Code Lifetime-Based Memory Reduction for Virtual Execution Environments

Apala Guha Kim Hazelwood Mary Lou Soffa

Department of Computer Science
University of Virginia

Abstract

The need for adaptability in a rapidly expanding embedded systems market makes it important to design virtual execution environments (VEEs) specifically targeting embedded platforms. We believe the first step in this direction should be to replace the performance focus of traditional VEE design with a combined memory and performance focus, given the memory constraints on embedded systems. In this work, we present techniques that reduce the large code cache sizes of VEEs by continually eliminating dead cached code as the guest application executes. We use both a time-based heuristic and an execution count-based heuristic to predict code lifetime. When we determine that the lifetime of code has ended, we remove it from the code cache. We found that at least 20% code cache reduction can be achieved on average, without a significant performance degradation.

1. Introduction

Virtual execution environments (VEEs) host and control the execution of guest applications by inspecting and modifying instructions at runtime. VEEs are well-suited to provide adaptability for embedded, mobile and battery-powered devices. For example, it would be beneficial if embedded applications could adapt to the remaining battery power as they execute, increasing the battery life of the system or, at least, making the power degradation more predictable. Applications hosted by VEEs can be adapted dynamically to achieve improved power predictability. VEEs can also be used to adapt legacy binaries to leverage microarchitectural improvements occurring in embedded processor families. Also, VEEs can offer improved security to embedded systems such as PDAs (personal digital assistants) which often download third-party software.

However, VEEs have been primarily designed for general-purpose environments [5, 7, 9, 17, 23], resulting in VEE benefits being primarily exploited by general-purpose machines [7, 9, 15, 16, 24]. Traditional VEE design strives to improve performance regardless of the memory overheads since memory is cheap in general-purpose environments. While VEEs for embedded platforms are available [8, 17], they also focus on performance and do not explicitly explore the memory impacts of VEE design choices.

A VEE introduces an extra software layer between the application and the hardware and consumes machine cycles to fulfill its functionality, often slowing down the guest application. Researchers have found that pure emulation-based VEEs for general-purpose environments have 300x performance overhead compared to native execution [3]. To overcome performance overhead, translation-based VEEs store translated, native code in a software code cache for future reuse. For example, Strata [14, 15, 23], which uses a code cache, has a performance ranging between 1.03x and 1.11x compared to native execution for different hardware platforms. The code cache has been shown to have a 500% memory overhead [12]. Our goal is to avoid such large memory overheads and design VEEs taking both performance and memory overheads into account.

In this paper, we reduce the code cache size by recognizing that all translated code is not executed throughout the guest application execution. After a translated code region has been executed for the last time, it is dead. Dead code regions occupy space in the code cache unless they are evicted. Our experiments show that on the average, 60% of code has a short lifetime. The goal of this work is to develop efficient methods for dynamically identifying dead code regions and evicting them. Eviction of dead code regions reclaims occupied space from the code cache, reducing its memory requirements. Also, performance may be enhanced by such evictions because the spaces created by eviction will be filled again with code regions which are currently live. Packing long-lived code regions into a smaller area offers improved instruction locality. Our experiments also show that a high percentage (about 90%) of long-lived code regions have a high execution count while only a small percentage of short-lived code regions (about 20%) have a high execution count. Therefore, preserving only long-lived code regions is beneficial because such code regions occupy a smaller code cache area and yet, cover a greater part of the entire execution.

Code cache eviction based on lifetime has been explored before for general-purpose environments [12]. Code cache reduction techniques for general-purpose environments was motivated by the fact that many modern general-purpose applications (e.g., Microsoft Word) execute a large amount of code and hence have a high rate of code region formation which can lead to uncontrolled memory expansion when hosted by a VEE. However, embedded systems will be affected by the memory expansion of VEEs even for much less complicated benchmarks. For example, SPEC benchmarks, which were found to have a considerably lower rate of code region formation, can cause problems in embedded devices. Among the SPEC benchmarks, `gcc` has the largest code cache size of about 4 MB. Researchers have found that VEE data structures occupy about as much memory as the code cache [21]. Also, the VEE binary occupies 1-1.5 MB of memory space. On an embedded device with 64 MB of RAM, executing the operating system and a few applications as large as `gcc` would make the system run out of memory.

The first challenge in evicting code regions is dynamically identifying which code regions are dead. Heuristics must be used to approximately determine the lifetime of code regions. Previous work eliminated dead code regions by allowing them to mature for some time and then monitoring them for execution. If not executed during the monitoring period, code regions were presumed to be dead. Meanwhile, the other code regions were promoted to long-lived status. In our work, we explore this time-based heuristic for an embedded environment.

We also found that lifetime and execution count of code regions is strongly related, motivating us to use a heuristic based on the code region execution count (which does not depend on time elapsed) and to compare the two heuristics. The execution count-based heuristic eliminates most short-lived code regions and few long-lived code regions. In addition, our execution-count heuristic is simpler to calculate and manage than the time-based heuristic, making it more efficient in an embedded environment.

We found that the main trade-off between the time-based and the count-based heuristic is that the time-based heuristic offers better memory efficiency but is also more inaccurate than the count-based heuristic, resulting in more evictions of code regions which are not dead and need to be regenerated. The increased amount of code region regenerations result in the time-based heuristic having worse performance.

The second challenge is that evictions produce varying-sized, scattered holes (fragmentation) in the code cache and memory efficiency can be improved only by reusing the holes. Default code cache management techniques are not suitable for reusing such holes because information about the size and location of each hole is not maintained. Also since numerous holes can be formed, such information will occupy considerable space and considerable time may be spent searching for holes large enough to accommodate new code regions.

To avoid fragmentation, previous work divided the code cache into three parts - nursery, probation cache and persistent cache, promoting a code region from the nursery to the probation cache and then optionally promoting it to the persistent cache as it executes [12]. To simplify code cache management, we divide the code cache into temporary and permanent areas only (corresponding to the nursery and persistent caches). The temporary area stores all code regions until it is determined whether the code region is long-lived. Code regions are promoted to the long-lived area based on our heuristics. After several promotions, the temporary area should mostly contain dead code regions and can be completely flushed and reused. A size limit is placed on the temporary code cache to avoid memory expansion due to dead code regions. Flushes to reclaim space are triggered when the size limit is reached. The temporary area is guaranteed to be small because it is limited to a small size and is frequently reused. The permanent area is also expected to be small because a small subset of all generated code regions gets promoted to the permanent area.

A third challenge is avoiding excessive performance degradation due to frequent flushing of the temporary code cache. In the code cache, branch instructions are patched (or linked) so that they point to other code regions in the code cache instead of code in the original application binary. Such linking not only retains VEE control over code cache execution but also speeds up code cache execution as branches do not always have to refer to the VEE translator for resolving their target addresses. However, flushing requires that all incoming links to a code region be removed. Since temporary area flushes are anticipated to be frequent, unlinking may present a considerable overhead.

We explored the trade-offs of allowing and disallowing links to code regions in the temporary code cache. Allowing links speeds up execution in the temporary code cache but slows down flushes. Disallowing links requires the VEE to search the code cache address of each code region to be executed, and also generate the code region if it is not already in the code cache. The count-based heuristic favored the configuration without links while the time-based heuristic favored the configuration with links.

A fourth challenge is deciding whether to promote code regions to the permanent area early or late. In early promotions, code regions are promoted as soon as they become eligible. In late promotions, each temporary area code region is checked at the point of a flush to determine whether it is eligible for promotion. Early code promotions send long-lived code regions to the permanent area as soon as possible and reap more benefits from code cache locality in the permanent area. However, greater control over the temporary code cache execution is required for early promotion, possibly slowing it down. We explored the trade-offs of both early and late promotions. Late promotions were beneficial to memory efficiency. However, for the count-based heuristic, early promotion provided better performance. For the

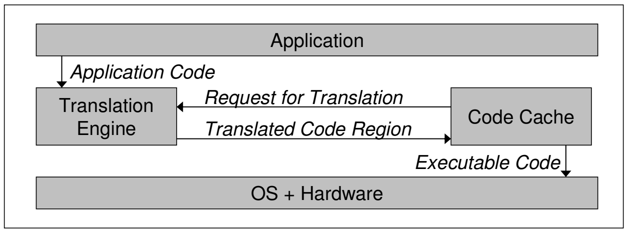

Figure 1. Block diagram of a typical translation-based VEE consisting of a translation engine and a code cache.

time-based heuristic, late promotion provided both improved performance and memory efficiency.

The specific contributions of this paper are as follows:

- strategies to reduce the code cache size based on lifetime and execution count characteristics of code regions
- implementation and evaluation of heuristics based on execution count and time passed since generation to distinguish between long-lived and short-lived code
- implementation of different code cache management techniques to evaluate trade-offs presented by linking versus not linking into the temporary code area and carrying out early versus late promotions
- achievement of at least 20% reduction in code cache size on the average, without significantly impacting performance

In Section 2, we provide an overview of the internal workings of a VEE, including the translation engine and the code cache. In Section 3, we discuss how code region characteristics motivate this work and their relationship to the heuristics used. We describe our code cache organization and code cache management techniques in Section 4, the actual implementation in Section 5, and experimental results in Section 6. We describe related work in Section 7 and conclude in Section 8.

2. Translation-Based VEEs

A VEE hosts and controls the execution of a given application. It can dynamically modify the application code to achieve additional functionality that is not present in the original application (such as dynamic binary instrumentation). Figure 1 is a simplified diagram of a translation-based VEE. The VEE consists of two main components – a translation engine and a software code cache. The translation engine is responsible for generating code and dynamically inserting it into the code cache. The software code cache is a memory area managed by the translator, which stores translated or modified code from the guest application. This code executes natively on the underlying layers. While the VEE may appear below the OS layer or may be co-designed with the hardware, the internal software architecture still corresponds to Figure 1.

2.1 Translation Engine

The translation engine fetches application code from the guest application, translates it and inserts it into the code cache. The application code can be translated at several different granularities, e.g., traces (single-entry, multiple-exit code units), basic blocks (single-entry, single-exit code units), or pages. Some extra code may be interleaved with the original code to achieve the VEE's goal. For example, Pin [17], a dynamic binary instrumentation VEE, interleaves instrumentation code with the guest application code.

The translation engine translates incrementally i.e., it translates only a small part of the guest application binary at a time. Therefore, it is possible that the next code to be executed is not present in the code cache. In such a case, a request for the next instruction address is generated, triggering translation to start.

In most cases, the translator stops translation at a branch instruction because branch instructions may have multiple outcomes. Based on the outcome, control continues on the fall-through path or jumps to some target address. Since it is not known beforehand which path the control will follow, some paths may not be translated and the next instruction may not be present in the code cache. Each such branch instruction points to an auxiliary construct called an exit stub. When the branch instruction is executed, control is diverted to the exit stub which triggers a context switch to the translator with a request to start translation at the next instruction address.

After generating requested code, the translation engine may patch the requesting branch to point to the newly inserted code if the requesting branch is a direct branch (the branch target is fixed) or a call to a known subroutine. This phenomenon is referred to as *linking*.

2.2 Code Cache

The code cache is a memory area allocated to store the translated application code and exit stubs (to transfer control back to the translator). The code cache may or may not have a size limit.

If a point is reached when there is not enough free space to insert a code region into the code cache, more code cache space is allocated. If the code cache is limited and the limit is reached, then some cached code must be deleted (or *flushed*) to make room for the new code. The code cache may be flushed entirely or in parts. Whenever a code region is marked for flushing, all branch instructions that point to it must be redirected to their stubs, so that control does not enter the flushed areas. When all the incoming links for the code region are removed, it can be flushed. Overheads of removing incoming links are especially relevant to our work because we flush the temporary code cache area frequently.

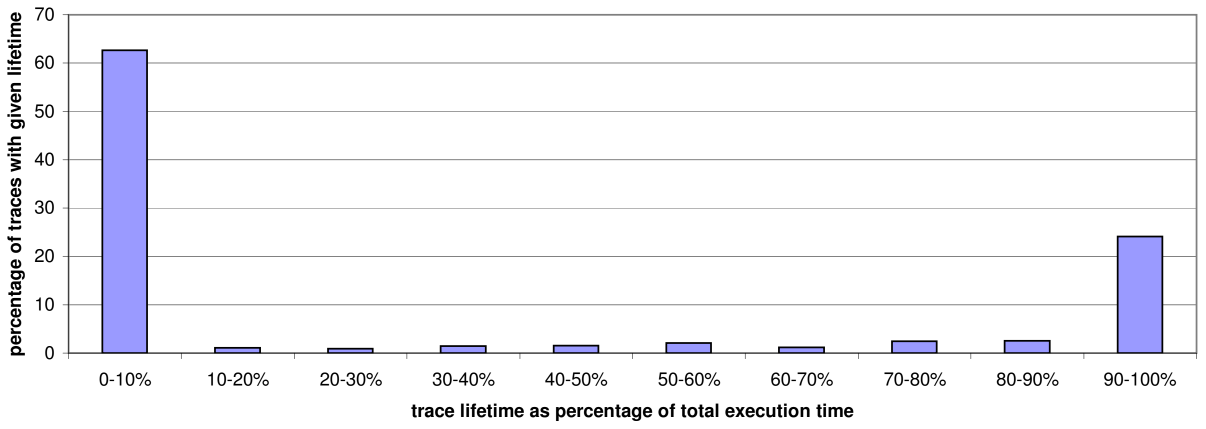

Figure 2. Percentages of code regions with certain lifetimes. 0-10% and 90-100% are by far the most major categories.

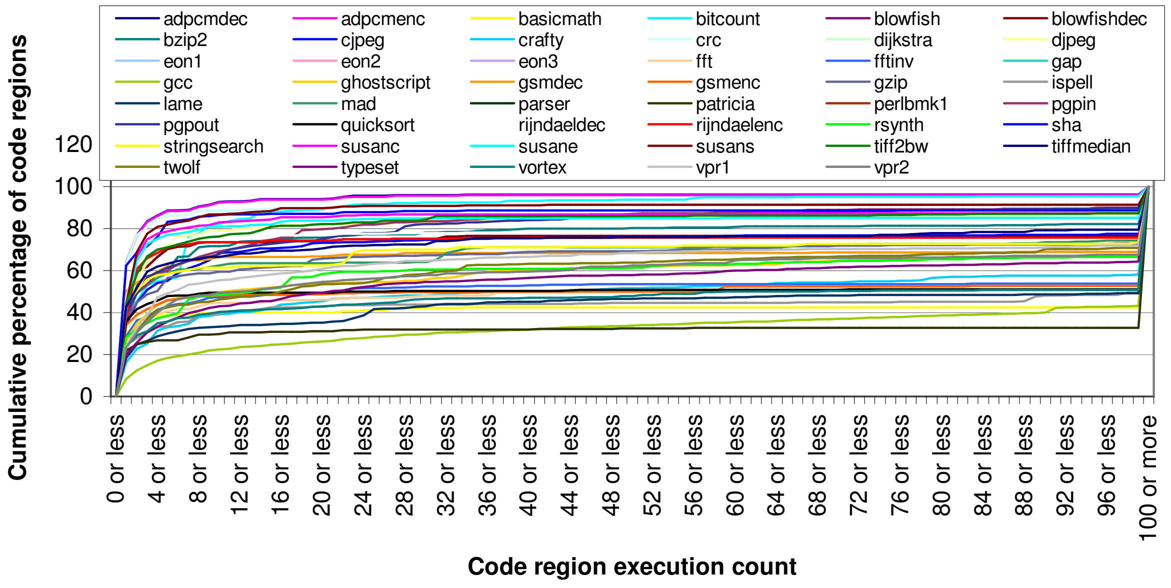

Figure 3. Percentages of code regions with execution counts within certain thresholds. Most code regions have execution counts of five or less.

3. Translated Code Characteristics

In this section, we discuss the code region characteristics that motivated our work. We evaluated the lifetime and execution count characteristics of code regions for the SPEC2000 integer suite [13] and the MiBench embedded benchmark suite [11]. We used Pin 2.0 for XScale [17] to host the execution of these benchmarks on a iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. All code regions formed by Pin are traces (single-entry multiple-exit code sequences).

Through experimentation, we found that the majority of code regions live for a small fraction of the total guest application execution time. Figure 2 shows the percentage of code regions in each lifetime category. The graph shows that on average, 60% of code regions live less than 10% of the guest application execution time, motivating us to detect these dead code regions and reclaim the space occupied by them.

Intuitively, a code region in the code cache should be allowed to remain for 10% of the guest application lifetime and then monitored for execution to determine whether a code region is long-lived. However, it is difficult to dynamically determine 10% of the guest application time before a program terminates. So time intervals between some predetermined events are used to decide when to start monitoring.

The graph also shows that the 0-10% and 90-100% lifetime categories are the most important. This fact, apart from the ease of code cache management, inspired us to divide the code cache into the temporary and permanent areas.

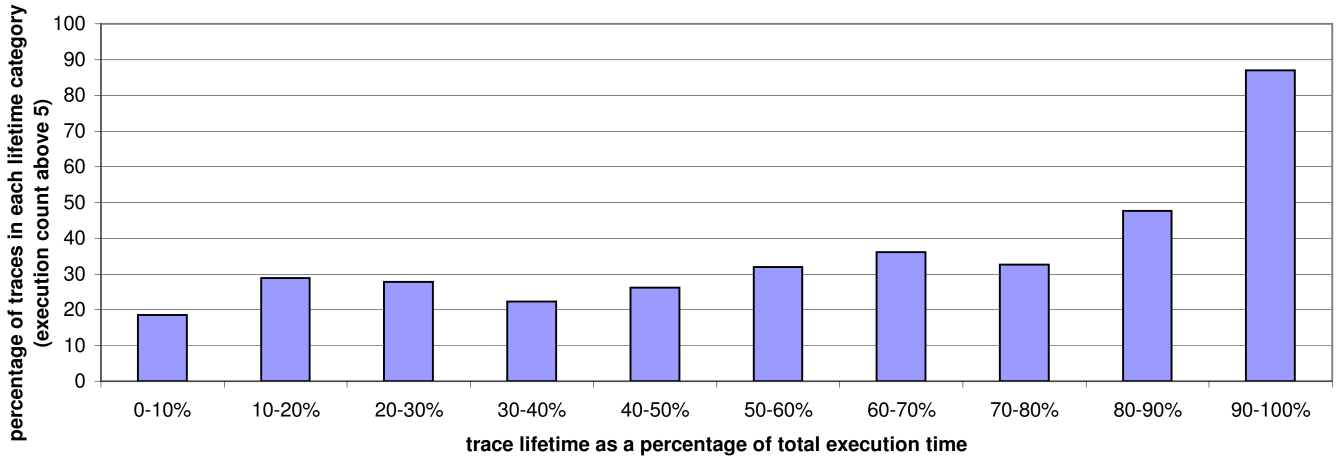

Figure 4. Percentages of code regions in each lifetime category, which have high execution counts of above five. Most long-lived code regions are included while few short-lived code regions are included.

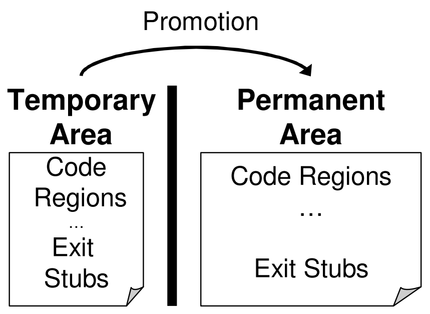

Figure 5. Code cache partitions. The temporary area is size limited and at one end of the code cache. This example depicts cache areas containing code regions and exit stubs.

We also examined the relationship between the execution count of a code region and its lifetime. First, we examined the execution counts of all code regions in a guest application. Figure 3 shows the percentage of code regions that execute a certain number of times. The graph has a step around five for all the benchmarks investigated. The step implies that a large percentage of code regions can be eliminated by placing the execution threshold at five, and in the context of embedded applications, five can be considered a high execution count.

Next, we explored the percentage of code regions in each lifetime category that has an execution count of more than five. Figure 4 shows that the lifetime category of 90-100% has a large fraction (90%) of code with a high execution count. The opposite is true for code regions in the 0-10% (20% code regions have a high execution count) category. This motivated us to set an execution count threshold of five to determine whether a code region is long-lived.

4. Code Cache Management

In this section we describe our code cache management schemes, which includes our code cache organization and two trade-offs that we explored. We considered whether to allow links into the temporary code cache and whether to use early or late promotions.

4.1 Restructuring the Code Cache

Once heuristics have identified a code region as dead, it is evicted. Simply evicting short-lived code regions from the code cache can form holes and complicate code cache management. Therefore, the code cache is partitioned into temporary and permanent areas as shown in Figure 5. Code regions initially enter the temporary area and are promoted to the permanent area based on our heuristics. After several promotions, we assume the temporary area contains mostly dead code regions and we flush and reuse it. We set a size limit of 128 KB on the temporary code cache to prevent memory expansion due to dead code regions and to initiate flushes to reclaim occupied space.

Memory requirements of code caches are expected to decrease in this scheme as a small percentage of code regions are long-lived, making the permanent area small. The temporary area is overwritten many times, and as a result, is also small. Performance can improve due to better instruction locality in the permanent area; however, overheads of promoting code regions, flushing cache areas and in some cases, regenerating evicted code regions, will be incurred.

4.2 Links in the Code Cache

Links are used in the code cache to improve the speed of execution. However, all links must be removed on a code cache flush. In our design, we allow all code regions to link into the permanent code cache because the permanent code cache is flushed rarely, if at all. However, temporary area flushes are anticipated to be frequent and managing incoming links

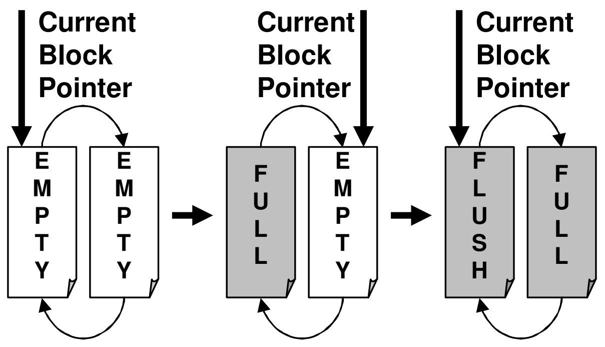

Figure 6. Temporary area cache blocks form a circular queue and are flushed in FIFO order to provide approximately uniform time to each code region to mature.

to the temporary area may present a considerable overhead. Therefore, we explore disallowing incoming links to code regions in the temporary area. Meanwhile, we acknowledge that suppressing links can cause performance overheads as there are more context switches between the code cache and the translator.

4.3 Early and Late Promotions

Code promotion can occur early or late. Early code promotions occur as soon as a code region is eligible for promotion. Late code promotions occur at the point of a temporary area flush. Early code promotions send long-lived code regions to the permanent area as soon as possible and reap more benefits from code locality in the permanent area. Late promotions reduce the number of context switches between the code cache and the translator because many code regions are promoted after a single context switch at the point of a flush. We explore the trade-offs of early and late promotions in our work.

5. Implementation

This section describes the implementation details of the reorganized code cache and code region promotions - the heuristics used to decide promotions and the actual process of promotion.

5.1 Code Cache Reorganization

The restructured code cache allocates 128 KB in the temporary area and 64 KB in the permanent area initially. The permanent area may increase in size as code regions are inserted into it, but the temporary area remains fixed at 128 KB.

The temporary area is flushed from time to time, however, the entire temporary area is never flushed at once. This is because some time needs to be given to each code region to execute and possibly complete its lifetime. If the entire temporary area is flushed at once, enough time is not given to the code regions that entered the temporary area close to the flush. Therefore, as shown in Figure 6, the temporary area is divided into blocks of 64 KB each and one block is flushed at a time in FIFO order.

5.2 Heuristics

A time-based heuristic and an execution count-based heuristic were motivated in Section 3. The time-based heuristic requires that a bit be maintained for each code region to indicate whether it executed while it was being monitored. Code region monitoring starts when the cache block containing the code region is next in line to be flushed. In the case of the count-based heuristic, the number of executions of a code region since its generation is tracked by maintaining an integer counter for each code region. We used an execution threshold of five.

5.3 Code Region Promotion

Code region promotion boils down to copying it from the temporary area to the permanent area and removing instrumentation en route. Besides copying, the links to and from the code region also have to be updated.

When to promote a code region depends on the heuristic. On every execution of a code region, the counter is first updated. Then a decision may be made whether to promote the code region.

If links to the code region are disabled, the translator is entered for each execution of the code region. The translator then updates the counter value and decides whether to promote the code region. Therefore, promotions are always early if links are not allowed.

However, if links to the temporary code cache are allowed, the translator cannot be used to update the counter. So instrumentation code is added to the beginning of each code region to update the counter value. Additionally, in the case of early promotions, code to decide whether to promote a code region is also added to the beginning of the code region. Updating and making decisions with the count heuristic required more instrumentation than the time-based heuristic.

However, the time-based heuristic with links and early promotion does not require any instrumentation code. Since, in early promotion, control will always enter the translator when the code region becomes eligible, one context switch per promoted code region is inevitable. In this case, all links to code regions are removed once monitoring starts. Whenever such a code region is executed, control enters the translator and promotes the code region. The memory overhead of extra code is avoided without any extra performance cost.

There is a performance overhead of updating counters before executing each code region. Also, when links are allowed, there is a memory overhead of inserting instrumentation code into the code cache. Insertion of extra code into the code cache also reduces code locality.

6. Experimental Results

This section evaluates the memory efficiency and the performance of our design. We analyze the sources of performance

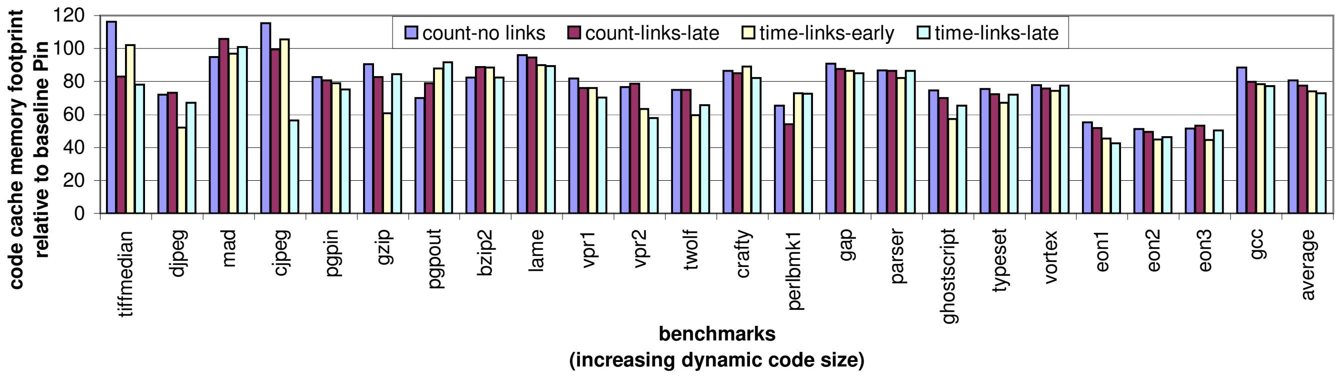

Figure 7. Code cache usage as percentage of code cache usage by baseline Pin (benchmarks with code cache sizes greater than or equal to that of `tiffmedian` are shown). Savings for the benchmarks are shown to be 20% on average.

degradation and discuss the resulting memory and performance. Both of the heuristics were evaluated by allowing and disallowing links into the temporary code cache. Absence of links implies early promotion. However, the cases in which links were allowed, we evaluated both early and late promotions. The count heuristic with links and early promotion and the time-based heuristic with no links have been excluded from the graphs as they have very high performance overheads.

As a baseline, we used Pin 2.0 for the XScale platform [17]. We implemented our solutions by directly modifying the Pin source code. Pin uses traces as code regions and assumes an unlimited code cache.

For the experiments, we ran the SPEC2000 integer suite [13] and the MiBench embedded benchmark suite [11] on a iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. It has a 200 MHz StrongARM-1110 processor with 64 MB RAM, 16 KB instruction cache and a 8 KB data cache. 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. In all the graphs, the benchmarks are arranged in increasing order of code cache size.

6.1 Memory Efficiency

Figure 7 shows the reduction in code cache size achieved by our schemes. Our schemes always allocate 128 KB in the temporary code cache and 64 KB in the permanent code cache. So, in measuring memory efficiency, only the benchmarks with code cache usage higher than 192 KB have been considered (we are specifically targeting the larger benchmarks). For the benchmarks originally having code caches smaller than 192 KB, marginally larger code caches were produced by our techniques.

Figure 7 shows that there is at least a 20% savings in code cache memory consumption for all the schemes. Some of the benchmarks with code cache footprint close to 192 KB suffer a loss because they are small compared to the code cache sizes we are targeting. It is clear that the time-based heuristic has better memory efficiency than the count-based heuristic, on average. But, as we shall see, the time-based heuristic eliminates many code regions prematurely. For the count-based heuristic, late promotion has better memory efficiency on average. This is because late promotion inserts extra code into the temporary code cache and thus accommodates less code regions at a time. So late promotion chooses code regions from a smaller set and promotes fewer code regions during a flush. The same is true for the time-based heuristic.

6.2 Performance Evaluation

Figure 8 shows the performance of the new schemes. In contrast to memory efficiency, the count-based heuristic performs better than the time-based heuristic on average, the reason for which will become clear when we delve into an analysis of overheads. For the count-based heuristic, count with links and early promotion (not shown here) has very poor performance because the code at the beginning of each code region is very complicated and occupies considerable space. Executing such code five times for every code region is very expensive. For the count-based heuristic, the absence of links performs better. This indicates that the efficiency of flushing outweighs the performance degradation due to absence of links. It is worth noting that in case of the largest benchmark, `gcc`, the count-based heuristic without links has by far the best performance because `gcc`'s working set is scattered over a large set of code regions and since this scheme promotes the largest number of code regions, it best covers the working set of `gcc` in the permanent area.

For the time-based heuristic, enabling links is better than disabling links (not shown in the graph) because code regions are left around in the code cache for a period of time before monitoring starts and disabling linking degrades performance heavily during this period. For the time-based heuristic, late promotion has better performance than early promotion because late promotion promotes less code regions and achieves better code locality in the permanent

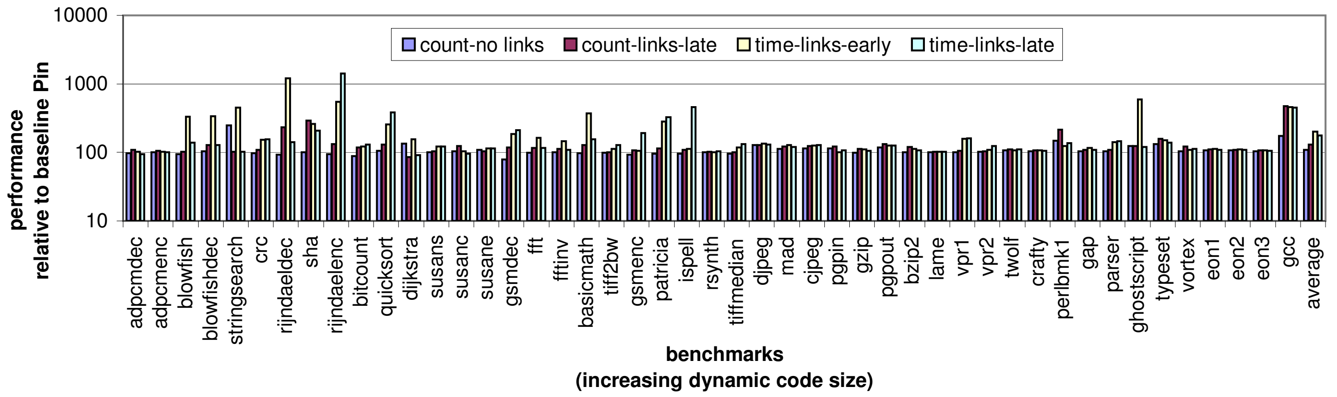

Figure 8. Performance with respect to baseline Pin (note the exponential axis). Count heuristic is better in general and the count heuristic with no links is the best.

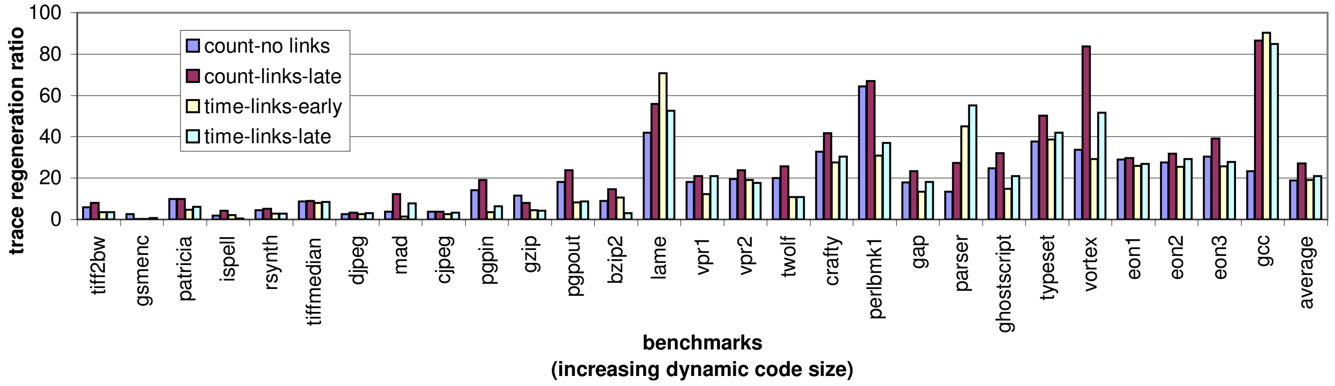

Figure 9. Extent of code region regeneration. The time-based heuristic has higher regenerations in most cases, explaining the cause of its worse performance. Benchmarks shown from `tiff2bw` because the ratios are close to zero before it.

code cache. Also, the time-based heuristic suffers considerably for the smaller benchmarks because better memory efficiency is not achieved in these but the performance overheads still incur. Among the larger benchmarks, poor performance for `gcc`, for example, can be attributed to the high number of regenerations.

6.3 Regeneration Overheads

Code region generation is an expensive task. Therefore, the higher the percentage of code region regeneration, the higher is the performance degradation. Figure 9 shows the ratio of the number of code region regenerations to the total number of code regions. Although the average percentage of regenerations is almost the same for all of the schemes, in some of the bigger benchmarks, they differ considerably. In the cases where the schemes differ, the count heuristic with no links is the best, explaining the cause of its superior performance. Figure 9 also explains why the time-based heuristic performs worse than the count-based heuristic. The time-based heuristic achieves better memory efficiency but is more inaccurate in its selection of code region and incurs more regeneration overhead.

6.4 Flushing Overheads

Flushes are also responsible for performance degradation. Figure 10 shows the number of temporary area flushes for our designs. Again, the count-based heuristic with no links has the least number of flushes. This may be due to the fact that the regenerations are lowest in this case.

6.5 Discussion

The results show that the count-based heuristic has better performance than the time-based heuristic and worse memory efficiency. This may be due to the fact that the time-based heuristic wrongly eliminates more code regions compared to the count-based heuristic, leading to more regenerations and more flushes.

For the count-based heuristic, not linking has better performance than linking into the temporary code cache. The opposite is true of the time-based heuristic. Absence of links for the count-based heuristic is better because the performance degradation is limited to five context switches between the code cache and the translator. The faster flushes achieved due to absence of links outweighs the context switch overheads. But for the time-based heuristic, perfor-

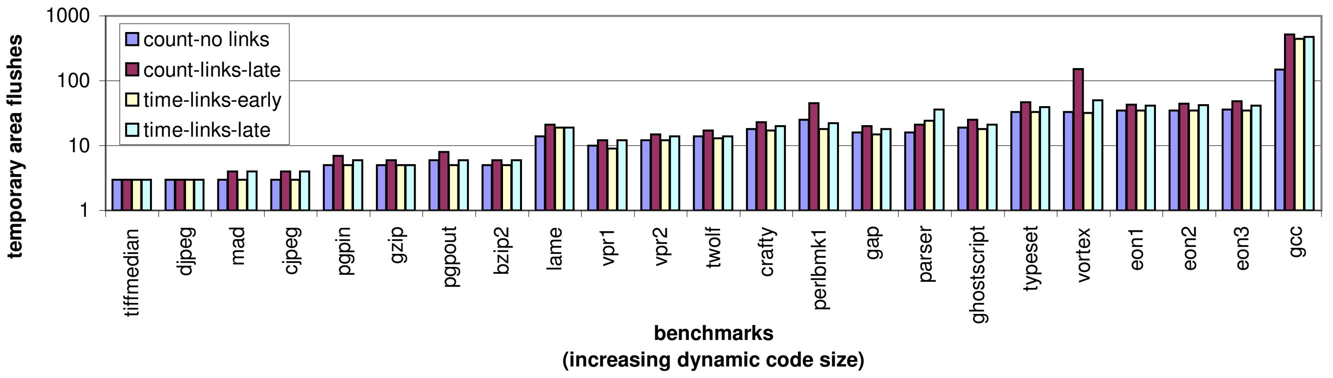

Figure 10. Number of temporary code cache flushes. Flushes are influenced by regenerations and also degrade performance. Benchmarks from `tiffmedian` are shown.

mance degradation due to absence of links continues until code region monitoring starts. Since this period of time is considerably longer than the time taken to execute a code region five times, the performance degradation outweighs the benefits of flushing.

The count-based heuristic requires more instrumentation than the time-based heuristic, and even more in the case of early promotion, resulting in early promotion with links being infeasible for the count-based heuristic. In case of the time-based heuristic, late promotion is better in both performance and memory than early promotion. Late promotion promotes less code regions and thereby achieves better code cache locality.

The different combinations evaluated have different memory and performance trade-offs. The count-based heuristic offers better performance while the time-based heuristic offers better memory efficiency. The actual combination to be used depends on the parameter (memory or performance) of focus. The VEE can adapt to changing conditions in an embedded device and strive for better memory efficiency or performance. For example, when few applications are being executed and the system has enough power, the focus of VEE hosted applications can be performance. After some time if the system starts a large graphics application, the focus of the VEE hosted applications can change to memory efficiency.

7. Related Work

Several VEEs have been developed for general-purpose machines. Among them are Dynamo [2], DynamoRIO [5], Strata [14, 15, 23] and Pin [17]. These VEEs provide features such as optimization and security. Pin [17] and DELI [8] are VEEs which support embedded platforms.

Apart from VEEs, in the embedded world, there are several Java virtual machines available. Standards such as Java card, J2ME/CLDC and J2ME/CDC have been built for embedded JVMs. KVM [18] was the first virtual machine developed by Sun Microsystems for embedded platforms. However KVM is a pure bytecode interpreter and performance suffers as a result. Later, Sun developed Hotspot [19] for embedded devices. JEPES [22] and Armed E-Bunny [6] are examples of research on embedded JVMs. There have also been efforts to reduce memory footprint in embedded JVMs [1, 22].

For VEEs, researchers have found that the code cache occupies five times the native application footprint [12]. Other researchers have found that the data structures required to support the code cache occupy about as much memory space as the code cache itself [21]. These facts have motivated several approaches to reduce the memory requirements of VEEs. Code cache management schemes inspired by generational garbage collection were explored [12]. Client-server approaches to manage code cache sizes in embedded systems [20, 25, 26] have also been explored. Other researchers have been motivated to reduce code cache memory expansion because many applications may be hosted by VEEs simultaneously, for example on a server [4]. Their work aimed to adaptively increase the code cache size limit as the guest application execution progresses. Finally, we have also explored techniques to reduce code cache sizes by focusing on exit stubs rather than code regions [10].

8. Conclusions

Our goal is to design memory and performance aware VEEs for embedded systems. While many approaches to this goal are possible, in this work we exploited code-lifetime and execution-count characteristics to achieve code cache size reduction. We found that both time-based and execution count-based heuristics can be used, resulting in different trade-offs. The time-based heuristic achieved better memory efficiency (around 25% code cache reduction on average) by being more selective in promoting code, but at the cost of more regenerations and degraded performance. The count-based heuristic promoted more code regions resulting in less code cache reduction (20% on average), but incurred less regenerations and less performance penalty. The results demonstrate that code characteristic-based VEE design has strong potential in embedded environments.

References

[1] David F. Bacon, Perry Cheng, and David Grove. Garbage collection for embedded systems. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 125–136, 2004.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000.

[3] D. L. Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2004.

[4] Derek Bruening and Saman Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 74–85, 2005.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, 2003.

[6] M. Debbabi, A. Mourad, and N. Tawbi. Armed e-bunny: a selective dynamic compiler for embedded java virtual machine targeting arm processors. In *Symposium on Applied Computing*, pages 874–878, Santa Fe, NM, 2005.

[7] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software. In *1st Int'l Symposium on Code Generation and Optimization*, pages 15–24, March 2003.

[8] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268, 2002.

[9] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Int'l Symposium on Computer Architecture*, pages 26–37, 1997.

[10] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. Reducing exit stub memory consumption in code caches. In *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, pages 87–101, Ghent, Belgium, January 2007.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench : A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, 2001.

[12] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *Transactions on Code Generation and Optimization (TACO)*, 3(3):263–294, September 2006.

[13] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 2000.

[14] J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. Evaluating fragment construction policies for sdt systems. In *Conference on Virtual Execution Environments*, pages 122–132, 2006.

[15] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Conference on Virtual Execution Environments*, pages 2–12, Ottawa, Canada, 2006.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, 2002.

[17] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[18] Sun Microsystems. *J2ME Building Blocks for Mobile Devices*. 2000.

[19] Sun Microsystems. *CLDC HotSpot Implementation Virtual Machine*. 2005.

[20] J. Palm, H. Lee, A. Diwan, and J. E. B. Moss. When to use a compilation service? In *LCTES '02: Proceedings of the 2002 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*.

[21] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 74–88, 2007.

[22] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. Compiling java for low-end embedded systems. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 42–50, San Diego, CA, July 2003.

[23] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *1st Int'l Symposium on Code Generation and Optimization*, pages 36–47, March 2003.

[24] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Int'l Symposium on Microarchitecture*, pages 271–282, 2005.

[25] L. Zhang and C. Krintz. Adaptive unloading for resource-constrained vms. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2004.

[26] S. Zhou, B. R. Childers, and M. L. Soffa. Planning for code buffer management in distributed virtual execution environments. In *Conference on Virtual Execution Environments*, pages 100–109, 2005.