Generational Cache Management of Code Traces in Dynamic Optimization Systems

Kim Hazelwood and Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University
{hazelwood, smith}@eecs.harvard.edu

Abstract

A dynamic optimizer is a runtime software system that groups a program's instruction sequences into traces, optimizes those traces, stores the optimized traces in a software-based code cache, and then executes the optimized code in the code cache. To maximize performance, the vast majority of the program's execution should occur in the code cache and not in the different aspects of the dynamic optimization system. In the past, designers of dynamic optimizers have used the SPEC2000 benchmark suite to justify their use of simple code cache management schemes. In this paper, we show that the problem and importance of code cache management changes dramatically as we move from SPEC2000, with its relatively small number of dynamically generated code traces, to large interactive Windows applications. We also propose and evaluate a new cache management algorithm based on generational code caches that results in an average miss rate reduction of 18% over a unified cache, which translates into 19% fewer instructions spent in the dynamic optimizer. The algorithm categorizes code traces based on their expected lifetimes and groups traces with similar lifetimes together in separate storage areas. Using this algorithm, short-lived code traces can easily be removed from a code cache without introducing fragmentation and without suffering the performance penalties associated with evicting long-lived code traces.

1. Introduction

Dynamic optimization systems [3, 5, 10, 16] apply code optimizations to existing binaries at runtime. A typical dynamic optimizer performs four major tasks. First, it observes execution and generates a series of *code traces* that represent the common dynamic sequencing of instructions. Second, it applies optimizations and/or transformations to the generated code traces. Third, it stores the code traces in a software-based *code cache*. Finally, it intercepts execution and directs it to the code cache for all future executions of the optimized code traces until program termination.

The benefits of dynamic optimization range from leveraging runtime information for optimization to optimizing mobile code. Speedups are often achieved as a result of increased instruction locality and code specialization. Implementations of dynamic optimizations systems have achieved speedups averaging 7% over optimized code [3]. Unlike static optimization passes, the major tradeoff of dynamic optimization is that the time required to observe runtime behavior, perform optimizations, and update program code directly impacts runtime performance. By focusing efforts on frequently-executed regions of a program and by maximizing the amount of program execution that occurs directly inside the dynamic optimizer's code cache, the runtime overhead of dynamic optimization can be minimized.

Code cache management aims to maximize the amount of execution time spent in the code cache, while imposing only minimal runtime overhead. In the past, code cache designers have used simple management techniques, such as flushing the cache upon detection of a program phase shift [2] or creating an unbounded code cache [5]. The designers evaluated each of these cache management schemes using the SPEC2000 benchmark suite.

In this paper, we show that the problem and importance of code cache management changes dramatically as we move away from the SPEC2000 benchmarks and look at today's large interactive Windows applications, such as Microsoft Word and Adobe Acrobat. Not only does the number of code traces generated at runtime increase dramatically, but the rate at which they are generated also increases dramatically. We argue that while a typical code expansion of 500% can be tolerated for the SPEC2000 benchmarks, this code expansion is unacceptable for applications with code footprints on the order of 10's to 100's of megabytes, thereby making unbounded code caches undesirable.

Furthermore, our observations of the typical lifetime of code traces show that the majority of traces generated of-

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE

ten fall into one of two categories: (1) those that live for less than 20% of the execution, and (2) those that live for more than 80% of the execution. This observation motivates our algorithm for employing *generational code caches* that allow us to easily evict short-lived code traces from the code cache without introducing unnecessary fragmentation or suffering the performance penalties associated with evicting and regenerating long-lived traces.

Our generational code cache management algorithm yields multiple code caches, each of which must be independently maintained. Therefore we discuss the *local* policies that drive the individual code caches, as well as the *global* policy of interaction between the multiple caches.

The contributions of this paper are as follows:

- A characterization of the different code caching needs of SPEC2000 and large interactive applications.
- Implementation details of code cache management techniques, and solutions for the problems of undeletable traces and unloaded DLLs.
- A proposal for and evaluation of generational code cache management.

The remainder of the paper is organized as follows. Section 2 provides a historical review of code cache management, including solutions implemented in early dynamic optimization infrastructures. Section 3 discusses the different code caching needs of interactive and non-interactive applications, and explains why solutions for one of these kinds may not be effective for the other. Section 4 discusses issues that complicate code cache management and introduces a pseudo-circular local code cache management policy that minimizes fragmentation, eliminating the need for expensive defragmentation operations. Section 5 introduces a global technique for leveraging generational information about code traces to provide for separate code caches for short- and long-lived code traces, thereby reducing the number of conflict misses in the code cache. Section 6 evaluates generational cache management in terms of code cache miss rates and resulting overheads. Finally, Section 7 concludes.

2. Background

Several dynamic optimization infrastructures have been developed. Dynamo [3] is a system developed at Hewlett-Packard Laboratories that provides a software-based mechanism for selecting and optimizing traces of HP-UX instructions. Dynamo employed a preemptive flushing mechanism [2] for cache management, which detected program phase changes and flushed the code cache at those points. Several successors to Dynamo have since surfaced, including DELI and DynamoRIO.

DELI [10] is a VLIW version of Dynamo geared toward embedded-processor applications that was developed by Hewlett-Packard Laboratories in conjunction with ST Microelectronics. While DELI's code cache management mechanism is not discussed in their latest publication, they do mention facilities for controlling the timing of code cache flushes. It is not known whether DELI is equipped to perform more fine-grained cache management.

DynamoRIO [5] is a dynamic optimization research infrastructure collaboratively developed by Hewlett-Packard and MIT. DynamoRIO executes on the IA-32 architecture in Windows or Linux. The cache management mechanism in the publicly available version of DynamoRIO defaults to an unbounded code cache. However, a user may impose a maximum code cache limit via an environment variable, and in this case, DynamoRIO employs a circular buffer mechanism similar to that proposed in our prior work [12].

Other dynamic optimizers have been discussed, such as Wiggins/Redstone [9] developed at Compaq and Microsoft's Mojo [8] for Windows applications; however the code cache management schemes employed in these systems have not been publicly disclosed.

In our prior work [12], we investigated several code cache management schemes and found that exploiting temporal locality of code is important, but that attention must be paid to minimizing overhead and fragmentation. We extend our prior work by discussing important implementation details of code cache management, introducing a novel cache management algorithm, and evaluating the design using both SPEC2000 and interactive Windows applications.

3. SPEC2000 vs. Interactive Applications

As we discussed in Section 1, previous efforts in code cache management were driven by performance results on the SPEC2000 benchmarks. The limited number of code traces generated while executing the SPEC2000 benchmarks implied that code cache management was not an important issue. This situation changes dramatically as we look at large interactive applications, where every mouse movement or screen refresh translates into massive amounts of code. While other researchers have discussed salient features of interactive applications and their impact on research in other contexts [7, 14], we point out the specific features of these large applications that make them so challenging for a code cache management scheme. We also show that certain existing solutions, such as unbounded code caches, are impractical for such large applications.

We use DynamoRIO for our investigations, as it is publicly available on Windows and Linux. Our Windows platform is Windows 2000 server, and our Linux platform is Red Hat Linux 7.2. The benchmarks selected for this investigation include the SPEC2000 suite executed to completion

Name	Seconds	Description
access	202	Database App
acroread	376	PDF Viewer
defrag	46	System Util
excel	208	Spreadsheet App
iexplore	247	Web Browser
mpeg	257	Media Player
outlook	196	E-Mail App
pinball	372	3D Game Demo
powerpoint	173	Presentation
solitaire	335	Game
winzip	92	Compression
word	212	Word Processor

Table 1. Interactive Windows benchmarks used in our evaluation.

on Linux using the reference inputs, and several common Windows applications described in Table 1. The inputs to the Windows applications were manual user interaction performing everyday tasks for the duration of time listed in Table 1, and the verbose logs generated during execution were reused for all of our simulations.

3.1. Maximum Code Cache Size

Our first study looked at the amount of memory required in order to support unbounded code caches. Figure 1 shows the maximum code cache size reached when we allowed DynamoRIO to execute with an unbounded cache. On average, the SPEC2000 benchmarks resulted in a 736 KB code cache (in addition to the size of original executable and the DynamoRIO executable). The largest of the SPEC2000 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 SPEC2000 benchmarks (Figure 1a), 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 our interactive benchmarks, we see a twenty-fold increase in the required code cache size. On these benchmarks, our code cache averaged 16.1 MB, with the `word` benchmark requiring a 34.2 MB code cache to avoid code cache management. While most users today have much more than 34 MB available, they also tend to run more than one application at a time.

3.2. Code Expansion

Code caches experience a large amount of code expansion when compared to the original executable size because code caches store superblocks of original code. Superblocks are single-entry multiple-exit regions. This feature simplifies optimization, but potentially causes duplication in many traces in the code cache.

Our second study therefore looks at the size of the unbounded code cache as a multiple of the original code footprint. The application code footprint is defined as the size of the static code executed by DynamoRIO, including system libraries. We calculated the code size expansion using the following equation:

$$codeExpansion = \frac{finalCacheSize}{applicationFootprint} \quad (1)$$

A code expansion of 100%, for example, means that we've doubled the size of the original application (original application code + cached code).

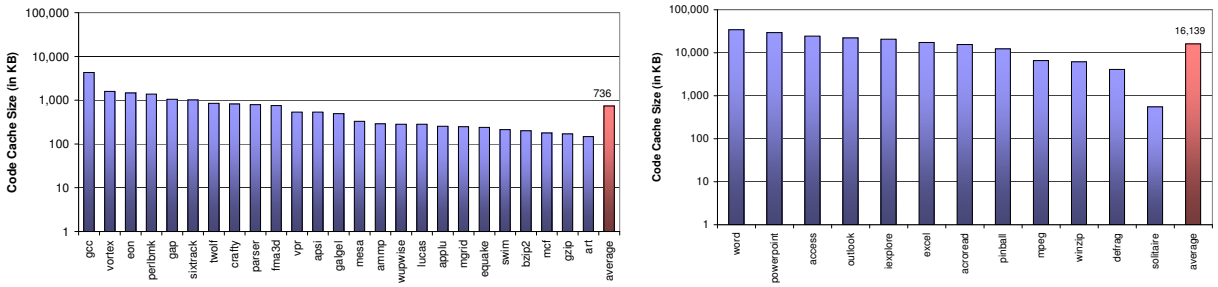
As we see in Figure 2, the code expansion factor is roughly 500% for both SPEC2000 and our interactive benchmarks. Therefore, this indicates that the main factor contributing to the final code cache size is the original application size, which is dramatically larger for the interactive applications. The expansion factor is fairly steady, with only a 59% and 111% standard deviation for the interactive and SPEC2000 benchmarks, respectively. This implies that as the sizes of new software releases grow, unbounded code cache sizes will grow proportionately.

3.3. Trace Generation Frequency

Next, we studied the rate at which code traces were inserted into the code cache. This metric allows us to evaluate the strain that will be placed on the cache management system when cache-size limitations are imposed. Figure 3 compares the trace insertion rate, measured in KB/sec, of SPEC2000 and interactive applications. This figure shows that most of the SPEC benchmarks generate less than 5 KB of code traces each second—with exceptions including `gcc` which generated 232 KB/s and `perlbmk` which generated 89 KB/s. On the other hand, only one of the interactive applications — `solitaire` — generated less than 5 KB/s. These results lead us to believe that larger applications typically result in a higher frequency (as well as number) of traces generated at runtime.

3.4. Unmapped Memory

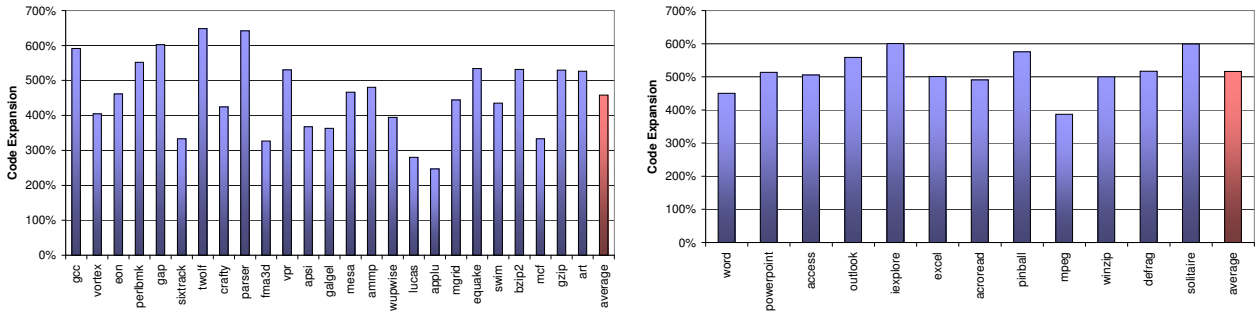
Finally, we looked into a matter that was unique to the Windows applications in our suite of benchmark applications. Anytime a region of memory containing program code is unmapped, the corresponding code traces must be deleted from the code cache. This is because other code could be loaded into the same address space, resulting in a stale code-cache entry. This phenomenon occurs frequently



(a) SPEC2000 Benchmarks

(b) Interactive Windows Benchmarks

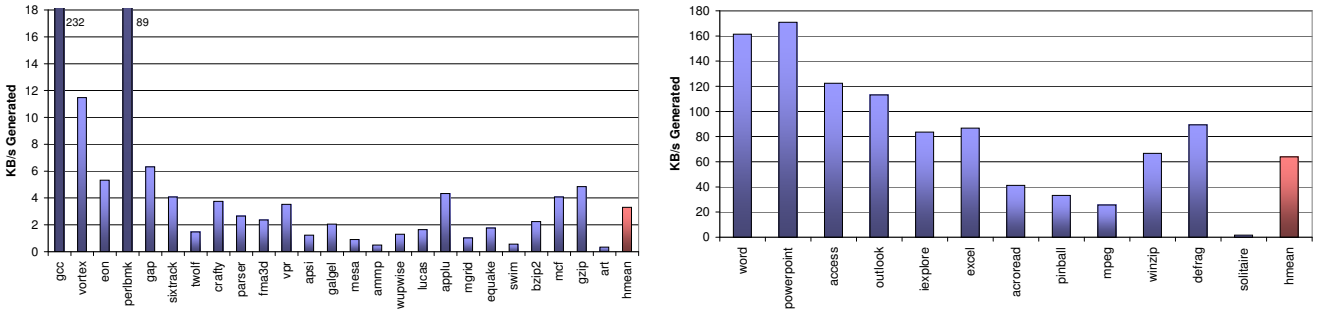
Figure 1. Maximum cache size required to avoid cache management plotted on a logarithmic scale. Note that the maximum cache size of our interactive benchmarks averaged over 20 times larger than SPEC2000.



(a) SPEC2000 Benchmarks

(b) Interactive Windows Benchmarks

Figure 2. Code expansion with an unlimited code cache.



(a) SPEC2000 Benchmarks

(b) Interactive Windows Benchmarks

Figure 3. Amount (in KB) of code generated in the code cache each second. Note that the y-axis of the Windows benchmarks is ten times larger than the corresponding axis in the SPEC2000 graph.

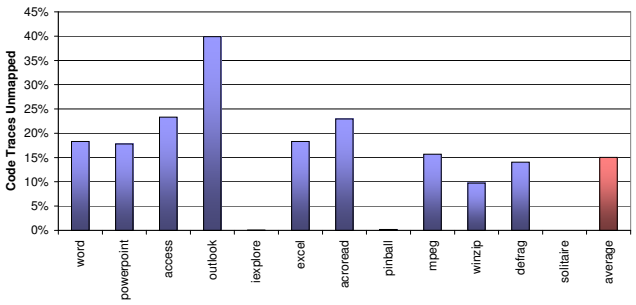


Figure 4. Percentage of code traces that must be removed from the code cache due to unmapped memory.

in Windows applications that load and unload dynamically-linked libraries. In fact, Figure 4 shows that an average of 15% of the interactive benchmark's code must be deleted from the dynamic optimizer's code cache due to unmapped memory. This trait of Windows applications can result in inefficient code caches, because fragmented *holes* appear in the code cache due to the deleted code traces. Therefore, an effective code cache management mechanism must support trace deletion, while minimizing fragmentation.

4. Local Management Policies

Now that we have shown code cache management to be an important problem with several unique challenges, we move on to our discussion of code cache management mechanisms designed to overcome these challenges.

There are two granularities at which we must consider code cache management policies. *Local code cache management* refers to replacement policies implemented within a single code cache. Examples of local policies include least-recently used (LRU), circular buffer, and preemptive flushing. *Global code cache management* refers to the hierarchy and policy of interaction between multiple code caches. While our main focus in this paper is global code cache management policies, it is important to also discuss effective local policies that may be implemented, as the two are often complementary.

4.1. Trace Selection

All local code cache policies must begin with trace selection. During trace selection, the decision of *when* and *how* to generate and cache a code trace is made. Rather than interpreting code, DynamoRIO begins by copying every basic block (a single-entry single-exit sequence of instructions) into a *basic block cache* prior to executing the block. Certain basic blocks are marked as *trace heads* because they are either (a) the target of a backward branch (signaling a loop) or (b) an exit from an existing trace. Counters are associated with each trace head, and each execution of a trace head block will increment the counter. When the trace head counter exceeds the *trace creation threshold*, currently set at 50 executions, DynamoRIO enters *trace generation mode*.

Trace generation involves grouping basic blocks together to form a superblock of code, and copying the superblock into a separate *trace cache*. Superblocks are used because they are well suited for applying low-overhead compiler optimizations. Decisions on which basic blocks to group together to form a superblock are made by following the *Next-Executed Tail* policy described by Duesterwald and Bala [11], which simply follows the path of execution rather than employing complex path profiling techniques. Trace generation continues until either (a) a backward branch is encountered, or (b) the start of an existing trace is encountered.

4.2. Replacement Challenges

The driving forces behind any local code cache management policy must include all of the following attributes: low overhead, emphasis on temporal locality, and minimized fragmentation. The first motivating factor—low overhead—is important because the overhead of code cache management directly impacts the application's runtime performance. Therefore, management policies that require a walk of the entire code cache or a complex calculation are not feasible. The second factor—temporal locality—has been shown to be very important to the performance of code cache management policies [12]. Finally, the third factor—minimized fragmentation—is important for two reasons. First, we are inserting and deleting code traces that vary in size, therefore fragmentation can occur. Second, in applications that unmap memory, deletions must occur immediately. Therefore, a policy must either include a defragmentation step, or make efforts to minimize the fragmentation that occurs.

The problem of caching elements of varying sizes also appears in the area of web caching [4, 6, 13]; however, web cache maintenance can occur during periods of reduced internet activity. Code cache maintenance, on the other hand, directly impacts application performance.

Each of these factors was considered in the design of the circular buffer (FIFO) policy proposed in our prior work [12], where we found superior performance over LRU with no fragmentation. However, several issues complicate the design, making a pure circular buffer unachievable in a real dynamic optimizer. These issues include *undeletable traces* and *program-forced evictions*, and they are handled by the scheme we describe in Section 4.3.

Undeletable Traces Several situations may arise that require a dynamic optimizer to mark a code trace undeletable from the code cache. For instance, during an exception that occurs in a trace inside the code cache, execution of that trace must be suspended while the exception is handled. Yet, because control may later return to the trace, it is vital that the trace is not deleted in the meantime.

Program-Forced Evictions In Figure 4, we showed that code cache management is not the only reason for evictions from the code cache, and that any unmapped code region requires deletions of corresponding traces from the code cache. Such evictions will introduce fragmentation in all code cache mechanisms, including the circular buffer.

4.3. Pseudo-Circular Management

These complications warranted the design of a variant of the circular buffer scheme, which we call a *pseudo-circular* local management policy. 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 trace into the cache will evict zero or more existing traces in the cache depending on the free cache space and the size of the trace to be inserted. However, our scheme deviates from normal circular eviction when an undeletable trace is detected in the list of potential eviction candidates. At this point, the circular buffer mechanism resets its eviction pointer to begin evicting directly after the undeletable fragment, and the eviction process begins again.

Our circular management policy does not change its behavior in response to 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 to try inserting into those before evicting traces located at the eviction pointer, this approach complicates the cache management design, and may reduce the benefits of temporal locality.

5. Generational Code Caches

We now move on to our proposal for global cache management based on trace generations. Section 5.1 motivates our use of generational code caches by showing that there is a clear tendency in our benchmark suites toward traces living either for a short time (short-lived) or for most of the program's execution (long-lived). We have borrowed where appropriate from the literature on generational garbage collection, as we will discuss in Section 5.2, and based our architecture on a simple partitioning of the existing trace cache into two distinct and separately managed regions: a *nursery code cache* that stores all newly generated traces; and a *persistent code cache* that stores the long-lived traces. Figure 5 illustrates this basic architecture.

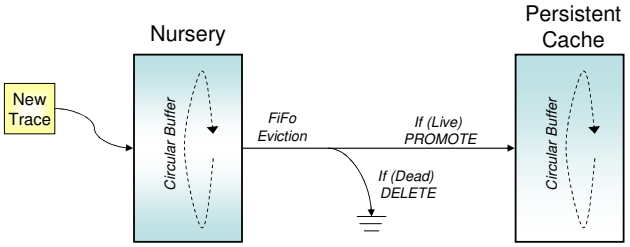


Figure 5. Conceptual view of the generational code cache architecture.

We assume that each code cache in our generational architecture adheres to the local pseudo-circular management policy described in Section 4. One could, of course, choose to implement a different local management policy; we leave this as a question for future studies.

As in generational garbage collection, the nursery is where traces stay until they have reached an appropriate age for promotion to the persistent cache. In our design, an eviction from the nursery cache indicates that a trace has "come of age." Section 5.3 describes the technique we use to know if we should promote the evicted trace, or delete it altogether. The decision is non-trivial because, as discussed in Sections 5.2 and 5.3, we do not have sufficient information to determine whether the trace evicted from the nursery will ever be referenced again. In other words, borrowing from the concept of liveness in compilers, we must ask if the evicted trace is live or dead.

Before continuing, we note that DynamoRIO already supports multiple code caches per thread. The separation of the basic block and trace caches (described in Section 4.1) can be considered a form of generational code cache management, where the execution count determines which cache stores the dynamically generated code segments. We extend this philosophy to identify storage for traces based on their expected lifetime. Because the trace cache contains the frequently-executed code (i.e. the hot code), we apply our generational approach only to it, and thus propose to move the design of dynamic optimization systems from a single unified trace cache to multiple distinct trace caches per thread.

5.1. Trace Lifetimes

To justify the need for generational code caches, we investigated the *lifetimes* of traces generated during benchmark execution. A trace lifetime is defined by Equation 2.

$$lifetime_i = \frac{lastExecution_i - firstExecution_i}{totalApplicationExecutionTime} \quad (2)$$

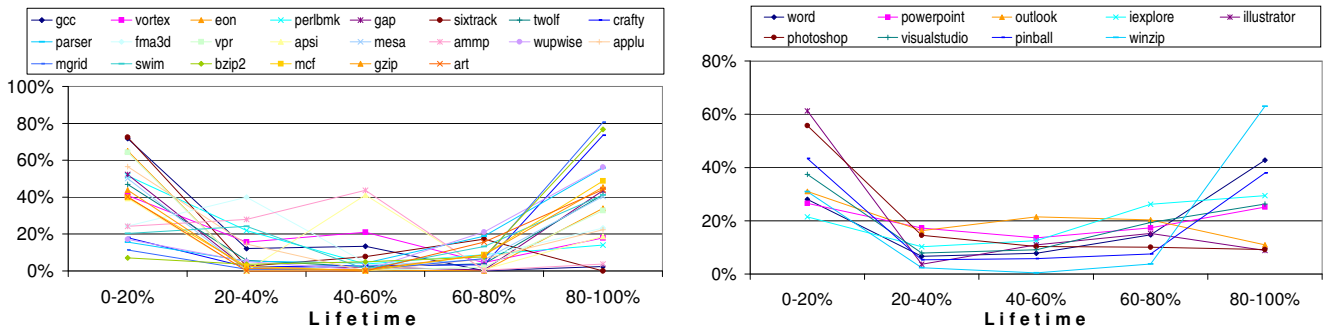


Figure 6. Lifetimes of traces as a percentage of total execution time. The y-axis shows the unweighted (static) percentage of traces that fall into each lifetime category.

Figure 6 shows the typical lifetimes of hot traces in the code cache calculated using Equation 2. The main observation from this figure is that the majority of frequently-executed code traces fall into one of two categories: short-lived (shown as less than 20% of execution time) and long-lived (shown as greater than 80% of execution time) with a relatively small percentage of traces falling in the middle. The fact that so many of the code traces fall into the extreme categories indicates that it may be advantageous for us to treat these code traces differently. Furthermore, the fact that the lifetimes in Figure 6 exhibit a U-shaped curve implies that two separate but equally-sized caches may work well.

5.2. Parallels in Garbage Collection

The problem of managing varying-sized elements with varying lifetimes already exists in other domains. Code cache management is in many ways similar to the problem of garbage collection. One technique in garbage collection is to take a generational approach to collecting unused memory [1, 17]. Generational garbage collection makes use of the *generational hypothesis*, which states that young objects are much more likely to die than old objects, therefore young objects are typically grouped and collected together.

We have leveraged some of the advances in the area of garbage collection in our approach to the code cache management problem, but 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 to memory management. 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.

Our code cache management design takes into account these features and constraints to form an effective generational code cache management system. Our approach to the problem of deciding whether to promote a trace to the persistent cache involves the use of an intermediate *probation cache* between the nursery and persistent caches, as shown in Figure 7.

5.3. The Probation Cache

The probation cache is similar to a *victim cache* in microarchitecture, which is used to detect and recover from untimely evictions from the instruction or data caches. The main distinction is that traces in the probation cache will never be reinserted into the nursery. Instead, the trace will either be promoted to the persistent cache or deleted, as described in the algorithm in Figure 8.

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 is experimentally determined in Section 6.1.

5.4. Code Relocation

In order to support the promotion of traces to a persistent cache, the dynamic optimizer must contain the functionality to move instructions from one address to another and provide the necessary fix-up for any address-relative jump instructions contained in the instructions. Fortunately, this

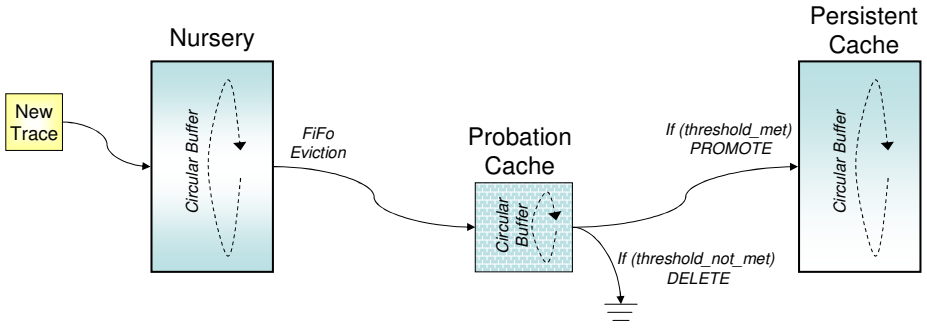


Figure 7. Generational code cache architecture using a probation cache.

Generational Eviction

```
insertNewTrace {
  if (nursery_fifo points to valid trace) {
    if (probation_fifo points to valid trace) {
      if (probation_trace.accessCount > threshold) {
        if (persistent_fifo points to valid trace) {
           evict persistent trace
        }
        promote probation trace to persistent cache
      }
      else {
        evict probation trace
      }
    }
    promote nursery trace to probation cache
  }
  insert new trace in nursery
}
```

Figure 8. An algorithm for generational code cache management with a probation cache. The `insertNewTrace` routine is called when a new trace is inserted into the nursery. The `*_fifo` variables point to the next insertion point in each circular buffer.

is a basic functionality of any dynamic optimizer. By the time a trace resides inside a trace cache, it has already been moved from the original program location into the basic-block cache, and once again from the basic-block to the trace cache.

6. Evaluation

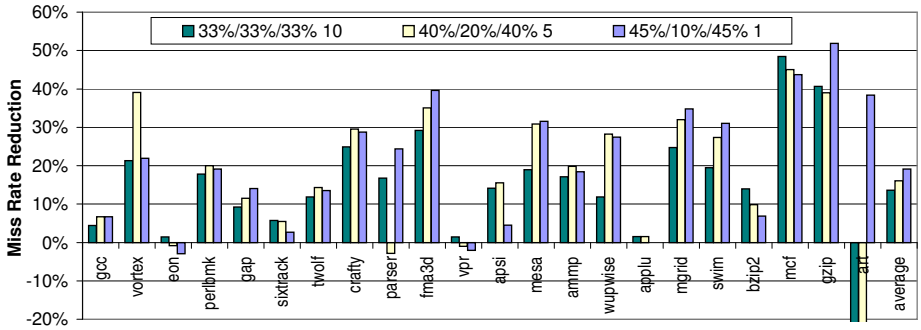
We evaluate our design using DynamoRIO and a generational cache simulator. DynamoRIO executed our benchmarks using an unbounded code cache, and we used the verbose log of cache accesses to drive our cache simulator. Our baseline for comparison for each benchmark is a single pseudo-circular cache sized at $(maxCache * 0.5)$ where $maxCache$ is the size required to avoid cache management for that benchmark. We compare the miss rate of this single pseudo-circular cache to that of multiple generational pseudo-circular caches sized such that the total size of all generational caches combined (including the probation cache) equals the size of the 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. 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, we needed to lower the promotion threshold. If the threshold was set too high, long-lived traces did not reach the promotion threshold before eviction from the probation cache.

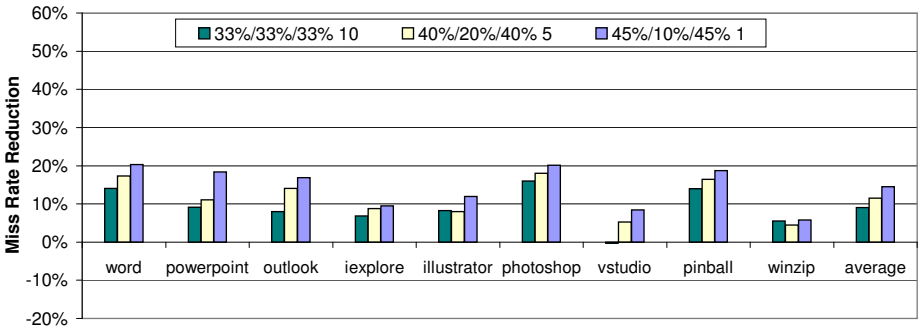
6.1. Code Cache Miss Rates

Figure 9 compares the miss rates of three instances of generational cache layouts 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 trace to the persistent cache.

An outlier in Figure 9 was `art`, which is also the smallest benchmark in terms of bytes of traces produced at runtime. Since we purposely sized our 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.



(a) SPEC2000 Benchmarks



(b) Interactive Windows Benchmarks

Figure 9. 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% is allotted to the probation cache, 33% to the persistent cache, and evicted traces with a minimum of 10 executions are promoted from the probation to persistent caches. Average values are the unweighted arithmetic mean across all benchmarks.

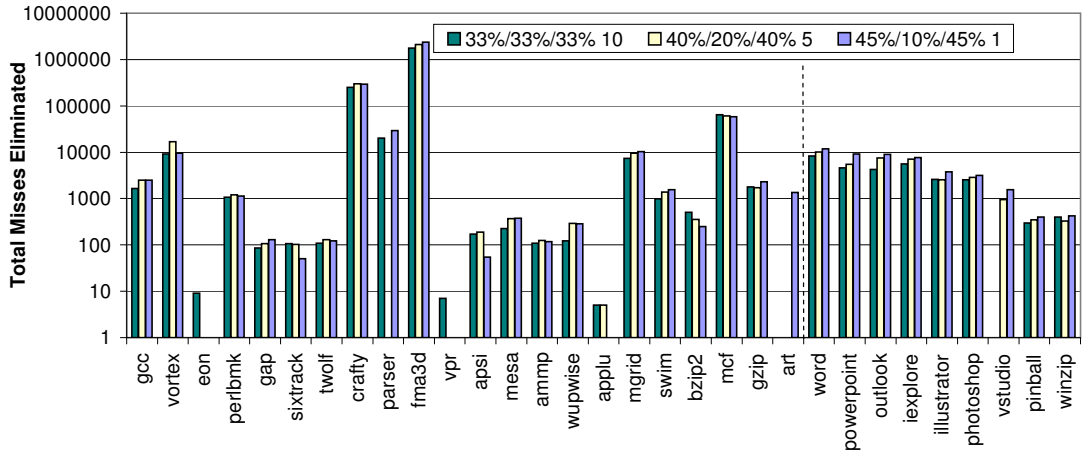


Figure 10. Total number of cache misses eliminated using generational code caches, as compared to a single unified cache. Note the logarithmic y-axis.

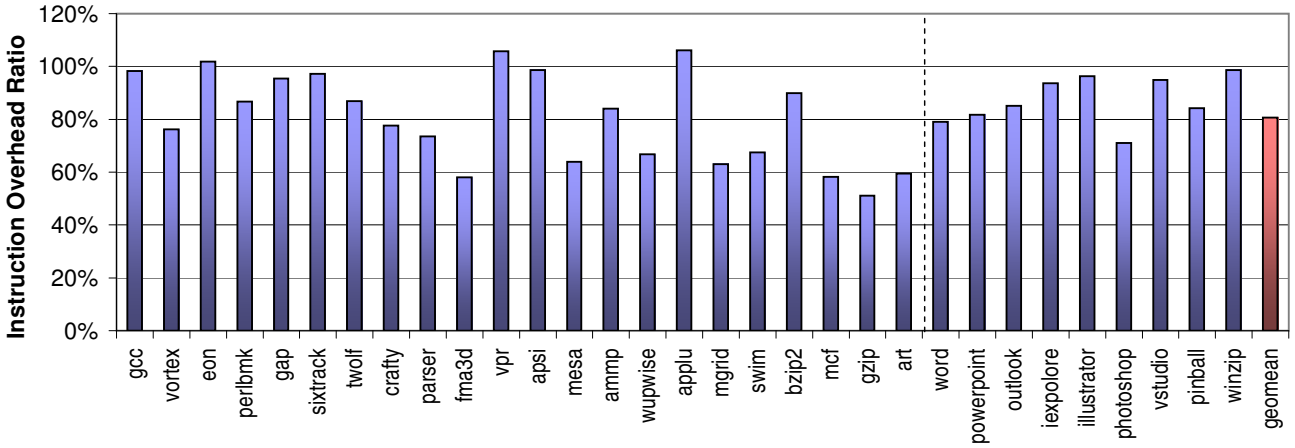


Figure 11. Instruction overhead ratio of generational code caches to a unified cache. Values below 100% represent a reduction in overhead, therefore smaller is better. The generational caches have 45%–10%–45% proportions between the nursery, probation, and persistent caches.

Finally, to provide perspective to the resulting miss rates, we present the actual *number* of eliminated misses in Figure 10. From this figure, we see that the miss rate reductions were often the result of eliminating many thousand code cache misses in the benchmarks.

6.2. 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 trace optimization. Conflict misses in the code cache require us to repeat these expensive steps unnecessarily.

We used the Pentium-4 performance monitors to get a realistic sense of the overhead of trace generation, evictions, promotions, and context switches within the DynamoRIO framework. Using the PAPI [15] interface and the Pentium instruction counters, we read the instruction counts before and after key events. We plotted these measurements against trace sizes where applicable, and generated the best-fit formulas listed in Table 2 using Microsoft Excel. For a 242-byte trace (the median across all benchmarks), the estimated overhead of trace generation is 69,834 instructions, eviction is 3,316 instructions, and promotion is 13,354 instructions.

A single conflict miss in the trace cache results in two DynamoRIO context switches, one trace regeneration, and one copy if the trace is executed enough to be upgraded from the basic block to the trace cache (which is the same cost as a promotion). For an average trace, this amounts to approximately 85,000 instructions. Therefore, while the generational cache management scheme includes the additional overhead of promoting traces to the probation and persistent caches, this carries far less overhead than premature eviction and trace regeneration.

Description	Overhead (instructions)
Trace Generation	$865 * (traceSizeBytes)^{(0.8)}$
DR Context Switch	25
Evictions	$2.75 * traceSizeBytes + 2650$
Promotions	$22 * traceSizeBytes + 8030$

Table 2. Overheads used in our evaluation.

We compared the overhead of trace generation, context switches, evictions and promotions in a unified code cache to that of our generational code cache. In Figure 11 we show the overhead ratio between generational caches and a unified cache, calculated using Equation 3.

$$overheadRatio = \frac{generationalCacheOverhead}{unifiedCacheOverhead} \quad (3)$$

Figure 11 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 code cache miss rate reductions were not large enough to offset the overhead of promotion between the generational caches. These three benchmarks performed better with a larger probation cache, as shown in Figure 9a. This result

shows that further exploration of the configuration space may improve overall performance of the benchmarks. Finally, we were encouraged by the fact that overhead reductions were seen in all of the large Windows benchmarks in Figure 11.

To provide a perspective on our overhead reductions, we calculated the impact on final execution performance and found that it was highly dependent on the actual number of mispredictions eliminated (which was shown in Figure 10). For `gzip` where 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.

7. Conclusions

This paper showed that large, interactive applications impose limiting constraints on a code cache management system. The salient features of such applications complicate management heuristics and strain the system due to the frequency and number of code traces generated at runtime.

We explored generational code caches as a means for solving the problem of code cache management in dynamic optimizers. Our motivation was based on an analysis of the lifetime of code traces residing in the code cache for various applications, and the observation that the majority of code traces were either very short- or very long-lived. We found that, by replacing a single cache with multiple generational caches, we can decrease the miss rates and resulting overhead of nearly all benchmarks, large or small, often significantly.

Acknowledgments

We wish to acknowledge Hewlett-Packard and MIT for the use of DynamoRIO. We also wish to thank Jim Smith for various discussions that motivated us to write this paper, and Derek Bruening for his input and ideas during several discussions of code cache management within the DynamoRIO infrastructure. Finally, we would like to thank Glenn Holloway and the anonymous reviewers for their helpful feedback.

This research was supported in part by grants from Compaq, HP, IBM, Intel, and Microsoft.

References

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

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

[3] 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.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[5] 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.

[6] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Usenix Symposium on Internet Technologies and Systems*, Monterey, CA, 1997.

[7] J. P. Casmira, D. R. Kaeli, and D. P. Hunter. Tracing and characterization of NT-based system workloads. *Digital Technical Journal*, 10(1):6–21, December 1998.

[8] 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.

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

[10] 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.

[11] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, October 2000.

[12] 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.

[13] S. Irani. Page replacement with multi-size pages and applications to web caching. In *29th ACM Symposium on Theory of Computing*, pages 701–710, May 1997.

[14] D. C. Lee, P. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications in windows NT. In *25th International Symposium on Computer Architecture*, pages 27–38, 1998.

[15] 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.

[16] 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.

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