each other. It is best to have more information on each trace than simply its execution frequency. Since a trace has multiple exits, and some may be indirect branches, we looked at ways to record the hot connections between traces. One method is to record the top n predecessors and successors for each trace, along with each transition's frequency. We tried adding this to our time stamp counter method, making implementation easy by calling a shared `C` routine from the top of each trace, observing the time stamp counter before and after the routine so that it executes with time stopped. However, this only added more overhead. In the end we switched to the scheme described in the next section.

Exit Counters

Since we are interested in the transitions between traces, we implemented a simple profiling scheme where a counter is associated with each exit from a trace. We located these counter increments in the exit stubs and linked trace exits *through the stubs*. This illustrates the power of the exit stubs for periodic profiling. Instrumentation code is placed in the stub, where it can either be targeted or bypassed by pointing the trace exit branch to the stub or directly to its target trace (or indirect branch lookup routine). Turning the instrumentation on or off is then an atomic operation since it involves changing a single jump operand (just like linking). Unlinking the exit altogether requires a separate entry in the exit stub for the code to return control to DynamoRIO, which can also be targeted atomically. (Even with thread-private code caches, atomic unlinking is important for cache consistency, as described in Section 6.2.)

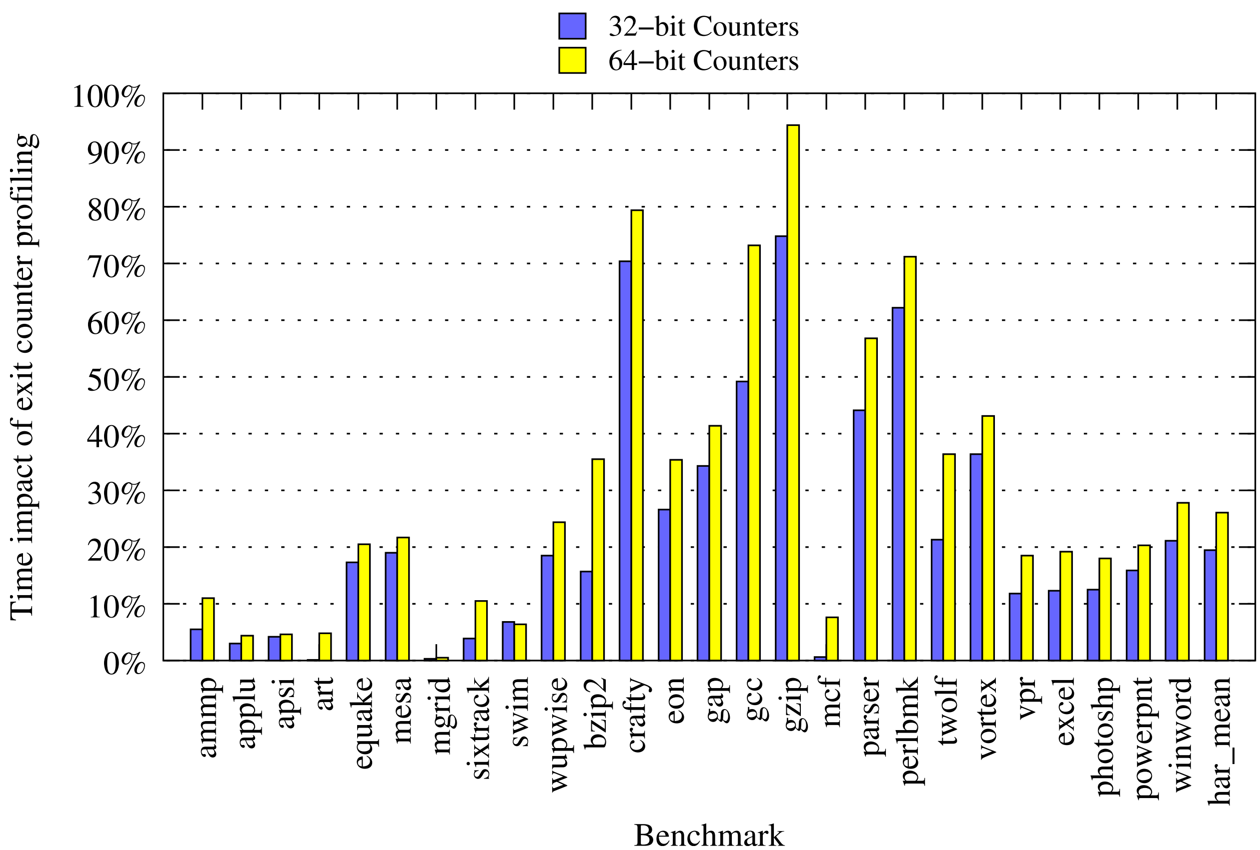

Figure 7.14: Performance impact of exit counter profiling. 64-bit counters are needed for long-running traces, but carry higher overhead.

Exit counters are a convenient method for measuring completion frequencies of traces. Such data for our benchmarks is given in Table 2.16. We implemented exit counters in DynamoRIO both as 32-bit and 64-bit counters. 32-bit counters are more efficient, but they overflow in programs with long-running traces, which happens in several of the SPEC CPU2000 benchmarks (`ammp`, `mgrid`, `sixtrack`, and `gzip`). Figure 7.14 gives the overhead of both 32-bit and 64-bit counters on our benchmark suite. Both are higher than program counter sampling, but much lower than time stamp counter profiling.

Sample exit stubs with 64-bit counters are shown in Figure 7.15. The figure illustrates an optimization that we employ for direct branches. When we emit a trace we may not have examined the target of each direct branch, so we emit a stub that performs a full save of the arithmetic flags (see Section 4.4). When we later link that direct exit, however, we examine its target for whether or not it writes these flags before reading them. If it does, we do not need to bother preserving them around our counter increment. Table 7.16 shows the percentage of exit stubs for which we must save the flags. It gives both a static count of stub locations and a dynamic count that indicates the number of executions of flags saves. Statically, the average is about one in eight stubs that needs its flags preserved; but dynamically, only one in sixteen executions of a stub requires flags savings. Omitting the flag preservation is a big boost in performance, as the final column in Table 7.16 shows (see Section 4.4 for a discussion of `eflags` expenses on IA-32). We currently optimize for all six arithmetic flags at once. Given our flag preservation scheme that separates out the overflow flag, we could optimize for fragments that write it but not the other flags.

Unfortunately, our `eflags` optimization makes the first link not atomic, since it must overwrite several instructions in the stub. However, the first link of a direct exit is always requested by the thread owning the target fragment while in DynamoRIO code and not in the code cache, making the lack of atomicity acceptable for thread-private fragments. Thread-shared fragments would require some sort of synchronization of the linked fragment. All subsequent links and unlinks are carried out atomically since they only modify the exit branch itself (because of the separate unlinked entry point in the exit stubs — see Figure 7.15).

Exit counter profiling gives us the execution frequency of every exit from a trace. In order to calculate total time spent in a trace, we can post-process the data and multiply each exit count by the sum of estimated execution times of each instruction on the path from the fragment entrance to that particular exit. This could be done with a static table of instruction cycle counts. In our analysis we simply count each instruction as equal, and the results are still accurate enough to match the ordering of hot traces found by program counter sampling.

Figure 7.17 gives two examples of output from exit counter profiling. The top example shows a benchmark that is amenable to this technique, with a handful of very hot traces that make up nearly all of the program's execution time and contain only direct branches. The bottom example shows the problem with exit counter profiling: it does not identify the targets of indirect branches. We know the frequency with which an indirect exit was taken, but we do not know where it went.

```
-------- exit stub 0: -------- <target: 0x08058241>
0x4053083c   a3 c0 e8 09 40       mov    %eax -> 0x4009e8c0
0x40530841   0f 90 05 0b e9 09 40 seto   -> 0x4009e90b
0x40530848   9f                   lahf   -> %ah
0x40530849   83 05 7c 6b 4f 40 01 add    $0x01 0x404f6b7c -> 0x404f6b7c
0x40530850   83 15 80 6b 4f 40 00 adc    $0x00 0x404f6b80 -> 0x404f6b80
0x40530857   81 05 08 e9 09 40 00 add    $0x7f000000 0x4009e908 -> 0x4009e908
             00 00 7f
0x40530861   9e                   sahf   %ah
0x40530862   a1 c0 e8 09 40       mov    0x4009e8c0 -> %eax
0x40530867   e9 b0 0a 01 00       jmp    $0x4054131c <fragment 4304>
0x4053086c   a3 c0 e8 09 40       mov    %eax -> 0x4009e8c0
0x40530871   b8 5c 6b 4f 40       mov    $0x404f6b5c -> %eax
0x40530876   e9 45 38 d4 ff       jmp    $0x402740c0 <fcache_return>
...
-------- exit stub 2: -------- <target: 0x08057ee1>
0x405308ba   83 05 cc 6b 4f 40 01 add    $0x01 0x404f6bcc -> 0x404f6bcc
0x405308c1   83 15 d0 6b 4f 40 00 adc    $0x00 0x404f6bd0 -> 0x404f6bd0
0x405308c8   e9 42 f6 04 00       jmp    $0x4057ff0f <fragment 5509>
0x405308cd   90                   nop
0x405308ce   83 15 d0 6b 4f 40 00 adc    $0x00 0x404f6bd0 -> 0x404f6bd0
0x405308d5   81 05 08 e9 09 40 00 add    $0x7f000000 0x4009e908 -> 0x4009e908
             00 00 7f
0x405308df   9e                   sahf   %ah
0x405308e0   b8 ac 6b 4f 40       mov    $0x404f6bac -> %eax
0x405308e5   e9 d6 37 d4 ff       jmp    $0x402740c0 <fcache_return>
0x405308ea   a3 c0 e8 09 40       mov    %eax -> 0x4009e8c0
0x405308ef   b8 ac 6b 4f 40       mov    $0x404f6bac -> %eax
0x405308f4   e9 c7 37 d4 ff       jmp    $0x402740c0 <fcache_return>
```

Figure 7.15: Example direct exit stub code for exit counter profiling with 64-bit counters, which require two instructions to increment, an `add` and an `adc`. These two instructions operate on a 64-bit counter stored in memory (the `0x404f6b..` addresses). Both stubs are currently linked, but the first stub's target does not write the six conditional flags that the counter increment modify, and so those flags must be saved and restored around the increment (see Section 4.4 for information about saving and restoring these flags). The second stub's target does write those flags, and so it has been optimized in place, leaving dead code up until the unlinked entry point (the unlinked path is comprised of the final three instructions).

Benchmark	Static must-save %	Dynamic must-save %	Time impact
ammp	11.2%	0.2%	-24.7%
applu	9.6%	0.0%	-12.7%
apsi	20.6%	1.0%	-16.6%
art	6.0%	0.0%	-17.2%
equake	7.5%	25.3%	-14.3%
mesa	14.1%	22.5%	-18.6%
mgrid	11.9%	0.0%	-3.8%
sixtrack	13.3%	0.1%	-24.5%
swim	12.3%	0.0%	-8.1%
wupwise	19.8%	6.3%	-36.5%
bzip2	5.8%	0.4%	-41.8%
crafty	8.7%	9.5%	-18.6%
eon	13.3%	10.7%	-30.5%
gap	14.2%	6.6%	-28.8%
gcc	14.2%	5.5%	-27.7%
gzip	9.1%	6.4%	-53.1%
mcf	11.3%	0.2%	-11.4%
parser	9.4%	5.4%	-36.9%
perlbmk	17.2%	4.1%	-27.7%
twolf	11.8%	5.4%	-28.2%
vortex	4.8%	0.9%	-13.0%
vpr	8.8%	4.6%	-19.4%
excel	19.1%	14.9%	-13.9%
photoshp	5.8%	0.2%	-12.4%
powerpnt	15.4%	6.7%	-3.1%
winword	18.6%	10.8%	-7.2%
average	12.1%	5.7%	-23.3%

Table 7.16: Percentage of direct exit stubs whose targets do not write the six arithmetic flags in the `eflags` register, requiring their preservation across the exit counter increment. The first column gives the static count of such stubs, while the second column gives the dynamic percentage of exits taken that needed the flags saved. The final column gives the time impact of optimizing by only saving the flags when necessary.

```
ammp:
Frag #  1563 (0x08056fb9) = size  1138, count  586.58 M, time 519725.58 (35.9%)
        Stub # 0:  4.6% => 1571
        Stub # 1: 92.5% => 1562
        Stub # 2:  2.9% => 1579
Frag #  1537 (0x08056501) = size  2618, count  262.80 M, time 385033.97 (26.6%)
        Stub # 0:  8.4% => 1575
        Stub # 1: 29.4% => 1572
        Stub # 2: 62.2% => 1536
Frag #  1533 (0x080564f0) = size   230, count 6518.63 M, time 120365.27 ( 8.3%)
        Stub # 0:  3.7% => 1537
        Stub # 1: 96.0% => 1533
Frag #  1641 (0x08058002) = size   416, count  296.34 M, time  79473.90 ( 5.5%)
        Stub # 0: 96.9% => 1641
        Stub # 1:  3.1% => 1644

gap:
Frag #   672 (0x08069723) = size  1412, count  260.78 M, time  78234.26 (10.0%)
        Stub # 5: 100.0% => <indirect>
Frag #  3480 (0x0806aa66) = size   529, count  351.06 M, time  48653.97 ( 6.2%)
        Stub # 0: 72.4% => 3480
        Stub # 1:  1.0% => 3489
        Stub # 5: 26.4% => <indirect>
Frag #  2653 (0x0808d4f5) = size  3921, count  132.80 M, time  42144.60 ( 5.4%)
        Stub # 1:  8.8% => 2680
        Stub # 3:  4.5% => <indirect>
        Stub #14:  2.3% => 3982
        Stub #15: 84.1% => <indirect>
```

Figure 7.17: Example output from post-processing exit counter results. The top example is from `ammp`, while the bottom is from `gap`, both from SPEC CPU2000 [Standard Performance Evaluation Corporation 2000].

7.4 Chapter Summary

This chapter presented the performance results of DynamoRIO on several sets of benchmarks. DynamoRIO performs well on floating-point applications and servers. DynamoRIO's overhead is noticeable on some integer applications and desktop applications with many indirect branches or less code reuse than the other benchmarks. We analyzed the reasons our performance does not match native for every application: a combination of indirect branch misprediction penalties along with added instructions and data cache pressure for performing indirect branch lookups. We also presented the profiling tools that helped us get our performance to where it is, and other tools that can be used to profile applications. This chapter has focused on time, while the previous chapter showed memory results. In the next chapter we turn from the implementation of DynamoRIO to its interface for building custom runtime tools.

Chapter 8

Interface for Custom Code Manipulation

One of our primary goals is to make DynamoRIO a customizable platform on which to build runtime tools. We know from experience with compilers that we must export an interface at the right abstraction level: a high-level tool should not have to worry about low-level details. But we have requirements that compilers do not have to worry about: efficiency and transparency. DynamoRIO's tool interface must minimize the overhead of all its actions, which our adaptive level-of-detail instruction representation (Section 4.1) helps with, as well as encourage tools to maintain transparency with respect to the application, by supplying them with DynamoRIO's own transparent heap, input/output routines, and other resources.

Our interface classifies runtime tools into two categories: those that operate on an entire program (e.g., general instrumentation) versus those that are only interested in frequently executed code (e.g., dynamic optimization). We abstract away low-level details and provide two access points to support these two types of usage. This frees the tool builder to focus on the code manipulation task at hand, rather than details of the runtime system. It also promotes sharing and re-use of tool components through standard, straightforward interfaces, while still providing full flexibility for any code transformation.

In this chapter, we first describe our model for tool building: the DynamoRIO client (Section 8.1). Next we present the Application Programming Interface (API) that we export to clients (Section 8.2), and give several client examples (Section 8.3). Finally, we discuss the limitations on clients (Section 8.4).

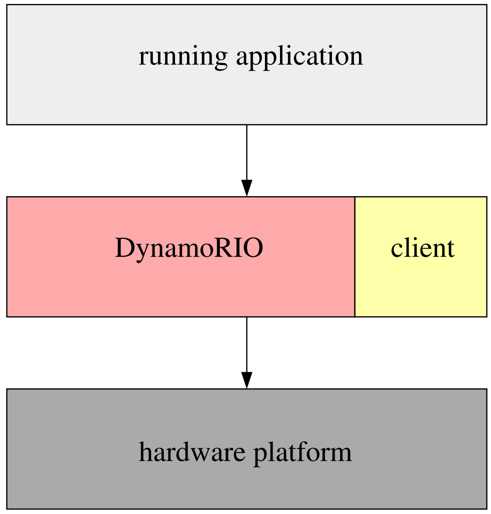

Figure 8.1: DynamoRIO and a custom *client* jointly operate on a target application, forming a custom runtime tool.

8.1 Clients

DynamoRIO can be extended with an external *client* that performs custom manipulation of the runtime code stream. A client is coupled with DynamoRIO to form a runtime tool; the two jointly operate on an input program (Figure 8.1). The client supplies specific hook functions that are called by DynamoRIO at appropriate times during execution of the program. The client can also use a rich Application Programming Interface (API) provided by DynamoRIO [MIT and Hewlett-Packard 2002] for manipulating instructions and acting transparently. Our API is described in Section 8.2. This section discusses modes of application control and the client hook routines. Example client code is given in Section 8.3.

8.1.1 Application Control

DynamoRIO's normal model of control is to execute every piece of a target application from start to finish. To extend DynamoRIO, a client is built as a shared library that is loaded in by DynamoRIO once it takes over an application (see Section 5.5). The client library path is specified as a runtime parameter to DynamoRIO (see MIT and Hewlett-Packard [2002] for specific information on the workflow for building and using a client). The client's hooks are then called throughout the execution of the application, allowing the client access to all of the application's executed code.

Control Routine	Description
`void dynamorio_app_init()`	Performs per-process initialization. Must be called before the application creates any threads or makes any other interface calls.
`void dynamorio_app_exit()`	Performs per-process cleanup.
`void dynamorio_app_start()`	Instructs DynamoRIO to start executing out of its code cache from this point onward (for the current thread only).
`void dynamorio_app_stop()`	Instructs DynamoRIO to stop executing out of its code cache (for the current thread only).
`void dynamorio_app_take_over()`	Calling this routine is similar to `dynamorio_app_start` except that all subsequent `dynamorio_app_` calls are ignored. This is useful for overriding existing `dynamorio_app_start()` and `dynamorio_app_stop()` calls.

Table 8.2: Rather than executing an entire application transparently, the explicit control interface can be used by inserting calls into the application's source code to specify which parts of it should be controlled by DynamoRIO.

Explicit Control Interface

An alternative model is for the application to be built specifically for execution under DynamoRIO's control (e.g., using DynamoRIO to optimize interpreters, as in Section 9.3). We export an *explicit control interface* to allow an application to initialize and launch DynamoRIO on its own. Table 8.2 lists the routines in the explicit control interface, which set up DynamoRIO and start and stop execution from its code cache. These routines must be inserted in the application itself by the user, usually through modifying application source code. Figure 8.3 gives an example of a program using the explicit control interface.

The initialization and termination routines should be placed at program startup and shutdown, though they can be anywhere so long as they are before and after, respectively, all other interface invocations. The start and stop routines are the heart of the explicit control interface. Execution of the `dynamorio_app_start()` routine causes control to be transfered to DynamoRIO, for the current application thread. The program will appear to return from this call with no visible side effects except those associated with the invocation of an empty function. However, the execution of the program now occurs within DynamoRIO's code cache. From this point onward, until the

```
#include "dynamorio.h"
void main() {
    int rc = dynamorio_app_init();
    assert(rc == 0);
    dynamorio_app_start();
    printf("Hello world, from the code cache!\n");
    dynamorio_app_stop();
    printf("Hello world, natively!\n");
    dynamorio_app_exit();
}
```

Figure 8.3: Example code showing use of the explicit control interface. The first `printf` routine will execute inside of DynamoRIO's code cache, while the second will execute natively.

stop routine is invoked, a client is able to monitor and transform the application's code. Execution continues within the code cache until DynamoRIO encounters a call to the `dynamorio_app_stop` routine, at which point the application thread will return to native execution.

Standalone Clients

We also support the use of DynamoRIO as a library of IA-32 instruction manipulation routines. In this mode, DynamoRIO does not control any application or execute anything from a code cache — it is simply a utility library for disassembling, decoding, and handling code statically or at runtime. We call a program that uses DynamoRIO in this manner a *standalone client*. The client must call our `dr_standalone_init()` routine, which returns a special machine context that can be passed to subsequent API routines that normally expect a context for a target application under DynamoRIO control (see Section 8.2).

8.1.2 Client Hooks

The core of a client's interaction with DynamoRIO occurs through *hooks* that the client exports. DynamoRIO calls these at appropriate times, giving the client access to key events during execution of the application.

We classify clients into two categories: those that are interested in all application code and those that are only interested in frequently executed code. For the former we provide a hook on basic block creation, while for the latter we provide a hook on trace creation. Through these hooks the

Client Routine	Description
`void dynamorio_init()`	Client initialization
`void dynamorio_exit()`	Client finalization
`void dynamorio_fork_init(void *context)`	Client re-initialization for the child of a fork
`void dynamorio_thread_init(void *context)`	Client per-thread initialization
`void dynamorio_thread_exit(void *context)`	Client per-thread finalization
`void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb)`	Client processing of basic block
`void dynamorio_trace(void *context, app_pc tag, InstrList *trace)`	Client processing of trace
`void dynamorio_fragment_deleted( void *context, app_pc tag)`	Notifies client when a fragment is deleted from the code cache
`int dynamorio_end_trace(void *context, app_pc trace_tag, app_pc next_tag)`	Asks client whether to end the current trace

Table 8.4: Client hooks called by DynamoRIO at appropriate times. The client is not expected to inspect or modify the `context` parameter, which is an opaque pointer to the current thread context. The `tag` parameters serve to uniquely identify fragments by their original application origin.

client has the ability to inspect and transform any piece of code that is emitted into the code cache. This interface gives full performance control to the client. Since fragment creation comprises such a small part of DynamoRIO's overhead (Section 7.2), typically only the code inserted into each fragment will impact overall execution efficiency, not the tool's analysis time. Our API also provides the ability to inspect and re-transform fragments once they are in the cache (Section 8.2.3).

Table 8.4 shows the full set of hooks that a client can export. The main fragment creation hooks are `dynamorio_basic_block` and `dynamorio_trace`. DynamoRIO calls `dynamorio_basic_block` each time a block is created, giving the client the opportunity to inspect and potentially modify every single application instruction before it executes. The basic block is passed as a pointer to an `InstrList` (our data structure for representing a sequence of instructions, as described in Section 4.1). Basic blocks (and traces) are identified by their starting application address, which we call the *tag*. The basic blocks passed to the client are slightly different from the original application code, since DynamoRIO elides unconditional control transfers (see Section 2.4). Indirect branch mangling has not yet occurred, however, and so the client need not be aware of its details.

DynamoRIO calls `dynamorio_trace` each time a trace is created, just before the trace is placed in the trace cache. Clients only interested in hot code can ignore basic blocks and focus on traces, already-identified frequently-executed code sequences. A trace is passed to the client as an `InstrList` that has already been completely processed, showing the client the exact code that will execute in the code cache (with the exception of the exit stubs). This means that DynamoRIO's indirect branch transformations have already been performed on the instruction stream that the client receives, so it may need to be aware of some of the details of how DynamoRIO inlines indirect branch comparisons in traces (see Figure 4.19).

In addition to the main fragment creation hooks, there are also hooks called by DynamoRIO for client initialization and termination: both process-wide and for each thread, to support thread-private client data. On Linux, DynamoRIO provides a routine for re-initialization for the child of a fork. Another hook, `dynamorio_fragment_deleted`, is called each time a fragment is deleted from the block or trace cache. Such information is needed if the client maintains its own data structures about emitted fragment code that must be kept consistent across fragment deletions. The final hook, `dynamorio_end_trace`, is used for custom trace creation (Section 8.2.4).

The `void *context` parameter that these hooks take in is the thread-local machine context that is used by DynamoRIO (see Section 5.2.2). The client is expected to treat it as an opaque pointer and never modify it, but simply pass it around for use when calling API routines.

8.2 Runtime Code Manipulation API

A client of DynamoRIO is provided with a powerful interface for custom runtime code transformations. While much of the interface focuses on instruction manipulation, there is also explicit support for transparent file and memory operations, state preservation, efficient single-fragment replacement, and even customization of trace building, exit stubs, and fragment prefixes. The following sections discuss each of these parts of our API. For a complete listing of the routines and data structures in the API, see our public release documentation [MIT and Hewlett-Packard 2002].

8.2.1 Instruction Manipulation

DynamoRIO exports a rich set of functions to manipulate IA-32 instructions, using the adaptive level-of-detail data structures discussed in Section 4.1. Decoding, encoding, and generating instructions from scratch are all supported at different levels of detail.

Instruction Generation

Instruction generation is simplified through a set of macros. A macro is provided for every IA-32 instruction. The macro takes as arguments only those operands that are explicit and automatically fills in the implicit operands (many IA-32 instructions have implicit operands), making instruction generation similar to using assembler syntax, where only explicit operands need be specified. As an example, consider the `push` instruction. It implicitly modifies the stack and the stack pointer, but all a user wants to specify is what is being pushed. An instruction that pushes the `eax` register can be generated with this call:

```
INSTR_CREATE_push(context, opnd_create_reg(REG_EAX))
```

Operands are represented by the `Opnd` data structure. The types of operands are listed in Table 8.5. Each IA-32 instruction template specifies a certain operand type for each of its operand slots. In addition to the type, an operand has a size, which is typically one, two, or four bytes, although for multimedia and other instructions it can be quite large.

A second method of generating an instruction bypasses the IA-32 instruction set abstraction level by specifying an opcode and complete list of operands. Both this method and the first method produce a Level 4 instruction. A third method of generating an instruction from scratch is to specify raw bytes rather than an opcode and operands, which will create a Level 1 instruction (or even a Level 0 sequence of instructions bound into one `Instr`).

Decoding

Decoding instructions from raw bytes is supported at multiple levels of detail. Table 8.6 lists the routines we export to build an instruction from raw bytes at each level. Level 4 is not listed since it is only entered when a decoded instruction is modified through its data structures, or when an instruction is generated from scratch. We do list *Level CTI*, which is the level we use to build basic

Operand Type	Notes
Null	Empty operand
Immediate Integer	
Immediate Float	Only used for certain implicit operands
Address	For control transfers targeting an absolute address
Instruction	For control transfers targeting another instruction
Register	
Base+Disp	Memory reference: base register + scaled index register + displacement
Far Address	An absolute address with a segment selector
Far Base+Disp	A memory reference with a segment register prefix

Table 8.5: Operand types supported by our instruction representation.

Level	Initial Decoding	Upgrade Routine
Level 0	`decode_next_pc`	N/A
Level 1	`decode_raw`	`instr_expand`
Level 2	`decode_opcode`	`instr_decode_opcode`
Level 3	`decode`	`instr_decode`
Level CTI	`decode_cti`	`instr_decode_cti`

Table 8.6: Decoding routines for each level of detail. The initial decoding routine is used to create an instruction at a specific level of detail from raw bytes. The upgrade routine is used to perform further decoding on an already-created instruction and raise its level of detail from a lower level.

blocks, where control transfer instructions (CTI) are at Level 3 while all other instructions are at Level 0 (as described in Section 4.1.1).

Once an instruction is built at a specific level, a certain amount of information has been determined about it. If further information is desired, the level of detail must be raised. If the instruction is at Level 1 or above, asking for yet-unknown details will auto-magically cause the instruction's level of detail to be raised. For example, asking for the opcode of a Level 1 instruction causes it to become a Level 2 instruction. Asking for one of its operands will raise it up to Level 3.

The Level 0 to Level 1 transition is special because it involves moving from one instruction to (potentially) a list of instructions, which requires an `InstrList`. For this reason, Level 0 instructions must be explicitly *expanded* to Level 1. We support several expansion routines. A single instruction can be expanded with the `instr_expand` function. We also provide iterators for walking an `InstrList` and expanding each instruction along the way:

- `instrlist_first_expanded`
- `instrlist_last_expanded`
- `instr_get_next_expanded`
- `instr_get_prev_expanded`

Finally, we export a routine that will expand and link up control transfer instructions with their targets (if those targets are within the instruction sequence), `instrlist_decode_cti`.

We also provide routines for printing out decoded instructions and operands, from the `Opnd`, `Instr`, and `InstrList` data structures as well as directly from raw bytes. These disassembly routines are invaluable when debugging runtime tools.

Modification

All fields of an `Instr` can be iterated over and modified: the prefixes, opcode, operands, raw bits, annotation (Section 8.2.2), and custom exit stub code (Section 8.2.5), if it is a fragment-exiting instruction. Changes in the prefixes, opcode, or operands will raise the instruction's level to Level 4, which will require a full encoding, while setting its raw bits will lower it to Level 1 (or even Level 0).

Encoding

Our encoding routines take an instruction or list of instructions at any level of detail and perform the most efficient encoding for that level. If the raw bytes are available and valid, they are simply copied. Only if full encoding must be performed is it done, by walking the IA-32 instruction templates for that opcode and finding the best match for its operands.

When encoding a control transfer instruction that targets another instruction, two encoding passes must be performed: one to find the offset of the target instruction, and the other to link the control transfer to the proper target offset.

8.2.2 General Code Transformations

This section describes our API support for building custom code transformations. The routines in this section are used with the instruction manipulation routines to provide a powerful interface for quickly building runtime code manipulation clients.

State Preservation

To facilitate code transformations, DynamoRIO makes available its register spill slots and other state preservation functionality. We export API routines for saving and restoring registers to and from thread-local spill slots. We also provide a generic thread-local storage field for use by clients, making it easy to write thread-aware clients.

Since saving and restoring the `eflags` is required for almost all code transformations, and since it is difficult to do so efficiently (see Section 4.4), we export routines that use our efficient method of arithmetic flag preservation.

We also export convenience routines for making *clean* (i.e., transparent) native calls from the code cache. These routines save the application state and switch to DynamoRIO's stack for full transparency. See Section 8.2.6 for more details.

Our API provides routines to save and restore the floating point and multimedia state, which must be done explicitly since our context switch (Section 3.3.4) and clean call mechanisms do not do so (because floating point and multimedia operations are rarely performed during code manipulation).

Annotations

We provide a special field in the `Instr` data structure that can be used by a client for annotations while it is processing instructions. The client can store and retrieve information from this field, which is treated as an opaque pointer by DynamoRIO.

Meta Instructions

DynamoRIO treats instructions inserted into a basic block by a client in the same way as original application instructions — e.g., control transfer instructions are transformed to maintain control.

However, there are cases where a client may wish to insert instructions that are not modified by DynamoRIO. Our interface supports this through *meta instructions*. Meta instructions are marked using the API routine `do_not_mangle`. Through meta instructions, a client can add its own internal control flow or make a call to a native routine that will not be brought into the code cache by DynamoRIO. However, such native calls need to be careful to remain transparent — see our clean call support in Section 8.2.6.

Processor Feature Identification

Our API contains routines that identify features of the underlying IA-32 processor, making it easy to perform architecture-specific optimizations (e.g., in Figure 9.2).

8.2.3 Inspecting and Modifying Existing Fragments

A client is not limited to only examining and transforming application code prior to its insertion in the code cache. We provide two key routines for inspecting and modifying existing fragments:

- `InstrList* dr_decode_fragment(void *context, app_pc tag);`
- `bool dr_replace_fragment(void *context, app_pc tag, InstrList *il);`

Clients may wish to re-examine, re-transform, or re-optimize code after it is placed in the code cache. To do this, clients must re-create the `InstrList` for a fragment from the cache, modify it, and then replace the old version with the new. For example, consider a client that inserts profiling code into selected traces. Once a threshold is reached, the profiling code calls `dr_decode_fragment` and then rewrites the trace by modifying the `InstrList`. Once finished, `dr_replace_fragment` is called to install the new version of the trace.

DynamoRIO is able to perform this replacement while execution is still inside the old fragment, allowing a fragment to generate a new version of itself. This is accomplished by delaying the removal of the old fragment until a safe point. All links targeting and originating from the old fragment are immediately modified to use the new fragment (this is another example of the usefulness of storing incoming links — see Section 2.2). This means that the current thread will continue to execute in the old fragment only until the next branch. Since there are no loops except in explicit links, the time spent in the old fragment is minimal, and all future executions use the new fragment.

This *efficient single-fragment replacement* is valuable. By using a new copy of the fragment to be replaced it avoids issues with self-modifying code. And by delaying the deletion, it can be used from a separate thread (see Section 8.2.7).

8.2.4 Custom Traces

Our API includes an interface for customizing DynamoRIO's trace building scheme. A client can direct the building of traces through a combination of the client hook `dynamorio_end_trace` and this API routine:

```
void dr_mark_trace_head(void *context, app_pc tag);
```

DynamoRIO's trace building mechanism centers around certain basic blocks that are considered *trace heads*. A counter associated with each trace head is incremented upon each execution of that basic block. Once the counter exceeds a threshold, DynamoRIO enters trace generation mode. Each subsequent basic block executed is added to the trace, until a termination point is reached. (For more information on trace building in DynamoRIO, see Section 2.3.)

By default, DynamoRIO only considers targets of backward branches and exits of existing traces to be trace heads. Our interface allows a client to choose its own trace heads, marking them with `dr_mark_trace_head`. When DynamoRIO is in trace generation mode, it calls the client's `dynamorio_end_trace` routine before adding a basic block to the current trace. The client can direct DynamoRIO to either end the trace, continue extending the trace, or use its default test (which stops upon reaching an existing trace or trace head) for whether to end the trace. For an example of using this interface, see Section 9.2.4.

8.2.5 Custom Exits and Entrances

Frequently, an optimization will make an assumption in order to optimize a sequence of code. If the assumption is violated, some clean-up action is required. To maintain the linearity of traces, DynamoRIO provides a mechanism for implementing this kind of clean-up code in the form of *custom exit stubs*. Each exit from a trace or basic block has its own stub (Section 2.1). When it is not linked to another fragment, control goes to the stub, which records where the trace was exited and then performs a context switch back to DynamoRIO. The client can specify a list of instructions to be prepended to the stub corresponding to any exit from a trace or basic block fragment, and can specify that the exit should go through the stub even when linked. The body of the fragment can then be optimized for the assumption, with conditional branches directing control flow to the custom stub if the assumption is violated. Without this direct support, a client would be forced to add branches targeting the middle of the trace, destroying the linear control flow that may be assumed by other optimizations.

The ability to customize exit stubs also enables a simple method for adaptive profiling, which is discussed in Section 7.3.3: instrumentation code is placed in the stub, where it can either be targeted or bypassed by changing the exit branch itself to either point to the stub or to its target fragment (if linked).

DynamoRIO uses *prefixes* on traces to restore the arithmetic flags and a scratch register, allowing our indirect branch lookup to jump to a trace without storing the target in memory in order to restore the final register value (Section 4.3). These prefixes are also useful to clients for the same types of control transfers to other fragments. Our API provides a routine that can be called during client initialization to request prefixes on all fragments, including basic blocks. Another routine allows a control transfer instruction exiting a fragment to be marked such that it will target the prefix rather than the main entry point of its target fragment.

8.2.6 Instrumentation Support

To facilitate building instrumentation clients, we provide routines for performing calls to native routines from the code cache. These routines ensure that such a native call is *clean*, i.e., transparent. They save the application state and switch the stack to DynamoRIO's stack for the current thread. Only then is the call invoked. Once it returns, the application state is restored. Figure 8.7 gives an example of a client that inserts a clean call to a profiling routine in each basic block that contains a system call. The profiling routine will then be called every time a system call is about to be executed by the application.

We provide higher-level convenience routines for inserting clean profiling calls for different types of control transfer instructions. A client simply supplies a routine that will be passed the

```
static void at_syscall(app_pc addr) {
    dr_printf("syscall executed in block 0x%08x\n", tag);
}
EXPORT void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb) {
    Instr *instr, *next_instr;
    for (instr = instrlist_first(bb); instr != NULL;
         instr = next_instr) {
         next_instr = instr_get_next(instr);
         if (!instr_opcode_valid(instr)) continue; /* avoid Level 0 */
         if (instr_is_syscall(instr)) {
             /* save app state */
             dr_prepare_for_call(context, bb, instr);
             /* push an argument */
             instrlist_meta_preinsert(bb, instr,
                 INSTR_CREATE_push_imm(context, OPND_CREATE_INT32(tag)));
             /* actual call */
             instrlist_meta_preinsert(bb, instr,
                 INSTR_CREATE_call(context, opnd_create_pc((app_pc)at_syscall)));
             /* clean up argument, restore app state */
             dr_cleanup_after_call(context, bb, instr, 4);
         }
    }
}
```

Figure 8.7: An example of how to use our convenience routines for inserting clean calls to a profile routine that will be called prior to each system call's execution.

target and direction of each branch in the application. This makes it very simple to build a client to analyze branches. A complete sample client using these branch instrumentation routines is shown in Figure 8.8.

8.2.7 Sideline Interface

We have a special version of our API that supports examining and modifying other threads' code, enabling the use of DynamoRIO to not only examine and manipulate code online, but also *sideline*, in a separate thread. The canonical application of sideline operation is dynamic optimization, where overheads of optimization can be reduced by performing optimizations in separate, lower-priority threads. Our fragment replacement interface (see Section 8.2.3) is perfect for this scheme, as a fragment can be replaced at any time.

Enabling transformations to be performed in a separate thread requires surprisingly few additions to our API. To make sideline fragment replacement possible, we must prevent the client thread and the application thread from both updating fragment data structures at the same time, using mutual exclusion. If the application thread remains in the code cache until after the replacement is complete, which is the common case, no contention cost is incurred. The sideline version of our API adds this synchronization to all routines that operate on fragments, as well as thread-local heap allocation, to enable the sideline thread to use the target thread's context for simplicity of de-allocation.

8.2.8 Thread Support

Our interface contains a number of features to facilitate client interaction with multiple threads. First are the already-mentioned client hooks for thread initialization and termination, which allow the client to be aware of new threads and to keep per-thread state. And, as mentioned in Section 8.2.2, we provide a thread-local storage slot for use by clients. Our API also exports routines for allocating thread-local memory using the same heap management as that used by DynamoRIO (which also aids in transparency — see Section 8.2.9). Finally, as a convenience, we export routines for using simple mutexes for synchronization.

8.2.9 Transparency Support

One of the most important and often overlooked requirements for a runtime tool is transparency (see Chapter 3). We designed our interface to make it easy for a client to remain transparent by exporting the methods and resources that DynamoRIO itself uses to isolate its actions.

For code transformations, our state preservation functions (Section 8.2.2) make it easy to preserve registers, condition codes, and other processor state. Our API provides routines for memory allocation, both global and thread-private, using DynamoRIO's transparent allocation. This frees the client from having to implement its own separate allocation scheme, and reminds the client that it should not use default allocators like `malloc`. We also export routines for reading and writing files, which use either the system call interface directly or library wrappers that do not perform buffering and do not have any re-entrancy or other transparency problems (see Section 3.1.1). These use the same routines that DynamoRIO uses for its own file manipulation (which it needs for debugging and other purposes).

Our transparency support makes it simple for a client to perform common actions without needing to worry about transparency. A client that instead uses the same buffers or memory allocation routines as the application has a good chance of adversely affecting program correctness. An additional benefit to clients of our transparency support is platform independence: the same code will work on both Linux and Windows, which is not true even for some C library routines (e.g., the `FILE` data type cannot be exported from a shared library on Windows).

8.3 Example Clients

This section gives three examples of application analysis clients built using the DynamoRIO interface: call profiling using our branch profiling interface, inserting counters directly into application code, and computing basic block size statistics. A further example, of dynamic optimization, is in Chapter 9's Figure 9.2. In addition to profiling applications by building custom clients, our built-in trace profiling, described in Section 7.3.3, can be used to identify and analyze frequently executed sequences of code in applications.

8.3.1 Call Profiling

The first example, whose source code is shown in Figure 8.8, uses our interface's control flow instrumentation routines (see Section 8.2.6) to gather profile data on application branches. The client uses a basic block hook to instrument every direct call, indirect call, return, conditional branch, and indirect jump that the application executes. The basic block hook inserts a call to a procedure for each type of instruction, using the interface-provided instrumentation routines. For illustration purposes, our example simply prints the source, target, and direction of each branch to a text file for post-processing.

This client is thread-aware, using a separate log file for each thread in the application. DynamoRIO's interface facilitates writing thread-aware clients by providing thread-local storage and per-thread initialization and cleanup routines. The client uses the provided file type and handling routines to avoid transparency problems (an additional benefit is platform independence: this same code will work on both Linux and Windows).

```
EXPORT void dynamorio_thread_init(void *context) {
    /* we're going to dump our data to a per-thread file */
    File f = dr_open_log_file("instrcalls");
    assert(f != INVALID_File);
    /* store it in the slot provided in the context */
    dr_set_drcontext_field(context, (void *)f);
}
EXPORT void dynamorio_thread_exit(void *context) {
    File f = (File) dr_get_drcontext_field(context);
    dr_close_file(f);
}
static void at_call(app_pc instr_addr, app_pc target_addr) {
    File f = (File) dr_get_drcontext_field(dr_get_current_drcontext());
    dr_fprintf(f, "CALL @ 0x%08x to 0x%08x\n", instr_addr, target_addr);
}
static void at_call_ind(app_pc instr_addr, app_pc target_addr) {
    File f = (File) dr_get_drcontext_field(dr_get_current_drcontext());
    dr_fprintf(f, "CALL INDIRECT @ 0x%08x to 0x%08x\n", instr_addr, target_addr);
}
static void at_return(app_pc instr_addr, app_pc target_addr) {
    File f = (File) dr_get_drcontext_field(dr_get_current_drcontext());
    dr_fprintf(f, "RETURN @ 0x%08x to 0x%08x\n", instr_addr, target_addr);
}
static void at_conditional(app_pc instr_addr, app_pc target_addr, bool taken) {
    File f = (File) dr_get_drcontext_field(dr_get_current_drcontext());
    dr_fprintf(f, "CONDITIONAL @ 0x%08x to 0x%08x %staken\n",
        instr_addr, target_addr, taken?"":"NOT ");
}
static void at_jump(app_pc instr_addr, app_pc target_addr) {
    File f = (File) dr_get_drcontext_field(dr_get_current_drcontext());
    dr_fprintf(f, "JUMP INDIRECT @ 0x%08x to 0x%08x\n", instr_addr, target_addr);
}
EXPORT void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb) {
    Instr *instr, *next_instr;
    /* only interested in calls & returns, so can use DynamoRIO instrlist
     * as is, do not need to expand it! */
    for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
        next_instr = instr_get_next(instr);
        /* we can rely on all ctis being decoded, so skip un-decoded instrs */
        if (!instr_opcode_valid(instr))
            continue;
        /* instrument calls and returns -- ignore far calls/rets */
        if (instr_is_call_direct(instr)) {
            dr_insert_call_instrumentation(context, bb, instr, (app_pc)at_call);
        } else if (instr_is_call_indirect(instr)) {
            dr_insert_mbr_instrumentation(context, bb, instr, (app_pc)at_call_ind);
        } else if (instr_is_return(instr)) {
            dr_insert_mbr_instrumentation(context, bb, instr, (app_pc)at_return);
        } else if (instr_is_cbr(instr)) {
            dr_insert_cbr_instrumentation(context, bb, instr, (app_pc)at_conditional);
        } else if (instr_is_mbr(instr)) {
            dr_insert_mbr_instrumentation(context, bb, instr, (app_pc)at_jump);
        }
    }
}
```

Figure 8.8: Code for a client that examines each basic block and collects statistics on all types of branch instructions.

8.3.2 Inserting Counters

Our second example shows how to insert counters directly into application code. It counts the number of direct calls, indirect calls, and returns. This could be accomplished by again using our branch profiling convenience routines, but it is much more efficient to perform the increment inline in the application code, rather than suffering the overhead of clean calls to routines that do nothing but a single increment.

8.3.3 Basic Block Size Statistics

Our final example computes size statistics for basic blocks. Since it uses floating point operations, which are not saved or restored by DynamoRIO context switches, it must explicitly save and restore the floating point state (see Section 8.2.2).

8.4 Client Limitations

We would like to allow a client to modify instruction streams in any way it chooses, but unfortunately our interface has some limitations. This section discusses both fundamental limitations of clients for any runtime system and specific limitations of our implementation.

A fundamental restriction on clients is that they remain transparent If a client violates transparency, there is little that Dynamorio can do about it (though it may be the desired effect of some clients). Application correctness may fail with no chance for recovery.

A client that seriously changes basic block control flow can disrupt DynamoRIO's trace creation. We do not disallow this, but we caution clients to perform dramatic changes to basic block control flow at their own risk.

Another limitation of our implementation is that it stores information on arithmetic flag behavior of a fragment prior to calling the client hook for transforming it, for both basic blocks and traces. If a client modifies the flags behavior of the fragment, DynamoRIO might do the wrong thing in the fragment's flag restoration prefix. We could solve this by re-building the flag information, but would want some kind of notification by the client that it changed the flag behavior to avoid a performance hit.

```
/* keep separate counters for each thread, in this thread-local data structure: */
typedef struct {
    int OF_slot; /* used for saving overflow flag */
    int num_direct_calls;
    int num_indirect_calls;
    int num_returns;
} per_thread;

EXPORT void dynamorio_init() {
    /* initialize our global variables */
    num_direct_calls = 0;
    num_indirect_calls = 0;
    num_returns = 0;
    dr_mutex_init(&mutex);
}

EXPORT void dynamorio_exit() {
    dr_printf("Instrumentation results:\n");
    dr_printf("\tsaw %d direct calls\n", num_direct_calls);
    dr_printf("\tsaw %d indirect calls\n", num_indirect_calls);
    dr_printf("\tsaw %d returns\n", num_returns);
}

EXPORT void dynamorio_thread_init(void *context) {
    /* create an instance of our data structure for this thread */
    per_thread *data = (per_thread *)
        dr_thread_alloc(context, sizeof(per_thread));
    /* store it in the slot provided in the context */
    dr_set_drcontext_field(context, data);
    data->num_direct_calls = 0;
    data->num_indirect_calls = 0;
    data->num_returns = 0;
    dr_log(context, LOG_ALL, 1, "countcalls: set up for thread %d\n",
           dr_get_thread_id(context));
}

EXPORT void dynamorio_thread_exit(void *context) {
    per_thread *data = (per_thread *) dr_get_drcontext_field(context);
    /* add thread's counters to global ones, inside our lock */
    dr_mutex_lock(&mutex);
    num_direct_calls += data->num_direct_calls;
    num_indirect_calls += data->num_indirect_calls;
    num_returns += data->num_returns;
    dr_mutex_unlock(&mutex);
    /* clean up memory */
    dr_thread_free(context, data, sizeof(per_thread));
}
```

Figure 8.9: Part 1 of code for a client that inserts counters of calls and returns directly into application code. The meat of the code is in the basic block hook, shown in Figure 8.10.

```
EXPORT void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb) {
    Instr *instr, *next_instr;
    per_thread *data = (per_thread *) dr_get_drcontext_field(context);
    /* only interested in calls & returns, so can use DynamoRIO instrlist
     * as is, do not need to expand it! */
    for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
        /* grab next now so we don't go over instructions we insert */
        next_instr = instr_get_next(instr);
        /* we can rely on all ctis being decoded, so skip un-decoded instrs */
        if (!instr_opcode_valid(instr))
            continue;
        /* instrument calls and returns -- ignore far calls/rets */
        if (instr_is_call_direct(instr)) {
            /* since the inc instruction clobbers 5 of the arith flags,
             * we have to save them around the inc.
             * we could be more efficient by not bothering to save the
             * overflow flag and constructing our own sequence of instructions
             * to save the other 5 flags (using lahf). */
            dr_save_arith_flags(context, bb, instr, &(data->OF_slot));
            instrlist_preinsert(bb, instr, INSTR_CREATE_inc(context,
                OPND_CREATE_MEM32(REG_NULL, (int)&(data->num_direct_calls))));
            dr_restore_arith_flags(context, bb, instr, &(data->OF_slot));
        } else if (instr_is_call_indirect(instr)) {
            dr_save_arith_flags(context, bb, instr, &(data->OF_slot));
            instrlist_preinsert(bb, instr, INSTR_CREATE_inc(context,
                OPND_CREATE_MEM32(REG_NULL, (int)&(data->num_indirect_calls))));
            dr_restore_arith_flags(context, bb, instr, &(data->OF_slot));
        } else if (instr_is_return(instr)) {
            dr_save_arith_flags(context, bb, instr, &(data->OF_slot));
            instrlist_preinsert(bb, instr, INSTR_CREATE_inc(context,
                OPND_CREATE_MEM32(REG_NULL, (int)&(data->num_returns))));
            dr_restore_arith_flags(context, bb, instr, &(data->OF_slot));
        }
    }
}
```

Figure 8.10: Part 2 of code for a client that inserts counters of calls and returns directly into application code. This is the basic block hook that inserts the actual counter increments. The rest of the code is shown in Figure 8.9.

One other limitation is in translating the machine context for an exception or signal handler (see Section 3.3.4) in the presence of arbitrary client code transformations. Future work could involve providing an interface asking the client how to translate code it has manipulated to obtain an original context to show the application's handler.

8.5 Chapter Summary

This chapter presented our interface for building custom runtime code manipulation tools. Key components in our interface include our adaptive level-of-detail instruction representation from

```
EXPORT void dynamorio_init() {
    num_bb = 0;
    ave_size = 0.;
    max_size = 0;
    dr_mutex_init(&stats_mutex);
}

EXPORT void dynamorio_exit() {
    dr_printf("Number of basic blocks seen: %d\n", num_bb);
    dr_printf("               Maximum size: %d instructions\n", max_size);
    dr_printf("               Average size: %5.1f instructions\n", ave_size);
}

EXPORT void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb) {
    Instr *instr;
    int cur_size = 0;
    /* we use fp ops so we have to save fp state */
    byte fp_raw[512 + 16];
    byte *fp_align = (byte *) ( (((uint)fp_raw) + 16) & 0xfffffff0 );
    proc_save_fpstate(fp_align);
    for (instr = instrlist_first_expanded(context, bb);
         instr != NULL;
         instr = instr_get_next_expanded(context, bb, instr))
        cur_size++;
    dr_mutex_lock(&stats_mutex);
    if (cur_size > max_size)
        max_size = cur_size;
    ave_size = ((ave_size * num_bb) + cur_size) / (double) (num_bb+1);
    num_bb++;
    dr_mutex_unlock(&stats_mutex);
    proc_restore_fpstate(fp_align);
}
```

Figure 8.11: Code for a client that examines each basic block and computes overall statistics on basic block sizes.

Section 4.1 and explicit support for client transparency and fine-grained adaptive optimization through replacement of existing fragments. We carefully designed the interface to be at the proper abstraction level: general enough that arbitrary tools can be built, but narrow enough to facilitate new tool creation. Simply providing source code is not the right interface — a well-thought-out API is essential to an extensible system. In fact, within the DynamoRIO project there were a number of internal instrumentation tasks that we might have accomplished by modifying DynamoRIO code itself, but found it more attractive to build a separate client. This modular separation, isolating the new instrumentation code from the core DynamoRIO code, allows a client to be developed and tested against a single, stable version of the core system, instead of every change to the client resulting in a new DynamoRIO.

In the same way that compiler infrastructures [Wilson et al. 1994, Trimaran] helped advance compiler research, we hope that our runtime code manipulation infrastructure and interface will prevent the need for others to expend the huge effort required to build a system that runs large, complex applications.

Chapter 9

Application Case Studies

This chapter presents several example uses of runtime code manipulation technology for such diverse purposes as instrumentation (Section 9.1), dynamic optimization (Section 9.2 and Section 9.3), and security (Section 9.4).

9.1 Instrumentation of Adobe Premiere

To illustrate the characteristics of large, commercial applications, the robustness of DynamoRIO, and the facility with which complex programs can be studied using DynamoRIO's interface, this section presents a case study of using a DynamoRIO client to count instructions in Adobe Premiere. We could have chosen any number of large programs, and only picked Premiere for its use of self-modifying code, to contrast it with the dynamically generated code in the desktop applications in our benchmark suite. Our Premiere workload is nothing more than starting the application, loading three of the sample movies that come with the program, and playing one of them several times. Even this simple scenario executes a tremendous amount of code and requires every component of DynamoRIO described in this thesis.

Adobe Premiere created 87 threads over the course of execution, though the peak number of simultaneously live threads was 25. Before the program was closed, 44,388 callbacks, 86 APCs, and 25 exceptions were received. Over 400,000 basic block fragments and over 13,000 traces were created from 1.7MB of application code, almost all of it in the primary thread.

We built a simple client to count the number of instructions executed, the number of floating-point instructions executed, and the number of system calls invoked. In our run of Premiere nearly

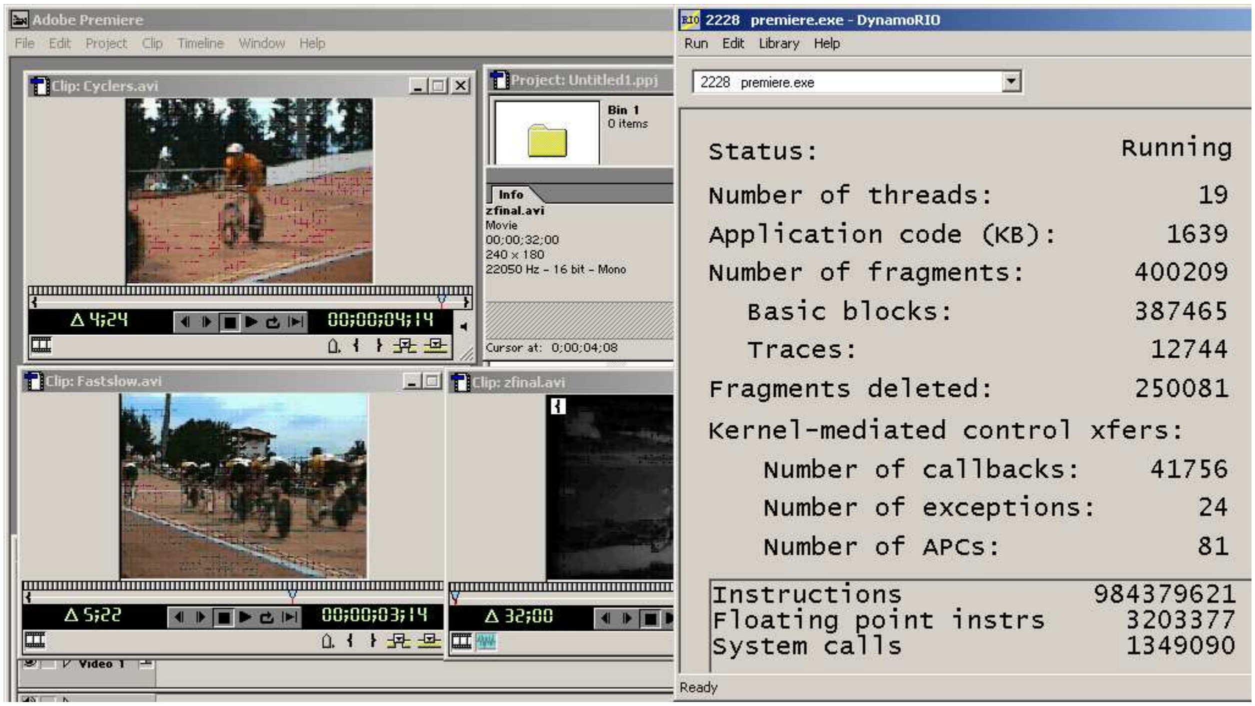

Figure 9.1: A screen snapshot of a DynamoRIO client measuring the number of different types of instructions executed by Adobe Premiere. The visual display shows both DynamoRIO's own statistics and the client's custom measurements.

one billion instructions were executed, including 3.2 million floating-point instructions and 1.3 million system calls. Figure 9.1 shows a screen snapshot of Premiere and a visual display of both DynamoRIO's internal statistics and the client's custom statistics. Without the client, Premiere's initialization period is nearly twice as slow under DynamoRIO as natively, but performance improves later as more code is re-used. Interactive use masks DynamoRIO's overhead, causing Premiere to feel close to native speed. Using the client noticeably impacts performance, which is not surprising given that condition-code-transparent counter increments are being inserted into every basic block to count instructions.

During execution, DynamoRIO's cache consistency algorithm (Section 6.2) marked 8 code regions that were writable as read-only. These regions were written to 9 times, each one trapped and handled. These writes combined with libraries being unloaded accounted for 1363 cache consistency flushes, although most of them (1027) were due to false sharing and resulted in no fragments actually being flushed. However, Premiere does dynamically generate and modify code, all of which was treated as self-modifying by our algorithm. DynamoRIO marked 8 different pages as requiring sandboxing, since an instruction on each page wrote to a target on the same page. Sandboxing instrumentation was added to 900 fragments, which were executed 949,972 times. The instrumentation detected modification of a fragment during its own execution 371 times. Premiere's self-modifying code follows typical patterns of modifying instructions' immediate fields, accepting a performance hit for the modification in exchange for improved speed in future executions (versus the alternative of using indirection for the immediate).

Premiere's total virtual size was 390MB and committed memory was 243MB, to which DynamoRIO added 48MB of address space and 44MB of committed memory when configured with unlimited code cache sizes. Using our adaptive working set algorithm (Section 6.3.3), with parameters set at the default 10 regenerated per 50 replaced, reduced DynamoRIO's memory usage by 30% in both virtual size and committed pages.

Adobe Premiere as a prototypical large, complex, multi-threaded application that illustrates the dynamic capabilities required to study modern applications. Premiere imports routines from 21 libraries, which are loaded at the beginning of execution. By the end of our workload there were 137 libraries in the address space, due to the many plugins and other modules that supply the bulk of Premiere's functionality. A static tool would have no access to any of these libraries. DynamoRIO enables tools to target these types of programs with no extra effort from the tool builder. Tools are not limited to only observing the runtime code stream — modification of any instruction is possible using DynamoRIO, as the next section shows in the form of dynamic optimization.

9.2 Dynamic Optimization

In addition to extending the reach of optimization to modern, dynamic behavior, optimizing a program at runtime allows the user to improve the performance of binaries without relying on how they were compiled, as many software vendors are hesitant to ship binaries that are compiled with high levels of static optimization because they are hard to debug. Furthermore, several types of optimization are best suited to a dynamic optimization framework. These include adaptive, architecture-specific, and inter-module optimizations.

Adaptive optimizations require instant responses to changes in program behavior. When performed statically, a single profiling run is taken to be representative of the program's behavior, while at runtime, ongoing profiling identifies which code is currently hot, allowing optimizations to focus only where they will be most effective. Architecture-specific code transformations may be done statically if the resulting executable is only targeting a single processor, unduly restricting the executable, or by using dynamic dispatch to select among several transformations prepared for different processors, bloating the executable size. Performing the optimization dynamically allows the executable to remain generic and specialize itself to the processor on which it happens to be running. Inter-module optimizations cannot be done statically in the presence of shared and dynamically-loaded libraries, but runtime optimizations have a view of the code that cuts across the static units used by the compiler for optimization.

Dynamic optimizations have a significant disadvantage versus static optimizations. The overhead of performing the optimization must be amortized before any improvement is seen. This limits the scope of optimizations that can be done online, and makes the efficiency of the optimization infrastructure critical. For this reason, while there are numerous flexible and general compiler infrastructures for developing static optimizations [Wilson et al. 1994, Trimaran], there are very few for the development of dynamic optimizations.

Another disadvantage of optimizing a binary with no source code is that information that can facilitate optimization is missing. Instead of understanding the program at the programming language level, it must be addressed at the much broader instruction set architecture level. On CISC architectures in particular, operating on binaries is challenging due to memory aliasing and the lack of registers (see Section 4.1).

Another important contrast with static compilation is transparency. Unlike a static compiler optimization, a dynamic optimization cannot use the same memory allocation routines or input/output buffering as the application, because the optimization's operations are interleaved with those of the application (see Chapter 3).

In the following sections we present four sample optimizations [Bruening et al. 2003] implemented with the DynamoRIO client interface (Chapter 8). Section 9.2.5 then shows the performance impact of our sample optimizations.

9.2.1 Redundant Load Removal

First, we implemented a traditional static compiler optimization, redundant load removal, dynamically. Because there are so few registers in IA-32, local variables are frequently loaded from and stored back to the stack. If a variable's value is already in a register, a subsequent load can be removed. The compiler should be able to eliminate redundant loads within basic blocks, but we found that `gcc` at its highest optimization level still emits a number of redundant loads within blocks. It also produces redundant loads across basic block boundaries, which are more difficult for the compiler to identify. This optimization shows that even code compiled at high optimization levels stands to benefit from dynamic application of traditional optimizations. Unfortunately this is an aggressive optimization that could be incorrect when applied to thread-shared, volatile memory locations, and there is no simple way to check for those at the binary level (there is no analogous information to the `volatile` programming language annotation that compilers rely on).

9.2.2 Strength Reduction

On the Pentium 4 the `inc` instruction is slower than `add 1` (and `dec` is slower than `sub 1`). The opposite is true on the Pentium 3, however. A DynamoRIO client can perform this strength-reduction optimization by simply scanning each basic block for `inc` instructions, as shown in Figure 9.2. Analysis is required to determine if the condition codes differences between `inc` and `add` [Intel Corporation 2001, vol. 2] are acceptable for this block. If so, the `inc` is replaced by `add 1`.

This is a perfect example of an architecture-specific optimization that is best performed dynamically, tailoring the program to the underlying processor. It would be awkward to perform at load time, as a loader would have to rewrite all code in all shared libraries, regardless of how little of that code is actually run, and would need to specially handle libraries loaded later or dynamically-generated code. Furthermore, the variable-length IA-32 instruction set makes it difficult to analyze binaries prior to execution, because the internal module boundaries are not known and identifying all of the code is challenging.

```
EXPORT void dynamorio_init() {
    enable = (proc_get_family() == FAMILY_PENTIUM_IV);
    num_examined = 0;
    num_converted = 0;
}
EXPORT void dynamorio_exit() {
    if (enable)
        dr_printf("converted %d out of %d\n",
            num_converted, num_examined);
    else dr_printf("kept original inc/dec\n");
}
EXPORT void dynamorio_trace(void *context, app_pc tag, InstrList *trace) {
    Instr *instr, *next_instr;
    int opcode;
    if (!enable) return;
    for (instr = instrlist_first_expanded(context, trace, bb);
         instr != NULL; instr = next_instr) {
        next_instr = instr_get_next_expanded(context, trace, instr);
        opcode = instr_get_opcode(instr);
        if (opcode == OP_inc || opcode == OP_dec ) {
            num_examined++;
            if (inc2add(context, instr, trace))
                num_converted++;
        }
    }
}
static bool inc2add(void *context, Instr *instr, InstrList *trace) {
    Instr *in;
    uint eflags;
    int opcode = instr_get_opcode(instr);
    bool ok_to_replace = false;
    /* add writes CF, inc does not, check ok! */
    for (in=instr; in != NULL; in=instr_get_next_expanded(context, trace, in)) {
        eflags = instr_get_eflags(in);
        if ((eflags & EFLAGS_READ_CF) != 0) return false;
        /* if writes but doesn't read, we can replace */
        if ((eflags & EFLAGS_WRITE_CF) != 0) {
            ok_to_replace = true;
            break;
        }
        /* simplification: stop at first exit */
        if (instr_is_exit_cti(in)) return false;
    }
    if (!ok_to_replace) return false;
    if (opcode == OP_inc )
        in = INSTR_CREATE_add(context,
            instr_get_dst(instr,0),OPND_CREATE_INT8(1));
    else
        in = INSTR_CREATE_sub(context,
          instr_get_dst(instr,0),OPND_CREATE_INT8(1));
    instr_set_prefixes(in, instr_get_prefixes(instr));
    instrlist_replace(trace, instr, in);
    instr_destroy(context, instr);
    return true;
}
```

Figure 9.2: Code for a client implementing an `inc` to `add 1` strength reduction optimization. Since `inc` and `add` have different condition code semantics [Intel Corporation 2001, vol. 2], analysis must be done to determine if the change is acceptable for this block.

```
call prof_routine
jmp hashtable_lookup
```

⇓

```
cmp real_target, hot_target_1
je hot_target_1
cmp real_target, hot_target_2
je hot_target_2
call prof_routine
jmp hashtable_lookup
```

Figure 9.3: Code transformation by our indirect branch dispatch optimization. A profiling routine rewrites its own trace to insert dispatches for the hottest targets among its samples, avoiding a hashtable lookup.

9.2.3 Indirect Branch Dispatch

As an example of adaptive optimization, we perform value profiling of indirect branch targets. DynamoRIO, like Embra [Witchel and Rosenblum 1996] and Dynamo [Bala et al. 2000], inlines one target of an indirect branch when it builds a trace across the branch (see Section 2.3). However, whenever the indirect branch has a target other than the inlined target, a hashtable lookup is required. This lookup is the single greatest source of overhead in DynamoRIO (see Sections 4.2 and 4.3). To mitigate the overhead, a series of compares and conditional direct branches for each frequent target are inserted prior to the hashtable lookup. This is similar to the "inline caching" of virtual call targets in Smalltalk [Deutsch and Schiffman 1984] and Self [Hölzle 1994], but applied to returns and indirect jumps as well as indirect calls.

The optimization works as follows: when an indirect branch inlined in a trace has a target different from that recorded when the trace was created, it usually transfers control to the hashtable lookup routine. The optimization diverts that control transfer to a code sequence at the bottom of the trace. This code sequence consists of a series of compare-plus-conditional-branch pairs followed by a call to a profiling routine (Figure 9.3). After the call is a jump to the hashtable lookup routine. Initially there are no compare-branch pairs and control immediately goes to the profiling routine, which records the target of the indirect branch each time it is called. Once a threshold is reached in the number of samples collected, the profiling routine rewrites the trace (using the interface presented in Section 8.2.3) to add compare-branch pairs for the hottest targets.

The profiling call is kept in the trace but is only reached if none of the hot targets are matched, adaptively replacing the hashtable lookup with a series of compares and direct branches.

No profiling is done to determine if the inserted targets remain hot; once a target is inserted, it is never removed. Improving this is an area of future work, requiring the development of always-on, low-overhead profiling techniques.

9.2.4 Inlining Calls with Custom Traces

As an example of our custom trace interface (Section 8.2.4), we built a client that attempts to inline entire procedure calls into traces. The standard DynamoRIO traces focus on loops and often end up with a hot procedure call's return in a different trace from the call. This causes many hashtable lookups as the call is invoked from different call sites and the inlined return target keeps missing. Our custom traces mark calls as trace heads and returns as trace termination points (see Section 2.3). A trace will be terminated if a maximum size is reached, to prevent too much unrolling of loops inside calls. Once a return is reached, the trace is ended after the next basic block. This inlines the return and nearly guarantees that the inlined target will match. Our implementation is aggressive, assuming that the calling convention holds and that the return can be removed entirely.

9.2.5 Experimental Results

Figure 9.4 shows the performance results of the optimizations from the previous sections, relative to base DynamoRIO performance. The first bar in Figure 9.4 gives the performance for our redundant load removal optimization, which achieves a forty percent speedup for `mgrid` and also does well on a number of other floating-point benchmarks. Its effects on the integer benchmarks are less dramatic. The second bar shows the results for the `inc` to `add 1` transformation, which is able to speed up a number of benchmarks, including over a ten percent improvement on `sixtrack`. The adaptive indirect branch target optimization does well on several of the integer benchmarks, especially `vortex` and `gap`. The fourth bar shows the result of our custom traces, which speed up `gap`, `crafty`, and `twolf`. Finally, the figure shows the performance of running all four of our sample optimizations at once. The resulting mean execution time for the floating-point benchmarks is a

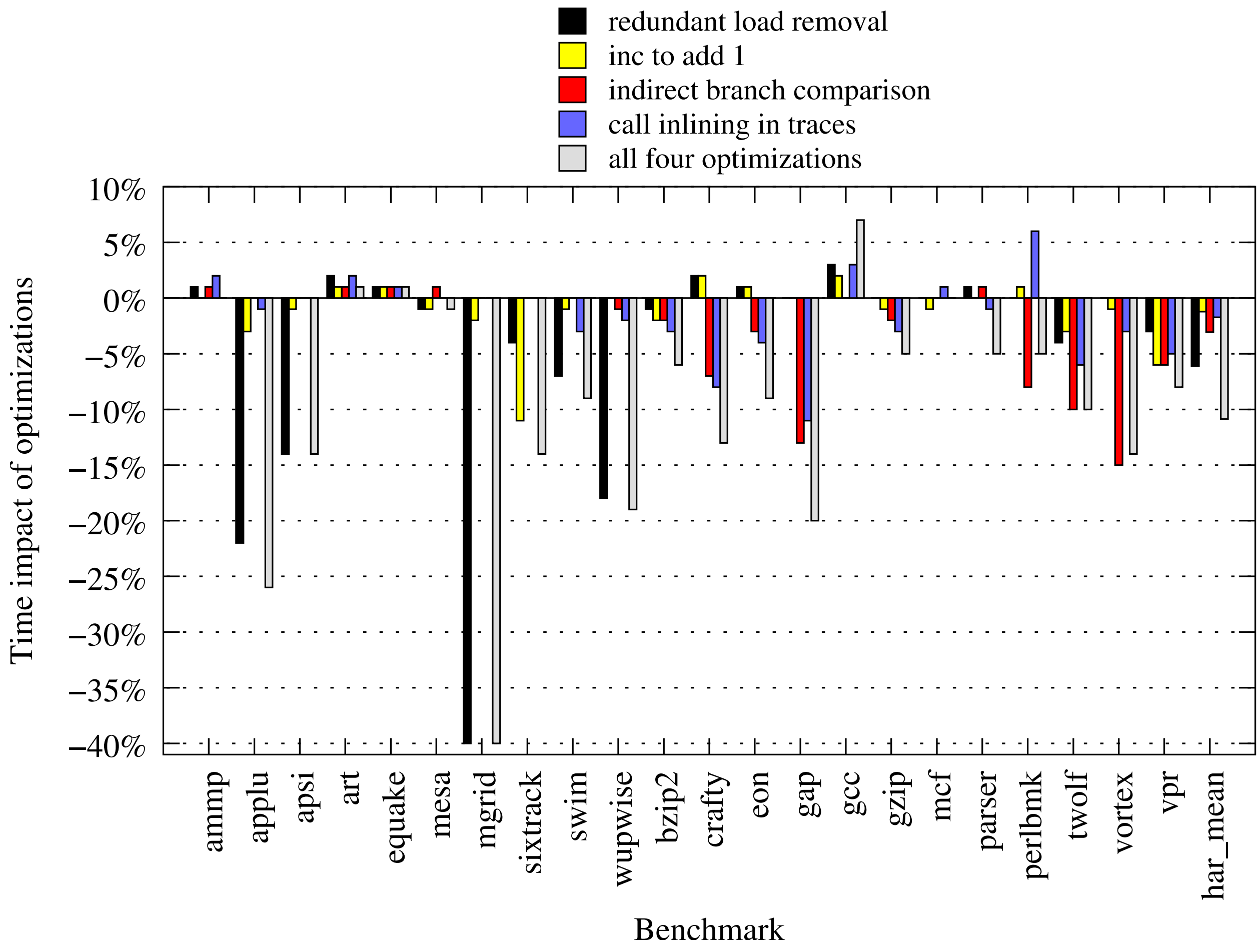

Figure 9.4: The performance impact of our four sample dynamic optimizations, and of applying them all in combination.

12% improvement over native.

Our optimizations result in slight slowdowns relative to base DynamoRIO performance on a few benchmarks. The largest slowdowns are on `perlbmk` and `gcc`, which consist of multiple short runs with little code re-use. The time spent performing the optimizations outweighs any benefits for these benchmarks, thwarting overhead amortization. Section 11.2.4 discusses future work in improving dynamic optimization on the IA-32 architecture.

9.3 Interpreter Optimization

As the previous section showed, low-level dynamic optimizations can be successful. We now shift our attention to an area where higher-level information is needed: removing interpreter overhead from programming language implementations. This section describes a method for raising DynamoRIO's traces from the low-level native code of an interpreter up to the level of the interpreted application, enabling optimizations such as dynamic partial evaluation.

For domain-specific languages, scripting languages, dynamic languages, and virtual machine-based languages, the most straightforward implementation strategy is to write an interpreter. A simple interpreter consists of a loop that fetches the next high-level instruction, dispatches to the routine handling that instruction, and then repeats. This simple mechanism can be improved in many ways, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some interpretive overhead.

There is a long history of approaches to removing interpretive overhead from programming language implementations. Threaded interpreters [Moore and Leach 1970] are one step toward removing the dispatch cost in the interpreter loop. Piumarta and Riccardi [1998] take threading further with dynamically-generated bytecode sequences. Often, once an interpreted language becomes popular, pressure builds to improve performance until eventually a native *just-in-time* (JIT) compiler is developed for the language. However, implementing a JIT is a challenging effort, affecting a significant part of the existing language implementation and adding a significant amount of code and complexity to the overall code base.

9.3.1 The Logical Program Counter

Our approach is a combination of native JIT compiler and *partial evaluation* [Jones et al. 1993, Jones 1996] techniques, using minimal interpreter instrumentation to achieve significant performance improvements. Just like a JIT, our goal is to remove the overhead of dispatching on high-level instructions and to compile them down into native instruction sequences. We start with the base DynamoRIO system, which builds traces representing frequently executed sequences of application code. Ideally, appropriate optimizations would specialize each trace for its component high-level instructions. Unfortunately, DynamoRIO's trace building, which targets hot loops in the application instruction stream (Section 2.3), is too low-level. For an interpreter dispatch loop, a single trace head will be chosen, for the top of the loop. This is a poor choice, as the body of the loop follows a different path for each high-level instruction type being dispatched upon. Threaded interpreters pose a related problem for DynamoRIO. The pervasive use of indirect jumps for control transfer foils DynamoRIO's trace head identification heuristics, and, even if a trace head were identified, the address of a particular high-level instruction handling routine does not uniquely identify a commonly occurring sequence of native instructions.

Introducing a new notion of a *logical program counter* can raise the abstraction level of trace building from the native instruction stream of the interpreter to the high-level instruction stream of the interpreted application. Instead of only the native program counter (PC) driving trace building, what is needed is to combine the native PC with a logical PC to uniquely identify the current overall computation point. By *logical PC* we mean some unique identifier into the control flow structure of the interpreted application (such as an interpreted function plus the offset into its bytecode stream). Neither the logical nor native PC alone is sufficient. For example, the native PC corresponding to the start of the handler for a `CALL` bytecode would be executed for each call site encountered in an interpreted application. The logical PC, on the other hand, might stay constant over a significant amount of interpreter execution (consider the interpretation of the `invokevirtual` Java virtual machine [Lindholm and Yellin 1999] instruction). We call the <*logical PC*, *native PC*> pair the *abstract PC*.

9.3.2 Instrumenting the Interpreter

Though it may be possible to derive the logical PC automatically in some cases, our prototype relies on annotations inserted into the interpreter by the interpreter writer. The interpreter writer must supply two types of information to DynamoRIO: the identification of logical control flow actions and the location of the immutable program representation used to drive interpretation.

Every time that either the native PC changes (by sequential execution or a branch or jump) or the logical PC changes (by a change in the controlling state of the interpreter), the abstract PC has changed. DynamoRIO provides the interpreter writer with a simple API for identifying relevant changes in the control state of the interpreter. Calls to these API functions enable DynamoRIO to identify *logical trace heads*, build *logical traces*, and link them together, in an analogous manner to

Instrumentation Routine and Description
`logical_direct_jump(new_logpc)` When called at a particular abstract pc <native PC, logical PC>, promises that the interpreter will always make the logical control transfer to <native PC + 1, `new_logpc`>).
`logical_indirect_jump(new_logpc)` The actual target varies based on runtime data (e.g., a value on the stack for a `RETURN` bytecode) and no promises can be made.
`logical_relative_jump(offset)` Corresponds to sequential native execution, promising that the current <native PC, logical PC> will always advance to <native PC + 1, logical PC + `offset`>.
`set_region_immutable(start, end)` Marks a region of memory as immutable for the duration of execution.
`add_trace_constant_address(addr)` Identifies an address whose value (i.e., when de-referenced) is guaranteed to be the same whenever control reaches the abstract PC of the call, allowing DynamoRIO to fold de-references into constants.
`set_trace_constant_stack_address(addr, val)` Also identifies a constant location, but on the stack, with the current value provided so that DynamoRIO can note its stack offset. DynamoRIO will only fold de-references of `addr` to a constant when control is within the stack frame of this API call.

Table 9.5: API routines inserted into an interpreter to communicate with DynamoRIO changes in the logical PC and information about immutable data. (See Sullivan et al. [2003] for further details.)

its native trace building. Each API function (see Table 9.5) corresponds to a type of native control transfer. Each function identifies an abstract PC, with the interpreter writer providing the logical PC and DynamoRIO the native PC.

In order for logical traces to be optimized via partial evaluation, immutable program data must be labeled. The interpreter writer must identify regions of memory that hold immutable representations of the application, as well as identifying other memory locations that will be constant for given <logical PC, abstract PC> pairs. We use three API functions for providing this information, also shown in Table 9.5.

9.3.3 Logical Trace Optimization

Using immutable program data information supplied by the above annotations, we apply dynamic partial evaluation to our logical traces. The key insight of our partial evaluation is that if we know the logical PC value at the start of a logical trace, then:

1. De-references from the immutable high-level instruction representation, indexed by the logical PC, can be statically folded to constants;
2. Conditional branches based on now-constant values can be removed entirely; and
3. Direct increments to the logical PC can be identified and tracked, thus enabling continued partial evaluation.

We apply three key optimizations to our logical traces: constant propagation and folding, call-return matching to eliminate expensive indirect branch overhead, and dead code elimination enabled by constant folding.

We have applied our annotations to two real-world interpreters: OCaml [Leroy 2003] and Kaffe [Kaffe.org]. OCaml is a well-implemented threaded interpreter for a variant of the ML language. Kaffe is a very slow implementation, using recursive tree walking, of Java; Kaffe's interpreter can afford to be extremely slow, because the implementation also includes a reasonable JIT compiler. For Kaffe, we generalized the logical PC API to allow for multiple logical PC values and to allow those values to be on the stack (to handle recursive interpretation). We evaluated our approach on three Kaffe micro-benchmarks and eight OCaml micro-benchmarks. The base DynamoRIO system has a slight slowdown on each benchmark. Using logical traces initially makes things worse, due to the code expansion, but the enabled trace optimizations are able to speed up all of the benchmarks beyond native performance, as Figure 9.6 shows. (See Baron [2003] for a breakdown of the contribution of each optimization.) While our results do not match the full performance improvements of hand-crafted native compilers, our system provides an appealing point on the language implementation spectrum, requiring minimal effort on the part of the interpreter writer.

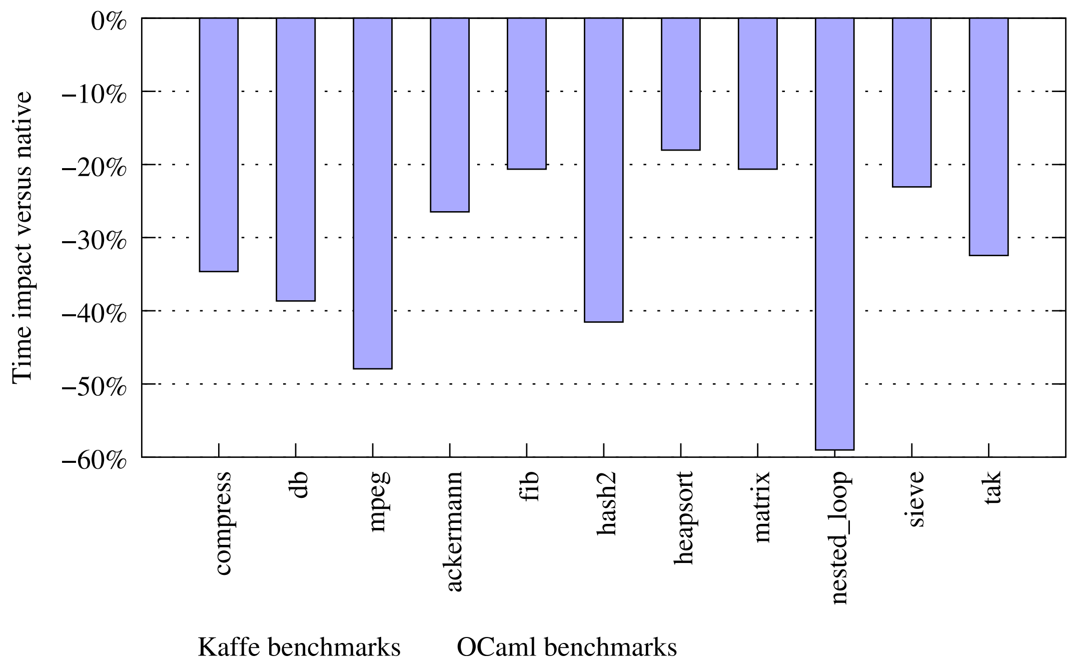

Figure 9.6: The performance impact of logical trace optimizations on three Kaffe benchmarks and eight OCaml benchmarks.

9.4 Program Shepherding

Perhaps the greatest threat to our information infrastructure is the remote exploitation of program vulnerabilities. The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program, coercing it into performing actions that it was never intended to perform. These attacks violate the *execution model* followed by legitimate programs, exploiting the difference between the hardware instruction set interface and the narrower execution model of typical programs. We present a technique called *program shepherding* [Kiriansky et al. 2002, Kiriansky 2003] that makes use of runtime code manipulation to efficiently monitor and enforce an application's execution model, preventing a wide range of security attacks. The ability of a runtime code manipulation system to observe every single application instruction, modify existing instructions, and insert new instructions, all with minimal performance impact, are crucial to enabling our secure execution environment to be built.

We first describe what comprises an application's execution model (Section 9.4.1) and then detail the approach of program shepherding (Section 9.4.2) and the security policies that can be built with it (Section 9.4.3). We call out a specific calling convention technique (Section 9.4.4) and discuss how to protect DynamoRIO's own memory from attacks (Section 9.4.5). Protection is also relevant to transparency (Section 3.3.2) by preventing inadvertent application errors from corrupting DynamoRIO state. Secure execution environment related work is discussed in Section 10.2.8.

9.4.1 Execution Model

The execution model of a program includes several components. At the lowest level, the Application Binary Interface (ABI) [UNIX Press 1993] specifies the register usage and calling conventions of the underlying architecture, along with the operating system interface mechanism. Higher-level conventions come from the source language of the program in the form of runtime data structure usage and expected interaction with the operating system and with system libraries. Finally, the program itself is intended by the programmer to perform a limited set of actions.

Even the lowest level, the ABI, is not efficiently enforceable. For example, the underlying hardware has no support for ensuring that calls and returns match, and it is prohibitively expensive to implement this in software. For this reason, the execution model is a convention rather than a strict set of rules. However, most security exploits stem from violations of the execution model. The most prevalent attacks today involve overwriting a stored program address with a pointer to injected malicious code. The transfer of control to that code is not allowed under the program's execution model — enforcing the model would thwart many security attacks.

Much work has been done on enforcing the execution model's specifications on data usage, from sandboxing the address space [Wahbe et al. 1993] to enforcing non-executable privileges on data pages [PaX Team] and stack pages [Designer]. However, these schemes have significant performance costs, as restrictions on data usage are very difficult to enforce efficiently. This is because memory references all look very similar statically and must be disambiguated dynamically. Distinguishing memory references requires expensive runtime checks on every memory access.

Most security attacks target not just any data, but data storing program addresses. Even limiting data protection to these locations, protecting the data is extremely difficult, since these addresses are stored in many different places and are legitimately manipulated by the application, compiler, linker, and loader. We restrict our enforcement of the execution model to the set of allowed control transfers. We focus on control transfers rather than on data, since they inhabit a smaller and more easily managed space than arbitrary data restrictions, and because nearly all unintended program actions surface as unintended control flow, although they may begin as abnormal data operations. Invariably, an attack that overwrites data has as its goal a malicious transfer of control.

9.4.2 Program Shepherding Components

The program shepherding approach to preventing execution of malicious code is to monitor all control transfers to ensure that each satisfies a given security policy, which is based on the program's execution model. This requires verifying every branch instruction, which is not easily done via static instrumentation due to the dynamism of shared libraries and indirect branches. Even ignoring the difficulties of statically handling dynamic behavior, the introduced checks impose significant performance penalties. Furthermore, an attacker aware of the instrumentation could design an attack to overwrite or bypass the checks. Static instrumentation will not work.

Program shepherding fits naturally in a runtime code manipulation infrastructure. In addition to providing a control point from which to examine and modify or instrument every application control transfer, a runtime system's code cache is pivotal to performing *efficient* security checks. Caching allows many security checks to be performed only once, when the code is copied to the cache. If the code cache is protected from malicious modification (Section 9.4.5), future executions of the trusted cached code proceed with no security or emulation overhead. The performance impact of program shepherding's core techniques (Figure 9.7) is in the noise for most benchmarks. Section 9.4.5 discusses the performance impact of protecting DynamoRIO's own memory from being compromised, which does add overhead.

Program shepherding is comprised of three techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing. The following subsections explain each technique and how it is implemented in DynamoRIO.

Restricted Code Origins

As many security attacks inject malicious code, a key security policy feature is restricting execution to code that belongs to the application. In monitoring all code that is executed, each instruction's origins are checked against the security policy to see if it should be given execute privileges. Typical code origins categories are: from the original image on disk and unmodified, dynamically generated but unmodified since generation, and code that has been modified. Finer distinctions

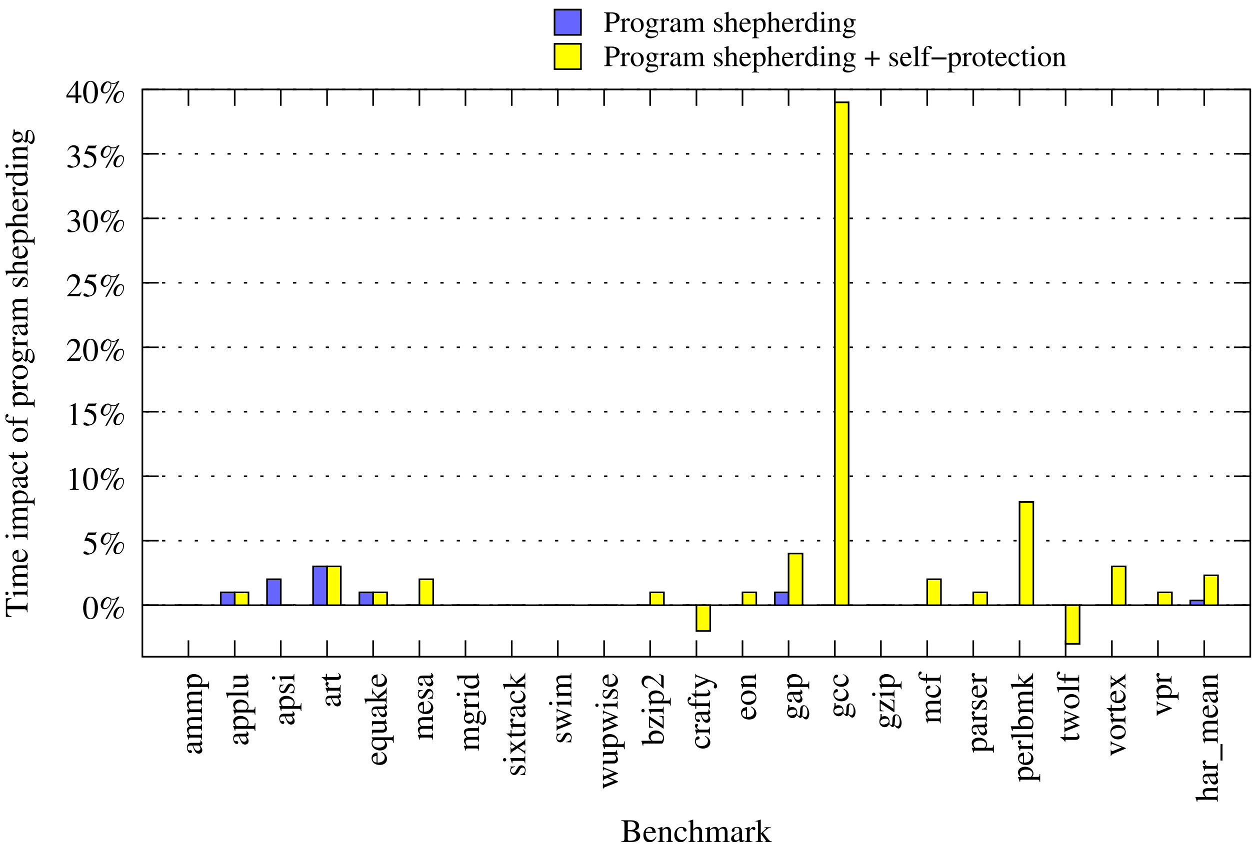

Figure 9.7: The performance impact of program shepherding's three techniques for a typical policy [Kiriansky et al. 2002]. Without self-protection (Section 9.4.5), the overhead is in the noise. With protection, `gcc` has a significant slowdown.

could also be made. Restricting execution to trusted code is accomplished by adding checks at the point where DynamoRIO copies a basic block into the code cache. These checks need be executed only once for each basic block.

Code origin checking often requires that DynamoRIO know whether code has been modified from its original image on disk, or whether it is dynamically generated. DynamoRIO's cache consistency algorithm (see Section 6.2) already keeps track of whether code has been modified in order to avoid stale code in its cache. A few simple additions to the information it tracks for its own correctness purposes are all we need.

Though not available on IA-32, a hardware execute flag for memory pages can provide similar features to our restricted code origins. However, it cannot by itself duplicate program shepherding's features because it cannot stop inadvertent or malicious changes to protection flags. Program shepherding can use un-circumventable sandboxing, described below, to prevent this from happening on inserted checks around system calls that might change execute flags. Furthermore, program shepherding provides more than one bit of privilege information: it distinguishes different types of execute privileges for which different security policies may be specified.

Restricted Control Transfers

Program shepherding allows arbitrary restrictions to be placed on control transfers in an efficient manner. These restrictions can be based on both the source and destination of a transfer as well as the type of transfer (direct or indirect call, return, jump, etc.). For example, we can forbid execution of shared library code except through declared entry points. Another example is providing some enforcement of the calling convention by requiring that a return instruction only target the instruction after a call. Fully enforcing the calling convention such that a callee only returns to its specific caller is much more difficult, but in Section 9.4.4 we present a novel scheme for doing so efficiently, using rarely-used multimedia extensions on the Pentium 4 processor.

DynamoRIO's comprehensiveness makes monitoring control flow transfers very simple. For direct branches, the desired security checks are performed at the point of basic block linking. If a transition between two blocks is disallowed by the security policy, they are not linked together. Instead, the direct branch is linked to a routine that announces or handles the security violation. These checks need only be performed once for each potential link, so a link that is allowed becomes a direct jump with no overhead.

Indirect control transfer policies add no performance overhead in the steady state, since no checks are required when execution continues on the same trace. Otherwise, the hashtable lookup routine translates the target program address into a fragment entry address. A separate hashtable is used for different types of indirect branches (returns, indirect calls, and indirect jumps) to enable type-specific restrictions without sacrificing performance. Security checks for indirect transfers that only examine their targets have little performance overhead, since we place in the hashtable only targets that are allowed by the security policy. A sample policy might match targets of indirect branches against entry points of imported and dynamically resolved symbols to enforce restrictions on inter-segment transitions, and targets of returns versus instructions after call sites. Security checks on both the source and the target of a transfer will have a slightly slower hashtable lookup routine. We have not yet implemented any policies that examine the source and the target, or apply transformations to the target, so we do not have experimental results to show the actual performance impact of such schemes.

Finally, we must handle kernel-mediated control flow, which is already intercepted by DynamoRIO (see Section 5.3). If a security policy wishes to restrict targets of signals or callbacks, we would simply place security checks at DynamoRIO's interception points. These non-explicit control transfers are infrequent enough that extra checks upon their interception do not affect overall performance.

Un-Circumventable Sandboxing

Program shepherding provides direct support for restricting code origins and control transfers. Execution can be restricted in other ways by adding sandboxing checks on other types of operations, inserted into a basic block when it is copied to the code cache. With the ability to monitor all transfers of control, program shepherding is able to guarantee that these sandboxing checks cannot be bypassed. Normal sandboxing has no such guarantee, and can never provide true security — if an attack can gain control of program execution, it can jump straight to the sandboxed operation, bypassing the checks. DynamoRIO only allows control flow transfers to the top of basic blocks or traces in the code cache, preventing this. Any branch that targets the middle of an existing block will go back to DynamoRIO and end up copying a new basic block into the code cache that will duplicate the bottom half of the existing block (see Figure 9.8). The necessary checks will be added to the new block, and the block will only be entered from the top, ensuring that we follow the security policy. Restricted code cache entry points are crucial not just for custom security policies that want un-circumventable sandboxing, but also for enforcing the other shepherding features by protecting DynamoRIO itself. This is discussed in Section 9.4.5.

When sandboxing system calls, if the system call number is determined statically, we avoid the sandboxing checks for system calls we are not interested in. This is important for providing performance on applications that perform many system calls.

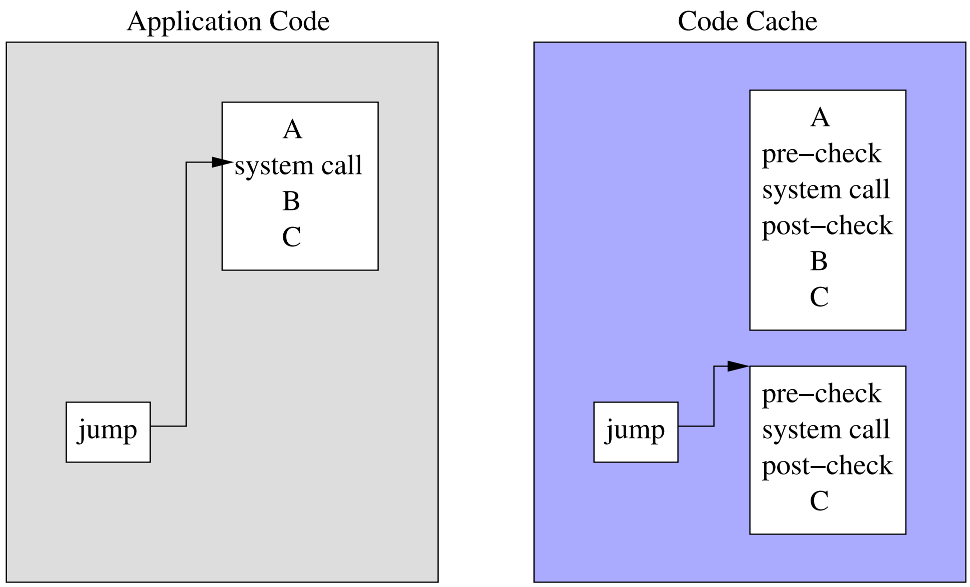

Figure 9.8: How un-circumventable sandboxing provides unique entry points, preventing sandboxing checks from being bypassed by tail-duplicating basic blocks and only allowing entry from the top.

9.4.3 Security Policies

Program shepherding's three techniques can be used to provide powerful security guarantees. They allow us to strictly enforce a safe subset of the instruction set architecture and the operating system interface, more closely matching the program's execution model. However, the execution model does vary by application, and cannot always be matched exactly, resulting in tradeoffs between program freedom and security: if restrictions are too strict, many false alarms will result when there is no actual intrusion. See Kiriansky et al. [2002] for discussion of the potential design space of security policies to provide significant protection for reasonable restrictions of program freedom. To give an idea of how effective program shepherding can be, Table 9.9 shows the effects of a sample security policy toward stopping the categories of attacks described in Kiriansky et al. [2002].

Program shepherding's guaranteed sandboxing can also be used for intrusion detection. A security policy must decide what to check for (for example, suspicious calls to system calls like `execve`) and what to do when an intrusion is actually detected. These issues are beyond the scope of our project, but have been discussed elsewhere [Goldberg et al. 1996, Ko et al. 2000].

<table>
<tr><th colspan="6">Attack Type</th><th>Code Origins</th><th>Transfers</th><th>Sandboxing</th></tr>
<tr><td colspan="6">Injected code</td><td>stopped</td><td></td><td></td></tr>
<tr><td rowspan="7">Existing Code</td><td colspan="5">Chained calls</td><td></td><td>stopped</td><td></td></tr>
<tr><td rowspan="6">Other Transfer</td><td colspan="4">Return</td><td></td><td>hindered</td><td>hindered</td></tr>
<tr><td rowspan="5">Call or jump</td><td rowspan="2">Inter-segment</td><td colspan="2">Not imported</td><td></td><td>stopped</td><td></td></tr>
<tr><td colspan="2">Imported</td><td></td><td></td><td>hindered</td></tr>
<tr><td rowspan="3">Intra-segment</td><td rowspan="2">Have entry info</td><td>Mid-func</td><td></td><td>stopped</td><td></td></tr>
<tr><td>Func entry</td><td></td><td></td><td>hindered</td></tr>
<tr><td colspan="2">No info</td><td></td><td></td><td>hindered</td></tr>
</table>

Table 9.9: Capabilities of program shepherding against different attack classes. The order of the techniques indicates the preferred order for stopping attacks. If a technique to the left completely stops an attack, we do not invoke other techniques (e.g., sandboxing is capable of stopping some attacks of every type, but we only use it when the other techniques do not provide full protection).

9.4.4 Calling Convention Enforcement

This section focuses on a particular instance of restricting control transfers, restrictions on return instructions. The most prevalent type of remote exploit attack today involves overwriting return addresses to gain control of a target program. A lot of work has been done on protecting return addresses and on detecting changes in return addresses [Cowan et al. 1998, Frantzen and Shuey 2001, Vendicator], which are discussed further in Section 10.2.8. Simple policies, such as requiring that return instructions target call sites only, can provide significant security with negligible cost in our implementation. Full calling convention enforcement — requiring that callees return only to their specific callers — requires a much more heavyweight implementation.

We have developed a novel scheme for protection of return addresses that enforces the full calling convention with minimal overhead. Unlike all other software schemes of which we are aware, this algorithm is able to efficiently prevent an algorithm-aware attacker from overwriting any of its storage and thus compromising the protection.

Intel's processors have included a return stack buffer (RSB) since the Pentium Pro (see Section 4.2). The RSB is of limited size and is used as a branch predictor for return instructions. On a call the return address is pushed onto the RSB, and on a return the top RSB value is popped and used as the predicted target of the return. Since the hardware is storing each return address, it is only natural to propose using the RSB to enforce the calling convention [Kiriansky 2003, McGregor et al. 2003].

As modern processors do not allow software control of the RSB, we have implemented a call stack using the multimedia registers of the Pentium 4. Most programs do not make use of these processor extensions. The Pentium 4 SSE2 extensions include eight 128-bit registers (the *XMM* registers) that can hold integral values. For a program that does not use these registers, they can be stolen and used as a call stack.

Using registers provides several key advantages over storing a call stack in memory. First, it scales to multiple threads very naturally, with no overhead, as the operating system will save and restore the register state for each thread. Second, it allows isolation from application access, both reading and writing. Program shepherding's complete control of all executed code allows us to ensure that our stolen registers are never accessed by application instructions. Contrast this to using memory to store a call stack. Either a dispatch by thread, or special thread-local storage, must be used to extend the scheme to multiple threads. Memory protection must be invoked on each access, maintaining both write-protection and read-protection from the application. Write-protection is not enough, as an attack could be devised using the ability to view the stored addresses in the call stack. These memory protection calls on every call and return are prohibitively expensive, and avoiding them by using registers is a significant advantage.

The SSE2 instruction set includes instructions for transferring a 16-bit value into or out of one of the eight 16-bit slots in each XMM register. Unfortunately, storing a 32-bit value is much less efficient. However, just the lower 16 bits of return addresses are sufficient to distinguish over 97% of valid addresses, as shown in Table 9.10. For a number of applications there are no return addresses that share their least significant 16 bits. Using just the lower 16 bits, then, does not sacrifice much security. It also allows twice as many return addresses to be stored in our register stack.

We implemented a scheme where the XMM registers form a rotating stack. The final 16-bit slot is used to store the call depth, leaving room for 63 return address entries. On a call, the return address is stored in the first slot and the rest of the slots are shifted over. When the call depth exceeds 63, the oldest 32 values are copied to memory that is then protected. On a return, the first slot's value is compared with the actual return address. Then the slots are all shifted down. When the call depth reaches 0, the most recent stored values are swapped back in to the first 32 register

benchmark	total	shared	percent shared
ammp	321	4	1.2%
applu	372	2	0.5%
apsi	1153	14	1.2%
art	160	0	0.0%
equake	270	2	0.7%
mesa	400	0	0.0%
mgrid	389	0	0.0%
sixtrack	2707	86	3.2%
swim	393	2	0.5%
wupwise	548	4	0.7%
bzip2	197	0	0.0%
crafty	895	16	1.8%
eon	1866	63	3.4%
gap	1779	39	2.2%
gcc	7723	843	10.9%
gzip	199	0	0.0%
mcf	214	0	0.0%
parser	1061	10	0.9%
perlbmk	2696	106	3.9%
twolf	1012	12	1.2%
vortex	3371	135	4.0%
vpr	1014	12	1.2%
average	1880	116	2.6%

Table 9.10: Return address sharing for each reference data set run of the SPEC CPU2000 benchmarks [Standard Performance Evaluation Corporation 2000], compiled with `gcc -O3`. The first column gives the total number of unique return addresses dynamically encountered. The second column lists the number of addresses that share their least significant 16 bits, while the final column shows the percentage of total addresses that share their bottom bits. For benchmarks with multiple datasets, the highest percentage run is shown.

slots. Only copying half of the stack avoids thrashing due to a frequent series of small call depth changes. Expensive memory protection is only required on every call depth change of 32.

Figure 9.11 shows the performance impact of maintaining a complete shadow call stack in the XMM registers, and checking that each return matches the appropriate call site. The results shown are for a prototype implementation, and we expect that better results are achievable by optimizing the shadow stack implementation. The overhead comes from the instructions required to push and pop our shadow stack on each call and return, and so the overhead is greater in programs that have more calls and returns.

To handle `setjmp()` and `longjmp()`, the `jmp_buf` should be write-protected between the

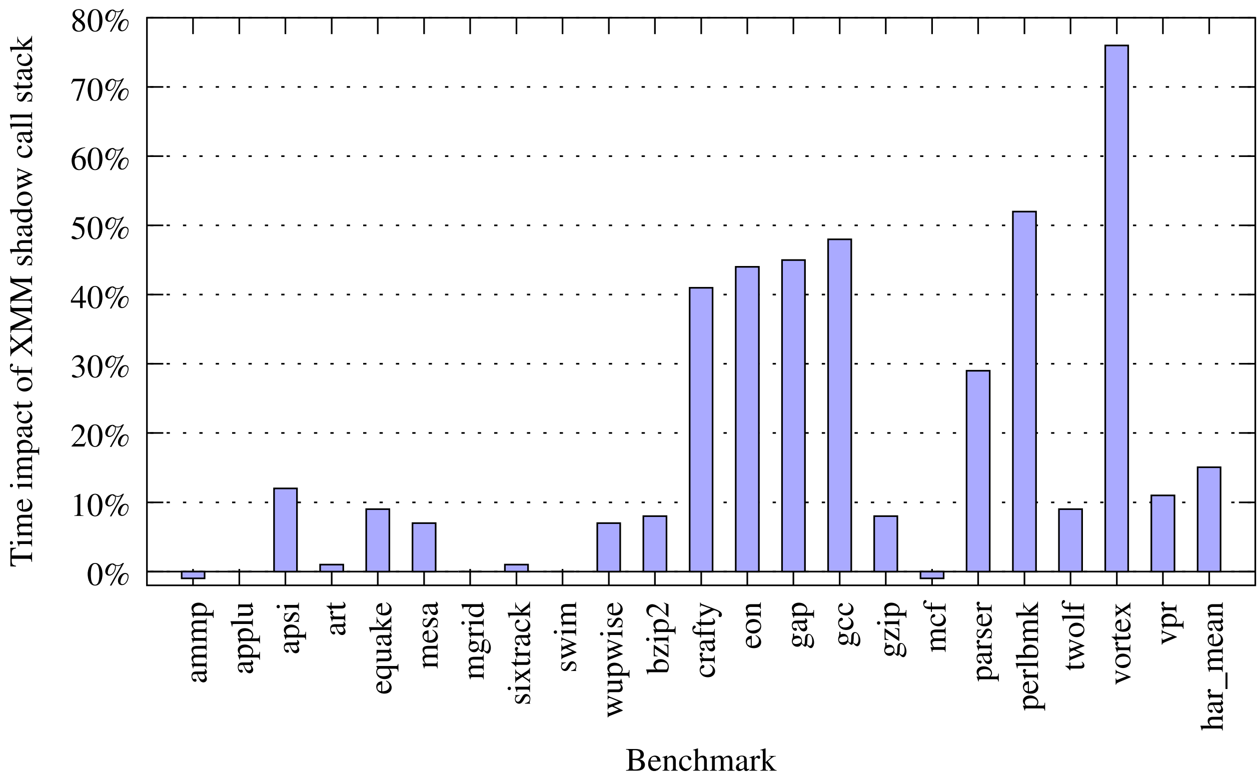

Figure 9.11: The performance overhead of our XMM register call stack scheme. This is a prototype implementation, and we expect that these results can be improved considerably.

`setjmp()` and the corresponding `longjmp()`, and the return address stack must be unwound to the proper location. We have not implemented this yet, and our system reports calling convention violations when it encounters `longjmp()` (for example, in `perlbmk` in SPEC CPU2000). One method for handling `longjmp()` used by Prasad and Chiueh [2003] is to keep popping the shadow stack on a miss until a hit is found (or the stack bottoms out). This ensures that any return to an ancestor will result in the proper stack unwinding.

9.4.5 Protecting DynamoRIO Itself

Program shepherding could be defeated by attacking DynamoRIO's own data structures, including the code cache, which are in the same address space as the application. This section discusses how to prevent the application from modifying DynamoRIO's data. Not only does this prevent targeted, malevolent writes to our data, but it also stops inadvertent writes due to application bugs. For full

Page Type	DynamoRIO mode	Application mode
Application code	R	R
Application data	RW	RW
DynamoRIO code cache	RW	R (E)
DynamoRIO code	R (E)	R
DynamoRIO data	RW	R

Table 9.12: Privileges of each type of memory page belonging to the application process. R stands for Read, W for Write, and E for Execute. We separate execute privileges here to make it clear what code is allowed by DynamoRIO to execute, although IA-32 does not distinguish execute privileges.

transparency in a runtime code manipulation system, its data should be protected so that erroneous application writes are faithfully passed to the application and do not result in the runtime system failing (see Section 3.3.2).

Memory Protection

We divide execution into two modes, each with different privileges: DynamoRIO mode and application mode. DynamoRIO mode corresponds to DynamoRIO code, while application mode corresponds to the code cache and the DynamoRIO-generated routines that are executed inside the cache without performing a context switch back to DynamoRIO. For the two modes, we give each type of memory page the privileges shown in Table 9.12. DynamoRIO data includes the indirect branch hashtable and other data structures.

All application and DynamoRIO code pages are write-protected in both modes. Application data is of course writable in application mode, and there is no reason to protect it from DynamoRIO, so it remains writable in DynamoRIO mode. DynamoRIO's data and the code cache can be written to by DynamoRIO itself, but they must be protected during application mode to prevent inadvertent or malicious modification by the application.

Protecting all critical DynamoRIO data does involve some sacrifices, since such data cannot be written from our own code cache routines. An example is the target that is stored for our in-cache trace head counter increment scheme (see Section 2.3.2). We cannot use this scheme and still protect everything, since that pointer must be written from the cache and therefore cannot be protected from the application. We must carefully design everything such that all important state is only written from DynamoRIO mode, without sacrificing too much efficiency. Another example is using a prefix on indirect branch targets that restores a register, to avoid needing to store the target address in memory (see Section 4.3.2). The performance impact of self-protection is shown in Figure 9.7. Only `gcc` has significant overhead, and we believe that with optimization of our page protection scheme by being aware of the operating system's algorithm for propagating page privilege changes this overhead can be reduced.

If a basic block copied to the code cache contains a system call that may change page privileges, the call is sandboxed to prevent changes that violate Table 9.12. Program shepherding's un-circumventable sandboxing guarantees that these system call checks are executed. Because the DynamoRIO data pages and the code cache pages are write-protected when in application mode, and we do not allow application code to change these protections, we guarantee that DynamoRIO's state cannot be corrupted.

We should also protect DynamoRIO's Global Offset Table (GOT) [Tool Interface Standards Committee 1995] on Linux (and corresponding structure on Windows) by binding all symbols on program startup and then write-protecting the GOT, although our prototype implementation does not yet do this.

Multiple Application Threads

As discussed in Section 6.1.2, DynamoRIO's data structures and code cache are thread-private: each thread has its own unique code cache and data structures. Application system calls that modify page privileges are checked against the data pages of all threads. When a thread enters DynamoRIO mode, only that thread's DynamoRIO data pages and code cache pages are unprotected.

A potential attack could occur while one thread is in DynamoRIO mode and another thread in application mode modifies the first thread's DynamoRIO data pages. We could solve this problem by forcing all threads to exit application mode when any one thread enters DynamoRIO mode. We have not yet implemented this solution, but its performance cost would be minimal on a single processor or on a multiprocessor when every thread is spending most of its time executing in the code cache. However, the performance cost would be unreasonable on a multiprocessor when threads are continuously context switching. Investigating alternative solutions is future work.

On Windows, we also need to prevent the system call `NtSetContextThread` from setting register values in other threads. DynamoRIO's hashtable lookup routine uses a register as temporary storage for the indirect branch target. If that register were overwritten, DynamoRIO could lose control of the application. Our restriction of this system call has not interfered with the execution of any of the large applications we have been running. In fact, we have yet to observe any calls to it.

9.5 Chapter Summary

Improving the performance of applications while they execute opens up whole new areas of optimization and program deployment. It allows the optimization focus to shift from program developers to users' runtime environments. Yet, as this thesis has shown, building runtime code manipulation systems is challenging. Leveraging the DynamoRIO tool platform allows optimizations to be developed without having to build the system infrastructure. For language interpreters, this allows new domain-specific languages to achieve reasonable performance without the significant effort of building a just-in-time compiler, using the notion of the logical program counter. This can be expanded to a more general *location of interest* for application to a wider variety of interpreters [Leger 2004]. This expanded notion of logical traces could be combined with our custom trace building client API (Section 8.2.4) to create a general interface for directing trace building.

Program shepherding is another example of utilizing the power of DynamoRIO to create an unprecedented tool. Program shepherding's three techniques are easily implemented with little overhead in DynamoRIO, and are successful at stopping the majority of today's security attacks. Program shepherding does have limitations, though, relying on DynamoRIO's injection techniques to take over a target application (see Section 5.5), which could be bypassed if an attacker has other methods of manipulating a target machine. Similarly, typical code origins policies often assume that the executable image on disk is safe, which may not be the case if the machine has already been attacked. However, once a machine is breached in any way, security guarantees from other parts of the system must be discarded. We believe that program shepherding will be an integral part of future security systems. It is easily deployable and coexists with existing operating systems, applications, and hardware. Many other security components can be built on top of the un-circumventable sandboxing provided by program shepherding.

This chapter has demonstrated the versatility and wide variety of uses of runtime code manipulation technology. Our examples here only scratch the surface, and we look forward to future research in the area.

Chapter 10

Related Work

Many areas of research involve generating or manipulating code at runtime, or building tools that operate on binaries at link time or runtime. This chapter first describes these related areas of work (Section 10.1) and then walks through technology features present in these systems (Section 10.2) that are relevant to DynamoRIO.

10.1 Related Systems

Runtime code manipulation has been used in special-purpose systems for years. Dynamic loaders patch code at runtime, debuggers allow modification of programs being debugged, and just-in-time compilers employ dynamic code generation. However, few general frameworks for creating runtime code manipulation tools have been developed. Systems relevant to our work are those that do provide custom tool development, at link time or later, as well as those that provide comprehensive interposition between a running application and the underlying hardware. This thesis is about the combination of these features.

Table 10.1 summarizes the high-level characteristics of the major systems discussed in this section. Systems for building custom tools that operate on binaries (Section 10.1.1 and Section 10.1.2) are predominantly static. Static binary manipulation emerged as a reaction to separate compilation and third-party modules for which source code is not available. However, static operation on binaries faces challenges with code discovery: distinguishing code from data, and finding all code that could be executed. Furthermore, static tools cannot handle dynamic behavior well. Runtime tool platforms have been developed to overcome these shortcomings. Most of these platforms have fo-

System Name	Runtime?	Comprehensive?	Interface?	Code Cache?
Runtime Code Manipulation				
DynamoRIO	√	√	√	√
Strata	√	√	√	√
Valgrind	√	√	√	√
Binary Instrumentation				
Dyninst	√		√	
Vulcan	√		√	
Detours	√		√	
Chaperon	√	√		√
Pin	√	√	√	√
ATOM			√	
Etch	profile run		√	
EEL			√	
InCert			√	
Morph			√	
Hardware Simulation and Emulation				
Shade	√	√		√
Embra	√	√		√
Simics	√	√		√
Daisy	√	√		√
Crusoe	√	√		√
DELI	√	√	√	√
virtual machines	√	some		√
Binary Translation				
Aries	√	√		√
Walkabout	√	√		√
Dynamite	√	√		√
FX!32	√	√		√
UQBT				
VEST, mx				
Binary Optimization				
Dynamo	√	√		√
Mojo	√	√		√
Wiggins/Redstone	√			
post-link systems				
Dynamic Compilation				
DyC, Tempo, Fabius	√		√	
JITs	√			

Table 10.1: A comparison of related systems that manipulate code at link time or later. Four questions are asked of each: whether the system operates at runtime; whether it is *comprehensive*, systematically interposing itself between *every* instruction a target application executes and the underlying hardware; whether the system exports an interface for constructing custom tools; and whether it uses code caching technology. Code caching systems are further compared in Table 10.2.

cused on small numbers of instrumentation points, rather than more-difficult-to-support wholesale code transformations. While many tools can and have been built with this type of instrumentation, most forms of optimization and other broad classes of tools are not supported.

Comprehensive runtime interposition is present in several types of systems. Many of these, including hardware simulators and emulators (Section 10.1.3) and instruction set translators (Section 10.1.4), either intercept only a subset of executed code, or require hardware support to be efficient. For example, virtual machine monitors avoid binary manipulation when they can, for simplicity and efficiency. Only on architectures that are not virtualizable (such as IA-32) must they resort to binary manipulation, and then they restrict themselves to manipulating only the code that is needed for virtualization (privileged-mode code). Simulators and translators have never achieved better than a worst case of several times slowdown. Additionally, these systems often execute an entire operating system workload, which is the wrong level of abstraction for building deployable tools targeting applications in actual use.

Binary optimizers (Section 10.1.5) have attained good performance, but most operate statically and do not support dynamic behavior. Dynamic binary optimization in software was first achieved by Dynamo [Bala et al. 2000]. Despite its initial success, Dynamo and the dynamic optimizers that followed it never made the difficult transition from running single-threaded benchmarks to transparently executing all applications efficiently. Furthermore, Dynamo aborted and returned to native execution when performance was bad, which is fine for optimization, but not for general tool support.

Runtime code generation has been used in dynamic compilers (Section 10.1.6), but they require source code access and cannot be used to study arbitrary binaries.

The contribution of our system, DynamoRIO, is in providing efficient, transparent, and comprehensive runtime code manipulation, and combining it with customizable tool support. Our basic techniques of code caching and linking are well-known, but have only recently been applied to building custom code transformations. Simulators and translators have not been used as control points for manipulating native applications, just like virtual machines existed for cross-architectural purposes for many years before they were used for native virtualization.

The following sub-sections discuss these related systems in terms of their goals and how they relate at a high level to DynamoRIO. Specific technologies and components of these systems that are relevant to our work are discussed in Section 10.2.

10.1.1 Runtime Code Manipulation

We know of only two other systems that explicitly provide support for manipulating any or all of an arbitrary binary's code on commodity hardware, both developed concurrently to DynamoRIO: Strata and Valgrind.

Strata [Scott et al. 2003, Scott and Davidson 2001] is a runtime code manipulation system whose goal is generality and re-targetability. It is useful for prototyping runtime tools that can be evaluated on small benchmarks. However, it has no support for transparency or threads and cannot run large, complex applications. Various Strata modules can be replaced to build custom tools, but this is not an ideal interface, for two reasons. It does not isolate the core system from applications of the system; and, by not providing an API, but rather handing the tool builder the source code to the whole system, it is not at the proper abstraction level for development of most tools. Recently Strata has been fitted with an interface called FIST [Kumar et al. 2003], which provides an event-response model for instrumentation. However, the interface focuses on inserting jumps to handler routines, and not on arbitrary code transformations.

Valgrind [Seward 2002] was originally built as a memory debugger, but it now exports an interface for building general tools [Nethercote and Seward 2003]. Partly because it translates IA-32 instructions to micro-operations for easier handling, its performance does not approach native performance.

10.1.2 Binary Instrumentation

Several dynamic instrumentation systems have been developed in the last few years. Dyninst [Buck and Hollingsworth 2000], Kerninst [Tamches and Miller 1999], Detours [Hunt and Brubacher 1999], and Vulcan [Srivastava et al. 2001] can insert code into running processes. All three target IA-32, along with other architectures. Dyninst and Kerninst are based on dynamic instrumentation technology [Hollingsworth et al. 1994] developed as part of the Paradyn Parallel Performance Tools project [Miller et al. 1995], while Detours and Vulcan rely on special features of the Windows operating system. These tools are successful at observing modern, dynamic applications, but because they modify the original code in memory by inserting trampolines, they suffer from transparency problems. Additionally, extensive modification of the code quickly becomes unwieldy through these mechanisms, especially in the face of variable-length IA-32 instructions. For example, attempting to perform even simple optimizations such as the `inc` to `add 1` transformation for the Pentium 4 (Section 9.2.2) would not be viable with these systems, as `add 1` is a longer instruction than `inc` and a trampoline would have to be used.

Two recent runtime tools use code caches, giving greater control than trampoline-based methods, but do not provide interfaces that take advantage of their caches. Pin [Intel Corporation 2003] is a recently developed dynamic instrumentation system for IA-64, currently being ported to IA-32. Like the other dynamic tools, Pin focuses on inserting calls to instrumentation routines. It does allow modifying code, but only by replacing entire application procedures, and it does not support fine-grained code manipulation. Insure++ [Parasoft], a link-time instrumentation tool for memory error detection, ships with a purely runtime memory error detector called Chaperon for IA-32 Linux. It certainly uses a code cache, but details of how it operates are not published.

Static instrumentation systems include ATOM [Srivastava and Eustace 1994] and Morph [Zhang et al. 1997] for Alpha, InCert [Geodesic Systems 2001] for IA-32, and EEL [Larus and Schnarr 1995] for SPARC. Their instrumentation interfaces are the model for the more recent dynamic tools. Static instrumentation has several drawbacks: lack of support for dynamic application behavior, and modification of the executable itself. Etch [Romer et al. 1997] partially addresses this by combining a profiling run with static instrumentation on IA-32 Windows. It operates by first executing the program (through the Windows debugger interface) to discover the code boundaries and the modules dynamically loaded, and then instrumenting the now-known code. However, there is no guarantee that all code that will ever be executed was seen in that run, and furthermore Etch does not handle dynamically-modified or generated code.

All of these binary instrumentation systems are designed for creation of custom tools for information gathering. However, their methods and interfaces are not suited for code modification, in particular systematic, fine-grained transformations such as optimization. DynamoRIO, on the other hand, can be used to build optimization tools as easily as instrumentation tools.

10.1.3 Hardware Simulation and Emulation

Many code caching techniques were pioneered in instruction set emulators, such as Shade [Cmelik and Keppel 1994], and whole-system simulators like Simics [Magnusson et al. 1998] and Embra [Witchel and Rosenblum 1996]. Embra is a processor and cache simulator that is part of the SimOS [Rosenblum et al. 1995] whole-system simulator. Embra is capable of running large, real-world programs, including commercial operating systems. It handles self-modifying code and can self-host. However, it targets MIPS, a much simpler architecture than IA-32. On MIPS, self-modifying code is simple to detect and make forward progress on because of the explicit instruction cache flush required. Embra achieved unprecedented performance, an order of magnitude faster than other simulators, but its base performance is not close to native on most benchmarks, and no numbers are given for the operating system itself or the other large programs they report running. Embra has a simulation interface, but no interface for building other types of tools.

Emulators designed for ISA compatibility include Daisy [Ebcioglu and Altman 1997], a custom VLIW with support for using dynamic binary translation in software to emulate existing VLIW architectures, and Crusoe [Klaiber 2000], a custom VLIW with support for a software system for translating IA-32 code. Crusoe includes novel support for self-modifying code [Dehnert et al. 2003]; however, we could not use their techniques as we have neither a hardware-assisted IA-32 emulator nor control over the hardware page protection.

DELI [Desoli et al. 2002], like DynamoRIO, descended from the original Dynamo [Bala et al. 2000] dynamic optimization system. DELI is a runtime code translation system that exports an interface for customizing its behavior, which focuses on providing caching and linking services for binary translators and emulators. Its underlying platform is a custom VLIW embedded processor, and its primary goal is to support ISA compatibility by flexibly emulating other embedded processors.

Software virtual machines for IA-32 include Denali [Whitaker et al. 2002], Connectix VirtualPC [Connectix], VMWare [Bugnion et al. 1997], and Xen [Barham et al. 2003]. These systems are also able to run large, complex programs: commercial operating systems. However, they can ignore all user-mode code, only having to worry about privileged code. This means that they cannot be used as tools to study application behavior except in terms of its interactions with the operating system. One exception is the Connectix VirtualPC [Connectix] Macintosh product that translates from IA-32 to PowerPC, which must translate user mode code as well as privileged code, and is similar to whole-system emulators like Embra [Witchel and Rosenblum 1996] in that respect. Little technical information is available about its operation, however.

These simulators and virtual machines perform dynamic binary manipulation, and are able to execute entire operating systems and their workloads, to study whole-system behavior (for cache simulation or application tracing) or execute multiple operating systems simultaneously on the same hardware. In contrast, DynamoRIO's goal of easy deployment is about building tools that operate on a single application in a lightweight manner for use on commodity, production platforms. This means that our software layer must execute *on top of* the operating system. Counter-intuitively, this is more challenging in many ways than running the whole system. On top, one must transparently operate within the confines of the operating system, intercepting its transfers of control and handling its threads, all the while pretending that one is not occupying the application's address space to avoid interfering with its behavior. Furthermore, when executing underneath the operating system it is much harder to study individual application aspects such as threads without extensive knowledge of operating system internals, a difficult task for closed systems like Windows.

10.1.4 Binary Translation

Static binary translation systems built for architectural compatibility include DEC's Alpha migration tools for VAX and MIPS [Sites et al. 1992] and FX!32 [Chernoff et al. 1998], DEC's system for IA-32 Windows migration to Alpha which makes use of static translation combined with dynamic profiling and emulation. FX!32 deliberately avoids translation at runtime to avoid the performance hit, relying instead on ahead-of-time offline translation. General frameworks for translation have also been designed [Cifuentes and Emmerik 2000]. None of these translation systems provides an interface for customization. Their primary goal is translation from one ISA to another.

Dynamic translation systems manipulate code at runtime while translating from one instruction set to another, making them similar to instruction set emulators. They include Aries [Zheng and Thompson 2000] for PA-RISC to IA-64, Walkabout [Cifuentes et al. 2002, Ung and Cifuentes 2000] for IA-32 to SPARC, and Dynamite [Robinson 2001] for IA-32 to MIPS. Walkabout and

Dynamite separate the source and target architectures to create extensible systems that can be re-targeted. Dynamic translators do not have the pressure for performance of a native-to-native system, since they are not competing against a statically-optimized binary. The fact that they are translating allows leeway in final performance, and none of them have been shown to approach native performance when applied in a native-to-native setting. None exports any interface for customized transformations for tool building.

10.1.5 Binary Optimization

A number of static optimizers operate at the post-link stage on binaries: OM [Srivastava and Wall 1992] and alto [Muth et al. 2001] for Alpha, Spike [Cohn et al. 1997, Cohn and Lowney 1996] for Windows executables on Alpha, FDPR [Henis et al. 1999] for the IBM pSeries servers, and the recent Ispike [Luk et al. 2004] for Itanium. Their static operation limits them to programs with no dynamic loading or dynamically-generated or modified code, and their static view of the program for optimization purposes is only an estimate of runtime behavior.

DynamoRIO is based on dynamic optimization technology. Its ancestor is Dynamo [Bala et al. 2000, Bala et al. 1999], the original software dynamic optimization system for PA-RISC. Dynamo's sole goal was optimization, and it gave up (returning control to native execution) if it was not performing well. Its cache management and other policies were geared toward embedded applications, and it ended up giving up on many benchmarks. It was never scaled up to run large, complex applications, nor did it export an interface for customization.

Mojo [Chen et al. 2000] targeted Windows NT running on IA-32, and was able to execute several large desktop applications, but never tried to be anything other than a dynamic optimizer. Wiggins/Redstone [Deaver et al. 1999] employed program counter sampling to form traces which were then specialized for a particular Alpha micro-architecture, but no experimental results are available. The SIND project [Palmer et al. 2001] built a prototype dynamic optimization system, which was designed to be used for more general purposes, though no further results are available. A recent dynamic optimizer for IA-64, ADORE [Lu et al. 2004], uses hardware performance counters to dynamically deploy cache prefetching. It modifies the original program code in order to insert optimized traces into the program stream. This technique does not easily support general code transformations.

Hardware dynamic optimization of the instruction stream is performed in superscalar processors [Carmean et al. 2001, Kumar 1996, Taylor et al. 1998]. The Trace Cache [Rotenberg et al. 1996] allows such optimizations to be performed off of the critical path. The rePLay [Fahs et al. 2001, Patel and Lumetta 1999] framework extends trace caches with the notion of frames, which are single-entry, multiple-exit sequences that nearly always execute to completion. Another hardware proposal [Merten et al. 2001] stores traces in main memory for persistence across runs. None of these hardware schemes is able to open up any interfaces for software customization. Hardware profiling that exports its results to software has been proposed [Merten et al. 1999], but the hardware does not provide code manipulation support.

10.1.6 Dynamic Compilation

Dynamic compilation has been used in application-specific ways to produce specialized code at runtime for extensible operating systems [Pu et al. 1988] and graphics [Pike et al. 1985]. Systems for general dynamic code generation include ‘C [Engler et al. 1996], its compiler tcc [Poletto et al. 1997], and its fast code generation scheme VCODE [Engler 1996]. These tools require source code modification and cannot be used to study arbitrary binaries.

Dynamic compilation has proven essential for efficient implementation of high-level languages [Deutsch and Schiffman 1984, Adl-Tabatabai et al. 1998]. Some just-in-time (JIT) compilers perform profiling to identify which methods to spend more optimization time on [Sun Microsystems]. The Jikes Java virtual machine [Arnold et al. 2000, Krintz et al. 2001] utilizes idle processors in an SMP system to optimize code at runtime. Jikes optimizes all code at an initial low level of optimization, embedding profiling information that is used to trigger re-optimization of frequently executed code at higher levels. Self [Hölzle 1994] uses a similar adaptive optimization scheme.

Staged dynamic compilers [Leone and Dybvig 1997] postpone a portion of compilation until runtime, when code can be specialized based on runtime values. Such systems include DyC [Grant et al. 1999] and Tempo [Consel and Nöel 1996] for the C language and Fabius [Lee and Leone 1996] for ML. These systems usually focus on spending as little time as possible in the dynamic compiler, performing extensive offline pre-computations to avoid needing any intermediate representation at runtime. Some systems include an adaptive runtime component that continually selects hot code for re-compilation [Feigin 1999, Voss and Eigenmann 2000]. Kistler [Kistler and Franz 2001] proposes "continuous program optimization" that involves operating system re-design to support adaptive dynamic optimization.

10.2 Related Technology

The previous section discussed related systems from the high-level viewpoint of their goals. This section turns that around and looks at each code caching system in terms of its technology features. Table 10.2 summarizes the comparison, focusing on code cache capacity, thread support, and kernel-mediated control transfer interception. It also shows which systems share DynamoRIO's targets: stock hardware, on top of the operating system, and supporting the IA-32 architecture and the Windows operating system.

10.2.1 Code Cache

The basic components of DynamoRIO (presented in Chapter 2) are caching, linking, and building traces. Code caches have been around for a long time. Linking blocks in a code cache was called *chaining* in Shade [Cmelik and Keppel 1994]. Embra [Witchel and Rosenblum 1996] extended chaining to indirect branches with *speculative chaining*. Dynamo [Bala et al. 2000] further reduced indirect branch overhead by building traces across indirect branches using lightweight runtime profiling (Section 2.3.1). Trace building techniques in other systems are discussed in Section 2.3.3.

10.2.2 Transparency

Previous systems have discussed some aspects of transparency: Shade [Cmelik and Keppel 1993] and Daisy [Ebcioglu and Altman 1997] discuss timing and error transparency; Valgrind [Seward 2002] discusses avoiding modification to the application's memory layout, to support bailing out to native execution; and Strata [Scott et al. 2003] uses non-transparent code cache return addresses (which we rejected in Section 4.2.1) for the performance improvement, accepting the loss in transparency.

Machine contexts for signal handlers are translated to their native values in Dynamo [Bala et al. 2000]. Mojo [Chen et al. 2000] translates the program counter back to its native value for exception

System Name	Perf	Stock HW?	x86?	Δ Code?	Cache Cap?	On OS?	Win32?	Thrds?	KMCT?
DynamoRIO	0.6x-1.6x	√	√	√	√	√	√	√	√
Strata	1x-3x	√	√			√			
Valgrind	2x-8.5x	√	√			√		√	√
Chaperon	N/A	√	√	N/A	N/A	√			√
Pin	N/A	√	√		N/A	√			
Shade	3x-6x	√		√		√			√
Embra	3.5x-9x	√		√					√
Simics	26x-75x	√							√
Daisy	N/A		√	√					√
Crusoe	N/A		√	√					√
DELI	N/A			√					√
virtual machines	N/A	√	√	√	√				√
Aries	N/A	√		N/A	N/A	√		√	√
Walkabout	1x-177x	√	√			√			
Dynamite	N/A	√	√	N/A	N/A	√		N/A	N/A
FX!32	N/A	√	√			√	√	√	√
Dynamo	0.8x-bail	√				√			√
Mojo	1x-2x	√	√			√	√	√	√

Key to the features compared:

Perf Performance when applied native-to-native (no translation). We do not attempt to compare cross-architecture performance, and we only consider caching of all user-mode code (ruling out most virtual machines) for fair comparison. We used the most recent performance numbers we could find:

- Strata: Scott et al. [2003]
- Valgrind: Nethercote and Seward [2003]
- Shade: Cmelik and Keppel [1994]
- Embra: Witchel and Rosenblum [1996]
- Simics: Magnusson et al. [1998]
- Walkabout: Cifuentes et al. [2002]
- Dynamo: Bala et al. [1999]
- Mojo: Chen et al. [2000]

Stock HW? Does the system operate on unmodified, stock hardware?
x86? Does the system run on or translate from or to IA-32?
Δ Code? Does the system handle code modification?
Cache Cap? Does the system have other than a hardcoded cache capacity limit?
On OS? Does the system operate *on top of* the operating system?
Win32? Does the system execute applications on top of Windows?
Thrds? Does the system handle multiple threads?
KMCT? Does the system follow kernel-mediated control transfers?

Table 10.2: A comparison of the features provided by the runtime systems from Table 10.1 that use code caches. For commercial systems with little technical documentation, not all information was available, indicated by “N/A”.

handlers.

10.2.3 Architectural Challenges

Instruction representation in code caching systems is not discussed much, particularly on RISC architectures. Valgrind [Seward 2002] translates IA-32 into RISC-like micro-operations for easier processing, but loses performance in the process. Runtime code generation tools for IA-32 also focus on a RISC-like subset of IA-32 [Engler 1996].

Return address prediction discrepancies have been explored elsewhere [Kim and Smith 2003a, Kim and Smith 2003b]. Branch reachability solutions and proactive linking are both employed in the Dynamo system [Bala et al. 1999], and the performance advantages of open-address hashtables have been recognized in other systems [Small 1997].

Re-targetability has been a primary focus of several runtime systems [Scott et al. 2003, Cifuentes et al. 2002, Robinson 2001]. While the core of a system can be made easily re-targetable, our experience has shown that scaling up to large, modern, complex applications requires extensive architecture-specific work.

10.2.4 Operating System Challenges

Systems that operate underneath the operating system do not have to worry about threads, nor do those targeting embedded systems. Threads complicate many operations, including cache management. Mojo [Chen et al. 2000] uses a thread-private basic block cache and a thread-shared trace cache, which is managed in a heavyweight manner by suspending all other threads. Valgrind [Seward 2002] replaces the entire `pthreads` [Leroy] library on Linux in order to support threads.

Many systems intercept signals, and some delay asynchronous signals [Cmelik and Keppel 1994, Bala et al. 2000, Seward 2002], though few discuss issues with emulating the kernel. Mojo [Chen et al. 2000] and FX!32 [Chernoff et al. 1998] are the only systems we know of that handle Windows kernel-mediated control transfers. FX!32 does this by wrapping the entire Windows API (some 12,000 routines at the time, and it has grown since then) and modifying all arguments that are pointers to callback routines. This immense effort, primarily needed for mixing native and translated or emulated calling conventions, allows it to intercept callbacks and asynchronous procedure calls (see Section 5.3). Mojo was the first to use the much simpler technique that DynamoRIO built on, patching the user mode dispatcher for each type of kernel-mediated control transfer, though they give little information beyond the location to patch. We have extended these efforts with a uniform treatment of Linux and Windows control transfers in terms of keeping state, and in showing how to avoid the problems that Windows callbacks present with their unknown resumption points. We also show which system calls must be monitored in order to retain control.

Most systems do not discuss handling `fork` or `exec` system calls. Shade [Cmelik and Keppel 1994] mentions that it will lose control on an `exec`, and on a `fork` it fails to duplicate its open files and will end up sharing log files. DynamoRIO properly creates for the child process a new copy of each of its kernel objects that are not cloned by `fork` and ensures that it maintains control across an `exec` system call.

10.2.5 Code Cache Consistency

Any system with a software code cache is subject to the problem of cache consistency. However, most RISC architectures require an explicit instruction cache flush request by the application in order to correctly execute modified code, usually in the form of a special instruction [Keppel 1991]. Systems like Shade [Cmelik and Keppel 1994], Embra [Witchel and Rosenblum 1996], Dynamo [Bala et al. 2000], and Strata [Scott et al. 2003] watch for this instruction (or system call, on architectures where it is a privileged instruction and the system in question is executing on top of the operating system). They all invalidate their entire code cache upon seeing that any code has been modified (see Section 10.2.6 for further discussion of granularity of code cache eviction in these systems). DELI [Desoli et al. 2002] states that it handles self-modifying code, but gives no details on how this is achieved. Presumably it uses the same technique as the previously mentioned systems, as the architectures that DELI emulates require explicit application actions to execute modified code.

The IA-32 architecture requires no special action from applications to execute modified code (although the 386 and 486 processors require a branch in order to avoid stale code in the prefetch buffer, the Pentium and later processors all invalidate their prefetch buffers immediately upon detecting code modification). This makes it much more difficult to detect code changes, since the hardware must be matched and code modifications acted upon after each code write, rather than after a series of code modifications is complete, as in the case of the RISC architectures.

Due to the challenge of correctly detecting code changes, and because some programs do not require the feature, many systems targeting IA-32 do not handle modified code. As far as we know, these include Strata [Scott et al. 2003], Valgrind [Seward 2002], PIN [Intel Corporation 2003], Denali [Whitaker et al. 2002], Walkabout [Cifuentes et al. 2002], FX!32 [Chernoff et al. 1998], and Mojo [Chen et al. 2000], .

Other systems that target IA-32 must, like DynamoRIO, turn to page protection. Daisy [Ebcioglu and Altman 1997] uses hardware-assisted page protection, making use of a page table bit that is inaccessible to the application to indicate whether a page has been modified. When a write is detected on a code page, that whole page is invalidated in the code cache. Similarly, Crusoe [Klaiber 2000] uses page protection, with hardware assistance in the form of finer-grained protection regions than IA-32 pages. Although they have an IA-32 emulator, they augment their page protection with similar mechanisms to our sandboxing for detecting self-modifying code on writable pages [Dehnert et al. 2003]. VMWare [Bugnion et al. 1997] also uses a combined strategy of page protection and sandboxing. Connectix [Connectix] purportedly uses page protection as well, though details are not available.

None of these IA-32 systems that correctly handle self-modifying code needs to worry about multiple threads, as they all execute underneath the operating system. Ours is the only software code cache that we know of that has tackled the combination of multiple application threads and cache consistency on IA-32.

Software code cache issues with multiple threads are related to the cache coherency problem in software distributed shared memory [Brian N. Bershad and Sawdon 1993, Carter et al. 1991, Li and Hudak 1989]. Cache coherency implementations are made more efficient by relaxing the sequential consistency model [Lamport 1979] to weak consistency [Dubois et al. 1986], release consistency [Gharachorloo et al. 1990], and other models that allow significant optimizations of the coherence protocol. In a similar manner, we present in Section 6.2.6 an efficient code cache invalidation scheme based on relaxing the consistency model. However, distributed memory systems are able to modify the programming model in order to obtain information from the application without which the consistency model cannot safely be met, which we cannot do. This has prevented us from making our consistency relaxation one hundred percent bulletproof (see Section 6.2.6).

10.2.6 Code Cache Capacity

There has been little work on optimally sizing software code caches. Nearly every system known to us (the exceptions are the virtual machines Connectix VirtualPC [Connectix] and VMWare [Bugnion et al. 1997]) sizes its cache generously and assumes that limit will rarely be reached. Furthermore, cache management is usually limited to flushing the entire cache, or splitting it into two sections that are alternately flushed [Chen et al. 2000]. Valgrind [Seward 2002] performs first-in-first-out (FIFO) single-fragment replacement when its cache fills up. For many of these systems, the cache is an optimization that, while critical for performance, is not critical to the workings of the system. They can fall back on their emulation or translation core. And for systems whose goal is performance, their benchmark targets (like SPECCPU [Standard Performance Evaluation Corporation 2000]) execute relatively small amounts of code.

Shade [Cmelik and Keppel 1993] proposed making its cache management more sophisticated, but worried about the link tracking required to remove single fragments from the code cache. Hazelwood and Smith [2004] also focus on link tracking overhead, but the numbers given are calculated from equations derived from instruction count profiling, which is an inaccurate measure of overhead on IA-32, where instruction latencies vary significantly. Rather than measuring a real system, this equation was fed into a cache simulator. Furthermore, the simulation was based on the code cache being sized at one-tenth of the maximum cache size, which skews the results due to the difference in benchmarks' working set sizes. In DynamoRIO we have not observed link tracking overhead to be significant, and its advantages are overwhelming.

Hazelwood and Smith [2002] studied cache eviction policies and concluded that a first-in-first-out (FIFO) policy works as well as any other policy in terms of miss rate, including those with profiling such as a least-frequently-used (LFU) or least-recently-used (LRU) policy. The conclusion of later work [Hazelwood and Smith 2004] is that the best scheme is to divide the cache into eight or so units, each flushed in its entirety. Multi-fragment deletion can certainly be cheaper than single-fragment deletion. However, these cache studies do not take into account cache consistency events in real systems, which could drastically change all of their equations by increasing the frequency of evictions, and prevent forward progress when a flush unit contains both an instruction writing code and its target.

Dynamo [Bala et al. 2000] attempted to identify working set changes by pre-emptively flushing its cache when fragment creation rates rose significantly. Other work on identifying application working sets have focused on data references, attempting to improve prefetching and cache locality [Shen et al. 2004, Chilimbi 2001], or on reducing windows of simulation while still capturing whole-program behavior [Sherwood et al. 2002]. Many of these schemes are computation-intensive and require post-processing, making them un-realizable in a runtime system that needs efficient, incremental detection. Some schemes do operate completely at runtime, but require hardware support [Dhodapkar and Smith 2002].

10.2.7 Tool Interfaces

Most runtime tools use an interface modeled after that used by ATOM [Srivastava and Eustace 1994], which provides event hooks to allow a tool to insert a call to an instrumentation routine before or after every procedure, basic block, control transfer between basic blocks, or even every instruction. However, modification of code is usually not supported, or if it is, an entire procedure or block must be replaced by using a trampoline to jump to the new version.

DELI [Desoli et al. 2002] provides an interface that allows full control over the code emitted into the code cache. It primarily targets emulators or just-in-time compilers who want to use DELI as a caching and linking service. Transparency support for tools operating on applications is not provided. Full details on its instruction representation and manipulation interface are not available.

10.2.8 Security

This section discusses the large body of work related to our secure execution environment, *program shepherding* (Section 9.4). Reflecting the prevalence of buffer overflow and format string attacks, there have been several other efforts to provide automatic protection and detection of these vulnerabilities. We summarize the more successful ones.

StackGuard [Cowan et al. 1998] is a compiler patch that modifies function prologues to place *canaries* adjacent to the return address pointer. A stack buffer overflow will modify the canary while overwriting the return pointer, and a check in the function epilogue can detect that condition.

This technique is successful only against sequential overwrites and protects only the return address.

StackGhost [Frantzen and Shuey 2001] is an example of hardware-facilitated return address pointer protection. It is a kernel modification of OpenBSD that uses a Sparc architecture trap when a register window has to be written to or read from the stack, and it performs transparent `xor` operations on the return address before it is written to the stack on function entry and before it is used for control transfer on function exit. Return address corruption results in a transfer unintended by the attacker, and thus attacks can be foiled.

Techniques for stack smashing protection by keeping copies of the actual return addresses in an area inaccessible to the application are also proposed in StackGhost [Frantzen and Shuey 2001] and in the compiler patch StackShield [Vendicator]. Keeping copies of return addresses is also done via binary rewriting [Prasad and Chiueh 2003] and dynamically using our own DynamoRIO client interface [Zovi 2002]. These proposals suffer from various complications in the presence of multi-threading or (as in our SSE2 scheme) deviations from a strict calling convention by `setjmp()` or exceptions. The static methods also have problems with dynamically-loaded or generated code. Furthermore, unless the memory areas are unreadable by the application, there is no hard guarantee that an attack targeted against a given protection scheme can be foiled. On the other hand, if the return stack copy is protected for the duration of a function execution, it has to be unprotected on each call, and that can be prohibitively expensive (`mprotect` on Linux on IA-32 is 60–70 times more expensive than an empty function call). Techniques for write-protection of stack pages [Cowan et al. 1998] have also shown significant performance penalties.

FormatGuard [Cowan et al. 2001] is a library patch for dynamic checks of format specifiers to detect format string vulnerabilities in programs that directly use the standard `printf` functions.

Enforcing non-executable permissions on IA-32 via operating system kernel patches has been done for stack pages [Designer] and for data pages in PaX [PaX Team]. Our system provides execution protection from user mode and achieves better steady state performance. Randomized placement of position independent code was also proposed in PaX as a technique for protection against attacks using existing code; however, it is open to attacks that are able to read process addresses and thus determine the program layout.

Type safety of C code has been proposed by the CCured system [Necula et al. 2002], which extends the C type system, infers statically verifiable type-safe pointers, and adds runtime checks only for unsafe pointers. Cyclone [Jim et al. 2002] provides a safe dialect of C in a similar fashion, but requires annotations in conversion of legacy code. The reported overhead of these systems is in the 30%–300% range.

Other programming bugs stemming from violations of specific higher-level semantic rules of safe programming have been targeted by static analyses like CQUAL [Foster et al. 2002], ESP [Das et al. 2002], MC [Hallem et al. 2002], and static model checkers SLAM [Thomas Ball 2002] and MOPS [Chen and Wagner 2002]. In an unsafe language like C, techniques that claim to be sound do not hold in the presence of violations of the memory and execution model assumed in the analyses [Thomas Ball 2002]. Our system may be used to complement these and enforce the execution model of the application.

Static analyses have been applied for detection of common buffer overflow [Wagner et al. 2000] and format string [Shankar et al. 2001] vulnerabilities, with relatively low false positive rates. Static analysis information can be used to augment program shepherding [Kiriansky 2003], in a manner similar to the hybrid approach of using static analysis and runtime model checking in Wagner and Dean [2001], combining static analysis to construct a model of the system calls possibly generated by a program and a runtime component to verify that. The system call model is generated from an assumed valid execution model — context-insensitive representation as a call graph, or context-sensitive with stack enforcement. Our system is as at least as accurate in detection of disallowed system call sequences, since it disallows deviations from the chosen execution model. Therefore our techniques subsume the need to further model and dynamically check system calls, and we present a practical system with minimal overhead. The *mimicry attacks* introduced [Wagner and Dean 2001] and further analyzed by Wagner and Soto [2002] show how attackers can easily evade intrusion detection at the system call level. We have also outlined [Kiriansky et al. 2002] a simple mimicry attack violating information flow [Heintze and Riecke 1998].

Software fault isolation [Wahbe et al. 1993] modifies a program binary to restrict the address range of memory operations. Execution monitors [Schneider 2000] were applied in SASI [Erlingsson and Schneider 1999] to enforce a memory model via static code instrumentation. MiSFIT [Small 1997] is a tool for making end-user software extensions in C++ safe by using software fault isolation on IA-32.

Several other runtime approaches have been developed recently. Strata [Scott and Davidson

2002] uses dynamic translation with worse performance to enforce a subset of our techniques. Detecting invalid code origins can be performed via hardware-supported information flow tracking [Suh et al. 2004].

10.3 Chapter Summary

DynamoRIO builds on the code caching technology used in many previous systems. This chapter shows the wide variety of systems that use code caching, with varying goals, and compares specific code caching techniques used in other systems to those needed to achieve DynamoRIO's goals. As modern applications continue to become more dynamic, code caching will become even more pervasive. Runtime code manipulation is being used in more and more research and commercial projects, for its ability to efficiently and comprehensively operate on today's applications.

Chapter 11

Conclusions and Future Work

This concluding chapter discusses what we accomplished as well as future work.

11.1 Discussion

We achieved our goals of making DynamoRIO deployable, efficient, transparent, comprehensive, practical, universal, and customizable. It took a tremendous amount of effort to build the system, with many design decisions along the way and each misstep spoiling the whole system. Though each decision required novel research and impacted the rest of the system, in order to build the end result we did not have time for exhaustive exploration of every design point.

The goal of universality, targeting large, complex, commercial applications, was the most difficult to meet. There is a huge difference between correctly executing a single-threaded, static benchmark and correctly executing a large multi-threaded desktop or server application. The operating system interactions and transparency issues were more challenging than the efficiency problems (although there is still room for improvement in reducing overheads), and we hope that others will benefit from the lessons we learned in scaling DynamoRIO up to target modern applications.

The biggest limitation of DynamoRIO is memory usage, which restricts deployment scalability. Executing any single application is not a problem, but if one wished to run every process on a machine under DynamoRIO, the additional memory required could be problematic. We discuss ideas for reducing memory usage in the next section.

One of the most non-intuitive aspects of DynamoRIO is that the cost of building the code cache does not dominate performance. In fact, the fragment creation cost is typically less that one percent of total execution time. Modern architectures, with multi-level caches, will grind to a halt if there is not massive code reuse. The cost of executing a piece of code for the first time is already high natively (it may be on a new page that needs to be brought in from disk, etc.), so our one-time added cost of building a fragment for new code has less relative impact, and is amortized across future executions of the same code.

Our main performance problem is in not conforming to expected application patterns for which modern processors are heavily optimized. An application under our control has different branch behavior, particularly with respect to indirect branches, and thus cannot take advantage of every hardware optimization. If systems like ours become common, hardware vendors will optimize for our type of pattern, enabling better performance for runtime code manipulation systems. In Section 11.2.5 we discuss several specific hardware modifications that would facilitate runtime code manipulation.

Although some of the challenges we faced are not present on other architectures or operating systems, IA-32 and Windows are ubiquitous. Replacing them is by far the bigger challenge than targeting them.

11.2 Future Work

This section discusses salient issues for future work, as well as tools we would like to see built using our interface.

11.2.1 Memory Reduction

While desktop applications have little sharing among threads, server applications can have hundreds of threads all performing similar tasks. Thread-private caches are still the best choice for executing applications with a reasonable memory footprint. However, memory resources can be stretched thin when running multiple highly-threaded servers simultaneously. Our hybrid shared and private cache study (Section 6.5) reduced memory usage significantly while maintaining performance, but all fragments must be shared to eliminate the scalability problem entirely. Although most of the design decisions in this thesis apply to both thread-shared and thread-private caches, our cache capacity solutions are tailored to thread-private caches. Developing efficient cache capacity schemes for shared caches, as well as making shared indirect branch table modifications efficient, are required for fully functional shared caches.

When executing multiple processes on the same machine, DynamoRIO privatizes shared library code into each process' code cache. This can be good for process-specific trace optimization, but if memory is in short supply, a mechanism for a process-shared code cache could be employed. This would require inter-process synchronization and would entail costly cache management operations, but such a cache should be fairly stable and its upkeep costs should amortize well.

Our adaptive working set cache sizing algorithm could be extended to proactively shrink the code cache, reclaiming memory during idle periods. Along the same lines, generational traces that add more layers of profiling than the current two-stage basic block-trace arrangement may result in identification of a tighter working set, leading to smaller code caches and better performance. Generational caches could be extended to persistent caches, with important traces loaded in at program start time to avoid beginning with a cold cache.

11.2.2 Client Interface Extensions

Our interface could be extended to allow customization of cache policies, addition of operations to context switches, and control over other internal aspects of the system. This would allow creation of a wider array of tools. For example, it was more efficient to implement our program shepherding scheme (see Section 9.4) by modifying DynamoRIO directly than as a tool built from our current interface.

11.2.3 Sideline Operation

When operating on multi-processor machines, idle processors can be used for *sideline* optimization, re-optimization, and even garbage collection of the code cache, using out-of-line cycles to reduce overhead. A runtime system must pay the cost for every action it takes, and the benefits of each action must outweigh the time to perform it. Sideline operations can allow more costly and potentially risky actions with significant potential benefits to be undertaken at runtime.

11.2.4 Dynamic Optimization

Originally, we built DynamoRIO to investigate novel dynamic optimizations, including automatic parallelization [Bruening et al. 2000], automatic SIMD vectorization, value profiling for code specialization, sideline optimization, and prefetching. While Dynamo [Bala et al. 2000] achieved success with optimizations for PA-RISC, most of our optimizations failed to yield the anticipated results. This is due to the challenges in optimizing CISC code with few registers and frequent memory operations running on processors with hardware enhancements like trace caches and return address predictors. One approach is to target IA-32 applications running on IA-64 systems [Garnett 2003], where memory alias hardware may alleviate some of the difficulties we faced in targeting IA-32 processors.

Building better dynamic optimizations on IA-32 remains future work. A promising area of study is communication with the compiler. Especially on a CISC platform, compiler hints about memory aliasing could open up many opportunities for successful optimizations. The communication can be two-way, with profile-directed optimization feeding the contents of the code cache back to the compiler for offline optimization, seeding the cache with pre-optimized traces for future runs. Another idea is to monitor hardware performance counters dynamically to zero in on pieces of the application that are performing poorly. For example, identifying code that has bad cache behavior and inserting prefetching.

11.2.5 Hardware Support

While DynamoRIO successfully operates on commodity hardware, there are several hardware modifications that would provide significant benefits. For example, an interface for control of the processor's trace cache could result in better cooperation between the software and hardware code caches.

Another target area is indirect branch prediction. The return address predictor should be exposed to software. As Section 4.2 explains, our translation of return instructions to indirect jumps causes a significant performance reduction. If we could control the hardware's return stack buffer we could regain native performance on return instructions.

Hardware support would also make error handling and context translation simpler, in particular the support of precise interrupts (see Section 3.3.5). Several related systems have hardware support for rolling back state at an exception [Ebcioglu and Altman 1997, Klaiber 2000]. Another useful feature would be condition-code-free conditional branches (see Section 4.4 for a discussion of problems with IA-32 condition codes).

A final desired feature is expansion of the hardware performance counter features to efficiently monitor many of them at once while an application is executing in order to perform targeted dynamic optimizations, as mentioned in Section 11.2.4.

11.2.6 Tools

We envision DynamoRIO enabling numerous new runtime tools. The ability to easily monitor control flow facilitates the implementation of tools for tracing, logging, path coverage, and deterministic replay of execution paths. Another important benefit to operating at runtime is adaptability: code transformations need not be permanent. Instrumentation can be inserted for the duration of an execution period of interest and later removed, reducing profiling and analysis overhead.

An interesting advantage of our thread-private caches is thread-private tool operations. Thread-specific breakpoints, profiling, and instrumentation are all easily and efficiently implemented with DynamoRIO. Without thread-private code, a dispatch on the current thread must be performed every time the code in question is executed.

Modern debuggers rely on special interfaces exposed by the operating system to control the target process. DynamoRIO introduces the possibility of an in-process debugger that does not rely on any operating system support. Being in the same process, such a debugger's operations would be more efficient and more flexible.

11.3 Summary

Runtime code manipulation has been used for decades, but always in specialized ways. There is a growing need to shift program analysis and modification tools from offline to online, which requires a general-purpose runtime infrastructure. This thesis conclusively shows that this is feasible in software without hardware support. We present *DynamoRIO*, a system for runtime code manipulation that is efficient, transparent, and comprehensive, able to observe and manipulate every executed instruction in an unmodified application running on a stock operating system and commodity hardware. DynamoRIO handles large, complex, modern applications with dynamically-loaded, generated, or even modified code, and exports an interface for customization and extensibility, acting as a platform for building tools that manipulate programs while they are running. It has a wide variety of significant potential uses: program analysis and understanding, profiling, instrumentation, optimization, dynamic code decompression, code streaming, translation, even security.

Compiler infrastructures were influential in advancing compiler research. Before such infrastructures were available, a researcher had to be content with toy program analyses and optimizations, or else had to modify a complex system like `gcc` in order to run real programs. But once infrastructures like SUIF [Wilson et al. 1994] and Trimaran [Trimaran] were released, researchers could build novel program transformations that targeted real applications. Our hope is to fill that role in the dynamic world. We hope to promote collaboration between researchers through a common platform and accelerate the development of other research.

We have made the DynamoRIO system available to the public in binary form [MIT and Hewlett-Packard 2002], to be used with custom-built clients via the interface described in Chapter 8. We are excited that others have begun using the system for novel research [Zovi 2002, Hazelwood and Smith 2003]. We believe that systems like ours will be ubiquitous and essential components of future computer systems. We have conclusively shown that such systems are technically feasible. In fact, DynamoRIO's security application from Section 9.4 is currently being commercialized.

Bibliography

1. Adl-Tabatabai, A., Cierniak, M., Lueh, G., Parikh, V. M., and Stichnoth, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, 280–290.

2. Advanced Micro Devices, Inc., 1998. SYSCALL and SYSRET Instruction Specification, May. Publication 21086.

3. Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, 47–65.

4. Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent dynamic optimization: The design and implementation of Dynamo. Tech. Rep. HPL-1999-78, HP Laboratories, June.

5. Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, 1–12.

6. Ball, T., and Larus, J. R. 1996. Efficient path profiling. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO '96)*, 46–57.

7. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauery, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP '03)*, 164–177.

8. Baron, I. 2003. *Dynamic Optimization of Interpreters using DynamoRIO*. Master's thesis, M.I.T.

9. Berger, E. D., Zorn, B. G., and McKinley, K. S. 2002. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, 1–12.

10. Berndl, M., and Hendren, L. 2003. Dynamic profiling and trace cache generation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, 276–285.

11. Bovet, D. P., and Cesati, M. 2002. *Understanding the Linux Kernel*, 2nd ed. O'Reilly & Associates, Sebastopol, CA.

12. Brian N. Bershad, M. J. Z., and Sawdon, W. A. 1993. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93)*, 528–537.

13. Bruening, D., and Duesterwald, E. 2000. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*.

14. Bruening, D., Devabhaktuni, S., and Amarasinghe, S. 2000. Softspec: Software-based speculative parallelism. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*.

15. Bruening, D., Duesterwald, E., and Amarasinghe, S. 2001. Design and implementation of a dynamic optimization framework for Windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 19–30.

16. Bruening, D., Garnett, T., and Amarasinghe, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, 265–275.

17. Buck, B. R., and Hollingsworth, J. 2000. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4) (Winter), 317–329.

18. Bugnion, E., Devine, S., and Rosenblum, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, 143–156.

19. Carmean, D., Upton, M., Hinton, G., Sager, D., Boggs, D., and Roussel, P. 2001. The Pentium 4 processor. In *Proceedings of Hot Chips 13*.

20. Carter, J., Bennett, J., and Zwaenepoel, W. 1991. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP '91)*, 152–164.

21. Chen, H., and Wagner, D. 2002. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the ACM Conference on Computer And Communications Security (CCS 2002)*, 235–244.

22. Chen, W., Lerner, S., Chaiken, R., and Gillies, D. M. 2000. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 81–90.

23. Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli,

S. B., AND YATES, J. 1998. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2) (Mar.), 56–64.

24. CHILIMBI, T. M. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, 191–202.

25. CIFUENTES, C., AND EMMERIK, M. V. 2000. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3) (Mar.), 60–66.

26. CIFUENTES, C., LEWIS, B., AND UNG, D. 2002. Walkabout — a retargetable dynamic binary translation framework. In *Proceedings of the 4th Workshop on Binary Translation*.

27. CMELIK, R. F., AND KEPPEL, D. 1993. Shade: A fast instruction-set simulator for execution profiling. Tech. Rep. UWCSE 93-06-06, University of Washington, June.

28. CMELIK, R. F., AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1) (May), 128–137.

29. COHN, R., AND LOWNEY, P. G. 1996. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO '96)*, 80–89.

30. COHN, R., GOODWIN, D., LOWNEY, P. G., AND RUBIN, N. 1997. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, 17–24.

31. CONNECTIX. Virtual PC.
`http://www.microsoft.com/windows/virtualpc/default.mspx`.

32. CONSEL, C., AND NÖEL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, 145–156.

33. CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, Cambridge, MA.

34. COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 63–78.

35. COWAN, C., BARRINGER, M., BEATTIE, S., AND KROAH-HARTMAN, G. 2001. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 191–199.

36. DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI '02), 57–68.

37. DEAVER, D., GORTON, R., AND RUBIN, N. 1999. Wiggins/Redstone: An on-line program specializer. In *Proceedings of Hot Chips 11*.

38. DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. 2003. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, 15–24.

39. DESIGNER, S. Non-executable user stack. `http://www.openwall.com/linux/`.

40. DESOLI, G., MATEEV, N., DUESTERWALD, E., FARABOSCHI, P., AND FISHER, J. A. 2002. DELI: A new run-time control point. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO '02)*, 257–268.

41. DEUTSCH, L. P., AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '84)*, 297–302.

42. DHODAPKAR, A. S., AND SMITH, J. E. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA '02)*, 233–244.

43. DIETZFELBINGER, M., KARLIN, A., MELHORN, K., HEIDE, F. M. A. D., ROHNERT, H., AND TARJAN, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing*, 23(4) (Aug.), 738–761.

44. DOMANI, T., GOLDSHTEIN, G., KOLODNER, E. K., AND LEWIS, E. 2002. Thread-local heaps for java. In *Proceedings of the International Symposium on Memory Management (ISMM '02)*, 183–194.

45. DREPPER, U., AND MOLNAR, I. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

46. DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA '86)*, 434–442.

47. DUESTERWALD, E., AND BALA, V. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, 202–211.

48. EBCIOGLU, K., AND ALTMAN, E. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, 26–37.

49. ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. ‘C: A language for efficient, machine-independent dynamic code generation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’96)*, 131–144.

50. ENGLER, D. 1996. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’96)*, 160–170.

51. ERLINGSSON, U., AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, 87–95.

52. FAHS, B., BOSE, S., CRUM, M., SLECHTA, B., SPADINI, F., TUNG, T., PATEL, S. J., AND LUMETTA, S. S. 2001. Performance characterization of a hardware framework for dynamic optimization. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO ’01)*, 16–27.

53. FEIGIN, E., 1999. *A Case for Automatic Run-Time Code Optimization*. Senior thesis, Harvard College, Division of Engineering and Applied Sciences, Apr. `http://www.eecs.harvard.edu/hube/publications/feigin-thesis.pdf`.

54. FOSTER, J., TERAUCHI, T., AND AIKEN, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’02)*, 1–12.

55. FRANTZEN, M., AND SHUEY, M. 2001. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, 55–66.

56. FREDMAN, M. L., KOML’OS, J., AND SZEMER’EDI, E. 1984. Storing a sparse table with o(1) worst case access time. *Journal of the Association for Computing Machinery*, 31(3), 538–544.

57. FREE SOFTWARE FOUNDATION. GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.

58. GARNETT, T. 2003. *Dynamic Optimization of IA-32 Applications Under DynamoRIO*. Master’s thesis, M.I.T.

59. GDB. The GNU Project Debugger. `http://www.gnu.org/software/gdb/gdb.html`.

60. GEODESIC SYSTEMS, 2001. InCert traceback for Windows performance test report, Apr. `http://www.geodesic.com/news/pdf/TesComPerformanceTest.pdf`.

61. GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1990. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA ’90)*, 15–25.

62. GNU C LIBRARY. `http://www.gnu.org/software/libc/libc.html`.

63. GNU Compiler Connection Internals. Trampolines for Nested Functions. http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html.

64. Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. 1996. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1–13.

65. Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. 1999. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, 293–304.

66. Hallem, S., Chelf, B., Xie, Y., and Engler, D. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, 69–82.

67. Hazelwood, K., and Smith, M. D. 2002. Code cache management schemes for dynamic optimizers. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architecture (Interact-6)*, 102–110.

68. Hazelwood, K., and Smith, M. D. 2003. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO '03)*, 169–179.

69. Hazelwood, K., and Smith, J. E. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, 89–99.

70. Heintze, N., and Riecke, J. G. 1998. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, 365–377.

71. Henis, E. A., Haber, G., Klausner, M., and Warshavsky, A. 1999. Feedback based post-link optimization for large subsystems. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*.

72. Hollingsworth, J. K., Miller, B. P., and Cargille, J. M. 1994. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High-Performance Computing Conference (SHPCC '94)*, 841–850.

73. Hölzle, U. 1994. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University.

74. Hunt, G., and Brubacher, D. 1999. Detours: Binary interception of win32 functions. In *Proceedings of the USENIX Windows NT Workshop*, 135–144.

75. Insight Software Solutions, Inc. MacroExpress. http://www.macros.com/.

76. INTEL CORPORATION. 1999. *Intel Architecture Optimization Reference Manual*. Order Number 245127-001.

77. INTEL CORPORATION. 2001. *IA-32 Intel Architecture Software Developer's Manual*, vol. 1–3. Order Number 245470, 245471, 245472.

78. INTEL CORPORATION, 2003. Pin — A Binary Instrumentation Tool, Nov. `http://rogue.colorado.edu/Pin/`.

79. INTEL VTUNE PERFORMANCE ANALYZER. `http://www.intel.com/software/products/vtune/`.

80. JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 275–288.

81. JONES, N. D., GOMRADE, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

82. JONES, N. D. 1996. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3) (Sept.), 480–503.

83. KAFFE.ORG. The Kaffe Java virtual machine. `http://www.kaffe.org/`.

84. KEPPEL, D. 1991. A portable interface for on-the-fly instruction space modification. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, 86–95.

85. KIM, H., AND SMITH, J. E. 2003a. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, 25–35.

86. KIM, H., AND SMITH, J. E. 2003b. Hardware support for control transfers in code caches. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO '03)*, 253–264.

87. KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 191–206.

88. KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2003. Execution model enforcement via program shepherding. Tech. Rep. LCS-TM-638, M.I.T., May.

89. KIRIANSKY, V. 2003. *Secure Execution Environment via Program Shepherding*. Master's thesis, M.I.T.

90. KISTLER, T., AND FRANZ, M. 2001. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6) (June).

91. KLAIBER, A., 2000. The technology behind Crusoe processors. Transmeta Corporation, Jan. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

92. KNUTH, D. 1998. *The Art of Computer Programming*, 2nd ed., vol. 3 (Sorting and Searching). Addison-Wesley, Reading, MA.

93. KO, C., FRASER, T., BADGER, L., AND KILPATRICK, D. 2000. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, 145–156.

94. KRINTZ, C., GROVE, D., SARKAR, V., AND CALDER, B. 2001. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8) (Mar.).

95. KUMAR, N., MISURDA, J., CHILDERS, B. R., AND SOFFA, M. L. 2003. FIST: A framework for instrumentation in software dynamic translators. Tech. Rep. TR-03-106, University of Pittsburgh, Sept.

96. KUMAR, A. 1996. The HP PA-8000 RISC CPU: A high-performance out-of-order processor. In *Proceedings of Hot Chips VIII*.

97. LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9) (Sept.), 241–248.

98. LARUS, J., AND SCHNARR, E. 1995. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, 291–300.

99. LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, 1–11.

100. LEE, P., AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 137–148.

101. LEE, D. C., CROWLEY, P. J., BAER, J., ANDERSON, T. E., AND BERSHAD, B. N. 1998. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA '98)*, 27–38.

102. LEGER, C. 2004. *An API for Dynamic Partial Evaluation under DynamoRIO*. Master's thesis, M.I.T.

103. LEONE, M., AND DYBVIG, R. K. 1997. Dynamo: A staged compiler architecture for dynamic program optimization. Tech. Rep. 490, Department of Computer Science, Indiana University, Sept.

104. LEROY, X. The LinuxThreads library.

http://pauillac.inria.fr/~xleroy/linuxthreads/.

105. Leroy, X., 2003. The Objective Caml system release 3.06, Sept. http://pauillac.inria.fr/ocaml.

106. Levine, J. R. 1999. *Linkers and Loaders*. Morgan-Kaufman, San Francisco, CA.

107. Li, K., and Hudak, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4) (Nov.), 321–359.

108. Lindholm, T., and Yellin, F. 1999. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley.

109. Lu, J., Chen, H., Yew, P., and Hsu, W. 2004. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, vol. 6 (Apr.), 1–24.

110. Luk, C., Muth, R., Patil, H., Cohn, R., and Lowney, G. 2004. Ispike: A post-link optimizer for the intel itanium architecture. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*.

111. Magnusson, P. S., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Lundholm, F., Moestedt, A., Nilsson, J., Stenström, P., and Werner, B. 1998. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX Annual Technical Conference*, 119–130.

112. McGregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B. 2003. A processor architecture defense against buffer overflow attacks. In *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, 243–250.

113. Merten, M. C., Trick, A. R., George, C. N., Gyllenhaal, J. C., and Hwu, W. W. 1999. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA '99)*, 136–147.

114. Merten, M. C., Trick, A. R., Barnes, R. D., Nystrom, E. M., George, C. N., Gyllenhaal, J. C., and Hwu, W. W. 2001. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6), 567–589.

115. Microsoft Corporation, 1999. Microsoft Portable Executable and Common Object File Format specification, Feb. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

116. Microsoft Corporation. 2001. *Microsoft SQL Server 2000 Resource Kit*. Microsoft Press, Redmond, WA.

117. Microsoft Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

118. MICROSOFT DEVELOPER NETWORK LIBRARY. `http://msdn.microsoft.com/library/`.

119. MICROSOFT VISUAL STUDIO. `http://msdn.microsoft.com/vstudio/`.

120. MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. 1995. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11) (Nov.), 37–46.

121. MIT AND HEWLETT-PACKARD, 2002. DynamoRIO dynamic code modification system binary package release, June. `http://www.cag.lcs.mit.edu/dynamorio/`.

122. MOORE, C. H., AND LEACH, G. C. 1970. Forth – a language for interactive computing. Tech. rep., Mohasco Industries, Inc., Amsterdam, NY.

123. MUTH, R., DEBRAY, S., WATTERSON, S., AND BOSSCHERE, K. D. 2001. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, vol. 31 (Jan.), 67–101.

124. NEBBETT, G. 2000. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN.

125. NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: type-safe retrofitting of legacy code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, 128–139.

126. NETHERCOTE, N., AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verification (RV '03)*.

127. PALMER, T., ZOVI, D. D., AND STEFANOVIC, D. 2001. SIND: A framework for binary translation. Tech. Rep. TR-CS-2001-38, University of New Mexico, Dec.

128. PARASOFT. Insure++.
`http://www.parasoft.com/jsp/products/home.jsp?product=Insure&itemId=65`.

129. PATEL, S., AND LUMETTA, S. 1999. rePLay : a hardware framework for dynamic program optimization. Tech. Rep. CRHC-99-16, University of Illinois, Dec.

130. PATEL, S. J., TUNG, T., BOSE, S., AND CRUM, M. M. 2000. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO '00)*, 303–313.

131. PAX TEAM. Non executable data pages. `http://pageexec.virtualave.net/docs/`.

132. PIETREK, M. 1996. Under the hood. *Microsoft Systems Journal*, 11(5) (May).

133. PIETREK, M. 1997. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, 12(1) (Jan.).

134. PIETREK, M. 2002. An in-depth look into the win32 portable executable file format. *MSDN Magazine*, 17(2) (Feb.).

135. PIKE, R., LOCANTHI, B., AND REISER, J. 1985. Hardware/software trade-offs for bitmap graphics on the Blit. *Software - Practice and Experience*, 15(2), 131–151.

136. PIUMARTA, I., AND RICCARDI, F. 1998. Optimizing direct-threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, 291–300.

137. POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*, 109–121.

138. PRASAD, M., AND CHIUEH, T. 2003. A binary rewriting defense against stack-based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, 211–224.

139. PSBENCH@YAHOO.COM. PS6bench Photoshop benchmark (Advanced). `http://www.geocities.com/Paris/Cafe/4363/download.html#ps6bench/`.

140. PU, C., MARSALIN, H., AND IOANNIDES, J. 1988. The Synthesis kernel. *Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Systems*, 1(1) (Winter).

141. RICHTER, J. 1999. *Programming Applications for Microsoft Windows*, 4th ed. Microsoft Press, Redmond, WA.

142. ROBINSON, A., 2001. Why dynamic translation? Transitive Technologies Ltd., May. `http://www.transitive.com/documents/Why_Dynamic_Translation1.pdf`.

143. ROMER, T., VOELKER, G., LEE, D., WOLMAN, A., WONG, W., LEVY, H., AND BERSHAD, B. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, 1–7.

144. ROSENBLUM, M., HERROD, S., WITCHEL, E., AND GUPTA, A. 1995. The SimOS approach. *IEEE Parallel and Distributed Technology*, 4(3), 34–43.

145. ROTENBERG, E., BENNETT, S., AND SMITH, J. E. 1996. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO '96)*, 24–35.

146. SCHNEIDER, F. B. 2000. Enforceable security policies. *Information and System Security*, 3(1), 30–50.

147. SCOTT, K., AND DAVIDSON, J. 2001. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*.

148. SCOTT, K., AND DAVIDSON, J. 2002. Safe Virtual Execution using software dynamic translation.

In *Proceedings of the 2002 Computer Security Application Conference*.

149. SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J., AND SOFFA, M. L. 2003. Reconfigurable and retargetable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, 36–47.

150. SEWARD, J., 2002. The design and implementation of Valgrind, Mar. `http://valgrind.kde.org/`.

151. SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. 2001. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 201–220.

152. SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality phase prediction. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*.

153. SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, 45–57.

154. SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. 1992. Binary translation. *Digital Technical Journal*, 4(4).

155. SMALL, C. 1997. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies*.

156. SOLOMON, D. A., AND RUSSINOVICH, M. 2000. *Inside Microsoft Windows 2000*. Microsoft Press, Redmond, WA.

157. SRIVASTAVA, A., AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, 196–205.

158. SRIVASTAVA, A., AND WALL, D. W. 1992. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1) (December), 1–18.

159. SRIVASTAVA, A., EDWARDS, A., AND VO, H. 2001. Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research, Apr.

160. STANDARD PERFORMANCE EVALUATION CORPORATION, 1999. SPEC web99 benchmark. `http://www.specbench.org/osg/web99/`.

161. STANDARD PERFORMANCE EVALUATION CORPORATION, 2000. SPEC CPU2000 benchmark suite. `http://www.spec.org/osg/cpu2000/`.

162. SUH, G. E., LEE, J., AND DEVADAS, S. 2004. Secure program execution via dynamic informa-

tion flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*.

163. SULLIVAN, G., BRUENING, D., BARON, I., GARNETT, T., AND AMARASINGHE, S. 2003. Dynamic native optimization of interpreters. In *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators (IVME '03)*, 50–57.

164. SUN MICROSYSTEMS. The Java HotSpot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

165. TAMCHES, A., AND MILLER, B. P. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, 117–130.

166. TAYLOR, S. A., QUINN, M., BROWN, D., DOHM, N., HILDEBRANDT, S., HUGGINS, J., AND RAMEY, C. 1998. Functional verification of a multiple-issue, out-of-order, superscalar alpha processor - the DEC alpha 21264 microprocessor. In *Proceedings of the Design Automation Conference*, 638–643.

167. THOMAS BALL, S. K. R. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, 1–3.

168. TOOL INTERFACE STANDARDS COMMITTEE, 1995. Executable and Linking Format (ELF), May.

169. TRIMARAN. Trimaran infrastructure for research in instruction-level parallelism. `http://www.trimaran.org/`.

170. UNG, D., AND CIFUENTES, C. 2000. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 41–51.

171. UNIX PRESS, C. 1993. *System V Application Binary Interface*, 3rd ed. Prentice-Hall, Inc.

172. VENDICATOR. Stackshield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

173. VERITEST. Business Winstone and Content Creation Winstone benchmark suites. `http://www.etestinglabs.com/benchmarks/`.

174. VERITEST, 2002. WebBench 5.0 web server benchmark. `http://www.etestinglabs.com/benchmarks/webbench/`.

175. VOSS, M., AND EIGENMANN, R. 2000. A framework for remote dynamic program optimization. In *Proceedings of the ACM Workshop on Dynamic Optimization (Dynamo '00)*, 32–40.

176. WAGNER, D., AND DEAN, D. 2001. Intrusion detection via static analysis. In *Proceedings of the*

IEEE Symposium on Security and Privacy, 156–169.

177. WAGNER, D., AND SOTO, P. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM Conference on Computer And Communications Security (CCS 2002)*, 255–264.

178. WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 3–17.

179. WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5) (December), 203–216.

180. WALDSPURGER, C. A. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 181–194.

181. WHITAKER, A., SHAW, M., AND GRIBBLE, S. 2002. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, 195–209.

182. WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S., TSENG, C., HALL, M., LAM, M., AND HENNESSY, J. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12) (Dec.), 31–37.

183. WINE. Windows compatability layer for X and UNIX. `http://www.winehq.com/`.

184. WITCHEL, E., AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 68–79.

185. ZHANG, C. X., WANG, Z., GLOY, N. C., CHEN, J. B., AND SMITH, M. D. 1997. System support for automated profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, 15–26.

186. ZHENG, C., AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3) (Mar.), 47–53.

187. ZOVI, D. D., 2002. *Security Applications of Dynamic Binary Translation*. Bachelor of Science Thesis, University of New Mexico, Dec.