

Code Cache Management Schemes for Dynamic Optimizers

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

Abstract

A dynamic optimizer is a software-based system that performs code modifications at runtime, and several such systems have been proposed over the past several years. These systems typically perform optimization on the level of an instruction trace, and most use caching mechanisms to store recently optimized portions of code. Since the dynamic optimizers produce variable-length code traces that are modified copies of portions of the original executable, a code cache management scheme must deal with the difficult problem of caching objects that vary in size and cannot be subdivided without adding extra jump instructions. Because of these constraints, many dynamic optimizers have chosen unsophisticated schemes, such as flushing the entire cache when it becomes full. Flushing minimizes the overhead of cache management but tends to discard many useful traces. This paper evaluates several alternative cache management schemes that identify and remove only enough traces to make room for a new trace. We find that by treating the code cache as a circular buffer, we can reduce the code cache miss rate by half of that achieved by flushing. Furthermore, this approach adds very little bookkeeping overhead and avoids the problems associated with code cache fragmentation. These characteristics are extremely important in a dynamic system since more complex strategies will do more harm than good if the overhead is too high.

1. Introduction

Dynamic optimization encompasses the idea of applying code optimizations to existing program binaries at runtime. The benefits range from leveraging runtime information to supporting technology for commercial approaches, such as Java. A dynamic optimizer works by observing runtime user behavior and runtime constants, then using that data as a guide for performing

optimizations on frequently-executed segments of code. These optimizations may include code re-layout, function inlining, and constant/copy propagation, among others. Following optimization, the new code segment is stored in a code cache. Execution of the optimized code segments occurs directly from the code cache for the remainder of the current program execution (or until the code segment is flushed from the code cache). Due to the increased instruction locality and code specialization, speedups are often achieved. In fact, recent implementations of dynamic optimization systems have achieved speedup values averaging 7% over +02 optimized code [2]. The major tradeoff of dynamic optimization is that, unlike static optimization passes, the time required to observe runtime behavior, perform optimizations, and update program code directly impacts runtime performance. It is very important to keep the overhead to a minimum, or we may lose the benefits of dynamic optimization altogether.

One method for reducing the overhead of dynamic optimization is to make smarter choices regarding code cache management. Because it is not feasible to maintain all optimized code traces produced during an execution in a single code cache, a cache management scheme must be employed. The management scheme should have low overhead and should exploit temporal locality by attempting to keep useful, active code in the dynamic optimizer's code cache. If the same portions of code are repeatedly flushed and regenerated in the cache, then we clearly need to take a different approach when deciding which portions of code are stale and should be flushed. Yet, complex cache management strategies may do more harm than good if the overhead of the scheme is too high. A middle ground that balances the benefits of smarter management choices with the complexity of a management algorithm should be thoroughly investigated.

Several dynamic optimization systems exist that could benefit from smarter cache management, among them are Dynamo [2], Mojo [5], and Wiggins-Redstone [6]. Each system works by (1) performing runtime profiling

to determine *hot traces* (frequently executed portions of contiguous code), (2) copying the hot traces into a software-based cache mechanism (possibly performing optimizations on the traces en route), and (3) executing future instances of the hot traces directly from the code cache.

This paper explores cache management strategies as they apply to variably-sized elements and analyzes five alternatives to the currently implemented full cache flush scheme. While the results of this paper were produced with a dynamic optimizer in mind, this work could be equally applicable to dynamic translators and hardware-based code caching mechanisms. The remainder of the paper is organized as follows. Section 2 discusses our experimental methodologies, defines terminology, and presents background data that is used as the foundation of our research. Section 3 discusses the issue of fragmentation that arises when we deal with elements of various sizes. Section 4 discusses several proposed cache management strategies, which are evaluated in terms of resulting miss rate, overhead, and fragmentation in Section 5. Finally, Section 6 presents related work and Section 7 concludes.

2. Methodology

Throughout the paper, we use the term *traces*. A trace is a superblock region [9] that is typically used as a basis for optimization (see Figure 1). Traces contain a single entry point and multiple exit points. Internal loops and side entries are not allowed, however a trace may span procedural boundaries. Inside the code cache, a trace is laid out as shown in the right half of Figure 1. Code is often duplicated and specialized within the code cache. For example, a second code trace may include the function call starting with block E in Figure 1, but the procedure may be specialized by choosing the E-F-H path instead of the E-G-H path shown in the figure.

To provide a feel for the size of a typical code trace,

	avg	min	max	stdev
gzip	97	41	1,742	88.27
vpr	102	41	2,223	122.23
gcc	96	41	4,450	97.09
mcf	96	41	3,310	194.75
crafty	121	41	3,431	129.60
parser	105	41	1,931	92.16
eon	126	41	4,810	307.49
perlbmk	90	41	4,253	103.80
gap	101	41	4,044	105.36
vortex	115	41	4,978	138.95
bzip2	99	41	2,085	105.24
twolf	126	41	2,628	196.33
average	106	41	3,324	140.11

Table 1 – Size (in bytes) of code traces produced by Dynamo.

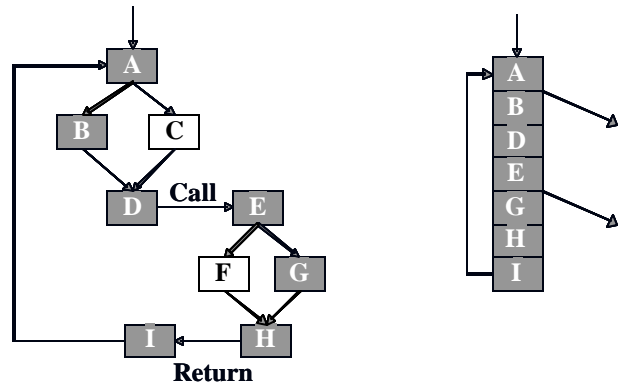


Figure 1 - Example of a code trace

Table 1 shows the average size of a code trace produced by the Dynamo system running on x86 (as discussed in Section 2.1). Across the SPECint2000 benchmarks using reference inputs, the average trace size is 106 bytes. Yet, as we can see from Figure 2 and the standard deviation column of Table 1, the sizes of individual traces vary greatly during execution of a single benchmark. Table 2 then shows us the number of distinct traces that are produced throughout a single execution of each benchmark. On average, we may experience anywhere from 1,200 to 200,000 traces per execution. Based on the average trace size (from Table 1), Table 3 then shows us that while we can typically fit around 10,000 traces in a 1 MB code cache, we can only fit 625 in a 64 KB cache. This clarifies our point that even for larger code caches, all traces cannot reside in the code cache throughout program execution and emphasizes the need for smart code cache management.

2.1. Our Execution Environment

All results used in this paper were generated using a research version of HP Labs' Dynamo 2.0 dynamic optimizer on an Intel Pentium-II based machine running RedHat Linux 6.2. The Dynamo 2.0 research tool was

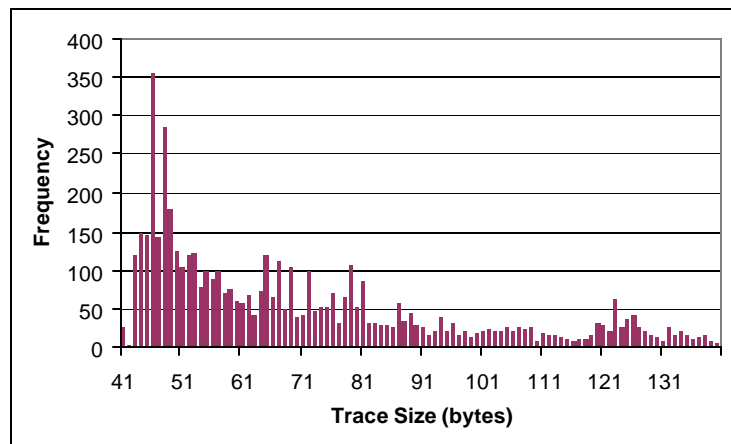


Figure 2 - Distribution of trace sizes for 186.crafty.

gzip	7,364
vpr	8,509
gcc	200,302
mcf	1,272
crafty	7,153
parser	7,856
eon	6,238
perlbmk	75,436
gap	9,827
vortex	39,207
bzip2	4,822
twolf	6,572
average	31,213

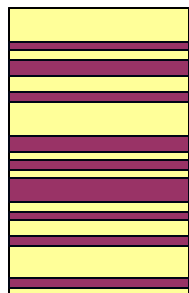
Table 2 – Distinct number of code traces produced during execution. If we omit gcc as an outlier, the average drops to 15,841.

described by Bruening et al [4] and Smith [11]. Targeted for x86 architectures running Linux or Win32, Dynamo 2.0 differs from that as described by Bala et al [2]. It is now a research tool equipped with several application programming interface (API) hooks to allow information regarding the progress of dynamic optimization to be relayed to the user, while the source code and internals remain a black box. The industrial version of Dynamo tracked its own progress and provided an automatic bailout mechanism when it recognized that native execution would be better; however, the research version does not provide this functionality.

We used all twelve SPEC2000 integer programs to generate results. The official test, training, and reference inputs were run to completion under the control of Dynamo using the SPEC2000 *runspec* script. We set environment variables indicating that Dynamo was to dump a trace of all code cache accesses, insertions, and evictions. We then sent this trace through a simulator that implemented the various code cache replacement schemes and analyzed the results.

3. Code Cache Fragmentation

An important issue that must be considered when designing a dynamic optimization cache management scheme is the problem of code cache fragmentation. Just as our hard disk becomes fragmented over time when we create and delete variably-sized files, the cache of the dynamic optimizer may also become fragmented. But because trace generation and replacement occur so frequently in a dynamic optimizer, the problem cannot be ignored.



**Figure 3 - A
Fragmented Code
Cache**

	1 MB	512 KB	256 KB	128 KB	64 KB
gzip	10,810	5,405	2,703	1,351	676
vpr	10,280	5,140	2,570	1,285	643
gcc	10,923	5,461	2,731	1,365	683
mcf	10,923	5,461	2,731	1,365	683
crafty	8,666	4,333	2,166	1,083	542
parser	9,986	4,993	2,497	1,248	624
eon	8,322	4,161	2,081	1,040	520
perlbmk	11,651	5,825	2,913	1,456	728
gap	10,382	5,191	2,595	1,298	649
vortex	9,118	4,559	2,280	1,140	570
bzip2	10,592	5,296	2,648	1,324	662
twolf	8,322	4,161	2,081	1,040	520
average	9,998	4,999	2,499	1,250	625

Table 3 - Number of traces in a typical code cache for each benchmark and cache size.

Figure 3 shows an example of a fragmented cache. The darker areas indicate free space. Consider the case where a fragment of size 1 KB must be inserted into the cache. While the sum of the free space in the cache may add up to 1 KB, a *contiguous* segment of size 1 KB is not available. In this case, we can either employ an expensive defragment operation or simply lose any fragmented free space that is too small to store a code trace. Fragmentation is a serious issue; most known techniques for defragmentation are much too expensive to be implemented in a runtime system. Yet the need for contiguous free space in a code cache necessitates either a low-overhead runtime defragmentation solution, or a management scheme that avoids fragmentation within the code cache altogether.

4. Cache Management Strategies

The problem of cache management in a dynamic optimization system is much more complicated than the standard tasks of instruction and data cache management in modern microprocessors. Unlike hardware caching mechanisms, which focus on replicating data to a location closer to the CPU, dynamic optimizers use caching to create a space in memory where they have the freedom to modify the contents and layout of a dynamic sequence of code. Furthermore, while data and instructions can be cached in fixed-sized blocks, requiring the same partitioning for code traces would negatively interact with the performance optimizations and essentially jeopardize the performance benefits of the optimized segment.

For these reasons, many dynamic optimization systems were designed to perform aggressive, unsophisticated code cache management. The original Dynamo system [2] is one example. As the code cache of the Dynamo system reaches full capacity, the entire cache is flushed in order to make room for new

optimized fragments. While attempts are made to recognize changes in an application's working set and preemptively flush the code cache, this is not always possible before the cache fills. Overall, the main motivation for flushing is to capture phase changes within a program and leverage these changes for simple cache management.

We can envision several other design motivations behind a code cache management scheme. In particular, the manager can leverage temporal locality of program code, the overall frequency count of program code, or even the size of a code trace when deciding which trace to evict from the code cache. While we expect schemes that focus on temporal locality to perform best, we must also take into account overhead of the schemes, as it could potentially negate the benefits of smarter cache management. In the following subsections, we discuss the current full cache flush scheme, along with five alternative cache management strategies that are based on one of these three motivating factors.

4.1. Full Cache Flush

One cache management scheme that is currently employed is the full code cache flush mechanism. Traces begin filling the cache at its lowest address and continue filling toward higher addresses. As soon as a trace is encountered that cannot be inserted into the cache, all traces are flushed, and the current trace becomes the first element inserted into the empty code cache. While this is a very low-overhead cache management strategy, it has the adverse side effect of flushing hot traces from the cache. If these hot traces are subsequently rebuilt and reinserted into the cache, unnecessary overhead is encountered. Table 4 shows the number of full cache flushes that occur during execution of each of the Spec benchmarks under the control of Dynamo. Table 4 shows us that as the code cache size decreases linearly, the

	1 MB	512 KB	256 KB	128 KB	64 KB
gzip	5	5	5	5	48
vpr	2	2	5	10	139
gcc	169	1,415	4,613	18,951	222,522
mcf	1	1	1	1	3
crafty	1	17	2,266	10,200	154,493
parser	1	167	3,024	6,525	18,456
eon	3	6	936	5,277	12,952
perlbmk	16	197	2,283	15,656	73,125
gap	1	4	19	692	9,735
vortex	6	254	3,509	28,810	116,378
bzip2	3	3	3	6	17
twolf	1	2	5	104	3,879

Table 4 - Number of code cache flushes that occur during execution of the ref input set of SPECint2000 for varying code cache sizes. Because a cache flush always occurs during program exit, the possible number of flushes that could be reported never falls below one.

number of cache flushes increases exponentially. For embedded or other memory-restricted systems, this large number of code cache flushes will certainly limit the performance benefits attainable by the dynamic optimizer. Yet, even standard systems cannot be expected to keep pace with trends in software code sizes. Furthermore, Table 5 shows us that 61.5% of the flushed traces are regenerated in the cache, clarifying our point that useful traces are often flushed from the cache and unnecessary overhead is encountered to regenerate those traces.

4.2. Least-Recently Accessed

An alternative cache management strategy is the Least-Recently Accessed (LRA) strategy. LRA attempts to recognize phase changes in the code by removing traces that have not been accessed recently. For this scheme, we again insert traces into the cache in the order that they are created. However, when the code cache becomes full, the trace that has not been accessed in the greatest amount of time becomes the first candidate for eviction. In the event that this trace will not free enough space to hold the newly optimized trace, the subsequent trace(s) in the code cache are also evicted (since we need to free enough *contiguous* space for the new trace). This scheme has the benefit of leveraging temporal locality, yet it has the unfortunate side-effect of removing innocent victim traces from the cache in order to create adequate contiguous space. Figure 4 gives an indication of the average number of evictions that typically occur in order to make room for the insertion of a single trace. From this figure, we see that it is usually necessary to evict one to two traces from the cache for each new trace that is inserted (or 70% of the time, an eviction will result in the eviction of one additional victim trace.) A second side-effect of this scheme is that it will create code cache fragmentation.

gzip	84.6%
vpr	38.7%
gcc	42.9%
mcf	29.9%
crafty	58.8%
parser	80.5%
eon	61.5%
perlbmk	63.3%
gap	64.5%
vortex	65.6%
bzip2	81.3%
twolf	66.1%
average	61.5%

Table 5 – Percentage of flushed traces that are later regenerated in the cache, averaged over the five cache sizes.

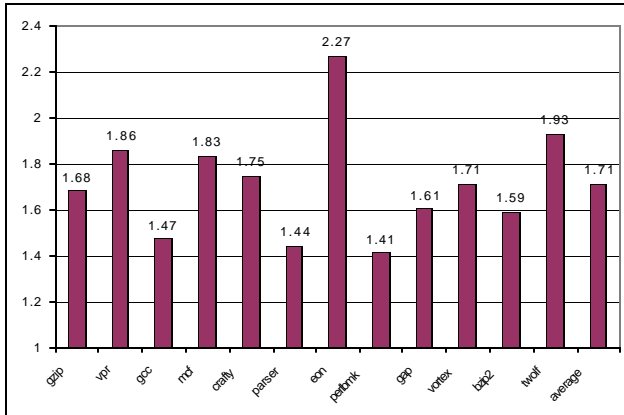


Figure 4 - Average number of trace evictions under LRA before a new element can be inserted.

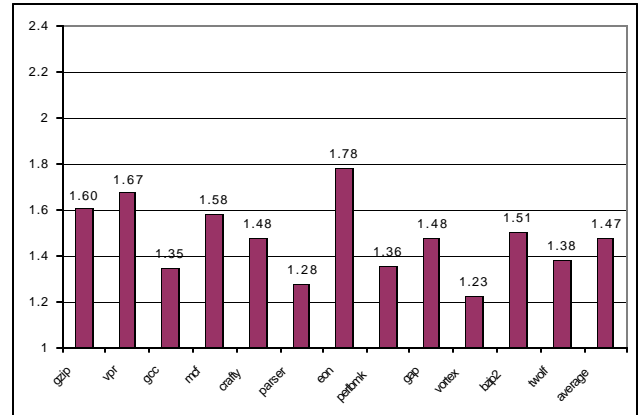


Figure 5 - Average number of trace evictions under LFA before a new element can be inserted.

4.3. Least-Frequently Accessed

By maintaining a counter indicating the number of accesses to each optimized trace in the code cache, we can determine the Least-Frequently Accessed (LFA) element. As under LRA, we evict the LFA trace and any subsequent traces necessary to make enough room for the replacement trace. Yet, also like the LRA scheme, LFA will suffer from the effects of code cache fragmentation. Figure 5 shows that 47% of evictions result in a victim trace eviction, thus victims are evicted at a lower rate than LRA. Finally, while this scheme will effectively recognize hot traces and allow them to remain in the code cache, it may have the adverse effect of removing *new* traces from the cache, which haven't yet acquired a high access count.

4.4. Least-Recently Created

One of the simplest temporal locality code cache management strategies is a model that treats the code cache as if it were a circular buffer. Called the Least-Recently Created (LRC) method, traces in the code cache are replaced in the same order as they were inserted. In

the case where the next candidate for replacement will not free enough space in the code cache for the new trace, subsequent traces are also removed. Though the average number of fragments removed is larger than LRU and LFU, the victim traces removed in this scheme were already next in line for eviction. In the case where an evicted trace frees much more space than is needed for the newly optimized trace, the free space will be used by the next trace inserted into the code cache, thus avoiding fragmentation. In terms of bookkeeping overhead, we merely update a pointer after each trace insertion. And, by proactively removing additional LRC traces from the code cache, a limited amount of additional cache management overhead can be eliminated in the future. Without prior knowledge of the size of the replacement element, the previous schemes, LRA and LFU, could not effectively remove elements in a proactive manner.

4.5. Largest Element

The next two schemes we explored place priority on the size of the element we were attempting to insert into the code cache. The first of these schemes works by

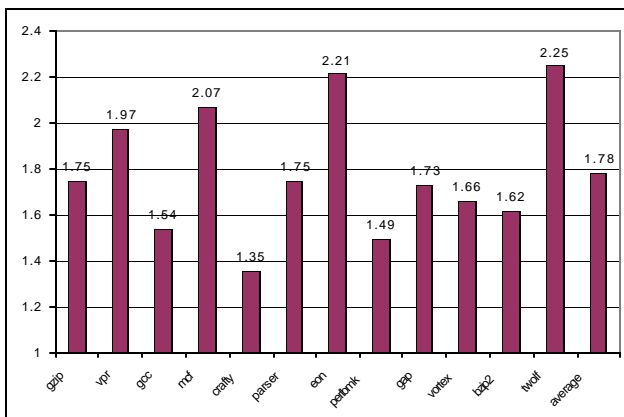


Figure 6 - Average number of fragments removed per insertion using the LRC removal scheme.

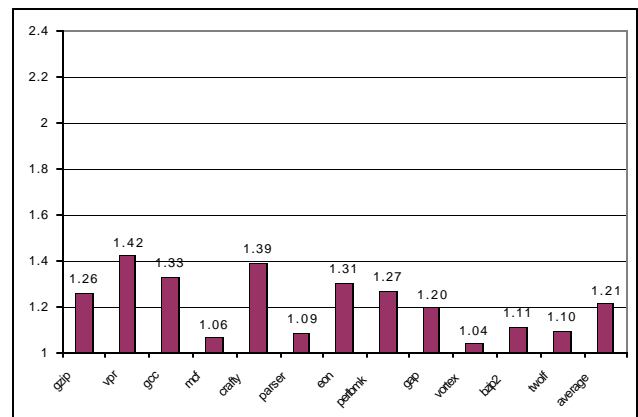


Figure 7 - Average number of fragments removed per insertion using the LE removal scheme.

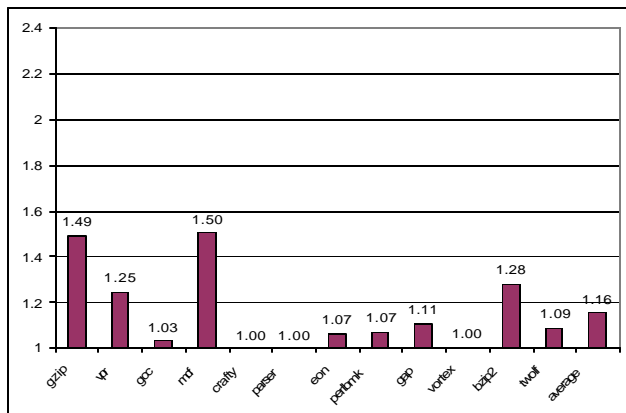


Figure 8 - Average number of elements that must be removed when attempting to find the best-fit element.

evicting the largest trace in the code cache. Called the Largest Element (LE) strategy, it works to minimize the number of evictions that must occur within the code cache, but with no interest placed on temporal locality. Again, subsequent victim traces are removed when the largest element does not produce enough free space for the new trace. Figure 7 (when compared with Figure 4 through Figure 6) shows that the average number of evictions per insertion that is necessary using the LE replacement scheme is much lower than the other schemes (dropping from as high as 1.78 down to 1.21). Yet this scheme will suffer from fragmentation, and the bookkeeping overhead includes maintaining a sorted list of the sizes of each trace in the code cache.

4.6. Best-Fit Element

The final strategy we explored was one that attempts to minimize fragmentation by searching the code cache for the best-fit trace to evict. In the Best-Fit Element (BFE) scheme, the code cache is scanned in search of the smallest element that is greater than or equal to the size of the newly optimized trace. When the best-fit trace is found, it is evicted from the cache, ideally leaving just enough room for the new trace. In the case where all traces in the cache are smaller than the new trace, traces are then grouped in subsequent pairs of two, and the best-fit search continues. Figure 8 shows that in 5 out of 6 instances, only one best-fit trace must be evicted from the cache to make room for an incoming trace. In fact, as Figure 9 indicates, we can usually find an eviction candidate that is within one byte of the best-fit size. But this scheme will have the highest overhead of all schemes we have presented, as we may have to do multiple scans of the trace sizes.

5. Results

We simulated the cache management schemes described in Sections 4.1 - 4.6 on a Pentium II-based system with the RedHat Linux 6.2 operating system.

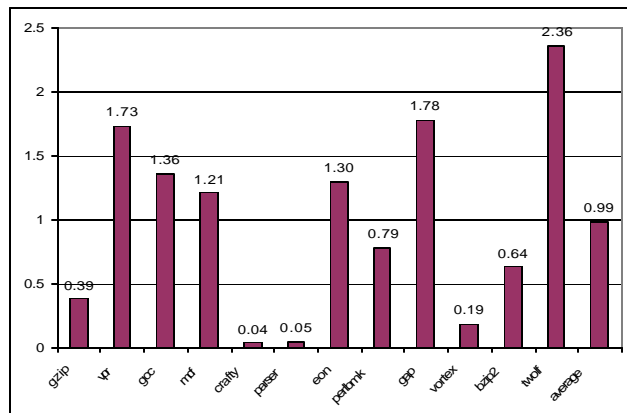


Figure 9 – Average size difference (bytes) between best-fit candidate and element to be inserted into code cache.

Traces were generated by a full-length execution of each of the SPECint2000 benchmarks running under the control of the Dynamo 2.0 dynamic optimization research tool. Because Dynamo was released to us as a black box, we were not able to implement our strategies directly within their system, thus we used the verbose output from Dynamo, which logs the code cache insertions, deletions, and accesses. This log was sent through our own trace-driven simulator, which implemented each code cache replacement policy. A portion of the verbose output generated by Dynamo is shown in Figure 10. Lines 1 and 4 indicate the insertion of a trace into the cache. The traces are numbered and contain a tag indicating their original starting address and trace size in bytes. Lines 2 and 5 indicate an entry into the code cache in order to access a trace. Listed is the original pc as well as the pc within the code cache. A more verbose level of output would also show the exact code within a trace.

```
(1) Fragment 1, tag 0x4013a173, size 45
(2)   Entry into F1(0x4013a173).0x401ac000
(3)   Exit from F1(0x4013a173).0x401ac018
(4) Fragment 2, tag 0x080488cd, size 61
(5)   Entry into F2(0x080488cd).0x401ac040
(6)   Exit from F2(0x080488cd).0x401ac068
```

Figure 10 – Sample verbose output from Dynamo.

5.1. Code cache miss rates

Figure 11 shows the miss rate of each of our code cache management schemes for various code cache sizes, averaged across all benchmarks. From this graph, we see that the best performers are LRC and LRA, regardless of cache size. Both LRC and LRA focus on temporal locality and effectively detect changes in the application's working set. The worst performers, LE and BFE, both focus on trace size rather than temporal locality. And while the graph indicates that the frequency of accesses is more important than trace size (LFA

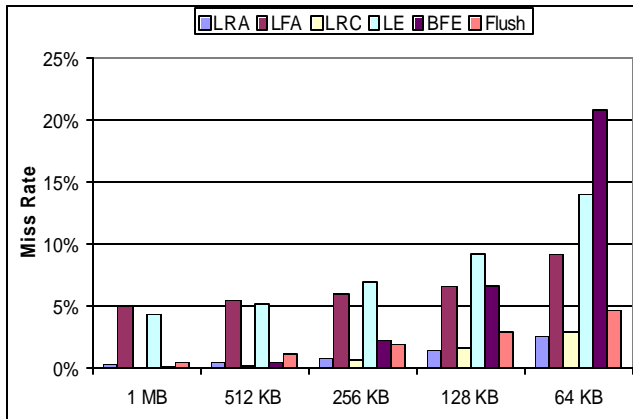


Figure 11 - Miss rate of each code cache replacement scheme for various code cache sizes.

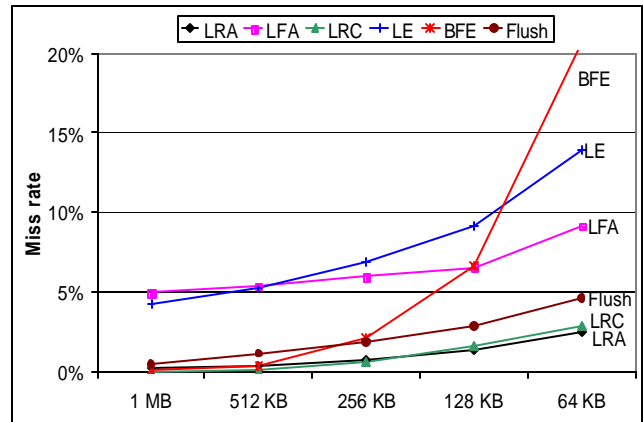


Figure 12 - Effect of cache size on miss rate for various replacement schemes.

performed better than LE and BFE for small cache sizes), frequency is still not as important as temporal locality. From Figure 11, we can therefore deduce that the most important metric in code cache replacement is temporal locality as expected.

Figure 12 more clearly depicts the effects of cache size on the miss rate of each replacement scheme. Most schemes tend to follow a similar (nearly linear) trend with the exception of BFE and LE. While BFE performs well in our large code caches, we see a sharp spike in miss rates as we move to smaller caches. This may be because the hottest traces tend to be similar sizes, and thus the BFE scheme continuously removes useful traces. LE appears to be more affected by decreases in code cache size than most of the other schemes, as its data points appear to rise faster than all others except BFE. This could indicate that the most useful traces in the code cache happen to also be the largest.

While Figure 11 and Figure 12 showed results averaged over all of the benchmarks, Figure 13 takes a

look at the performance of the six replacement schemes for each benchmark individually, using a fixed code cache size of 64 KB. We chose 64 KB because nearly all replacement schemes performed well when we dealt with a large code cache, and we felt that investigating the schemes on a small code cache would provide more insight. From the graph, we see several noticeable spikes for the BFE scheme for 186.crafty, 197.parser, and 255.vortex. Yet all other schemes perform very well for these benchmarks. One possible explanation is that for these benchmarks, the typical working set of traces may contain several similarly-sized elements, which continuously replace each other in the code cache using the BFE model. And by revisiting Figure 2, we can see that for crafty, the offending trace size is probably at 46 bytes where we see a spike in the graph. We also notice that while all replacement schemes perform extremely well on 164.zip (less than 1% miss rate), they all consistently perform poorly on 176.gcc (all over 20% miss rates). For these two cases, the miss rate is clearly

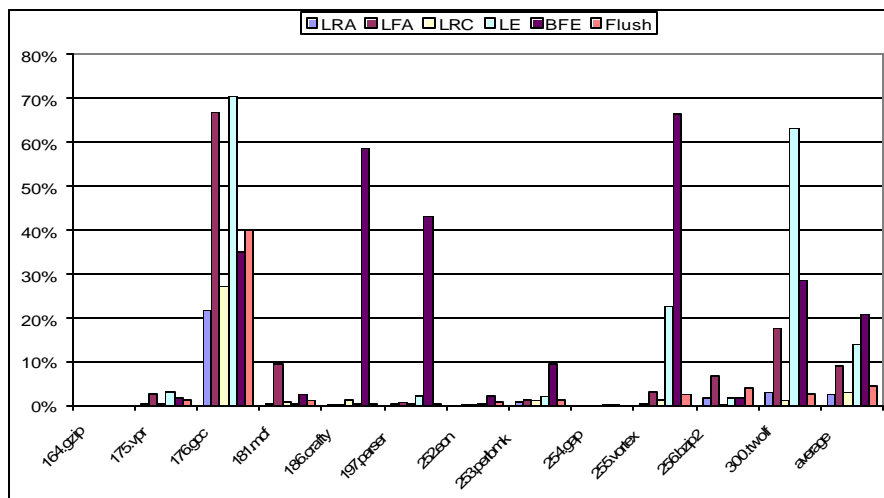


Figure 13 - Code cache miss rates with various replacement schemes and a fixed code cache size of 64KB.

scheme	fragmentation	additional victims	miss rate	management
LRA	Yes	71%	2.48%	priority queue
LFA	Yes	41%	9.11%	priority queue
LRC	None	78%	2.88%	pointer
LE	Yes	21%	13.91%	priority queue
BFE	Minimal	16%	20.77%	multiple sorted lists
Flush	None	N/A	4.61%	pointer

Table 6 - Summary of each scheme.

dominated by the relatively large or small working set, rather than the replacement scheme. We can verify this by revisiting Table 2, where we notice that `176.gcc` produces 200,302 traces during execution, while `164.gzip` produces only 7,364 traces.

5.2. Results Summary

As we combine our resulting miss rates with our discussion on fragmentation and complexity of cache management, we can make various conclusions regarding the effectiveness of each code cache replacement scheme. The best performers in the miss rate category were LRA, LRC, and Flush with overall miss rates of 2.48%, 2.88%, and 4.61% respectively (see Table 6). Yet, the LRA scheme will suffer from the effects of code cache fragmentation, and must be combined with either periodic flushing or code cache defragmentation. In fact, of the cache management strategies explored in this paper, only two did not suffer from the problem of fragmentation. For obvious reasons, full cache flushing avoids fragmentation. In LRC, replacement occurs in a circular manner, thus any free space left over after a replacement will be filled during the subsequent replacements. In terms of overhead, the LRC scheme must simply maintain a pointer to the next free location in the code cache (treating the cache as a circular buffer). The minimal code cache maintenance, combined with a very low miss rate make the LRC scheme a very attractive alternative to the more drastic scheme of full code cache flushing.

6. Related Work

Several groups are currently developing dynamic optimization systems. Dynamo is a system developed at HP Labs that provides a software-based mechanism for selecting and optimizing program fragments [2][4][11]. Wiggins/Redstone is a dynamic optimization and specialization system developed at Compaq [6]. Mojo, developed at Microsoft, is a dynamic optimizer that focuses on x86/WinNT binaries [5]. Dynamo, Wiggins/Redstone, and Mojo all perform optimizations transparently on an unaltered binary at runtime, while storing code traces in a software-based code cache.

There exists a smattering of prior work in the area of improving the performance of dynamic optimizers. Several researchers have proposed lightweight optimizations that are tailored for runtime execution [8][12][14]. Another major interest area has been in techniques to reduce the cost of monitoring application behavior [10][3] and then applying optimizations only to the hottest portions of the executable [1]. There is certainly a large body of work that discusses caching and cache management. As stated earlier, we are restricted in the kinds of cache management approaches we can use because we cache variable length items and must maintain the contiguity of the entire cached element. Under these restrictions, the closest related work appears in the area of memory overlays before the widespread use of virtual memory [7].

7. Conclusions

Code cache management in the dynamic optimization and translation domains are a crucial, but particularly challenging issue. The high cost of preparing traces for insertion into a cache, combined with the fragmentation issues involved in replacing variably-sized elements has caused many dynamic optimization system developers to sidestep the issue and implement either an enormously large code cache, or an unsophisticated replacement scheme such as full cache flushing. This paper explored five alternatives to the full flush model, and discussed the benefits and tradeoffs of each model. By weighing the factors of (1) code cache miss rate, (2) fragmentation, and (3) complexity of code cache management, we found the Least-Recently Created (LRC) scheme to be a viable solution that succeeds in reducing the code cache miss rate by nearly half of that achieved by the full flush model.

Our future work involves extending our research to include an investigation of the effects of multithreading, interrupts and context switches on code cache management. In addition, we hope to investigate various hybrid or adaptive code cache management schemes. Finally, we hope to directly implement these code cache management schemes in a dynamic optimizer to get more specific details regarding the overhead of each scheme.

Acknowledgments

We would like to thank Hewlett-Packard Laboratories Cambridge for the use of Dynamo. We also wish to acknowledge the independent reviewers for their feedback on an earlier version of this paper, as well as Tom Conte for his insight into an early version of this research. Kim Hazelwood and Michael D. Smith are funded by research grants from Compaq, Hewlett-Packard, IBM, Intel, and Microsoft.

References

- [1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. "Adaptive Optimization in the Jalapeño JVM," Proceedings of 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00), Minneapolis, Minnesota, October 15-19, 2000.
- [2] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System." Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation. 2000, pp. 1-12.
- [3] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo." HP Labs Technical Report HPL-1999-78.
- [4] Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows." Fourth ACM Workshop on Feedback-Directed and Dynamic Optimization, 2001.
- [5] W-K. Chen, S. Lerner, R. Chaiken, D. Gillies. "Mojo: A Dynamic Optimization System." Third ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000, pp. 81-90.
- [6] D. Deaver, R. Gorton and N. Rubin. "Wiggins/Redstone: An On-line Program Specializer." Proceedings of IEEE Hot Chips XI Conference, August 1999.
- [7] Peter J. Denning. "Before Memory was Virtual." From the book *In the Beginning: Recollections of Software Pioneers*, IEEE Press, 1997.
- [8] Kim M. Hazelwood and Thomas M. Conte. "A Lightweight Algorithm for Dynamic If-Conversion during Dynamic Optimization," Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00) Philadelphia, PA. October 2000, pp. 71-80.
- [9] W. Hwu, et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*. Boston: Kluwer Academic Publishers, May 1993.
- [10] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu. "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," Proceedings of the 26th International Symposium on Computer Architecture, May, 1999, pp. 136-147.
- [11] Michael D. Smith, "Dynamic Optimization: An Online Opportunity." Keynote Speech. 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00) Philadelphia, PA. October 2000.
- [12] Michael D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," Proceedings of the ACM Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00) Boston, MA. January 2000.
- [13] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation.
<http://www.spec.org/osg/cpu2000/>.
- [14] Omri Traub, Glenn Holloway, and Michael D. Smith. "Quality and Speed in Linear-Scan Register Allocation," Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 142-151, June 1998.