Balancing Memory and Performance through Selective Flushing of Software Code Caches

Apala Guha  Kim Hazelwood  Mary Lou Soffa
Department of Computer Science
University of Virginia

ABSTRACT
Dynamic binary translators (DBTs) are becoming increasingly important because of their power and flexibility. However, the high memory demands of DBTs present an obstacle for all platforms, and especially embedded systems. The memory demand is typically controlled by placing a limit on cached translations and forcing the DBT to flush all translations upon reaching the limit. This solution manifests as a performance inefficiency because many flushed translations require retranslation. Ideally, translations should be selectively flushed to minimize retranslations for a given memory limit. However, three obstacles exist: (1) it is difficult to predict which selections will minimize retranslation, (2) selective flushing results in greater book-keeping overheads than full flushing, and (3) the emergence of multicore processors and multi-threaded programming complicates most flushing algorithms. In this paper, we present a partial flushing approach aimed at reducing retranslation overhead and improving overall performance, given a fixed memory budget. Our technique applies uniformly to single-threaded and multi-threaded guest applications.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—memory management (garbage collection), optimization, run-time environments

General Terms
Algorithms, Design, Experimentation, Management, Measurement, Performance

Keywords
dynamic binary translation, software dynamic translation, virtual execution environments, code cache, flushing, eviction

1. INTRODUCTION
Dynamic binary translators (DBTs) form a software layer of abstraction between the guest application and the operating system and hardware to monitor and translate guest application instruction streams. This capability enables services such as runtime security [7, 20], dynamic optimization [4] and dynamic instrumentation [21]. These services are important across all platform types such as servers, desktops and embedded systems. Some DBT services such as dynamic power management [27] and dynamic scratchpad management are especially important in embedded systems. However, DBTs have a high memory overhead of about 5-10 times the native instruction footprint of each guest application [16, 19]. The high DBT memory footprint is due to three components: 1) code regions that are translated and stored in a software code cache, 2) auxiliary code that is also stored in the code cache for maintaining control over the guest application and 3) data structures for supporting the code cache.

Traditionally, DBT memory demand has been controlled by placing a limit on the size of the code cache. DBTs stay within the memory limit by flushing translated code regions and their corresponding auxiliary code and data structures throughout execution. However, flushing gives rise to performance overhead in two ways: 1) book-keeping must be done for each flush and 2) flushed code regions may need to be retranslated. Therefore, the memory problem is replaced by a performance degradation problem.

It is desirable to reduce the performance overhead as much as possible. Among the two sources of flushing overhead, retranslation is by far the most prominent one. Ideally, flushing should be done in such a manner that retranslations are minimized for the given memory limit.

Selectively flushing code regions that will not be used in the future, or at least in the near future, can reduce retranslation. However, there are several challenges in selective flushing. The main challenge is that it is difficult to dynamically select which code regions to remove because expensive profiling is needed. A by-product of selective flushing is that it complicates code cache management by forming holes (fragmentation) in the code cache. The issue has been further complicated by recent trends towards multicore architectures and multi-threaded programming. Code caches for multi-threaded guest applications are shared by all threads. Thread-shared caches must ensure that no thread is executing in the code regions selected for eviction. DBTs fulfill this condition by checking that each thread that was executing in the selected code regions exits to the runtime once and is
never allowed to return to the selected code regions. Such monitoring of threads adds to the complexity of flushing. Another complication is that it is difficult to know which threads were executing in the selected code regions in the first place. Due to these challenges, full flush (no selection) has become the standard. Partial flushing has been studied, but for single-threaded caches [16].

The goal of this research is to use partial flushing to reduce the retranslation overhead leading to overall performance improvement. It is also our goal for our flushing technique to be applicable to code caches for both single-threaded and multi-threaded applications. We use profiling to select code regions to evict. We also address all the challenges of profiling, code cache management, and thread management associated with partial flushing.

We use an approximation of the LRU (least recently used) heuristic to select code regions for eviction. LRU is a popular replacement algorithm and has been successfully applied in scenarios such as hardware caches. We group code regions that have not been used recently, but we do not try to rank within this group. The code cache is treated as a circular buffer and the LRU code region that is closest to the insertion pointer in this circular buffer is evicted. We identify LRU code regions by profiling.

Traditionally, large amounts of code have been flushed at well-defined flush points. This method frees up much more space than required at a time, but is the preferred method because it reduces book-keeping complexity. Code regions belonging to these groups may get retranslated, but some retranslations can be avoided if the minimum amount of code is flushed at a time. Therefore, we allow code regions to live for as long as possible by evicting only if there is a new code region demanding space. Only as much space as is required by the new code region is reclaimed. We profiling each code region until it is evicted or it is determined that the code region should not be evicted. We enable such continuous LRU profiling with reasonable overhead.

We also manage the fragmentation formed by partial eviction. The code cache manager scavenges for space between code regions in the fragmented code cache. This adds to the book-keeping overhead and we ensure that the overhead is reasonable.

Finally, we solve the problem of managing threads during evictions. Our strategy initially selects all code regions for eviction and uses profiling to discard from the eviction set, rather than the other way around. We can evict code regions after each thread has exited the code cache once. Code regions move in and out of the eviction set. Code regions cannot be re-entered when they belong to the eviction set. We design data structures and algorithms to efficiently support this mechanism.

For this paper, we specifically target and evaluate on embedded platforms, where the memory pressure is expected to be the most severe. Embedded systems such as PDAs and netbooks are increasingly suffering from memory pressure as they get closer to general-purpose systems in terms of the applications they support. We evaluate on both an ARM-based PDA and an ATOM-based netbook. We also analyze the potential for translation time reduction in our benchmarks. We evaluate how much of this potential we are able to exploit and why.

The specific contributions of this research are the following:

- A partial flushing technique that improves code cache performance for both single-threaded and multi-threaded applications, given a fixed memory bound.
- Demonstration that LRU can be effectively used for selective flushing and design of an efficient continuous LRU strategy for software code caches.
- Development of a code cache manager to handle fragmented software code caches.
- Evaluation of the impact of the flushing technique including a comparison of the overheads and the benefits, in two different embedded environments.
- Analysis of the potential for improvement in each benchmark and how much potential we exploited.

We provide background on DBTs and flushing technology in Section 2. Next we describe our proposed technique in Section 3. We discuss design issues in Section 4. We evaluate and analyze the performance of our technique in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2. BACKGROUND

Figure 1 is a simplified diagram of a translation-based, process-level DBT. The core of the DBT is a translator responsible for translating the guest application code dynamically. The translator caches translated code which executes natively from the software code cache. Code translation is performed on demand and requests have to be generated for translations of new code. There are repeated context switches between the translator (for translation of new code) and the code cache (for execution of translated code). The context switch involves saving and restoring state. DBTs create and cache exit stubs (constituents of auxiliary code) to facilitate context switches.

Code is translated into program traces containing one or more basic blocks. These traces have a single entry and one or more exits (one trace exit for each basic block). Trace exits initially target exit stubs. Control passes from the trace exit to the exit stub and then to the translator. Although exit stubs are crucial for correct functionality, there is a performance penalty to context switch to the translator every time a branch is executed. Thus, branches are patched to directly point to their target code if available in the code cache in a process known as linking. Linking is possible only for a direct branch, i.e., a branch whose target does not change during the execution. For indirect branches (for
example, returns), the code cache locations being targeted by the branch are observed and stored as future predictions.

DBTs also use data structures, as shown in Figure 1. The main data structure is a code cache directory, which stores an entry for each cached trace. Each entry contains the original program address and the corresponding code cache address of a trace. The translator searches the directory for an existing translation in the code cache before translating code at a requested program address. Data structures also record how traces are linked, since branches may need to be unlinked if the target trace is ever removed from the code cache (flushed). Lists of incoming and outgoing links of a trace are associated with its code cache directory entry.

DBTs host both single-threaded and multi-threaded guest applications. For single-threaded guest applications, the single thread alternates between executing in the code cache and in the runtime. For multi-threaded guest applications, there are two choices. A thread-private code cache can be allocated to each thread. However, this option has been found to be very inefficient in memory even for general-purpose platforms [8, 15]. Therefore, a single thread-shared code cache is allocated. Simplicity of code cache management is traded off for memory efficiency, in the second choice. Multiple threads simultaneously execute in the code cache. However, for simplicity, only one thread at a time is allowed to execute the runtime in many DBTs [15]. Such a design choice does not degrade performance significantly because the runtime is expected to execute for only a small fraction of the total execution time.

Code caches are typically size-limited. Flushes occur upon reaching the size limit, to make space for new traces. Flushing is fairly simple for a single-threaded code cache. When the thread enters the runtime and requests more space, the runtime may determine that a flush will be needed before allocating the space. Since there is a single thread and that thread is executing the runtime, the runtime is certain that no threads are executing in the code cache. In case of a full flush, the code cache can be immediately deallocated. In case of a partial flush, the incoming links to the selected traces have to be removed before reclaiming the space. In addition, code cache entries corresponding to the evicted traces are discarded to ensure that the runtime does not dispatch the thread to any of these traces anymore.

For a thread-shared code cache, the runtime needs to ensure that no threads are executing in the selected traces during a flush. Traces selected for eviction are considered to belong to the old cache generation while all other traces are considered to belong to the new cache generation. As shown in Figure 2, the runtime unlinks old traces to expedite the exit of threads. Unlike single-threaded code caches, unlinking is needed for both full and partial flushes. The runtime ensures that each thread that was executing in the old traces have exited from them once. It avoids dispatching threads into the old traces by discarding the corresponding code cache entries. When all the threads that were executing in the old traces have exited, the old traces can be discarded. The threads that exit the old traces are not blocked, to avoid deadlock. Instead, these threads are dispatched to new generation traces that are not marked for eviction. In the case of full flush, threads are dispatched to new traces in a newly allocated code cache. For a partial flush, both the old and new generation traces share the same code cache. Since the old and new generation traces coexist for some time, the sum of their sizes must be within the memory limit. Thus, the flush is triggered some time before reaching the memory limit i.e., at a high water mark.

3. SELECTIVE FLUSHING

Figure 3 shows the conceptual differences between a traditional full flush and our selective flushing technique. In both cases, a trace is initially in the active (threads are executing it), linked state. A flush is triggered upon reaching the high water mark. Traces are unlinked although they may continue to be active. When all threads have exited the traces i.e., the traces have become inactive, the two techniques begin to diverge. For a traditional full flush, the traces are immediately evicted. In our technique, traces begin to be profiled for execution. If there is a request for execution of an inactive trace, it is promoted to the active state by dispatching the requesting thread to it. Linking to this trace is again allowed. However, if the trace does not become active, it will eventually get overwritten by new traces. The mechanisms of flush triggering, profiling, promoting and allocating space are described in Section 3.1, Section 3.2, Section 3.3 and Section 3.4 respectively. Figure 4 shows the successive states of the code cache when our partial flush technique is applied. Figure 4(a) is the key for understanding the different code cache states.

3.1 Triggering Flushes

As mentioned in Section 2, a flush is triggered at the high water mark. All traces are unlinked to expedite the exit of threads. In a traditional full flush, all code cache directory entries corresponding to the unlinked traces are discarded, so that these traces cannot be located or re-entered. However, in our technique, there is a possibility of re-entry. Therefore, the code cache directory entries are not discarded. Re-entry must still be disallowed until all threads exit once. Thus, additional data structures record whether the trace belongs to the current generation or to the old generation. Initially, all traces are in the current generation. Upon unlinking, the traces are marked to belong to the old generation. Old generation traces cannot be re-entered without first promoting them to the current generation. Figure 4(b) shows the state of the code cache at a flush trigger point. The code cache contains old generation traces and some free space.

3.2 Profiling Traces

Continuous LRU is our profiling strategy. When all threads
3.3 Promoting Traces

Trace promotion to the active state simply implies changing its generation tag from the old to the current generation, and re-enabling trace linking. Threads can enter promoted traces. However, our promotion is in-place (the trace is not moved from its original position), which gives rise to fragmentation within the code cache. Figure 4(c) shows the resulting situation. New traces are being inserted into the free area of the code cache. At the same time, scattered old generation traces are being promoted to the new generation.

3.4 Allocating Space to Traces

Traces are inserted into the contiguous, free area of the code cache in Figure 4(c). However, upon reaching the state shown in Figure 4(d), there is no more contiguous free space in the code cache to allocate from. From this point forward, the code cache manager must search for free spaces in a fragmented code cache. The code cache manager treats the code cache as a circular buffer. It can assign already free space to a trace. It can also overwrite old generation traces if they are inactive. The code cache manager overwrites as few traces as possible, to allow profiling for longer periods of time.

The code cache manager may not be able to use all holes as some may not be large enough. Thus, the situation in Figure 4(e) results. The traces in unused holes continue to exist in the code cache and may get promoted or overwritten in the future. When the sum of the sizes of the current generation traces crosses the high water mark, the code cache manager triggers the next flush as shown in Figure 4(f). All existing traces are marked to be of the previous generation, as shown in Figure 4(g).

Figure 4(g) shows that the code cache is full of old traces which are being executed by threads. Therefore, it may seem that the code cache manager is unable to allocate space to new traces until all threads have exited the code cache once. However, there are really two kinds of traces in the old generation now. The first is active while the second has been inactive since the previous flush point. These inactive traces may be overwritten to allocate space to new traces. To do so, the code cache manager must be able to distinguish between active, old traces and inactive, old traces. Therefore, the trace generation tag must have three possible values: 1) current, 2) old and active and 3) old and inactive.

4. DESIGN ISSUES

In this section, we will discuss the design issues we faced and how we resolved them. Section 4.1 describes how we arrange traces. Section 4.2 describes the data structure and algorithm used by the code cache manager to allocate space. Finally, Section 4.3 describes two different and popular linking strategies employed by DBTs and how we adapt our partial flushing technique for each of them.

4.1 Trace Arrangement

Figure 5 shows the different arrangements of traces and their exit stubs within the code cache – separated and contiguous. The separated arrangement in Figure 5(a) has been found to exhibit better performance [18]. However, the problem with this arrangement is that traces and their exit stubs are not co-located. It has to be ensured that when some exit stub is deleted to make space, the corresponding trace is also deleted. As the traces and exit stubs are not co-located, it is more complicated to ensure this condition. However, the arrangement in Figure 5(b) does not suffer from this problem. The traces and exit stubs are co-located and can be deleted together. We therefore employ this arrangement, and ensure that any resulting performance degradation is outweighed by the performance improvement of partial flushing by comparing with a baseline that uses the arrangement in Figure 5(a).
4.2 Managing Fragmentation

The code cache manager needs to know the locations of the various traces in the code cache to be able to scavenge for space. This information is already available in the code cache directory. However, the code cache directory is searched and sorted using the original program address of the trace as the key. The original program address of a trace has no relation with the actual code cache address of that trace. Therefore, there also needs to be a directory in which the code cache entries can be searched and sorted using their code cache addresses. Replicating all code cache entries requires a large amount of memory, and maintaining consistency between the replicas of the code cache entries adds to the complexity. Instead, we form a directory of pointers to the code cache entries and search and sort them using the code cache address as the key. We name this directory the code cache map. The code cache map is updated whenever a trace is inserted or evicted.

Figure 6 shows the algorithm for finding space. The code cache manager initially holds a pointer to the first trace in the code cache map and also has an initial hole. The manager iterates over traces in the code cache map. For each trace it tests whether the trace can be added to the existing hole. If the trace is not deletable or the trace is not contiguous with the existing hole, the trace cannot be added and the code cache manager discards the existing hole as too small. The discarded hole may get filled in a future pass. The manager also tries to start a new hole at or after the current trace, depending on whether or not the trace is deletable. If at any step, the manager finds the hole to be large enough, it immediately stops. It allocates space from the hole and adjusts the hole. Before exiting, the manager saves the current state of the hole and the current cache map pointer so that it knows where to resume searching.

4.3 Trace Linking

The design of our proposed technique interacts with the linking strategy of the runtime. As traces move through the various states shown in Figure 3(b), we have to ensure that the linking policy of the runtime is being enforced. To this end, we implement our technique for two different linking policies - lazy and proactive. Figure 7 illustrates the two linking policies.

Lazy linking records and places a link between the source and the target when the path first traverses. As shown in Figure 7(a), the runtime implements lazy linking by storing the location of each branch in its corresponding exit stub. The runtime reads the exit stub for arguments such as the target address. In the case of lazy linking, the runtime uses the exit stub to also locate the branch to link. The runtime registers the link with the code cache entries of the source and target traces when it places the link. Link data structures are discarded upon an unlink and reinstated when the path is again traversed. Lazy linking integrates seamlessly with our technique and no special handling is needed.

However, applying our partial flush technique is more complicated in the case of proactive linking. Proactive linking places the link as soon as the source and target traces appear in the code cache, regardless of whether the path will ever
be traversed or not. As shown in Figure 7(b), the runtime implements proactive linking by examining all the outgoing branches of the trace being translated. Each outgoing branch is registered with the code cache entry of the target trace. If the target trace has not been inserted into the code cache yet, a tentative code cache entry for the target trace is formed and the outgoing branch is registered with the tentative entry. When the target trace is eventually inserted into the code cache, the runtime updates the tentative code cache entry with relevant information such as code cache address. Also, all registered links are immediately placed. The code cache entry is the only way to locate branches to be linked (exit stubs do not duplicate this data to save memory). Also the links are registered with code cache entries only once, when a trace is being translated. We have to ensure that this information is not lost as traces get evicted and retranslated.

When a trace is evicted in a proactive linking runtime, we examine if it has any registered incoming links. If not, the code cache entry can be deleted. If there are registered incoming links, we merely change the code cache entry to a tentative one by invalidating fields such as code cache address. This method ensures that link information is preserved. However, the source trace may get evicted in the meantime. In such a case, the link information that is being preserved so carefully will become stale. Therefore, before placing each link, it has to be checked that the source trace still exists. The link must be removed if found to be stale. Therefore, a trace moving from the inactive to the active state must re-register its outgoing links, in case they have been removed as stale.

Apart from direct links, all information for indirect branch prediction has to be removed on an unlink. However, indirect branch prediction resembles lazy linking in principle i.e., predictions are added as they are found. Therefore, indirect branch handling also integrates easily with our technique.

5. PERFORMANCE EVALUATION

The goal of our evaluation is to compare the execution times of traditional full flush and our proposed partial flush technique. The execution time has several components: 1) translation of a trace and insertion into the code cache, 2) context switching between the code cache and the translator, 3) execution within the code cache, 4) flushing overhead and 5) indirect branch handling. Our goal is also to investigate how each of these components contributed to the change in total execution time. For the most important contributors, we will also investigate why there was a change. Also, we will identify benchmark characteristics which can indicate potential for improvement through partial flushing. Some such characteristics are code cache pressure and the number of retranslations needed by the benchmark. For benchmarks which do not have a lot of potential to improve, we have to ensure that we do not produce too much overhead by applying our partial flushing technique.

We describe our evaluations in the following sections. Section 5.1 evaluates and discusses the performance of our proposed technique when applied to single-threaded code caches. Section 5.2 explores the performance of our proposed technique when applied to thread-shared code caches.

5.1 Single-Threaded Evaluation

For our experimental environment, we used an an iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. The iPAQ has a 200 MHz StrongARM-1110 processor with 64 MB RAM, 16 KB instruction cache and a 8 KB data cache. For the dynamic binary translator, we used Pin [14] for ARM. We implemented and used a lazy linking policy in the runtime for this set of experiments. The runtime uses a code cache limit of 256 KB and triggers a flush when the code cache is 100% full. For our test programs, we used two different benchmark suites: 1) the MiBench [13] suite with large datasets and 2) the SPEC2000 [17] integer suite with test inputs (there was not enough memory on the embedded device to execute larger inputs, even natively). In all these experiments, we are really interested in improving the performance of long-running benchmarks given a fixed memory budget. Therefore, we did not consider benchmarks with baseline execution times below 100 seconds. This decision eliminated some MiBench and SPEC2000 benchmarks.

Figure 8 shows the normalized execution times for the single-threaded benchmarks with partial flushing. All the benchmarks show some improvement in execution time, the average being about 17%. This speedup over full flush shows that the overheads of execution in the unlinked mode and

```plaintext
1 Initial conditions:
2 holeStart = <start of code cache>
3 holeEnd =
4 <start code cache address of first trace>
5 cacheMapPointer =
6 <pointer to code cache entry for first trace>
7
8 Algorithm:
9 if (existing hole large enough)
10 allocate space from existing hole
11 adjust hole
12 return
13
14 for trace pointed by cacheMapPointer
15 if (trace is not overwriteable or
16 not contiguous with existing hole)
17 account hole in used up space
18 discard hole
19 else //if trace is overwriteable and
20 //contiguous with existing hole
21 add the trace area to existing hole
22 if (trace is last trace in code cache)
23 add space between end of trace and
24 end of code cache to hole
25 else //if trace is not the last trace
26 add space between end of trace and
27 start of next trace to hole
28 if (existing hole large enough)
29 allocate space from existing hole
30 adjust hole
31 return
32 advance cacheMapPointer to next trace
33 goto line 14
```

Figure 6: Algorithm for managing fragmentation. The terms first, last and next should be considered in the spatial (rather than temporal) context.
Figure 7: Lazy linking and proactive linking. A branch in source trace S targets trace T. The branch is initially unlinked (pointing to exit stub for T) and may eventually get linked (pointing to T).

Figure 8: Execution time for our proposed partial flush technique with respect to full flush. We reduce execution time by about 17% on the average, for single-threaded benchmarks.

Figure 9: Fraction of speedup resulting from each DBT task. Reduction in translation time is the greatest contributor of speedup.

extra book-keeping needed by partial flush are outweighed by the improvements in translation time.

We also studied the source of the speedup by splitting up the total execution time into components. Figure 9 shows the fraction of execution time reduction caused by each component. The components on the positive side of each bar contributed to speedup while components on the negative side contributed to slowdown. The effective speedup is calculated by subtracting the total bar height on the negative side from the total bar height on the positive side. The effective speedup in Figure 8 and Figure 9 may not exactly match because the results in Figure 9 are somewhat contaminated by profiling time. We measured the difference between actual execution time and profiled execution time for each benchmark and found that profiled execution time is higher than the actual execution time by 8% on average.

From Figure 9 it is clear that the main contributor to speedup is the reduction in translation time, resulting from fewer retranslated traces. The next most important contributor is context switch time, though it contributes to slowdown because there are more context switches. The reason for having more context switches is that the code cache suffers from fragmentation during partial flush and can accommodate fewer traces compared to full flush. As a result, a trace in a partial flushing system survives through more code cache generations on an average. Surviving across each generation implies there will be one context switch to promote the trace and there will be one context switch for placing each link to the trace, leading to more context switches overall. It is worth noting that not much extra time is spent in flushing i.e., the book-keeping overhead of our proposed technique is small. Application execution and indirect branch handling time also remain fairly stable.

In order to understand why the reduction in translation time varies among benchmarks, we studied the cache pressure on each of them. The cache pressure is the ratio of the unlimited cache size of a benchmark to the memory limit. Figure 10 shows that cache pressure varies from about 1(bzip2) to 10(gcc). A lower cache pressure indicates that fewer retranslations are needed and there is less room for improvement. Indeed vpr and lame were among the benchmarks with the lowest cache pressure and also registered the least improvement in translation time and the least speedup.

We also measured the impact of fragmentation. Our goal was to ensure that fragmentation does not increase as generations progress. If fragmentation steadily increases, partial flushing will become useless after a point i.e., our technique will not scale. Figure 11 shows the fragmentation in the code cache for gcc, the largest benchmark. Fragmentation remains fairly stable across generations and within 10%, showing that our technique is scalable. Although the amount of memory lost in fragmentation is unusable by the system, it still improves performance compared to full flush.

5.2 Multi-Threaded Evaluation

For our experiments on thread-shared code caches, we used an ATOM N270 netbook with a 1.6GHz processor supporting two hardware thread contexts. The processor has a
Figure 10: Cache pressure for single-threaded benchmarks. Cache pressure is the ratio of the unlimited code cache size to the memory limit (256 KB in this case).

Figure 11: Code cache fragmentation for the largest benchmark gcc. Fragmentation remains stable and usually below 10%.

32KB instruction cache, 24KB data cache with write-back and a 512 KB L2 cache. The memory size is 1 GB. It supports Linux kernel 2.6.24. For the ATOM-based netbook, we used Pin [15, 21] targeting the x86 architecture. The runtime in this case implemented proactive linking. The runtime uses a code cache limit of 512 KB and triggers a flush when the code cache is 70% full (unless otherwise stated). We used the PARSEC [5] suite with native inputs. PARSEC consists of multi-threaded benchmarks and we executed them on the netbook with two threads.

We first explored which of the multi-threaded benchmarks would need flushing activity for the given memory limit. Figure 12 shows the ratio of the unlimited cache size of each benchmark to the given memory limit. Benchmarks will undergo flushing only if their cache size crosses the high water mark. Therefore, in our case, if the cache pressure is at least 0.7, we expect flushing to occur. The benchmarks in this category are canneal, bodytrack, fluidanimate, freqmine and facesim. For the other two benchmarks, blackscholes and svoptions, we are interested in ensuring that overhead is reasonable rather than obtaining improvements.

Figure 13 shows the normalized execution times for the benchmarks. Average-small is the average normalized execution time for the small benchmarks, i.e., the benchmarks which do not undergo flush activity. Average-large is the average normalized execution time for the large benchmarks. Figure 13 shows that many trace translations we have reduced using our technique is more pronounced in canneal.

We measured the fragmentation impact for multi-threaded benchmarks. Our goal was to ensure that fragmentation does not increase as code cache generations progress. Figure 15 shows the fragmentation for the largest benchmark, facesim. The fragmentation is stable, showing that our technique is scalable for thread-shared caches.

6. RELATED WORK

DBTs have been developed for a wide range of platforms. For example, Dynamo [4], DynamoRIO [7], Strata [18, 25], Valgrind [23] and Pin [21] target general-purpose computing platforms. Although development of DBTs for embedded systems has been limited in comparison, some systems such as Pin [14], DELI [9] and Strata [1, 3, 22] exist. DBTs provide services such as optimization [4], instrumentation [21] and security [7, 20].

Memory management policies for DBTs have been researched before. Thread-shared software code caches [8, 15] emerged as a memory optimization over thread-private caches, at the cost of increased complexity. Given these efficient systems for supporting thread-shared caches, our research differs from previous work in two ways. First, many previous memory management policies have been designed for purposes other than reducing memory demand. For example, Dynamo [4] triggers a cache flush when the rate of trace generation becomes too high, to improve performance. This is a full cache flush which is executed for performance reasons and not for memory constraints. DynamoRIO [6, 8] manages the code cache only for consistency events such as self-modifying code and not for capacity. They also dynamically detect the working set size and adaptively size the code cache. But they only scale up the code cache limit adaptively, which may not be suitable in a memory-constrained environment. The second reason our research differs from previous memory management work is that most of the prior policies target full or partial eviction for single-threaded code caches. Strata [1, 2, 3] considers partial flushes for single-threaded benchmarks on an embedded platform. Similarly, generational partial code cache eviction schemes to limit memory demand have been studied before [11, 16], but only for single-threaded code caches.
Traces. We have designed an efficient code cache manager: using scratchpad and main memory in dynamic synthesis for embedded systems [12].

![Figure 12: Cache pressure for multi-threaded benchmarks. Cache pressure is the ratio of the unlimited code cache size to the memory limit (512 KB).](image1)

![Figure 13: Execution time for partial flush applied to thread-shared caches, normalized with respect to full flush. Average speedup is 15%.](image2)

![Figure 14: The number of trace translations normalized with respect to full flush.](image3)

![Figure 15: Code cache fragmentation for the largest benchmark facesim. Fragmentation remains stable and within 3%.](image4)

Pin [14, 15] accounts for both single-threaded and thread-shared code caches, but only supports a full flush. Approaches other than flushing have been studied for memory management. The approach that is closest to flushing is code compression [1, 26]. Compression has a lower retranslation cost compared to eviction but it frees less space at a time. Compression also faces the same challenges as eviction such as selecting which traces to compress, code cache management and thread management. Storage of code on a server and loading into an embedded system on demand has also been explored [24, 28, 29]. Some other approaches include exit stub optimization [10] and path selection optimization [12] within each code cache generation. Client-server approaches and approaches that apply within cache generations can be combined with our technique to improve performance further.

7. CONCLUSIONS

We have demonstrated that partial flushing is more efficient than traditional full flushing, for a fixed memory budget. We have demonstrated that LRU is an effective heuristic for selective flushing. We have designed a continuous LRU profiling strategy that can efficiently select traces. We have designed an efficient code cache manager for software code caches. Also, we have designed an efficient thread management technique so that partial flushing can be applied to single-threaded as well as thread-shared software code caches. We found that we improve more performance in benchmarks with higher cache pressure. This fact is especially encouraging, because as the cache pressure gets higher, the performance degradation produced by flushing is expected to be more severe. Most of our performance gain was from improving the translation time. We improved translation time by reducing the number of retranslations. The benefits of our technique outweighed the associated overheads of code cache management, profiling in unlinked mode and trace arrangement. For single-threaded code caches, we improved performance by 17% on the average. For thread-shared code caches, we improved performance by 15% on the average.

8. REFERENCES


