Memory Optimization of Dynamic Binary Translators for Embedded Systems

APALA GUHA, KIM HAZELWOOD, and MARY LOU SOFFA, University of Virginia

Dynamic binary translators (DBTs) are becoming increasingly important because of their power and flexibility. DBT-based services are valuable for all types of platforms. However, the high memory demands of DBTs present an obstacle for embedded systems. Most research on DBT design has a performance focus, which often drives up the DBT memory demand. In this article, we present a memory-oriented approach to DBT design. We consider the class of translation-based DBTs and their sources of memory demand; cached translated code, cached auxiliary code and DBT data structures. We explore aspects of DBT design that impact these memory demand sources and present strategies to mitigate memory demand. We also explore performance optimizations for DBTs that handle memory demand by placing a limit on it, and repeatedly flush translations to stay within the limit, thereby replacing the memory demand problem with a performance degradation problem. Our optimizations that mitigate memory demand improve performance.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Run-time environments*

General Terms: Design, Measurement, Performance

ACM Reference Format:

Guha, A., Hazelwood, K., and Soffa, M. L. 2012. Memory optimization of dynamic binary translators for embedded systems. *ACM Trans. Architec. Code Optim.* 9, 3, Article 22 (September 2012), 29 pages.
DOI = 10.1145/2355585.2355595 <http://doi.acm.org/10.1145/2355585.2355595>

1. INTRODUCTION

Dynamic binary translators (DBTs) form a software layer of abstraction between the guest application and the host environment (operating system and hardware) to dynamically monitor and translate guest application instruction streams. This capability enables services such as runtime security [Bruening et al. 2003; Hu et al. 2006; Kiriansky et al. 2002], dynamic optimization [Bala et al. 2000] and dynamic instrumentation [Luk et al. 2005]. While these services are important across all platform types, desktops and embedded systems, some DBT services such as dynamic power management [Wu et al. 2005] and dynamic scratchpad management [Miller and Agarwal 2006] that are especially important in embedded systems can be enabled. While it can be argued that embedded systems execute realtime applications that are not suitable for dynamic binary translation, it is also true that there is a large class of embedded application that is not realtime or only soft realtime and these applications can benefit from DBTs.

However, DBTs have a high memory overhead of about 5–10 times the native instruction footprint of each guest application [Hazelwood and Smith 2006; Janapareddi et al. 2007]. As memory is constrained on embedded systems, the high memory demands of DBTs present a problem. The DBT memory footprint has three components: 1) code

Author's address: A Guha; email: apala.guha@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/09-ART22 \$15.00

DOI 10.1145/2355585.2355595 <http://doi.acm.org/10.1145/2355585.2355595>

Fig. 1. Memory distribution among translated code, auxiliary code, and data structures. The results are averages taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin on ARM [Luk et al. 2005].

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. Figure 1 shows the relative importance of the different sources of memory demand within the DBT. For example, on average, translated code, auxiliary code and data structures respectively constitute 23%, 41% and 36% of the total memory demand in Pin, an industrial-strength DBT. Similar data on memory demand has been found [Bruening 2004; Janapareddi et al. 2007] across different DBTs. Therefore, not only is the total DBT memory demand high, but it is also important to optimize all three sources of memory demand.

The memory demand of DBTs is often handled by placing a limit on the DBT memory requirements. The DBT must flush translated code and its corresponding auxiliary code and data structures throughout execution to stay within the memory limit. However, flushing gives rise to performance degradation for two reasons: 1) book-keeping must be done to support flushing and 2) flushed translations may require retranslation. Therefore, in this situation, memory demand manifests itself as performance degradation.

A major challenge to optimizing the DBT memory demand arises from the performance optimizations applied to DBTs. Code caching is a ubiquitous performance optimization that gives rise to memory demand. Code caching necessitates auxiliary code and data structures, which add to the memory demand. Another major challenge is that the impact of DBT design decisions on the memory demand is not clear. Path selection (selection of code for translation and linking of cached code regions) impacts different components of the DBT memory demand in opposing ways. For example, a particular path selection design may increase the translated code size but decrease the data structure size. Finally, memory-limited DBTs also present a challenge for performance improvement. Selective flushing is desirable for such systems, but it is difficult to dynamically select which code regions to flush. A byproduct of selective flushing is that it complicates code cache management by fragmenting the code cache. The issue has been further complicated by recent trends towards multicore architectures and multithreaded programming. For multithreaded guest applications, threads usually share a single code cache for memory efficiency. It is difficult to maintain a consistent code cache state across all threads in a selective flushing system.

Our approach is to use custom memory optimizations to reduce the memory footprint of the DBT and to improve its performance in memory-limited situations. We first design a *balanced* path selection policy that will consider the interactions between the memory demand components and will be beneficial for the memory demand overall, and also the performance in memory-limited situations. After determining the path selection policy, we strive to further tune for better memory efficiency and performance, without introducing further interactions among the memory demand components. In this step, we found auxiliary code to be a good candidate and optimized it in isolation from any other memory demand component. After tuning the memory

Fig. 2. Schematic view of a dynamic binary translator. The code cache consisting of translated code and auxiliary code, and data structures are the sources of DBT memory demand.

demand, we developed a selective flushing approach. In this approach, we reduce flushing overheads, manage code cache fragmentation and develop a flushing technique uniformly applicable to single-threaded and multithreaded guest applications. Finally, this article combines the approaches to provide a comprehensive solution. While the different contributions offer varying amounts of improvement, we evaluate the overall impact of a memory-oriented DBT design.

The contributions of this article are the following:

- A demonstration that path selection is a memory optimization tool and the design of a balanced path selection strategy for memory efficiency and performance.
- A demonstration of the memory demands of auxiliary code and the design of strategies to reduce auxiliary code size.
- The design and implementation of a selective cache flush strategy that applies to both single-threaded and multithreaded guest applications.
- The combination of path selection, auxiliary code optimization, and selective flushing strategies into a full system.
- The evaluation of each strategy separately and evaluation of the combined system on embedded platforms.

In the remaining part of the article, Section 2 provides background on DBT technology. Section 3 describes our research on balanced path selection policies. Section 4 describes further tuning of the memory demand components by optimizing auxiliary code size. Section 5 describes the selective cache flushing technique. Section 6 describes the integrated system. Section 7 provides a survey of related work and Section 8 concludes the article.

2. BACKGROUND

Figure 2 is a simplified diagram of a translation-based DBT. The DBT core 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. The DBT shown in this example is a process-level DBT implemented in software. This is the type of DBT we use for evaluation. However, the techniques can be extended to system-level DBTs and codesigned DBTs that use software code caching.

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. Exit stubs are by far the major component of auxiliary code. The other component of auxiliary code is indirect branch prediction tables. Although exit stubs are crucial for correctness, 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, that is, a branch whose target does not change during the execution. For *indirect* branches (e.g., 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 2. 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 multithreaded guest applications. For single-threaded guest applications, the single thread alternates between executing in the code cache and in the runtime. For multithreaded 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 [Bruening et al. 2006; Hazelwood et al. 2009]. 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 [Hazelwood et al. 2009]. 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 be old traces while all other traces are considered to be current traces. Current traces can constitute existing traces that were not selected for eviction or newly formed traces. As shown in Figure 3, the runtime unlinks old traces to expedite the exit of threads. Unlike single-threaded code caches, unlinking is performed 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 at the beginning of the flush. Also, the fact that they are unlinked ensures that control cannot pass into them from other cached traces. 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 current traces. Since the old and current traces coexist for some time, the sum of their sizes must be within the memory limit.

Fig. 3. Schematic of a thread-shared cache flush. All threads from the old cache must be dispatched to the current cache before eviction.

Fig. 4. Topology of the path selection choices.

Thus, the flush is triggered some time before reaching the memory limit that is, at a *high water mark*.

3. PATH SELECTION FOR HOLISTIC MEMORY EFFICIENCY AND PERFORMANCE

A path consists of traces and links between traces. The sizes of the memory demand components of a DBT depend on *path selection* which denotes the way that code is selected for each trace and the way traces are linked. It is challenging to design a path selection strategy for improved holistic memory efficiency because of the complex interactions among the three sources of memory footprint. For example, some strategies that reduce the translated code size may increase the data structure size as well as the total memory demand and vice versa. Additionally, some path selection aspects may degrade performance when there are no memory constraints, while their improved memory efficiency may improve performance in memory-limited situations.

For designing path selection, we consider all three sources of memory demand while traditionally only the code cache memory demand (translated code and auxiliary code) has been considered. We show that we reach different conclusions by incorporating data structures into the total memory demand.

Our goal is to present a path selection strategy that holistically optimizes all three memory sources (translated code, auxiliary code, and data structures) without degrading performance. We explore the interactions of path selection with memory demand and performance to motivate the design of balanced path selection strategies. We enumerate all aspects involved in a path selection design and evaluate a comprehensive set of approaches for each. Finally, we propose a path selection strategy that balances memory efficiency and performance.

3.1. Path Selection

Figure 4 depicts the path selection design space. When forming a trace starting at a certain program address, the first basic block is fully included, since all instructions are guaranteed to execute. For multiblock traces, it has to be decided whether and

Fig. 5. Example code used to describe different path selection strategies in Section 3.1.

Fig. 6. Translation to single-block and multiblock traces. White boxes are parts of the trace, while gray boxes are exit stubs. The gray boxes are labeled with the corresponding branch targets.

how to extend the trace at the end of each basic block. Similarly, links between traces may be placed *proactively* (aggressively) or *lazily*. We discuss the trade-offs involved in each path selection choice, in the following sections. We use the snippet of code in Figure 5(a) as a running example to explain the configuration choices and their tradeoffs. Figure 5(b) shows the control flow graph corresponding to Figure 5(a). The execution follows path ABD once and then follows path ACD repeatedly before exiting to E.

3.1.1. Single-block vs. Multiblock Translation. Figure 6(a) shows a singleblock trace starting at A. Figure 6(b) is an example of a multiblock trace starting at A. White blocks are part of the trace while shaded blocks represent exit stubs. In Figure 6(b), if B appears on some other program path, B will have to be translated again (duplicated) because side entries to traces are not allowed. Single-block traces will not suffer from such duplication. However, in Figure 6(b), there is only one off-trace branch for A, while in Figure 6(a), there are two off-trace branches for A because both outcomes of a conditional branch need to be handled in translated code. Therefore, single-block traces have more branches and exit stubs per unit of translated code. Also, more links have to be recorded for single-block traces, increasing the proportion of data structures. Another side effect of single-block traces is that there is one code cache directory entry per basic block, increasing the proportion of data structures further. A higher proportion of auxiliary code for single-block traces implies that a smaller proportion of the code cache is available for translated code, leading to lower code cache locality. The lack of duplication for single-block traces also implies that temporally close code may not be spatially close, again leading to lower code cache locality. Moreover, the number of context switches will be higher as code is translated one basic block at a time.

3.1.2. Multiblock Trace Selection and Termination

Trace Selection. We can select the basic blocks nonspeculatively by executing the last basic block to determine the next basic block, or speculating by predicting the next basic block. In Figure 7(a), we execute A and find that B is the next basic block, so

Fig. 7. Extending a trace. White areas are parts of the trace while gray areas are exit stubs.

Table I.

Number of static branches pointing to different types of basic blocks. Results are averaged over the SPEC2000 integer and MiBench embedded benchmarks, hosted by Pin [Luk et al. 2005].

Basic Block Type	Average No. of branches pointing to Basic Block	Memory Savings
Fall throughs of conditional branches	0.082	5%
Targets of conditional branches	1.429	-41%
Targets of unconditional branches	3.219	-14%

we append B to A. In Figure 7(b), we speculate that C is the next likely basic block (from offline profiling data, for instance) and we append C to A. There are more context switches in forming nonspeculative traces as these are translated one block at a time. However, if the speculation is incorrect, there will be wasted space for translated code and data structures.

Trace Termination. After translating each basic block, the translator must determine whether to extend the trace further. It should ideally continue to extend the trace if the next basic block executes only in this single program path because it will not be duplicated elsewhere and will obviate the need for data structures. However, it should instead start a new trace if the basic block executes in other program paths because it will be duplicated otherwise. For example, suppose if both B and C are targeted by basic blocks other than A, we should produce the trace in Figure 6(a). However, if B always follows A, we should produce the trace in Figure 6(b).

We experimentally explore the trace termination condition separately because the number of program paths executing a basic block is independent of other aspects of path selection. Therefore, separate evaluation can reduce the number of path selection combinations. We found that the number of program paths that execute the next basic block is correlated with the type of the branch ending the last translated basic block. As shown in Table I, we found that if the last branch is a not taken, conditional branch, its fall-through basic block is rarely targeted by branches, while if the last branch is taken, its target basic block is usually targeted by multiple branches. We compared the memory demand of extending at each type of branch with the memory demand for single-block traces (no extensions allowed). As shown in Table I, extending the trace at not-taken, conditional branches offers memory savings of 5% while extending the trace at taken, conditional branches and unconditional branches offers memory losses of 41% and 14% respectively. Therefore, we terminate traces at taken branches.

3.1.3. Link Formation. There are two popular linking strategies; lazy and proactive. Assume, that in Figure 5(b), basic block E is translated before the loop ever executes. *Lazy linking* places the link when a path first traverses. Figure 8(a) shows an example of lazy linking. As shown in Figure 8(b), lazy linking is implemented by storing the location of the branch in the exit stub. The translator uses the location of the branch to determine where to patch. The translator stores data about the link in the source and target code cache entries only after it places the link.

Fig. 8. Proactive versus lazy linking.

Proactive linking places a link as soon as the source and target traces appear in the code cache, regardless of whether the path will ever traverse. Figure 8(c) shows an example of proactive linking. When a trace is being translated, a proactive linking configuration examines each of the trace's outgoing branches. As shown in Figure 8(d), each potential link is recorded. If the target trace is not yet in the code cache, a tentative code cache entry for the target is formed and a tentative incoming link is registered with the target. When the target eventually gets translated, the tentative code cache entry is updated with the code cache address of the target. The potential links to the target are already associated with the code cache entry and can be immediately placed, thereby enforcing a proactive linking policy. If the target is already in the code cache when the source is being translated, the link is placed at the same time it is recorded.

Lazy linking needs less data structure space than proactive linking because proactive linking uses tentative code cache entries and tentative links which lazy linking does not. Some of the code cache entries and links created by proactive linking may never get used. However, lazy linking stores branch locations in exit stubs resulting in larger exit stubs than proactive linking, leading to larger auxiliary code. From the performance perspective, proactive linking anticipatorily links traces and is more effective in reducing the number of context switches between the code cache and the translator. The larger exit stubs used by lazy linking increase the proportion of auxiliary code in the code cache, leading to a reduction in code cache locality.

3.2. Experimental Evaluation

We use the nomenclature shown in Table II for the trace selection strategies. We implemented and evaluated two strategies for speculative trace selection. In the first strategy, we use data gathered in an offline profiling run to speculate which way a branch will go. We speculate about a branch only if it shows a particular bias for at least 90% of its executions in the profiling run. Although an offline profile-based strategy is not practical, we evaluate it to explore a comprehensive spectrum of path selection strategies. In the second strategy, we translate a contiguous stream of code until the trace size reaches a certain threshold size or it encounters an indirect or direct, unconditional branch. Such a strategy is equivalent to speculating that no conditional

Table II. Trace Selection Strategies and Their Descriptions

Strategy	Description
Single	Single basic block trace
Dynamic	Nonspeculative selection of multiblock traces
Threshold-based	Speculative selection of contiguous code up to a threshold size for multiblock traces
Profile-based	Speculative selection based on profile data for multiblock traces

branch will be taken. The speculative strategy using profiling is highly informed, while the second strategy uses minimum information. Each of the trace selection strategies will be combined with both lazy and proactive linking, to form path selection strategies.

We evaluated the memory and performance effects of the different path selection strategies. For memory effects, we measured the sum of the space occupied by the translated code, auxiliary code, and the data structures. For performance, we measured the execution times of applications hosted by a DBT. In the performance experiment, we used a uniform limit of 512 KB on the total memory demand.

We used threshold-based trace selection combined with proactive linking as our baseline. We chose this baseline because it is used by Pin [Luk et al. 2005], a production-quality DBT. The memory and performance measurements are normalized with respect to the baseline.

We used Pin for XScale [Hazelwood and Klauser 2006] as our DBT. We implemented our strategies by directly modifying the Pin source code. The SPEC benchmarks were run on test inputs, since there was not enough memory on the embedded device to execute larger inputs (even natively). The MiBench benchmark suite provides large and small input datasets for the benchmarks. We used the large inputs in our experiments. To calculate the average, for all experiments we used a weighted arithmetic mean as described in literature [John 2004]. This is our experimental environment throughout the article, unless mentioned otherwise.

We ran the SPEC2000 integer [Henning 2000] and MiBench embedded benchmark [Guthaus et al. 2001] suites on a 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. Although newer platforms have larger memory sizes, the applications targeting newer platforms consume more memory than the available benchmarks. Therefore, similar problems apply to newer platforms and our techniques can be used to mitigate these problems.

We divided the benchmarks into three groups—short-running (0–100s), medium-length (100–1000s) and long-running (1000+ s), according to their baseline execution times. We categorize the benchmarks because longer benchmarks are better able to amortize translation overheads and the effects of DBT optimizations become clearer as the benchmarks run longer.

We present the results on memory efficiency in Section 3.2.1 followed by the results on performance in Section 3.2.2. We discuss the results and propose a path selection strategy in Section 3.2.3.

3.2.1. Memory Efficiency. Figure 9 shows the normalized memory demands of the benchmarks. Figure 9(a) presents the total memory demand for all the benchmarks. There are few intersections in the graph indicating that there is a consistent ranking among the different strategies for most of the benchmarks. Therefore, we use the summary graphs of Figure 9(b) and Figure 9(c). Figure 9(b) shows how the memory efficiency would rank the strategies if we consider the code cache only, while Figure 9(c) shows how it would rank the strategies if we consider both the code cache and data structures. There is great variation between the rankings of Figure 9(b) and Figure 9(c) because

(a) Normalized memory demand of the benchmarks, with the benchmarks being arranged in increasing order of baseline execution time.

(b) Path selection strategies ranked according to their memory efficiency when only the code cache size is considered.

(c) Path selection strategies ranked according to their memory efficiency when the total memory demand is considered.

Fig. 9. Normalized memory demands of different path selection strategies, with threshold-based trace selection, proactive linking as the baseline.

the strategies use different proportions of data structures. Therefore, it is misleading to consider the code cache size only to measure the memory demand. Also, since the ratio of code cache size to the data structure size is different for each configuration, there is no straightforward method to calculate the data structure size given the code cache size.

As shown in Figure 9(c), all the lazy linking schemes perform better than all the proactive linking schemes. Therefore, the increase in auxiliary code size due to lazy linking is outweighed by the decrease in data structures. The linking strategy has the most effect on memory efficiency. The influence of the linking strategy is followed by the strategy of deciding the number of basic blocks in a trace. The reduction in data structures and auxiliary code due to multiblock traces outweighs the increase in duplication. Multiblock traces have better memory efficiency than single-block traces. We found that the degree of speculation does not influence the memory efficiency much as long as the decisions are accurate. Both nonspeculative (dynamic selection)

(a) Normalized performance of medium-length benchmarks.

(b) Normalized performance of long-running benchmarks.

Fig. 10. Normalized performance of different path selection strategies, with threshold-based trace selection, proactive linking as the baseline and 512 KB as the uniform memory limit.

and highly accurate, speculative (profile-based selection) trace selection perform well because neither waste space. The best memory efficiency should be provided by combining lazy linking with multiblock traces and accurate trace selection. Profile-based trace selection and dynamic trace selection combined with lazy linking have these characteristics and offer the best memory efficiency. A 20% memory savings can be achieved with these path selection strategies.

3.2.2. Performance. Figure 10 shows the normalized execution times of medium-length and long-running benchmarks. Short-running benchmarks do not get much time to amortize translation overheads by executing in the code cache, resulting in no clear winner among the different path selection strategies in this category. Therefore, we omit the results on the short-running benchmarks.

The path selection strategies with the best memory efficiency have the best performance. Dynamic and profile-based trace selection strategies combined with lazy linking fulfill these characteristics and provide the best runtime performance improvement of 11% for the medium-length category and the long-running category. Code cache locality and context switch overhead are not as important as memory efficiency because dynamic selection has less code cache locality than profile-based selection. Also, dynamic selection, being nonspeculative, carries more context switch overhead than profile-based selection. Yet dynamic selection performs almost as well as profile-based selection.

3.2.3. Discussion. Based on experimental results, we recommend considering data structures in the total memory demand. We also recommend the use of multiblock traces that are formed by selecting contiguous basic blocks as long as the control flow remains sequential. Whether the control flow remains sequential should be determined nonspeculatively or using highly-informed speculation. The traces should be linked lazily. Therefore, dynamic selection or profile-based selection combined with lazy linking are the path selection strategies of choice for memory-constrained scenarios. With

Fig. 11. Exit stub structure. Exit stubs contain both code and data.

```

1 sub sp, sp, 80h # save the guest application context
2 stm sp, [mask = 0xff]
3 ld r0, [addr of args] # load the address of the data
4 ld pc, [addr of handler] # branch to translator

```

(a) Pin's exit stub code for ARM

```

1 mov eax, [predef memory location] # save eax
2 mov [addr of stub data], eax # load address of data in eax
3 jmp [addr of translator handler] # branch to translator

```

(b) DynamoRIO's exit stub code for x86

Fig. 12. Examples of exit stub code from different DBTs.

profile-based selection, a profiling run must occur. Our experiments show that dynamic selection can get close to profile-based selection without the profiling run. Therefore, our final recommendation is dynamic selection with lazy linking. Dynamic selection with lazy linking improves memory efficiency by 20% and performance by 11%.

4. CODE CACHE EXIT STUB OPTIMIZATION

The next research step was to determine, given some path selection strategy, whether it is possible to optimize the components of memory demand further. However, the requirement in this step is that such optimizations should not impact the path selection strategy and should not produce interactions among the different components of memory demand. Exit stubs are a component of memory demand that fit these requirements. First, exit stubs exist only to handle trace exits and no other component of memory demand depends on them. Secondly, we found that exit stubs occupy a major percentage of the code cache and therefore have great potential for optimization.

It is challenging to reduce exit stub memory footprint, as exit stubs are needed to maintain control over the program execution. Exit stubs become unreachable when their corresponding trace exits are linked, but deleting them is risky because there is no guarantee that exit stubs will remain unreachable throughout execution. They will be needed if the corresponding trace exits are ever unlinked.

4.1. Exit Stubs

The structure of an exit stub is shown in Figure 11. An exit stub consists of both code and data. The code is responsible for saving the guest application context, loading the address of exit stub data so that the translator can locate the data, and branching to the translator. The data contains arguments to the translator such as the target address of the trace exit. It may also contain arguments such as a hash of the target address to quickly search the code cache directory for the target trace. The arguments also indicate the branch type, for example, whether it is direct or indirect. Finally, the exit stub stores the translator entry address because the branch to the translator has a large offset and is therefore, indirect.

Figure 12 shows the exit stub code for two different DBTs: Pin and DynamoRIO. In line 1 of Figure 12(a), the guest application stack pointer is shifted to make space for

Fig. 13. Two arrangements of stubs for a given trace in the code cache.

Table III.
Percentage of code cache consisting of
exit stubs, measured over the Spec2000
benchmark suite.

DBT	Exit Stub Percentage
Pin	66.67%
DynamoRIO	62.78%

saving the context. In line 2, the application context is saved using the `store multiple register (stm)` command. The mask `0xff` specifies that all 16 registers have to be saved. In line 3, register `r0` is loaded with the address of exit stub data. In line 4, the program counter is loaded with the translator handler's address, which is essentially a branch instruction.

Figure 12(b) shows DynamoRIO's exit stub code. The `eax` register is saved in a predefined memory location in line 1. In line 2, `eax` is loaded with the address of exit stub data and line 3 transfers control to the translator handler.

Figure 13 shows two possible layouts of stubs for a given trace. In Figure 13, stubs 1 and 2 are exit stubs corresponding to a single trace. The code and data for each stub may appear together as shown in Figure 13(a), or they may be arranged so that all the code in the chain of stubs appear separately from all the data in the chain of stubs, as shown in Figure 13(b). The layout in Figure 13(b) is better for cache efficiency. As we show in Section 4.2, the first layout conserves more space.

Table III shows the space occupied by exit stubs in Pin and DynamoRIO [Bruening 2004]. As the numbers demonstrate, exit stubs form an important candidate for optimization.

4.2. Exit Stub Size Reduction

Both code and data size are optimized by exit stub size reduction. Code space is saved by identifying the common code in all stubs. We use a common routine for saving the context and remove corresponding instructions from the stubs. However, the program counter has to be saved before entering the common routine. Such a measure is needed because the program counter is the single register that will get modified upon branching to the translator. Figure 14 shows the code in the stub after this optimization. Here, we take advantage of the fact that the ARM `st` instruction allows the store target to be calculated and the store to be executed together. Using the `stm` instruction required the target to be calculated in a separate instruction. Therefore, the size of the exit stub code is reduced by one instruction. However, saving the rest of the guest application context now requires two more instructions to adjust the stack pointer once again and to save the remaining registers using `stm`. Thus, the dynamic instruction count effectively increases. Factoring of common code is a simple technique that has been implemented in some form in systems (e.g., DynamoRIO).

To further improve the code size, we adhere to the layout in Figure 13(a) to avoid storing or loading the address of stub data, since the stub data appears at a fixed offset from the start of the stub. The stub's start address is known from the program counter saved in the context. Here we see the advantage of saving the program counter. Some

```

1 st [sp - 80h]!, pc
2 ld r0, [addr of args]
3 ld pc, [addr of handler]

```

Fig. 14. Exit stub after factoring out code to save the guest application context.

```

1 st [sp - 80h]!, pc
2 ld pc, [addr of handler]

```

Fig. 15. Exit stub after altering the layout of stub code and data.

Fig. 16. Memory usage (reported in kilobytes) of Pin baseline (leftmost bar) and after incorporating our optimizations (rightmost bars).

instruction and data cache locality is sacrificed in adopting this exit stub layout. The resulting exit stub code is shown in Figure 15.

We avoid storing the type of branch in the stub data area and use specialized translator handlers for each branch type. This optimization increases complexity of management because the number of possible translator entry points increase. Additionally, we store the possible translator entry addresses at the code cache level rather than exit stub level. This optimization reduces memory demand but increases complexity because the trace is no longer a self-contained entity. If the trace ever needs to be moved to a different location, each exit stub will need to be modified to load the translator entry address from the correct location.

We also avoid storing any derivable data within the stub. For example, we do not store the hash of the target address but compute it every time it is needed. Thus, we save space by avoiding the storing of all the arguments to the translator. This gives rise to a trade-off with performance. After all the data size optimizations, the stub stores only the target address.

This scheme is applied to all exit stubs for direct branches and calls which constitute a majority of the exit stubs. Flushing and invalidations are unaffected as we have modified only the stub structure. Our mechanism is independent of the flushing strategy and the number of threads being executed by the program.

4.3. Memory Efficiency

The first set of experiments focus on the memory improvement of our approach. Figure 16 shows the memory used in the code cache as kilobytes allocated. The category *original* is the number of KBs allocated in the baseline version. The average savings in memory in this scheme is 37.4%. The savings is due to the fact that memory is saved from all stubs for direct branches and calls, which are the dominant form of control instructions (they form 90% of the control instructions in the code cache for the SPEC benchmarks).

Fig. 17. Performance of `exit` stub reduction with cache limit of 1280 KB.

4.4. Performance under Cache Pressure

Our next set of experiments measures the performance in the case of a limited code cache. We set the cache limit to 1280 KB. (See Figure 17). Our approach performed 5–6% better on average. The performance improvement is due to a smaller code cache and a reduction in code cache flushes.

5. SELECTIVE CACHE FLUSH

In memory-limited situations, the memory demand problem of DBTs is replaced by a performance degradation problem due to flushing. Selectively flushing traces that will not be used in the future, or at least in the near future, can reduce flush overhead. However, selective flushing is challenging. It is difficult to dynamically select which traces 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 multithreaded programming. Code caches for multithreaded guest applications are shared by all threads. Thread-shared caches must ensure that no thread is executing in the traces selected for eviction. DBTs fulfill this condition by checking that each thread that was executing in the selected traces exits to the runtime once and is never allowed to return to the selected traces. 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 traces in the first place. Due to these challenges, full flush (no selection) has become the standard.

We wish to use selective flushing to reduce the retranslation overhead for overall performance improvement. We also want our flushing technique to be applicable to both single-threaded and multithreaded applications. We use profiling to select traces to evict. We also address all the challenges of profiling, code cache management, and thread management associated with partial flushing.

5.1. Selective Flushing

Figure 18 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, that is, 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, and linking to this trace is again allowed. However, if the trace does not become active, it will eventually get overwritten

Fig. 18. The states traversed by a cached trace for different flush techniques.

Fig. 19. The successive code cache states when applying our partial flush technique.

by new traces. The mechanisms of flush triggering, profiling, promoting and allocating space are described in Section 5.1.1, Section 5.1.2, Section 5.1.3, and Section 5.1.4 respectively. Figure 19 shows the successive states of the code cache when our partial flush technique is applied, with the key shown in Figure 19(a).

5.1.1. Triggering Flushes. A flush is triggered at the high water mark, and all traces are unlinked to expedite the exit of threads. In a traditional full flush, all directory entries corresponding to the unlinked traces are discarded, so that these traces cannot be located or reentered. However, in our technique, reentry is still possible. Therefore, the code cache directory entries are not discarded, and additional tag data structures record whether the trace is old or current. Initially, all traces are in the current state. Upon unlinking, the traces are tagged as old. Old traces cannot be reentered without first promoting them to current. Figure 19(b) shows the state of the code cache at a flush trigger point. The code cache contains old traces and some free space.

It is worth noting that we select all traces for eviction and use profiling to discard from the eviction set, rather than the other way around. The advantage of selecting all traces for eviction is that we know that all threads need to be monitored and we do not have to identify which threads were executing in the eviction set.

5.1.2. Profiling Traces. LRU is our profiling strategy. When all threads have exited the code cache once after a flush trigger, all old traces are assumed to belong to the LRU set and are monitored continuously from that point onwards until they are promoted or evicted. If there is a request for execution of some trace in the LRU set, the trace is made active and removed from the LRU set.

Profiling is enabled by unlinking the traces, which forces the translator to be invoked before every LRU trace execution. If the translator determines that the requested trace is already in the LRU set, it activates the trace. This strategy makes profiling simple as no instrumentation code has to be inserted. It also ensures that a trace is automatically profiled until it leaves the LRU set. The tradeoff is that there may be performance degradation due to execution in the unlinked mode. For thread-shared caches, the performance overhead is masked because traces already undergo unlinking near flush points. We simply exploit this unlinking activity to facilitate profiling. However, for single-threaded caches using full flush, unlinking indeed presents a performance overhead, but that overhead is outweighed by the reduction in translation time. Furthermore, since we promote traces out of the LRU set on the first request, there are few executions in unlinked mode.

5.1.3. Promoting Traces. Trace promotion to the active state simply implies changing its tag from old to current, and reenabling 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 code cache fragmentation. Figure 19(c) shows the resulting situation. Traces are being inserted into the free area of the code cache. At the same time, scattered old traces are being promoted to the current state.

5.1.4. Allocating Space to Traces. Traces are inserted into the contiguous, free area of the code cache in Figure 19(c). However, upon reaching the state shown in Figure 19(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 suitable 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 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 19(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 traces crosses the high water mark, the code cache manager triggers the next flush as shown in Figure 19(f). All existing traces are tagged old, as shown in Figure 19(g).

Figure 19(g) shows that the code cache is full of old traces. 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 kind 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.

Here we see another way in which we differ from traditional flushing. Usually flushing of groups of traces is performed at well-defined flush points. Such flushing frees up much more space than required at a time. Yet it is the preferred method because it reduces book-keeping complexity. Traces belonging to these groups may get retranslated after varying amounts of time. Some of these retranslations can be avoided if minimum required traces are flushed at a time. Therefore, we allow code regions to live for as long as possible. We do so by evicting only if there is a new trace demanding space. Only as much space as is required by the new code region is reclaimed. Profiling for each trace continues until the code region is evicted or it is determined that the trace should not be evicted. We enable such continuous LRU profiling with reasonable overhead.

Fig. 20. 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.

Fig. 21. Fraction of speedup resulting from each DBT task. Reduction in translation time is the greatest contributor of speedup.

5.2. Single-Threaded Performance Evaluation

For evaluating the performance of single-threaded benchmarks, we used a code cache limit of 256KB and trigger a flush when the code cache is 100% full. 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 20 shows the normalized execution times for the single-threaded benchmarks when our partial flushing technique is applied. 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 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 21 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.

From Figure 21 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 compared to full flushing. 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 flushes on an average. Surviving across each flush 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

Fig. 22. Cache pressure for multithreaded benchmarks. Cache pressure is the ratio of the unlimited code cache size to the memory limit (512 KB in this case).

Fig. 23. Execution time for partial flush applied to thread-shared caches, normalized with respect to full flush. Average speedup is 15%.

switches overall. It is worth noting that not much extra time is spent in flushing, that is, the book-keeping overhead of our proposed technique is small. Application execution and indirect branch handling time also remain fairly stable.

5.3. Multithreaded Performance 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 32KB instruction cache, 24KB data cache with write-back and a 512KB L2 cache. The memory size is 1GB. It supports Linux kernel 2.6.24. For the ATOM-based netbook, we used Pin [Hazelwood et al. 2009; Luk et al. 2005] targeting the $\times 86$ architecture. 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 [Bienia et al. 2008] suite with native inputs. PARSEC consists of multithreaded benchmarks and we executed them on the netbook with two threads. This is the experimental setup for all the multithreaded experiments in this research, unless otherwise stated.

We first explored which of the multithreaded benchmarks would need flushing activity for the given memory limit. Figure 22 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 swaptions, we are interested in ensuring that overhead is reasonable rather than obtaining improvements.

Figure 23 shows the normalized benchmark execution times. Average-small is the average normalized execution time for the benchmarks that do not undergo flush

Fig. 24. Trace translation count for partial flush applied to thread-shared caches, normalized with respect to full flush.

activity. Average-large is the average normalized execution time for the benchmarks that do undergo flush activity. On average, the performance improvement for the large benchmarks is 15% while that for the small benchmarks remains the same. Therefore, we have ensured that we get performance improvements for the large benchmarks and do not cause overhead for the small benchmarks. However, we fail to improve performance for canneal.

To understand the performance results, we analyzed how many trace translations we have reduced using our technique. Figure 24 shows the normalized translation count for partial flush. bodytrack, freqmine and facesim had the most cache pressure and show the greatest reduction in translations. They also show the best performance improvements, among the large benchmarks. canneal and fluidanimate have relatively less cache pressure and also show less translation reduction. Not surprisingly, their performance improvements are the lowest among the large benchmarks. canneal actually shows slowdown. The slowdown is due to the fact that canneal is also the shortest-running benchmark. It is an order of magnitude shorter than bodytrack, the next longer benchmark in the large category. Therefore, the overheads due to our technique is more pronounced in canneal.

6. A DYNAMIC BINARY TRANSLATOR FOR EMBEDDED SYSTEMS

Sections 3 through 5 strive to design a DBT for embedded systems using different approaches. In this section, the designs are unified to produce a whole system. We select which approaches to combine, and determine how beneficial it is to combine them. If some approaches cannot be combined seamlessly, we design modified strategies. The unification demonstrates the total improvements we can achieve.

6.1. Balanced Path Selection

Path selection combines trace selection and link formation. We found dynamic trace selection to be the most beneficial strategy. However, there are several issues in integrating dynamic trace selection with partial flushing and thread-shared code caches. We discuss these issues and their solutions in Section 6.1.1 and Section 6.1.2. For the linking strategy, we found lazy linking to be most beneficial. Lazy linking integrates seamlessly with partial flushing and thread-shared code caches because it only requires some extra data (the branch location) to be stored in the exit stub. Also, the code has to be modified to ensure that links are formed only on demand. These modifications do not conflict with any aspect in the rest of the system and therefore lazy linking does not present any issues.

6.1.1. Interaction between Dynamic Trace Selection and Selective Cache Flush. Dynamic trace selection needs one context switch for every basic block it adds to a trace. Each such

Fig. 25. Separated traces and exit stubs are suitable for dynamic trace selection, while contiguous traces and exit stubs are suitable for selective flushing.

Fig. 26. Dynamic trace selection and selective cache flushing are integrated. Consecutive trace blocks are separated by exit stubs. Exit stubs corresponding to fall-throughs can be overlaid.

basic block has to be allocated a space contiguous to the trace under translation. If contiguous space cannot be found, the trace under translation must be terminated. Consequently, the basic block must form a separate trace head and a code cache entry needs to be allocated for it. In a selective cache flushing system, finding contiguous space is more complicated because the code cache is fragmented. The code cache manager scavenges holes for space and the probability of finding contiguous space is lower compared to unfragmented code caches. The code cache manager first checks if there is an existing hole that is contiguous with the trace. If such a hole exists, the code cache manager tries to allocate space in the existing hole. If there is enough space, the trace can be successfully extended. If there is not enough space in the hole or such a hole does not exist, the trace cannot be extended anymore.

A second problem that occurs with selective cache flush is the trace arrangement strategy. Figure 25 shows the different arrangements of traces and their exit stubs within the code cache: separated and contiguous. For selective cache flushing, we adopt the trace arrangement strategy shown in Figure 25(b) where traces and exit stubs form a single unit, to simplify management of the fragmented code cache. However, for dynamic trace selection, the arrangement in Figure 25(a) is more convenient. Dynamically adding basic blocks and their corresponding exit stubs to the trace maintains the arrangement in Figure 25(a). For the arrangement in Figure 25(b), each basic block in a trace will be followed by its exit stubs, that is, consecutive basic blocks belonging to the same trace will be separated by exit stubs. Such an arrangement reduces code cache locality.

Also, the space savings reaped by dynamic trace selection from the trace arrangement in Figure 25(b) are less than those from the arrangement in Figure 25(a). In Figure 25(a), both the trace exit and the exit stub corresponding to the fall-through can be overwritten because both of them may border on free space. However, for the arrangement in Figure 25(b), the trace exit corresponding to the fall-through will never border on free space and as a result can never be reclaimed. Figure 26 shows the trace

format resulting from integrating dynamic trace selection and selective cache flushing. The traces and exit stubs form a single, contiguous unit but the consecutive basic blocks are separated by exit stubs. Only the space occupied by the exit stub corresponding to the fall-through trace exit can be overlaid when a trace is extended.

6.1.2. Interaction between Dynamic Trace Selection and Thread-Shared Code Caches. Dynamic trace selection also encounters issues when integrated with a thread-shared code cache. This interaction also arises because the traces are extended one basic block at a time. Two or more threads may strive to construct different traces at the same time. The code cache manager will interleave the basic blocks from the different traces in this situation. If the basic blocks for a trace cannot be allocated contiguously, they must form different traces. Thus, the potential benefit from dynamic trace selection is reduced. Therefore, we always check whether we have been able to allocate contiguously or not.

6.2. Exit Stub Optimization

We studied the reduction strategy for the baseline system which uses proactive linking. However, our path selection strategy recommends lazy linking. Lazy linking requires larger exit stubs compared to proactive linking. Therefore, we expect the benefits of exit stub optimizations to be reduced when combined with the path selection strategy. Apart from this interaction, reduction in exit stub size integrates seamlessly because it merely recommends a different format for exit stubs. The different format has no interaction with the number of threads executing in the code cache and the flushing technique.

6.3. Cache Flush

The selective cache flush requires some data structures. However, we only considered the code cache size when evaluating selective cache flushing. In the whole system, we must take into account the size of data structures to determine when we reach the memory limit. We discuss the impact of data structures and how we handle them in Section 6.3.1, Section 6.3.2, and Section 6.3.3.

6.3.1. Trace Tagging Data Structures. Traces must be tagged as 1) current, 2) old and active, or 3) old and inactive in an selective cache flushing system. The tag requires extra space, but the extra space can be reduced by noting that it has only three possible values and thus needs only two bits. At the same time, all traces are aligned to a 4-byte word boundary on the ARM platform, so the code cache address field will always end in two zeros. We use these two bits to encode the tag of the trace. Therefore, there is no extra space requirements for trace tags.

6.3.2. Code Cache Map. The code cache manager maintains a list of pointers to code cache directory entries, in order of their code cache addresses. We call this data structure, the *code cache map*. The code cache map enables the code cache manager to scavenge for holes in the code cache. The size of the code cache map is included in the total memory demand, along with the code cache size, the code cache directory size and the link data structure size.

6.3.3. Including Data Structure Size in Memory Demand. The total memory demand should be considered in an selective cache flush. At each high water mark point, the code cache and the data structures make up the total memory demand. The ratio between the data structure size and the code cache size varies slightly at high water mark points, for a given path selection strategy. This variation will cause the actual code cache size to be slightly different at each high mark point. However, for a selective cache flush, traces are scattered throughout the code cache and it is difficult to change the limit on the code cache size.

Fig. 27. Normalized total memory demand of the full system, which combines balanced path selection with reduction in exit stub optimization. Although cache flushing is not used, the data structures required by selective cache flushing are accounted for.

We solve this problem by permanently limiting the code cache to the size at the first high water mark point. This strategy does not disregard the memory limit by any significant amount, as for a given path selection strategy, the variation in code cache and data structure sizes between high water mark points is small.

6.4. Memory Evaluation

We roughly estimated the expected memory savings from our combined techniques first. For the estimation, let us assume the total memory demand of the baseline system to be 1. According to Figure 1, the translated code size, auxiliary code size and the data structure size of the baseline system are 23%, 41% and 36% respectively. Therefore, the code cache size of the baseline system is 64% (sum of translated code and auxiliary code sizes). We applied our exit stub optimizations to the code cache only. For the reduction in exit stub size optimization), we obtained a code cache size reduction of 37%. Therefore, we expect the code cache size to be 40% after this optimization. The total memory demand after this optimization is 76% (sum of code cache and data structures). Dynamic trace selection combined with lazy linking further improved the memory efficiency of the whole system by 20%. Therefore, the total memory demand after combining balanced path selection is 61%, that is, we expect to improve memory efficiency by 39%.

Figure 27 shows the actual normalized memory demand of the whole system. There are memory savings for all the benchmarks, ranging from 20% to 44%. The average memory savings for the short, medium, and long benchmarks are 30%, 37% and 36% respectively. These numbers show that memory efficiency benefits from our different techniques have been combined because the memory savings of the combined system is better than the memory savings of any of our techniques in isolation. Also, we have produced results quite close to the expected memory efficiency. The actual memory savings is less than the estimated memory savings because 1) memory efficiency was lost in integrating dynamic trace selection, 2) lazy linking requires larger exit stubs than proactive linking, reducing the benefit of exit stub optimizations, and, 3) the code cache map required by selective cache flushing has been incorporated into the memory demand.

Fig. 28. Normalized execution time of the full system, which combines balanced path selection with reduction in exit stub size and selective cache flush.

6.5. Performance Evaluation

Figure 28 shows the normalized execution time of the benchmarks. There are no clear gains for the short benchmarks. However, this is not surprising because short benchmarks do not execute long enough to amortize the overheads of the proposed techniques. For the medium benchmarks, performance gains begin to manifest more clearly. There is an average improvement of 10% in the execution time, with a maximum improvement of 37% in `perlbnk`. For the long category, performance of every benchmark is improved. The average reduction in execution time is 27%, with a maximum execution time reduction of 89% in `parser`. One especially encouraging fact is that, for `gcc`, the longest-running benchmark, a 5% performance improvement occurs in the combined system, while path selection combined with full flushing was unable to improve the performance of `gcc` over the baseline.

6.6. Interaction between Dynamic Trace Selection and Thread-Shared Code Caches

We quantified the interference between thread-shared code caches and dynamic trace selection by measuring the number of traces formed. We first executed the multi-threaded benchmarks in a single-threaded fashion and measured the number of traces formed by a dynamic trace selection strategy. Then we executed the benchmarks with two threads and again measured the number of traces formed by a dynamic trace selection strategy. The increase in traces formed quantifies the interaction between thread-shared code caches and dynamic trace selection.

Figure 29 shows the ratio of the number of traces formed when using two threads to the number of traces formed when using a single thread. The increase in the number of traces formed ranges from 0% to 4%, averaging 3%. Therefore, the increase in the number of traces formed is small and results similar to single-threaded code caches can be expected for thread-shared code caches.

7. RELATED WORK

We first provide related work on DBT applications in Section 7.1. Then we describe previous research on embedded DBTs in Section 7.2. We discuss previous research related to path selection, exit stub optimization and code cache management in Section 7.3, Section 7.4 and Section 7.5 respectively.

Fig. 29. Traces formed by dynamic trace selection in a thread-shared code cache, normalized wrt the traces formed by dynamic trace selection in a single-threaded code cache.

7.1. Dynamic Binary Translators

Dynamic binary translators provide software adaptation in various forms [Bala et al. 2000; Bruening et al. 2003; Desoli et al. 2002; Scott et al. 2003; Sridhar et al. 2006]. For example, DBTs can enforce security policies [Bruening et al. 2003; Hu et al. 2006; Kiriansky et al. 2002; Scott and Davidson 2002]. Some DBTs such as Pin [Luk et al. 2005] and Valgrind [Nethercote and Seward 2007] and HDTrans [Sridhar et al. 2006] support dynamic instrumentation. Dynamo [Bala et al. 2000], Mojo [ke Chen et al. 2000] and ADORE [Chen et al. 2004] are DBTs that support dynamic optimization. Several DBTs provide translation between platforms [Baraz et al. 2003; Dehnert et al. 2003; Ebcioglu and Altman 1997; Sathaye et al. 1999; Ung and Cifuentes 2000; Zheng and Thompson 2000].

7.2. Dynamic Binary Translators for Embedded Systems

Although development of DBTs for embedded systems has been limited compared to general-purpose platforms, there are quite a few embedded DBTs in use. Strata has been targeted to ARM and PISA [Baiocchi et al. 2007; Baiocchi and Childers 2009; Baiocchi et al. 2008; Moore et al. 2009]. Pin [Hazelwood and Klauser 2006] and Valgrind provide dynamic binary instrumenters for ARM. DELI [Desoli et al. 2002] exposes client API for fine-grained control over the guest application execution to provide services such as ISA emulation, software patching and sandboxing on the Lx/ST210 embedded VLIW processor.

7.3. Path Selection

Most previous research on path selection has been from the performance perspective. Previous work has researched both trace selection [Duesterwald and Bala 2000; Hiniker et al. 2005; Hiser et al. 2006] and linking strategies [Bala et al. 2000; Bruening et al. 2003; Desoli et al. 2002; ke Chen et al. 2000; Luk et al. 2005]. Such approaches do not consider the memory demands of path selection or its performance in memory-constrained environments.

For most DBTs, the trace is the unit of choice compared to functions and methods [Bala et al. 2000; Bruening et al. 2003; Desoli et al. 2002; ke Chen et al. 2000; Luk et al. 2005; Nethercote and Seward 2007; Scott et al. 2003]. Next-Executing-Tail (NET) [Duesterwald and Bala 2000] is the most popular trace selection algorithm. The Pin strategy is same as the threshold-based strategy. Recently, Pin has been used to simulate the last-executed iteration (LEI) strategy to better select traces for optimization [Hiniker et al. 2005]. HDTrans [Sridhar et al. 2006] uses single-block traces, and

elides unconditional branches, but continues translating the fall through instructions after direct calls.

For the linking strategy, most DBTs use proactive linking [Bala et al. 2000; Bruening et al. 2003; Luk et al. 2005]. Some DBTs use a deferred linking policy [Desoli et al. 2002; ke Chen et al. 2000] in which they proactively link the exits of the trace under construction, but link the entries to that trace lazily. Such a policy will still create more unnecessary links than lazy linking. However, deferred linking will reduce some context switches compared to lazy linking. Strata for embedded systems [Baiocchi et al. 2007; Baiocchi et al. 2008] recommends a full proactive linking policy. The reason for recommending a full proactive linking policy may be that when only the code cache size is considered in a memory-constrained environment, proactive linking outperforms both lazy linking and deferred linking.

7.4. Exit Stub Optimization

Hiser et al. study the allocation of trampolines in a separate region (a trampoline pool) in comparison to interleaving them with fragments [Hiser et al. 2006]. They find the technique lowers I-cache pressure. While a trampoline pool is a performance optimization, Strata for ARM also performs memory optimizations on exit stubs [Baiocchi et al. 2008].

7.5. Code Cache Management

Memory management policies for DBTs have been researched before, although data structure sizes have been traditionally ignored. Thread-shared software code caches [Bruening et al. 2006; Hazelwood et al. 2009] 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 policies have been designed for purposes other than reducing memory demand. For example, Dynamo [Bala et al. 2000] 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 [Bruening and Amarasinghe 2005; Bruening et al. 2006] 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. They also use a basic block cache and a trace cache, whereas we use only a trace cache. However, most of our approaches can be applied to basic block caches as well. Combining our approaches with cache division (such as into basic block and trace caches) can improve our results. The second reason our research differs from previous memory management work is that most of the prior memory management policies target full or partial eviction for single-threaded code caches. Strata [Baiocchi et al. 2007; Baiocchi and Childers 2009; Baiocchi et al. 2008] 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 [Guha et al. 2008; Hazelwood and Smith 2006], but only for single-threaded code caches. Pin [Hazelwood and Klauser 2006; Hazelwood et al. 2009] accounts for thread-shared code caches, but only supports a full flush.

8. CONCLUSIONS

This article explored approaches to reducing the memory demand of DBTs targeting embedded systems. These approaches reduce the pressure on the memory subsystem, a resource which is constrained in embedded systems. For DBTs that handle the problem of memory demand by placing a limit on the memory usage, the memory demand

problem manifests itself as a performance degradation problem. Our memory optimizations help reduce this performance degradation by reducing the number of flushes. Additionally, we designed strategies to selectively preserve traces and their corresponding exit stubs and data structures across flushes to reduce the performance overhead further.

We found that the absolute and relative memory demand of translated code, auxiliary code, and data structures depends on the path selection (trace selection and linking) strategy used by the DBT. The performance of the DBT also depends on the path selection strategy. Therefore, we evaluated a comprehensive set of path selection strategies to propose a strategy for both holistic memory efficiency and performance. We also found that the data structure size has been ignored in previous research, and incorporating data structure size into the memory demand leads us to better designs. We proposed a dynamic trace selection strategy, a trace termination strategy and lazy linking. We then recognized that for any given path selection strategy, the exit stub size can be further reduced without impacting any of the other memory demand components. We designed several optimizations to reduce the exit stub code size and exit stub data. We also designed a selective cache flushing strategy. We used a pseudo-LRU heuristic to classify traces. We developed this strategy to be uniformly applicable across single-threaded and multithreaded guest applications. We also combined all these approaches into a single system. During integration, we found several conflicts among the strategies and devised techniques to handle them.

We evaluated our techniques on two different embedded platforms. We found that path selection can improve memory efficiency of the whole system by 20% on the average and performance by 11%. We found that exit stub optimizations can reduce the code cache size by 37% and the performance by 5–6%. Selective cache flushing improved the performance of single-threaded benchmarks by 17% and that of multithreaded benchmarks by 15%. The combined system exhibited memory savings from 30%, 37%, and 36% for short, medium and long benchmarks respectively and performance improvements of 10% and 27% for medium and long benchmarks respectively.

REFERENCES

- BAIOCCHI, J., CHILDERS, B. R., DAVIDSON, J. W., HISER, J. D., AND MISURDA, J. 2007. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 75–84.
- BAIOCCHI, J. A. AND CHILDERS, B. R. 2009. Heterogeneous code cache: using scratchpad and main memory in dynamic binary translators. In *Proceedings of the 46th Annual Design Automation Conference*. 744–749.
- BAIOCCHI, J. A., CHILDERS, B. R., DAVIDSON, J. W., AND HISER, J. D. 2008. Reducing pressure in bounded DBT code caches. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 109–118.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation*. 1–12.
- BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. 2003. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium R-based systems. In *Proceedings of the 36th International Symposium on Microarchitecture*. 191.
- BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.
- BRUENING, D. AND AMARASINGHE, S. 2005. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*. 74–85.
- BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. 265–275.

- BRUENING, D., KIRIANSKY, V., GARNETT, T., AND BANERJI, S. 2006. Thread-shared software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*. 28–38.
- BRUENING, D. L. 2004. Efficient, transparent and comprehensive runtime code manipulation. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.
- CHEN, H., LU, J., HSU, W., AND YEW, P. 2004. Continuous adaptive object-code re-optimization framework. In *Proceedings of the Asia-Pacific Computer Systems Architecture Conference*. 241–255.
- DEHNERT, J., GRANT, B., BANNING, J., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSO, J. 2003. The trans-meta code morphing software. In *Proceedings of the International Symposium on Code Generation and Optimization*. 15–24.
- DESOLI, G., MATEEV, N., DUESTERWALD, E., FARABOSCHI, P., AND FISHER, J. A. 2002. Deli: A new run-time control point. In *Proceedings of the 35th International Symposium on Microarchitecture*. 257–268.
- DUESTERWALD, E. AND BALA, V. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th Architectural Support for Programming Languages and Operating Systems*. 202–211.
- EBCIOGLU, K. AND ALTMAN, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture*. 26–37.
- GUHA, A., HAZELWOOD, K., AND SOFFA, M. L. 2008. Code lifetime based memory reduction for virtual execution environments. In *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*.
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench : A free, commercially representative embedded benchmark suite. In *Proceedings of the Workshop on Workload Characterization*. 3–14.
- HAZELWOOD, K. AND KLAUSER, A. 2006. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 261–270.
- HAZELWOOD, K., LUECK, G., AND COHN, R. 2009. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the International Symposium on Memory Management*. 20–29.
- HAZELWOOD, K. AND SMITH, M. D. 2006. Managing bounded code caches in dynamic binary optimization systems. *Trans. Code Gener. Optim.* 3, 3, 263–294.
- HENNING, J. L. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7, 8–25.
- HINIKER, D. J., HAZELWOOD, K., AND SMITH, M. D. 2005. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th International Symposium on Microarchitecture*. 141–154.
- HISER, J. D., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., AND CHILDERS, B. R. 2006. Evaluating fragment construction policies for SDT systems. In *Proceedings of the International Conference on Virtual Execution Environments*. 122–132.
- HU, W., HISER, J., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., EVANS, D., KNIGHT, J. C., NGUYEN-TUONG, A., AND ROWAN-HILL, J. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the International Conference on Virtual Execution Environments*. 2–12.
- JANAPAREDDI, V., CONNORS, D., COHN, R., AND SMITH, M. D. 2007. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the International Symposium on Code Generation and Optimization*. 74–88.
- JOHN, L. K. 2004. More on finding a single number to indicate overall performance of a benchmark suite. *SIGARCH Comput. Archit. News* 32, 1, 3–8.
- KE CHEN, W., LERNER, S., CHAIKEN, R., AND GILLES, D. 2000. Mojo: A dynamic optimization system. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*. 81–90.
- KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*. 191–206.
- LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNY, G., WALLACE, S., JANAPAREDDI, V., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*. 190–200.
- MILLER, J. E. AND AGARWAL, A. 2006. Software-based instruction caching for embedded processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 293–302.
- MOORE, R. W., BAIocchi, J. A., CHILDERS, B. R., DAVIDSON, J. W., AND HISER, J. D. 2009. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 147–156.
- NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*. 89–100.

- SATHAYE, S., LEDAK, P., LEBLANC, J., KOSONOCKY, S., GSCHWIND, M., FRITTS, J., BRIGHT, A., ALTMAN, E., AND AGRICOLA, C. 1999. Boa: Targeting multi-gigahertz with binary translation. In *Proceedings of the Workshop on Binary Translation*. 2–11.
- SCOTT, K. AND DAVIDSON, J. 2002. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Computer Security Applications Conference*. 209–218.
- SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J., AND SOFFA, M. L. 2003. Reconfigurable and retargetable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 36–47.
- SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. 2006. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the International Conference on Virtual Execution Environments*. 175–185.
- UNG, D. AND CIFUENTES, C. 2000. Machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Dynamic and adaptive compilation and optimization*. 41–51.
- WU, Q., MARTONOSI, M., CLARK, D. W., JANAPAREDDI, V., CONNORS, D., WU, Y., LEE, J., AND BROOKS, D. 2005. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th International Symposium on Microarchitecture*. 271–282.
- ZHENG, C. AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer* 33, 47–52.

Received October 2010; revised December 2011; accepted June 2012