Maintaining Consistency and Bounding Capacity of Software Code Caches

Derek Bruening and Saman Amarasinghe

MIT Computer Science and Artificial Intelligence Laboratory
and Determina Corporation
{iye,saman}@csail.mit.edu

Abstract

Software code caches are becoming ubiquitous, in dynamic optimizers, runtime tool platforms, dynamic translators, fast simulators and emulators, and dynamic compilers. Caching frequently executed fragments of code provides significant performance boosts, reducing the overhead of translation and emulation and meeting or exceeding native performance in dynamic optimizers. One disadvantage of caching, memory expansion, can sometimes be ignored when executing a single application. However, as optimizers and translators are applied more and more in production systems, the memory expansion from running multiple applications simultaneously becomes problematic. A second drawback to caching is the added requirement of maintaining consistency between the code cache and the original code. On architectures like IA-32 that do not require explicit application actions when modifying code, detecting code changes is challenging. Again, consistency can be ignored for certain sets of applications, but as caching systems scale up to executing large, modern, complex programs, consistency becomes critical. This paper presents efficient schemes for keeping a software code cache consistent and for dynamically bounding code cache size to match the current working set of the application. These schemes are evaluated in the DynamoRIO runtime code manipulation system, and operate on stock hardware in the presence of multiple threads and dynamic behavior, including dynamically-loaded, generated, and even modified code.

1 Introduction

Software code caching began as a technique for reducing the overhead of emulation and dynamic translation, and has been adapted for runtime tools and dynamic optimizers, where it enables meeting and even exceeding native performance. Unfortunately, the benefits of caching come at the cost of cache consistency and memory expansion. For running a single, static application, such as a SPEC CPU2000 [35] benchmark, neither of these issues is a concern, as application code is fixed at load time and has a small footprint (relative to data size). However, as code caching systems become more prevalent and are scaled up to large, modern, complex programs, consistency becomes critical; and, as optimizers and translators are applied more and more in production systems, the memory expansion from running multiple applications simultaneously becomes problematic.

Executing application code from a cache requires maintaining consistency between the cache and the original code. Previous systems have handled consistency by monitoring requests to flush the hardware instruction cache. On architectures like IA-32 that do not require explicit application actions when modifying code, detecting code changes is much more challenging. While consistency can be ignored for certain sets of applications, modern programs with plugins and dynamically-generated code require cache consistency for correct execution, as consistency events are not limited to self-modifying code and include common actions like library unloading, rebasing, and rebinding, and re-use of memory regions for generated code. Enforcing consistency in software is expensive, both in detecting changes and synchronizing code cache views among multiple threads. Existing systems that support consistency on IA-32 have custom hardware and execute underneath the operating system, avoiding interactions with threads. One contribution of this paper is a scheme for efficiently maintaining consistency in the presence of multiple threads, called *non-precise flushing*, that requires no hardware support.

Nearly all code caching systems size the code cache with a generous static bound and assume it will rarely be reached, and when it is, the cache is typically flushed in its entirety. Prior work on managing code caches has focused on eviction policies and not on how to bound the cache size in the first place. Hard limits that work for small benchmark suites do not adapt to modern applications, whose code sizes vary dramatically and often range in the megabytes. This paper's second contribution is a novel algorithm for dynamically

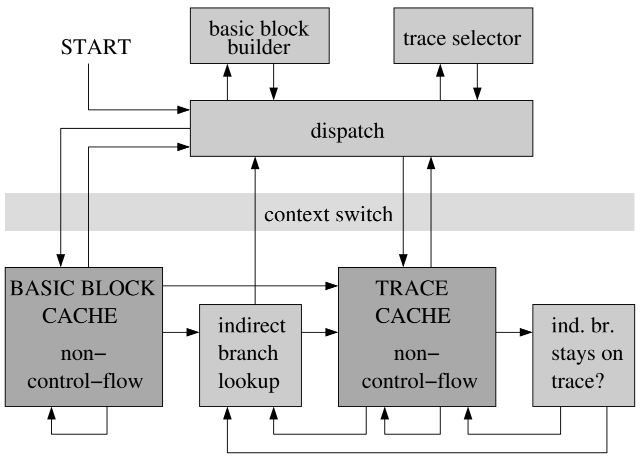

Figure 1. Flow chart of DynamoRIO. A context switch separates the code cache from DynamoRIO code (though it all executes in the same process and address space). Application code is copied into the two caches, with control transfers (shown by arrows in the figure) modified in order to retain control.

Program	Benchmark description	Thrds	Fragmnts	Perf
SPECFP	Floating-point	1	3232	1.02x
SPECINT	Integer	1	8806	1.17x
SPECJVM	Java apps under Sun Java2 JVM 1.4.2.06	10	59323	2.44x
excel	Microsoft Excel 9.0: 2.4MB spreadsheet re-calculations	4	77174	1.11x
photoshp	Adobe Photoshop 6.0: modeled on PS6bench [28] Action	8	171962	1.36x
powerpnt	Microsoft PowerPoint 9.0: modifies 84 slides	5	105054	1.63x
winword	Microsoft Word 9.0: searches & modifies a 1.6MB document	4	98358	1.31x

Table 1. Our benchmark suite, and for each benchmark the simultaneously-live threads, code cache fragments, and slowdown versus native when executed under DynamoRIO.

limiting code cache size to match the current working set of the application.

The cache consistency (Section 3) and capacity (Section 4) algorithms described in this paper are fully implemented and evaluated in a real code caching system, DynamoRIO [3], which is described in the next section.

2 Evaluation Methodology

DynamoRIO is a system for general-purpose runtime code manipulation whose goals are transparency, efficiency, and comprehensiveness [3]. It executes a target application by copying its code, one basic block at a time, into a code cache. Indirecting control through the cache permits custom transformations of the code for any number of runtime tools: instrumentation, profiling, compatibility layers, decompression, security, etc.

Figure 1 shows the components of DynamoRIO and the flow of operations 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 (shown with arrows in the figure), which must be modified to ensure that DynamoRIO retains control. Overhead is amortized by directly *linking* blocks together that are joined by direct branches, and performing fast table lookups to transition between blocks via indirect branches, avoiding the costly context switch back to DynamoRIO code in both cases. Additional performance is gained by *eliding unconditional control transfers* in basic blocks and stitching together frequently executed sequences of basic blocks into *traces*. Traces are kept in their own code cache, and enable inlining of indirect branches to avoid lookup overhead. Both the basic block cache and trace cache are *thread-private* (Section 4.1). We use the term *fragment* to refer to either a basic block or a trace in the code cache.

DynamoRIO is capable of running large, complex, real-world applications on stock IA-32 hardware under both Windows and Linux. This allows us to evaluate our cache management algorithms in realistic scenarios. Our benchmark set consists of the SPEC CPU2000 [35] benchmarks (minus the FORTRAN 90 programs) on Linux and the SPEC JVM98 [34] benchmarks and four large desktop applications on Windows (Table 1). Although the SPEC CPU suite is single-threaded, it is excellent for measuring performance impact on computationally-intensive applications. Our approach is to build cache management algorithms that target large commercial desktop applications while not losing performance on SPEC CPU. Our desktop benchmarks consist of long-running batch computations [3], which are more computationally bound than interactive scenarios that tend to mask slowdowns. We include the SPECJVM benchmarks for intensive tests of generated code. Cache consistency events are so frequent here that code caching perfor-

Benchmark	Memory unmappings	Generated code regions	Code region modifications
SPECFP	112	0	0
SPECINT	29	0	0
SPECJVM	7	3373	4591
excel	144	21	20
photoshp	1168	40	0
powerpnt	367	28	33
winword	345	20	6

Table 2. The number of memory unmappings, generated code regions, and modifications of code regions in our benchmarks.

mance is difficult to achieve. The efficient schemes in this paper allow DynamoRIO to approach a two times slowdown. We use the entire virtual machine as a benchmark and thus include all ten of the Java applications, including the correctness tests.

3 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 de-allocation of memory, e.g., the unmapping of a shared library containing the code, or dynamic modification of the code.

3.1 Memory Unmapping

Unmapping of files is relatively frequent in large Windows applications (Table 2). 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 the simpler of the two consistency problems to solve, as we need only watch for explicit requests to the kernel to unmap files or free address space (the system calls `munmap` and `mremap` on Linux and `NtUnmapViewOfSection`, `NtFreeVirtualMemory`, and `NtFreeUserPhysicalPages` [25] on Windows). When we see such a request, we flush from the cache all fragments that contain pieces of code from the target region (Section 3.6).

3.2 Memory Modification

While true self-modifying code is only seen in a few applications, such as Adobe Premiere and games like Doom, general code modification is surprisingly prevalent. The Windows loader directly modifies code in shared libraries for rebasing [23], and modifies the Import Address Table [27] for rebinding. Since this table is often kept in the first page of the code section, modifications to it look like code modifications if the entire section is treated as one region. Code memory re-use occurs with trampolines used for nested function closures [15], 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. Memory modification also occurs with just-in-time (JIT) compiled code, 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. Table 2 summarizes the frequency of memory region modifications in our benchmarks.

On most architectures, software must issue explicit requests to clear the processor's instruction cache when modifying code [20]. In contrast, IA-32 maintains instruction cache consistency in hardware, making every write to memory a potential code modification. Therefore, we must monitor all memory writes to detect those that affect code, either by sandboxing each write with inserted instrumentation (Section 3.3) or by using hardware page protection. Page protection provides good performance when modifications are infrequent, but forces monitoring to follow page boundaries, leading to cases of *false sharing*.

Our 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 thus may safely be executed. The list is initially populated with the memory regions marked executable *but not writable* when DynamoRIO takes control. Both the Windows and Linux executable formats 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 ignore user memory management through `malloc` and other calls because it is infeasible to identify all internal memory parceling).

When execution reaches a region not on the executable list, the region is added, and (if necessary) 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 (Section 3.6), remove the region from the executable list, mark the region as writable, and then re-execute the faulting write. However, 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.

For error transparency 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. Similarly, if the application changes the protection on a region, we must update our information so that a later fault will be handled properly.

3.3 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, code region re-use, 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 could emulate the write instruction, although at a performance hit due to the complexity of IA-32. Instead, we mark the region as writable and turn to sandboxing. Each fragment from a writable region verifies that its own code is not stale by storing a copy of its source application code. A check is inserted at the top of the fragment comparing the current application code with the stored copy, which must be done one byte at a time. If the copies are different, the fragment is exited and immediately flushed. If the check passes, the body of the fragment may be executed, but each memory write must be monitored 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. 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 [18, vol. 3]. If all applications followed this, it would obviate the need to check for modification after each write.

This strategy will not detect one thread modifying code while another is inside a fragment corresponding to that code — the code modification will not be detected until the next time the target fragment is entered. To mitigate this possibility we terminate fragments at application synchronization operations (Section 3.6). Another problem is that Windows does not support an alternate exception handling stack, forcing us to use sandboxing for any code on the stack, and opening up pathological cases where the stack pointer is later pointed at a writable region [3, p. 151].

Sandboxing has a significant space penalty from its redundant code copies and added write instrumentation. Additionally, sandboxing incurs a significant performance hit: when applied to all fragments, SPEC CPU is an average 14 times slower, ranging from 3 to 47 times slower on individual benchmarks. Although optimization of our instrumentation could improve performance, sandboxing should generally be avoided in favor of page protection.

3.4 Memory Regions

The unit of consistency events, the memory region, must be at least as large as a page in order to utilize page protection. 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, or is modified in a separate phase from execution, but small regions are more flexible for separating a code writer from its target and avoiding false sharing and unnecessary flushing.

DynamoRIO uses an adaptive region granularity to fit regions to the current pattern of code modification. Our 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. Our executable list merges adjacent regions with the same protection privileges, 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.

3.5 Mapping Regions to Fragments

Whatever region sizes we use, we must be able to locate all fragments in the code cache containing code from a particular region. Since our runtime system stitches together basic blocks across unconditional control transfers and builds traces out of frequently executed sequences of basic blocks, any given fragment might contain code from several widely separated regions. DynamoRIO solves this by storing a list of fragments with each executable list region entry. A given fragment may have multiple entries, one for each region from which it contains code.

An alternative mapping strategy is to save space by not storing anything: since the original code is read-only and cannot have changed, each fragment's source code can be

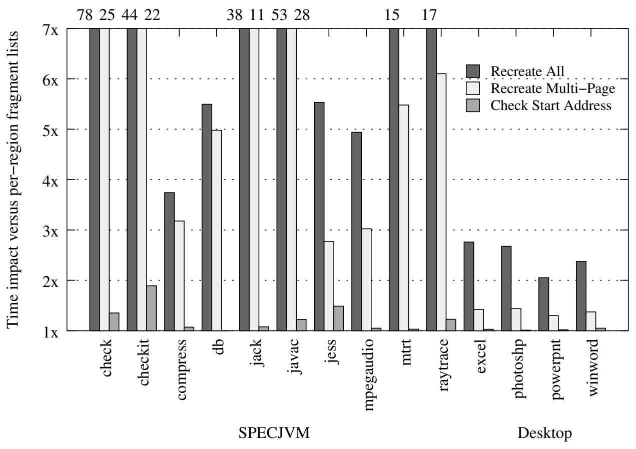

Figure 2. Performance impact of schemes for mapping regions to fragments, versus our chosen solution of per-region fragment lists, on our multithreaded benchmarks.

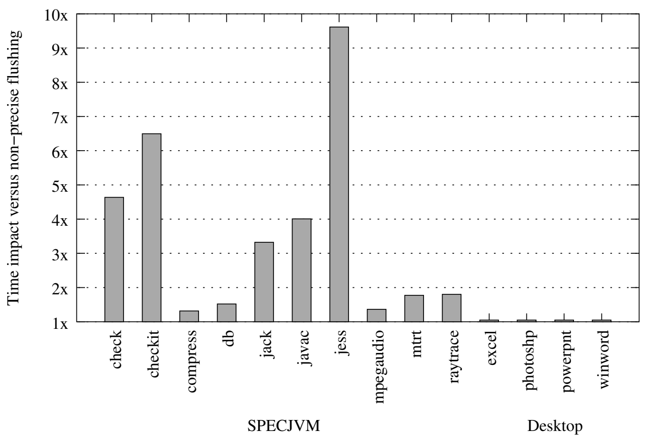

Figure 3. Performance impact of suspending all threads on every cache flush, versus our non-precise flushing, on our multithreaded benchmarks.

examined to determine which regions it touches. The work in re-creating can be reduced by recording, for each basic block built, whether it occupies more than one page, allowing a simple starting address check for all such blocks, limiting recreation to multi-page fragments. However, any recreation has prohibitive performance impact, as shown in Figure 2. Even avoiding all recreation by restricting fragments to never cross page boundaries (allowing a start address check for all fragments) has an average 14% slowdown, because it must consider every single fragment — potentially hundreds of thousands — in order to find the handful that are in the target region.

3.6 Non-Precise Flushing

Even if code caches are thread-private, a memory unmapping or code modification affects all threads' caches, since they share the same address space, and requires synchronization of cache flushing. The actual invalidation of modified code in each thread's code cache must satisfy the memory consistency model in effect. In contrast to software distributed shared memory systems [24], a runtime code caching 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.

To support all applications on IA-32 we must follow sequential consistency [21], which requires immediate invalidation of all affected fragments from the code cache of every thread; otherwise, stale code could be executed. The only mechanism to identify in which fragment a thread is executing in the code cache, short of prohibitively expensive instrumentation, is stopping the thread to examine it. Thus, the only viable flushing approach is brute-force: suspending all threads and forcibly moving those inside of to-be-invalidated code. Threads may frequently be found inside of to-be-deleted regions, as it may have been data that was written to rather than 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 an impractical solution, as shown in Figure 3.

We have developed a relaxation of the consistency model that allows a more efficient invalidation algorithm that we call *non-precise flushing*. Our relaxed consistency model is similar to weak consistency [13] in that it takes advantage of synchronization properties of the application. 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 any indirect branch lookup table(s). Our unlinking operation is a single write made atomic by using the `lock` prefix [18, vol. 3] when the target crosses cache line boundaries. The actual deletion of the fragments can be delayed until a safe point when each thread in question has left the code cache at least once and therefore cannot be inside of a stale fragment.

Non-precise flushing still requires synchronization with each thread, but for the unlinking stage this only needs to ensure that each target thread is not accessing link data structures. Since threads spend most of their time in the code cache, in the common case no synchronization is required. For thread-shared caches, all threads must by synchronized simultaneously before acting on the target fragments to prevent re-linking of invalidated fragments, while thread-private caches can be dealt with one thread at a time. Once the target thread(s) are at a safe point (either in the cache or at a runtime system wait point), the flusher checks whether they have any fragments in the flush region (Section 3.5), 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 exits, it checks the queue and if it is the last thread out performs the actual deletion of the fragments. Once a thread leaves it is free to re-enter, as stale fragments are unreachable once exited.

Non-precise flushing prevents any new execution of stale code, leaving only the problem of handling a thread currently inside of a stale fragment. Since DynamoRIO's fragments only contain loops via self-links, an unlinked invalidated fragment body will be executed at most once. Here we turn to our relaxed consistency model, where we assume that the thread modifying the code is properly synchronized with the thread executing the code. By terminating fragments at application synchronization operations, we ensure that any execution of stale code can only occur if there was a race condition that allowed it natively as well. Our relaxed model ensures sequential consistency when considering data or considering code independently, but weak consistency when considering all of memory. Code writes will never be seen out of order, and data writes are not affected at all — the only potential re-ordering with respect to sequential consistency is between a data write and a code write.

This relaxation matches the limitations of our sandboxing scheme (Section 3.3), 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. 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. We break all of our fragments at such instructions, as well as at loops and system calls, which are also frequent components of synchronization primitives. The performance impact of these fragment barriers is negligible on our benchmarks.

However, a condition variable can be used without explicit synchronization using implicitly atomic single-word single-cache-line reads and writes, and we cannot afford to break fragments on every memory access on the chance that it might be a condition variable. Thus, theoretically, a pathological case that breaks our consistency algorithm can be constructed if a code path that reads a condition variable prior to jumping to modifiable code is executed frequently enough to be completely inlined into a single trace. It is extremely unlikely that synchronization code employed prior to entering dynamically generated code will occupy the same memory region as the generated code itself, so breaking traces at transitions between compiled code modules and generated code regions further narrows this already tiny window in which stale code can be executed. We have not encountered any instances of this violation of our consistency algorithm.

4 Code Cache Capacity

When executing a single application in isolation, there may be no reason to limit the code cache size. However, when executing many programs under a code caching system simultaneously, or with significant cache fragmentation from consistency invalidations, memory usage can become problematic, causing thrashing and performance degradation. We can reduce memory usage significantly by imposing a bound on the code cache size, but such bounds come with their own performance cost in capacity misses. To achieve 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 adaptively sized for each application, and re-sized as an application moves through different phases.

4.1 Thread-Private Versus Shared

A significant design decision affecting cache bounding is how caches are shared among threads. The code cache corresponds to application code, which is shared by every thread in the address space. A runtime system must choose whether its cache will be similarly shared, or whether each thread will have its own private code cache. 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 (otherwise a register must be stolen). 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 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

Benchmark		Thrds	Basic blocks		Traces	
			shared	$\frac{\text{thrds}}{\text{frag}}$	shared	$\frac{\text{thrds}}{\text{frag}}$
SPECJVM		10	18.0%	3.0	9.5%	2.2
batch	excel	4	0.8%	2.9	0.2%	2.1
	photoshp	10	1.2%	4.3	1.0%	3.8
	powerpnt	5	1.0%	2.8	0.1%	2.2
	winword	4	0.9%	2.8	0.1%	2.1
interactive	excel	10	8.4%	3.5	2.5%	3.2
	photoshp	17	4.5%	2.3	1.7%	2.1
	powerpnt	7	8.0%	2.2	10.9%	2.0
	winword	7	9.8%	4.1	3.3%	2.9

Table 3. Fragment sharing across threads. Our desktop batch scenarios are compared to interactive use, which creates more threads. `Thrds` **is the number of threads ever created;** `shared` **is the percentage of fragments executed by more than one thread; and** $\frac{\texttt{thrds}}{\texttt{frag}}$ **is the average number of threads executing each shared fragment.**

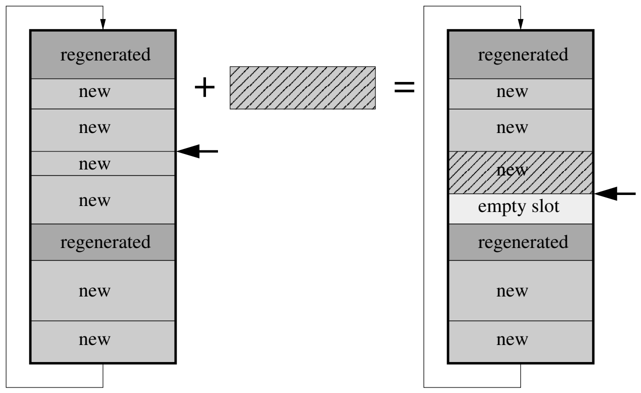

Figure 4. Our FIFO fragment eviction policy, which treats the cache as a circular buffer. A new fragment displaces enough old fragments at the current head to make room. The figure also shows fragments marked as either regenerated or new, which drives our adaptive working-set-size algorithm (Section 4.4.)

costly. Another key operation, resizing the indirect branch lookup hashtable, requires adding synchronization to the performance-critical in-cache lookup routine. 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 thread-shared and thread-private caches, 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 3 shows the percentage of code fragments that are used by more than one thread in our multithreaded benchmarks. Even in interactive desktop applications there is a remarkable lack of shared code, which matches previous results [22] 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 in DynamoRIO.

4.2 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: a single memory store at the top of each fragment incurs an average performance impact of nearly eight percent on our SPEC CPU suite. 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 [17].

Figure 4 illustrates our 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. 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 remaining sources of cache fragmentation are deletion of trace heads as each trace is built (since the trace replaces the basic block at the trace entry point) and cache consistency evictions. Both of these cause holes in arbitrary locations in the cache. To combat these types of fragmentation, we use *empty slot promotion* [3, p. 161–163]. When a fragment is deleted 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 level of indirection to separate the FIFO from the actual

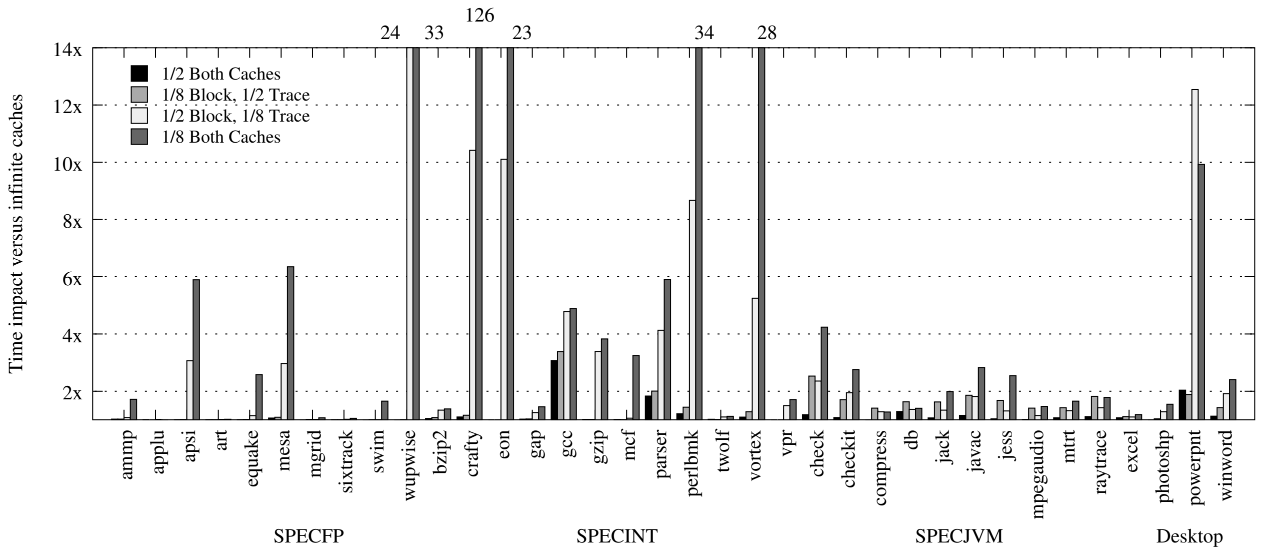

Figure 5. Performance impact of shrinking the basic block cache and the trace cache to one-half and one-eighth of their largest sizes, versus unlimited cache sizes.

cache address order.

Examining only capacity, batch eviction of contiguous code cache space is more efficient than single-fragment deletion because it better amortizes costs [16]. However, consistency evictions target scattered locations in the cache, and cannot be easily batched. Furthermore, they can thwart batch allocation units that must be de-allocated together to achieve any benefit. Consistency events are common enough in modern applications that their impact must be designed for.

4.3 Cache Size Effects

To study the performance effects of limited cache size, we imposed a hard upper bound equal to a fraction of the space used by each benchmark in an unbounded cache. Performance can be significantly impacted by an order of magnitude or more (Figure 5). While a different policy than our FIFO that uses profiling (ignoring the overhead of such profiling that makes it unsuitable for runtime use), such as LRU or 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 expected. Evicting a hot trace is more detrimental than losing a basic block, 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 limited space constraints. Benchmarks that adapt well are those that contain much initialization or other code that adds to the total cache usage but is not performance-critical. Thus, maximum unbounded cache usage is not a good metric to use for sizing the cache.

4.4 Adaptive Working-Set-Size Detection

We developed a novel scheme for automatically adapting the code cache to the current working set size of the application, to reduce memory usage but avoid thrashing. Not only are user-supplied cache size requirements eliminated, but our dynamically adjusted limit supports applications with phased behavior that will not work well with any hardcoded limit. The insight of our algorithm is that non-performance-critical code such as initialization sequences are good candidates for eviction since they 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 either to a simple cache or to each component in a generational cache.

Our sizing technique is driven by the ratio of *regenerated* fragments to *replaced* fragments. We begin with a small cache, and once it fills up, we evict old fragments (using our FIFO policy) to make room for new ones. The number of fragments evicted is the *replaced* portion of the ratio.

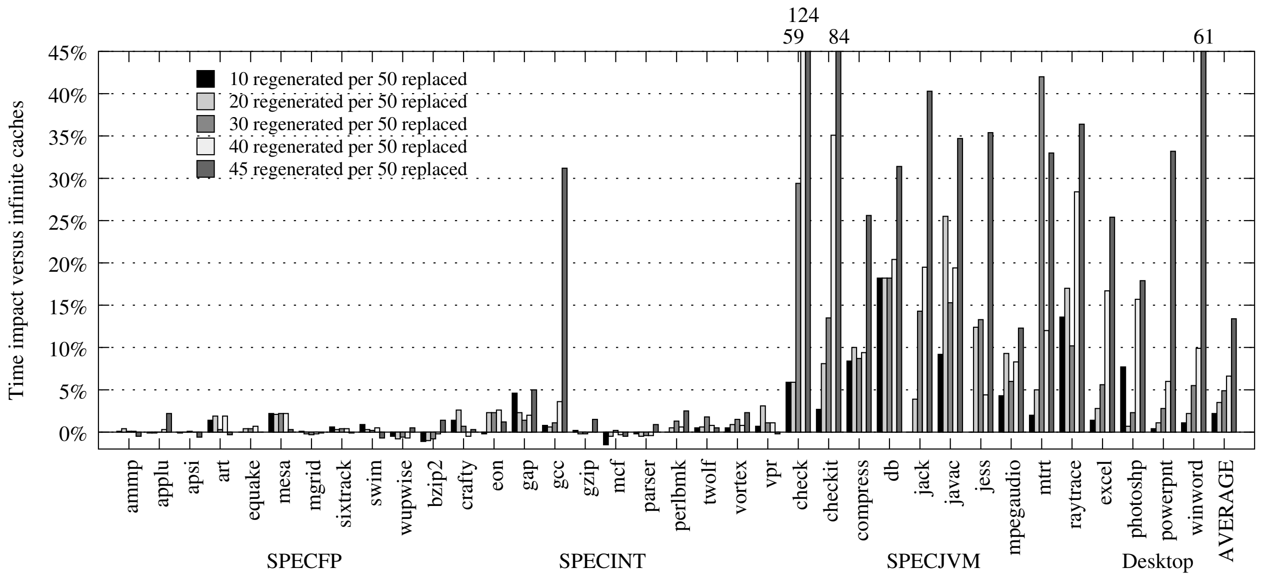

Figure 6. Performance impact of our adaptive working set with varying ratio thresholds: 10 regenerated / 50 replaced, 20/50, 30/50, 40/50, and 45/50, versus an unbounded cache.

We record in a hashtable every fragment that we remove from the cache. 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). For efficiency we check only the starting address for regeneration, potentially considering two traces with different tails to be identical. If the new fragment was previously present, we increment our count of *regenerated* fragments. Figure 4 illustrates the marking of fragments as new or regenerated. Only if a significant portion of new fragments are regenerated should the cache be larger than it is, in which case we double its size. The periodic ratio checks allow us to adapt to program behavior only when it changes — our checks are in runtime system code and incur no cost while execution is in the code cache. If the working set changes, we will replace the old fragments with new fragments.

To evaluate how the ratio threshold affects performance, we kept the replaced fragments parameter constant at 50 and varied the regenerated component from 10 up to 45. As Figure 6 shows, most of our benchmarks see little effect at the smaller parameters, with a harmonic mean of only a 3.5% slowdown at 10 regenerated fragments. The impact of higher ratios is significant in some cases, due to the overhead of freeing fragments and of regenerating those that are re-used but deleted, with an average 13.4% slowdown at 45 regenerated fragments. The applications most affected are those that execute large amounts of code with little reuse that will not amortize extra time spent in DynamoRIO (`gcc` and our desktop benchmarks) as well as those that are already experiencing code cache churn due to cache consistency flushes (`SPECJVM`), where restricting cache expansion exacerbates DynamoRIO's performance problems with such dynamically generated code.

The resulting cache sizes from these parameters are shown in Table 4. The most striking result is how much more resistant the trace cache is to shrinking than the basic block cache: for many of the benchmarks, the trace cache quickly reaches the core working set size and remains the same with successively higher regeneration thresholds. Conversely, since re-used code is promoted to the trace cache, the basic block cache can be shrunken significantly, an average of one-third and one-half at our two lowest thresholds, and up to 97% (for `excel` and `winword`) at our highest threshold. There is a tradeoff between memory and performance, and the lower ratio thresholds should typically be chosen, since they achieve sizable memory reductions at low performance cost.

We next explored the frequency for resize checks by executing our benchmark suite with a constant ratio but varying the replaced-fragment denominator. The results in Figure 7 show that while less frequent checks do affect a few benchmarks negatively, overall there is no significant impact. Checking too frequently may be too easily influenced by temporary spikes, and rarely not reactive enough. Thus, we chose our value of 50 from the center of this range. Further exploring the parameter space is left as future work.

Another area of 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. Explicit application actions like unloading libraries that imply reductions

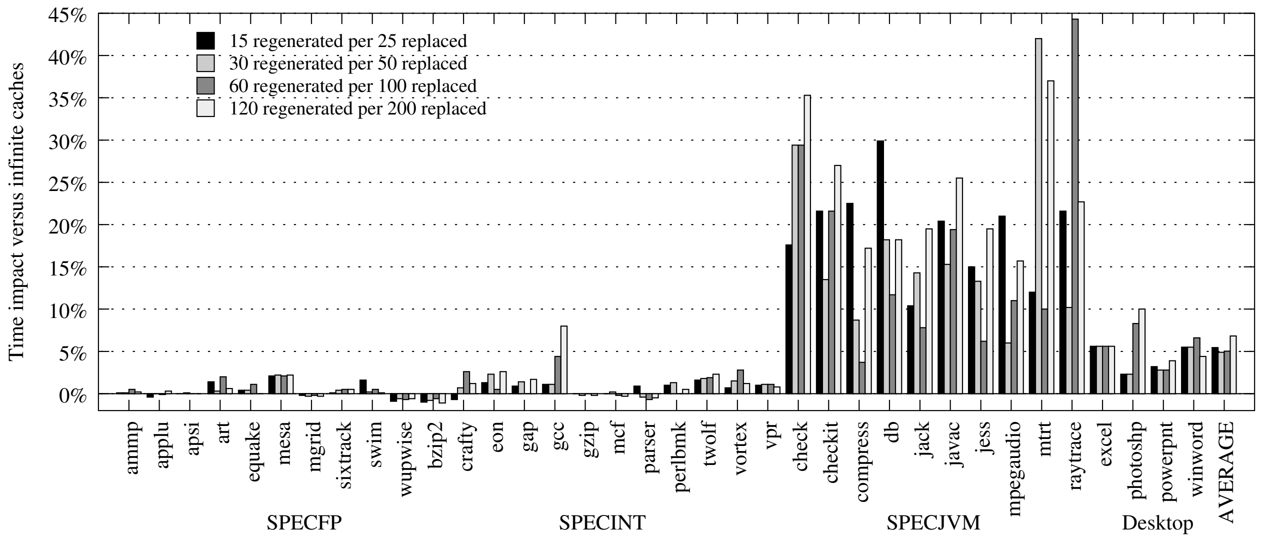

Figure 7. Performance impact of our adaptive working set with varying resize check frequency: 15 regenerated / 25 replaced, 30/50, 60/100, and 120/200, versus an unbounded cache.

Parameters:	∞	10	20	30	40	45
Benchmark	Basic block cache (KB)					
SPECFP	77	67	66	65	58	58
SPECINT	198	157	149	137	124	111
SPECJVM	1512	896	734	728	665	677
excel	2618	1632	1312	1024	64	64
photoshp	5521	3360	1856	1472	1184	1216
powerpnt	4242	2592	1632	992	736	480
winword	3135	2240	1664	1248	1024	64
ave impact	—	**-33%**	**-49%**	**-58%**	**-69%**	**-76%**
Benchmark	Trace cache (KB)					
SPECFP	127	109	109	109	109	105
SPECINT	350	305	307	306	300	292
SPECJVM	889	655	659	647	634	571
excel	608	430	416	404	366	347
photoshp	3371	2336	2336	2304	2114	1992
powerpnt	2259	1520	1536	1504	1487	1440
winword	1126	915	909	896	832	640
ave impact	—	**-23%**	**-23%**	**-24%**	**-27%**	**-32%**

Table 4. Code cache sizes when using our adaptive working set algorithm with regeneration thresholds of 10, 20, 30, 40, and 45, per 50 replaced fragments. The main thread's cache sizes are given for multi-threaded benchmarks, and the average for suites.

in code are the best candidates for driving cache shrinkage. Efficiently identifying whether the application is using the full code cache or not is challenging. Detecting true idle periods requires periodic interrupts, which are problematic on Windows without either a dedicated runtime system thread or a runtime system component that lives in kernel space.

5 Related Work

Many code caching techniques were pioneered in instruction set emulators [8] and whole-system simulators [36]. Software code caches are also coupled with hardware support for ISA compatibility [14, 10, 11] and used to virtualize hardware [4, 9]. Recent runtime tool platforms have turned to code caches to avoid the transparency and granularity limitations of traditional methods of inserting trampolines directly into application code [30, 26, 19]. Dynamic translation systems use code caches to reduce translation overhead [29, 37, 7], while dynamic optimizers attempt to exceed native performance by placing optimized versions of application code into code caches [2, 5]. A final category of systems with code caches are just-in-time (JIT) compilers that store generated code [1].

5.1 Code Cache Consistency

Any system with a software code cache is subject to the problem of cache consistency. Most RISC architectures require an explicit instruction cache flush request by the application to correctly execute modified code, usually in the form of a special instruction [20]. Systems like Shade [8], Embra [36], Dynamo [2], and Strata [30] watch for this instruction and invalidate their entire code cache upon seeing it. DELI [11] states that it handles self-modifying code, but gives no details on how this is achieved.

The IA-32 architecture requires no special action from applications to execute modified code, making it more challenging to detect code changes. Due to this difficulty, and because some programs do not require the feature, many systems targeting IA-32 do not handle modified code [5, 7, 30, 31, 19]. Other systems that target IA-32 must, like DynamoRIO, turn to page protection. Daisy [14] 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 [10] 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 [10]. VMWare [4] and Connectix [9] purportedly use page protection combined with sandboxing as well, though details are not available. Since most of these systems execute underneath the operating system, they avoid the problems of multiple threads. The exception is VMWare's multiprocessor support, which should have similar consistency problems to ours, but for which no technical information is available. 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.

5.2 Code Cache Capacity

There is little prior work on optimally sizing software code caches. Nearly every system known to us (the exceptions are virtual machines [4, 9]) 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 [5], although Valgrind [31] performs 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 SPEC CPU [35]) execute relatively small amounts of code.

Cache eviction studies have concluded that a FIFO policy works as well as any other policy in terms of miss rate, including LFU or LRU [17]. The conclusion of later work [16] 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 [2] 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 has focused on data references, attempting to improve prefetching and cache locality [6, 32], or on reducing windows of simulation while still capturing whole-program behavior [33]. 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 [12].

6 Conclusions

While code caching technology is relatively mature, little work has been done on maintaining cache consistency in the presence of multiple threads or on sizing code caches. These are two important milestones toward deployment of dynamic translators and optimizers on production systems running large, modern, complex applications. This paper contributes in both of these areas, with an efficient scheme for keeping a software code cache consistent across multiple threads without hardware support, as well as a novel runtime algorithm for dynamically bounding code cache size to match the current working set of the application. We evaluate these algorithms in DynamoRIO, a real code caching system running on stock IA-32 hardware under both Windows and Linux, on modern applications with dynamic behavior, including dynamically-loaded, generated, and even modified code.

While our cache consistency algorithm works well with both thread-private and thread-shared caches, our cache sizing algorithm takes advantage of the efficiency of thread-private single-fragment deletion. We leave efficient sizing of thread-shared caches as future work. A second area of future work for our sizing scheme is extending it to reduce cache sizes during idle periods.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 280–290, June 1998.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI '00), pages 1–12, June 2000.

[3] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., Sept. 2004.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *16th ACM Symposium on Operating System Principles (SOSP '97)*, pages 143–156, Oct. 1997.

[5] W. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, Dec. 2000.

[6] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 191–202, June 2001.

[7] C. Cifuentes, B. Lewis, and D. Ung. Walkabout — a retargetable dynamic binary translation framework. In *4th Workshop on Binary Translation*, Sept. 2002.

[8] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[9] Connectix. Virtual PC. `http://www.microsoft.com/windows/virtualpc/default.mspx`.

[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization (CGO '03)*, pages 15–24, Mar. 2003.

[11] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *35th International Symposium on Microarchitecture (MICRO '02)*, pages 257–268, Nov. 2002.

[12] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th International Symposium on Computer Architecture (ISCA '02)*, pages 233–244, June 2002.

[13] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th International Symposium on Computer Architecture (ISCA '86)*, pages 434–442, June 1986.

[14] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th International Symposium on Computer Architecture (ISCA '97)*, pages 26–37, June 1997.

[15] GNU Compiler Connection Internals. Trampolines for Nested Functions. `http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html`.

[16] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *International Symposium on Code Generation and Optimization (CGO '04)*, pages 89–99, Mar. 2004.

[17] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *Workshop on Interaction between Compilers and Computer Architecture (Interact-6)*, pages 102–110, Feb. 2002.

[18] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 1–3. 2001. Order Number 245470, 245471, 245472.

[19] Intel Corporation. Pin — A Binary Instrumentation Tool, 2003. `http://rogue.colorado.edu/Pin/`.

[20] D. Keppel. A portable interface for on-the-fly instruction space modification. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, pages 86–95, Apr. 1991.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):241–248, Sept. 1979.

[22] D. C. Lee, P. J. Crowley, J. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on Windows NT. In *25th International Symposium on Computer Architecture (ISCA '98)*, pages 27–38, June 1998.

[23] J. R. Levine. *Linkers and Loaders*. Morgan-Kaufman, San Francisco, CA, 1999.

[24] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[25] G. Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN, 2000.

[26] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *3rd Workshop on Runtime Verification (RV '03)*, Boulder, Colorado, USA, July 2003.

[27] M. Pietrek. An in-depth look into the Win32 Portable Executable file format. *MSDN Magazine*, 17(2), Feb. 2002.

[28] psbench@yahoo.com. PS6bench Photoshop benchmark (Advanced). `http://www.geocities.com/Paris/Cafe/4363/download.html#ps6bench/`.

[29] A. Robinson. Why dynamic translation? Transitive Technologies Ltd., May 2001. `http://www.transitive.com/documents/Why_Dynamic_Translation1.pdf`.

[30] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *International Symposium on Code Generation and Optimization (CGO '03)*, pages 36–47, Mar. 2003.

[31] J. Seward. The design and implementation of Valgrind, Mar. 2002. `http://valgrind.kde.org/`.

[32] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, Oct. 2004.

[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 45–57, Oct. 2002.

[34] Standard Performance Evaluation Corporation. SPEC JVM Client98 benchmark. `http://www.spec.org/osg/jvm98`.

[35] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, 2000. `http://www.spec.org/osg/cpu2000/`.

[36] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[37] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3):47–53, Mar. 2000.