Reducing Exit Stub Memory Consumption in Code Caches

Apala Guha, Kim Hazelwood, and Mary Lou Soffa

Department of Computer Science
University of Virginia

Abstract. The interest in translation-based virtual execution environments (VEEs) is growing with the recognition of their importance in a variety of applications. However, due to constrained memory and energy resources, developing a VEE for an embedded system presents a number of challenges. In this paper we focus on the VEE's memory overhead, and in particular, the code cache. Both code traces and exit stubs are stored in a code cache. Exit stubs keep track of the branches off a trace, and we show they consume up to 66.7% of the code cache. We present four techniques for reducing the space occupied by exit stubs, two of which assume unbounded code caches and the absence of code cache invalidations, and two without these restrictions. These techniques reduce space by 43.5% and also improve performance by 1.5%. After applying our techniques, the percentage of space consumed by exit stubs in the resulting code cache was reduced to 41.4%.

1 Introduction

Translation-based VEEs are increasingly being used to host application software because of their power and flexibility. The uses of VEEs include binary retranslation [1,2], program shepherding [3,4], power management [5] and many others. Although VEEs have been used in PC environments, they have not been thoroughly explored in the embedded world.

Embedded systems are widely used today. Personal mobile devices, sensor networks, and consumer electronics are fields that make extensive use of embedded technology. VEEs can have the same benefits in the embedded world as they do in the general-purpose world. For example, there are many different embedded instruction-set architectures, and as a result, little reuse of embedded software. Binary retranslation VEEs can address this issue. Security is important on embedded devices such as PDAs which often download third-party software, and program shepherding VEEs are important in this respect. In some situations, VEEs may be more important for embedded devices than general-purpose devices. For example, power management VEEs are arguably more important to battery-powered embedded devices than general-purpose machines.

VEEs introduce an extra software layer between the application and the hardware and use machine resources in addition to the guest application. For instance, Strata [3,6] has an average slowdown of 16% for the x86 architecture. Meanwhile, DynamoRIO has been shown to have a 500% memory overhead [7].

A code cache is used in most VEEs to store both application code and exit stubs. If the next instruction to be executed is not in the code cache, exit stubs (or trampolines) are used to return control to the VEE to fetch the next instruction stream. It is beneficial to reduce the space demands of a code cache. First, a small code cache reduces the pressure on the memory subsystem. Second, it improves instruction cache locality because the code is confined to a smaller area within memory, and is therefore more likely to fit within a hardware cache. Third, it reduces the number of cache allocations and evictions. Solutions for managing the size of code caches (using eviction techniques) have been proposed elsewhere [7], yet those studies focused on code traces. To our knowledge, this is the first body of work that focuses specifically on the memory demands of exit stubs. Furthermore, this is the first code cache investigation that focuses on an embedded implementation.

Exit stubs typically have a fairly standard functionality. They are duplicated many times in the code cache and are often not used after the target code region is inserted into the code cache, providing ample opportunity for optimization.

In this paper, we explore the memory overhead of stubs in a translation-based VEE and examine techniques for minimizing that space. We present four techniques that work for both single-threaded and multi-threaded programs. The first two techniques delete stubs that are no longer needed but assume unlimited code caches and the absence of flushing. They remove stubs in their entirety when these stubs are guaranteed not to be needed anymore. Although there are many such applications which do not violate these assumptions and still have a reasonable code cache size, it is also important to be able to handle situations where flushing occurs. The last two techniques, therefore, lift these restrictions and identify stub characteristics that can reduce space requirements.

The specific contributions of this paper are:

– A demonstration of the overhead of exit stubs in a code cache.
– The presentation of two schemes that use a deletion approach to reduce the space occupied by stubs in an application independent and partially VEE-independent manner.
– The presentation of two schemes which identify characteristics of stubs which can be further optimized to minimize their space requirements.
– Experimental results that demonstrate these schemes not only save space but also improve performance.

In Sect. 2, we provide an overview of the internal workings of a VEE, including the translation engine, the code cache, and an overview of exit stubs. In Sect. 3, we describe the four techniques developed for reducing the size of the code cache by optimizing the exit stubs. The experimental evaluation of our techniques is presented in Sect. 4. We describe related work in Sect. 5 and conclude in Sect. 6.

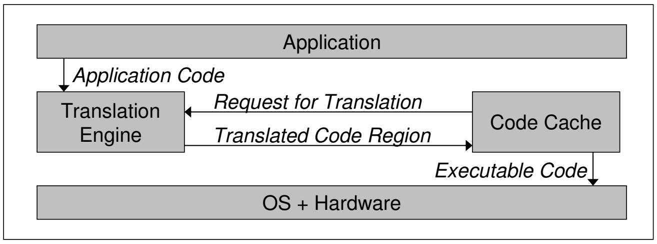


Fig. 1. Block diagram of a typical translation-based VEE

2 Background

A VEE hosts and controls the execution of a given application. It can dynamically modify the application code to achieve functionality that is not present in the original application. Fig. 1 is a simplified diagram of a translation-based VEE. The VEE is shown to consist of two components – a translation engine and a software code cache. The software code cache is a memory area managed by the translator, which contains code from the guest application. The translation engine is responsible for generating code and inserting it into the code cache dynamically. 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 software architecture still corresponds to Fig. 1.

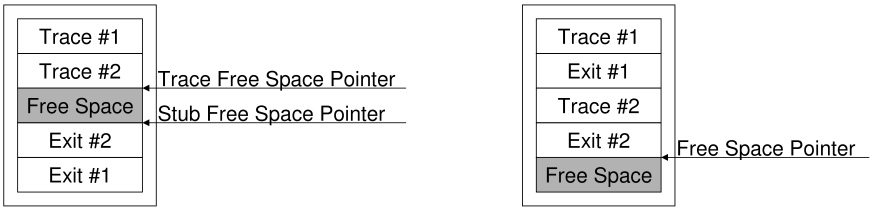
2.1 Translation Engine

The translation engine dynamically translates and inserts code into the code cache. As the cached code executes, it may be necessary to insert new code into the code cache. Requests to execute target code are generated on the fly and sent to the translation engine. After generating and inserting the desired code, the translation engine may patch the requesting branch to point to the newly inserted code. If the requesting branch is a direct branch or call, patching is always done to link the branch to its target. However, in the case of indirect branches, linking does not occur because the branch target may change.

2.2 Code Cache

The code cache is a memory area allocated to store the guest application code. It may be allocated as a single contiguous area of memory or in regions (called *cache blocks*), which may appear at arbitrary memory locations. If composed of several blocks, these blocks may be allocated at one time or on demand.

The code cache consists of application code and exit stubs. Application code is fetched from the guest application (and may or may not appear in the same physical order as in the original application). Some extra code may be interleaved with the original code to achieve the VEE's goal. For example, Pin [8] interleaves instrumentation code with the guest application code. The application code can



(a) Traces and stubs filled from opposite ends of code cache

(b) Traces and stubs contiguous in code cache

Fig. 2. Two typical arrangements of traces and exit stubs in code cache

be cached at several different granularities, e.g., traces (single-entry, multiple-exit code units), basic blocks (single-entry, single-exit code units), or pages. For every branch instruction in the application code, there exists an exit stub in the code cache.

Fig. 2 shows two typical arrangements of application code and exit stubs in the code cache. In Fig. 2, the code is inserted into the code cache as traces (which is common). A trace has several trace exits. There is a stub for each trace exit, and the stubs for a particular trace form a chain. Each block marked *Exit* in Fig. 2 symbolizes a chain of stubs for a given trace. As shown in Fig. 2(a), a code cache may be filled by inserting traces and stubs in different portions (opposite ends, for example) of the code cache, or a trace and its corresponding stubs can be contiguous in the code cache as shown in Fig. 2(b).

If a point is reached when there is not enough free space for insertion of a code unit into the code cache, new cache blocks are allocated. If the code cache is limited and the limit is reached, or allocation is not allowed, then some cached code must be deleted to make room for the new code. Deletions may occur at several granularities: (1) the entire code cache may be flushed, (2) selected cache blocks may be flushed, (3) selected traces may be flushed. Whenever a code unit is marked for flushing, all branch instructions that point to it must be redirected to their stubs, so that control may return to the translator in a timely manner. When all the incoming links for the code unit are removed, it can be flushed. Flushing may occur all at once, or in a phased manner. For example, for multi-threaded host applications, code caches are not flushed until all threads have exited the corresponding code cache area.

2.3 Exit Stubs

An exit stub is used to return control to the translator and communicate the next instruction to translate, if it is not available in the code cache. Some code and data are needed to transfer control to the translator. The code will typically consist of saving the guest application's context, constructing arguments to be passed or loading the address of previously stored arguments and branching into

```
1  sub sp, sp, 80h
2  stm sp, [mask = 0xff]
3  ld r0, [addr of args]
4  ld pc, [addr of handler]
```

(a) Pin's exit stub code for ARM

```
1  mov eax, [predef memory location]
2  mov [addr of stub data], eax
3  jmp [addr of translator handler]
```

(b) DynamoRIO's exit stub code for x86

```
1  pusha
2  pushf
3  push [target PC]
4  push [fragment addr]
5  push reenter code cache
6  jmp [addr of fragment builder]
```

(c) Strata's exit stub code for x86

Fig. 3. Examples of exit stub code from different VEEs

the translator handler. The data typically consists of arguments to be passed, such as the target address for translation.

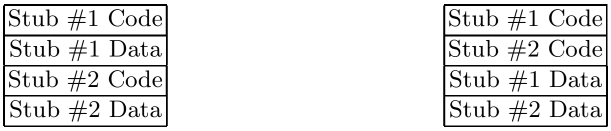
Fig. 3 shows the exit stub code for three different VEEs – Pin, DynamoRIO, and Strata. In line 1 of Fig. 3(a), the application stack pointer is shifted to make space for saving the context. In line 2, the application context is saved using the `store multiple register (stm)` command. The mask 0xff specifies that all 16 registers have to be saved. In line 3, register `r0` is loaded with the address of translator arguments. In line 4, the program counter is loaded with the translator handler's address, which is essentially a branch instruction.

Fig. 3(b) shows DynamoRIO's exit stub code. The `eax` register is saved in a pre-defined 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.

Fig. 3(c) shows a Strata exit stub. The context is saved and arguments are passed to the translator. The address for reentering the code cache is saved for tail call optimization. The last instruction transfers control to the translator.

Exit stubs have associated data blocks that are arguments passed to the translator. The exact arguments depends on the particular VEE. In Pin, the target address for the branch instruction corresponding to the stub is stored. The exit stub can correspond to direct or indirect branches or calls to emulate code and each branch is serviced differently. So, the exit stub stores the type of branch it services, and other data, such as a hash code corresponding to the region of code requested.

Fig. 4 shows two possible layouts of stubs for a given trace. In Fig. 4, stubs 1 and 2 are constituents of the *Exit* block in Fig. 2. The code and data for each stub may appear together as shown in Fig. 4(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 Fig. 4(b). As we show in Sect. 3.3, the first layout conserves more space.



(a) Intermixed stub code and data (b) Separated stub code and data

Fig. 4. Two arrangements of stubs for a given trace in the code cache

Initially, all branches are linked to their corresponding exit stubs. However, when a branch gets linked to its target, these exit stubs become unreachable.

Table 1. Percentage of code cache consisting of exit stubs

VEE	**Exit Stub Percentage**
Pin	66.67%
Strata	62.59%
DynamoRIO	62.78%

Table 1 shows the space occupied by exit stubs in Pin, Strata and DynamoRIO [9]. As the numbers demonstrate, the large amount of space occupied by stubs show that a lot of memory is being used by code that does not correspond to the hosted application. The data for Pin and Strata were obtained using log files generated by the two VEEs. The data for DynamoRIO[9] was calculated from space savings achieved when exit stubs are moved to a separate area of memory.

3 Methodology

In this section, we describe our approaches for improving the memory efficiency of VEEs by reducing the space occupied by stubs. We describe four techniques, all of which are applicable to both single-threaded and multi-threaded applications. The first set of solutions eliminate stubs, while the latter set reduces the space occupied by stubs. However, the first two approaches are restricted to the case of unbounded code caches where no invalidations of cached code occur. The last two approaches are free of these restrictions.

3.1 Deletion of Stubs (D)

In the deletion of stubs (D) scheme, we consider deleting those stubs that become unreachable when their corresponding branch instructions are linked to their targets. We delete only those exit stubs that border on free space within the code cache. For example, assume the code cache is filled as in Fig. 2(a) and the stubs are laid out as in Fig. 4(a). If the branch corresponding to stub 1 is linked

```
1  if (branch corresponding to an exit stub is linked to a trace)
2      if (end of exit stub coincides with free space pointer)
3          move free space pointer to start of exit stub
```

Fig. 5. Algorithm for deletion of stubs

and stub 1 is at the top of the stack of exit stubs, stub 1 can be deleted. We chose not to delete stubs that are in the middle of the stack of exit stubs, as this will create fragmentation which complicates the code cache management. (We would have to maintain a free space list which uses additional memory. In fact, each node on the free space list will be comparable to the size of the exit stub.) Deletions are carried out only in those areas of the code cache where insertions can still occur. For example, if the code cache is allocated as a set of memory blocks, deletions are only carried out in the last block.

Fig. 5 shows our stub deletion algorithm. A code cache has a pointer to the beginning of free space (as shown in Fig. 2) to determine where the next insertion should occur. If the condition in line 1 of Fig. 5 is found to be true, the free space pointer and the exposed end of the stub are compared in line 2. If the addresses are equal, then the stub can be deleted by moving the free space pointer to the other end of the stub and thereby adding the stub area to free space, as shown in line 3.

The limitation of this scheme is that the trace exits whose stubs have been deleted may need to be unlinked from their targets and reconnected to their stubs during evictions or invalidations. So, this scheme can work only when no invalidations occur in the code cache (because we would need to relink all incoming branches back to their exit stubs prior to invalidating the trace). It can work with a bounded code cache only if the entire code cache is flushed at once, which is only possible for single-threaded applications.

We did not explore other techniques such as compaction and regeneration of stubs for this paper. We believe that a compaction technique is complicated to apply on-the-fly and needs further investigation before its performance and space requirements can be optimized enough for it to be useful. Regeneration of stubs on the other hand, creates a sudden demand for free code cache space. This is not desirable because regeneration of stubs will be needed when cache eviction occurs and cache eviction usually means that there is a shortage of space. Creating space by removal of traces relates to the same problem of requiring the stubs for branches which link to these traces.

3.2 Avoiding Stub Compilation (ASC)

In Scheme D, many stubs whose corresponding trace exits were linked to their targets could not be deleted because the stubs were not at the edge of free space in the code cache. To alleviate this problem, we observed that among such stubs there are many that never get used. The reason is that the trace exits corresponding to them get linked before the trace is ever executed. This linking occurs if the targets are already in the code cache. In these situations, it is not

```
 1  for every trace exit
 2      targetaddr = address of trace exit
 3      targetfound = false
 4      for every entry in a code cache directory
 5          if (application address of entry == targetaddr)
 6              patch trace exit to point to code cache address of entry
 7              targetfound = true
 8           break
 9      if (targetfound == false)
10         generate stub
```

Fig. 6. Algorithm for avoiding compilation of traces

necessary to compile the stub because it will never be used. This strategy saves not only space but also time.

Fig. 6 displays the algorithm for the avoiding stub compilation (ASC) scheme. For every trace exit, the target address is noted in line 2. A flag is reset in line 3 to indicate that the target of the trace exit does not exist in the code cache. Line 4 iterates over all entries in the code cache directory. The application address of each directory entry and the target address are compared in line 5. If a match is found, the trace exit is immediately patched to the target in line 6. After the code cache directory is searched, if the target has not been found, the stub for the trace exit is generated in line 10.

For this scheme, it is always possible for a translator to find target traces that exist already in the code cache and hence avoid compilation. Thus this scheme is independent of the particular VEE's architecture. However, it suffers from the same limitation as Scheme D in that individual deletions may not be allowed. The next schemes overcome these limitations and result in greater savings of time and space.

3.3 Exit Stub Size Reduction (R)

There are some circumstances when exit stubs are necessary, such as during a multi-threaded code cache flush, or a single trace invalidation. In these cases, traces must be unlinked from their target, and relinked to their exit stub. Schemes D and ASC are not adequate for applications that exhibit this behavior. We now present a scheme to minimize the size of a stub and still maintain its functionality.

In the exit stub size reduction (R) scheme, some 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 in order to remember the stub location. Fig. 7 shows the code in the stub after this optimization. Saving the context may take several instructions, depending on the architecture. Here, only one register, the program counter is being saved. Factoring of common code is a simple technique that has been implemented in some form in systems (e.g., DynamoRIO), already.

Save Program Counter
Branch to translator Handler
Target Address
Translator Handler Address

Fig. 7. Structure of stub after optimization using reduction in stub size

We handle only the case of direct branches and calls as they are the only branches that are actually linked to their targets. As a result, we know the kind of service being requested and hence can avoid storing the type of service in the stub data area. A specialized translator handler handles these stubs.

We also avoid storing any derivable data within the stub. We reconstruct the derivable arguments to the translator before entering the translator handler. The code for reconstructing the derivable arguments is put in a common bridge routine between the stubs and the translator handler. Thus, we save space by avoiding the storing of all the arguments to the translator and avoid storing code to construct these arguments in the stub. This gives rise to a trade-off with performance but since stubs are not heavily accessed, such reconstruction is not a large time penalty. The stub in Fig. 7 stores only the target address which is not derivable. Storage of derivable data such as a hash of the target address which may enable a faster search of the code cache directory is avoided.

We adhere to the layout in Fig. 4(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. (This is not possible for the configuration in Fig. 4(b).) The stub's start address is known from the program counter saved in the context.

This scheme is applied to all exit stubs corresponding to direct branches and calls. Phased flushing and invalidations can be handled by Scheme R 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.

3.4 Target Address Specific Stubs (TAS)

The main task of a stub is to provide the translator with the address of code to be translated next. Yet more than one source location often targets the same address, but uses a different exit stub. For our final scheme, target address specific stub generation (TAS), we ensure that trace exits requesting the same target address use the same exit stub.

Fig. 8 shows the algorithm used in this scheme. The target address of each trace exit is examined. The stub corresponding to the target address of each trace exit is searched in line 3. If the required exit stub exists, it is designated as the exit stub for the trace exit in line 4. Otherwise a new stub is generated and this stub's location is recorded in lines 6-7.

Reuse of exit stubs can occur at several different granularities. For example, stubs may be reused across the entire code cache. Or the code cache may be partitioned and stubs may be reused only inside these partitions. In our implementation,

```
1  for each trace exit
2      targetaddr = target address of trace exit
3      if there exists a stub for targetaddr in this block
4          designate this stub as the exit stub for this trace exit
5      else
6          generate an exit stub for this trace exit
7          store address of exit stub for targetaddr
```

Fig. 8. Algorithm for using target address specific stubs

we reused stubs only at the partition level. These partitions are the *blocks* in line 3 of Fig. 8. The granularity of reuse is important because flushing cannot be carried out at a granularity finer than that of reuse. If flushing is carried out at a finer granularity then there is the danger of not having the required stubs within the code cache portion being flushed. We used the medium granularity as that provides good performance as shown elsewhere [7].

The challenge in applying this technique is that it is not known beforehand whether the trace being compiled will fit into the current block and will be able to reuse the stubs from the current block. We optimistically assume that the trace being compiled will fit into the current block. If it does not ultimately fit, we simply copy the stubs into the new block. However, the case in which the current block gets evicted, copying cannot be carried out. To safeguard against such a situation, we stop reusing stubs when a certain percentage of the code cache size limit has been used, such that the remaining unused portion is larger than any trace size in the system.

All the techniques mentioned above are complementary and can be combined together. If all four techniques are combined, then there is the limitation of cache flushing. However if only the R and TAS techniques are combined, then this limitation does not exist. However, flushing has to be carried out at an equal or coarser granularity than that of stub reuse in TAS.

4 Experimental Results

We evaluated the memory efficiency and performance of our proposed techniques. As a baseline, we used Pin [8] running on an ARM architecture. We implemented our solutions by directly modifying the Pin source code. Before starting the evaluation, we improved indirect branch handling in Pin to predict branch targets without returning to the translator (which was not implemented in the ARM version). This makes Pin faster and the process of experimentation easier. This optimization was included in both the baseline and our modified versions of Pin.

For the experiments, we ran the SPEC2000 integer suite[1] 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 benchmarks were run on test inputs, since there was

[1] We omitted SPECfp because our iPAQ does not have a floating-point unit.

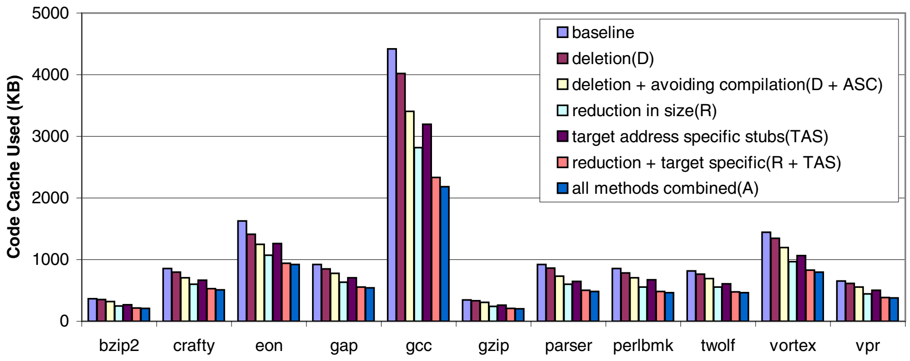


Fig. 9. Memory usage (reported in kilobytes) of Pin baseline (leftmost bar) and after incorporating our optimizations (rightmost bars)

not enough memory on the embedded device to execute larger inputs (even natively). Among the SPEC2000 benchmarks, we did not use `mcf` because there was not enough memory for it to execute natively, regardless of input set. We chose SPEC2000 rather than embedded applications in order to test the limits of our approach under memory pressure.

Pin allocates the code cache as 64 KB cache blocks on demand. Pin fills the code cache as shown in Fig. 2(a) and lays out stubs as shown in Fig. 4(b).

4.1 Memory Efficiency

The first set of experiments focused on the memory improvement of our approaches. Fig. 9 shows the memory used in the code cache in each version of Pin as kilobytes allocated. The category *original* is the number of KBs allocated in the baseline version.

For Scheme D, the average memory savings is 7.9%. The benefits are higher for the larger benchmarks. For example, it offers little savings in `bzip2` and `gzip`, which have the two smallest code cache sizes. But in `gcc` which has the largest code cache size, it eliminates 9% of the code cache space. This shows that this scheme is more useful for applications with large code cache sizes, which is precisely what we are targeting in this research.

The next scheme combines ASC and D. Here, the average memory savings increase to 17.8%. Similar to Scheme D, it is more beneficial for applications with larger code cache sizes and less so for those with smaller code caches.

In Scheme R, the memory efficiency is considerably improved from the previous schemes. The average savings in memory in this scheme is 37.4%. The increase is due to the fact that memory is saved from all stubs corresponding to 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).

Scheme TAS shows a memory efficiency improvement of 24.3%. Furthermore, it is complementary to Scheme R. The combination of schemes TAS and R result in a 41.9% improvement in memory utilization.

Table 2. Percentage of code cache occupied by exit stubs after applying our techniques

Scheme	Exit Stub Percentage
Baseline	66.67%
Deletion (D)	63.92%
Avoiding compilation + Deletion (ASC + D)	59.68%
Reduction in Stub Size (R)	51.24%
Target address specific stubs (TAS)	55.76%
Reduction in size + Target specific stubs (R + TAS)	43.37%
All schemes combined (A)	41.40%

The four schemes combined together achieve memory savings of 43.6%. Therefore, we see that the bulk of the benefit comes from schemes TAS and R which do not carry flushing restrictions.

Table 2 shows the percentage of code cache occupied by stubs (with respect to the code cache size after every optimization) before and after each of our solutions. We were able to reduce stub occupancy from 66.7% to 41.4%.

4.2 Performance Evaluation

In this section, we evaluate the performance of our approaches. Fig. 10 shows the normalized performance of our schemes with respect to the baseline version. Scheme D has almost the same performance as the original version. Extra work is being done in Scheme D to delete stubs. At the same time, more traces inserted into a cache block resulted in improved instruction cache locality. Code cache management time is reduced due to less cache block allocations.

In Scheme ASC + D, some extra time is spent searching the code cache directory for each branch instruction to determine whether an exit stub needs to be compiled. At the same time, the amount of compilation is reduced. Combining these factors with Scheme D yields an improvement in performance.

Scheme R performs as well as ASC + D. Here performance suffers from the fact that derivable arguments are constructed on each entry into the translator due to a direct branch or call instruction. As before, better instruction cache locality and reduced compilation and code cache management time contribute positively to performance. Using TAS also yields about the same improvement.

Using a combination of techniques yields overall performance improvement of about 1.5%, which is especially encouraging given that our main focus was memory optimization. It is important to note here is that the techniques perform better for benchmarks with larger code cache sizes. For example `gcc` yields 15-20% improvement when combination techniques are applied to it. Smaller benchmarks such as `bzip2` and `gzip` do not reap great benefits in comparison. Benchmarks that use a lot of indirect branches such as `eon` also do not show considerable improvement. This is due to the fact that the indirect branch handling methods in the XScale version of Pin could benefit from further refinement.

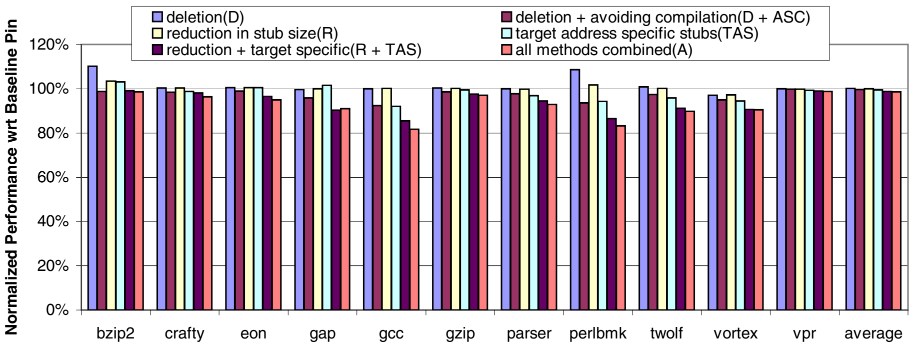


Fig. 10. Performance of proposed solutions as normalized percentages. (100%) represents the baseline version of Pin and smaller percentages indicate speedups.

4.3 Performance Under Cache Pressure

In our next set of experiments, we measure the performance of our approaches in the case of a limited code cache. We evaluated the R and TAS approaches in the presence of cache pressure. We set the cache limit at 20 cache blocks (1280 KB) as this is a reasonable code cache size on our given system. We did not include `gcc` because the ARM version of Pin fails with this code cache limit for `gcc`, even without any of our modifications.

Our approaches performed 5-6% better on average. The performance improvement is due to a smaller code cache and a reduced number of code cache flushes. In the limited cache situation, Scheme R performs better than the Scheme TAS (the case was opposite in the unlimited code caches). This is because Scheme R needs fewer cache flushes. The combined Scheme R + TAS performs best in all cases except `perlbmk`.

5 Related Work

Several VEEs have been developed for general-purpose machines. Among them are Dynamo [10], DynamoRIO [11] and Strata [6]. These VEEs provide features such as optimization and security. In the embedded world, there are relatively few VEEs, with most being Java virtual machines. Standards such as Java card, J2ME/CLDC and J2ME/CDC have been built for embedded JVMs. JEPES [12] and Armed E-Bunny [13] are examples of research on embedded JVMs. Pin [8] is a VEE which supports the XScale platform, but is not a JVM.

There have been many research efforts to reduce the memory footprint of embedded applications [12,14,15]. Management of code cache sizes has been explored [7]. Hiser et al. [16], explore the performance effects of putting fragments and trampolines in separate code cache areas and eliding conditional transfer instructions. However, to the best of our knowledge, reducing memory footprint of VEEs by reducing the size of exit stubs has not been explored before.

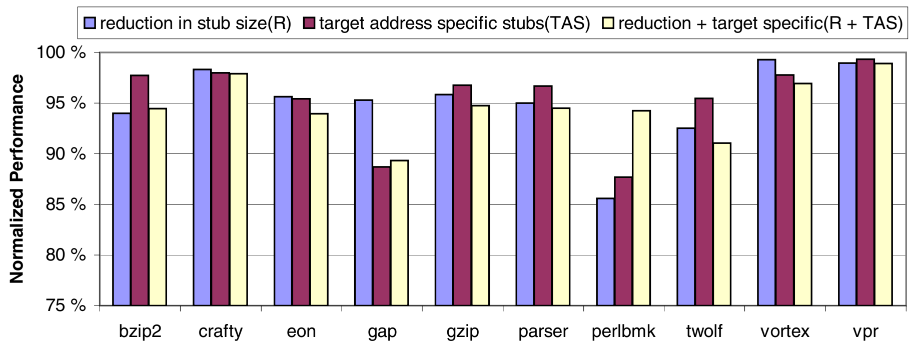


Fig. 11. Performance of `exit stub reduction` and `target address specific stub generation` with cache limit of 20 blocks (1280 KB)

6 Conclusions

Memory usage by VEEs needs to be optimized before they can be used more extensively in the embedded world. In this paper, we explore memory optimization opportunities presented by exit stubs in code caches. We identify reasons that cause stubs to occupy more space than they require and solve the challenge by developing schemes that eliminate a major portion of the space consumed by exit stubs. We show that memory consumption by the code cache can be reduced up to 43% with even some improvement in performance. We also show that performance improvement is even better for limited size code caches which are used when the constraints on memory are even more severe.

References

1. Dehnert, J., Grant, B., Banning, J., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing software. In: 1st Int'l Symposium on Code Generation and Optimization. (2003) 15–24
2. Ebcioglu, K., Altman, E.R.: DAISY: Dynamic compilation for 100% architectural compatibility. In: 24th Int'l Symposium on Computer Architecture. (1997)
3. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J.W., Evans, D., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J.: Secure and practical defense against code-injection attacks using software dynamic translation. In: Conference on Virtual Execution Environments, Ottawa, Canada (2006)
4. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: 11th USENIX Security Symposium. (2002)
5. Wu, Q., Martonosi, M., Clark, D.W., Reddi, V.J., Connors, D., Wu, Y., Lee, J., Brooks, D.: A dynamic compilation framework for controlling microprocessor energy and performance. In: 38th Int'l Symposium on Microarchitecture. (2005)
6. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., Soffa, M.L.: Reconfigurable and retargetable software dynamic translation. In: 1st Int'l Symposium on Code Generation and Optimization. (2003) 36–47

7. Hazelwood, K., Smith, M.D.: Managing bounded code caches in dynamic binary optimization systems. Transactions on Code Generation and Optimization (TACO) **3** (2006) 263–294
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Janapareddi, V., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Conference on Programming Language Design and Implementation, Chicago, IL (2005)
9. Bruening, D.L.: Efficient, Transparent and Comprehensive Runtime Code Manipulation. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (2004)
10. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A transparent dynamic optimization system. In: Conference on Programming Language Design and Implementation. (2000) 1–12
11. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: 1st Int'l Symposium on Code Generation and Optimization. (2003) 265–275
12. Schultz, U.P., Burgaard, K., Christensen, F.G., Knudsen, J.L.: Compiling java for low-end embedded systems. In: Conference on Languages, Compilers, and Tools for Embedded Systems, San Diego, CA (2003)
13. Debbabi, M., Mourad, A., Tawbi, N.: Armed e-bunny: a selective dynamic compiler for embedded java virtual machine targeting arm processors. In: Symposium on Applied Computing, Santa Fe, NM (2005)
14. Bacon, D.F., Cheng, P., Grove, D.: Garbage collection for embedded systems. In: EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software, New York, NY, ACM Press (2004)
15. Zhou, S., Childers, B.R., Soffa, M.L.: Planning for code buffer management in distributed virtual execution environments. In: Conference on Virtual Execution Environments, New York, NY, ACM Press (2005)
16. Hiser, J.D., Williams, D., Filipi, A., Davidson, J.W., Childers, B.R.: Evaluating fragment construction policies for sdt systems. In: Conference on Virtual Execution Environments, New York, NY, ACM Press (2006)