Thread-Shared Software Code Caches

Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji
Determina, Inc.
{*derek,vladimir,tim,sanjeev*}*@determina.com*

Abstract

Software code caches are increasingly being used to amortize the runtime overhead of dynamic optimizers, simulators, emulators, dynamic translators, dynamic compilers, and other tools. Despite the now-widespread use of code caches, techniques for efficiently sharing them across multiple threads have not been fully explored. Some systems simply do not support threads, while others resort to thread-private code caches. Although thread-private caches are much simpler to manage, synchronize, and provide scratch space for, they simply do not scale when applied to many-threaded programs. Thread-shared code caches are needed to target server applications, which employ hundreds of worker threads all performing similar tasks. Yet, those systems that do share their code caches often have brute-force, inefficient solutions to the challenges of concurrent code cache access: a single global lock on runtime system code and suspension of all threads for any cache management action. This limits the possibilities for cache design and has performance problems with applications that require frequent cache invalidations to maintain cache consistency.

In this paper, we discuss the design choices when building thread-shared code caches and enumerate the difficulties of thread-local storage, synchronization, trace building, in-cache lookup tables, and cache eviction. We present efficient solutions to these problems that both scale well and do not require thread suspension. We evaluate our results in DynamoRIO, an industrial-strength dynamic binary translation system, on real-world server applications. On these applications our thread-shared caches use an order of magnitude less memory and improve throughput by up to four times compared to thread-private caches.

1 Introduction

Dynamic tools and other systems that operate at runtime often employ *software code caches* to store frequently executed sequences of translated or instrumented code for use on subsequent executions, thereby avoiding the overhead of re-translation. While caches can improve performance, their size must be carefully managed to avoid occupying too much memory and ultimately degrading performance. They also must be kept consistent with their corresponding original application code. Both tasks are complicated by the presence of multiple threads.

Any code caching system that targets applications with multiple threads faces a choice: increase memory usage by using thread-private caches, or increase the complexity of cache management by sharing the code cache. Some systems opt to not support multiple threads, in particular simulators and emulators that model a single processor [14, 15, 19, 28, 40]. Those that support multiple threads but choose thread-private caches enjoy straightforward and more efficient cache management, synchronization, and scratch space, and work well on applications with little code sharing among threads, such as interactive desktop programs [6, 26]. However, as we will show in Section 3, modern server applications have significant amounts of sharing among threads, and thread-private caches use prohibitive amounts of memory resulting in poor performance on these programs.

Existing systems that use thread-shared caches typically solve the thorny problem of evicting code from the cache via a brute-force solution: suspend all other threads or otherwise force them out of the cache immediately. This solution requires that cache management be kept to a minimum, which may not be practical for applications that incur many cache invalidations. Suspension also does not scale well on multiprocessor machines where it prevents concurrent execution. These shortcomings limit the applicability of such systems in production environments.

One contribution of this paper is a discussion of the design space and the key challenges of building thread-shared code caches. We analyze the need for sharing and the impact of thread-shared caches, which on server applications use an order of magnitude less memory and achieve up to four times better throughput compared to thread-private caches on some workloads (Section 3). We discuss the choices of what to share (Section 4) and how to provide scratch space (Section 5). Further contributions lie in specific solutions to various problems: synchronization (Section 6), trace building (Section 7), in-

cache indirect branch lookup tables (Section 8), and code cache eviction (Section 9).

2 Experimental Methodology

We evaluate our solutions in DynamoRIO [6], an industrial-strength dynamic binary translation system, operating on a benchmark suite targeting web and database servers. DynamoRIO is a native-to-native system that executes a target application process out of a basic block code cache. It also builds traces out of frequently executed sequences of basic blocks and places them in a trace code cache. Trace building is accomplished by profiling certain basic blocks marked as *trace heads*, which include loop heads and exits from existing traces. We refer to both traces and basic blocks with the generic term *block*. Other than trace building (which is described further in Section 7), the discussions in this paper are relevant to any software code caching system that deals with multiple threads and are not limited to a system with this particular basic-block-and-trace setup.

Our server benchmarks, listed in Table 1, target Microsoft Internet Information Services (IIS) 6.0 and Microsoft SQL Server 2000 SP3a, using default configurations except as explicitly noted. Both applications were run on the same Windows 2003 Advanced Server Dell PowerEdge 6600 machine equipped with 4 2.2GHz Xeon processors with hyperthreading enabled (resulting in 8 logical processors), 12K μops trace cache, 8KB L1 data cache, 512KB L2 cache, 2MB L3 cache, and 4GB RAM. The server was targeted over a 1Gbps Ethernet connection by a Windows XP SP1 2.2GHz Pentium 4 Dell Dimension 2400 client machine with 1GB RAM.

Our first two benchmarks use the Apache HTTP server benchmarking tool `ab` versus IIS 6.0 in 5.0 compatibility mode. `ab` was run with 60 simultaneous connections and 120,000 iterations, targeting the IIS SDK sample `iissamples/sdk/asp/docs/ColorPicker.asp`, which results in code running in `inetinfo.exe` when IIS is in low isolation mode (*ab low*) and in both `inetinfo.exe` and `dllhost.exe` when IIS is in medium isolation mode (*ab med*). Our second pair of benchmarks uses the Guestbook ASP application [38], again with IIS in either low (*guest low*) or medium (*guest med*) isolation mode, but also exercising SQL Server for dynamic content. For each benchmark we executed the workload twice (each runs for fifteen minutes) and reported the average throughput (Kb/s for *ab* and transaction count for *guestbook*) from the two runs. Our benchmarks are all CPU-bound.

3 Sharing Prevalence and Impact

A major design decision for any code caching system that supports multiple application threads is whether to use thread-shared or thread-private code caches. Thread-shared caches reflect the original application code, which lives in a thread-shared address space. However, thread-private caches are much simpler to manage for consistency and capacity, require no synchronization for most operations, can use absolute addresses for thread-local scratch space (Section 5), avoid indirection in performance-critical lookup table accesses (Section 8), and support thread-specific specialization for optimization or instrumentation.

To illustrate the challenges of thread-shared caches, consider the seemingly simple task of removing a block of code from the thread-shared code cache. It cannot be removed until it is known that no threads are executing inside that block. Yet, instrumentation of every block is too expensive, as is suspending every single thread to discover where it is every time a block needs to be removed (which may be frequent for purposes of cache consistency: Section 9.2). Thread-shared caches require more complex and sophisticated algorithms.

Thread-private caches do have an obvious and significant disadvantage: duplication of code in multiple threads' caches. The scope of this depends on the amount of code shared among threads. Desktop applications have been shown to share little code [6, 26], with a primary thread performing most of the work and the other threads executing disparate tasks. However, server applications deliberately spawn threads to perform identical jobs. Table 2 shows the amount of sharing among both basic blocks and traces for our benchmarks. Over one-half of basic blocks and two-thirds of traces are shared by at least two threads, and typically by tens of threads. This is strikingly different from desktop applications, which share less than ten percent of their blocks [5].

Although significant research attention has been given to exploring highly scalable event-driven architectures [25, 39], commercial server applications are mostly based on multi-threaded architectures. The concurrency model of our target server applications [30, 31] is based on pools of worker threads that handle connections and requests. Heuristics are used to control scalability by dynamic sizing of the number of threads: reacting to system load, expanding for bursty workloads, and shrinking after periods of inactivity, all within configurable minimums and maximums. SQL Server also supports *lightweight pooling* based on *fibers*, user-mode-scheduled threads of execution that reduce context switching overheads, with real (kernel-mode-scheduled) threads used only to migrate across processors. The best vendor-reported TPC-C [37] benchmark scores for SQL Server are produced in fiber mode, and DynamoRIO seamlessly supports fibers. Yet lightweight pooling is not enabled by default and is generally not recommended [21], due to incompatibilities with various extensions, e.g., incorrect expectations for thread local storage. In our goal to provide a transparent platform, we strive to provide minimal performance

Benchmark	Server	Processes
ab low	IIS low isolation	inetinfo.exe
ab med	IIS medium isolation	inetinfo.exe, dllhost.exe
guest low	IIS low isolation, SQL Server 2000	inetinfo.exe, sqlservr.exe
guest med	IIS medium isolation, SQL Server 2000	inetinfo.exe, dllhost.exe, sqlservr.exe

Table 1. Our database and web server benchmark suite and the processes that make up each target server application. DynamoRIO executes separately inside each process.

		Threads		Basic blocks			Traces		
Benchmark	Process	ever	peak	total	shared	threads per block	total	shared	threads per block
ab low	inetinfo.exe	138	127	100647	50%	22	7560	70%	27
ab med	inetinfo.exe	170	159	89688	47%	29	6185	68%	40
	dllhost.exe	175	162	51309	58%	53	4211	85%	71
guest low	inetinfo.exe	224	203	126654	58%	31	13865	79%	26
	sqlservr.exe	153	140	96893	51%	26	6980	74%	17
guest med	inetinfo.exe	192	141	95660	50%	16	7229	66%	11
	dllhost.exe	221	165	75772	68%	48	9929	84%	30
	sqlservr.exe	157	144	97081	54%	29	6907	62%	18
average		178	155	91713	54%	31	7858	73%	30

Table 2. Fragment sharing across threads. For each process in each benchmark, we show the number of threads ever created, the peak number of simultaneously-live threads, and for each of basic blocks and traces: the total count, the percentage executed by more than one thread, and the average number of threads executing each shared block.

degradation not only for applications tuned for optimal native execution workloads but also for sub-optimally designed or configured applications. We must also not perturb self-tuning heuristics. Our benchmarks therefore use the default thread worker model for SQL Server and default thread pool parameters for IIS.

We evaluated the performance and memory usage of DynamoRIO using both thread-private and thread-shared caches, implementing the designs described in the following sections. The results show that thread-shared caches are a clear winner for server applications, and the average throughput reduction compared to native on these CPU-bound benchmarks is now only 11%. Figure 1 gives the throughput of DynamoRIO using shared caches versus private caches, resulting in an average improvement of 1.9x. Shared caches achieve up to four times the throughput of private caches, due to reduced pressure in the hardware instruction cache, instruction TLB, branch predictors, branch target buffer, and other address-indexed structures. Since server threads are often executing for a short amount of time, and when blocked or pre-empted may be replaced by a possibly related thread, frequent context switching is a lot less expensive with shared caches. Even more dramatic is the memory usage, shown in Figure 2. The memory expansion from thread-private code caches quickly becomes egregious when the thread count numbers in the hundreds. Thread-shared code caches bring memory usage down and allow applications that benefit from using all available memory to avoid scalability limits.

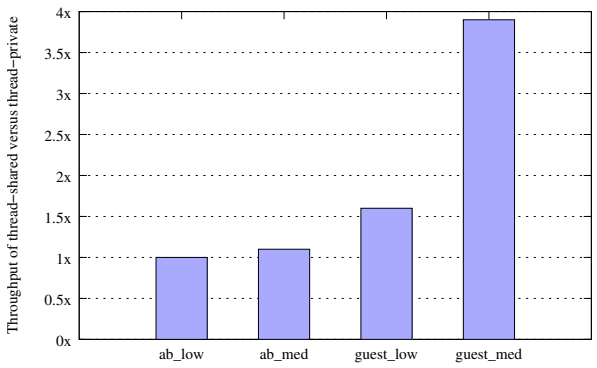

Figure 1. Throughput achieved by DynamoRIO using thread-shared caches versus thread-private caches

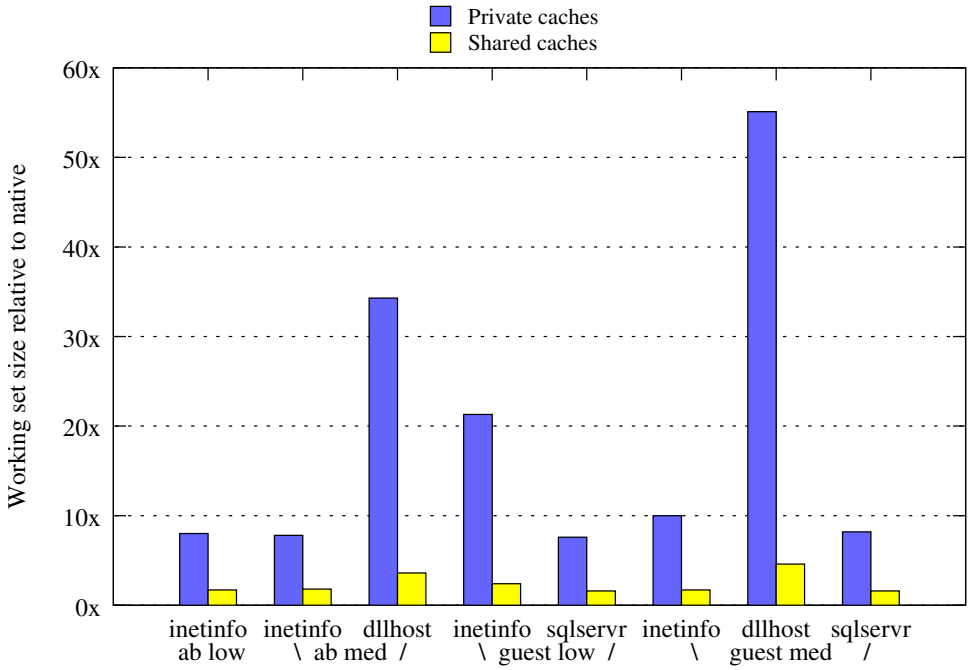

Figure 2. Memory usage comparison of thread-private and thread-shared caches, in terms of peak working set size (as reported by the operating system) versus native usage. The processes that make up each benchmark are shown individually.

4 Sharing Choices

Each component of a runtime system can be separately made thread-shared or thread-private: basic blocks, traces, trace building markers and profiling data, and indirect branch target lookup tables. Mixtures can also be used. For example, even when using thread-shared basic blocks in general, DynamoRIO keeps basic blocks that correspond to observed self-modifying application code in a thread-private cache to allow quick synchronization-free deletion when modifications are detected.

In addition to the code cache, every runtime system maintains associated data structures for managing the cache and its blocks. Runtime system heap management parallels cache management, with thread-private requiring no synchronization and thread-shared requiring assurance that no thread holds a pointer to a structure before it can be freed. Whether pointers to private structures are allowed inside shared structures, or vice versa, is another source of complexity. DynamoRIO avoids such arrangements.

With an all-shared or an all-private code cache, *links* connecting blocks in the cache have no unusual restrictions. However, when mixing shared and private, links between the two require care. Private code can target shared code with no extra cost, but shared code must dispatch by thread or use indirection through a thread-local pointer to reach the private code for the executing thread. If any type of cross-cache links are allowed, data structure management becomes more complicated. If lists of incoming links are used for proactive linking and fast unlinking [6], the system can end up with pointers to shared data embedded in private data structures. As we mentioned above, DynamoRIO avoids this mixture and thus does not allow cross-cache links. This is not problematic due to our rare use of thread-private blocks, which we only use for cases such as self-modifying code that are not often on critical performance paths.

5 Thread-Local Storage

Any runtime system requires scratch space to be available at arbitrary points during application execution, in order to operate while preserving application state. The simplest, most efficient, and most transparent form of scratch space access is absolute addressing, as it does not affect application register or stack usage. This addressing mode is supported by IA-32, DynamoRIO's target architecture. However, absolute addressing only works well with thread-private caches. For thread-shared code we need thread-private scratch space accessible via a shared instruction. Our choices are using the stack, which is neither reliable nor transparent; stealing a register, which incurs a noticeable performance hit on the register-poor IA-32 architecture; and using a segment, which is not available on all platforms but is on IA-32.

Segments are used by both Windows and recently

Linux [17] to provide thread-local storage space. We can either use the same space and try not to interfere with the application's slots, or we can create our own segment and steal a segment register. The offset of our scratch space from the segment base must be a known constant. DynamoRIO uses Windows-provided thread-local storage. Windows provides 64 storage slots within each thread's segment, with an additional 1024 entries added in Windows 2000 but which require an extra indirection step to access and so cannot be used as primary scratch space. DynamoRIO abides by the storage's allocation scheme to prevent conflicts with the application. To avoid indirection and thus improve performance we use multiple slots, and transparency problems are still possible when we compete for this limited resource with applications with hard requirements for direct access.

When mixing thread-shared and thread-private code, we first tried to use segment space for the shared code and absolute addressing pointing at a different location for the private code. In an alternative experimental configuration with shared basic blocks and private traces, the portion of a trace's code coming from its constituent basic blocks used the segment space while the newly generated part of the trace used the absolute space. This mix increased the data cache footprint enough to cause a noticeable performance hit. Absolute addressing can be mixed with segment addressing, but they should both point at the same linear addresses for best performance.

6 Synchronization

Sharing blocks across threads requires synchronized access to data structures and code cache modifications. Our first version of sharing used a single monolithic lock for all runtime system code, where only one thread could be out of the code cache at a time. The contention on this lock was high and performance suffered, as shown in Figure 3. Most of this overhead is only incurred at startup and mostly impacts short workloads, while longer continuous workloads spend less time in the runtime system and consequently are less affected in steady state. If thread-private data structures need to be populated, bursty workloads may also see some impact whenever thread pools shrink and expand. Yet, only a slightly finer-grained approach is required to achieve good performance across varied workloads. We use two main locks: a *basic block building lock* that is held across looking up, building, and adding a new basic block; and a *change linking lock* that is held across any changes to the link state of a block, including replacing any placeholder containing state for the new block to assume (such as *persistent basic block profiling data* that preserve execution counts across block deletion [6, p. 49]). If private blocks exist and any form of cross-cache linking is allowed, then the change linking lock must be held during private block linking as well. Additionally, if the trace headness at-

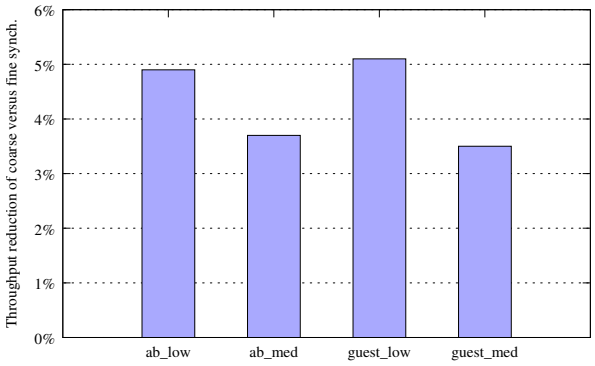

Figure 3. The performance impact of coarse-grained synchronization that employs a single monolithic lock around all runtime system code, versus our scheme of slightly finer-grained locks for basic block building and link modification.

Benchmark	Process	Reduction in contention instances
ab low	inetinfo.exe	53.1%
ab med	inetinfo.exe	85.0%
	dllhost.exe	53.4%
guest low	inetinfo.exe	67.0%
	sqlservr.exe	70.3%
guest med	inetinfo.exe	80.3%
	dllhost.exe	58.8%
	sqlservr.exe	75.5%
average		67.9%

Table 3. The reduction in lock contention instances when using our finer-grained locks for basic block building and link modification compared to using a coarse-grained monolithic lock.

tribute of each basic block is shared (see Section 7), the lock is required during initial linking of a new private block (when trace headness is discovered). As Table 3 shows, the contention on these two finer-grained locks is substantially lower than with the monolithic approach.

In addition to these high-level operation locks, each global data structure requires its own lock. These include block lookup hashtables, tables mapping code cache addresses to application addresses, and lists of available slots in the code cache. These locks must be efficient to avoid contention. Using spin locks or thread yields is not sufficient, as block lookups are on a relatively critical path. We found that only with fast read-write locks where a reader incurs very little overhead and all contention is handled by operating system-supported waits and notifies

could we eliminate all performance impact of our locks.

Since DynamoRIO operates on multithreaded applications it must be careful about interactions of its own synchronization with that of the application. A thread in the code cache should be executing completely in the application's context and should not hinder execution of runtime system code. Our invariant is that *no runtime system lock can be held while in the code cache*. This greatly simplifies both the *safe points* [6, p. 116] necessary for supporting application threads suspending each other and the synchronization needed for code cache consistency [5]. However, this limits the choices for trace building synchronization, as discussed in Section 7.

7 Trace Building

The mechanisms of trace building in a shared code cache require more changes and decisions than simply applying locks at the appropriate points in a private trace building scheme. This section discusses building Next Executing Tail (NET) traces [18], which each begin from a basic block called a *trace head*. Traditional NET trace heads focus on loops by including targets of backward branches as well as exits from existing traces [6]. Trace heads are profiled, and as soon as a trace head's execution counter exceeds a threshold value, the subsequent sequence of basic blocks that is executed after the trace head is concatenated together to become a new trace. In this scheme there are several independent choices of what to share and what remains thread-private: basic blocks, trace headness (whether a basic block is considered a trace head), trace head counters, and traces themselves. Hybrid choices are also possible, where some traces are private and some (perhaps those found to be common) are promoted to shared traces, or the reverse where shared traces are turned into thread-private traces for thread-specific specialization.

While trace headness sharing is typically tied to basic block sharing, and trace head counter sharing is typically tied to sharing traces themselves, the connections are not necessary. Having counters shared but traces private could be desirable if trace-specific optimizations are performed on the traces, or if shared thread-local storage is expensive and private traces have a performance advantage.

NET trace building involves executing basic blocks one at a time, incrementally discovering the hot path during the next execution after a trace head becomes hot. This entails multiple trips in and out of the code cache. Given our invariant of no locks while in the code cache (Section 6), this rules out a giant trace building lock. Instead, we use thread-private temporary data structures to build up traces and only synchronize at the point where a trace is ready to be emitted into the code cache. To prevent wasteful concurrent trace building work, we set a flag on a shared trace head once trace building has started from that head. Another thread will not attempt to build a trace from a flagged trace head but will instead continue executing from existing blocks. This race is not uncommon, as Table 4 shows. Without this flag, many duplicate traces would be simultaneously built and wasted.

Benchmark	Process	Trace building races	Ratio of races to traces
ab low	inetinfo.exe	4985	65.9%
ab med	inetinfo.exe	1029	16.6%
	dllhost.exe	2532	60.1%
guest low	inetinfo.exe	7730	55.8%
	sqlservr.exe	2886	41.3%
guest med	inetinfo.exe	662	9.2%
	dllhost.exe	6638	66.9%
	sqlservr.exe	3572	51.7%
average		3754	45.9%

Table 4. Trace creations attempted while another trace from the same trace head was being created, and the ratio of these races to the total number of traces built.

Each target block in the next executed tail must be unlinked in order to return to the runtime system after execution and continue the process. Since we cannot hold a lock while in the code cache, we cannot prevent the target block from being re-linked by another trace in progress or otherwise being modified. We solve this by again using thread-private temporary structures, this time for the basic block itself. This also avoids disrupting other threads by eliminating unlinking of shared blocks.

If both traces and basic blocks are shared, a trace head is no longer needed once a shared trace has been built from it. As block deletion is an involved operation in a shared cache (Section 9.2), one option is to not delete the head but instead *shadow* it. Shadowing makes the head inaccessible by ensuring that the trace has precedence in all lookup tables and by *shifting* the trace head's links to the trace. If the trace is later deleted, the links can be shifted back to restore the head. However, as this does waste space, DynamoRIO deletes a trace head using our two-step lazy deletion (Section 9) as soon as its corresponding trace is emitted.

8 In-Cache Lookup Tables

In a software code cache, indirect branches must be dynamically resolved by looking up the corresponding code cache address for a given application address. These indirect branch lookup tables present more synchronization complications than other runtime system data structures because they are accessed from the code cache and are on the critical performance path. A runtime system's

performance has been shown to be primarily limited by its indirect branch performance [6], which is the only aspect of code cache execution that differs significantly from native execution.

As with the other components, lookup tables can be thread-shared or thread-private. Even if all blocks are shared, thread-private tables simplify table entry deletion and table resizing, as described below. Private tables do occupy more memory than shared, but they do not significantly impact scalability the way thread-private basic blocks and traces do. In our benchmarks, the total memory usage of thread-shared blocks is four times greater than the memory usage from thread-private lookup tables. The main disadvantage of thread-private tables is that the table of every thread must be traversed in order to remove a shared block. There is also additional data cache pressure.

Thread-shared tables require synchronization with other threads in runtime system code to coordinate concurrent writes to the table as well as to make sequences of reads and writes atomic. We use read-write locks to accomplish this. However, we can avoid the cost of a read lock for the in-cache lookup if we make the key table operations atomic with respect to reads from the cache. The key operations are: adding a new target block; removing a block; and resizing the table.

Adding a new block can be made atomic with respect to table reads from the cache by first adding subsidiary fields and only then using a single IA-32 atomic write to the primary tag field to enable the new entry. Removing entries is a little more difficult and depends on the type of hashtable collision handling. We use an open-address hashtable with linear probing [13], where shifting entries on deletion produces shorter collision chains than using a sentinel. However, shifting cannot be used without a read lock in the cache. Our solution is to use a sentinel that is not a hit but does not terminate the collision chain, which can be written atomically to the tag field. Atomic removal is required for thread-private as well as thread-shared caches on cache consistency events (see Section 9), as blocks must be invalidated in all threads' caches by the thread processing the event. For thread-private tables, the owning thread can clean up the sentinel and perform deletion shifting on its own table when back in runtime system code. For thread-shared tables, we cannot do any shifting or replacement of a sentinel unless we know all threads have exited the cache since the sentinel's insertion.

Resizing is the most difficult of the three operations. As there is a large range in amount of code executed by different applications, no single table size will provide both small data cache footprint and small collision chains — the table must be dynamically sized as the application executes new code. Thread-private lookup tables can be resized by their owner at any point. For thread-shared tables, we point at the new table but do not free the old table right away. A reference counting scheme is used to lazily reclaim the memory.

After implementing both thread-private and thread-shared lookup tables in DynamoRIO, we have observed thread-private to be marginally faster, possibly due to shared tables' lack of sentinel replacement resulting in longer collision chains. As future work we plan to explore the effects of more sophisticated shared table management. The numbers in this paper are from DynamoRIO configured to use thread-private tables.

Lookup routines can also be either shared or private. Shared routines must use indirection to access their lookup tables (unless a hardcoded table size is used, which does not scale well, or several registers are permanently stolen, which will have serious performance problems on IA-32), another disadvantage of sharing. DynamoRIO's shared routines store table addresses and lookup masks directly in thread-local storage (Section 5) in order to avoid a second indirection step. The best general approach may be a hybrid that uses thread-private lookup routines and tables but switches to thread-shared if the application turns out to use many threads with significant sharing.

9 Code Cache Eviction

Code must be evicted from software code caches for two reasons: *cache consistency* and *cache capacity*. This section shows that while invalidating code by making it unreachable is similar between thread-private and thread-shared caches, actually freeing code is very different in the thread-shared world.

9.1 Unlinking

Any software code cache must be kept consistent with the application's original code, which can change due to code modification or de-allocation of memory. These events are more frequent than one might expect and include much more than rare self-modifying code: unloading of shared libraries; rebasing of shared libraries by the loader; dynamically-generated code reusing the same address, or nearby addresses (*false sharing*) if the method of change detection is not granular enough — and dynamically-generated code includes nested-function trampolines and other code sequences not limited to just-in-time compilers; hook insertion, which is frequent on Windows; and rebinding of jump tables. All of these application changes must be handled by invalidating the corresponding code in the code cache. Our experience running commercial server software revealed as a common source of bugs attempts to execute from already unloaded libraries. Rather than data corruption these latent bugs more often result in execution exceptions that are handled within the application, and our platform has to faithfully reproduce those exceptions for bug transparency.

The presence of multiple threads complicates cache invalidation, even with thread-private caches (as stale code could be present in every thread's cache). Bruening and Amarasinghe [5] present an algorithm for *non-precise flushing* using a slightly relaxed consistency model that is able to invalidate modified code in a lazy two-step scheme that avoids the up-front cost of suspending all threads on every invalidation event. (Self-modifying code is handled differently [5], in a precise fashion, which is one reason that DynamoRIO keeps self-modifying blocks thread-private.) Non-precise flushing assumes that the application uses synchronization between a thread modifying code and a thread executing that code. A code cache system can then allow a thread already inside a to-be-invalidated block to continue executing and only worry about preventing future executions. This results in a two-step scheme: one, making all target blocks inaccessible, which must be done proactively; and two, actually freeing the stale blocks, which can be performed lazily. Here we extend that algorithm and verify that the first step works with the addition of our shared indirect branch table solutions from Section 8, but the second step requires a novel scheme for freeing memory, which we present in Section 9.2.

Methods for removing blocks from indirect branch target tables atomically, concurrent with other threads' access to the table, were discussed in Section 8. In addition, the first step requires *unlinking* of all target blocks. Unlinking is the act of redirecting all entrances to and exits from a block (including self-loops) to instead exit the code cache. These redirections involve modifying direct branch instructions. IA-32 provides an atomic four-byte memory write (though despite the implications of the IA-32 documentation [23, vol. 3], cross-cache-line code modifications are *not* atomic with respect to instruction fetches). A branch modification is a single write and can be made atomic by ensuring that the branch instruction's immediate operand in the code cache does not cross a processor cache line. A thread executing that branch will see either the linked state or the unlinked state — nothing inconsistent. This part of the flushing algorithm does not change from thread-private to thread-shared caches.

9.2 Delayed Deletion

While making a block inaccessible in the future can be done with atomic operations, actually freeing that block is much more difficult. We must be certain that no thread is currently inside the block. One strategy is to impose a barrier at entrance to the code cache and wait for all threads to exit. When no thread is in the code cache, clearly any block can be freed. We implemented this strategy but ran into several problems. Threads may remain in the cache for an unbounded amount of time before coming out, if in a loop or at a system call. This leads to two problems: first, freeing of blocks may be delayed indefinitely; and second, the application may make no forward progress because of the cache entrance barrier. A third problem is that the lock used to precisely count the threads as they exit ends up with very high contention (since every thread acquires it on every cache entrance and exit). The end result is both non-prompt deletion and poor performance.

As a real-world example of how important it is to free memory invalidated by cache consistency events, consider `VSSTat.exe`, the tray bar process for McAfee VirusScan [29]. It loads and then unloads the same shared library over one thousand times in the course of a few minutes of execution. While this is clearly suboptimal native behavior, our platform should faithfully maintain the application's characteristics. Yet without actual block freeing, our overall memory usage was *fifty times* what it should have been. While a more targeted solution to library reloading can remove this source of cache deletion, page or subpage consistency events due to cross-modified or self-modified code still demand more efficient general deletion support.

Our solution is to not require that all threads be out of the cache simultaneously, but rather that all threads that were in the cache at the time of the target block's unlinking have exited the cache at least once. This avoids the heavyweight entrance barrier and solves the performance problem. To determine whether a thread has exited since the unlink, we use timestamps, and to find the last thread, reference counting. A *global timestamp* is incremented on each unlink of a set of blocks (e.g., for each cache consistency event). That set of blocks is placed as a new entry in a pending-deletion list. The list entry also records the global timestamp and the total number of threads at the time of the unlink (which is the reference count). We use the total to avoid having to know how many threads are in the cache versus waiting at synchronization points. Furthermore, requiring every thread to indicate it is no longer using the set of blocks solves problems not only with stale code cache execution but with accesses to data structures corresponding to stale blocks.

Each thread remembers the timestamp at which it last walked the pending-deletion list. As each thread encounters a synchronization point (entering or exiting the cache, or thread death), it walks the pending-deletion list and decrements the reference count for every entry whose timestamp is greater than its own. After the walk, the thread sets its timestamp to the current global timestamp. The pending-deletion list is kept sorted (by prepending new entries) so that each walk can terminate at the first entry that has already been visited by that thread.

When a pending-deletion entry's reference count reaches zero, its blocks are guaranteed to be no longer in use either for execution in the code cache or examination by the runtime system. The blocks' data structures can now be freed, and their cache space re-used. Re-use of cache space is not as simple as for thread-private caches, where schemes like *empty-slot promotion* [5] are effec-

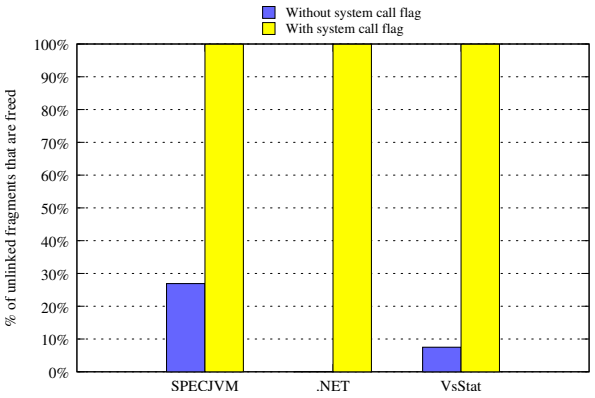

Figure 4. The percentage of unlinked blocks that are successfully freed by our algorithm with and without the system call flag addition, evaluated on applications with numerous cache consistency events: dynamically generated code in SPECJVM [36] and a sample .NET Framework SDK application, and repeatedly-unloaded libraries in `VSSTat.exe`, the tray bar process for McAfee VirusScan [29].

tive. Victim blocks adjacent to the empty slot cannot be proactively evicted (the multi-stage delayed deletion process must be undergone to ensure they are unused first), making the slots only useful to blocks that fit inside them. Our solution is to use free lists of various sizes for shared code caches.

Our delayed deletion algorithm still had one problem: deletion delay is unbounded due to loops and system calls in the cache. We address the system call problem with a flag that is set prior to executing a system call and cleared afterward. If the flag is set for a thread, that thread is not included in the reference count for a to-be-deleted set of target blocks. This scheme requires that during the unlinking step all post-system-call points are also unlinked prior to reading the system call flag. Otherwise a race in reading the flag could result in the uncounted thread accessing stale and freed data. DynamoRIO routes all system calls through a single point, facilitating an inexpensive unlink (if system calls are instead scattered throughout the code cache, an always-present check for whether to exit may be more efficient than having to unlink every single one of them). Upon exiting the code cache, the thread must abandon any pointer to any blocks (such as a last-executed pointer that some systems maintain) as it may have already been freed. Figure 4 shows that this system call addition to the algorithm makes a critical difference in effectiveness.

In practice we have not had a problem with loops. Though DynamoRIO's goal is to quickly enter steady state in the form of loops in the cache, server applications' steady-state loops often contain system calls, which our algorithm does address. Another factor is that applications with significant amounts of consistency events tend to exit the cache more frequently as they execute the code being unlinked. If problems with loops do arise, as a last resort one can occasionally suspend the looping threads in order to proactively free memory.

In addition to freeing blocks made invalid by consistency events, freeing is required in order to impose limits on the size of the cache for capacity management. Unfortunately, straightforward single-block eviction strategies such as first-in-first-out or least-recently-used that have been shown to work well with thread-private caches [5, 20] simply do not work with thread-shared caches, as no block can be freed immediately in an efficient manner. Capacity schemes must account for either a high cost of freeing or for a delay between asking for eviction and actual freeing of the space.

10 Related Work

Software code caches are found in a variety of systems. Dynamic translators use code caches to reduce translation overhead [10, 33, 41], while dynamic optimizers perform native-to-native translation and optimization using runtime information not available to the static compiler [4, 8]. Similarly, just-in-time (JIT) compilers translate from high-level languages to machine code and cache the results for future execution [1, 3, 16, 22]. Instruction set emulators [11] and whole-system simulators [28, 40] use caching to amortize emulation overhead. Software code caches are also coupled with hardware support for hardware virtualization [7, 12] and instruction set compatibility [14, 15, 19, 24]. To avoid the transparency and granularity limitations of inserting trampolines directly into application code, recent runtime tool platforms are being built with software code caches [6, 27, 32, 34].

Not all software code cache systems support multiple threads. Whole-system simulators, hardware virtualization systems, and instruction set compatibility systems typically model or support only a single processor, resulting in a single stream of execution (an exception is VMWare's multiprocessor support, but for which no technical information is available). Other tools and research systems target platforms on which kernel threads are not standard.

Many dynamic translation and instrumentation systems that do support threads have limited solutions to threading issues. Valgrind [32] is single-threaded and multiplexes user threads itself. Aries [41] uses a single global lock around runtime system code and supports freeing cache space only via forcing all threads out of the cache. DynamoRIO [6] originally used thread-private code caches. FX!32 [9] supports multiple threads but does not support cache consistency or runtime cache

management, using only persistent caches built via offline binary translation. Mojo [8] uses thread-shared trace caches but thread-private basic block caches. Its cache management consists of suspending all threads, which it only does upon reaching the capacity limit of the cache as it does not maintain cache consistency and cannot afford the suspension cost at more frequent intervals. Pin [27] has an adaptive thread-local storage approach, using absolute addressing until a second thread is created, when it switches to a stolen register. Further information on its handling of threads is not available.

Some threading problems are more easily solved in other types of runtime systems. Dynamic translators and just-in-time compilers are able to set up their own thread-local scratch space by allocating themselves a register, as opposed to native-to-native systems that must steal from the application in order to operate transparently.

Language virtual machines (e.g., Java virtual machines) often virtualize the underlying processors and perform thread scheduling themselves. They do not allow full pre-emption but rather impose synchronization points where thread switches may occur, typically at method entries or loop backedges. These points are used to simplify garbage collection by requiring all mutators (application threads) to be at synchronization points before garbage collection can proceed [2]. The overhead from such frequent synchronization is more acceptable in a virtual machine than a native-to-native system.

Garbage collection uses reference counting in a different way than our delayed deletion algorithm. Garbage collection determines what data is reachable from a root set, operating either in a stop-the-world fashion or by instrumenting stores to ensure that any references between collection sets are known. Deleting code blocks cannot use similar methods as instrumenting every block entrance and exit would be prohibitively expensive. Any thread can reach any block that is accessible via links or indirect branch lookup tables. Our reference count indicates not which threads are using target data, but which threads *might* be using target data.

Another difference between language virtual machines and other systems is that JIT-compiled code cache management operates at a coarser granularity, methods, than the blocks of code required for incremental code discovery in a system operating on arbitrary binaries. JIT compilers often go to great lengths to avoid compiling code that might ever need invalidation [35].

11 Conclusions

This paper presents and discusses our implementation of thread-shared code caches that avoids brute-force all-thread-suspension and monolithic global locks. Our final design includes medium-grained runtime system synchronization that reduces lock contention, trace building that combines efficient private construction with shared results, in-cache lock-free lookup table access in the presence of entry invalidations, and a delayed deletion algorithm based on timestamps and reference counts. We evaluated our solutions in the DynamoRIO runtime system on real-world server applications and found that our thread-shared caches reduce memory usage by up to factor of nine and improve throughput by up to a factor of four versus thread-private caches.

Our implementation supports mixing thread-private and thread-shared caches. We plan to extend this work by analyzing the differences in trace shape and quality between all-shared and all-private traces and by further exploring adaptive hybrid sharing schemes.

12 Acknowledgements

The authors thank Lina Tabch for her time and expertise with the server benchmarks in this paper.

References

[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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, 280–290.

[2] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., P.CHENG, CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1).

[3] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, 47–65.

[4] BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent runtime optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, 1–12.

[5] BRUENING, D., AND AMARASINGHE, S. 2005. Maintaining consistency and bounding capacity of software code caches. In *International Symposium on Code Generation and Optimization (CGO '05)*, 74–85.

[6] BRUENING, D. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T.

[7] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. In *16th ACM Symposium on Operating System Principles (SOSP '97)*, 143–156.

[8] CHEN, W., LERNER, S., CHAIKEN, R., AND GILLIES, D. M. 2000. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 81–90.

[9] 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.

[10] CIFUENTES, C., LEWIS, B., AND UNG, D. 2002. Walkabout — a retargetable dynamic binary translation framework. In *4th Workshop on Binary Translation*.

[11] 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.

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

[13] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, Cambridge, MA.

[14] 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 *International Symposium on Code Generation and Optimization (CGO '03)*, 15–24.

[15] DESOLI, G., MATEEV, N., DUESTERWALD, E., FARABOSCHI, P., AND FISHER, J. A. 2002. DELI: A new run-time control point. In *35th International Symposium on Microarchitecture (MICRO '02)*, 257–268.

[16] DEUTSCH, L. P., AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '84)*, 297–302.

[17] DREPPER, U., AND MOLNAR, I. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[18] DUESTERWALD, E., AND BALA, V. 2000. Software profiling for hot path prediction: Less is more. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, 202–211.

[19] EBCIOGLU, K., AND ALTMAN, E. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th International Symposium on Computer Architecture (ISCA '97)*, 26–37.

[20] HAZELWOOD, K., AND SMITH, M. D. 2002. Code cache management schemes for dynamic optimizers. In *Workshop on Interaction between Compilers and Computer Architecture (Interact-6)*, 102–110.

[21] HENDERSON, K. 2005. The perils of fiber mode. `http://msdn.microsoft.com/library/en-us/dnsqldev/html/sqldev_02152005.asp`.

[22] HÖLZLE, U. 1994. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University.

[23] INTEL CORPORATION. 2001. *IA-32 Intel Architecture Software Developer's Manual*, vol. 1–3. Order Number 245470, 245471, 245472.

[24] KLAIBER, A., 2000. The technology behind Crusoe processors. Transmeta Corporation, Jan. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[25] LARUS, J. R., AND PARKES, M. 2002. Using cohort scheduling to enhance server performance. In *USENIX Annual Technical Conference*, 103–114.

[26] 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 *25th International Symposium on Computer Architecture (ISCA '98)*, 27–38.

[27] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, 190–200.

[28] 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 *USENIX Annual Technical Conference*, 119–130.

[29] MCAFEE. VirusScan. `http://www.mcafee.com/`.

[30] MICROSOFT CORP. 1996. DCOM technical overview. `http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomtec.htm`.

[31] MICROSOFT CORPORATION. 2000. Internet Information Services 5.0 technical overview. `http://www.microsoft.com/windows2000/techinfo/howitworks/iis/iis5techoverview.asp`.

[32] NETHERCOTE, N., AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *3rd Workshop on Runtime Verification (RV '03)*.

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

[34] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J., AND SOFFA, M. L. 2003. Reconfigurable and retargetable software dynamic translation. In *International Symposium on Code Generation and Optimization (CGO '03)*, 36–47.

[35] SREEDHAR, V. C., BURKE, M., AND CHOI, J.-D. 2000. A framework for interprocedural analysis and optimization in the presence of dynamic class loading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, 208–218.

[36] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC JVM Client98 benchmark. `http://www.spec.org/osg/jvm98`.

[37] TPC. TPC-C. `http://www.tpc.org/tpcc/`.

[38] WEB WIZ. Guestbook. `http://www.webwizguide.info/asp/sample_scripts/guestbook_script.asp`.

[39] WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *18th ACM Symposium on Operating Systems Principles*, 230–243.

[40] WITCHEL, E., AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 68–79.

[41] ZHENG, C., AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3) (Mar.), 47–53.