Managing Bounded Code Caches in Dynamic Binary Optimization Systems

KIM HAZELWOOD
University of Virginia
and
MICHAEL D. SMITH
Harvard University

Dynamic binary optimizers store altered copies of original program instructions in software-managed code caches in order to maximize reuse of transformed code. Code caches store code blocks that may vary in size, reference other code blocks, and carry a high replacement overhead. These unique constraints reduce the effectiveness of conventional cache management policies. Our work directly addresses these unique constraints and presents several contributions to the code-cache management problem. First, we show that evicting more than the minimum number of code blocks from the code cache results in less run-time overhead than the existing alternatives. Such *granular* evictions reduce overall execution time, as the fixed costs of invoking the eviction mechanism are amortized across multiple cache insertions. Second, a study of the ideal lifetimes of dynamically generated code blocks illustrates the benefit of a replacement algorithm based on a generational heuristic. We describe and evaluate a generational approach to code cache management that makes it easy to identify long-lived code blocks and simultaneously avoid any fragmentation because of the eviction of short-lived blocks. Finally, we present results from an implementation of our generational approach in the DynamoRIO framework and illustrate that, as dynamic optimization systems become more prevalent, effective code cache-management policies will be essential for reliable, scalable performance of modern applications.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Code generation, memory management, optimization, run-time environments

General Terms: Languages, Management, Measurement, Performance

Additional Key Words and Phrases: Dynamic optimization, dynamic translation, just-in-time compilation, code caches

1. INTRODUCTION

The software market requires applications to run on many generations of hardware. Even if software vendors tune their binaries for the most prevalent

Authors' addresses: Kim Hazelwood, University of Virginia, Charlotteville, VA 22904; Michael D. Smith, Harvard University, Cambridge, MA 02138.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1544-3566/06/0900-0263 $5.00

hardware at release time, the code will immediately be mismatched with the legacy platforms it must run on and it rapidly becomes mismatched to new platforms as hardware implementation evolves. The same market pressures force hardware developers to support legacy ISA features despite dramatic low-level technology shifts.

Software systems that alter binary files at run time are an effective response to both of the above dilemmas. Such systems can reoptimize code for the hardware on which they are actually running. They can even translate legacy machine code into code for the latest ISA. In the class of *dynamic binary optimization systems*, we thus include dynamic optimizers [Bala et al. 2000; Bruening et al. 2003; Scott et al. 2003], dynamic binary translators [Altman et al. 1999; Dehnert et al. 2003; Ebcioglu and Altman 1997; Hookway and Herdeg 1997], and dynamic binary instrumentation engines [Luk et al. 2005; Nethercote and Seward 2003].

In general, execution of an application under a dynamic binary optimizer begins with the system emulating and profiling a running program's instruction stream to track its execution flow. When the system discovers a frequently executed code sequence, the sequence is analyzed, translated, optimized, and placed in a software-managed code cache. Caching enables reuse and amortizes the costs of code transformation and optimization; therefore, these systems attempt to spend as much of the application's execution in the code cache as possible.

This paper directly addresses the management techniques that are necessary and effective to keep the majority of a program's execution in a bounded-size, software-managed code cache. As we will discuss in detail in the next section, existing code cache-management techniques cannot scale to handle the working sets of modern applications (e.g., large interactive Windows applications), and this is a major impediment to the widespread acceptance of dynamic binary optimizers. Without effective code cache management, too much of a program's execution is spent in the dynamic binary optimizer (i.e., performing overhead tasks such as emulation, code transformation, and cache management), and this overhead directly degrades the program's performance.

An effective code cache-management policy should maintain a good working set of modified code in the code cache by evicting code that is unlikely to be used in the future when asked to make room for newly translated code. However, since dynamic binary optimizers use software-managed code caches, cache-management decisions and actions cannot depend upon excessive computations or data movement, which would incur high run-time overhead. We describe two effective management techniques for bounded code caches containing dynamically generated code sequences. In particular, the novel contributions of our work are:

1. A description of the problem of code-cache management in dynamic binary optimization systems and a discussion of the issues that complicate the problem domain.

2. An exploration of a range of granular code cache eviction policies and their costs and the design of a medium-grained approach that scales well and reduces the run-time overhead of cache management.
3. The design and evaluation of a generational approach to code cache management, which reduces the run-time overhead of cache management by efficiently identifying long-lived code blocks and making it easy to evict short-lived code blocks without introducing fragmentation into the cache.
4. An implementation of our generational code cache-management technique within the DynamoRIO framework and measurements of the performance benefits.

We begin in Section 2 with a detailed look at the code cache-management problem, its unique challenges, and its prior solutions. Section 3 takes an analytical look at the number of cached blocks that should be deleted together to reduce the eviction overhead. Section 4 introduces a generational approach to code caching, where we store short-lived and long-lived blocks in separately managed code caches. Section 5 presents an implementation of generational code caches in DynamoRIO, analyzes the wall-clock performance, discusses the implementation challenges, and validates earlier simulation results. Section 6 reviews related work in the field, and Section 7 concludes.

2. SOFTWARE-MANAGED CODE CACHES

This section begins with a brief overview of the role of code caches in dynamic binary optimization systems. It then focuses on the need for code cache management in modern applications and introduces various challenges that all code cache-management approaches must address. Although the discussions and evaluations in this and the following sections are based on the design of DynamoRIO [Bruening et al. 2001, 2003], the issues and challenges are true for the design of most existing dynamic binary optimization systems with bounded code caches.

Code caches are a vital element in any dynamic binary optimization system, as they enable reuse of transformed code and, therefore-help amortize the cost of code transformation over the entire run time of a program. Figure 1 illustrates the role of the code cache in a typical binary dynamic optimizer.

Dynamic optimization systems begin by *interpreting* the executing application. Interpretation involves tracking control flow, identifying instruction boundaries, and maintaining counters to recognize when code regions have been frequently executed. In a dynamic binary translator or just-in-time compiler, interpretation also includes a translation step, where each instruction is converted to an equivalent instruction (or set of instructions) in the native instruction set.

When a taken branch is encountered during interpretation, the system in Figure 1 checks whether the branch target is already present in the code cache. Typically, this check is implemented as a hash table lookup, which maps original program addresses to code cache addresses. A hit in the hash table results in an immediate jump to the appropriate code block in the code cache. A miss results in another check. If the target code block is “hot” (i.e., frequently executed), the

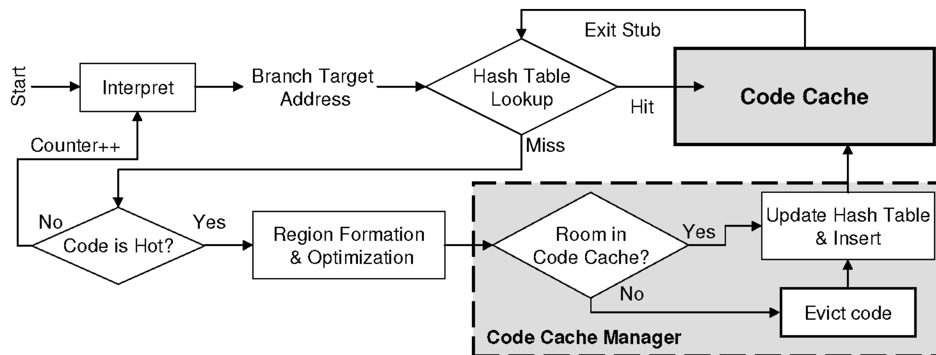


Fig. 1. Operation of a dynamic binary optimizer containing a code cache and cache manager.

system transforms the target code block through a process of region formation and optimization, and inserts it into the code cache before jumping to it. Region formation in DynamoRIO produces *superblocks*. A code cache manager is invoked whenever there is not enough room in the code cache for the newly generated superblock.

Initially, control returns to the dynamic binary optimizer from a superblock whenever execution reaches an exit point of the superblock. During region formation and optimization, the system appends a series of instructions—known as an *exit stub*—to each exit in a superblock. These instructions return control to the dynamic optimization system and pass information about the next instruction to execute. For performance reasons, most dynamic binary optimization systems attempt to patch the exit of one superblock to jump directly to the entry of another—a process known as *linking* or *chaining*. This improves performance as it avoids the overhead of frequently switching between the interpreter and the executing application.

2.1 The Need for Code Cache Management

Before moving forward, it is first important to recall the motivation for size-bounded code caches and code cache management. Some early dynamic binary optimizers avoided the need for cache management through the use of unbounded software caches (i.e., cache management is simply a check to see if there's space available, and if not, more memory is allocated for the code cache). Yet, commercial dynamic binary optimizers do need to use bounded code caches and nontrivial cache management techniques.

The widespread use of the SPEC 2000 benchmarks and the limited number of superblocks generated while executing those benchmarks implied that code cache management was not an important issue. For this benchmark suite, unbounded code caches or unsophisticated management policies such as flushing the cache when it became full, were reasonable solutions. However, this situation changes dramatically for production systems running a wider variety of software applications. Large interactive applications, for example, where every mouse movement or screen refresh translates into massive amounts of

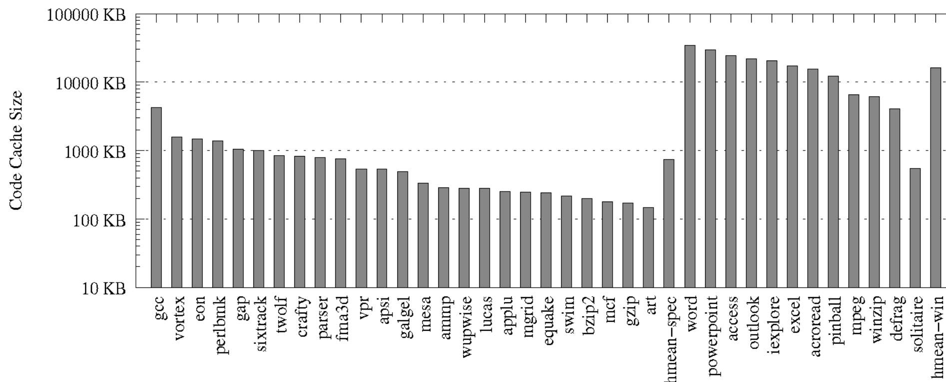


Fig. 2. Maximum cache size required the avoidance of cache management plotted on a logarithmic scale. Note that the maximum cache size of the interactive benchmarks averaged over 20 times larger than SPEC 2000.

code demonstrate the inability of these unsophisticated management schemes to scale appropriately.

Figure 2 shows the maximum code cache size reached when DynamoRIO executed various benchmarks with an unbounded cache. On average, the SPEC 2000 benchmarks resulted in a 736-KB code cache. This value is *in addition to* the size of original executable and the DynamoRIO executable. The largest of the SPEC 2000 benchmarks, `gcc`, required a 4.3-MB cache, followed by `vortex` which required a 1.6-MB cache. If we were to focus only on the SPEC 2000 benchmarks, we might conclude that `gcc` is an outlier and that most applications will fit in less than a 1-MB cache. However, as we move to the interactive benchmarks, we see a 20-fold increase in the required code cache size. On these benchmarks, the code cache averaged 16.1-MB, with the `word` benchmark requiring a 34.2-MB code cache to avoid code cache eviction. While most users today have more than 34-MB of main memory available, it is important to realize that this value is per application, and most users tend to run more than one application at a time. Furthermore, the use of main memory is slower than on-chip caches.

Our experiments show, on average, that the size of the code cache was five times larger than the amount of executed code (including system libraries) for each application. This significant code bloat occurs mainly because of the way in which DynamoRIO creates superblocks and transfers control from the code cache back to itself. Superblocks simplify optimization through tail duplication (i.e., the elimination of join points). The cost of this simplification, however, is a large amount of code duplication in any program with many conditional tests. The exit stubs that DynamoRIO uses to regain control from the code cache comprise at least three instructions each and an exit occurs roughly every six original program instructions executed.

Overall, we find that the main factor contributing to the final code cache size is the original application size, which is dramatically larger for the interactive applications. The five times expansion factor is fairly steady, with only a 59

and 111% standard deviation for the interactive and SPEC 2000 benchmarks, respectively. This implies that as the sizes of new software releases grow, unbounded code cache sizes will grow proportionately.

2.2 Challenges for Code Cache Management

The general idea of tracking and caching the current program's working set has been widely studied. Several features of code caches in dynamic binary optimization systems necessitate more new management methods which differ from those previously developed. Furthermore, because of the software-managed nature of our code caches, any time spent maintaining the cache directly impacts the performance of the executing application. In this section, we look at the unique challenges presented by software-managed code caches that the techniques we present in the later sections must overcome.

2.2.1 *Variable-Sized Superblocks.* An important property that distinguishes code caches from hardware caches concerns their contents. Hardware caches typically handle fixed-length blocks. The superblocks created by dynamic optimization systems, on the other hand, vary significantly in size, from tens of bytes to several kilobytes [Hazelwood and Smith 2004]. This wide range of superblock sizes results in significant fragmentation in the code cache under many replacement algorithms (e.g., LRU). Compaction (to remove fragmentation) is costly to perform at run time, and it is further complicated by the chaining of superblocks. We would, therefore, prefer a code cache-management technique that avoids fragmentation or simplifies the task of compaction, such as `FIFO` replacement [Hazelwood and Smith 2002].

2.2.2 *Superblock Chaining.* One of the major performance boosts in a dynamic optimizer results from superblock chaining [Bala et al. 2000; Cmelik and Keppel 1994], where off-trace branches in one cached region are patched to jump directly to their target regions in the code cache. Eviction of a superblock, however, can leave the code cache in an inconsistent state unless the link pointers used to implement chaining are updated to reflect the eviction. A common solution to this problem is to provide a side table of back pointers. Before evicting a superblock from the cache, the eviction mechanism can look in the back-pointer table to determine all other superblocks that are linked to the eviction candidate. These incoming links can be removed before proceeding with eviction. Unfortunately, this table carries run-time overhead for lookups and takes up memory that could otherwise be used for code caching. A back-pointer table is unnecessary in code caches that are managed by flushing the entire cache, as all links will be flushed along with the superblocks. All code cache-management policies except full flush must account for the performance and memory requirements of supporting link removal.

2.2.3 *Superblock Regeneration Overhead.* Servicing code cache misses differs from servicing hardware cache misses because elements stored in a code cache do not exist in their identical form anywhere else. Servicing conflict and capacity misses in the code cache requires regeneration of the previously cached

code, copying of this code into the code cache, and updating of hash tables and superblock links. The result is a significant run-time overhead: this process takes on the order of 50,000 instructions for a typical SPEC 2000 superblock in the DynamoRIO system [Hazelwood and Smith 2003]. Because of this high code-regeneration overhead, designers of dynamic optimization systems will go to great lengths to minimize code cache misses.

3. GRANULAR CODE CACHE MANAGEMENT

Code cache-management policies have been primarily limited to two extreme granularities. To make room for new superblocks in the cache, they evicted either the minimum necessary to make room for the new superblocks or they flushed the entire cache. In this section, we determine the factors that contribute to the overhead of code cache management, and learn how all of these factors come into play when determining the best granularity for evicting code from the code cache. In particular, we investigate the middle ground between evicting the minimal number of superblocks necessary and flushing the entire code cache.

3.1 Code Cache Eviction Granularity

There is an important trade-off with respect to evicting code from a code cache. Evicting a single superblock at a time will minimize the miss rate, but will lead to relatively high overhead for two reasons. First, there is some fixed overhead in invoking the eviction code, no matter how many superblocks are evicted. Evicting single superblocks leads to a high number of invocations and, therefore, a large amount of fixed overhead. Second, as we shall see, evicting single elements tends to maximize the number of dangling links that must be fixed.

At the other end of the spectrum, one can employ the cache-management scheme where the entire cache is flushed when either (a) the cache becomes full, or (b) a program phase shift is detected. This policy will greatly reduce the fixed overhead of invoking eviction and it will eliminate the need to adjust link pointers. However, flushing the cache results in noticeably higher miss rates.

In this section, we explore the spectrum of eviction granularities, including the middle ground between flushing the entire cache and eliminating a single element. We refer to this middle ground as medium-grained code cache evictions. The idea is illustrated in Figure 3. As this figure indicates, we partition the cache into *cache units* of roughly equal size, each containing several superblocks. Instead of evicting a single code block, an entire cache unit is evicted at a time, typically leaving enough room for several individual code blocks to be inserted before the code cache manager must be reinvoked.

Organizing the cache into units reduces the number of links that must be repaired during cache eviction. As Figure 4 illustrates, cache links can now be partitioned into two distinct types: *intra* and *interunit links*. Intraunit links target superblocks that reside in the same cache unit, and they do not require repair when flushing a cache unit because the dangling links will be flushed along with the target superblocks. Interunit links that target superblocks in other cache units will still require fix-up after a code cache eviction.

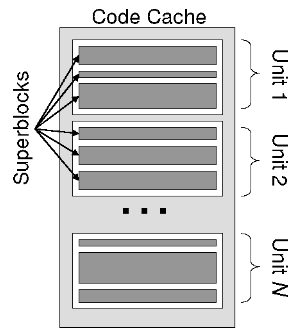


Fig. 3. Conceptual view of medium-grained cache eviction units.

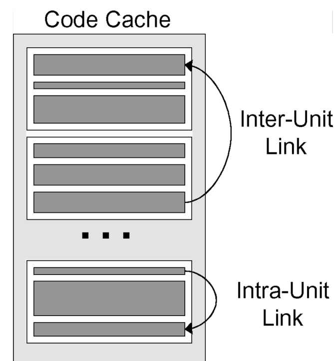


Fig. 4. Distinction between intra-unit and inter-unit links.

To better understand the trade-offs of evicting small versus large code units, we looked at the impact of eviction granularity on three important statistics that together determine the final performance of a code cache-management algorithm: *cache miss rates*, *eviction overheads*, and *link maintenance*. After briefly describing our methodology (Section 3.1.1), Sections 3.1.2–3.1.4 investigate each of these statistics independently before Section 3.1.5 presents an analysis of the overall performance.

3.1.1 *Methodology*. Our experimental setup involved the use of DynamoRIO version 0.9.3 on both the Windows and Linux operating systems. The verbose logs from DynamoRIO provided a code cache-access trace for an unbounded code cache, which was used as the input to our code cache simulator. Therefore, we were able to record the actual code regions that a code cache would manage (including region sizes and interregion links), and we were able to save and reuse the cache access logs for repeatable experiments. We also used the PAPI performance counter interface to the Pentium processors [London et al. 2001] to collect overhead estimates that were incorporated into the code cache simulator. Our overall performance estimates should be considered a lower bound in applications with phasing behavior, since simulations of a bounded code cache using an unbounded code cache trace will not take advantage of phasing behavior.

Table I. Benchmarks Used in Our Evaluation[a]

Name	Superblocks	Description
gzip	301	Compression
vpr	449	FPGA Place + Route
gcc	8751	C Compiler
mcf	158	Combinatorial Optimization
crafty	1488	Chess Game
parser	2418	Word Processing
eon	448	Computer Visualization
perlbmk	2144	PERL Language
gap	667	Group Theory Interpreter
vortex	1985	Object-Oriented Database
bzip2	224	Compression
twolf	574	Place + Route
iexplore	14846	Web Browser
outlook	13233	E-Mail App
photoshop	9434	Photo Editor
pinball	1086	3D Game Demo
powerpoint	14475	Presentation
visualstudio	7063	Development Env
winzip	3198	Compression
word	18043	Word Processor

[a]The middle column lists the number of hot superblocks that must be managed in the code cache.

We executed a combination of SPECint2000 and Windows benchmarks for this study. All 12 SPEC 2000 integer benchmarks were run on Linux. The eight interactive Windows applications (shown in Table I) were driven by manual user interaction.

For all results, we varied the eviction granularity from a full-code cache `FLUSH` down to a fine-grained `FIFO`. To ensure that the cache-management policy was truly stressed, we set the total size of the code cache for a given benchmark to *maxCache*/*cachePressureFactor*, where *maxCache* is the size that the code cache would reach for that benchmark if it were allowed to grow without bound, and $cachePressureFactor$ is the constant reduction factor used by all benchmarks.

3.1.2 *Cache Miss Rates.* Figure 5 shows that the code cache miss rate increases as the grain size for evictions increases. Each bar compares the miss rate of the specified granularity to the miss rate of `FLUSH`. Miss rates are a weighted average across all benchmarks in a benchmark family (SPEC 2000 and Windows). For the Windows benchmarks, for example, the weighted average was determined using Eq. (1):

$$unifiedMissRate = \frac{\sum_{i=\texttt{iexplore}}^{\texttt{word}} (cacheMisses_i)}{\sum_{i=\texttt{iexplore}}^{\texttt{word}} (cacheAccesses_i)} \quad (1)$$

The left-most bars in Figure 5 represent the coarsest granularity—treating the cache as a single unit and flushing it entirely. Moving to the right, the cache

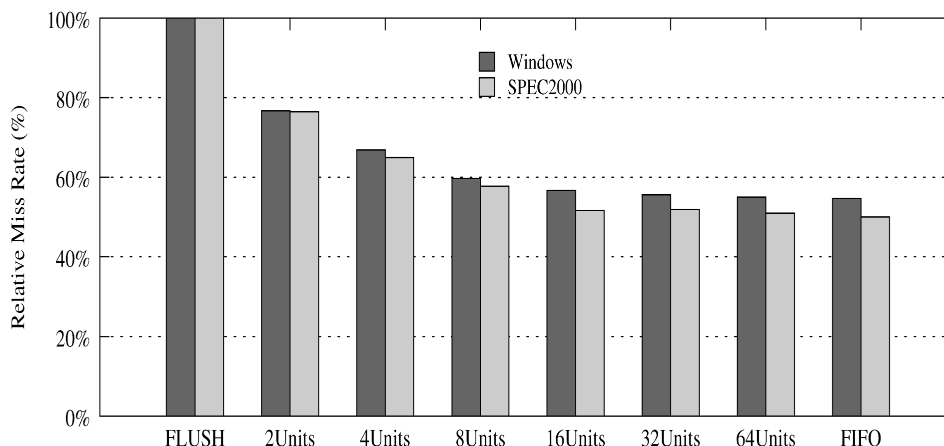


Fig. 5. Ratio of miss rates relative to FLUSH at varying granularities. Cache pressure factor is fixed at 2. This figure shows that finer-grained evictions result in lower relative miss rates.

is split into two equally sized cache units and each is flushed separately in a FIFO fashion. Finally, the rightmost bar represents the case where only enough superblocks are evicted to make room for the new one being inserted.

We used the PAPI performance counter interface to determine the actual overhead resulting from a code cache miss in the DynamoRIO framework. We collected a log of the overhead of 30,000 code cache misses in DynamoRIO, including the contributing factors of superblock generation, code cache insertion, and hash table updates. We then used a least-squares linear regression trend-line to develop Eq. (2):

$$missOverhead = 75.4 * sizeBytes + 1922 \tag{2}$$

This equation tells us the expected number of instructions required to service a code cache miss for a superblock of a given size (in bytes). The median superblock size is about 230 bytes in DynamoRIO and rarely do we encounter a superblock larger than about 500 bytes. We see from this equation that the size of the superblock plays a dominant role in the overhead of a cache miss because superblock formation itself is highly dependent on the amount of code that must be modified and cached. Servicing a cache miss for a 230-byte superblock, for example, tends to require approximately 19,000 instructions.

3.1.3 *Code Cache Evictions.* The reduction in miss rate through fine-grained cache evictions is balanced by an increase in evictions. Figure 6 shows the impact of eviction granularity on the number of times that the eviction mechanism must be invoked. The values shown in Figure 6 are relative to the number of evictions experienced by the fine-grained FIFO eviction scheme. It is interesting to note that moving from fine-grained FIFO to a slightly coarser 64-unit eviction policy increases the miss rate only slightly (Figure 5), but decreases the number of evictions by nearly a factor of 3 (Figure 6).

We repeated the process of instrumenting the DynamoRIO code to determine the overhead of a cache eviction. We collected information on the eviction size

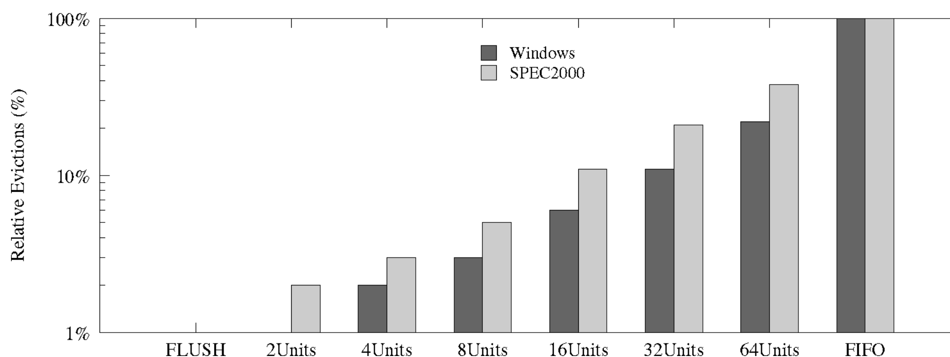


Fig. 6. Relative number of evictions at varying granularities (with respect to fine-grained FIFO eviction). Cache pressure factor is fixed at 2. The figure shows that fine-grained organizations result in more evictions.

(in bytes) and the number of instructions required to perform the eviction. This allowed us to to develop Eq. (3):

$$evictionOverhead = 2.77 * sizeBytes + 3055 \quad (3)$$

This equation tells us the average number of instructions required to evict a superblock of a given size (in bytes) from the code cache. Therefore, using Eq. (3), we see that an eviction of 230 code bytes, for example, requires approximately 3,700 instructions. We also see from the equation that the main factor contributing to the eviction overhead is the start-up cost (i.e., the constant term 3055). The components of this high cost are discussed further in Section 5. The dependence on the number of bytes evicted is a much less substantial factor. This implies that it is advantageous to evict larger blocks of code from the code cache.

3.1.4 *Superblock Link Repair.* As we described in Section 2.2, superblock chaining is a performance-critical optimization where off-trace branches from one cached superblock are patched to branch directly to their target superblocks in the code cache. To provide a concrete perspective on the benefits of superblock chaining, we executed the SPEC 2000 benchmarks under the control of DynamoRIO with and without superblock chaining enabled. We found that the slowdown resulting from disabling chaining varied from a 5 times slowdown on vpr to a 33 times slowdown on gzip. Therefore, we can safely confirm that superblock chaining is *crucial* to run-time performance and removing superblock chaining altogether is not an option.

Complications arise when we need to delete a superblock from the code cache, because we must unpatch any other superblocks that target the superblock we wish to delete. One advantage of the full-cache FLUSH mechanism is that there are no dangling links resulting from individual superblock deletions. As the code cache is flushed, all superblock links are eliminated simultaneously with the cached code.

An advantage of partitioning the code cache into several units, each of which will be flushed entirely, is that several superblock links will be eliminated as an entire cache unit is flushed. This eliminates the link repair overhead for those

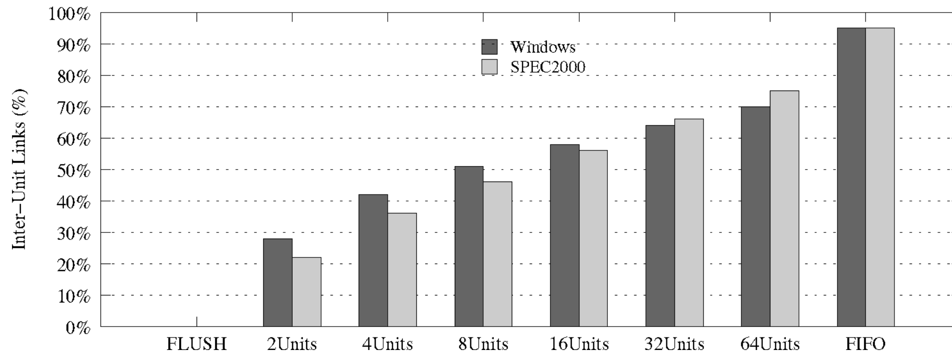


Fig. 7. Average percentage of links that target superblocks in different cache units.

intraunit superblock links (see Figure 4). Conversely, the *interunit* superblock links *will* need to be repaired and, therefore, their repair overhead must be considered.

Figure 7 shows the percentage of links that span the cache unit boundaries. There are no interunit links in the `FLUSH` scheme, because the entire cache is a single unit. However, as the cache is split into two separate units, a quarter of the links now span unit boundaries. On the right side of the figure, we note that although the `FIFO` mechanism places each individual superblock in a separate unit, not all links span unit boundaries because a superblock can link to itself (i.e., a loop backedge).

The final input in our analytical overhead study is the overhead of maintaining the state necessary to recognize and remove link pointers upon cache evictions. We inserted PAPI instruction count monitors around the DynamoRIO code that removes incoming link pointers to an eviction candidate. Equation (4) approximates the overhead of removing a given number of links pointing to an evicted superblock.

$$unlinkingOverhead = 296.5 * numLinks + 95.7 \tag{4}$$

The largest component of the overhead in Eq. (4) scales with the number of links that must be removed from an superblock. Therefore, minimizing this number is in our best interest. Since we showed that eliminating all links results in a significant decrease in performance, the best bet is to minimize the number of interunit superblock links that occur. As Figure 7 showed, this is achieved by increasing the grain size for evictions.

3.1.5 *Resulting Overhead Comparison.* We incorporated the overheads discussed previously into our code cache simulator and reran our benchmarks for each level of granular eviction. Figure 8 presents the overheads relative to the `FLUSH` mechanism. From this figure, we see a U-shaped curve that indicates that the improved miss rate of a medium-grained eviction benefits the overhead only until the overheads of superblock link repair and the number of evictions begin to dominate.

Figure 9 shows the performance of each granularity under increasing cache pressure. We see that the U-shaped curves become more pronounced as cache

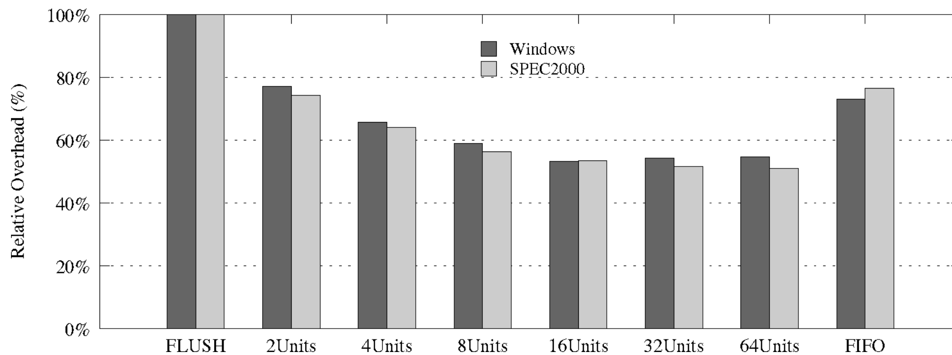


Fig. 8. Relative overhead including cache miss penalties, eviction penalties, and superblock link maintenance (with respect to FLUSH). Cache pressure is fixed at 2.

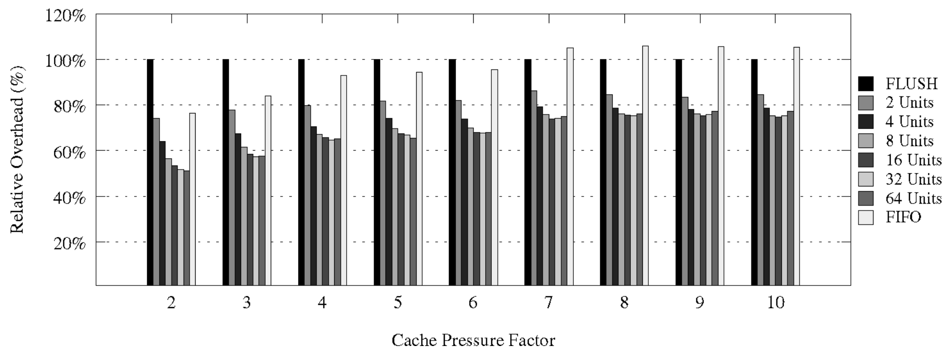


Fig. 9. Relative overhead as cache pressure increases. Results are averages of both Windows and SPEC 2000 benchmarks.

pressure increases. It is also interesting to note that the fine-grained FIFO policy starts out performing better than FLUSH, but as cache pressure increases, its performance decreases until it eventually begins to perform worse than FLUSH.

In high-cache pressure circumstances, cache-management overhead becomes a dominant factor in the overall dynamic optimization system overhead. To provide a perspective on the overhead reductions shown in Figure 9, we calculated the potential impact on final execution performance using the calculated instruction overheads, the measured cycles-per-instruction, and the processor clock frequency. With a cache pressure factor of 10, benchmarks such as *crafty* and *twolf* experience a 19.33 and 19.79% reduction in overall execution time, respectively, by simply changing the eviction granularity from FLUSH to 8-Unit FIFO. In practice, we expect large applications that stress the cache-management system to experience higher changes in execution time than others.

We can conclude that a policy where the code cache manager evicts medium-grained units from the cache can outperform both the fine-grained FIFO eviction policy and the coarse-grained FLUSH policy. Compared to a fine-grained policy, the expected increase in miss rate of medium-grained evictions is offset by a reduction in eviction invocations and reduced link-pointer maintenance. This

results in fewer code cache-management interruptions and as the estimates suggest, improved run-time performance.

3.2 Pseudocircular FIFO Replacement

The granular evictions in the previous section were designed as an extension of a standard `FIFO` (circular buffer) replacement policy [Hazelwood and Smith 2002]. For ease of simulation, the analysis above ignored two issues, *undeletable blocks* and *program-forced evictions*, which are not common, but would have to be considered in the design of a production dynamic binary optimizer. After introducing these issues, we describe an extension to the standard FIFO replacement algorithm for handling undeletable blocks and program-forced evictions, and we discuss the impact of these issues on the design of a system with granular evictions.

3.2.1 *Undeletable Blocks.* Several situations may arise that prevent a dynamic optimizer from deleting a block from the code cache. A common example is when an exception occurs in a cached block. When this occurs, execution must be suspended while the exception is handled. The suspended state may not correspond to any state in the original program and, as such, without additional mechanisms, we are unable to correlate the code cache address back to an original program address to regenerate the cached code for the exception return point. Since the exception handling code itself is also run under the control of the dynamic optimizer, it may cause insertions and evictions within the code cache. Yet, because control must return to the exception-causing code block, it is vital that it is not deleted in the interim. A discussion of mechanisms for handling precise exceptions in dynamic optimization systems is covered in great detail elsewhere [Ebcioglu and Altman 1997; Altman et al. 1999; Dehnert et al. 2003].

3.2.2 *Program-Forced Evictions.* The need for more space in a bounded code cache is not the only reason for code cache evictions. A dynamic binary optimizer should also evict (or invalidate) the code blocks that correspond to unmapped program regions (e.g., an unmapped DLL) and self-modified code. The problem with unmapped program regions is that program might map something else into the same address segment, and we don't want the old code translation to execute when control should jump to the newly mapped region.

Although typically rare, these issues warranted the design of a *pseudocircular* `FIFO` policy, which is a variant of the standard circular buffer `FIFO` scheme. From a distance, this policy behaves as a circular buffer. A simple cache pointer is maintained to keep track of the next element up for eviction. Insertion of a new superblock into the cache will evict zero or more existing superblocks (or cache units), depending on the free cache space and the size of the inserted superblock.

The pseudocircular scheme deviates from normal circular eviction when it detects an undeletable block in the list of potential eviction candidates. At this point, it instead moves the eviction pointer beyond the undeletable fragment and resumes the eviction process. The algorithm is presented in Figure 10.

```
insertNewBlock {
  free_space = leftover_space
  eviction list = NULL
  full_circle_check = FALSE
  while (free_space < new_superblock_size) {
    if (new_superblock_size > (end_of_cache - fifo_ptr)) {
      fifo_ptr = top_of_cache
      eviction list = NULL
      free_space = 0
      if (full_circle_check == TRUE) return ERROR
      full_circle_check = TRUE
    }
    while (fifo_ptr points to undeletable superblock) {
      fifo_ptr += size of undeletable superblock
      eviction list = NULL
      free_space = 0
    }
    if (fifo_ptr points to empty space) {
      free_space += size of empty space
      fifo_ptr += size of empty space
    }
    else {
      eviction list += superblock at fifo_ptr
      free_space += size of superblock at fifo_ptr
      fifo_ptr += size of superblock at fifo_ptr
    }
  }
  evict elements in eviction list
  insert new superblock in code cache
  leftover_space = (free_space - new_superblock_size)
}
```

Fig. 10. An algorithm for pseudocircular `FIFO` code cache eviction. The `insertNewBlock` routine is called when a new superblock is inserted into the cache. The variable `fifo_ptr` points to the next insertion point in the circular buffer. The algorithm is extended for granular evictions by adding cache units (rather than superblocks) to the eviction list.

Figure 10 also handles program-forced evictions, which inherently violate the circular eviction policy. While it would be possible to maintain a list of holes in the code cache, and insert into those before evicting superblocks, this complicates the cache-management design and may reduce the temporal locality benefits of `FIFO`. Instead, our solution is to add the size of any holes resulting from program-forced evictions to the size of the predecessor superblock in the code cache and then continue as normal.

The existence of undeletable blocks and program-forced evictions impacts the design of a dynamic binary optimizer with granular evictions. To avoid an infinite loop in the algorithm in Figure 10, we must have at least one more unit in the code cache than the number of outstanding exceptions possible. To avoid flushing too many unrelated superblocks with the program-forced evicted superblock, we might consider increasing the number of units beyond the best performing unit sizes in Section 3.1.

Overall, we described the factors that make up the overhead of code cache management and determined that a major contributor to cache eviction

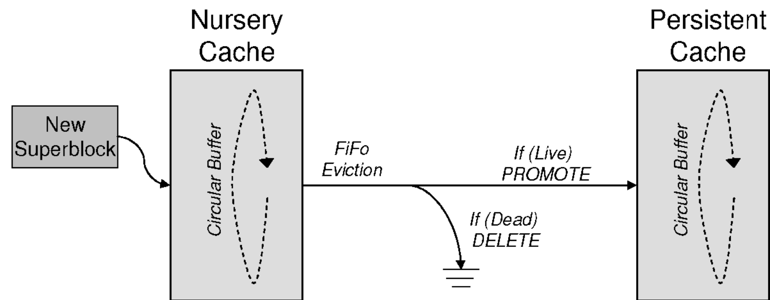


Fig. 11. Conceptual view of the generational code cache architecture.

overhead is a fixed cost to invoke the eviction mechanism, which indicated that it is advantageous to evict larger regions at a time. We also determined that implementation details of superblock chaining make a significant impact on the memory and run-time overhead of the implemented eviction policy. We concluded that medium-grained code cache evictions are the most scalable under pressure, ultimately resulting in a balance between cache miss rates, cache-management complexity, and run-time overhead. Finally, we extended the underlying circular buffer mechanism to handle the problem of undeletable code and program-forced evictions.

4. GENERATIONAL CODE CACHES

Now that we have an effective cache-management policy in place for a single code cache, we can begin to consider the structure and interaction policy between multiple code caches. Our proposal for this domain is based on superblock generations, borrowing where appropriate from the literature on generational garbage collection. Section 4.1 motivates the use of generational code caches by showing that there is a clear tendency in our benchmark suites toward superblocks living either for a short time (short-lived) or for most of the program's execution (long-lived).

Our architecture partitions the existing code cache into two distinct and separately managed regions: a *nursery cache*, which stores newly generated superblocks; and a *persistent cache*, which stores long-lived superblocks. Figure 11 illustrates this basic architecture. We assume that each code cache in the generational architecture adheres to the pseudocircular management policy described in Section 3.2. One could, of course, choose to implement a different local management policy.

Although we have leveraged some of the advances in the area of garbage collection [Appel 1989; Ungar 1984], in our approach to the code cache-management problem our domain differs in several respects. First, a code cache-management system has the added constraint that it must immediately evict certain elements from the cache, as in the case of unmapped memory, while garbage collectors can take a passive approach. Second, garbage collectors know for certain that an object is dead when it is no longer reachable from a root node. There is no such parallel in code cache management. A code object is dead when it will no longer be executed and this is generally not known

before program termination. Finally, code caches have the luxury of being able to make incorrect eviction decisions without affecting correctness (because code can be regenerated from the original copy), while a garbage collector must be much more conservative.

As in generational garbage collection, the nursery is where superblocks stay until they have reached an appropriate age for promotion to the persistent cache. In our design, an eviction from the nursery indicates that a superblock has "come of age." Section 4.2 describes the technique we use to know if we should promote the evicted superblock or delete it altogether. The decision is nontrivial because, as we will discuss later, we do not have sufficient information to determine whether the superblock evicted from the nursery will ever be referenced again. Finally, Section 4.3 presents our evaluation of generational code caching.

4.1 Superblock Lifetimes

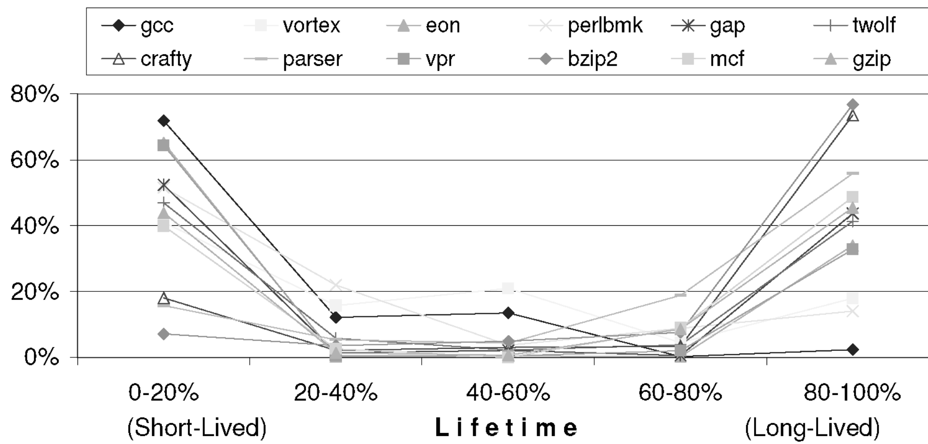
To justify the need for generational code caches, we investigate the *lifetimes* of superblocks generated during benchmark execution. A superblock lifetime is defined by Eq. (5). This equation is intended to determine the duration of time that $superblock_i$ should reside in the code cache in the ideal case, to provide a sense for the amount of code that is short- or long-lived.

$$lifetime_i = \frac{lastExecutionTime_i - firstExecutionTime_i}{totalApplicationExecutionTime} \tag{5}$$

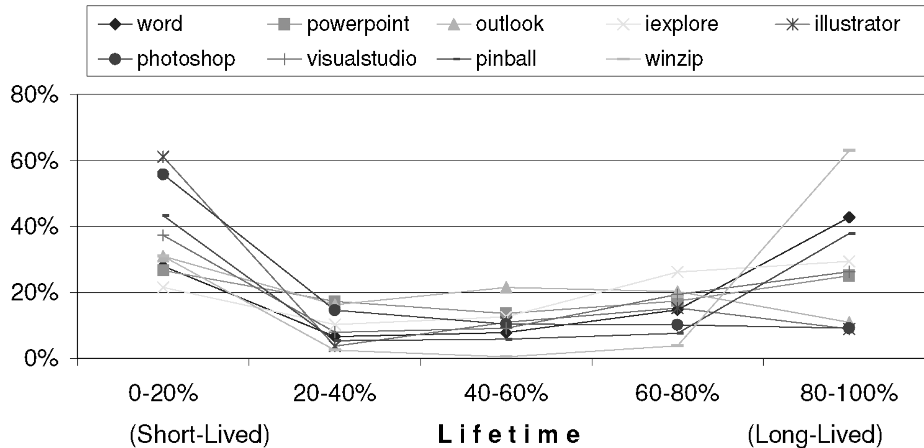
Figure 12 shows the typical lifetimes of superblocks for SPECint2000 and our Windows benchmarks, calculated using Eq. (5). Our motivating observation is that many of the frequently executed superblocks fall into one of two categories: short-lived (shown as less than 20% of the execution time) and long-lived (shown as greater than 80% of the execution time), with a relatively small percentage of superblocks falling in the middle. The fact that so many of the superblocks fall into the extreme categories indicates that it may be advantageous for us to treat these superblocks differently. The U-shaped curve of many of the lifetimes in Figure 12 implies that two separate, but equally sized, caches may work well.

One natural question that arises is whether the *important* superblocks are being maintained in the persistent cache, or stated another way, how good is *age* as a predictor for *access frequency*. Our notion of the importance of a superblock's lifetime would fail if there existed code that was executed at the beginning of a program run and then again at the end of a program run, but rarely in between. Such superblocks would be marked as *old* and would be promoted to the persistent cache, even though they were not in the working set of the application.

In reality, this does not occur. Superblocks with long lifetimes are steadily executed throughout the run-time of an application. Intuitively, this makes sense. We have a notion of *start-up code* in programs and we have a notion of *termination code*, but rarely, if ever, are these the same pieces of code.



(a) SPECint2000 lifetimes



(b) Windows application lifetimes

Fig. 12. Lifetimes of superblocks as a percentage of total execution time. The vertical axis shows the unweighted (static) percentage of superblocks that fall into each lifetime category.

4.2 The Probation Cache

An interesting design decision that arises in our generational cache architecture is the heuristic for determining whether a superblock is still alive at a given point in time and should be promoted from the nursery to the persistent cache. After investigating several alternatives, we developed one such heuristic that takes into account the features and constraints outlined earlier. Our heuristic involves the use of an intermediate *probation cache* between the nursery and persistent caches, as shown in Figure 13.

The probation cache is similar to a *victim cache* [Jouppi 1990] in microarchitecture, which is used to detect and recover from untimely evictions from instruction or data caches. The main distinction is that superblocks in the

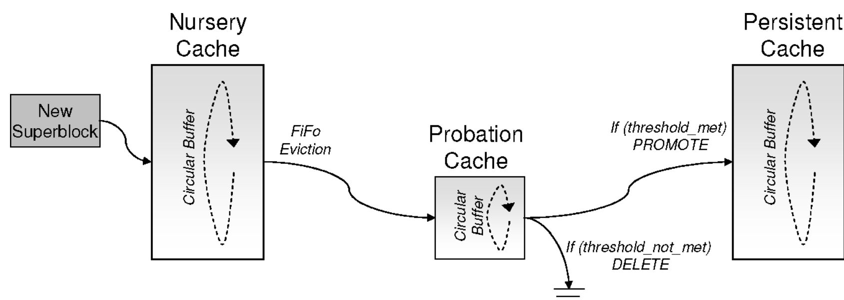


Fig. 13. Generational code cache architecture using a probation cache.

```
insertNewBlock {
  if (nursery_fifo points to valid superblock) {
    if (probation_fifo points to valid superblock) {
      if (probation_superblock.accessCount > threshold) {
        if (persistent_fifo points to valid superblock) {
           evict one or more persistent superblocks
        }
        promote probation superblock to persistent cache
      }
      else {
        evict one or more probation superblocks
      }
    }
    promote nursery superblock to probation cache
  }
  insert new superblock in nursery
}
```

Fig. 14. An algorithm for generational code cache management with a probation cache. The `insertNewBlock` routine is called when a new block is inserted into the nursery. The `*_fifo` variables point to the next insertion point in each circular buffer.

probation cache will never be reinserted into the nursery. Instead, the superblock will either be promoted to the persistent cache or deleted, as described in the algorithm in Figure 14.

The advantage of a probation cache, from an implementation perspective, is that it eliminates the need for profiling counters and comparisons in the larger nursery cache. Furthermore, it is possible to eliminate access counters in the probation cache as well by allowing each hit in the probation cache to trigger an upgrade to the persistent cache. The actual number of hits that should trigger this upgrade was experimentally determined, the results are shown in Section 4.3.1.

4.3 Evaluation

As in the previous sections, we evaluated our design using an unbounded code cache access trace from DynamoRIO to drive a generational cache simulator. Our baseline for comparison for each benchmark was a single pseudocircular cache sized such that the cache pressure factor equaled 3. We compared the miss rate of this configuration to that of multiple pseudocircular caches with a generational management policy. The caches were sized such that the total size

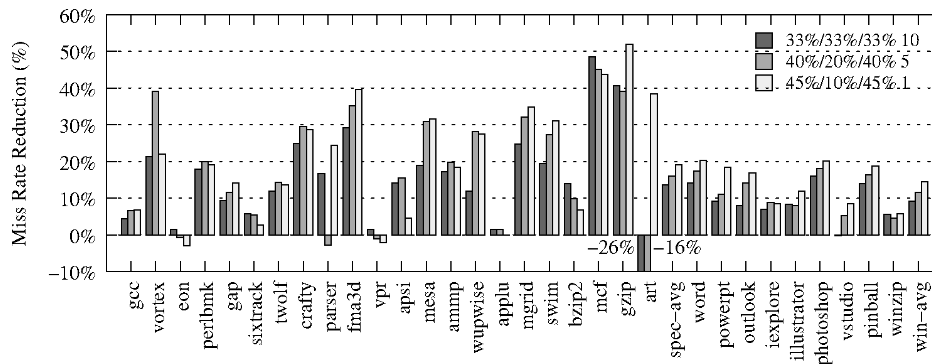


Fig. 15. Cache miss rate reduction of generational code caches over a unified cache. The three different bars are the proportions allocated to the generational caches. For example, on the left are results for a generational-caching scheme where 33% of the cache real estate is allotted to the nursery, 33% to the probatin cache, 33% to the persistent cache, and evicted superblocks with a minimum of 10 executions are promoted from the probation to the persistent cache. Average values are the unweighted arithmetic mean across all benchmarks.

of all generational caches combined (including the probation cache) equaled the size of the baseline (unified) cache.

We swept the space of generational code cache sizes to determine the cache proportions that result in the lowest miss rates for each application. While the best proportions varied by benchmark, we made two general observations.

First, there was no clear advantage to an unbalanced sizing of nursery and persistent caches that held across all benchmarks. While some benchmarks (`gcc` and `sixtrack`) preferred larger nursery caches, most of the Windows benchmarks preferred balanced caches. In fact, the superblock lifetimes in Figure 12 turned out to be a good predictor of the best cache configuration for each benchmark. In that figure, `gcc` and `sixtrack` had more of an L-shaped curve, indicating that more of their superblocks were short-lived, so larger nursery caches were better suited.

Second, we noted an undeniable link between the size of the probation cache and the promotion threshold. As we decreased the size of the probation cache, it was necessary to lower the promotion threshold. If the threshold was set too high, long-lived superblocks did not reach the promotion threshold quickly enough before they were evicted from the probation cache.

4.3.1 *Code Cache Miss Rates.* Figure 15 compares the miss rates of three instances of generational cache configurations to that of a single unified code cache. While the best cache configuration varied by benchmark, a generational cache configuration with a 45–10–45% size ratio between the nursery, probation, and persistent cache, respectively, performs best overall. Furthermore, this cache configuration performed best when we applied the policy where a single hit in the probation cache triggered a promotion. This phenomenon is particularly interesting because it simplifies the implementation and obviates the need for complex analysis to determine whether to promote a superblock to the persistent cache.

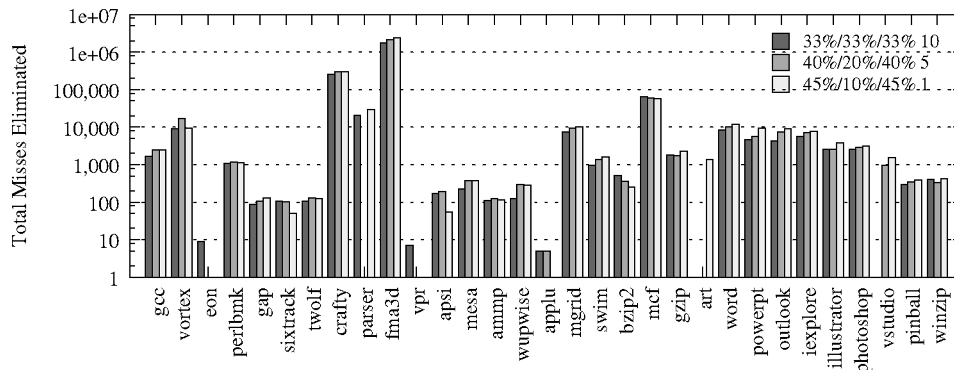


Fig. 16. Total number of cache misses eliminated using generational code caches, as compared to a single unified cache. The logarithmic vertical axis prevented us from plotting those cache miss changes that were slightly negative (`eon`, `vpr`, `art`–as shown in Figure 15).

An outlier in Figure 15 was `art`, which is also the smallest benchmark in terms of bytes of superblocks produced at run time. Since we purposely sized the caches so as not to fit the entire code footprint, this will have a greater impact on small programs that are characterized by execution within a small loop body. Interestingly, it is these very benchmarks for which cache management is least critical, because there is less of a need to apply cache management to them.

Finally, to provide perspective to the resulting miss rates, we present the actual *number* of eliminated misses in Figure 16. From this figure, we see that the miss rate reductions were often the result of eliminating many thousand code cache misses in the benchmarks.

4.3.2 *Resulting Performance.* The importance of minimizing code cache miss rates can only be appreciated when we contemplate the cost of a single code cache miss. Implementors of effective dynamic optimizers have gone to great lengths to keep execution in the code cache. All of this is done to avoid the high overhead of instruction interpretation, code generation, and superblock optimization. Avoidable misses in the code cache require us to repeat these expensive steps unnecessarily.

As in Section 3, we used the Pentium-4 performance monitors to get a realistic sense of the overhead of superblock generation, evictions, promotions, and context switches for a 45–10–45–1 configuration within the DynamoRIO framework (shown in Table II). In this configuration, there is no profiling overhead in the probation cache because every cache hit triggers a promotion. We incorporated the overheads into our generational code cache simulator.

A single avoidable miss in the superblock cache results in two DynamoRIO context switches, superblock regeneration, and link repair. Therefore, while the generational cache-management scheme includes the additional overhead of promoting superblocks to the probation and persistent caches, this carries far less overhead than early eviction and later regeneration of superblocks. In Figure 17 we show the overhead ratio between generational caches

Table II. Overheads Used in the Evaluation of Generational Code Caches

Description	Overhead (instructions)
Superblock generation	$75.4 * blockSizeBytes + 1922$
Superblock eviction	$2.77 * blockSizeBytes + 3055$
Link repair	$296.5 * numLinks + 95.7$
Superblock promotions	$13 * blockSizeBytes + 5028$
DynamoRIO context switch	25

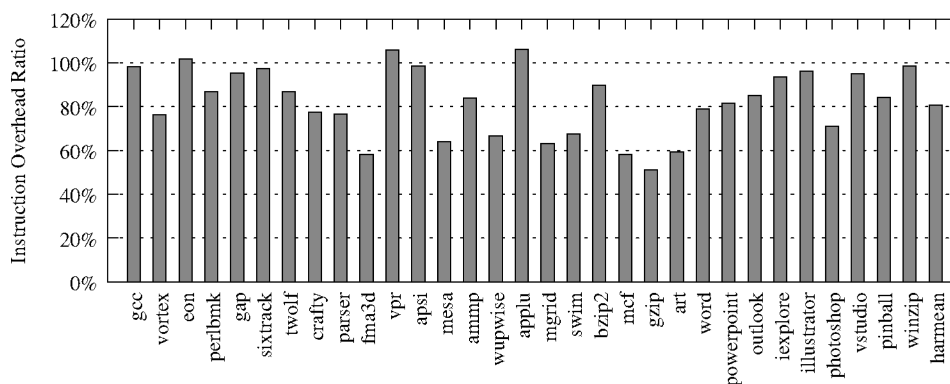


Fig. 17. Instruction overhead ratio of generational code caches to a unified cache. Smaller bars are better. Values below 100% represent a reduction in overhead. The generational caches have 45–10–45% proportions between the nursery, probation, and persistent caches.

(45–10–45–1 configuration) and a unified cache, calculated using Eq. (6).

$$overheadRatio = \frac{generationalCacheOverhead}{unifiedCacheOverhead} \tag{6}$$

Figure 17 shows that the average (geometric mean) overhead ratio is 80.7%, which represents a 19.3% reduction of instructions needed to service code cache misses. Specific benchmarks varied from a 51.1% ratio for `gzip` to a 106.2% for `applu`. Three of the benchmarks—*eon*, *vpr*, and `applu`—resulted in increases in overhead because the cache miss rate reductions were not large enough to offset the promotion overhead between the generational caches. These three benchmarks performed better with a larger probation cache, as we saw in Figure 15. We were particularly encouraged by the fact that overhead reductions were seen in all of the large Windows benchmarks in Figure 17.

To provide a perspective on the overhead reductions, we calculated the impact on final execution performance and found that it was highly dependent on the actual number of misses eliminated (which was shown in Figure 16). For `gzip` where generational cache management eliminated 2,288 misses, this resulted in an estimated 0.07% reduction in overall execution cycles. For `crafty` where 292,486 misses were eliminated, an 8.09% reduction in execution cycles is expected.

In summary, our new generational code cache-management algorithm categorizes superblocks based on their expected lifetimes and groups superblocks with similar lifetimes together in separate storage areas. This allows

short-lived superblocks to easily be removed from a code cache without introducing fragmentation and without suffering the regeneration penalties associated with evicting long-lived code superblocks. Superblock lifetimes turned out to be an effective guide for determining the best sizing of the individual caches in a generational approach. For example, applications with a high percentage of short-lived code prefer large nursery caches. For a single configuration that works well across all benchmarks, balanced caches are recommended.

5. PERFORMANCE OF A FULL IMPLEMENTATION

We now turn our attention to the performance results acquired from a direct implementation of generational code caches in an existing dynamic optimizer and a validation of our earlier simulation results. Throughout this section, we will also mention some of the more interesting implementation challenges we encountered.[1]

5.1 Performance Results

We directly implemented generational code caches in DynamoRIO following the design directions discussed in Section 4. We varied the size proportions of the three generational caches such that the sum of the sizes of the three caches were equal to the size of the unified cache to which they were compared. We also varied profile counter thresholds that were used to determine whether to promote a probation block to the persistent cache or to evict it altogether.

In this section, we report the run-time performance of each cache configuration. Run-time performance values were gathered by executing the SPECint2000 benchmarks[2] (with reference inputs) under the direct control of DynamoRIO using the official SPEC 2000 *runspec* script. The *runspec* script invokes each benchmark execution, verifies the validity of the output, and reports the official wall-clock performance results.

A fixed cache size was used for all benchmarks. This choice differs from the approach taken earlier, where code cache pressure was guaranteed by sizing the caches to be some fraction of the cache size required to avoid cache management for each benchmark. The reason for this choice was the requirement in DynamoRIO that each code cache be greater than a fixed minimum size. We also felt that fixing the code cache size for all applications would allow us to get a better sense of the cache configurations that work well for many applications. In a real implementation, it is impossible to know *a priori* the maximum cache size that will be reached during execution. Thus, it is also important to design for the common case.

We tested various configurations of generational cache sizes and compared them to a single, unified cache sized at 600 KB. Figure 18 reports the runtime performance results for the SPECint2000 benchmarks. We found that configurations with relatively large nursery caches appear to perform better overall and that the best configuration turned out to be 450–100–50–1.

[1] A full discussion of the implementation and all of its challenges can be found in Hazelwood [2004].
[2] While we would have liked to execute Windows applications, we are not currently equipped to gather repeatable performance or response time results from interactive applications.

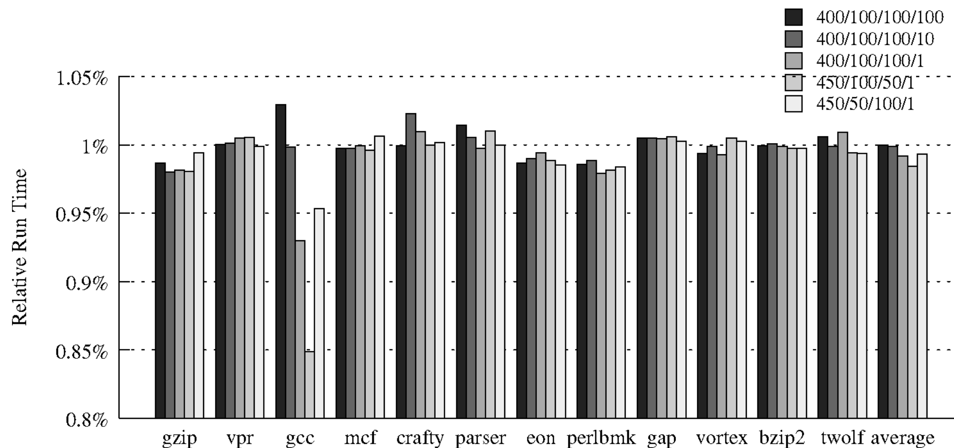


Fig. 18. Run-time performance of generational code caches for various configurations compared to the baseline performance of a unified FIFO cache sized at 600 KB. Lower numbers indicate a performance improvement using generational caches. A configuration X-Y-Z-T represents three generational code caches where the nursery cache is X KB, the probation cache is Y KB, the persistent cache is Z KB, and the threshold for promoting to the persistent cache is T executions.

While Section 4 reported that configurations that balance the nursery and persistent caches perform best, this is only true when cache pressure exists. When cache pressure is not present, as in the case of small benchmarks, the overhead of performing cache promotion is not offset by a reduced cache miss rate. This results in a net loss of performance for certain small benchmarks. In Figure 18 where the cache is sized at 600 KB, performance improvements are more evident for the *gcc* benchmark, which not coincidentally is the only benchmark that is significantly stressed at this cache size.

If we focus on the *gcc* benchmark, it soon becomes clear why the 450–100–50–1 configuration worked best (resulting in a 15% speedup over the baseline in Figure 18). This configuration dedicated the most real estate to the nursery cache. Now, if we recall the lifetime study of Section 4, we realize that most of the superblocks for the *gcc* benchmark were actually short-lived, so it makes perfect sense that a larger nursery would perform best. In fact, Figure 19 confirms that the speedup comes from all of the hits occurring in the nursery (rather than unncessarily paying for the overhead of promotion.)

This result validates our observation in Section 4 that superblock lifetimes are a good predictor of the best cache-size configurations. This result also shows how performing this study with only the SPEC 2000 benchmarks leads us to the wrong conclusion. The SPEC 2000 performance is dominated by the single *gcc* benchmark, which is unique in that it has a large number of short-lived superblocks.

5.2 Validating Simulated Miss Rates

In this section, we validate the miss rates gathered in Section 4 by collecting the actual miss rates of the generational code cache implementation in DynamoRIO. In DynamoRIO's normal execution mode, it is difficult to count

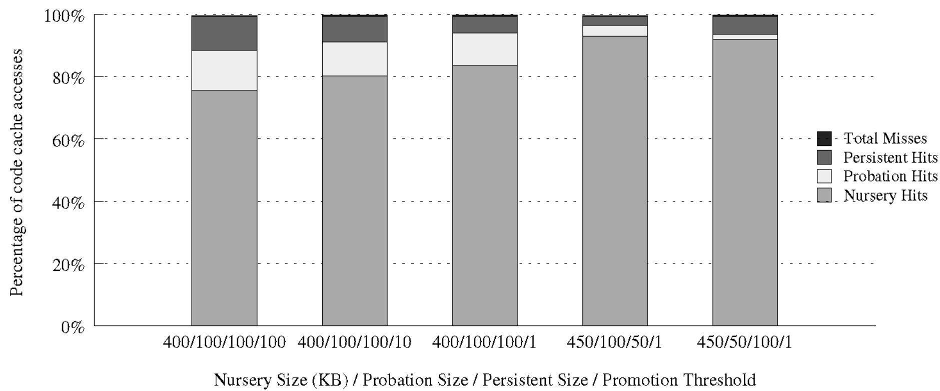


Fig. 19. Breakdown of generational cache accesses for the gcc benchmark with the total cache size set to 600 KB. Misses comprise less than 1% of cache accesses.

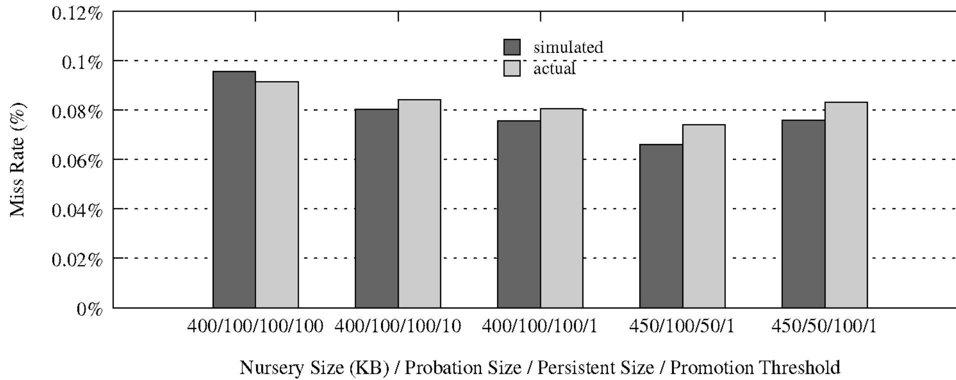


Fig. 20. This graph compares the simulated and actual miss rates of gcc for several configurations of generational code caches.

code cache hits because superblocks are chained together in the cache, such that control only exits the code cache on a cache miss. We were able to gather accurate miss rates by disabling superblock chaining within the code cache. This forced control to return to DynamoRIO after every superblock execution and enabled us to record the outcome (hit or miss) of every code cache access.

Figure 20 shows a comparison of simulated and actual miss rates for the *gcc* benchmark. In general, the resulting miss rates are fairly comparable. The small differences between simulated and actual miss rates can be explained by realizing that DynamoRIO actually forms different superblocks when the cache size changes, because of a limit on the size of formed superblocks that is expressed as a proportion of the overall code cache size. In our simulations in Section 4, we used a trace of superblocks that was fixed, regardless of the cache size. Changing the limit on a single superblock's size can result in an entirely different series of superblocks appearing in the code cache. In addition, more superblocks can fit in the code cache if individual superblocks are smaller: Therefore, the access pattern and hit rate will change. This causes our

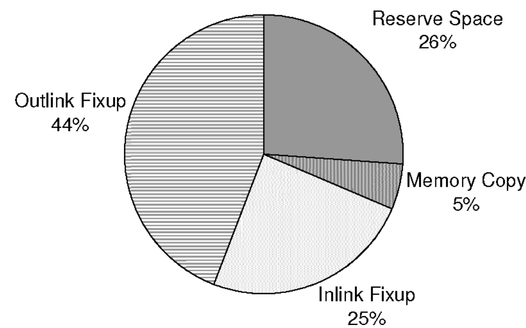


Fig. 21. Code block promotion overhead

simulator to overestimate the miss rate in some instances (the 400–100–100–100 configuration) and underestimate it in others.

5.3 Breakdown of the Overhead of Superblock Promotion

Now that we understand the final performance of a generational code cache implementation, we can investigate the factors that contribute to the overhead to get a sense of the potential benefits of careful tuning of the implementation. The next few subsections analyze the overhead experienced when a superblock is promoted from one cache to another. The breakdown is shown in Figure 21. The reported values come from analyzing several thousand different superblock promotions using the PAPI performance monitors.

5.3.1 *Reserving Code Cache Space.* We start by looking at the cost of reserving space in the code cache. This category includes the overhead of determining the number of superblocks that must be evicted to make room for the superblock entering the cache, evicting or promoting other superblocks, and updating the `FIFO` replacement pointer in the new cache. As we can see in the "Reserve Space" portion of the pie chart in Figure 21, these tasks account for (on average) 26% of the instructions executed in order to promote a superblock. A look at Table III furthermore shows that this overhead generally amounts to 869 assembly-level instructions, or just under 3000 clock cycles (per occurrence) on a Pentium-4 processor.

5.3.2 *Memory Copy Overhead.* After reserving space in the new cache, a memory copy is performed to relocate the superblock. Yet, we see from Figure 21 that copying the superblock amounts to only 5% of the promotion overhead. Regardless of the wide range of superblock sizes, the variance of memory copy overhead was very low. The number of instructions required to perform the memory copy was often identical, and the CPU cycles to perform the copy varied only slightly.

An interesting point is that of all the overheads associated with promoting superblocks, the memory copy overhead is the single overhead that is at or close to its lower bound. All other overheads are subject to implementation inefficiencies. The fact that the memory copy overhead accounts for such a small percentage of the overhead of promoting superblocks is encouraging,

Table III. Average Run-Time Overhead of Tasks Required for Generational Code Cache Management

Task	Instructions	Cycles
Reserve space	869	2782
Memory copy	173	173
Incoming link repair	886	2707
Outgoing link repair	1543	4009

because it implies that the implementation could benefit from careful fine tuning.

5.3.3 *Link Repair Overhead.* One interesting observation from the overhead analysis is the significant impact of link repair, which accounts for the majority of the overhead encountered during superblock promotion. We would expect that promoting superblocks from one code cache to another would incur a similar overhead as evicting superblocks—both actions require adjustment of all incoming link branches. Unlike eviction however, promotion also requires us to repair the outgoing links of the promoted superblock, since they are implemented as PC-relative branches.

We found great utility in maintaining data structures that represent the outgoing links within each superblock. The data structure stores pertinent information such as the addresses of the branch instructions (stored as offsets from the start address), the target of the branch, the branch type (direct, indirect), and the state of the link (linked, unlinked). Each superblock thus contains incoming and outgoing link data structures and the identical link structure on the outgoing list of one superblock will be accessible on the incoming list of the destination superblock. This allows all pertinent links to be repaired by walking the lists of the promoted superblock and obviates the need for any expensive decoding during link repair.

It is important to point out that link repair is implementation specific. The complexity of repairing interregion links is closely tied to the support available for doing so. Since DynamoRIO had little to no support for efficiently repairing outlinks, our extensions were not as streamlined as they could have been. We expect that this overhead can be reduced significantly with additional tuning.

In summary, we described and evaluated a direct implementation of generational code cache management in DynamoRIO, a mature dynamic optimization system. Using this implementation, we measured the performance advantage of generational code caching over the traditional circular `FIFO` approach found in DynamoRIO. We also analyzed the overhead of supporting generational code caches and characterized what these insights might mean for other code cache-management schemes.

6. RELATED WORK

In this section, we will discuss the space of dynamic binary optimization systems and focus on the existing techniques for code cache management. We will also briefly describe how our work relates to similar efforts in other applications of caching.

6.1 Dynamic Optimizers

A large set of dynamic optimization systems are designed to be completely transparent to the software, the hardware, as well as the user of the system. These systems are characterized by a native optimization engine that is designed to tailor the application to its run-time environment without any intervention or special preparation by the user or application writer.

Dynamo [Bala et al. 1999, 2000], from HP Labs, provides a software-managed mechanism for optimizing HP-UX applications. Dynamo cached superblocks and employed a preemptive code cache-flushing mechanism, which detected a program phase change and flushed the entire code cache at that point.

ADORE [Chen et al. 2004] is a research dynamic optimizer from the University of Minnesota. ADORE uses the Itanium processor's performance monitors to guide optimization of Itanium Processor Family (IPF) applications. Frequently executed superblocks are identified by the hardware and are optimized by the ADORE software system. ADORE limits the creation of superblocks to avoid the need to manage the code cache.

Microsoft Mojo [Chen et al. 2000] was one of the first dynamic optimizers to specifically target large, interactive Windows applications. Mojo contained separate basic block and superblock caches. Each cache was divided into two coarse-grained cache units and was fully flushed in an alternating order (i.e., FIFO order).

DynamoRIO [Bruening et al. 2001, 2003] is an infrastructure for dynamic optimization research developed by HP and MIT. DynamoRIO executes on IA-32 machines running Windows or Linux. By default, DynamoRIO implements an unbounded code cache. When a user specifies the use of a bounded code cache, DynamoRIO employs a fine-grained FIFO replacement algorithm (implemented as a circular buffer) as proposed in our prior work [Hazelwood and Smith 2002].

6.2 Dynamic Binary Translators

Dynamic binary translation is an invaluable technique for code compatibility and design space exploration. The DAISY dynamic binary translator developed at IBM [Altman et al. 2001; Ebcioglu and Altman 1997; Ebcioglu et al. 2001] allows PowerPC binaries to execute on the proprietary DAISY VLIW architecture. DAISY employs generational garbage collection for its caches, which is different from our proposed technique. IBM BOA [Altman et al. 1999] is similar framework that allows PowerPC code to execute on an EPIC processor. The Transmeta Crusoe processor [Dehnert et al. 2003] is shipped with CMS (Code-Morphing Software), which performs binary translation from IA-32 to an underlying VLIW architecture. Unfortunately, none of the code cache-management policies for these proprietary systems has been disclosed.

DELI [Desoli et al. 2002], developed by HP and ST Microelectronics, is a VLIW version of Dynamo geared toward embedded-processor applications. The system defines an API for creating and optimizing a private copy of executing code. DELI allows the user to specify when to flush the entire code cache.

Embra [Witchel and Rosenblum 1996] is a software system that uses binary translation to perform full-system simulation of a MIPS R3000. Basic blocks are stored in a translation cache that is flushed whenever self-modifying code is detected or when customized translations are desired.

Strata [Scott et al. 2003] and Walkabout [Cifuentes et al. 2002] are research infrastructures for dynamic binary translation that are specifically designed to be retargetable. Strata has been retargeted to run on SPARC, MIPS, and IA-32 architectures. Walkabout, which was based on UQDBT, has been retargeted to execute on both SPARC and IA-32 architectures. The code cache management policy in both Strata and Walkabout is to flush the entire code cache when it becomes full.

6.3 High-Level Language Virtual Machines

Many just-in-time compilers have been equipped with adaptive systems that allow code to be translated and optimized during execution. One such example is Jikes RVM [Arnold et al. 2000], which continuously monitors the performance of compiled code and generates optimized versions of the frequently executed regions. The code generated by Jikes RVM is garbage collected just like all other dynamically generated data.

6.4 Resource-Constrained Devices

Recent work has investigated the problem of caching small portions of Java programs on resource-constrained devices [Chen et al. 2003; Zhang and Krintz 2005; Zhou et al. 2005]. This problem is similar to code cache management because of the high overhead of replacing code on remote devices; however, it differs in that the unit of eviction is a fixed-size region. Most of these systems apply sophisticated cost functions to determine what to evict.

6.5 Other Work in Caching

Code caching differs from the hardware caching of instruction traces [Hinton et al. 2001] as the management of a hardware trace cache is simplified by the fixed-length, unoptimized nature of the traces. Others have performed filtering studies in the context of hardware-based caches [Ramirez et al. 2000; Rosner et al. 2001]. These filter caches [Sahuquillo and Pont 1999] resemble the nursery cache in our generational cache architecture, although they are implemented in hardware.

While most research in caching and memory management deals with fixed-size objects, a number of studies in web caching [Bahat and Makowski 2001; Wang 1999] have dealt with caching variable-sized objects. This work differs from ours in that web caches have the benefit of a backing store and idle periods (i.e., reduced Internet activity) when higher overhead management techniques can be applied.

7. CONCLUSIONS

Code cache-management policies in dynamic binary optimization systems have been primarily limited to two extreme granularities. To make room for a new superblock in a full code cache, many previous systems either evicted the

minimum number of the oldest superblocks necessary or simply flushed the entire cache. Using trace-driven simulation driven by the cache accesses and overheads of an existing dynamic optimizer, we have demonstrated the benefit of granular eviction policies and a simple generational approach to code caching.

Through a combination of simulation and analytical means, we investigated the factors that make up the overhead of code cache management and determined that the major contributor to this overhead is the fixed cost to invoke the eviction mechanism. Our findings indicated that it is advantageous to evict more than the minimum number of superblocks necessary each time, but not all of them. We also determined that implementation details of superblock chaining had a significant impact on the memory and run-time overhead of the implemented eviction policy. In the end, we found that medium-grained code cache evictions are the most scalable under pressure, ultimately resulting in a balance between cache miss rates, cache-management complexity, and run-time overhead.

Our generational code cache approach was motivated by an analysis of the lifetime of code traces residing in the code cache for various applications. We observed that many code traces were either very short- or very long-lived. Therefore, replacing a single cache with multiple generational caches can decrease the miss rates and resulting overhead of nearly all benchmarks, large or small, often significantly.

Our implementation of generational code cache management allowed direct measurement of the performance advantage of generational code caching over the standard circular FIFO approach found in DynamoRIO. Our breakdown of the run-time performance overhead of generational code caches in a dynamic optimizer presents several major contributions to the dynamic optimization community. Though the analysis of run-time overhead is specific to our implementation in DynamoRIO, we believe the findings provide insights to designers of code cache-management algorithms in general, both in terms of improving future trace-driven code cache simulators and for fine-tuning existing implementations.

Finally, in the future, we would like to explore other ideas in dynamic optimization that leverage our current work in code cache management. In particular, we are interested in integrating our techniques for code cache management with emerging mechanisms targeted at adaptation in applications with phased behavior. A perceived benefit of dynamic optimization systems is their ability to adapt to program phases and we believe that our code cache-management techniques could be successfully made to work with new and existing mechanisms for tracking and reacting to phase behavior. We are also interested in the interplay between region selection and code cache management. For example, new region selection algorithms could reduce the overhead of link management during code cache eviction by clustering the insertion of tightly linked superblocks.

REFERENCES

Altman, E. R., Gschwind, M., Sathaye, S., Kosonocky, S., Bright, A., Fritts, J., Ledak, P., Appenzeller, D., Agricola, C., and Filan, Z. 1999. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*.

ALTMAN, E., EBCIOĞLU, K., GSCHWIND, M., AND SATHAYE, S. 2001. Advances and future challenges in binary translation and optimization. *Proceedings of the IEEE*.

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19,* 2, 171–183.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 47–65.

BAHAT, O. AND MAKOWSKI, A. M. 2001. Optimal replacement policies for non-uniform cache objects with optimal eviction. In *IEEE INFOCOM Conference on Computer Communications*.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 1999. Transparent dynamic optimization. Tech. Rep. HPL-1999-77, Hewlett Packard. June.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Conference on Programming Language Design and Implementation*. 1–12.

BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. 2001. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization*. 265–275.

CHEN, G., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2003. Energy-aware code cache management for memory-constrained Java devices. In *IEEE International SOC Conference (ASIC/SOC'03)*.

CHEN, H., HSU, W.-C., LU, J., AND YEW, P.-C. 2004. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2004)*. 241–255.

CHEN, W.-K., LERNER, S., CHAIKEN, R., AND GILLIES, D. 2000. Mojo: A dynamic optimization system. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*. 81–90.

CIFUENTES, C., LEWIS, B., AND UNG, D. 2002. Walkabout - a retargetable dynamic binary translation framework. Tech. Rep. TR2002-106, Jan. Sun Microsystems Laboratories, Palo Alto.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review 22,* 1 (May), 128–137.

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 *1st International Symposium on Code Generation and Optimization*. 15–24.

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*. 257–268.

EBCIOGLU, K. AND ALTMAN, E. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th International Symposium on Computer Architecture*. 26–37.

EBCIOGLU, K., ALTMAN, E. R., GSCHWIND, M., AND SATHAYE, S. W. 2001. Dynamic binary translation and optimization. *IEEE Transactions on Computers 50,* 6, 529–548.

HAZELWOOD, K. 2004. Code cache management in dynamic optimization systems. Ph.D. thesis, Harvard University, Cambridge, MA.

HAZELWOOD, K. AND SMITH, M. D. 2002. Code cache management schemes for dynamic optimizers. In *6th Workshop on Interaction between Compilers and Computer Architectures*. 102–110.

HAZELWOOD, K. AND SMITH, M. D. 2003. Generational cache management of code traces in dynamic optimization systems. In *36th International Symposium on Microarchitecture*. San Diego, CA. 169–179.

HAZELWOOD, K. AND SMITH, J. E. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *2nd International Symposium on Code Generation and Optimization*. Palo Alto, CA, 89–99.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*.

HOOKWAY, R. J. AND HERDEG, M. A. 1997. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 3–12.

Jouppi, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th International Symposium on Computer Architecture*. 364–373.

London, K., Dongarra, J., Moore, S., Mucci, P., Seymour, K., and Spencer, T. 2001. End-user tools for application performance analysis using hardware counters. In *14th Conference on Parallel and Distributed Computing Systems*.

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 *Programming Language Design and Implementation*. Chicago, IL 190–200.

Nair, R. and Hopkins, M. 1997. Exploiting instruction level parallelism in processors by caching scheduled groups. In *24th International Symposium on Computer Architecture*. 13–25.

Nethercote, N. and Seward, J. 2003. Valgrind: A program supervision framework. In *Proceedings of the 3*rd *Workshop on Runtime Verification*. http://valgrind.kde.org/.

Ramirez, A., Larriba-Pey, J.-L., and Valero, M. 2000. Trace cache redundancy: Red & blue traces. In *6*th *International Conference on High Performance Computer Architecture*. 325–340.

Rosner, R., Mendelson, A., and Ronen, R. 2001. Filtering techniques to improve trace-cache efficiency. In *10*th *Conference on Parallel Architectures and Compilation Techniques*. 37–48.

Sahuquillo, J. and Pont, A. 1999. The filter cache: A run-time cache management approach. In *25th Euromicro Conference*. 1416–1423.

Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L. 2003. Retargetable and reconfigurable software dynamic translation. In *1st International Symposium on Code Generation and Optimization*. 36–47.

Ungar, D. 1984. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19,* 5 (May), 157–167.

Wang, J. 1999. A survey of web caching schemes for the Internet. *ACM Computer Communication Review 25,* 9, 36–46.

Witchel, E. and Rosenblum, M. 1996. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*. 68–79.

Zhang, L. and Krintz, C. 2005. The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices. *ACM Transactions on Architecture and Code Optimization 2,* 2, 131–164.

Zhou, S., Childers, B. R., and Soffa, M. L. 2005. Planning for code buffer management in distributed virtual execution environments. In *Conference on Virtual Execution Environments*.

Received August 2005; revised March 2006; accepted May 2006