Process-Shared and Persistent Code Caches

Derek Bruening
VMware, Inc.
bruening@vmware.com

Vladimir Kiriansky
VMware, Inc.
vkirians@vmware.com

Abstract

Software code caches are increasingly being used to amortize the runtime overhead of tools such as dynamic optimizers, simulators, and instrumentation engines. The additional memory consumed by these caches, along with the data structures used to manage them, limits the scalability of dynamic tool deployment. *Inter-process sharing of code caches* significantly improves the ability to efficiently apply code caching tools to many processes simultaneously.

In this paper, we present a method of code cache sharing among processes for dynamic tools operating on native applications. Our design also supports *code cache persistence* for improved cold code execution in short-lived processes or long initialization sequences. Sharing raises security concerns, and we show how to achieve sharing without risk of privilege escalation and with read-only code caches and associated data structures. We evaluate process-shared and persisted code caches implemented in the DynamoRIO industrial-strength dynamic instrumentation engine, where we achieve a two-thirds reduction in both memory usage and startup time.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – *Run-time Environments, Compilers, Optimization*.

General Terms Design, Algorithms, Security, Performance.

Keywords Dynamic Instrumentation, Binary Translation, Software Code Cache, Tool Scalability.

1. Introduction

Dynamic languages, emulators, and other tools typically 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. These caches, along with the data structures to manage them, however, consume significant amounts of memory. Initially, code caching tools were applied to only one process at a time, and the resulting memory expansion was deemed acceptable. As code caching technology has matured and its applications expanded to include security, optimization, auditing, profiling, and other features, applying code caches to all processes simultaneously, including on production systems, has become more desirable. When these tools are applied to many processes simultaneously the combined extra memory usage ultimately degrades performance.

This situation is similar to what operating systems originally faced with multiple applications executing similar code simultaneously. Their solution was to invent *shared libraries* to reduce resource usage and the cost of context switches. Unfortunately, code caches undo the benefits of shared libraries by making the shared code private again. To approach the scalability of native applications, code caching tools should take the shared library approach via *inter-process sharing of code caches*. Code cache sharing introduces new difficulties not present with shared libraries, which are statically generated and constant, while code caches vary across applications and executions:

- Code caches exported by separate processes must be merged.
- Code caches must be kept synchronized with their source application code.
- Different processes may have modules loaded at different addresses, different versions of modules, or varying dynamic modification to modules.
- Instrumentation added to the code cache also varies by tool and process.
- Code caches must be as secure as their source application code from malicious or inadvertent modification.

Code caching systems are expected to achieve all of these objectives transparently without introducing performance degradations.

This paper presents a method of inter-process code cache sharing for dynamic tools operating on native applications that solves the above challenges. Specific contributions include:

- A design for code cache sharing among processes that also supports persistence to improve cold code execution for short-lived processes or long application initializations.
- An adaptive-level-of-granularity code cache that supports minimal data structures, persistence, and inter-process sharing for typical unchanging application code while dynamically switching to a finer-grained model that performs better for dynamically-changing code.
- A security scheme that avoids privilege escalation while still allowing a high-privilege process to share and persist its code caches with low-privilege processes, as well as allowing low-privilege processes to share caches among other processes of the same user.
- A method to keep code caches, code cache exit trampolines, and data structures read-only in steady state with no overhead, protecting them from malicious or accidental modification.
- Implementation and evaluation of process-shared and persisted code caches in an industrial-strength dynamic instrumentation

VEE'08, March 5–7, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-796-4/08/03…$5.00

engine. We achieve a two-thirds reduction in both memory usage and startup time.

The next section describes the motivation for and goals of our inter-process sharing in more detail. Section 3 presents a threat model and describes our security solution. Section 4 gives details on our code cache design, which we have implemented in the DynamoRIO [5] industrial-strength dynamic instrumentation system, while Section 5 evaluates our results.

2. Inter-Process Sharing

Dynamic optimizers and runtime instrumentation engines are essentially in-process virtual machines whose data structures and code caches add to the memory usage of the native application. While modern operating systems share the application code contained in the executable and libraries among all processes on the system, the virtual machine's code cache and data structures are not shared, resulting in a scalability limit for dynamic tools that is lower than the native limit. An example is our benchmark of booting a laptop with all processes in code caches, which without sharing uses 110 megabytes additional memory beyond that used natively (Figure 3), even with a very efficient code caching system. Launching a number of desktop applications quickly adds tens of megabytes to that figure (Figure 5); the additional memory is likely to exceed available memory on some systems.

2.1 Goals

Our primary goal is to support systemwide deployment of dynamic code caching tools by raising the scalability ceiling without sacrificing security. Our approach is to employ inter-process sharing in a secure manner, including providing *self-protection*: if every application on a system is to be executed inside a code cache, malicious or accidental overwrites of the cache must be prevented in order to achieve robustness.

In addition to reducing memory usage, a secondary goal is to improve the performance of infrequently-executed, or *cold*, code, such as that found in short-lived applications and long application initialization sequences. As there is little opportunity to amortize code caching overhead without code re-use, code caching tools typically display poor performance on cold code.

2.2 Granularity of Sharing

A key design decision for inter-process sharing is the unit or granularity of sharing. In this paper we consider code caches that contain translations of native application code. Native applications are organized into *modules*, where the executable image itself and each shared library is a separate module. Modules are designed to allow inter-process sharing of their read-only sections: code and read-only data. The natural granularity of code cache sharing, then, is a mirror of the native code: at the module level.

We considered both larger and smaller units than modules but discarded both. Larger units of sharing, such as combining multiple modules, or sharing sequences of code that cross module boundaries, will only be shareable with other applications that have loaded the same set of modules. By sticking with intra-module code we align code cache shareability, removal, and versioning with the units of code that the application loads, unloads, and updates: modules. We also see no advantage to breaking up modules into smaller units for purposes of code cache sharing, other than perhaps splitting by page in order to reduce memory usage and validation costs (though as Section 3.2 shows we avoid per-page validation checks).

In our design we share only code that has been executed; we do not attempt to statically translate the entire module into a code cache. Even with this approach, an application that uses very little of a module may be forced to load a shared code cache that contains much more code than it needs. One alternative approach is to load a shared cache lazily, in pieces, rather than loading it all up front. We chose not to implement that, however, relying on the operating system's demand paging to bring into physical memory only the actual working set of the application. After all, most applications use a small fraction of any given shared library's native code, a parallel situation to our code caches.

We do not attempt to share or persist code not contained in modules. Such dynamically generated code is much less likely to be used identically in multiple processes in any case, along with being more difficult to version and identify.

2.3 Mechanism of Sharing

In addition to choosing the code cache units to share, we have two further key design decisions. First, we must decide whether live caches (i.e., caches being actively added to) will be shared, or only frozen caches no longer being modified. The former is more complex, requiring coordination among multiple processes when adding to the cache, as well as raising security and self-protection issues from the writable cache. Therefore, we chose to share only frozen, read-only caches.

The final design choice is the actual method of achieving inter-process memory sharing. This can be accomplished using either files or anonymous shared memory. An advantage of a file-based approach is that inter-process sharing and inter-execution persistence are both realizable from the same code cache design. We decided to go this route in order to improve both scalability and cold code performance. Many of the design requirements are identical between file-based and memory-based schemes: the code cache and its data structures must be made process-independent, and each of the challenges listed in Section 1 solved (merging; handling module versioning, rebasing, and modifications; allowing instrumentation; and maintaining security). There are additional complexities that arise only when using files, including handling underlying platform changes due to files being shared across machines and ensuring files on disk have not been tampered with. Section 3 discusses security issues with code cache files in more depth, while Section 4 presents details on our solutions to the other challenges in designing shareable and persistable code caches.

3. Security

Our goal is to avoid opening up new vulnerability vectors when executing from a shared or persisted code cache that do not exist natively. This section discusses our threat model and our solution for minimizing security concerns while still achieving substantial sharing.

3.1 Threat Model

We assume that a local user is able to create a new file, or modify an existing file, and give it arbitrary contents, provided the user's privileges allow such file writes. We also consider that a remote user can do the same thing, either via an exploit that allows arbitrary execution or only allows file creation. Given that our code caches are contained in files, we must limit modification of those files. The two primary concerns are:

Code modifiability If a user who does not have privileges to write to an application's executable or library files on disk is able to write to a persisted code cache file that will be executed by that application, then the caching system has introduced a vulnerability vector that was not present before. For example, a remote user of an ftp server can legitimately write a file, but must not be allowed to create a code cache file that will be picked up and executed by the next invocation of the ftp server.

Another aspect of code modifiability concerns in-memory code caches, as well as code cache data structures. Our solution for self-protection of the code caching system is to employ read-only caches and data structures as described in Section 4 and to avoid sharing writable memory as stated in Section 2.3.

Privilege escalation Unintentionally permitting a low-privilege user to control code executed by a higher-privilege user by creating or modifying a persisted code cache is a form of privilege escalation. Any inter-process communication used as part of the code cache sharing process, whether live or via file intermediaries, where a low-privilege process sends input to be acted on by a high-privilege process, is a potential vector of privilege escalation. We designed our process-shared and persisted code caches to avoid privilege escalation, as the next section shows.

3.2 Shareable Code Cache Generation and Accumulation

To avoid privilege escalation we cannot allow a code cache generated by a low-privilege process to be executed by a high-privilege process. We could attempt to perform verification by the high-privilege process that the cache matches the original code; however, guaranteeing that the verifier is both complete and free of exploitable errors is rarely achievable in practice. Thus, there would still exist a potential vector that does not exist natively for full compromise of the high-privilege process by a low-privilege process. We consider any privilege escalation risk to be unacceptable, and in our design we disallow sharing from low to high.

To enable sharing from high to low, we identify a user or set of users on the system to form a trusted computing base (TCB). On Windows a natural choice is the System user, while on UNIX it could be root or some set of capabilities if privileges are finer-grained. The set of users in the TCB will be considered equivalent with respect to privilege escalation, so clearly lower-privilege users should not be part of the TCB.

Our shared code caches form a two-level hierarchy with the TCB at the top and all other users at the bottom. The bottom-level (non-TCB) users are isolated from each other. A code cache produced by the TCB is usable by everyone, while a code cache produced by any other user is only usable by that user. On Windows, many service processes run as System, enabling significant global sharing of the code caches corresponding to common shared library code.

To avoid allowing code modifiability, we store both types of persisted cache files in protected directories writable only by the TCB. Any alternative requires full verification on each use of a code cache, which has significant performance implications (as well as requiring a verifier for the TCB caches, which we want to avoid). Even if an executable or library file is in fact writable by a lower-privilege user, we store its shareable persisted code cache in a protected directory for simplicity.

Since shareable caches are stored in protected directories, a TCB process must do the actual creation of persisted cache files. One option is to statically generate code caches containing all code contained in the native module files. However, there are two disadvantages here. The first is that such static operations suffer from the limits of static analysis: incomplete or incorrect code identification. Missing code is acceptable as the missed code will be discovered at runtime. Data incorrectly treated as code, however, while usually innocuous since never legitimately reached, is a problem for security systems that do not want to allow possible malicious execution of such non-code. The second disadvantage is that the entire module must be translated, while even the union of the code from that library used by all applications on the system is likely to be a fraction of the code in the module (see Figure 4 in Section 5.1). Limiting shared code to code that is actually executed keeps code caches small and working sets compact.

We thus use a process running as a TCB user to create each persisted code cache given information on what code was executed in a target process. We again have two design choices: we could supply only a list of starting basic block addresses to keep the input and thus the vulnerability risk narrow, or we could have each target process produce a full-fledged persistent code cache and have the TCB process verify and *publish* it. We chose the latter arrangement, with the pre-published persisted caches stored in a directory writable only by the producing user. Our design thus has a single directory of globally-shareable caches writable only by the TCB and for each user a directory of user-shareable caches writable only by the TCB plus a directory for generation of candidate caches writable by the user.

Verification involves ensuring that the code cache matches the native module code that it purports to represent, modulo translation performed by the runtime system. Publishing entails copying it to the protected directory with a name that makes it usable by other processes. We leave any merging of new code with an existing persisted cache to the process producing the new cache. While this decentralized merging combined with separate publishing can result in the loss of code if we have simultaneous production of new code from the same base cache, it does make the verifier simpler and thus further shrinks the set of security-critical code. Furthermore, we execute the publishing step in a restricted context with an inherited file handle and no other privileges, narrowing the vulnerability window further.

This cache generation scheme guarantees no privilege escalation, as nothing writable by other than the TCB is ever an input to the verifier when producing a TCB globally-shared cache. For non-TCB caches, there is a possibility of code modifiability where it did not exist before, if there is an error in the verifier. We consider this an acceptable risk, and much less serious than the risk of privilege escalation when using a verifier across privilege levels. There are often other existing vectors for executing new code at an existing privilege level given our threat model of local or remote write access.

3.3 Consistency

In addition to ensuring that code cache files have not been forged and were legitimately generated from executed application code, we must keep them synchronized with that application code. Application executables and libraries are not unchanging. Patches and security or feature updates or upgrades produce new versions on a regular basis, while local tools such as rebasing optimizers also legitimately modify module files. A persisted code cache must be invalidated if its corresponding application module differs from its state at the time of persistence. A module's stored version, checksum, and size can be used as a first order consistency check. If these are used to calculate the identifier in the shared cache namespace, then multiple simultaneous versions are naturally supported with no extra work.

A full consistency check requires a byte-by-byte comparison. In our design it is only performed offline during verification prior to publishing; at load time we use only checksums as consistency checks. In our threat model, an adversary able to maliciously modify an application module file and fool our checksums should be cause for far more worry than tricking a code caching system into executing code that the adversary can just as easily have executed natively. Consistency checks are primarily to support legitimate module changes. We use an MD5 checksum of the module code section (the only part of the module that matters to the persisted cache) that is stored in the persisted cache file and is checked versus the in-memory module at load time. As Figure 1 shows, even this checksum calculation has a noticeable performance hit. Thus, we support relaxing the checksum to include only an MD5 of the

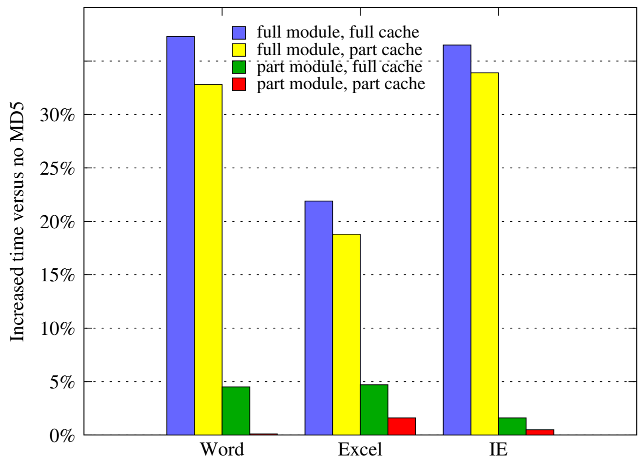

Figure 1. Overhead of consistency checks on our initialization-time benchmark set (see Section 5.2). A full module check calculates an MD5 over the entire application module code section, while a full cache check calculates an MD5 over the complete persisted cache file. Partial checks for both only consider the first and last pages. Full MD5 checks of the module are expensive; partial checks eliminate overhead and still catch all normal module update events. Full MD5 checks of the code cache file incur acceptable overhead.

first and last pages of the module, which will catch any update produced with a standard linker (but may miss a manual modification using a hex editor that does not update any of the header fields).

In addition, a persisted cache should be checked for self-consistency and completeness to guard against disk corruption or other errors. To minimize instances of incomplete files, when we produce a file we first create it as a temporary file. Only once the data is fully written and the disk cache flushed do we rename the temporary file to the official name. We also store and check an MD5 of the persisted file. Checking the entire file's MD5 incurs less than 5% overhead (Figure 1), which is acceptable.

The typical usage model of a code caching tool is to use it to control a process for its whole lifetime, where the tool can process each library as it is loaded, prior to any potential modification in memory. If instead a dynamic instrumentation system attaches to a process after process initialization, a full text section checksum comparison should be done to detect modifications of the in-memory image. This check should, however, allow legitimate library modifications performed by the loader for rebasing and rebinding.

4. Implementation

We implemented our process-shared and persistent code caches in the DynamoRIO [5] native-to-native dynamic instrumentation system. It operates in-process and executes each target application out of a code cache composed of *blocks* of single-entry, multiple-exit code sequences. DynamoRIO supports profiling and building long blocks of frequently executed code called *traces*, but this paper focuses on persisting the primary code cache, consisting of the application's dynamic basic blocks. Maintaining trace building support within persistent caches via trace head link control and using delayed trace building are beyond the scope of this paper.

This section discusses the changes and additions we had to make to DynamoRIO to support sharing and persisting its code cache. Most of the issues apply equally to process-shared or persisted code caches, so solving for one goes a long way toward supporting the other.

4.1 Data Structures

DynamoRIO provides fine-grained control over its code caches in order to support both consistency and capacity [4]. In particular, it allows unlinking (removing all incoming and outgoing jumps) and deletion of individual blocks, as well as an adaptive-region-sized cache consistency invalidation granularity. To implement such features it utilizes several types of data structures: one for each block; one for each exit from each block; one for each additional memory region that a block covers beyond its starting address, to ensure that it is invalidated on a consistency event; a list of incoming direct links to each block; and a backpointer from each code cache slot to that block's corresponding data structure.

In order to maintain the same set of data structures in a process-shared code cache, we face two challenges. The first is separating the read-only structures from the writable and placing them on different pages. As discussed in Section 2.3, we decided not to support writable shared memory due to its complexities, and thus we only consider sharing data structures on read-only pages. The second challenge is converting pointers to absolute addresses into relocatable values. This is required to support both application modules and code caches occupying different addresses in different processes. Relocated libraries are more and more frequent in modern operating systems supporting address-space randomization; and guaranteeing that the same address is available when loading a shared code cache cannot always be done.

Solving the two challenges carries secondary advantages as well: the fewer writable pages in the runtime system, the fewer opportunities for malicious or inadvertent writes to interfere with proper execution (our self-protection goal); and, if data structures are eliminated to avoid pointers and writable pages, memory is saved, further contributing to scalability.

Our solution was to create an entirely separate scheme of code cache management that operates on a coarse granularity (larger than a block of code), as opposed to the pre-existing fine-grained (per-block control) scheme. Our coarse-grain cache has no per-block data structures at all, other than an entry in a hashtable identifying the code cache entry point corresponding to the original application address. That hashtable value is an offset from the code cache start, rather than an absolute address. In order to achieve this minimalist design we gave up some of the power of the fine-grained approach: individual deletion is not supported; individual unlinking is only supported for external links (Section 4.3); blocks are not allowed to cross memory regions (they are split); and iterating over the code cache is much more expensive, but is not needed for cache management as the entire code cache for a particular module is treated as one unit that must be deleted all at once.

As finer-grained code cache deletion is needed for cache consistency-intensive applications (those with dynamically-generated or modified code) [4], we support switching from a coarse-grained code cache to a fine-grained cache for any particular module if it experiences numerous consistency events. We also support particular blocks inside a module that is primarily coarse-grained to be fine-grained; we use this power of adaptive and side-by-side granularity to ignore complexities that would make our coarse-grained strategy more difficult to achieve, allowing the fine-grained management to handle all of the corner cases. While this corner-case code will not be persisted or shared, the majority of code executed by a typical application is unchanging code residing in libraries; all of that code is coarse-grained and thus shareable and persistable in our design.

4.2 Code Transformations

In addition to changing our data structures, to support sharing and persistence we also had to modify some code transformations we normally applied. These changes only affect coarse-grain code caches: our fine-grain caches remain unchanged. They primarily involve local optimizations on the application code as it is copied into the code cache.

As each module's coarse-grain cache is treated as a single unit, we cannot elide direct unconditional jumps or calls that target separate non-coarse memory regions (such jumps are often removed by code caching systems as an optimization). Inter-module direct transfers do not normally happen natively but are enabled in our system by an optimization converting indirect calls through an import table into direct calls. Even intra-module elision is problematic: if a module contains a sub-region of code that has been written to by the application and thus been converted to fine-grained, we must keep the remaining coarse-grain code separated.

We would like our persisted code caches to be not only execution-independent but microarchitecture-independent, to allow for sharing of persisted caches across network file systems. Our system uses the underlying cache line size in two different ways: for correctness and for optimizations. For correctness, on IA-32 we must ensure that any data or code that is written without high-level synchronization must not cross cache lines. We link and unlink blocks in the code cache by writing to the four-byte operand of each jump instruction; when we emit each block into the cache we tweak the start alignment of the block and/or insert padding with `nop` instructions to ensure that those operands do not cross cache lines. We also arrange performance-critical routines like our indirect branch lookup routine and performance-critical data like our scratch space to be cache-line aligned. To produce a microarchitecture-independent persisted cache, we need to align for correctness assuming the smallest cache line size we support, but optimize assuming the largest. We store the cache line size in the persisted cache header and only use a cache if the current size lies in the range supported by that cache.

Our scratch space is accessed through segment offsets. We attempt to obtain the same offset in every process, but it is not guaranteed to be constant. We store the offset in the persisted cache header and require it to match the current offset in order to use the cache.

Finally, transformations of application call instructions result in absolute addresses in the code cache, which are problematic for module relocation, as discussed in Section 4.4.

4.3 Linking

This section describes details of how we link components of our persisted code caches together. Our coarse-grain code caches are built incrementally, in application execution order. When we decide to persist we have an opportunity to improve the layout of the code. We perform a *freezing* step prior to persisting to disk. During freezing we copy each block's successor (fall-through target) to the slot immediately after the block, in order to elide the jump instruction linking the two. This shrinks the code cache by about ten percent. We also eliminate any exit stub trampolines (which we use in un-frozen coarse units for more flexible linking) linking two blocks and use a direct jump between them. We are able to hardcode these jumps directly into our read-only frozen cache because we have given up the ability to unlink individual blocks, as discussed above.

Any exits from the frozen cache whose targets are not present must use exit stub trampolines. These stubs are kept separate from the code cache, both because they are writable and to keep the code cache more compact. A block targeting a stub reaches it via a hardcoded jump that never changes. If the block is later linked when its target materializes (as a non-frozen coarse-grain block, a fine-grain block, or a frozen coarse-grain block in a separate persisted module) the link will be routed through the stub. This is in contrast to fine-grained blocks, which can directly link to newly realized targets, as they are writable (and there are no jump reachability limitations on IA-32).

A link between persisted modules is routed through a stub at the source module, but directly targets the code cache entry point at the target module. An incoming link data structure (one list for each module) tracks these incoming jumps, enabling unlinking if one or the other of the modules is unloaded or invalidated.

A persisted module's exit stubs are kept read-only and made writable each time an exit from the code cache is linked to a new block, whether in another persisted module (which is rare: typically indirect transfers are used between modules) or in the same module but not part of the persistent cache. In steady state, once all code has been persisted, the stubs are never made writable.

Achieving efficient read-only stubs requires persisting as much of a module's code as possible, to minimize external links from code that is persisted. One common complication is a module with its import address table in the middle of two adjacent text sections, and on the same page as code. This table is normally read-only, but it is written by the loader during rebinding. We special-case this table writing to avoid converting any of the code to fine-grained mode, which is not persistable.

The target of a stub that exits the code cache, as well as an exit from a block ending in an indirect branch, are routed through special jump trampolines located adjacent to the frozen code cache. This indirection allows hardcoded jumps from the code cache to remain read-only. At load time these indirection trampolines are written just once to point at the appropriate runtime routines.

As described in Section 3.2, our code caches are shared in a two-level hierarchy: those produced by the trusted computing base (TCB) and those produced by the current user. Only code not present in the TCB cache will be found in a user cache. Exit stubs from the user cache whose targets exist in the TCB cache will be linked directly to the TCB cache at load time. As the user cache depends on a particular version of the TCB cache, it stores that version information in its header. If the TCB cache is updated, any code in the user cache that is now redundant should be removed (or it could simply be thrown out and the user cache re-persisted).

4.4 Relocation

One form of application dynamism that complicates persisted code caches is runtime code generation and modification, which we handle by reverting to our fine-grain code cache, as discussed in Section 4.1. Another form is library relocation, which is becoming more prevalent as operating systems employ address space layer randomization (ASLR) for security reasons. There are two challenges of relocation: relocating application code and relocating non-application instructions inserted by the code caching system.

In order to be successfully relocated natively, application modules must either be position-independent or contain relocation records for load-time re-basing. If the application code is position-independent, no additional work need be done by a native-to-native code caching system for those application instructions that are copied verbatim into its code cache. Unfortunately, IA-32 Windows libraries are not position independent and instead contain relocation records. A code caching system must use those records to re-base its code cache whenever a library is loaded at a different address. The re-basing process ruins sharing by writing to many of the code pages and making them process-private via copy-on-write. However, the native modules suffer from the same lack of sharing, so scalability versus native execution is not adversely affected.

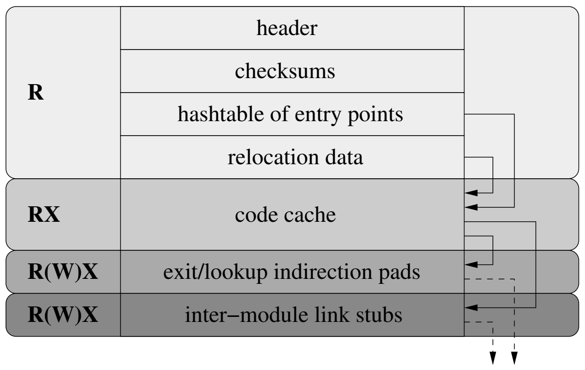

Figure 2. Our persistent code cache layout, indicating the memory attributes of each section. The indirection pads are written once at load time but are read-only afterward, while the inter-module link stubs are kept read-only but made writable each time an exit from the code cache is linked to a new block.

In addition to relocating the application instructions, any absolute application addresses stored in data structures must be updated. To avoid such updates, we do not store any absolute addresses: as explained in Section 4.1, we store offsets from the module base in our hashtable.

The second challenge is in relocating translated and inserted instructions that contain absolute addresses (an alternative is to convert the translation into position-independent code, which is not always easy to do efficiently). The most common instance is the translation of a `call` instruction into a `push immediate` followed by a `jmp` instruction. The immediate value is an absolute application address that must be relocated. Another instance is a jump out of the code cache to perform an indirect branch lookup or to exit the cache back to the runtime system proper. On IA-32, near jumps are pc-relative, making their targets dependent on the instruction location. As described in Section 4.3, we use indirection to keep each block and stub jump unchanging, leaving only the central jump trampolines to update at load time.

One final instance of absolute addresses is scratch space when it is not located in a register (or on the stack, though that is not transparent): an absolute address or a segment offset. Our solution is described in Section 4.2.

Relocation also complicates persisted cache accumulation. One approach is to consider the first persisted code to specify the canonical module base, with all later code additions being relocated prior to appending to the persisted file.

4.5 Cache Layout

Figure 2 shows the layout of our persisted caches. We chose not to use a native executable image format for simplicity. Our header contains version information both for the application module source of the code cache and for the runtime system that produced the cache, along with a section directory. Checksums are stored for consistency checks (Section 3.3). The hashtable of entry points identifies which blocks of application code are present in the code cache, while the relocation data is used when the module is loaded at other than the address at which it was persisted (Section 4.4). The two temporarily-writable sections, used for indirection and linking, were described in Section 4.3.

The cache layout was designed to be as position-independent as possible, with internal links within and among sections but all external links isolated to the two writable sections. This allows for maximum sharing among processes by keeping as much of the image read-only as possible.

4.6 Instrumentation

Dynamic instrumentation engines support building custom tools that insert instrumentation into the code cache. Persistent and shared code caches introduce two new problems: whether instrumentation should be preserved when persisting, and how instrumentation should be applied when using a persisted cache.

Inter-execution or inter-application re-use of instrumentation depends on the same tool being re-applied. Therefore the persistent cache header must indicate whether any instrumentation is present in the code cache, and if so, identify the tool and its version. The namespace of persisted code caches should include the tool identifier to support multiple simultaneous code caches for the same module but with different instrumentation. Another process (including a later instance of the same application) will only load an instrumented cache if the tool matches. As the typical tool usage model is to apply the same user-defined tool systemwide, rather than using a disparate set of tools simultaneously, tying the persisted files to the particular tool in use should work well. Tools that employ dynamically varying instrumentation will want to specify that their instrumentation should not be preserved. Finally, each tool must provide relocation information, or produce position-independent code.

With the scheme above, when a tool is executed for the first time, no persisted caches will be loaded because of a tool mismatch (the empty tool versus the present tool results in no match for an uninstrumented cache). An alternative is to change the model of inserting instrumentation and allow modification of persisted caches. Instead of changing code as it is copied into the cache, the tool would instead insert trampolines into the code cache. This is similar to the instrumentation process when modifying application code directly, without some of the pitfalls: since the code cache consists of dynamic basic blocks, all entry points are known, and each block can be padded to ensure that jump instructions can be inserted safely. For tools that do not apply systematic code translations and insert only a few calls to instrumentation routines, this model could work well and maintain sharing of most pages of the code cache.

5. Evaluation

This section presents objective evaluation of our implementation of process-shared and persistent code caches. We implemented the design described in this paper in DynamoRIO [5], an industrial-strength dynamic binary translation system targeting IA-32. The boot numbers given here are for a Lenovo desktop with a Core 2 Duo 2.2GHz processor running Windows XP with 2GB of RAM, while the desktop applications were run on a Dell Latitude D610 laptop with a Pentium M 2.2GHz processor running Windows XP with 2GB of RAM.

5.1 Scalability

Our first scalability test focuses on applying our code caching system on a systemwide basis (on every process on the system). We measured memory usage during boot and logon to our machine: we used auto-logon for automation and considered the machine fully booted once it reached an idle state.

Figure 3 shows the peak committed memory beyond native usage for each of five different configurations: fine-grain, coarse-grain without process-shared code caches, coarse-grain caches shared among all processes, coarse-grain caches shared in the two-level scheme proposed in this paper, and coarse-grain caches shared only among each user but not between users. Every Windows service and logon process was run under control of our code cache,

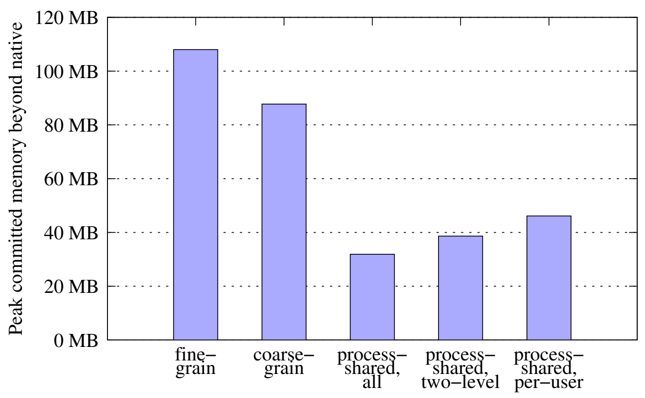

Figure 3. Peak committed memory beyond native usage during boot and logon, with 27 total processes executing from the code cache. Native usage was 135 MB. We measured five different configurations: fine-grain (without sharing), coarse-grain without sharing, (coarse-grain) caches shared among all processes, caches shared in the two-level scheme proposed in this paper, and caches shared only among each of the four users but not between users.

with a total of 27 processes executed (including some running our idle detection script) and 15 of those still running at idle time. These processes execute as four different users.

Note that the coarse-grain code cache design alone, independent of sharing, provides noticeable memory savings due to the reduction in data structures. With sharing we have significant savings. Our two-level design approaches the memory savings of unrestricted sharing, reducing memory usage by two-thirds, while minimizing any sacrifice in security.

Figure 4 displays the code cache sizes relative to native module sizes for the fifteen largest code caches from the caches shared among all users (the configuration with the largest caches) in our boot benchmark. We show the ratio considering only untranslated code as well as the translated code cache size (our code cache has an expansion ratio of about 70%, primarily from indirect branch processing code). Even sharing among 27 processes, only a fraction of the code in each module is executed: one-seventh on average.

Our second test encompasses running ten instances of a single application simultaneously, for two large desktop applications: Microsoft Internet Explorer and Microsoft Excel. As Figure 5 shows, inter-process sharing eliminates over 70% of the additional memory usage of the code caching system.

5.2 Performance

Our process-shared code cache design also supports persistent code caches. Persistence improves performance of cold application code: initialization sequences or execution of short-lived processes, where there are limited opportunities for amortization of overhead. Figure 6 shows the results on three benchmarks that start up and immediately shut down each of three typical large desktop applications: Microsoft Internet Explorer, Microsoft Excel, and Microsoft Word (we omitted Word from the ten-instances test as it is designed to never start a second process). Our benchmarks were fully automated, using the macro capabilities of Excel and Word and using Javascript with Internet Explorer in order to perform the shutdown without user input.

Figure 7 shows the time breakdown of these benchmarks. When not using persisted caches, the time copying blocks of code into

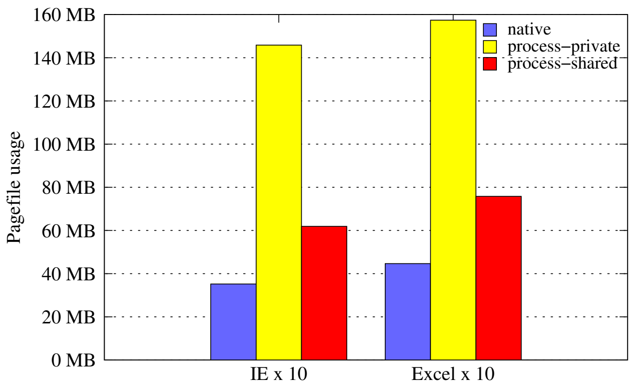

Figure 5. Pagefile usage of ten instances of Internet Explorer 6.0 and Excel 2000 processes executing simultaneously: natively, without process-shared code caches, and with process-shared code caches.

the code cache dominates execution time. Our persistent caches remove most of that time, shrinking runtime by 60% to 70%.

Generation of code cache files is a rare event compared to their use, making the performance of creating the files less important. If starting from scratch and generating dozens of new files at once, a delay of a few seconds can be measured, but that is a one-time event as subsequent runs incur zero cost. Generation can be staged to reduce the cost, but in our usage we have not felt that such action is necessary.

6. Related Work

Software code caches are found in a variety of systems. Dynamic translators use code caches to reduce translation overhead [10, 29, 40], while dynamic optimizers perform native-to-native translation and optimization using runtime information not available to the static compiler [3, 8]. Similarly, just-in-time (JIT) compilers translate from high-level languages to machine code and cache the results for future execution [1, 2, 16, 19, 35]. Instruction set emulators [11] and whole-system simulators [25, 38] use caching to amortize emulation overhead. Software code caches are also coupled with hardware support for hardware virtualization [6, 12, 37] and instruction set compatibility [14, 15, 17, 21]. 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 [5, 24, 26, 31, 32].

Studies have shown the potential benefit from re-using code caches across executions [18], which has been confirmed by at least one persistent cache implementation [27, 28]. Persistence across library re-loads but within a single execution has also been shown to improve code cache performance [23]. Even systems not utilizing full code caches can benefit from serialization of instrumentation code [36]. None of this work has explored inter-process sharing of code caches, which significantly affects the design of persisted caches.

FX!32 [9], DEC's system for IA-32 Windows migration to Alpha, combines an emulator with offline binary translation. Translated code is stored in native libraries and organized by module, similarly to our design. Security is not a high priority, and as far as we can tell a low-privilege application is allowed to produce translated code that would be used by a high-privilege application.

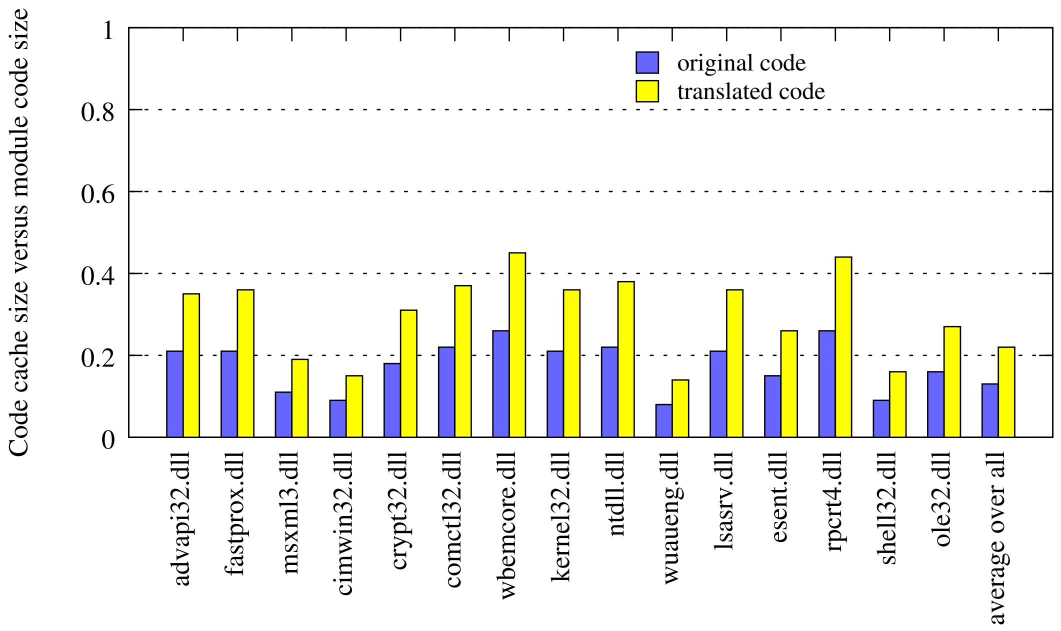

Figure 4. Code cache sizes relative to native module sizes for the fifteen largest code caches from the caches shared among all users in Figure 3, along with the average over all 206 modules. The original size is the code size prior to translation for placing into the code cache.

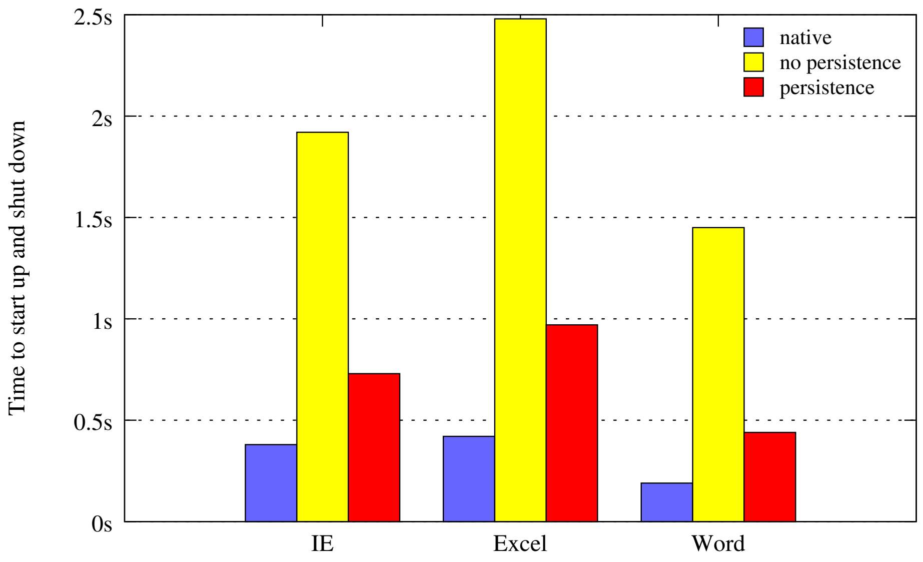

Figure 6. Time to start up and shut down Internet Explorer 6.0, Excel 2000, and Word 2000, each natively, without persistent code caches, and with persistent code caches.

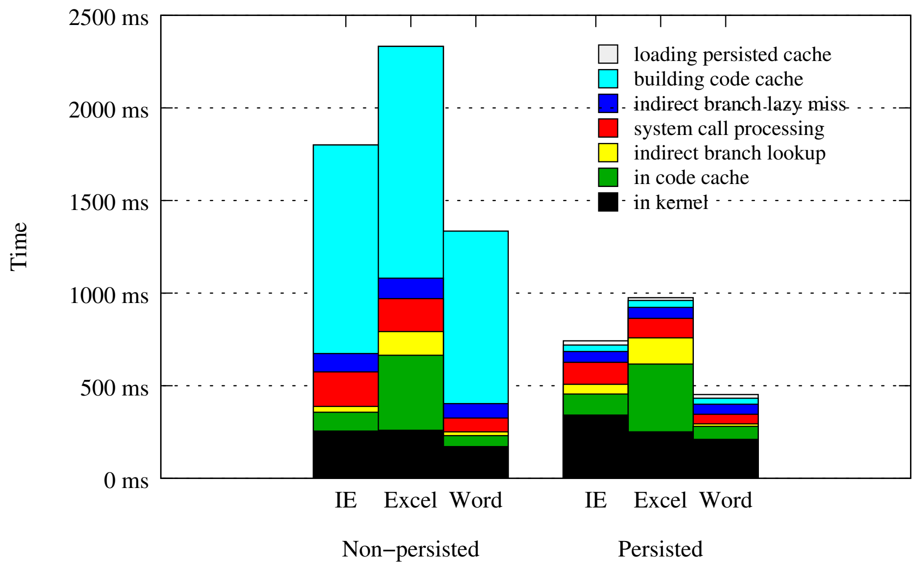

Figure 7. The breakdown of time spent when starting up and shutting down Internet Explorer 6.0, Excel 2000, and Word 2000. Persisting removes nearly all of the code cache creation time.

Transitive [29] reportedly employs process-shared code caches that are *not* persistent due to security concerns, but no published details are available.

Systems that operate below the operating system have the option of sharing code caches at the physical page level. However, it may be more practical to use virtual address tagging, as sharing across different address spaces (instead of isolating by flushing or using ASIDs) brings its own complications and costs, especially for software systems on current hardware [7].

Language virtual machines typically do not persist their JIT-compiled object code. Sharing of bytecode and other read-only information, as well as sharing of JIT-compiled code, across Java virtual machines running in separate processes has been evaluated in the absence of persistence [13].

The .NET pre-compiler *NGen* does produce native code that is persisted and shared across processes [20]. As .NET code units often have numerous dependencies, .NET 2.0 introduces a background service that tracks static dependencies and re-compiles NGen images when their dependencies change. NGen will only share code that has been cryptographically signed. If the NGen image for the code was installed into a secure directory, at load time no verification is performed; if the image is stored elsewhere, the .NET loader verifies the signature, which involves examining most of the pages in the image and usually eliminates any performance gains from persistence. A potential privilege escalation vector exists, then, if there is a bug in the installation tool that verifies signatures prior to inserting into the secure directory.

Static instrumentation tools such as ATOM [33] and Morph [39] for Alpha, Vulcan [34] and Etch [30] for IA-32, and EEL [22] for SPARC all produce persistent versions of instrumented binaries. Their disadvantages include difficulty statically discovering code as well as code expansion due to applying instrumentation to all code rather than only executed code, though Etch does attempt to address these issues by using a profiling run. HDTrans [32] evaluated static pre-translation to prime runtime code caches, but found the cost of relocation to be prohibitive.

7. Conclusions

The goal of this paper is to facilitate deployment of dynamic code caching tools on many processes simultaneously by improving scalability. We present a model for sharing software code caches among multiple processes in a secure manner. We describe the security design space and show how to prevent privilege escalation while still allowing significant sharing. Our shared code caches employ read-only code and data structures to reduce the risk of malicious or inadvertent data corruption. Our design also supports code cache persistence for improved performance during initialization or execution of short-lived processes, areas where code caches traditionally have poor performance due to limited opportunities for amortization of overhead.

We have implemented our design in the DynamoRIO industrial-strength dynamic instrumentation engine, and we show that persistent code caches achieve a 60% to 70% reduction in startup time. Our process-shared code caches increase scalability by reducing memory usage by two-thirds.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] Bungale, P. P., and Luk, C.-K. 2007. PinOS: A programmable framework for whole-system dynamic instrumentation. In *3rd International Conference on Virtual Execution Environments (VEE '07)*, 137–147.

[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] Czajkowski, G., Daynès, L., and Nystrom, N. 2002. Code sharing among virtual machines. In *16th European Conference on Object-Oriented Programming (ECOOP 2002)*, 155–177.

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

[18] Hazelwood, K., and Smith, M. D. 2003. Characterizing inter-execution and inter-application optimization persistence. In *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques*, 51–58.

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

[20] Kennedy, A., and Syme, D. 2002. Combining generics, pre-compilation and sharing between software-based processes. In *Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'04)*, 257–268.

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

[22] Larus, J., and Schnarr, E. 1995. EEL: Machine-independent executable editing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, 291–300.

[23] Li, J., Zhang, P., and Etzion, O. 2005. Module-aware translation for real-life desktop applications. In *1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*, ACM Press, New York, NY, USA, 89–99.

[24] 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.

[25] 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.

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

[27] Reddi, V. J., Connors, D., and Cohn, R. S. 2005. Persistence in dynamic code transformation systems. *SIGARCH Computer Architecture News*, 33(5) (Dec.), 69–74.

[28] Reddi, V. J., Connors, D., Cohn, R., and Smith, M. D. 2007. Persistent code caching: Exploiting code reuse across executions and applications. In *International Symposium on Code Generation and Optimization (CGO '07)*, 74–88.

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

[30] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., and Bershad, B. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *USENIX Windows NT Workshop*, 1–7.

[31] 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.

[32] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. P. 2006. HDTrans: An open source, low-level dynamic instrumentation system. In *2nd International Conference on Virtual Execution Environments (VEE '06)*, ACM Press, New York, NY, USA, 175–185.

[33] Srivastava, A., and Eustace, A. 1994. ATOM: A system for building customized program analysis tools. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, 196–205.

[34] Srivastava, A., Edwards, A., and Vo, H. 2001. Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research, Apr.

[35] Sun Microsystems. The Java HotSpot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[36] Waskiewicz, J., 2007. Dyninst object serialization/deserialization, May. `http://www.paradyn.org/PCW2007/paradyn_presentations/pdfs/jaw.pdf`.

[37] Whitaker, A., Shaw, M., and Gribble, S. 2002. Denali: Lightweight virtual machines for distributed and networked applications. In *USENIX Annual Technical Conference*, 195–209.

[38] 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.

[39] Zhang, C. X., Wang, Z., Gloy, N. C., Chen, J. B., and Smith, M. D. 1997. System support for automated profiling and optimization. In *16th ACM Symposium on Operating System Principles (SOSP '97)*, 15–26.

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