# Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems

Kim Hazelwood
Div. of Engineering and Applied Sciences
Harvard University
hazelwood@eecs.harvard.edu

James E. Smith
Dept. of Electrical and Computer Engineering
University of Wisconsin – Madison
jes@ece.wisc.edu

## Abstract

*Dynamic optimization systems store optimized or translated code in a software-managed code cache in order to maximize reuse of transformed code. Code caches store superblocks that are not fixed in size, may contain links to other superblocks, and carry a high replacement overhead. These additional constraints reduce the effectiveness of conventional hardware-based cache management policies. In this paper, we explore code cache management policies that evict large blocks of code from the code cache, thus avoiding the bookkeeping overhead of managing single cache blocks. Through a combined simulation and analytical study of cache management overheads, we show that employing a medium-grained FIFO eviction policy results in an effective balance of cache management complexity and cache miss rates. Under high cache pressure the choice of medium granularity translates into a significant reduction in overall execution time versus both coarse and fine granularities.*

## 1. Introduction

Dynamic optimization systems [4, 6, 12, 20] create a modified version of a program's code image while it executes. These systems generally perform four major tasks. First, they analyze the program's instruction stream to determine the flow of execution. Second, they perform analysis, translation and/or optimization on the frequently executed code sequences. Third, they cache the transformed code to enable reuse and reduce overhead. Finally, they execute code directly from the code cache for the remainder of program execution or until the code is evicted from the code cache to make room for newly transformed code.

The benefits of dynamic optimization systems, including just-in-time compilers [2, 5], and dynamic translators [1, 8, 11, 13], range from leveraging runtime information for optimization to instruction generation for enabling mobile code. Increased instruction locality and code optimization improves steady state performance of dynamic optimizers, just-in-time compilers, and dynamic translators. Offsetting this steady state performance increase is the time required to observe runtime behavior, perform transformations, and update program code. By focusing efforts on frequently-executed regions of a program and by maximizing the amount of time it is executed directly from the code cache, the performance overhead can be minimized.

Because it is important to maximize the amount of execution time spent in the code cache, the code cache management scheme should make room for newly transformed code by evicting code from the cache that is unlikely to be used in the future. Code cache replacement strategies can vary from very fine grained (replace a single block of code at a time) to very coarse grained (flush the entire cache when it fills). Thus far, implementations have tended to use one of these extremes. Yet there exists a wide range of middle ground replacement policies that have not been explored.

In this paper, we analyze the trade-offs of the different granularities and policies for evicting blocks from a code cache. We find that the best eviction policy is one that balances code cache miss rates, eviction overheads, and design complexity. The specific contributions of this paper are:

- An exploration of medium-grained code cache eviction policies for balancing overhead, code space, and cache miss rates.
- An analysis of superblock chaining within a code cache, and a discussion of its impact on replacement policies and runtime overhead.
- A description of the issues that complicate code cache management in dynamic optimization systems, thus distinguishing the problem from other domains.

The remainder of the paper is organized as follows. Section 2 defines dynamic optimization systems, describes several existing implementations, provides an overview of the problem of code cache management, and discusses solutions implemented in several existing systems. Section 3 describes several issues that distinguish code cache manage-

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

ment from hardware cache management and paging. Section 4 identifies components that contribute to code cache management overhead, and evaluates the impact of eviction granularity on that overhead. Section 5 analyzes the trade-offs of superblock chaining and incorporates this overhead into our evaluation, and Section 6 concludes.

## 2. Background

In this section, we describe several software systems for dynamic optimization, dynamic translation, and just-in-time compilation. We also describe the common problem of code cache management and discuss some of the existing solutions to the problem.

### 2.1. Existing Systems

The goal of most dynamic optimizers is performance improvement, though they also show promise in the areas of program introspection, sandboxing, and security [18]. Dynamic translators focus on providing code compatibility with new or existing architectures. Finally, just-in-time compilers support platform-independent code. Although these systems have a variety of goals, adequate performance is a requirement for all such systems, and this performance generally comes from code optimization and caching.

**Dynamic Optimizers** Dynamo [4] is a system developed at Hewlett-Packard Laboratories that provides a software-based mechanism for selecting and optimizing blocks of HP-UX instructions. Several successors to Dynamo have since been developed, including DELI and DynamoRIO. DELI [12] is a VLIW version of Dynamo geared toward embedded-processor applications that was developed by Hewlett-Packard in conjunction with ST Microelectronics. DynamoRIO [6] is a dynamic optimization research infrastructure developed as a collaboration between Hewlett-Packard and MIT. DynamoRIO executes on the IA-32 architecture in both Windows and Linux. Wiggins/Redstone [10], developed at Compaq, is a dynamic specialization system that applies data-specific and machine-specific optimizations to Alpha binaries at runtime. Finally, Mojo [7], developed at Microsoft, is a dynamic optimizer that was one of the first infrastructures to specifically target large, interactive Windows applications.

**Dynamic Translators** Several dynamic translation systems have been developed to provide code compatibility for new or existing architectures. FX!32 [16], developed by Digital, allows IA-32 binaries to execute on Alpha processors. Although FX!32 does not perform dynamic optimization in a strict sense—it optimizes between program runs—it has many features of a true dynamic optimization system. The DAISY [13] dynamic translator was developed at IBM and allows PowerPC binaries to execute on the DAISY VLIW architecture. A follow-up project at IBM is the BOA framework [1] which allows PowerPC code to execute on a VLIW/EPIC processor. Finally, the Transmeta Crusoe processor [11] is shipped with CMS (Code-Morphing Software) which performs binary translation from IA-32 to an underlying proprietary VLIW architecture.

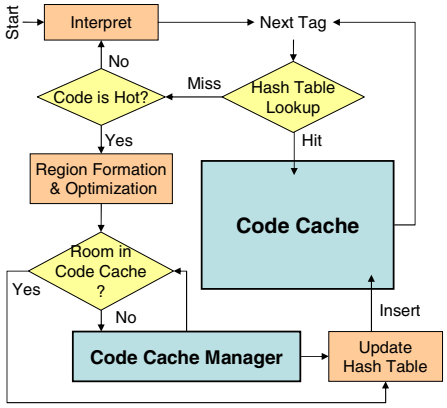


**Figure 1. Control flow of dynamic optimization systems including a code cache and cache management system.**

**Just-In-Time Compilers and Adaptive Systems** Many just-in-time compilers have been equipped with adaptive systems that allow code to be re-optimized during execution. For example, Jikes RVM [2] is a publicly-available Java Virtual Machine developed at IBM Research.

### 2.2. Overview of Code Cache Management

Code caches are a vital element in any dynamic optimization system, as they enable reuse of translated/optimized code, and therefore help amortize the cost of code modification over the entire program execution. Figure 1 illustrates the role of the code cache, as well as code block lookup and execution in a typical dynamic optimization system. Generally, execution begins with an interpretation step until the start of a basic block or method is detected. At this point, the hash table (which maps original PCs to PCs of transformed code blocks) is accessed. A hit in the hash table results in an immediate jump to the previously transformed block in the code cache. A table miss for a "hot" PC value results in transformation of the code block (translation and/or optimization), insertion into the code cache, and finally a jump into the code cache to execute the new code. The code cache is limited in size, therefore in order to make room for a new block, a code cache manager is necessary to provide the eviction policy

for the code cache. Eviction policies for modern systems are discussed in Section 2.3.

While Figure 1 assumes a single code cache structure, some implementations consist of multiple code caches. DynamoRIO, for example, includes two code caches. A *basic-block cache* stores all single-entry, single-exit regions that have been encountered during execution, which allows DynamoRIO to avoid the high overhead of interpretation during every execution of a basic block. Once a basic block's execution count exceeds a *hotness threshold* the system combines basic blocks to form superblocks (single-entry, multiple-exit regions) [17] that are stored in a separate code cache. This idea has been extended to support multiple superblock code caches that are distinguished by the lifetimes of the superblocks they contain [15].

### 2.3. Existing Code Cache Policies

The simplest code cache management policy is not to manage evictions at all, and to allow code caches to grow without bound. This policy is not feasible in a realistic system of course. Recent research [15] has shown that:

- Code caches tend to grow up to five times the size of the executed application code, and
- Modern interactive Windows applications often execute over 40 MB of code during the course of a few minutes of execution.

By combining these findings with the observation that users tend to execute several programs at once, we can conclude that code cache sizes are likely to be a limitation, especially as application sizes increase. Therefore, several code cache management policies for bounded caches have been implemented in recent systems.

Dynamo employed a preemptive flushing mechanism [3] for cache management, which detected a program phase change and flushed the entire code cache at that point. This policy was found to perform better than a naïve flush of the cache when it became full. The cache replacement policy in the publicly available version of DynamoRIO defaults to an unbounded code cache—an acceptable feature in a research tool. Using an environment variable, a user may impose a cache size limit, and in this case, DynamoRIO employs a fine-grained FIFO replacement algorithm, with the code cache being implemented as a circular buffer similar to that proposed in an earlier study [14]. While code cache management is not explicitly discussed in the publication on DELI [12], the authors do mention facilities for controlling the timing of full code cache flushes, which indicates that fine-grained cache management may not be implemented in DELI. Finally, in Mojo [7], the superblock cache was divided into two coarse-grained cache units. Each unit was fully flushed in an alternating order (i.e. FIFO).

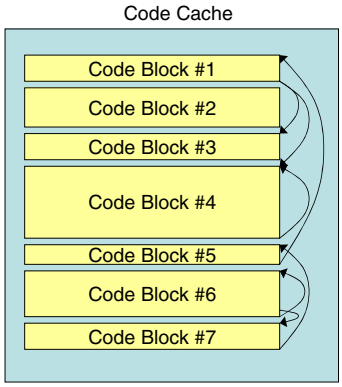


**Figure 2. Code cache with superblock links.**

Other related work includes a study of code cache management by Hazelwood and M. Smith [14]. In that work, several code cache management schemes were investigated, and it was found that exploiting temporal locality is important, and implementing a code cache as a circular buffer balances the issues of miss rate and fragmentation. Our research reported here extends that earlier work by exploring eviction policies over a range of granularities, analyzing and discussing the important issue of superblock chaining, and extending the evaluation to include interactive applications.

## 3. Code Cache Management Issues

Several features of code caches necessitate more sophisticated management methods than those used in hardware instruction or data caches. In this section, we describe the specific features of code caches and explain why hardware cache solutions are less effective in this domain.

### 3.1. Superblock Chaining

One of the major performance boosts in a dynamic optimization system results from *superblock chaining*[1] [4, 9]. This technique allows execution to remain inside the code cache by chaining together superblocks that execute in succession. Good performance is achieved because it is not necessary to return control to the dispatch system to determine the next superblock to execute. Superblock chaining is supported by patching jump instructions to the ends of superblocks residing inside the code cache. Figure 2 depicts a code cache containing links between elements.

Eviction of superblocks can result in dangling link pointers unless there is a method for determining and eliminating incoming links. For example, to evict code block #4 from the code cache in Figure 2, there must be a way to determine

[1] We refer to the overall process as "chaining", and to individual pointers as "links".

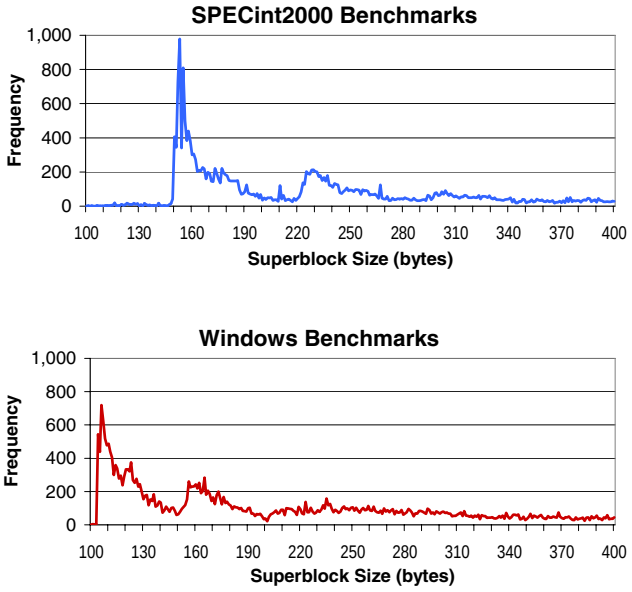


**Figure 3. Size distribution of superblocks in SPECint2000 and Windows benchmarks.**

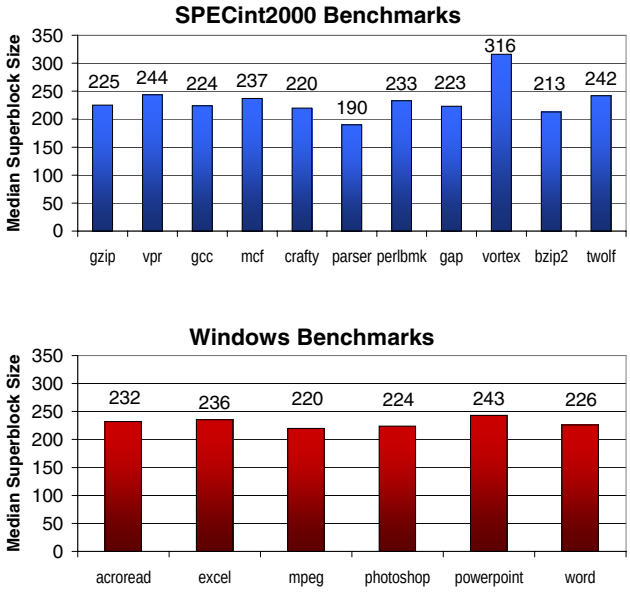


**Figure 4. Median superblock size (in bytes) of SPECint2000 and Windows benchmarks.**

that code block #1 is currently linked to code block #4, so the link can be removed before evicting code block #4. Otherwise, code block #1 will contain a dangling link pointer, which will result in incorrect program execution. Therefore, for any cache management policy to ensure code correctness, it must provide support for removing any incoming link pointers to eviction candidates.

A common solution to this problem is to provide a side table of back pointers. Before evicting a code block from the cache, the eviction mechanism can look in the back-pointer table to determine all other code blocks that are linked to the eviction candidate. These incoming links can be removed before proceeding with eviction. Unfortunately, this table carries runtime 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 implement a full flush eviction policy, as all links will be flushed along with the code blocks. Therefore, code cache management policies must take into account the performance and memory requirements of supporting link removal.

### 3.2. Superblock Regeneration Overhead

Servicing code cache misses is different from servicing hardware cache misses because elements stored in a code cache do not exist in their identical form anywhere else; that is, there is no backing store. Servicing a code cache miss requires regeneration of the cached code, which carries a very high overhead. The specific steps that must be completed before execution of the source program can resume are:

1. Save the processor state of the running program
2. Re-optimize/re-translate the next sequence of instructions from the source program
3. Store the altered instructions in the code cache
4. Update any hash tables and links
5. Restore the processor state of the executing program

Studies have shown that this process takes on the order of 50,000 instructions for a typical SPEC2000 superblock in the DynamoRIO system [15]. Due to this high code regeneration overhead, designers of dynamic optimization systems go to great lengths to minimize code cache misses.

### 3.3. Variable-Size Cache Entries

An important property of code caches that distinguishes them from conventional hardware caches is that the cached elements vary in size rather being fixed-length blocks. As Figure 3 illustrates, the size of superblocks stored in the code cache varies significantly. In fact, Figure 4 shows that the median superblock size varies—often significantly—between individual benchmarks.

Variable superblock sizes mean that an LRU or an LRU-like eviction algorithm would lead to internal fragmentation

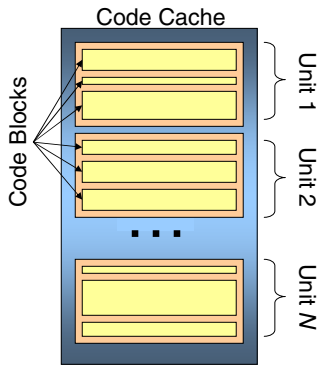


**Figure 5. Conceptual view of medium-grained code cache eviction units.**

in the code cache. To make matters worse, compaction (to remove fragmentation) would require adjusting all the link pointers. Consequently, in the following sections, we focus on FIFO algorithms, which, with circular buffer code cache implementations, do not lead to internal fragmentation. Furthermore, FIFO replacement policies are widely implemented in modern code caches [14, 15].

## 4. Exploring Granularities

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 will lead 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 via a back-pointer table.

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) when 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 will result in noticeably higher miss rates, as was shown in prior work [14].

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, and we illustrate the idea in Figure 5. As this figure indicates, the cache is partitioned into *cache units* of equal size, each containing several code blocks. Now, instead of evicting a single code block, an entire cache unit is evicted at a time, leaving enough room for several individual code blocks to be inserted before the code cache manager must be re-invoked.

### 4.1. Experimental Approach

To better understand the trade-offs of evicting small vs. large code regions, we studied the impact of eviction granularity on cache miss rates and eviction overheads.

Our experimental setup involved the use of the DynamoRIO dynamic optimization system and a code cache simulator. We used both the Windows and Linux versions of DynamoRIO release 0.93. The Windows environment was Windows 2000 Server and the Linux environment was Red Hat version 7.2. We used the PAPI [19] performance counter interface to the Pentium processors to collect the overhead estimates. In addition, we used the verbose output from DynamoRIO to drive the code cache simulator; therefore we were able to represent the actual code regions that a code cache would manage including actual region sizes and inter-region links. We were able to save and reuse the DynamoRIO logs to allow for repeatability in the experiments.

Table 1 lists the benchmarks used, including all 12 SPECint2000 benchmarks which were run on Linux and eight interactive Windows applications that were driven by

**Table 1. Benchmarks used in our evaluation. The middle column lists the number of hot superblocks that must be managed in the code cache.**

| 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 |

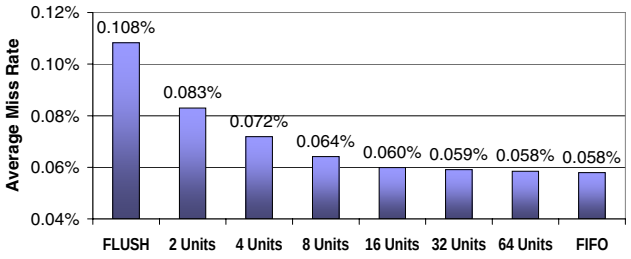


**Figure 6. Miss rates at varying granularities. Cache pressure fixed at 2.**

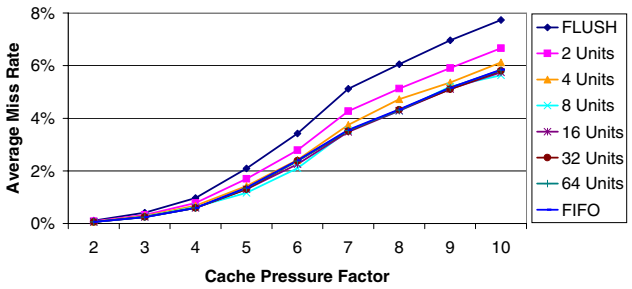


**Figure 7. Miss rates at varying granularities as cache pressure increases.**

manual user interaction performing everyday tasks. Interactive applications are particularly important in code cache studies because prior work [15] has shown that the rate and amount of generated code in these applications tests the limits of code cache management systems.

We focused on the cache management policy for managing frequently-executed superblocks. In DynamoRIO, a superblock is considered *hot* when it has been executed 50 times. The middle column of Table 1 lists the total number of hot superblocks produced during execution of each benchmark; this is the total number of superblocks that must be managed by the code cache at runtime.

### 4.2. Effect of Granularity on Cache Miss Rates

In practice, the size of a dynamic optimizer's code cache is generally fixed for all applications. The effect is that code cache management strategies result in bimodal cache performance. For applications that fit within the code cache, the choice of code cache management policy makes no difference. However, for applications that do not fit within the code cache, performance can suffer precipitously. We are most interested in studying code cache management behavior when it is being stressed, i.e. when the program working set exceeds the size of the code cache. Therefore, for all results in this paper, the code cache is sized to ensure that the replacement mechanism will be stressed.

For all results, the code cache eviction granularity was varied from a full code cache flush down to the finest-grained FIFO eviction policy, which evicts only enough superblocks to make room for the inserted superblock. To ensure code cache pressure, the size of the entire code cache was set to be $maxCache/n$, where $maxCache$ is the size that the code cache would reach if it was allowed to grow without bound for the particular benchmark, and $n$ is a *cache pressure factor* that we impose in order to ensure that the cache management policy is truly stressed. The $maxCache$ term ranges from 171 KB for the smallest benchmark—`gzip`—where 301 superblocks were cached, to 34.2 MB for the largest benchmark—`word`—where 18,043 superblocks must be cached. For our studies, we vary the cache pressure factor from 2 to 10.

A weighted average was used to calculate a single cache miss rate across all benchmarks. The weighted average was determined using Equation 1.

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

As was suggested earlier, the code cache miss rate is expected to increase as the grain size for evictions increases. This was verified using the code cache simulator, and the results are shown in Figure 6 for a code cache sized at $maxCache/2$. The leftmost bar represents the coarsest granularity possible—treating the cache as a single unit and flushing it entirely. Moving to the right, the cache 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 each individual superblock is treated as a single eviction unit, and only enough superblocks are evicted to make room for the new one being inserted. As the figure illustrates, miss rates decline as the cache evictions become more fine grained.

Figure 7 then shows how the miss rate of each eviction granularity scales under pressure. As the figure indicates, the differences in miss rates become much more pronounced as cache pressure increases. However, miss rates tell only part of the story.

### 4.3. Effect of Granularity on Code Cache Evictions

Reduction in miss rate through fine-grained cache evictions is balanced by the increased overhead of performing more evictions. Figure 8 shows the impact of eviction granularity on the number of times that the eviction mechanism

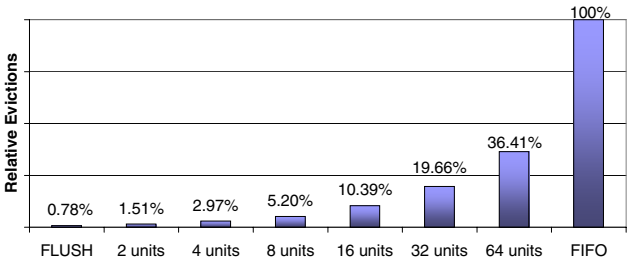


**Figure 8. Relative number of evictions compared to finest-grained FIFO eviction.**

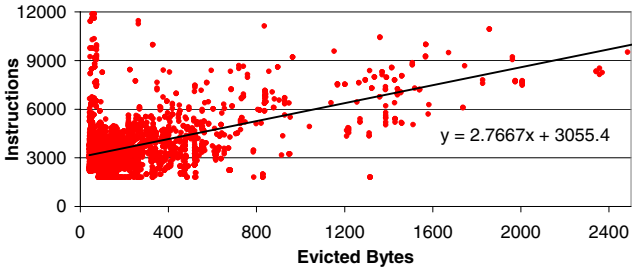


**Figure 9. Overhead, measured in instruction count, for code cache evictions.**

must be invoked. The values shown in Figure 8 are relative percentages with the baseline set to the number of times that eviction would be necessary had a fine-grained, minimum-element eviction scheme been implemented. It is interesting to note that moving from the finest-grained FIFO to a slightly coarser 64-unit eviction policy increases the miss rate only slightly (Figure 6) but the number of evictions decreases by nearly a factor of 3 compared to the fine-grained implementation (Figure 8).

Next, we used the PAPI performance counter interface [19] to study the actual overhead of code cache evictions in the DynamoRIO framework. We collected a log of over 10,000 code cache evictions, including their eviction size (in bytes) and the number of instructions required to perform the eviction. We then used a least-squares linear regression trendline (illustrated in Figure 9) to develop Equation 2.

$$evictionOverhead = 2.77 * sizeBytes + 3055 \quad (2)$$

This equation tells us the average number of instructions required to evict a superblock of a given size (in bytes) from the code cache. An eviction of 230 bytes of code, for example, would require 3,690 instructions. Interestingly, Equation 2 shows that the main factor contributing to the overhead of evictions is the start-up cost (i.e. the constant term 3055 in the equation) and the dependence on the number of bytes evicted is a much less substantial portion of the overhead. This implies that it is advantageous to evict larger blocks of code from the code cache.

We then repeated the process of inserting PAPI instruction counters to determine the expected overhead of a cache miss. We did so by collecting data on superblock generation, code cache insertion, and hash table updates as they occurred in DynamoRIO. Using this methodology, we determined the overhead estimate shown in Equation 3.

$$missOverhead = 75.4 * sizeBytes + 1922 \quad (3)$$

This equation tells us the expected overhead of a code cache miss for a superblock of a given size (in bytes). We see from this equation that the size of the superblock plays a much larger role in the overhead of a cache miss (as compared to Equation 2) 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, therefore, tends to require 19,264 instructions.

By combining the instruction penalty of eviction (Equation 2) with the instruction penalty of a cache miss (Equation 3), we can begin to understand the overhead trade-offs of code cache eviction granularity.

### 4.4. Effect of Granularity on Overhead

We introduced the notion of *overhead penalties* to the code cache simulator and re-executed the benchmarks to study the trade-offs encountered as the code cache granularity is varied. We used the overhead penalties defined in Equations 2–3, and limited the maximum code cache size to $maxCache/10$ as described in Section 4.2.

The results are shown in Figure 10 normalized to the coarsest-grained policy (`FLUSH`). This figure exhibits several interesting trends. The eviction policies on the far left perform worst because their high code cache miss rates are not offset by the reduction in the number of evictions. Moving to finer-grained evictions, there are reductions in overhead as a result of the improvement in hit rate. However, moving to the finest-grained eviction policies results in an increased overhead due to frequent invocations of the cache eviction mechanism.

Figure 11 shows how the overhead of cache eviction scales as cache pressure increases from a factor of 2 up to a factor of 10. Again, all results are relative to the coarsest-grained `FLUSH` mechanism. It's interesting to note that the finest-grained policy starts out performing better than `FLUSH`, but as cache pressure increases, its performance decreases until it eventually begins to perform worse.

Several observations can be made at this point. First, there is clearly a delicate balance between reducing cache misses and reducing eviction overhead. While fine-grained

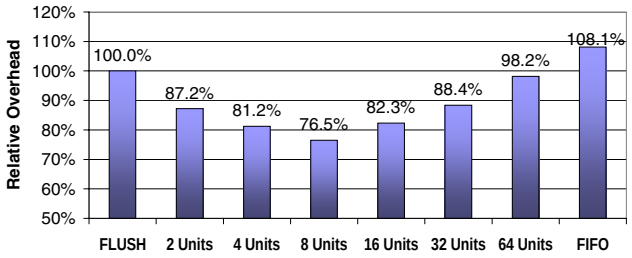


**Figure 10. Relative overhead of various code cache eviction granularities including miss rate overhead and eviction overhead. The code cache is sized at $maxCache/10$.**

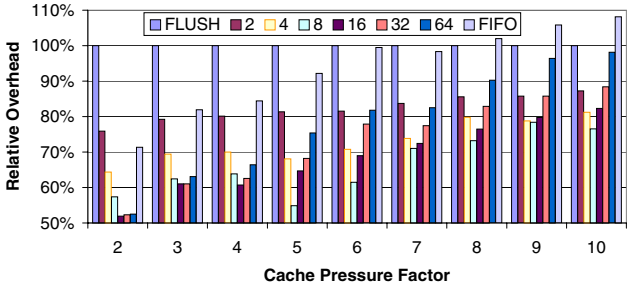


**Figure 11. Relative overhead of eviction granularities as cache pressure increases.**

cache eviction minimizes the miss rate, it also maximizes the number of evictions that must occur. Second, while the finest-grained eviction policy performs much better than a coarse-grained flush under low cache pressure, the scenario reverses under high cache pressure. Therefore, at this point our results indicate the the most robust and scalable eviction policy is to aim for medium-grained evictions.

There is an additional factor that must be considered before making concrete conclusions about the benefits of various code cache eviction granularities, however. Superblock chaining—as described in Section 3.1—is a subtle feature, but one that could result in a significant difference in the complexity of the eviction mechanism, which could skew the resulting runtime overhead.

## 5. Analysis of Superblock Chaining

As discussed in Section 3.1, one of the advantages 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. Therefore, there is no need to maintain a table of link back-pointers. This eliminates both the memory required for the back-pointer table and the overhead to maintain it. Yet, in Section 4.4, we found the miss rate of the full flush mechanism is higher than with other granularities.

Since we are exploring partitioning the code cache into several units, each of which will be flushed entirely, we can still achieve some of the superblock chaining benefits of the full flush mechanism. Several superblock links will be eliminated as an entire cache unit is flushed, therefore back pointers are unnecessary for those *intra-unit superblock links*—links that connect two code blocks residing in the same cache unit. However, there are two options when dealing with *inter-unit superblock links*—those that span cache unit boundaries. We can provide a back-pointer table exclusively for those inter-unit links, or we can eliminate inter-unit chaining from the code cache management system altogether.

### 5.1. Characteristics of Superblock Links

In this section, we begin by investigating the feasibility of eliminating inter-unit links (links that span the boundaries of a cache unit and therefore must be removed when the unit is flushed.) Intuitively, there are two scenarios where it would be feasible to eliminate inter-unit superblock links. The first situation would occur if superblock chaining was not highly beneficial to runtime overhead. In this case, the elimination of the back-pointer table and its subsequent maintenance overhead would outweigh the benefits of chaining. The second situation would occur if there turned out to be a very small number of inter-unit superblock links, such that even if the benefits of chaining were high, the case of inter-unit superblock chaining was so rare that it made sense to eliminate the memory and runtime overhead of maintaining back-pointer tables.

We begin by providing perspective on the memory footprint required to support back-pointer tables. We collected the average number of outbound links from each cached superblock and show the results in Figure 12. As the figure indicates, there are an average of 1.7 links originating from each superblock. Combining this with the observation that each back pointer requires roughly 16 bytes of memory[2], we can determine that the memory overhead of a complete back-pointer table is generally 11.5% the size of the code cache.

Next, we provide a concrete perspective on the benefits of superblock chaining. To this end, we executed the SPEC2000 benchmarks[3] under the control of DynamoRIO with and without superblock chaining enabled on a dual-

[2]In a linked list–an 8-byte pointer and an 8-byte link.

[3]Interactive application results were not included in this study. We were not equipped to measure the important interactive metric–response time.

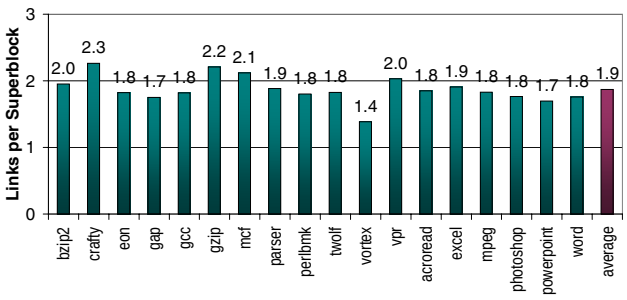


**Figure 12. Average number of outbound links originating from each superblock.**

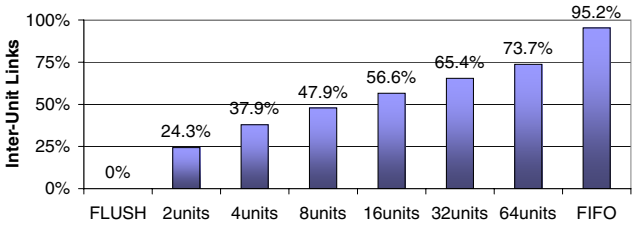


**Figure 13. Percentage of superblock links that target superblocks in different cache units.**

Xeon 2.4 GHz machine with 1 GB RAM. The results, shown in Table 2, are quite compelling. While the median execution time of `gzip` with chaining enabled was 230 seconds for the reference inputs, disabling chaining resulted in a execution time of 7,951 seconds—a dramatic 3357% slowdown. The cost is not in the hash table lookup but is caused by the memory protection changes (and associated system calls) that the DynamoRIO system does in order to protect the translation manager from the user code. In systems where this is not necessary, the slowdown is reduced, but is still significant.

At this point, the first scenario has been eliminated from our feasibility study on removing back pointers—we have shown that superblock chaining is *crucial* to runtime performance, and removing superblock chaining altogether is not an option.

**Table 2. Slowdown resulting from disabling superblock chaining. Benchmarks were executed under the control of DynamoRIO on Linux.**

| Benchmark | Linking Enabled | Linking Disabled | Slowdown |
|---|---|---|---|
| gzip | 230 sec | 7951 sec | 3357% |
| vpr | 333 sec | 2474 sec | 643% |
| gcc | 206 sec | 3284 sec | 1494% |
| mcf | 368 sec | 2014 sec | 447% |
| crafty | 215 sec | 3547 sec | 1550% |
| parser | 350 sec | 6795 sec | 1841% |
| perlbmk | 336 sec | 6945 sec | 1967% |
| gap | 195 sec | 4231 sec | 2070% |
| vortex | 382 sec | 4655 sec | 1119% |
| bzip2 | 287 sec | 4294 sec | 1396% |
| twolf | 658 sec | 6490 sec | 886% |

## 5.2. Interconnectivity of Superblock Links

The previous section took an extreme approach and determined the impact of eliminating all superblock links from the dynamic optimizer's code cache in order to provide a perspective on the impact of superblock chaining. What is really important for our investigations, however, is to determine the impact of removing only the *inter-unit* superblock links, as the contrasting *intra-unit* superblock links will be eliminated upon flushing the unit and therefore are not problematic.

Figure 13 shows the percentage of links that span the cache unit boundaries. There are no inter-unit links in the `FLUSH` scheme because the entire cache is a single unit. However, as the cache is split into two separate units, 24.3% 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). This now removes the second scenario from the back-pointer elimination feasibility study—we have shown that even with only two cache units, a non-trivial number of links span the unit boundary. Now that we have determined that a back-pointer table is unavoidable in all but the case of the full flush mechanism, we extend the analytical overhead study to include the overhead of maintaining a back-pointer table and removing link pointers upon cache evictions.

Using once again the methodology from Section 4.3, 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.

$$unlinkingOverhead = 296.5 * numLinks + 95.7 \quad (4)$$

We note that a large part of the overhead in Equation 4 can be directly attributed the to the number of links that must be removed from an eviction candidate. Therefore, minimizing this number is in our best interest. Since we showed that

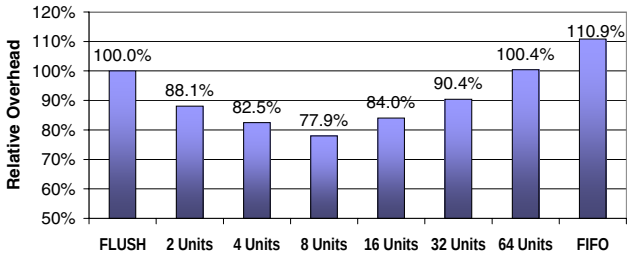


**Figure 14. Relative overhead including cache miss penalties, eviction penalties, and superblock link maintenance. Cache size set to $maxCache/10$.**

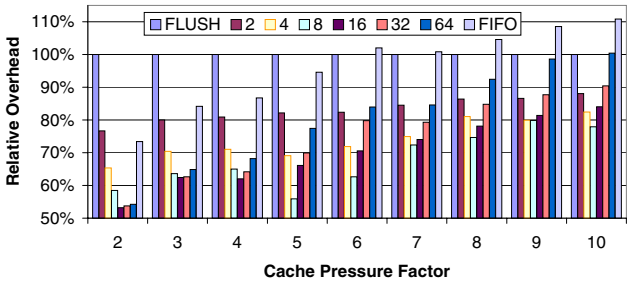


**Figure 15. Relative overhead (including link maintenance) as cache pressure increases.**

eliminating links results in a significant decrease in performance, our best bet is to minimize the number of inter-unit superblock links that occur. As Figure 13 showed, this is achieved by increasing the grain size for evictions.

### 5.3. Resulting Impact on Overhead

We introduced superblock link maintenance overhead (shown in Equation 4) to the code cache simulator and re-executed the benchmarks. For each cache eviction, the system was penalized for every inter-unit link that required removal. This resulted in the final overhead estimates, shown in Figure 14. One observation from this figure is that the overheads of all of the finer-grained policies have moved closer to `FLUSH` as a result of inter-unit superblock links. This is a result of the penalty for maintaining inter-unit links that is unnecessary in `FLUSH`. The largest changes occurred in the finer-grained policies, as they contained more inter-unit links than the coarser-grained policies.

Figure 15 shows the performance of each granularity under increasing cache pressure. Again we see the same trend where fine-grained `FIFO` starts out performing better than `FLUSH`, but the situation reverses as pressure increases. The main difference between Figure 15 and the earlier Figure 11 is that the introduction of link removal increased the overhead of all policies except `FLUSH`.

In high cache pressure circumstances, the overhead of cache management becomes a dominant factor in the overhead of a dynamic optimization system. To provide a perspective on the overhead reductions shown in Figures 14 and 15, we calculated the potential impact on final execution performance, using the calculated instruction overheads, the measured CPI, 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 applications that stress the cache management system to experience higher changes in execution time than others.

### 5.4. Discussion

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 a smaller number of code cache management interruptions, a smaller back-pointer table for maintaining link pointers, and as the estimates suggest, improved runtime performance. These results bring us one step closer to the end goal of designing robust, scalable dynamic optimization systems by understanding the inputs and trade-offs involved in choosing a code cache eviction policy.

While the actual trade-offs may vary slightly by system implementation, we feel that the insight gained from this work will be useful for dynamic optimization system designers as they consider the options for code cache management and recognize that they should not limit their options to the extreme eviction granularities.

Currently, we advocate treating the cache as a circular buffer and inserting elements in FIFO order to leverage the temporal locality of the code cache accesses. Our future work includes a more detailed analysis and visualization of the interconnectivity of superblocks within the cache. This study will help us to determine whether a better method exists for determining the placement of superblocks into the cache units to minimize inter-unit superblock links while still achieving low miss rates.

Other future work includes an investigation of a cache management strategy that dynamically adjusts the eviction granularity on-the-fly, based on the perceived cache pressure.

## 6. Conclusions

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. Using trace-driven simulation driven by the cache accesses and overheads of an existing dynamic optimizer, we explored several intermediate granularities of code cache evictions.

Through a combination of simulation and analytical means, we investigated the factors that make up the overhead of code cache management. We determined that the major contributor to cache eviction overhead is a fixed cost to invoke the eviction mechanism, which indicated that it is advantageous to evict larger regions from the code cache at a time. We also determined that implementation details of superblock chaining make a significant impact on the memory and runtime overhead of the implemented eviction policy. In the end, 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 runtime overhead.

## 7 Acknowledgments

We wish to acknowledge HP and MIT for granting us access to DynamoRIO. We also thank Glenn Holloway, Michael D. Smith, and the anonymous reviewers for their thorough and constructive reviews of this paper. This research was funded by NSF grant CCR-0311361, a Harvard fellowship, and grants from Intel, IBM, and Microsoft.

## References

[1] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, December 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization. Technical Report HPL-1999-77, Hewlett Packard, June 1999.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[5] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, December 2000.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Annual International Symposium on Code Generation and Optimization*, pages 265–275, March 2003.

[7] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March-April 1998.

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

[10] D. Deaver, R. Gorton, and N. Rubin. Wiggins/redstone: An on-line program specializer. In *IEEE Hot Chips XI*, 1999.

[11] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *First Annual International Symposium on Code Generation and Optimization*, pages 15–24, March 2003.

[12] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *35th International Symposium on Microarchitecture*, pages 257–268, 2002.

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

[14] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *6th Workshop on Interaction between Compilers and Computer Architectures*, pages 102–110, February 2002.

[15] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In *36th International Symposium on Microarchitecture*, December 2003.

[16] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, February 1997.

[17] W.-M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, July 1993.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, San Francisco, August 2002.

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

[20] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *First Annual International Symposium on Code Generation and Optimization*, pages 36–47, March 2003.