Planning for Code Buffer Management in Distributed Virtual Execution Environments

Shukang Zhou
Dept. of Computer Science
Univ. of Virginia
zhou@cs.virginia.edu

Bruce R. Childers
Dept. of Computer Science
Univ. of Pittsburgh
childers@cs.pitt.edu

Mary Lou Soffa
Dept. of Computer Science
Univ. of Virginia
soffa@cs.virginia.edu

ABSTRACT

Virtual execution environments have become increasingly useful in system implementation, with dynamic translation techniques being an important component for performance-critical systems. Many devices have exceptionally tight performance and memory constraints (e.g., smart cards and sensors in distributed systems), which require effective resource management. One approach to manage code memory is to download code partitions on-demand from a server and to cache the partitions in the resource-constrained device (client). However, due to the high cost of downloading code and re-translation, it is critical to intelligently manage the code buffer to minimize the overhead of code buffer misses. Yet, intelligent buffer management on the tightly constrained client can be too expensive. In this paper, we propose to move code buffer management to the server, where sophisticated schemes can be employed. We describe two schemes that use profiling information to direct the client in caching code partitions. One scheme is designed for workloads with stable run-time behavior, while the other scheme adapts its decisions for workloads with unstable behaviors. We evaluate and compare our schemes and show they perform well, compared to other approaches, with the adaptive scheme having the best performance overall.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Memory Management, Run-Time Environments

General Terms

Design, Experimentation, Performance

Keywords

Distributed Environments, Code Buffer, Dynamic Translation, Generational Cache, Adaptive Code Cache, Program Partitioning

VEE'05, June 11–12, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Over the last several years virtual execution environments (VEE) have been increasingly useful in system implementation. A VEE can reduce and manage complexity by providing a common runtime and a self-contained operating environment that facilitates the programmatic modification of an executing program. In addition to the well-known Java virtual machine (JVM), a wide variety of systems can also be classified as virtual execution environments, such as dynamic optimizers [2,5,6,17], dynamic software updaters [18], dynamic binary translators [8,9,11,27], dynamic instrumentation systems [3,19,23], and certain emulators and simulators [26,29].

Although a virtual machine (VM) can execute programs using interpretation, performance-critical VMs often employ software dynamic translation because a translator has the potential to produce significantly higher quality code and thus is able to utilize resources efficiently. Just-in-time (JIT) compilation, for instance, is used in many JVMs. After translating a code segment, a translation-based VEE typically stores the translated code in a code buffer (CB), and reuses the code for future invocations. The overhead of dynamic translation can be amortized if the translated code is reused frequently.

As VEE techniques have been applied to a range of computing environments, there is a set of environments in which devices have exceptionally tight memory and performance constraints, such as smart cards and sensors in distributed systems [4,10,15,22,25,28]. The software executing in such environments, however, has become quite complex. For example, a smart card might use the RSA protocol to authenticate a user's identity [25]. Furthermore, such constrained devices may need to support multitasking workloads. For instance, a sensor in an intrusion detection network concurrently monitors environmental events, tracks objects, and communicates with other sensors [1]. A consequence of this trend is that memory demands have become very high.

Due to severe memory limitations (e.g., an Atmel ATmega128 processor has 128 KB flash memory and 4 KB SRAM [4]), the original code of a single large program, or multiple small programs, may not fit in the memory of a resource-constrained device. To address such constraints, and inspired by program partitioning schemes for traditional systems [16,24,31,32], we propose to store the original software on a code server and to execute the VM using its code buffer on the device (as a client). A piece of original program code (e.g., trace, basic block sequence, method, program slices, etc.), called a *partition*, is downloaded from the code server to the client on-demand via a wireless link. The code executes on the client and a CB miss happens when a needed partition is not in the CB. Not only does this partition need

to be downloaded on-demand, it also needs to be re-translated by the VM before execution continues. As wireless bandwidth is limited in resource-constrained devices and dynamic translation is expensive, the CB has a high miss penalty, and it must be intelligently managed to keep miss rates low.

In dynamic optimization systems, a *generational buffer* is proposed [14], which tries to identify and use the lifetime of code to manage the code buffer. Two other similar techniques that have been proposed to manage the CB are *adaptive code unloading* [30] and *code collection* [24], both of which use online profiling to trigger a garbage collector. Another technique for embedded systems is *compiler-controlled function caching* [32]. Code replacement policies employed in hardware caches and operating systems, such as LRU, are related as well. However, these methods are unsuitable for distributed VEEs due to two reasons. First, although a technique might effectively reduce CB misses, its overhead can be too high to be practical (e.g., a VM must maintain usage information to employ online profiling). Second, most of these approaches cannot achieve a satisfactory CB miss rate when memory space is tightly limited, as these approaches lack the awareness of both program and memory size. To achieve an effective CB miss rate, a scheme needs to utilize knowledge of a program and its memory size, while having a low run-time cost.

In this paper we present a technique whereby the server "plans" for memory management using both the program and the memory size. Using program profiles helps the schemes maintain hot code partitions in the CB and thus avoid caching cold partitions. We move management decision-making to a server to keep the demands placed on the clients minimal. One scheme is used for programs that are insensitive to data inputs, having stable behavior across different data sets, while the other scheme adapts decisions for programs that are highly sensitive to inputs.

The contributions of this paper include:

- Planning for CB management before execution and moving decision-making from the client to a server;
- A simple yet effective planning management scheme for programs with stable behavior across different data inputs;
- An adaptive scheme for programs in which inputs can produce varying behaviors; and
- Experimental results that demonstrate the benefits of our schemes over previous approaches.

The remainder of the paper is organized as follows. Section 2 provides a background on resource-constrained systems, and Section 3 describes our planning schemes. Section 4 evaluates the schemes in terms of miss reduction and the impact on execution time. Section 5 surveys related work and Section 6 summarizes the paper.

2. BACKGROUND

Tight memory constraints and high CB miss penalties make good CB management decisions critical to performance. In general, with more knowledge about program behavior, better decisions can be made about buffer management. Program profiling can be used to identify likely execution paths. Indeed, it is known that most execution time is spent in a small portion (hot code) of a program, with recent studies showing similar results for *code traces* [5]. A trace is a sequence of basic blocks that are executed along a path. Hazelwood and Smith [13] showed that regardless of data inputs for SPEC2000 programs, code traces that account for roughly 85% of the dynamic instruction count are repeated during successive executions. Hence, basic block profiling works well in these programs and can identify sequences of hot code.

Although there are many programs in which profiling can capture execution stability, there are some programs with much variability across inputs. For instance, the program *blowfish* [21] behaves differently when encoding a plain-text file and an image file. Considering the importance of good CB decisions, a management scheme needs adaptivity for these unstable programs. The adaptivity can be achieved by gathering run-time information about hot basic blocks and paths.

When management decisions are made by the client, profiles can introduce considerable overhead. Offline profiles need to be accessible to the client and run-time information needs to be updated. Both are very expensive to use or maintain on the client. Indeed, it is typically infeasible for a severely restricted client to use profile information directly. Therefore, we propose to make a code server responsible for CB management, moving decision-making from a client to a more powerful server that can more easily maintain, update, and use profiles. Thus, the client only executes simple actions guided by the server, while the server manages the client's CB.

3. A PLANNING APPROACH

The key idea in our work is for a powerful platform (called a *cache planner*) to develop CB *cache plans* based on a program's

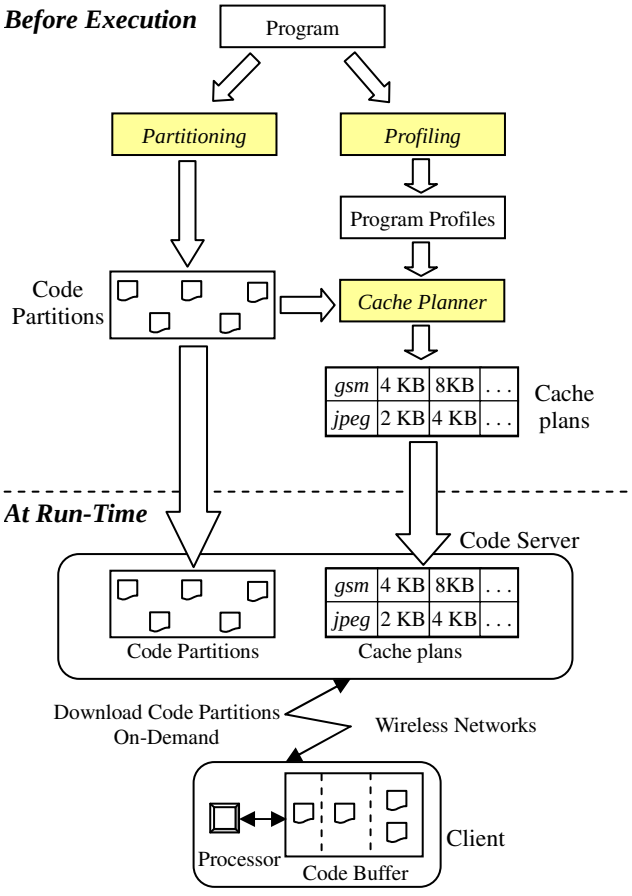

Figure 1. Workflow of our schemes.

code partitions and profiles, with the knowledge of a client's anticipated memory size. The plans are developed before execution of a program begins, with the goal of caching frequently executed (hot) partitions in the CB. The code server in charge of sending code partitions during the program's execution forwards these plans to the client as code is downloaded. Both of our schemes partition the CB into separate buffers, called *sub-buffers*, to hold code partitions based on code hotness.

Figure 1 shows the workflow of our management schemes. A program is firstly partitioned into code partitions using a partitioning scheme, and then stored in a code server connected to the client. Profiling is used to capture the hotness (execution frequency) of code partitions and to estimate the performance of potential CB management decisions. A cache planner uses program profiles and code partitions to develop management decisions (i.e., cache plans) before program execution. The plans are then stored on the code server with the corresponding code partitions. When a client needs a code partition, the server sends the appropriate plan with the partition. The sub-buffers are ordered by the hotness of partitions assigned to them. That is, one sub-buffer holds very hot code, while another may hold cold code that is executed infrequently. This approach is based on the fact that most programs spend a large part of their execution in a small portion of code.

To handle the sub-buffers, we use two policies. A *local policy* manages an individual sub-buffer and a *global policy* manages the relationship among sub-buffers. The local policy is essentially the replacement policy for the partitions in a sub-buffer. The replacement policy has to possess all of the following attributes: high temporal locality, low overhead, and minimum fragmentation. The first factor is the foremost motivation for CB. The second factor is important for application performance because it is a part of user perceivable system overhead. The third factor must be considered because our caching element, a code partition, has variable size, which easily causes fragmentation. We use First-In-First-Out (FIFO) as our local policy as it has good performance with little fragmentation [12]. Our global policy is to cache code partitions in sub-buffers based on their hotness.

We describe the overall strategy of CB memory planning and then describe two particular schemes. One scheme is a *fixed scheme* where code partitions are always housed in the same sub-buffer during execution. We then extend this scheme to an *adaptive one*, in which partitions are cached in sub-buffers based on a program's run-time behavior. We also describe a method, using "code density", which improves the code partition assignments made by either technique.

3.1 Overall CB Management Strategy

Our strategy generates plans for managing the CB, by using profiles to first determine the hotness of code partitions and then to assign partitions to sub-buffers. As management decisions are sensitive to program and memory size (in a small CB), our scheme generates a cache plan for each program and anticipated CB size. These plans are stored on a server and can be retrieved when a client connects with the server and provides a program name and the size of its CB.

A cache plan records management decisions: For a given program and CB size, a plan indicates the number of sub-buffers, the size of the sub-buffers, and a *cache blueprint* for each code partition in the program. The size of a sub-buffer is recorded as the percentage (proportion) of the total CB size to allocate to this sub-buffer. A cache blueprint indicates the assignment of a code partition to a sub-buffer and has two fields. One field indicates whether to cache the partition and the other indicates which sub-buffer to use if the partition is to be cached. An example of a cache plan is shown in Figure 2. There are *N* code partitions in the program, each of which has been assigned a cache blueprint. For example, partition 2 is cached in sub-buffer 0. In this figure, the CB is partitioned into three buffers, and the sub-buffer size proportions are 20%, 40%, and 40%, respectively.

Number of Sub-Buffers	3	
Sub-Buffer Allocation	20%-40%-40%	
Partition ID	**Cache Blueprint**	
	Cache?	**Sub-Buffer ID**
0	Yes	1
1	No	---
2	Yes	0
⋮	⋮	⋮
N-1	Yes	2

Figure 2. Example cache plan.

The cache plan, computed beforehand by the server, is used at the client side. Before a client executes a program, it informs the server which program is going to be run and the size of the CB. The server finds a corresponding plan and responds to the client with the number and size of the sub-buffers. The code partition containing the first instruction, and this partition's blueprint, are transferred to the client, and then the client starts program execution.

After translating a code segment (called a *translation unit*), a VEE directly executes the translated code for efficiency. Some VEEs add an instruction at the exit of each translation unit to return control to the VEE, while other VEEs directly link translated units to avoid unnecessary invocations of the VEE. No matter what mechanism is used, the VEE is notified when a desired unit is not found in the translated code buffer. Every time this occurs, our approach performs several more operations, compared to what a traditional VEE does, to use the cache plan to manage the CB.

At run-time, if a needed code partition is not cached in the CB, the client sends a request to the server for the partition. The server sends back the partition, with its cache blueprint attached. Note that a partition is both a translation unit and a caching unit. After the client receives the partition and cache blueprint, it translates the partition and follows the blueprint to cache the partition in the specified sub-buffer (or not to cache the partition). If the specified sub-buffer does not have enough free space to store it, other partition(s) in the specified sub-buffer are evicted using the local policy (FIFO).

To generate a cache plan, two steps are performed. The first step uses a given configuration as parameters and assigns code partitions to sub-buffers based on profile information about the execution frequency of partitions. In other words, this assignment determines the cache blueprint for each code partition. The configuration includes the number of sub-buffers, each sub-buffer's size proportion and the *assignment ratio*. Assignment ratio is the ratio of the total size of all partitions assigned to a sub-buffer over the sub-buffer size, which limits the total size of partitions that can be assigned to a sub-buffer.

How well a cache plan works is influenced by the number of sub-buffers, the size proportion of each sub-buffer, and the assignment ratio of each sub-buffer. Because the quality of a cache plan depends on these three factors, we produce a number of candidate plans. The best plan is selected among these candidates to be loaded onto the code server.

The second step selects the plan that is most likely to minimize the number of CB misses at run-time. This step iterates over all cache plans to determine a score for each one that indicates how well they may perform. The score is determined by running the application program with a training data set and collecting the number of CB misses. The cache plan with the smallest number of misses is the one that is selected.

Figure 3 shows pseudo-code to generate a cache plan. Line 2 determines a basic block execution frequency profile. Lines 4-8 generate and try a range of configurations to produce candidate plans. Line 10 does the second step that determines the best plan among the candidates. The routine `Assign_Blueprints()` (line 6) generates the blueprints for each cache configuration. A naïve algorithm to implement this function is shown in Figure 4.

```
1  Generate_Plan (program x) {
2    prof = First_Profile (x);
3
4    repeat
5      config = Generate_New_Config();
6      blueprints = Assign_Blueprints(x, prof, config);
7      candidate_plans.Add(config, blueprints);
8    until no more configurations;
9
10   plan = Second_Profile(x, candidate_plans);
11   return plan;
12 }
```

Figure 3. Cache plan generation algorithm.

```
1  Assign_Blueprints (program x, profile prof,
2                     configuration config) {
3    (buffer_number,buffer_portion[],assign_ratio[])
4       = Extract_Config_Values (config);
5    sort_parition[] = Sort_By_Freq(x, prof);
6
7    for (i=0; i < buffer_number; i++) {
8      buffer_size=CB_size×buffer_portion[i];
9      remain_size=buffer_size×assign_ratio[i];
10     flag = 1;
11
12     while (flag) {
13       sizeLimit = min(buffer_size, remain_size);
14       cand = Max_Freq_P(sizeLimit);
15
16       if cand not existing {
17         flag = 0; /* all partitions already tried */
18       } else {
19         assign cand to sub-buffer[i];
20         remain_size -= size(cand);
21         if (remain_size = 0) { frag = 0; }
22       } /* of else */
23     } /* of while */
24   } /* of for */
25
26   for each remaining partition p in sort_partition[]
27     { sort_partition[p].blueprint = No_Caching; }
28 }
```

Figure 4. Cache blueprint assignment algorithm using frequency as criterion.

In Figure 4, `Assign_Blueprints()` sorts code partitions based on their execution frequency (from a profile) in descending order (line 5). Lines 7-24 divide the CB into sub-buffers, give each sub-buffer a unique identifier (ID), and assign partitions to each sub-buffer. Line 8 calculates a sub-buffer's size and line 9 calculates the total size of partitions that can be assigned to this sub-buffer. Line 10 initializes a flag variable. Lines 12-23 determine blueprints, assigning hotter partitions to the sub-buffer with a smaller ID. Line 13 sets the size limit which is an upper bound on the partitions to be assigned to this sub-buffer. Line 14 seeks a candidate partition with maximal frequency (which has not been yet assigned to any sub-buffer) and not larger than the size limit. If no such partition exists, the assignment for this sub-buffer is done (line 17). Otherwise we assign the candidate partition to the sub-buffer (line 19-21). If there are partitions left unassigned after every sub-buffer has been processed, these partitions are marked `No_Caching` on lines 26-27. This mark indicates that these partitions will never be cached (they are too cold). The algorithm does not consider code partitions that remain unexecuted by the training input. These partitions are assigned to the sub-buffer with the largest ID (it holds the coldest code).

The planning approach is quite efficient. Clients only execute simple actions as directed by the server; hence, their run-time overhead is low. However, transferring the blueprint with a code partition does introduce a small additional amount of communication. If one byte is used to encode the cache blueprint and a code partition itself is 20 bytes, the transfer overhead of the blueprint is just 5%. Larger code partitions reduce the overhead further.

3.2 Fixed Scheme

In the fixed strategy, a code partition is always stored in the sub-buffer that it was assigned in the original plan. That is, the hotness of the code during execution is assumed to mirror the profile information.

3.3 Adaptive Scheme

The fixed scheme relies on the accuracy of profiles to guide the selection of cache plans. However, some programs (as described earlier) may have behavior that is not captured by a profile. Our adaptive scheme aims to overcome this problem by changing the assignment of code partitions to sub-buffers as a program executes. As before, the server is responsible for managing the partitions, but the adaptive scheme can change the assignment of a code partition to a sub-buffer based on its hotness at run-time. The scheme moves a code partition from one sub-buffer to the next in sequence. This process is called *promotion*. At run-time, the server maintains a time window (called a *miss window*) to monitor missing partitions. The server uses the miss window to decide which partitions should be promoted. Whenever a partition is promoted, its new sub-buffer position is recorded in a *registration list*. Before the server sends a partition to the client, it checks the registration list. If a partition is found in the list, a temporary blueprint is created on-the-fly that designates a different sub-buffer to hold the partition (i.e., the sub-buffer holding the next hottest code). Otherwise, the original blueprint is used.

Figure 5 shows pseudo-code for the server algorithm that decides which partitions to promote. Every time the server receives a request from the client, `monitor_miss()` is invoked.

```
1  index = 0;
2  monitor_miss (p_id) {
3    miss_win[index]=p_id;
4    if (index < win_size-1) {
5      index++;
6    } else { /* check miss window and promote if necessary */
7      clear registration_list[];
8      for each unique partition p in miss_win[] {
9        freq = frequency of p in miss_win[];
10       if (freq > promote_threshold)
11         add p to registration_list[];
12     }
13     clear miss_win[];
14     index = 0;
15   }
16 }
```

Figure 5. Partition promotion algorithm.

Line 3 records the missing partition's ID (`p_id`) in the miss window (`miss_win[]`). The window size is a pre-defined threshold (`win_size`). When the window is full, lines 6-15 check for promotion. (We skip line 7 for a moment — it will be discussed shortly.) Lines 8-9 scan every partition in the window to determine how often they occurred. Line 10 checks a promotion condition: any partition that occurs more frequently than a threshold (`promote_threshold`) is promoted by adding the partition to the registration list. Lines 13 and 14 flush the miss window for the next interval of execution.

In our experiments, we observed that input variability is limited in applications for distributed environments. If a cold partition in a profile is hot in actual runs, the partition seldom becomes hot throughout the whole program execution. Thus, the adaptive scheme has to also let promoted partitions cool and move back to their original sub-buffer. On line 7, the registration list is cleared immediately before the miss window is checked. In this way, any promotion is conservative as it is only visible until the miss window becomes full again. Furthermore, we have observed that it is necessary to promote a partition by only one level to avoid disturbing the original cache plan too much.

The server maintains a private miss window and a private registration list for each execution. Consequently, executions of the same program at different clients can be adapted individually and concurrently. Repeated adaptations for a program may indicate that the training input used in profiling is not representative, and a single update of the cache plan can save the multiple adaptations. However, we leave this as a question for future studies.

In the adaptive scheme, no special operation is needed at the client side; it is the code server that adapts. Hence, this scheme is as efficient as the fixed scheme for a client. The adaptive scheme does introduce some overhead in the server, but it is minimal.

3.4 Density: A Heuristic Algorithm

In Figure 4, we presented a naïve algorithm to assign blueprints by using the frequency of code partitions. It looks quite straightforward; however, the problem is complex. The naïve algorithm may favor some extremely large partitions with high frequency, overlooking a set of small partitions which together have a higher total frequency. CB management is a trade-off between code usage and code size. Rather than solely focusing on an individual partition's hotness, our goal is to find a set of partitions that has an upper bound on combined size and possesses the maximal total frequency simultaneously. Unfortunately, this is the knapsack problem and is NP-complete.

To tackle the problem, we introduce a new concept, called *density*, which is a criterion to measure the priority of code partitions to reside in CB. A partition's density is defined as a partition's execution frequency divided by its size.

Density = Execution Frequency / Size

We employ density to avoid caching extremely large partitions with slightly high frequency. However, relying exclusively on density may lead to another problem. Assigning a small, dense partition first may make it impossible to assign a large, hot partition with less density later. To avoid both pitfalls, we designed a heuristic algorithm to assign cache blueprints, as shown in Figure 6. We use density to find a candidate partition first, and then check its frequency to make sure that it will not prevent us from caching a hotter fragment with a lower density later on.

```
1  Assign_Blueprints (program x, profile prof,
2                     configuration config) {
3    (buffer_number,buffer_portion[],assign_ratio[])
4       = Extract_Config_Values (config);
5    sort_parition[] = Sort_By_Density(x, prof);
6
7    for (i=0; i < buffer_number; i++) {
8      buffer_size=CB_size×buffer_portion[i];
9      remain_size=buffer_size×assign_ratio[i];
10     from = 0; flag = 1;
11
12     while (flag) {
13       sizeLimit = min(buffer_size, remain_size);
14       cand = Max_Density_P(from, sizeLimit);
15
16       if cand not existing {
17         flag = 0; /* all partitions already tried */
18       } else {
19         threat = Max_Freq_New_P(sizeLimit);
20
21         if (cand == threat ||
22             (remain_size>size(cand)+size(threat)){
23           assign cand to sub-buffer[i];
24           remain_size -= size(cand);
25           if (remain_size = 0) { frag = 0; }
26         }
27         from ++;
28       } /* of else */
29     } /* of while */
30   } /* of for */
31
32   for each remaining partition p in sort_partition[]
33     { sort_partition[p].blueprint = No_Caching; }
34 }
```

Figure 6. Cache blueprint assignment algorithm using density as criterion.

The algorithm's overall structure is similar to the naïve algorithm described in Figure 4; hence we emphasize only the differences. Line 5 sorts the code partitions based on their density. Lines 10-30 assign the blueprints. Line 10 initializes the search start position `from` and a flag variable. Line 13 sets the size limit which is an upper bound on the partitions to be assigned to this sub-buffer. Line 14 seeks a candidate partition (not assigned yet) with maximal density and not larger than the size limit, starting from position `from` in the sorted partition list. If no such partition

exists, the assignment for this sub-buffer is done (line 17). Otherwise, we check if the candidate will restrict the assignment of hotter partitions. Line 19 looks for a partition with maximal frequency that is not larger than the size limit. If the candidate is also the one with the maximal frequency (line 21), or does not restrict the assignment of hotter partitions (line 22), it will be assigned to this sub-buffer (line 23-25). Line 27 updates the search start position. This is a greedy algorithm, and one can easily compose an example to show that it is not optimal. Our experiments in Section 4, however, show that it is able to produce satisfactory results in practice.

4. EVALUATION

We simulated our schemes to determine their effectiveness and to compare their performance with other approaches as well as with each other.

4.1 Experimental Methodology

We experimented with twelve MiBench [21] and MediaBench [20] programs on a SPARC/Solaris 9 workstation, using *gcc* with the compiler flags "*–O3 –static*". We believe our selection of benchmarks represents the applications which will be extensively used in the next generation of VEEs for smart cards and sensor networks (including ones performing biometric recognition and those used in ad-hoc networks).

The results are collected by using a profiler and a simulator. The profiler executes benchmarks and collects a log of partition accesses. Our simulator uses the CB size, the cache plan, the access log, and the size of each partition as inputs. It faithfully mimics the operations of buffer management and produces CB miss numbers as an output. Although simulation sometimes provides inaccurate or incomplete results when compared to actual execution, our simulators are trustworthy. Since all factors that affect a buffer's hit and miss action are considered in the simulator, the simulation result (miss numbers) will be consistent with those arising in actual execution.

We used a *fragment* (an instruction sequence that ends with a conditional branch, indirect branch, or return) as a partition in our experiments. A fragment is similar to a basic block, except that a basic block terminates at a branch target while a fragment does not. Our profiler was built on a software dynamic translator, Strata [27], which implements a virtual execution environment and uses a code fragment as its translation unit. Typically, a translation unit in a VEE is also a caching unit in the CB. Therefore, we use fragments as partitions in our experiments. Although the definition of the translation unit (i.e., code partition in this work) changes across VEE implementations and the variance may influence caching performance, these different partitions possess certain common properties. In particular, among the factors affecting CB management, variable size and hotness are universal. Therefore, our experimental results can demonstrate the benefits of our approaches in general VEEs. Although we did not experiment with other partitioning schemes, we believe that the qualitative trend will be similar.

For each benchmark, we use a training data input for profiling and a different reference input for evaluation. Table 1 shows the miss numbers of the reference inputs when running with our *baseline*, which is a unified circular buffer using a FIFO policy. Column 1 lists the benchmarks. Column 2 lists the number of unique partitions that have been downloaded in the execution, and the remaining columns list the number of misses when the CB size is 2 KB, 4 KB, 8 KB, and 16 KB. If the miss number is the same as the partition number (designated by "*" in the table), it means all misses are compulsory and the baseline is optimal.

Table 1. Miss numbers of the baseline. "*" designates that all misses are compulsory.

Benchmark	Partition Number	2 KB	4KB	8KB	16KB
blowfish_dec	241	1881118	344	241 *	241 *
blowfish_enc	239	1881116	342	239 *	239 *
crc32	283	317	283 *	283 *	283 *
dijkstra	397	21466	16013	397 *	397 *
gsm_dec	690	351957	350246	350152	691
gsm_enc	908	1211052	1184794	646302	638472
jpeg_dec	1149	38049	5298	1619	1259
jpeg_enc	1403	62031	7168	2326	1594
patricia	792	3054768	2859748	2755161	3921
susan_corner	541	51001	727	664	541 *
susan_edge	564	157639	778	722	717
susan_smooth	445	770	667	598	445 *

The technique most similar to our work is the generational buffer [14], which we compare our schemes against. The generational buffer scheme was proposed to manage the trace cache in dynamic optimization systems. It partitions the trace cache into three distinct and separately managed regions, trying to identify code lifetime at run-time. It uses a unified partitioning proportion for all programs and CB sizes, and thus no profiling is needed. Hazelwood and Smith showed that the generational buffer can effectively reduce the miss rate for SPEC2000 and Windows applications. However, we found that the partition proportion suggested in [14] (a 45%-10%-45% ratio for three separate buffer sizes and a promotion threshold of 1) works poorly for resource-constrained devices in distributed VEEs.

Since the programs and environments studied in [14] are significantly different from ours, we extend the generational scheme for a fair comparison, using the training inputs (also used in our schemes) to find the best configuration as well. We investigate two ways to find the best configuration. First, for each prospective CB size, we select the configuration with which the generational buffer produces the minimal geometric mean of normalized miss numbers (to the baseline) for training inputs across all benchmarks. We call this configuration *unified* and use it for all programs when that particular CB size is experimented. Second, for each combination of prospective CB size and program, we select the configuration with which the generational buffer produces the minimal miss number for the training input. We call this approach *individualized* and use it for the particular combination of program and CB size.

We compare the miss reduction of the unified generational, individualized generational, fixed, and adaptive schemes to the baseline. When we calculate miss reduction, the size of the unified buffer in the baseline equals the total size of all sub-buffers in the generational and planning schemes.

In the fixed and adaptive approaches, we limit the maximal

number of sub-buffers to three (3) because our experience shows that three sub-buffers are enough to classify the code partitions in most cases. In addition, finer partitioning of the CB introduces more fragmentation, which is expensive for resource-constrained devices.

4.2 Miss Number Reduction

Figure 7 compares the miss reduction of the generational and fixed schemes over the baseline for various CB sizes. For the generational schemes, our experiments include both the unified and individualized approaches. For the fixed scheme, both the naïve algorithm (Figure 4) and the improved density algorithm (Figure 6) are presented. Each chart shows the percentage of miss reduction when the CB size is 2 KB, 4 KB, 8 KB, and 16 KB. If a scheme produces the same number of misses as the baseline, the miss reduction is zero (0).

The most consistent result is that the individualized generational scheme performs better than (or as well as) the unified generational one for every benchmark with each CB size. Considering the performance of the generational buffer in DynamoRio, this indicates that a tightly constrained CB is quite different from the buffers in dynamic optimizers: CB performance becomes very sensitive to the program when the CB is small. Hence, a single configuration for all programs does not produce good results.

Although the individualized generational scheme gains through customized configuration by using profiling, the fixed scheme exploits more benefits from profiling information. As shown, in most cases, the fixed schemes perform better than the generational buffer. It may be that a small generational buffer is unable to capture the hottest code partitions accurately, since many partitions seem to be missing frequently. Another possible reason is the fragmentation caused by unnecessary partitioning of the CB. Indeed, the best cache plans selected by the fixed scheme (using density) divide the 2 KB CB into three sub-buffers for only 3 programs out of the 12 benchmarks, while the generational scheme inherently does such partitioning for all programs.

As shown, the density algorithm can generate higher quality cache plans than the naïve algorithm, especially when the CB size is small. The argument to use density as a planning criterion in constrained environment is thus well justified, as a tighter environment demands a more careful trade-off decision between the code hotness and size.

Not surprisingly, some programs are not friendly to the fixed scheme, because program behavior can not always be predicted in advance. An obvious example is *patricia* with a 16 KB CB in Figure 7. Here, the fixed scheme produces 15 times more misses than the baseline. A careful examination found that a code region is executed once for the training data set, but its execution frequency is over 33,000 for the reference input. The adaptive scheme is able to correct this problem, as shown in Figure 8. We only present the result for density because we know that density is better able to determine cache plans than frequency. The results for the adaptive scheme are collected when `win_size` = 50 and `promote_threshold` = 20, which is the best configuration

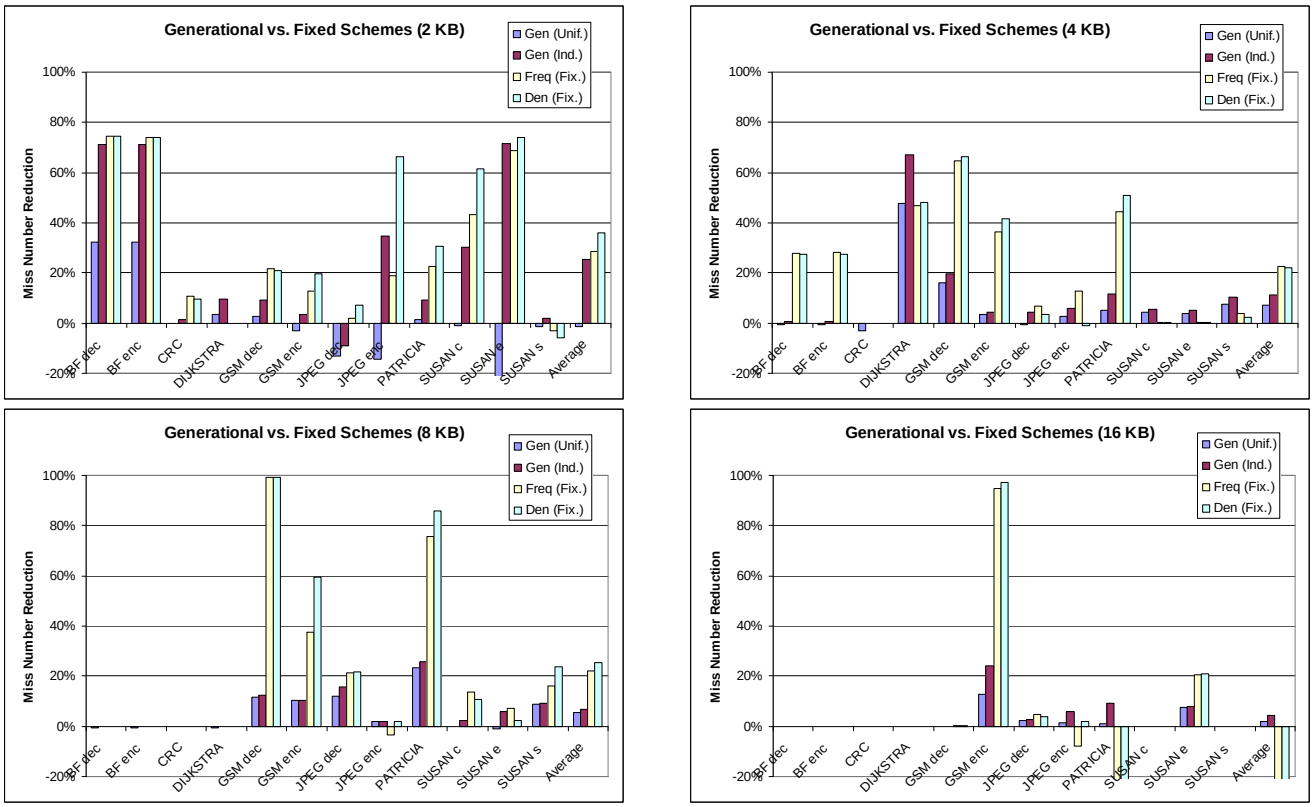

Figure 7. Comparison of miss reduction over the baseline. "Gen (Unif.)" stands for unified generational scheme; "Gen (Ind.)" stands for individualized generational scheme; "Freq (Fix.)" stands for fixed scheme using frequency in planning; and "Den (Fix.)" stands for fixed scheme using density in planning.

that we found in our experiments. Figure 8 shows that the adaptive approach can dramatically reduce the misses more than the fixed scheme for certain benchmarks at some CB sizes. In other cases, it does as well or is very close to the fixed scheme.

4.3 Impact on Execution Time

A common pitfall in cache performance evaluation is the overemphasis on miss (rate) reduction while omitting the fact that a significant miss reduction sometimes affects little in the overall optimization objective. In our work, nevertheless, CB miss reduction can translate into a meaningful performance improvement. In VEEs for resource constrained devices, the time for downloading a partition often dominates total execution time, as wireless bandwidth can be severely limited.

Figure 9 shows the impact of the miss reduction on the execution time. The speedup is calculated by a method which has been used in [31]. The calculation of total execution time is divided into four parts: the *download time* for partition transmissions, the *connection setup time* due to network delays, the *runtime environment time*, and the *real CPU time* for the program. Similar to the experimental settings used in [31], we assume a bandwidth of 106 kb/sec, a connection time of 20 ms per setup, a context switch (between the application and the VEE) consuming 100 dynamic instructions, and an internal clock frequency of 66 MHz. When we calculate the time for the fixed and adaptive schemes, we add a byte, which records a cache blueprint, to the size of each transferred partition. The additional overhead of cache blueprint is amortized by greater miss reduction. As demonstrated in the figure, the planning schemes have a greater speedup than the generational schemes in most cases. As these results show, it is important to reduce the CB miss rate and small reductions can lead to large run-time improvements.

From the results, our schemes have a significant improvement over a unified circular buffer and a generational buffer in terms of miss reduction, which translates into considerable performance speedup. In particular, the adaptive scheme does as well as the fixed scheme in most cases, and in a few cases does substantially better. Yet, it has low overhead. We conclude that the adaptive scheme is a good choice for resource-constrained devices in distributed VEEs.

5. RELATED WORK

DELI [9], an infrastructure for manipulating or monitoring running programs, provides a mode (called code streaming) for remote execution, where a remote application is loaded on-demand piece by piece. The authors discussed the method when the emulated ISA and the target ISA are the same, while we extended the idea to general VEEs, even if the original and target ISAs are different.

There are several buffer management approaches proposed for dynamic translation/optimization systems, including: flushing the

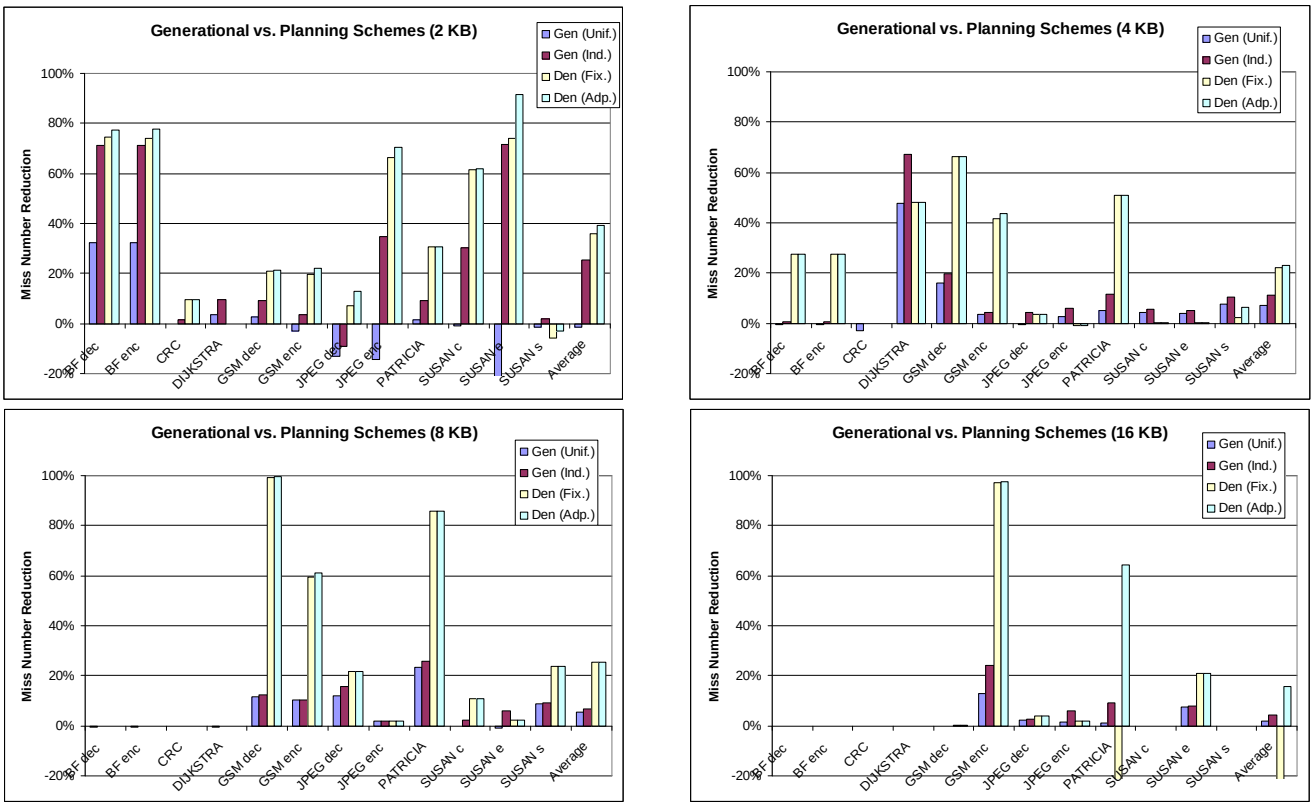

Figure 8. Comparison of miss reduction over the baseline. "Gen (Unif.)" stands for unified generational scheme; "Gen (Ind.)" stands for individualized generational scheme; "Den (Fix.)" stands for fixed scheme using density in planning; and "Den (Adp.)" stands for adaptive scheme using density in planning.

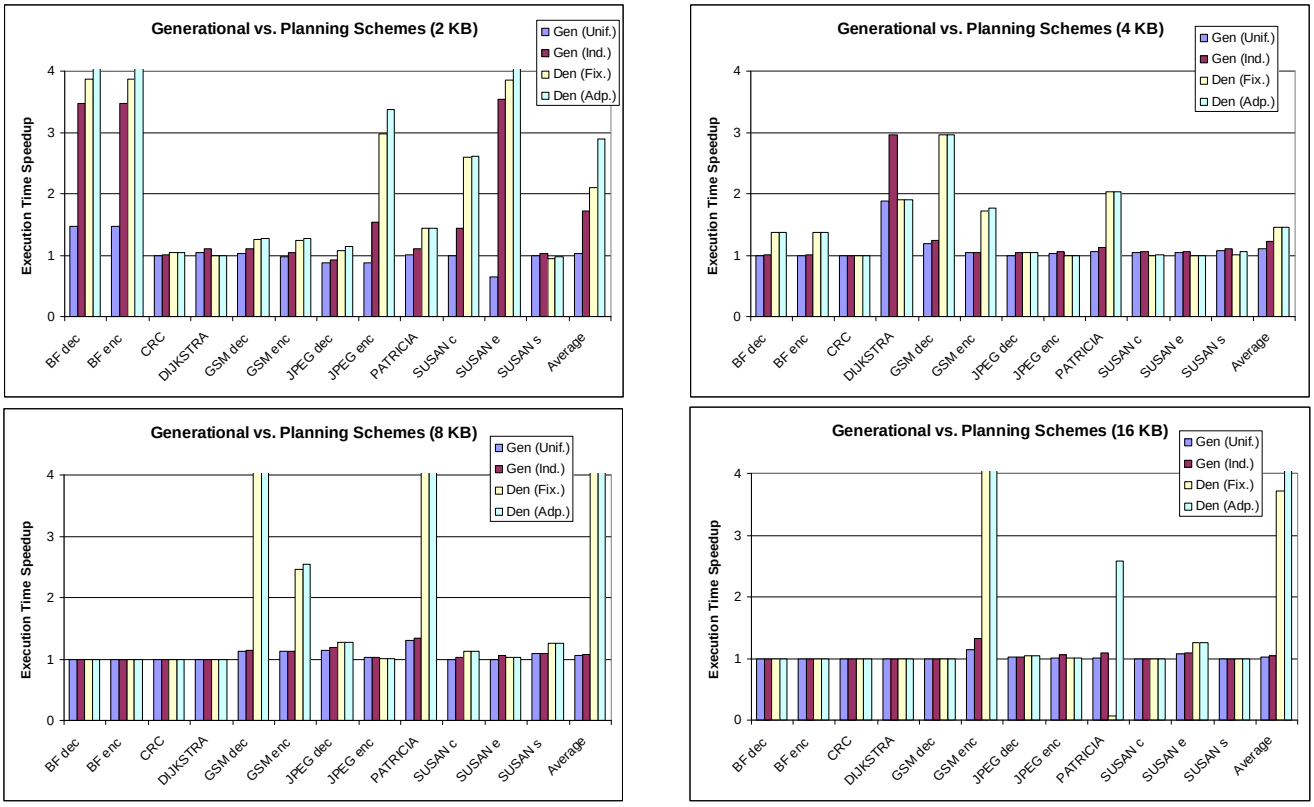

Figure 9. Execution time speedup over the baseline. "Gen (Unif.)" stands for unified generational scheme; "Gen (Ind.)" stands for individualized generational scheme; "Den (Fix.)" stands for fixed scheme using density in planning; and "Den (Adp.)" stands for adaptive scheme using density in planning.

buffer when it gets full (Dynamo [5] and DELI [9]), an unbounded buffer (Strata [27] and DynamoRio [6] by default), and a circular buffer. The generational buffer was proposed to overcome the limitations of these techniques [14]. The results presented in Section 4 show that these techniques are unsuitable for distributed systems with tight resource constraints. Zhang and Krintz proposed an adaptive code unloading method for JVMs with JIT compilation [30]. Their cached element is method based, which may include cold code. And the fragmentation is handled by the system's garbage collector, which is too expensive to execute on a constrained device. They mentioned a very simple offline profiling method in their paper but its result was disappointing. Popa et al. proposed a mechanism called code collection to support large applications on mobile devices [24], which is similar to our approaches. But their heuristic algorithm necessitates a client to collect run-time information for selecting code units to discard, while ours does not. And their scheme is also method-based and requires a garbage collector running on the client. Zhang et al. proposed a buffer management technique, function caching, for smart cards [32]. Their approach is not adaptive as the compiler fully controls the management, and their caching elements are functions which may contain unneeded code. Another undesirable side is that they only applied the technique in user defined calls where source code is available.

Another related research is profile guided code compression [7], which compresses cold code to achieve size reduction, and leaves hot code uncompressed to minimize run-time penalty. The number of their code categories is fixed to 2 (hot and cold), and the classification can be adaptive to neither a particular input nor available memory size.

Regarding the security concerns, our approaches can easily incorporate the tamper-resistant partitioning method [32], and transferred code could be encrypted.

Hazelwood and Smith have found that pure FIFO policy is not enough for real world applications due to some complications, such as undeletable cache code and program-forced evictions. Pseudo-circular policy, a variant of FIFO, can be used in this situation [14]. Our scheme is compatible to use the pseudo-circular policy as the local policy.

6. CONCLUSION

As the memory requirements of distributed VEEs is growing, we proposed to store the original programs on a code server, and to execute a VEE with its code buffer on the client. The original program is divided into code partitions and partitions are downloaded on-demand. This paper described two schemes to manage the CB with profile guidance. From experimental results, we showed that these schemes have fewer CB misses than a generational buffer and a unified circular buffer, which translates into significant speedup. In particular, the adaptive scheme performed better than the fixed scheme. Yet, it has low overhead and it is a good choice for resource-constrained devices in distributed VEEs.

7. ACKNOWLEDGEMENTS

This research is supported in part by the National Science Foundation, under grant CNS-0305198. We thank Naveen Kumar for his considerable contribution to the preliminary work [33] of this paper. We also thank the anonymous reviewers for their useful suggestions and comments on how to improve the paper.

8. REFERENCES

[1] T. Abdelzaher et al. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. *IEEE Intl. Conf. on Distributed Computing Systems*. March 2004.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño. *Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*. October 2000.

[3] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. *Conf. on Programming Language Design and Implementation (PLDI'01)*. June 2001.

[4] Atmel's ATmega128 Processor Online Document. *http://www.atmel.com/dyn/resources/prod_documents/2467S.pdf*. November 2004.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *Conf. on Programming Language Design and Implementation (PLDI)*. June 2000.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. *Intl. Symp. on Code Generation and Optimization (CGO'03)*. March 2003.

[7] S. Debray and W. Evans. Profile-Guided Code Compression. *Conf. on Programming Language Design and Implementation (PLDI)*. June 2002.

[8] J. C. Dehnert et al. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. *Intl. Symp. on Code Generation and Optimization (CGO'03)*. March 2003

[9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi and J. A. Fiser. DELI: A New Run-Time Control Point. *Proceedings of the 35th Annual Intl. Symp.on Microarchitecture (MICRO-35)*. November 2002.

[10] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card Operating Systems: Past, Present and Future. *Fifth USENIX/NordU Conference*. February 2003.

[11] K. Ebcioğlu and E. R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. *Intl. Symp. on Computer Architecture (ISCA'97)*. June 1997.

[12] K. Hazelwood and M. D. Smith. Code Cache Management Schemes for Dynamic Optimizers. *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*. February 2002.

[13] K. Hazelwood and M. D. Smith. Characterizing Inter-Execution and Inter-Application Optimization Persistence. *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques*. June 2003.

[14] K. Hazelwood and M. D. Smith. Generational Cache Management of Code Traces in Dynamic Optimization Systems. *Proceedings of the 36th Annual Intl. Symp. on Microarchitecture (MICRO-36)*. December 2003.

[15] Infineon's SLE 88CFX4002P Smart Card Document. *http://www.infineon.com/cmc_upload/documents/098/829/SPI_SLE88CFX4002P0104.pdf*. January 2004.

[16] G. Kortuem, S. Fickas, and Z. Segall. On-Demand Delivery of Software in Mobile Environments. *Nomadic Computing Workshop*. April 1997.

[17] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Intl. Symp. on Code Generation and Optimization (CGO'04)*. March 2004.

[18] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance. *Proceedings of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. October 2004.

[19] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. *Compendium of Workshops and Tutorials Held in conjunction with Intl. Conf. on Parallel Architectures and Compilation Techniques*. September 2002.

[20] MediaBench. *http://cares.icsl.ucla.edu/MediaBench*.

[21] MiBench. *http://www.eecs.umich.edu/mibench*.

[22] Doug Palmer. A Virtual Machine Generator for Heterogeneous Smart Spaces. *USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*. May 2004.

[23] Pin Website. http://rogue.colorado.edu/Pin/.

[24] L. Popa, C. Raiciu, R. Teodorescu, I. Athanasiu, and R. Pandey. Using Code Collection to Support Large Applications on Mobile Devices. *Proceedings of the 10th Annual Intl. Conf. on Mobile Computing and Networking (Mobicom'04)*. September 2004.

[25] RSA SecurID 5100 Smart Card Online Document. *http://www.rsasecurity.com/node.asp?id=1215*. June 2004.

[26] E. Schnarr, M. Hill, and J. Larus. Facile: A Language and Compiler For High-Performance Processor Simulators. *Conf. on Programming Language Design and Implementation (PLDI'01)*. June 2001.

[27] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and Retargetable Software Dynamic Translation. *Intl. Symp. on Code Generation and Optimization (CGO'03)*. March 2003.

[28] F. Vacherand. New Emerging Technologies for Secure Chips and Smart Cards. *The 3rd Intl. Micro and Nanotechnology Meeting (MINATEC)*. September 2003.

[29] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*. May 1996.

[30] L. Zhang and C. Krintz. Adaptive Code Unloading for Resoruce-Constrained JVMs. *Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*. June 2004.

[31] T. Zhang, S. Pande, A. Santos, and F. J. Bruecklmayr. Leakage-Proof Program Partitioning. *Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES'02)*. October 2002.

[32] T. Zhang, S. Pande, and A. Valverde. Tamper-Resistant Whole Program Partitioning. *Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. June 2003.

[33] S. Zhou, B. R. Childers, N. Kumar. Profile Guided Management of Code Partitions for Embedded Systems. *Conf. on Design, Automation and Test in Europe (DATE'04)*. February 2004.