Memory Optimization of Dynamic Binary Translators for Embedded Platforms

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Engineering

by

Apala Guha

August 2010

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Engineering

_____

Apala Guha

Approved:

_____	_____
Mary Lou Soffa (Advisor)	Jack Davidson (Chair)
_____	_____
Kim Hazelwood (Advisor)	Kamin Whitehouse
_____	_____
Evelyn Duesterwald	Joanne Bechta Dugan

Accepted by the School of Engineering and Applied Science:

_____

James Aylor (Dean)

August 2010

Abstract

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

We mitigate memory demand by identifying path selection (trace selection and linking) as a DBT design aspect that influences both the relative and absolute sizes of the different sources of memory demand. We explore a comprehensive set of path selection strategies to propose one that addresses both memory efficiency and performance. For a given path selection, we identified auxiliary code as offering many opportunities for further optimizing the memory demand. We designed approaches to reduce the size of individual auxiliary code blocks and the count of auxiliary code blocks. These optimizations reduce memory demand as well as improve performance by reaching the memory limit less often.

While reaching the memory limit less often reduces the time spent flushing, retranslation over-

head still remains. To address the retranslation overhead, we designed two approaches for selective flushing. First, we used a generational code cache which divides the code cache into less persistent and more persistent areas. We designed time-based and execution count based heuristics to classify traces into the different generations. We also designed a unified cache flushing strategy that is applicable to both single-threaded and multi-threaded guest applications. For unified cache flushing, we developed a pseudo LRU heuristic to determine which traces to preserve across flushes.

Finally, we combine the strategies of path selection, auxiliary code optimization and cache flushing into a comprehensive system. We identify the conflicts that arise in combining the different strategies and modify our designs to handle the conflicts. We evaluate each strategy as well as the combined system on embedded platforms to demonstrate the individual as well as combined merits of the strategies developed.

Acknowledgments

I would like to thank my advisors, Mary Lou Soffa and Kim Hazelwood for the seminal roles they played in my PhD. Apart from their excellent advising skills, they will always be special to me for the cheer and gaiety they can create. They really helped keep my spirits high during my entire graduate study. I would also like to thank the rest of my committee, consisting of Jack Davidson, Joanne Bechta Dugan, Evelyn Duesterwald and Kamin Whitehouse.

I thank my husband, Sandip for constantly assuring me that I am not the only one going through graduate school problems and exerting his calming influence upon me. I also thank my friends Vijay and Saayan for hearing out my frequent and long ramblings on graduate school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional compilation techniques bind application binaries to specific execution environments (architectures and operating systems). However, execution environments change continuously. For example, architectures are phased out and new architectures are defined. For a given architecture, the microarchitecture implementation continuously evolves. Even the operating system interfaces are modified and extended. Such changes in execution environments can render application binaries ineffective or obsolete. Even binaries that continue to be compatible may not leverage all the advantages of newer execution environments. Traditional compilation has other disadvantages too, for example, when additional features are desired from the execution of the binary. For example, users may desire secure execution of untrusted binaries or developers may want to instrument code to check for race conditions.

These problems can be addressed by 1) rewriting source code to support newer functionality and recompiling and 2) recompiling binaries for newer execution environments. However, rewriting and recompiling the large body of existing software not only requires much effort, it may not even be an option because the source code is not available in many cases. Automatic translation of binaries can solve this problem. However, performing binary translation prior to execution may not always be able to discover all code (as code is interspersed with data). Static binary translation has no information about which program paths will actually execute and must be conservative. Dynamic binary translation overcomes these challenges facing static translation and has emerged as a popular execution paradigm.

Dynamic binary translators (DBTs) provide an infrastructure to continuously monitor and translate the guest application instruction stream during execution. Therefore, DBT-based tools can dynamically translate from one architecture to another to provide compatibility [41]. DBT-based tools can also provide services such as secure execution [58, 62], instrumentation [69] and path-specific optimizations [9, 61] . While these are examples of services that are important across all platforms, some services are particularly important in the embedded context. For example, tools that can exploit system calls to configure hardware according to the power needs of the guest application are very important in battery-powered embedded environments [61]. Tools that can manage scratchpad memory [75] in embedded systems to aid static compilers can also be supported by DBTs.

1.1 Problems Facing DBTs for Embedded Systems

The DBT forms a software abstraction layer between the guest application and the host machine. The core of a DBT is a translator which fetches code from the guest application, translates it according to the service being provided by the DBT and caches the translated code in a software code cache (for reuse), from where it executes directly on the host machine. Figure 1.1 shows that the code cache contains translated code and auxiliary code. Auxiliary code is needed because the translator generates code regions on demand. Branches off such code regions are handled by the translator so that it does not lose control of the guest application. These branches are directed to auxiliary code that act as trampolines between translated code and the translator. DBTs maintain data structures to support the code cache. The data structures are used to facilitate reuse by locating already existing translations. A *code cache directory* which is a table containing an entry for each code region, is maintained. It maps the original program addresses of translated code regions to code cache addresses. Also, cached code regions are gradually *linked* by the translator to avoid transferring control to the translator each time a branch off a code region executes. The code cache directory also stores data structures to record links between code regions because these links may need to be removed if the target code region is ever evicted from the code cache.

It has been found [54, 59] that DBT memory demand can be 5-10 times that of the native

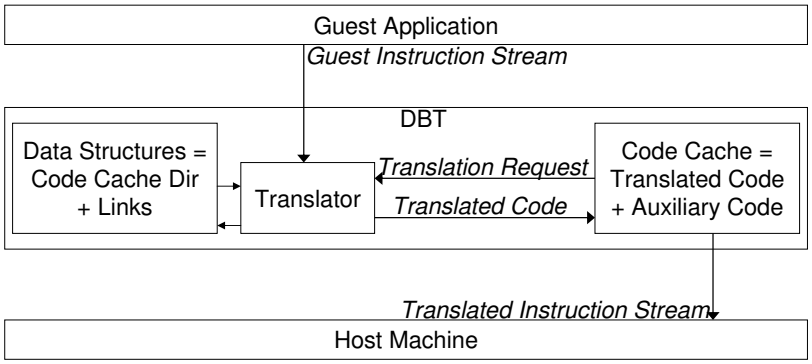


Figure 1.1: Schematic view of a dynamic binary translator. The code cache consisting of translated code and auxiliary code and data structures are the sources of DBT memory demand

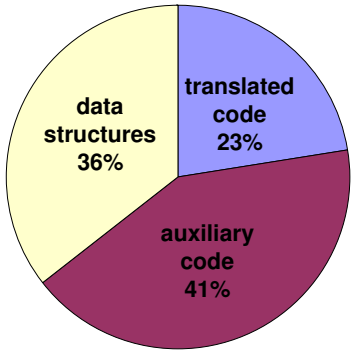


Figure 1.2: Memory distribution among translated code, auxiliary code, and data structures. The results are averages taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin on ARM [69].

instruction footprint of the guest application. As memory is constrained on embedded systems, the high memory demands of DBTs present a problem. The memory demand of DBTs can be attributed to the following sources: 1) translated code that is cached in a software code cache, 2) auxiliary code that is also cached in the software code cache to maintain control over the guest application, 3) data structures that support the software code cache and 4) DBT code. The particular service being offered by the DBT will have its own memory requirements. It should be noted that there are several other sources of memory demand in the system: The code, data and stack segments for the guest application also must be accommodated. As embedded systems are increasingly supporting multi-tasking, other applications will be simultaneously executing in the system and will require space. The operating system also shares the same memory space.

Figure 1.2 shows the relative importance of the different sources of memory demand within the DBT. For example, on average, translated code, auxiliary code and data structures respectively constitute 23%, 41% and 36% of the total memory demand in Pin, an industrial-strength DBT. Similar data on the code cache components has been found [15, 59] across different DBTs. Therefore, all three sources of memory demand are important and are addressed by in this dissertation.

It is important to reduce the memory demand because 1) decreased memory pressure is better for performance as well as power on an embedded system, and 2) many combinations of guest applications and DBT services may be disabled due to memory demand. Higher memory pressure impacts performance due to increased traffic between the disk and RAM, and also between the RAM and caches. This increased traffic increases power consumption as well, which is not conducive for battery-powered embedded devices. Additionally, embedded systems that we experimented upon have to support operating systems and multiple tasks in its memory. The space available to the guest application execution must hold its text, data and stack segments, the DBT code, the DBT code cache and data structures, and the data required by the DBT service. For example, we used a PDA with a 64 MB RAM, in which 15 MB was always occupied by the OS. The DBT code always occupied 2 MB when the DBT was in action. Therefore 47 MB was left to the guest application. Consider some shadow memory based DBT tools [79, 81], in which the shadow memory occupies as much space as the guest application. For such a scenario, the guest application is allowed to occupy only 23.5 MB in its code (consisting of both private and shared code), data and stack. In reality, it is allowed to occupy even less space because there are DBT code cache and data structures too. 23.5 MB is not large for embedded applications of today which cover everything from games to streaming media. Therefore, to enable as many DBT services on as many applications as possible, it is important to be able to reduce the memory demand of the DBT as much as possible.

DBT memory demand may give rise to high memory pressure on the platform leading to high memory management overheads by the operating system. The memory demand of DBTs is often handled by placing a limit on the DBT memory requirements. The DBT must flush translated code and its corresponding auxiliary code and data structures throughout execution to stay within the memory limit. However, flushing gives rise to performance degradation for two reasons: 1) book-

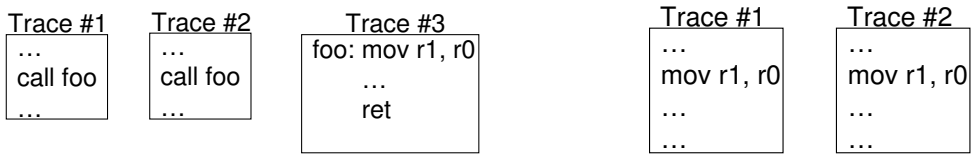
keeping must be done to support flushing and 2) flushed translations may require retranslation. Therefore, in this situation, increased memory demand manifests itself as a performance degradation.

1.2 Challenges to Reducing Memory Demand

The major challenges to reducing memory demand arise from performance optimizations that are applied to the DBT. Another challenge is that the impact of DBT design decisions on the memory demand is not usually clear. The impact of these design decisions on performance in a memory-limited situation is also unclear. The following sections describe these challenges in detail.

1.2.1 Code Caching

DBTs always suffer a performance degradation from native execution because they perform additional tasks such as translation during execution. Therefore, traditional DBT designs primarily focus on improving performance. However, performance optimizations for DBTs often require extra space. The most common performance optimization is caching translated code, which gives rise to the code cache. A code caching system necessitates the use of auxiliary code and data structures. Auxiliary code is needed because control has to be explicitly transferred back to the translator after executing cached code regions. In contrast, interpretive systems always maintain control over the guest application because the entire interpretation is performed within the runtime. Similarly, the code cache directory in conjunction with the code cache facilitates reuse of code regions, as it enables code regions to be located after they have been cached. Links are also necessitated to reap more benefit from caching by executing in the code cache for longer periods of time. Linking requires link data structures because links need to be quickly located and removed when the target code region is flushed. Therefore, DBT memory demand is intrinsic to improving performance and enhancing DBT usability. Memory optimizations may negatively impact performance and it has to be ensured that such degradation is within acceptable limits.



(a) Code cache in which function inlining has not been performed.

(b) Code cache in which function inlining has been performed.

Figure 1.3: Some performance optimizations increase memory requirements. In this example, function inlining reduces the dynamic instruction count at the cost of code cache expansion.

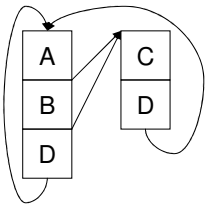
(a) Control flow graph. Consider the situation in which only paths ABD and ACD have executed.

(b) Translated and cached code regions corresponding to the control flow graph. Path ABCD has not yet executed. Aggressive linking DBTs will place link BC in anticipation and produce unnecessary link data structures.

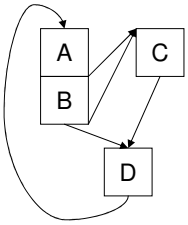
Figure 1.4: Performance optimizations such as aggressive linking to reduce context switches between the translator and the code cache can cause data structure size expansion.

1.2.2 Optimizing Performance of Cached Code

Given a caching system, DBT designers strive to further optimize performance by 1) reducing the dynamic instruction count of execution within the code cache, 2) increasing the code cache locality and 3) reducing the number of context switches between the translator and the code cache. Many optimizations that reduce the dynamic instruction count of execution within the code cache involve duplicating common code to avoid executing branches to the common code. Duplicating code enlarges the code cache. Figure 1.3 shows an example in which function inlining is used to eliminate call and return instructions but results in duplicating the inlined function. Similarly, increasing the



(a) Translation policy to maximize code cache locality. Allows duplication of code region D.



(b) Translation policy to minimize code duplication. Translates D as a separate code region. More code regions are formed and more code cache directory entries are required.

Figure 1.5: Two different translation policies applied to the control flow graph in Figure 1.4(a). Each translation policy impacts different components of the memory demand in different ways.

code cache locality involves placing code that execute close to each other in time in physically close locations. For code that is executed by multiple program paths, code cache locality implies duplicating it and placing it near code corresponding to each such path. The inlining example in Figure 1.3 enhances code cache locality at the expense of code duplication. Context switching between the code cache and the translator can be reduced by linking code regions. Linking strategies may range from less aggressive to more aggressive. Figure 1.4 shows an example in which the runtime has to decide whether to place links anticipatorily between B and C. An aggressive linking runtime will place the link anticipatorily although the path may never get traversed. The link will still need to be recorded in data structures. Therefore, aggressive linking will eliminate many control transfers to the runtime but will increase the size of link data structures. These examples show that many performance optimizations lead to memory expansion.

1.2.3 Translating Code Regions

The problem of reducing the memory demand of a DBT is further complicated by the fact that the DBT policies of translation impact the different components of memory demand in different ways. For example, a DBT may translate for better code cache locality. To this end, the translator can string together pieces of code that represent a dynamic program path into a code region. Such a translation policy applied to the control flow graph in Figure 1.4(a) will produce the translation

in Figure 1.5(a). This strategy produces tail duplication. A memory optimization may strive to reduce tail duplication and form smaller code regions. However, forming smaller code regions will imply that the code cache directory is able to locate at a finer granularity and therefore, stores more data. Figure 1.5(b) shows the translation produced by this strategy. Hence, the overall impact of translation policies on DBT memory demand is not always clear.

1.2.4 Flushing Code Regions

There are also challenges in using memory optimization approaches that are independent of the translation policy. For example, DBTs use flushing as a memory optimization strategy that is independent of the translation policy. However, flushing gives rise to a performance degradation because of book-keeping and retranslation overheads. Ideally, code regions should be selectively flushed to minimize retranslations for a given memory limit. However, selective flushing requires predicting what code will execute in the future, which requires profiling. Profiling needs to be effective so that accurate decisions can be made about code regions. Profiling also needs to be efficient since the profiling time will be credited to the total execution time. Lightweight profiles are efficient but may lead to inaccurate decisions which may not reduce overhead after all. Heavyweight profiling may yield better decisions but may increase the profiling overhead so much that there may not be any improvement to the overall execution time. Selective flushing also creates some code cache management issues because some code regions are evicted while others are preserved as the execution progresses. It may result in a fragmented code cache, for example. Also, for multi-threaded guest applications multiple threads share a code cache and a consistent code cache state has to be maintained at all times. Maintaining consistency of thread-shared code caches becomes even more complicated with a selective flushing system, as will be described later.

1.3 Research Overview

This research focuses on the core DBT memory demands. The memory demands due to the guest applications, the operating system, and the DBT service are outside the scope of this research.

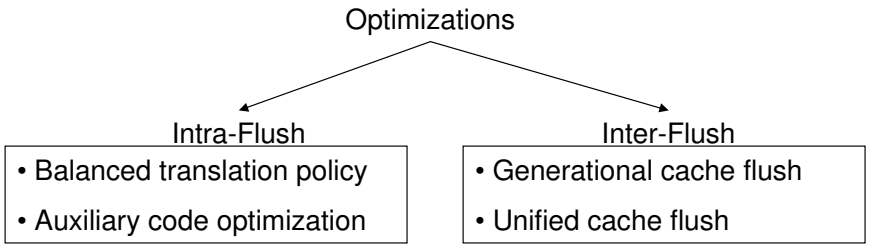


Figure 1.6: Categorization of optimizations developed in this research.Intra flush optimizations apply between two consecutive flushes while inter-flush optimizations apply across flushes.

Among the sources of DBT memory demand, this research focuses on the translated code, auxiliary code, and the data structures because these are the components whose sizes vary depending on the guest application. The memory demand due to DBT code remains stable across guest applications and is better handled by static rather than dynamic techniques.

The research reduces the memory footprint of DBTs. Consequently, there will be a reduction of memory pressure on the host machine. For DBTs that adhere to a user-defined memory limit, this research will result in improved performance.

Our approach is to use custom memory optimizations to reduce the memory footprint of the DBT and to improve its performance in memory-limited situations. To this end, our approaches can be divided into two broad categories. Figure 1.6 classifies the optimizations developed in this research. In the *intra-flush* category, the optimizations are applicable in the time intervals between consecutive flushes. Intra-flush optimizations reduce the DBT footprint so that 1) memory pressure is reduced and 2) flush points are reached less often and performance is improved. In the *inter-flush* category, the research focus is on flushes. Code regions are selectively preserved across code cache flushes to reduce retranslation overhead and improve overall performance.

For intra-flush optimizations, we first analyze how translation policies impact the different components of memory demand. It is important to understand the overall impact, as there can be opposing effects on different components. Also, some aspects of the translation policy may improve performance at the expense of memory expansion. These performance improvements may not be realizable in memory-limited situations because the extra flushing induced by the higher memory cost may cancel out any benefits. Therefore, the translation policy also has varying impacts on per-

formance depending on the memory constraints. Given these problems, we first design a *balanced* translation policy that will be beneficial for the memory demand overall, and also the performance in memory-limited situations.

After determining the translation policy, we further tune for better memory efficiency and performance. In this step, we select sources of memory demand where further optimization opportunities exist. For the given translation policy, no other component of the memory demand should be due to the optimization of our selected sources. Therefore, this is a problem of local optimization rather than global optimization. The only constraint is that we cannot perform any optimization that modifies the functionality of any memory demand component. In this step, we found auxiliary code to fit the requirements and optimized it in isolation from any other memory demand component.

For inter-flush optimizations, a spectrum of choices is possible. Profiling needs to be performed to decide whether to evict or promote each code region. The profiling has to be effective and efficient. Last usage time, execution count, state of call stack are some of the many indicators of code region lifetime we can profile. There is a tradeoff between the amount of profiling data gathered and the quality of the flush. For example, profiling can be carried out for a long period of time to yield better decisions at the expense of increased profiling overhead. Alternatively, profiling can be carried out for short periods. Code cache management also plays an important role in inter-flush optimizations. Code cache management decides how to handle the side-effects of selective flush, such as code cache fragmentation. Thus, the challenge is to choose a point in this multi-dimensional spectrum for which the benefits outweigh the overheads by a reasonable margin.

We developed two approaches that lie at two different points in this multi-dimensional spectrum. In the first approach, we used a generational code cache. The code cache is divided into two parts or *generations* that are ranked from least persistent to most persistent. Code regions are promoted to more persistent generations as they continue to execute. Selective flushing of less persistent generations occur more frequently and vice-versa. Generational code caches also solve the problem of fragmentation efficiently. We used profiling for short periods of time to decide whether to evict a code region or promote it to a more persistent generation. We monitored whether code regions are being executed at all and how many times they are being executed in the profile period. In

the second approach, we used a unified code cache. While such a cache did not have the benefits of a generational cache, it is more efficient in handling multi-threaded guest applications. In this approach, however, we used profiling for longer periods of time to enable better decisions. We name the first approach as the *generational cache flushing* approach and the second as the *unified cache flushing* approach.

Finally, this dissertation combines the approaches to provide a comprehensive solution. While the different contributions offer varying amounts of improvement, we evaluate the overall impact of a memory-oriented DBT design.

The following are the requirements we adhere to:

1. Generality - The research is independent of specific characteristics of particular DBTs. The research problems addressed and the solutions presented are applicable to the entire class of translation-based DBTs. The research is also independent of the particular service being provided by the DBT.

2. Efficiency - The research reduces memory footprint when a user-defined memory limit has not been placed on the DBT. The research improves performance when a user-defined memory limit has been placed on the DBT.

3. Transparency - The DBT system produced by the research is able to support the same set of guest applications and services as before.

1.4 Contributions of the Dissertation

The contributions of this dissertation are the following:

- A demonstration that path selection is a memory optimization tool and the design of a path selection strategy for memory efficiency and performance.
- A demonstration of the memory demands of auxiliary code and the the design of strategies to reduce auxiliary code size.

- The design and implementation of a generational cache flush and a unified cache flush strategy.
- The combination of path selection, auxiliary code optimization, and flushing strategies into a full system.
- The evaluation of each strategy separately and evaluation of the combined system on embedded platforms.

1.5 Organization of the Dissertation

In the remaining part of the dissertation, Chapter 2 provides background on DBT technology to facilitate understanding of the subsequent chapters. Chapters 3 through 7 form the body of this dissertation. Chapters 3 and 4 deal with intra-flush optimizations. Chapter 3 describes our research on balanced translation policies for improving the overall memory demand and performance [49]. Chapter 4 describes further tuning of the memory demand components by optimizing auxiliary code size [46]. Chapters 5 and 6 deal with the inter-flush optimizations. Chapter 5 describes the generational cache flushing technique [47] while Chapter 6 describes the unified cache flushing technique [48]. Chapter 7 describes a whole system comprising the research in the previous chapters. Chapter 8 provides a survey of related work that shows where this research stands in relation to work by other researchers. Chapter 9 concludes the dissertation and includes a discussion on future work.

Chapter 2

Background

Figure 1.1 shows the overall design of a DBT. A DBT forms a software abstraction layer between the guest application and the host machine. The core of a DBT is a translator that fetches code from the guest application and translates it according to the service being provided by the DBT. The translator stores these translations in a software code cache and the cached instructions execute directly on the host machine. Translation and caching of code regions occurs on-demand and therefore, control must be repeatedly transferred from the code cache to the translator for translation of new code regions. Auxiliary code is cached to facilitate the control transfers from the code cache to the translator. However, code regions are gradually linked by the translator to avoid too many control transfers.

The following sections provide more detail on the various aspects of a DBT. Section 2.1 describes the data structures used by a DBT. The following sections describe the various mechanisms of a DBT and how these mechanisms make use of the data structures. Section 2.1.1 describes the translation mechanism. Section 2.2 describes the process of control transfer between the code cache and the runtime. Section 2.3 describes linking while Section 2.4 describes flushing. Most of the mechanisms described in these sections apply to both single-threaded and multi-threaded guest applications. However, there are a few aspects in which the execution of multi-threaded guest applications differs from the execution of single-threaded guest applications. Section 2.5 describes these differences and the special mechanisms used for multi-threaded guest applications.

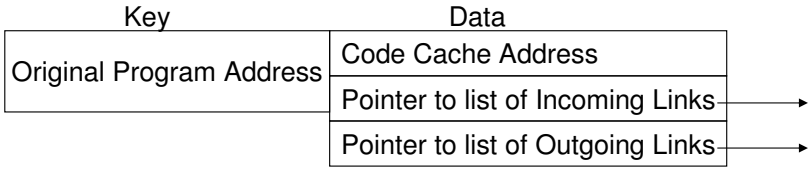


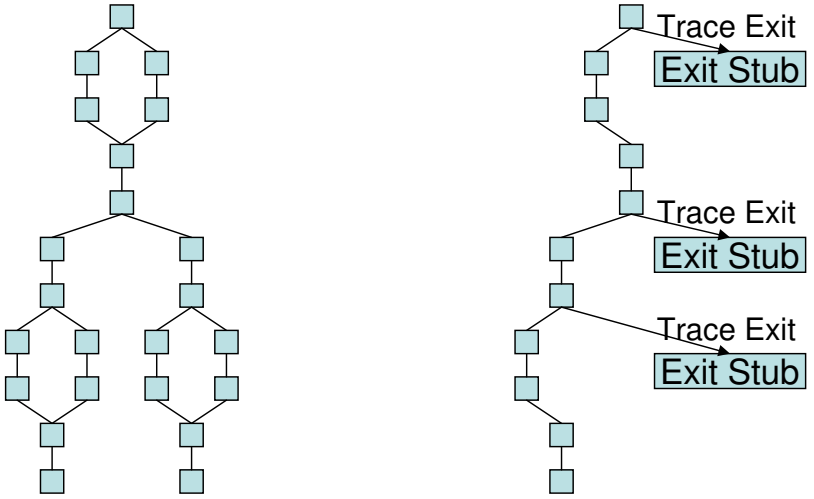
Figure 2.1: An entry in the code cache directory. There is an entry corresponding to each code region.

2.1 Required Data Structures

The translator uses data structures to keep track of code regions. The main data structure is the code cache directory which is a hash table containing an entry for each code region. It is used to map original program addresses to code cache addresses. Figure 2.1 shows a typical entry in the code cache directory. The original program address is the key which is used to search the directory. The data structures also keep track of links between code regions. The code cache entry for a code region contains a pointer to a list of incoming links for the code region and to a list of outgoing links for the code region.

2.1.1 Translating Code Regions

When the translator receives a translation request for some program address, it first searches the directory to determine whether a corresponding code region is cached and if so, where it can be found. If the code region is not found, the translator generates the code region and creates a directory entry for it using the original starting address as the key. The translated code region is a trace which represents a dynamic program path. Figure 2.2 shows an example of a trace corresponding to a code snippet in the guest application. Figure 2.2(a) is the control flow graph for the code snippet. Often, a trace is formed from the frequently traversed paths in the control flow graph. The trace in Figure 2.2(b) is a possible trace formation for this particular code snippet. The first basic block in the trace is the basic block starting at the requested program address. The tail basic blocks can be chosen in several different ways, for example, by following a hot path. Usually, traces are defined to be single-entry code regions. A single entry point facilitates optimization and easy lookup of traces. Additionally, allowing multiple entry points would require keeping track of all tail basic blocks in



(a) Control flow graph of a code snippet.

(b) A trace for the code snippet.

Figure 2.2: Code regions are traces representing dynamic program paths.

data structures. There are multiple branches off traces. The branches off a trace are known as *trace exits*.

2.2 Transferring Control between Translator and Code Cache

Figure 2.2(b) shows that there are several trace exits or branches off the trace. The runtime needs to properly direct these trace exits to maintain control over the guest application. These trace exits target *exit stubs* which constitute auxiliary code in the code cache. Exit stubs act as trampolines that transfer control from the code cache to the translator. These exit stubs are responsible for saving the guest application context before transferring control. The guest application state is restored before resuming execution in the code cache. The control transfer is known as a *context switch*. These exit stubs are also responsible for communicating the target address of the trace exit to the translator. Since the target address is specific to the trace exit, exit stubs are specialized for the particular trace exit they handle.

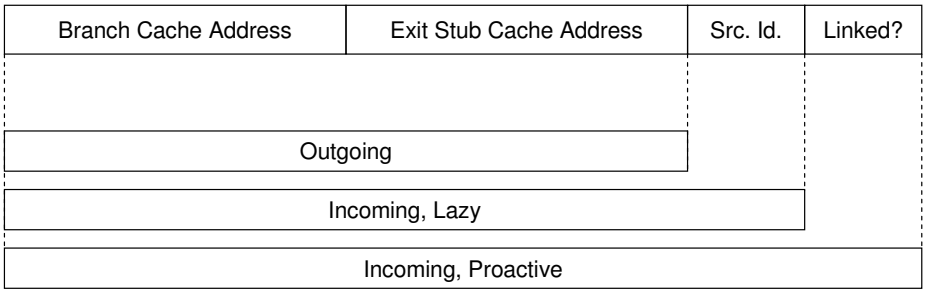
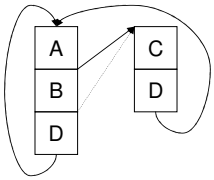
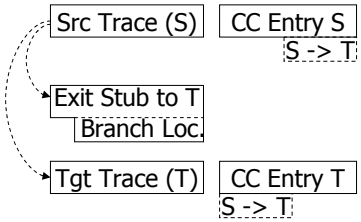


Figure 2.3: A link node.



(a) Lazy linking places a linking only when the path is traversed. For the example in Figure 1.4, lazy linking would not place link BC until the path is traversed.



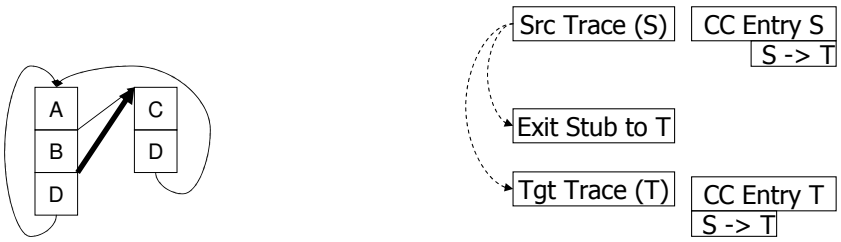
(b) Lazy linking stores the branch location in the exit stub. The link is placed and recorded only when the path first traverses.

Figure 2.4: Lazy linking.

2.3 Linking Traces

As mentioned in Section 2.2, execution can progress by passing control from trace to trace through the translator. However, each context switch between the code cache and the translator is expensive because saving and restoring of states must occur. This expense can be avoided by passing control directly from one trace to another. Control is directly passed by patching trace exits to point to their cached targets, if possible. The condition for patching is that 1) the target must be present in the code cache, and 2) the target of the trace exit may not vary during the execution i.e., the trace exit should be a *direct* branch. This process of patching trace exits to jump directly to their targets is known as *linking*.

Information about links is stored in the code cache directory, as shown in Figure 2.1. The list of incoming links is stored because these links need to be unlinked if the target trace is ever evicted.



(a) Proactive link places the link as soon as the source and target traces are cached, regardless of whether the path will ever traverse. For the example in Figure 1.4, proactive linking places BC even if the path never executes.

(b) Proactive linking records every potential link. If the target is not yet cached, a tentative entry for the link is stored in a tentative code cache entry for the target.

Figure 2.5: Proactive linking.

Unlinking requires backpatching of trace exits to point to their corresponding exit stubs. The list of outgoing links needs to be maintained to quickly unlink and force control back from an executing trace to the runtime, for example, to facilitate signal handling.

Figure 2.3 shows a typical node in a list of links. The outgoing link node is the simplest and consists only of the cache location of the branch and its corresponding exit stub. The incoming link node must contain a trace identifier in addition to these two data. The identifier is a number that is assigned to each trace and is unique over the entire execution. For example, if a trace is translated once, evicted and then translated again, it will have different identifiers each time it is translated. The trace identifier is needed in an incoming link node to check that the source trace corresponding to the branch and exit stub locations still exists in the code cache. There is another link node field which indicates whether the link is in place or not. This field is needed depending on the linking strategy. The following paragraphs explain the different linking strategies and the meaning of this field.

There are two popular linking strategies - lazy and proactive. Lazy linking is a less aggressive form of linking while proactive linking is more aggressive. *Lazy linking* places the link when the path first traverses. Figure 2.4(a) shows an example of lazy linking. As shown in Figure 2.4(b), lazy linking is implemented by storing the location of the branch in the exit stub. The translator uses the location of the branch to determine where to patch. The translator also stores data about the link in

the source and target code cache entries when it places the link.

Proactive linking places a link as soon as the source and target traces appear in the code cache, regardless of whether the path will ever traverse. Figure 2.5(a) shows an example of lazy linking. When a trace is being translated, a proactive linking configuration examines each of the trace's outgoing branches. As shown in Figure 2.5(b), each potential link is recorded. If the target trace is not yet in the code cache, a tentative code cache entry for the target is formed and a tentative incoming link is registered with the target. When the target eventually gets translated, it is bound to the already existing code cache entry and the code cache entry is updated with the code cache address of the target. The potential links to the target are already available in the code cache entry and can be immediately placed, thereby enforcing a proactive linking policy. If the target is already in the code cache when the source is being translated, the link is placed at the same time it is recorded. The use of tentative entries implies that there is a time gap between recording the entries and the actual linking. At the time of linking, it must be determined whether the link has been already placed. Also, it must be determined that the source trace still exists (using the trace identifier). Therefore, the link node needs a field to indicate the status of the link.

While all the above linking strategies apply to direct branches, some form of linking is possible for *indirect* branches (branches whose targets vary, such as returns). Although the targets of indirect branches may vary, an indirect branch may target the same location multiple times. Therefore, indirect branches can be profiled to store predictions about the target. However, each prediction needs to be checked for correctness before executing. Indirect branch prediction tables, prediction chains, sieves, and return address stacks are some popular methods for handling indirect branches. Due to the same reasons as direct branch linking, lists of incoming and outgoing indirect branches must also be maintained. The exact format of the link node depends on the particular indirect branch handling method used. Indirect branch linking is not discussed in depth in this dissertation.

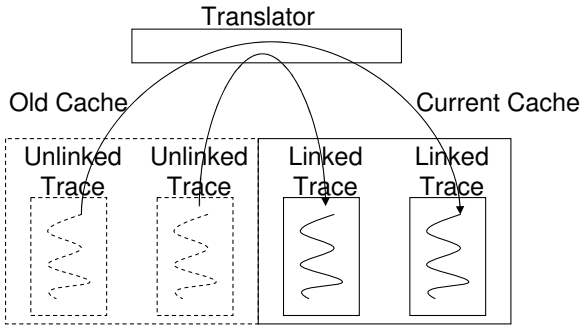


Figure 2.6: Schematic of a thread-shared cache flush. All threads from the old cache must be dispatched to the current cache before eviction.

2.4 Flushing Traces

Traces may need to evicted from the code cache for several reasons (unloading dynamic libraries, self modifying code, user changing instrumentation, re-optimization, bounded caches). Flushing has to 1) force control out of the traces selected for eviction, 2) evict the traces and 3) ensure that control cannot re-enter evicted traces. For single-threaded guest applications, control is guaranteed to have left cached traces when the runtime executes. The traces can then be evicted. The code cache directory entries corresponding to the evicted traces are also discarded to prevent re-entry. In the case of a partial flush, any links to the evicted traces are also removed, to guarantee that the evicted traces will not be re-entered. In addition to unlinking, the data structures for locating evicted traces are discarded, so that the runtime cannot direct control to evicted traces. Eviction for multi-threaded guest applications is discussed in Section 2.5.

2.5 Executing Multi-Threaded Guest Applications

DBTs host both single-threaded and multi-threaded guest applications. For single-threaded guest applications, the single thread alternates between executing in the code cache and in the runtime. For multi-threaded guest applications, there are two choices. A *thread-private* code cache can be allocated to each thread. However, this option has been found to be very inefficient in memory even for general-purpose platforms [14, 52]. Therefore, a single *thread-shared* code cache is allocated. Simplicity of code cache management is traded off for memory efficiency, in thread-shared code

caches. Multiple threads simultaneously execute in the code cache. However, for simplicity, only one thread at a time is allowed to execute the runtime in many DBTs [52]. Such a design choice does not degrade performance significantly because the runtime is expected to execute for only a small fraction of the total execution time.

Context switching is more complicated for thread-shared code caches. The guest application context that is saved at the time of a context switch corresponds to a particular thread. Therefore, there needs to be separate locations for saving the context of each thread, so that they do not overwrite each other. At the same time, the absolute address at which to store the thread context cannot be encoded into the context switching code because the code is shared by all threads. Some DBTs solve this problem by storing the context at an offset from the thread stack. Some DBTs perform register reallocation to free up a register to serve as the thread context pointer. The thread context is always saved at the location pointed to by this register.

There are added implications for flushing of thread-shared code caches. For a thread-shared code cache, the runtime needs to ensure that no threads are executing in the selected traces during a flush. Traces selected for eviction are considered to be old traces while all other traces are considered to be current traces. Current traces can constitute existing traces that were not selected for eviction or newly formed traces. As shown in Figure 2.6, the runtime unlinks old traces to expedite the exit of threads. Unlike single-threaded code caches, unlinking is needed for both full and partial flushes. The runtime ensures that each thread that was executing in the old traces have exited from them once. It avoids dispatching threads into the old traces by discarding the corresponding code cache entries. Also, the fact that they are unlinked ensures that control cannot pass into them from other cached traces. When all the threads that were executing in the old traces have exited, the old traces can be discarded. The threads that exit the old traces are not blocked, to avoid deadlock. Instead, these threads are dispatched to current traces. In the case of full flush, threads are dispatched to current traces in a newly allocated code cache. Since the old and current traces coexist for some time, the sum of their sizes must be within the memory limit. Thus, the flush is triggered some time before reaching the memory limit i.e., at a *high water mark*.

Chapter 3

Balanced Path Selection for Holistic Memory Efficiency and Performance

The sizes of the memory demand components of a DBT depend on *path selection* which denotes the way that code is selected for each trace and the way traces are linked. Figure 1.5 showed an example where path selection impacted different components of the memory demand in opposing ways. Allowing code duplication for better code cache locality increased the translated code size. Disallowing code duplication, however, increased the code cache directory size. Path selection also has implications for performance. In this example, better code cache locality is better for performance. However, if the total memory demand due to the increase in code duplication surpasses the total memory demand when disallowing code duplication, the path selection for better code cache locality will incur more flushing activity. Since flushing often results in a performance degradation, the benefits of better code cache locality may be canceled out.

It is challenging to design a path selection strategy for improved holistic memory efficiency because of the complex interactions among the three sources of memory footprint. For example, some strategies that reduce the translated code size may increase the data structure size as well as the total memory demand and vice-versa. Additionally, some path selection aspects may degrade performance in memory-unconstrained environments, while their improved memory efficiency may improve performance in memory-limited situations.

Traditionally, only code cache memory demand has been considered for optimization. But we

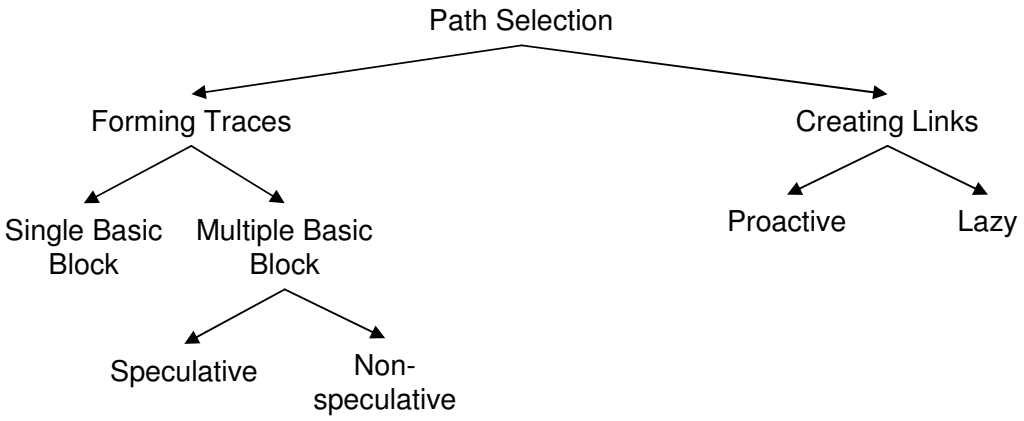


Figure 3.1: Topology of the path selection choices.

show that we reach better conclusions about the total memory demand when we also consider data structures and the interactions between the code cache size and the data structure size. For our performance measurements, we place a limit on the sum of the code cache and data structure sizes rather than only on the code cache size.

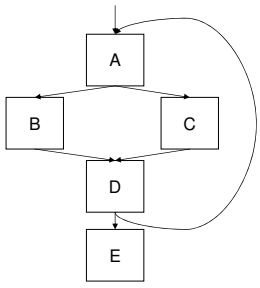
Our goal in this chapter is to present a path selection strategy that holistically optimizes all three memory sources (translated code, auxiliary code, and data structures) without degrading performance. We explore the interactions of path selection with memory demand and performance in this chapter to motivate the design of balanced path selection strategies. We enumerate all aspects involved in a path selection design and evaluate a comprehensive set of approaches for each. Finally, we propose a path selection strategy that balances memory efficiency and performance. The proposed path selection strategy specifies 1) how to define the granularity of a trace, 2) whether code should be speculatively selected for traces, 3) how informed the speculation (if used) should be, 4) when a trace should be terminated, and 5) how traces should be linked.

We describe the various aspects of path selection and their implications in Section 3.1. We evaluate path selection strategies and propose a strategy for holistic memory efficiency and performance in Section 3.2. Finally, we summarize in Section 3.3.

```
A: …
   branch to C if not equal
B: …
   branch to D
C: …

D: …
   branch to A if less than
E: …
```

(a) Original code snippet to be translated.



(b) Control flow graph of code snippet.

Figure 3.2: Section 3.1 describes the different path selection strategies by applying them to this example code.

3.1 Path Selection

Path selection determines how code is selected to form a trace and how traces are linked to each other. Figure 3.1 depicts the design space. Traces and links between traces make up the program paths in the code cache. When forming a trace, the first basic block is fully included, since all instructions are guaranteed to execute. The translator may stop or continue translation after the first basic block. Since the outcome of a branch ending the first basic block cannot be determined *a priori*, the translator may continue translation speculatively or non-speculatively (for example, by executing the partially formed trace to determine the branch outcome). Similarly, links between traces may be placed speculatively (proactively) or when the path actually executes for the first time (lazily). Each of the choices presents a tradeoff among the memory components (translated code, auxiliary code, and data structures) or a tradeoff between memory efficiency and performance, as discussed in the following sections. We use the snippet of code in Figure 3.2(a) as a running example to explain the configuration choices and their tradeoffs. We assume that there is an initial translation request for A. The execution follows path ABD once and then follows path ACD repeatedly before exiting to E. Figure 3.2(b) shows the control flow graph corresponding to Figure 3.2(a).

3.1.1 Single-Block vs. Multi-Block Translation

Given a program address, the translator may choose to translate a trace containing one or more basic blocks starting at that address. For example, Figure 3.3 shows two possible trace formations when the translator attempts to translate A from Figure 3.2. Figure 3.3(a) shows a single-block trace starting at A. Figure 3.3(b) is an example of a multi-block trace starting at A. White blocks are part of the trace while shaded blocks represent exit stubs.

In Figure 3.3(b), if B appears on some other program path, B will have to be translated again (duplicated) because side entries to traces are not allowed. Single-block traces will not suffer from such duplication. However, in Figure 3.3(b), there is only one off-trace branch for A, while in Figure 3.3(a), there are two off-trace branches for A. This phenomenon occurs because both outcomes of a conditional branch need to be handled in translated code. For multi-block traces, one of the outcome targets can be part of the trace. For single-block traces, both outcome targets are off trace. Therefore, single-block traces have more branches and exit stubs per unit of translated code. Also, more links have to be recorded for single-block traces, increasing the proportion of data structures.

Another side effect of single-block traces is that there are more code cache directory entries per unit of code, increasing the proportion of data structures further. For example, in Figure 3.3, if B appears in a single program path, the multi-block trace will save storing a code cache directory entry for B.

A higher proportion of auxiliary code for single-block traces implies that a smaller proportion of the code cache is available for translated code, leading to lower code cache locality. The lack of duplication for single-block traces also implies that temporally close code may not be spatially close, again leading to lower code cache locality. Moreover, the number of context switches will be higher as code is translated one basic block at a time.

In summary, single-block traces reduce code duplication but increase the proportion of auxiliary code and data structures. Regarding performance, single-block traces suffer from lower code cache locality and higher context switches. Table 3.1 presents these observations in a tabular format.

(a) Single-block trace. (b) One possible formation of a multi-block trace.

Figure 3.3: Translation to single-block and multi-block traces. White boxes are parts of the trace, while gray boxes are exit stubs. The gray boxes are labeled with the corresponding branch targets.

	Single-block traces	Multi-block traces
Code duplication	Less	More
Code cache entries	More	Less
Code cache locality	Less	More
Trace exits	More	Fewer
Exit stubs	More	Fewer
Links	More	Fewer
Context switches	More	Fewer

Table 3.1: Table summarizing the impact of the number of basic blocks in a trace.

3.1.2 Multi-Block Trace Selection and Termination

Figure 3.3(b) shows that the translator chose to translate B to extend the trace starting at A. Several other choices are possible for a multi-block trace. In this section we discuss strategies for forming a multi-block trace.

Trace Selection. We extend a trace by selecting one basic block at a time. We can select the basic blocks non-speculatively by executing the last basic block to determine what is going to be the next basic block, or speculating by predicting the next basic block. In Figure 3.4(a), we execute A and find that B is the next basic block to execute. We append B to A. In Figure 3.4(b), we speculate that C is the basic block likely to execute next (for example, from offline profiling data) and we append C to A.

There are more context switches in forming non-speculative traces as these are translated one

(a) Extending trace non-speculatively.

(b) Extending trace speculatively.

Figure 3.4: Extending a trace. White areas are parts of the trace while gray areas are exit stubs.

	Speculation	Non-speculation
Context switches	Fewer	More
Space waste	More	Less

Table 3.2: Table summarizing the impact of trace selection.

basic block at a time. However, if the speculation is incorrect, there will be wasted space for translated code and data structures. Table 3.2 summarizes these facts about trace selection.

Trace Termination. After translating each basic block, the translator must determine whether to extend the trace further. We should ideally continue to extend the trace if the next basic block appears in this single program path because it will not be duplicated elsewhere. We should start a new trace with the next basic block if it appears on other program paths because it will be duplicated otherwise. For example, suppose both B and C are targeted by basic blocks other than A. We should

Basic Block Type	**Average No. of branches pointing to Basic Block**
Fall throughs of conditional branches	0.082
Targets of conditional branches	1.429
Targets of unconditional branches	3.219

Table 3.3: Table showing the average number of branches pointing to different types of basic blocks. The results are averages taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin [69].

(a) Terminating trace when the next basic blocks, B and C, appear on other program paths.

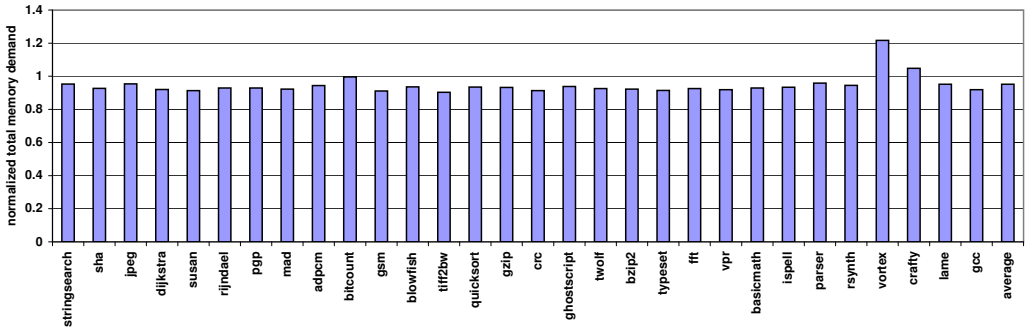
(b) Terminating trace when the next basic block, B, appears only on this path.

Figure 3.5: Termination of a trace based on the number of different paths that execute the next basic block. White areas are parts of the trace while gray areas are exit stubs.

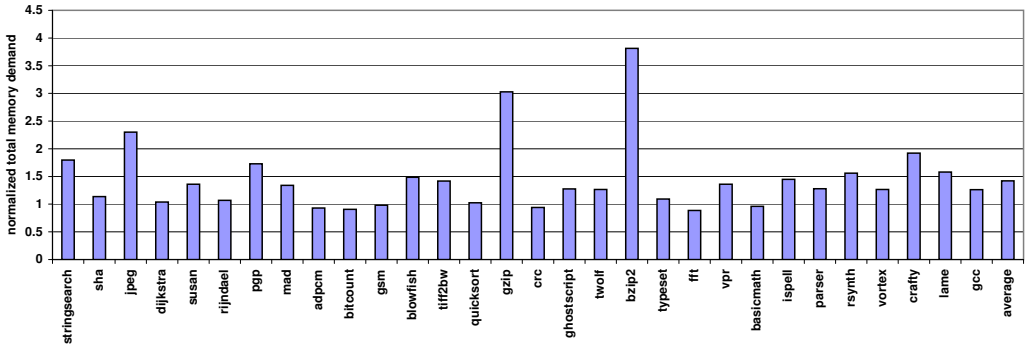
produce the trace in Figure 3.5(a). However, if B always follows A, we should produce the trace in Figure 3.5(b).

We experimentally explore the trace termination condition here because the number of program paths executing a basic block is independent of other aspects of path selection. Therefore, this aspect of path selection can be separately evaluated to reduce the number of path selection combinations. We hypothesized that the number of program paths that execute the next basic block is correlated with the type of the branch ending the last translated basic block. The number of paths executing the next basic block is equal to the number of branches found to be targeting the next basic block. If the branch ending the last basic block is taken, the target of the branch is the next basic block to be executed. If the branch ending the last basic block is not taken, the fall-through of the branch is the next basic block to be executed. Indeed, as shown in Table 3.3, we found that if the next basic block is a fall-through of the direct, conditional branch ending the last basic block, it is rarely targeted by branches, while if the next basic block is a target of a direct branch ending the last basic block, it is usually pointed to by other branches also. Therefore, we terminate traces based on branch type i.e., whether it is a not taken direct conditional branch, taken direct conditional branch or a direct, unconditional branch.

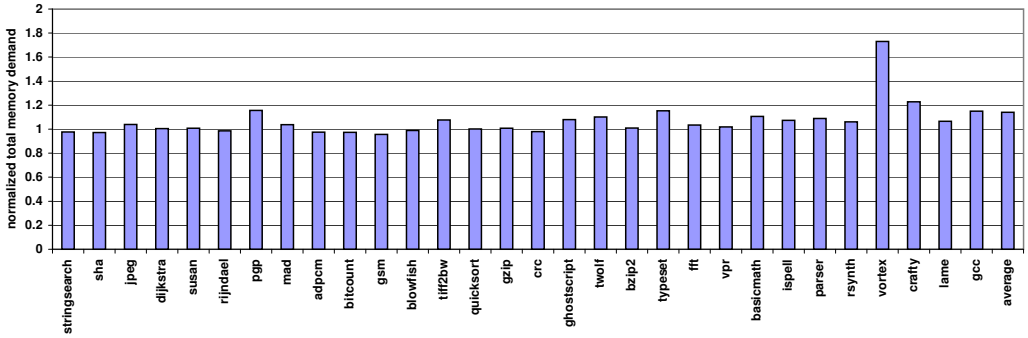
To confirm our hypothesis, we measured the DBT memory requirements when 1) not taken direct conditional branches are *elided* to the trace (Figure 3.6(a)), 2) taken direct conditional branches



(a) Normalized memory requirements of benchmarks when *not taken conditional* control transfers are elided. Memory efficiency improves in most cases and also on average.



(b) Normalized memory requirements of benchmarks when *taken conditional* control transfers are elided. Memory efficiency degrades 41% on average.



(c) Normalized memory requirements of benchmarks when *unconditional* control transfers are elided. Memory efficiency degrades 14% on average.

Figure 3.6: Normalized memory demand of terminating traces at different types of branches. The baseline is single-block traces. The results are taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin [69].

are elided to the trace (Figure 3.6(b)), and 3) direct unconditional branches are elided to the trace (Figure 3.6(c)). We then compared the results with single-block traces (essentially, not eliding any branch). If eliding any of the above categories of branches has better memory efficiency than the baseline, then it is considered worth eliding. Figure 3.6(a) shows the results of eliding not taken conditional branches. Memory efficiency improves in almost all the benchmarks with an average 5% improvement. Figure 3.6(b) shows the results of eliding taken conditional branches. Memory efficiency degrades 41% on average. Figure 3.6(c) shows the results of eliding unconditional branches. Memory efficiency degrades 14% on average. Although more branches point to targets of unconditional, direct branches, the degradation from eliding unconditional, direct branches is smaller because conditional, direct branches are higher in number. Also, allowing non-sequential control in a trace implies that tail basic blocks have to be tracked by the code cache directory. Such tracking is required because all cached code corresponding to some program addresses may need to be invalidated later on. We have not charged the cost of these extra code cache directory entries to the case where taken branches are elided. In reality, the memory efficiency will degrade further when eliding taken branches. Therefore, it is beneficial to elide only direct conditional branches that are not taken, which matches the data in Table 3.3.

These evaluations modify the trace selection strategies such that 1) non-speculative trace selection terminates traces at taken, direct branches, and, 2) speculative, profile-based trace selection terminates traces at direct branches predicted to be taken. All of these trace selection strategies continue to terminate traces at indirect branches.

3.1.3 Link Formation

Links can be formed proactively or lazily. Proactive linking places the link as soon as the source and target traces are cached. Lazy linking creates a link only when the corresponding path executes for the first time. Lazy linking needs less data structure space than proactive linking because proactive linking uses tentative code cache entries and tentative links which lazy linking does not. Some of the code cache entries and links created by proactive linking may never get used while each code cache directory entry and link created by lazy linking is guaranteed to be used. Additionally,

	Proactive Linking	Lazy Linking
Data structures	More	Less
Links	More	Less
Exit stub size	Less	More
Code cache locality	More	Less

Table 3.4: Table summarizing the impact of the linking strategy.

proactive linking needs larger link nodes. However, lazy linking stores branch locations in exit stubs resulting in larger exit stubs than proactive linking, leading to larger auxiliary code. From the performance perspective, proactive linking anticipatorily links traces and is more effective in reducing the number of context switches between the code cache and the translator. The larger exit stubs used by lazy linking increase the proportion of auxiliary code in the code cache, leading to a reduction in code cache locality. Table 3.4 summarizes the observations about the two linking policies.

3.2 Experimental Evaluation

We experimentally evaluated various path selection strategies to 1) evaluate their holistic memory efficiency and performance, 2) demonstrate that we arrive at correct conclusions about memory efficiency and performance only when we factor in data structures as well as the code cache, 3) investigate the tradeoffs among translated code, auxiliary code, and data structure sizes, 4) investigate the overall impact of the path selection on performance, and 5) propose a path selection strategy that achieves better memory efficiency without performance degradation in memory-constrained environments. We describe our experimental setup in Section 3.2.1. We present the results on memory efficiency in Section 3.2.2 followed by the results on performance in Section 3.2.3. We discuss the results and propose a path selection strategy in Section 3.2.4.

3.2.1 Experimental Setup

We implemented and evaluated two strategies for speculative trace selection. In the first strategy, we use data gathered in an offline profiling run to speculate which way a branch will go. We

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

Table 3.5: Trace selection strategies and their descriptions.

speculate about a branch only if it shows a particular bias for at least 90% of its executions in the profiling run. In the second strategy, we translate a contiguous stream of code until the trace size reaches a certain threshold size or it encounters an indirect or direct, unconditional branch. Such a strategy is equivalent to speculating that no conditional branch will be taken. The speculative strategy using profiling is highly informed, while the second strategy uses minimum information. We use the nomenclature shown in Table 3.5 for the trace selection strategies. Each of the trace selection strategies will be combined with both lazy and proactive linking, to form path selection strategies.

We evaluated both the memory and performance effects of the different path selection strategies. For memory effects, we measured the sum of the space occupied by the translated code, auxiliary code and the data structures. For all memory measurements in this dissertation, we assume a system without any memory limit i.e., there is no flushing activity. Since there is no flushing activity, the memory consumption monotonically increases. We measure the memory demand at the end of the execution for our memory experiments i.e., we always use the highest possible value of memory consumption. For performance, we measured the execution times of applications hosted by a DBT. Throughout this dissertation, the execution times are obtained using the 'time' command provided by the operating system. Each benchmark is executed three times in all performance experiments and the minimum of their execution times is used. To calculate the average, for all experiments in this dissertation, we use a weighted arithmetic mean as described in literature [60]. The weights corresponded to the values measured for the benchmarks in the baseline system.

In one of the performance experiments in this chapter, the DBT was limited to use half of the code cache and data structure memory it needs for each benchmark. Another performance

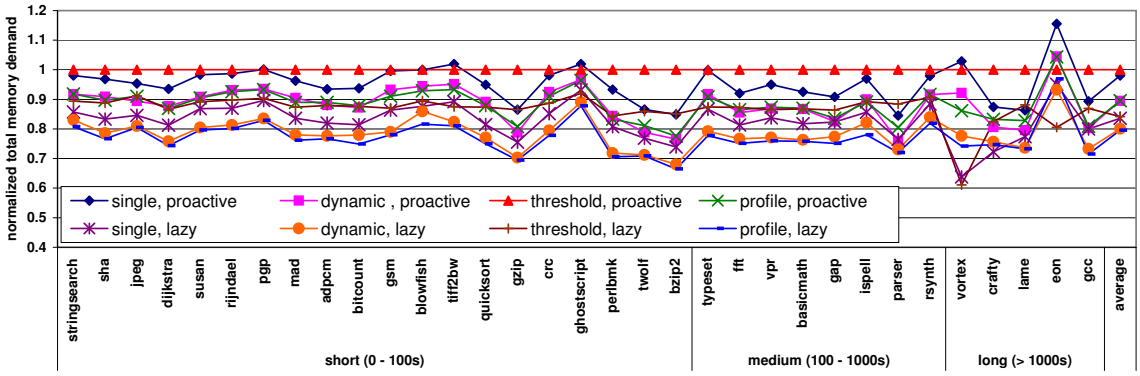
experiment used a uniform memory limit of 512 KB on all the benchmarks.

We used threshold-based trace selection combined with proactive linking as our baseline. We chose this baseline because it is used by Pin [69], a production-quality DBT. The memory and performance measurements are normalized with respect to the baseline.

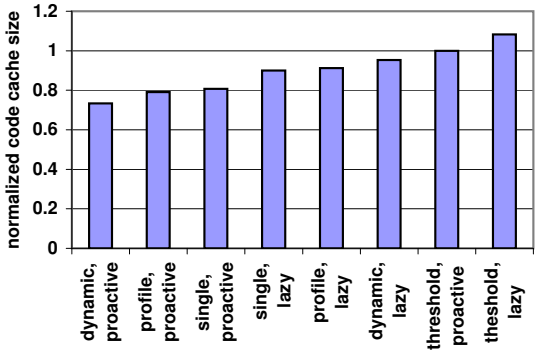
We used Pin for XScale [51] as our DBT. We implemented our strategies by directly modifying the Pin source code. However, our findings apply to other DBTs because the relative proportions of translated code, auxiliary code, and data structures are reported to be similar [8, 15]. Although Pin is generally used for dynamic binary instrumentation, we used it simply to host our benchmarks. When used without instrumentation, Pin uses the bare minimum data structures (only for tracking the traces and links) required.

We ran the SPEC2000 integer [55] and MiBench embedded benchmark [50] suites on a iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. The IPAQ has a 200 MHz StrongARM-1110 processor with 64 MB RAM, 16 KB instruction cache and a 8 KB data cache. 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.

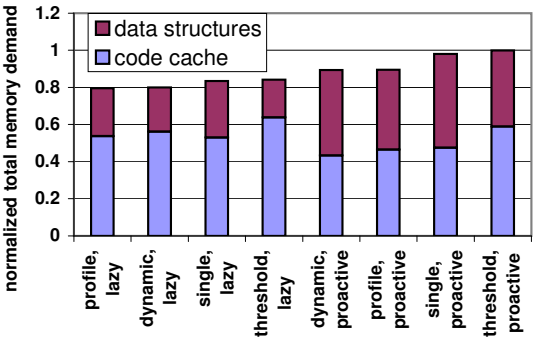
We divide the benchmarks into three groups - short-running, medium-length and long-running, according to their baseline execution times. The short-running benchmarks have execution times of less than 100 seconds. The medium-length benchmarks execute between 100 to 1000 seconds. The long-running benchmarks execute for more than 1000 seconds. We categorize the benchmarks because longer benchmarks are better able to amortize translation overheads and the effects of DBT optimizations become clearer as the benchmarks get longer. The total execution time of the long-running benchmarks exceeds that of all the benchmarks in the short and medium-length categories combined. In the graphs, where applicable, the benchmarks are arranged in increasing order of baseline execution time.



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



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



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

Figure 3.7: Normalized memory demands of different path selection strategies, with `threshold-based selection, proactive linking` as the baseline.

3.2.2 Memory Efficiency

Figure 3.7 shows the normalized memory demands of the benchmarks. Figure 3.7(a) presents the total memory demand for all the benchmarks. There are few intersections in the graph indicating that there is a consistent ranking among the different strategies for most of the benchmarks. Therefore, we use the summary graphs of Figure 3.7(b) and Figure 3.7(c). Figure 3.7(b) shows how the memory efficiency would rank the strategies if we consider the code cache only, while Figure 3.7(c) shows how it would rank the strategies if we consider both the code cache and data structures. There is great variation between the rankings of Figure 3.7(b) and Figure 3.7(c) because the strategies use different proportions of data structures. Therefore, it is misleading to consider the code cache size only to measure the memory demand. Also, since the ratio of code cache size to the data structure size is different for each configuration, there is no straightforward method to calculate the data structure size given the code cache size.

We first compared the strategies by fixing the linking strategy and varying the trace selection strategy. For lazy linking, multi-block traces (formed by dynamic or profile-based selection) have better memory efficiency than single-block traces. Code caches for single-block traces are slightly smaller or similar in size to the code caches for multi-block traces because there is less code duplication for single-block traces. But, single-block traces need more data structures per unit of translated code, which gives rise to larger data structure sizes for single-block traces. The smaller code caches are outweighed by the large data structures for single-block traces. Regarding the degree of speculation involved in trace selection, there is not much difference in memory efficiency between dynamic trace selection and profile-based trace selection because profile-based trace selection. However, threshold-based selection has worse memory efficiency than all the other selection techniques because it speculates inaccurately and wastes space. The results are similar for the proactive linking strategies.

Next, we compared the strategies by fixing the trace selection strategy and varying the linking strategy. For all the trace selection strategies, proactive linking produces smaller code caches than lazy linking because proactive linking needs smaller exit stubs leading to lower auxiliary code size. However, the total memory demand of lazy linking is less than that of proactive linking because the

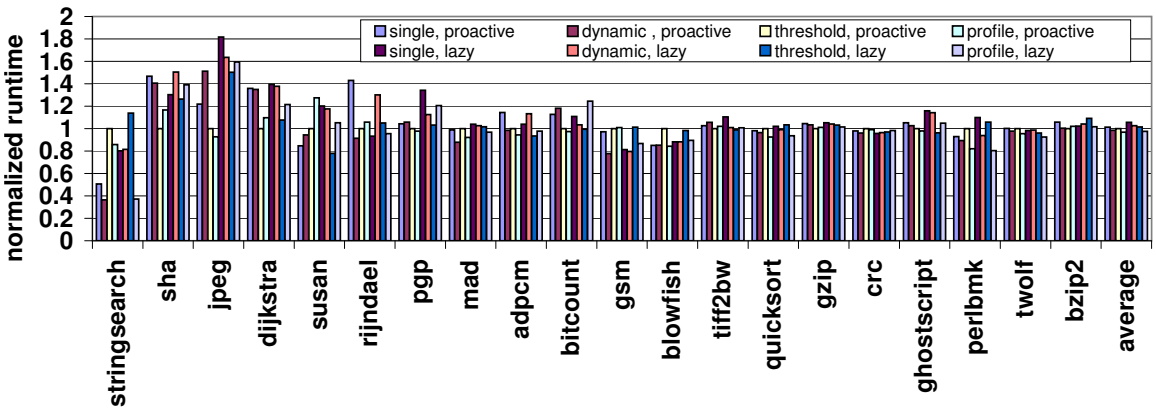
decrease in data structures due to lazy linking outweighs the increase in code cache size.

The following summarizes our memory efficiency evaluation:

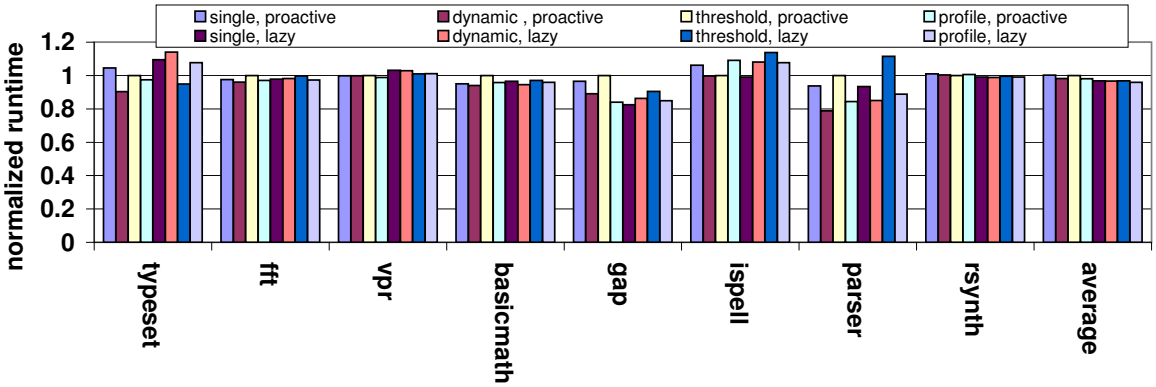
- Considering the code cache in isolation leads us to misleading conclusions about the memory demand. There are complex interactions among the memory components as shown by the variance in the relative allocation of space by the different path selection strategies.
- As shown in Figure 3.7(c), all the lazy linking schemes perform better than all the proactive linking schemes. Therefore, the increase in auxiliary code size due to lazy linking is outweighed by the decrease in data structures. The linking strategy has the most effect on memory efficiency.
- The influence of the linking strategy is followed by the strategy of deciding the number of basic blocks in a trace. The reduction in data structures and auxiliary code due to multi-block traces outweighs the increase in duplication. Multi-block traces have better memory efficiency than single-block traces.
- We found that the degree of speculation does not influence the memory efficiency much as long as the decisions are accurate. Both non-speculative (dynamic selection) and highly accurate, speculative (profile-based selection) trace selection perform well because neither waste space.
- The best memory efficiency should be provided by combining lazy linking with multi-block traces and accurate trace selection. Profile-based trace selection and dynamic trace selection combined with lazy linking have these characteristics and offer the best memory efficiency. A 20% memory savings can be achieved with these path selection strategies.

3.2.3 Performance

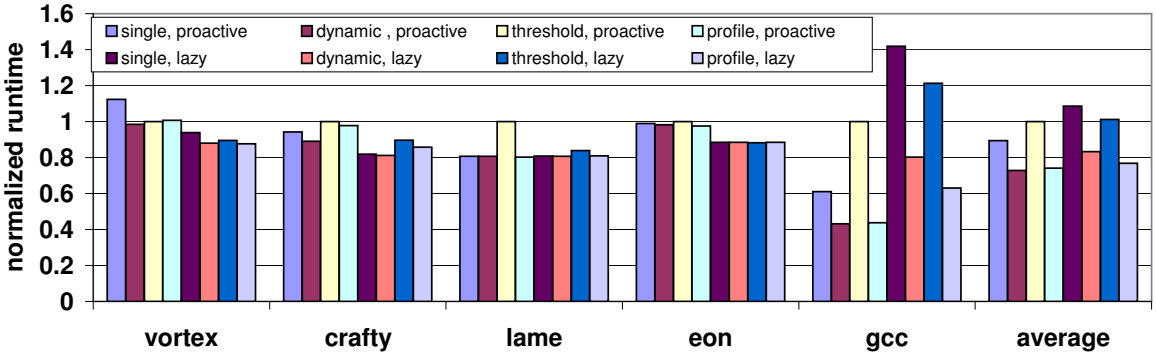
The normalized performance of the short-running, medium-length and long-running benchmarks are shown in Figure 3.8(a), Figure 3.8(b) and Figure 3.8(c) with half the total memory demand as the limit. We validated our results further by placing a uniform memory limit of 512 KB on all the benchmarks. Figure 3.9 shows that similar results were obtained with a uniform memory limit.



(a) Normalized performance of short-running benchmarks.

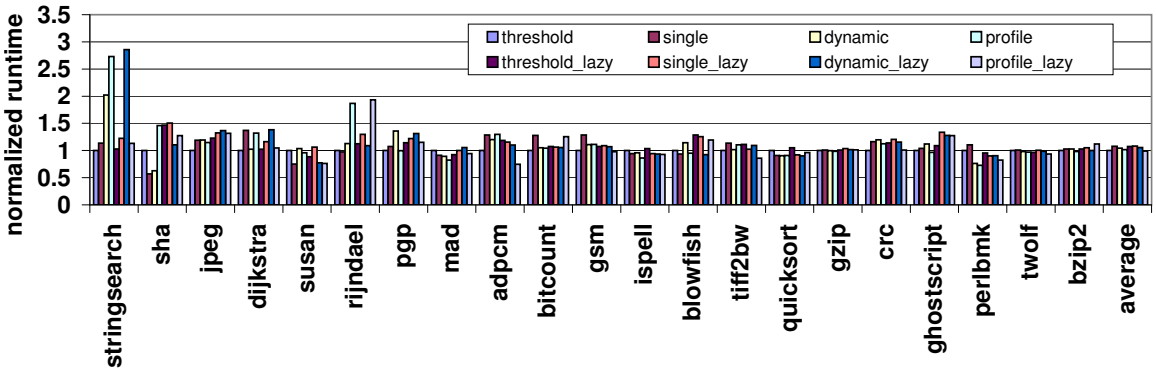


(b) Normalized performance of medium-length benchmarks.

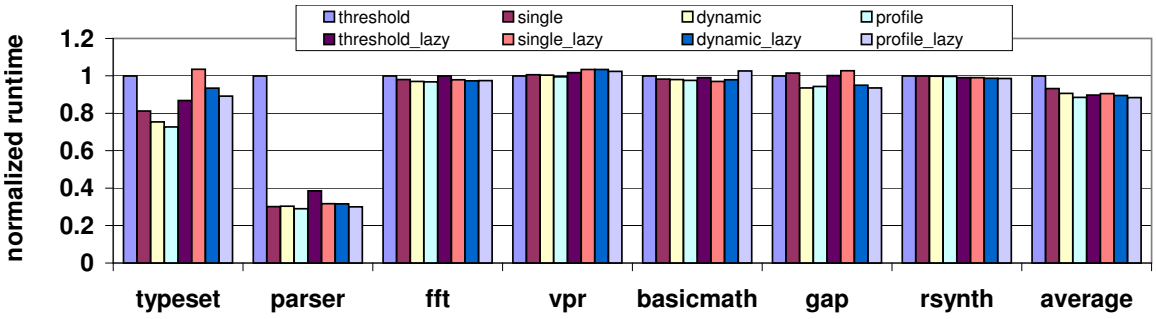


(c) Normalized performance of long-running benchmarks.

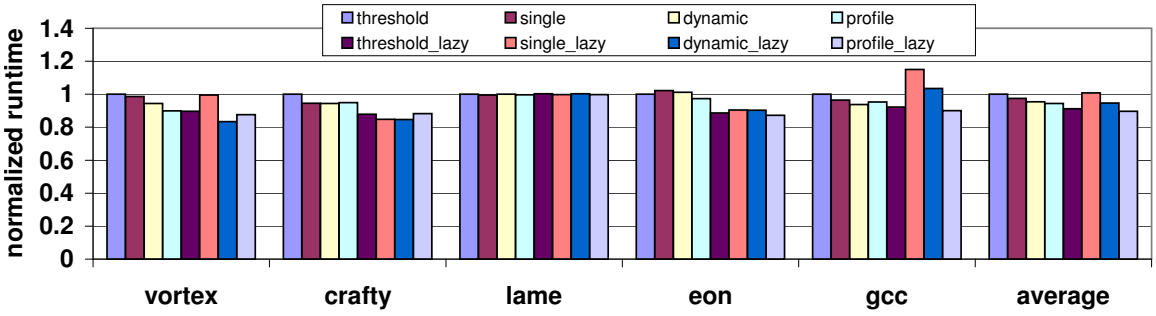
Figure 3.8: Normalized performance of different path selection strategies for the different benchmark categories, with `threshold-based selection, proactive linking` as the baseline. The memory limit is set to half of the total memory demand for each benchmark.



(a) Normalized performance of short-running benchmarks.



(b) Normalized performance of medium-length benchmarks.



(c) Normalized performance of long-running benchmarks.

Figure 3.9: Normalized performance of different path selection strategies, with `threshold-based selection, proactive linking` as the baseline and 512 KB as the uniform memory limit.

Short-running benchmarks do not get much time to amortize translation overheads by executing in the code cache, resulting in no clear winner among the different path selection strategies in this category. Also, the average performance difference among the different path selection strategies in the short-running benchmark category is minor. As we move into the medium-length and long-running benchmark categories, however, we see clearer patterns.

We first compared the strategies by fixing the linking strategy and varying the trace selection strategy. For lazy linking, multi-block traces (dynamic, profile-based and threshold-based selection) perform better than single-block traces because of greater code cache locality, fewer context switches and better memory efficiency leading to fewer flushes. As shown in Figure 3.10, single-block traces have the highest fraction of the code cache occupied by auxiliary code, leading to lowest code cache locality. Between speculative (profile-based and threshold-based) and non-speculative (dynamic) trace selection, speculation has a slight advantage when it is highly informed (as in profile-based) due to the fewer number of context switches required. Profile-based selection is the best followed closely by all the other trace selection strategies in the medium-length benchmark category. However, in the long-running benchmark category, profile-based selection is the best followed closely by dynamic selection only. These two selection strategies perform well in all the benchmark categories and outperform the other trace selection strategies by increasing margins as the benchmark length increases. The profit margin increases because as the execution time increases, there is more time to amortize translation overheads and the true benefits of the different trace selection strategies become clearer. The proactive linking strategies present a distribution similar to the lazy linking strategies.

Next, we compared the strategies by fixing the trace selection strategy and varying the linking strategy. Lazy linking clearly performs better than proactive linking due to better memory efficiency and fewer flushes. However, this is not the case with `gcc`, because the working set of `gcc` changes frequently during program execution. In this situation, the extra context switch overhead of lazy linking cannot be amortized because the working set changes rapidly. The large performance gains for `gcc` skew the average although for all other benchmarks in the long-running category, lazy linking is the clear winner.

The following is a summary of the performance evaluation:

- As in the case of memory efficiency, lazy linking combined with multi-block traces using accurate trace selection should be the best. Dynamic and profile-based trace selection strategies combined with lazy linking fulfill these characteristics and provide the best runtime performance. The best schemes improve performance by 5% for the medium-length category and by 20% for the long-running category.
- When considering the code cache size only, usually proactive linking is preferred for performance. Therefore, we see again that ignoring data structures leads us to misleading conclusions as lazy linking is preferred for holistic memory efficiency.
- The path selection strategies with the best memory efficiency have the best performance.
- Code cache locality and context switch overhead are not as important as memory efficiency because dynamic selection has less code cache locality (as shown in Figure 3.10) than profile-based selection. Also, dynamic selection, being non-speculative, carries more context switch overhead than profile-based selection. Yet dynamic selection performs almost as well as profile-based selection.

3.2.4 Discussion

We have demonstrated that when considering the code cache in isolation, we reach misleading conclusions about the memory efficiency of DBTs. In addition, we have shown that the total memory demand is not a simple function of the code cache size. Therefore, the space allocated for data structures has to be evaluated in addition to the code cache.

We also found that the linking strategy has the most effect on memory efficiency followed by the number of basic blocks in the trace. Both strategies present tradeoffs among the memory components. Non-speculation (dynamic selection) or well-informed speculation (profile-based selection) does not result in much difference in memory efficiency, although uninformed speculation (threshold-based selection) degrades memory efficiency considerably.

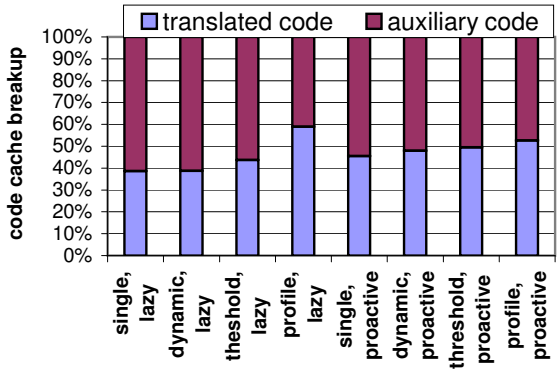


Figure 3.10: The division between translation code and auxiliary code within the code cache.

The linking strategy has the most effect for performance as well. Lazy linking performs better than proactive linking, although proactive linking has fewer context switches. This phenomenon occurs because lazy linking has better memory efficiency and flushes less often, outweighing the performance overhead of context switches and showing that memory efficiency is the most important factor influencing performance. As in the case of memory efficiency, the next most important factor is the number of basic blocks in trace. The better code cache locality of multi-block traces provides a slight advantage. Non-speculation (dynamic selection) vs. well-informed speculation(profile-based selection) has the least impact on performance, showing that context switch overhead is the least important factor.

Based on experimental results, we recommend the use of multi-block traces that are formed by selecting contiguous basic blocks as long as the control flow remains sequential. Whether the control flow remains sequential should be determined non-speculatively or using highly-informed speculation. The traces should be linked lazily. Therefore, dynamic selection or profile-based selection combined with lazy linking are the path selection strategies of choice for memory-constrained scenarios. With profile-based selection, a profiling run must occur. Our experiments show that dynamic selection can get close to profile-based selection without the profiling run. Therefore, our final recommendation is dynamic selection with lazy linking. Dynamic selection with lazy linking improves memory efficiency by 20% and performance by 5-20%.

3.3 Summary

Path selection strategies create interactions among memory demand components, and the code cache size in isolation cannot predict the holistic memory demand because the relative memory demands of the code cache and the data structures vary with the path selection. Therefore, it is important to improve the combined memory demands of the code cache and the data structures. The best holistic memory demand is offered by dynamic selection and lazy linking. The best performance is offered by same path selection strategy. Dynamic selection with lazy linking improves memory efficiency by 20% and performance by 5-20%. Dynamic selection entails non-speculative formation of multi-block traces from basic blocks that appear sequentially in the guest application. The linking strategy has the most impact and lazy linking beats proactive linking. The increase in code cache size due to lazy linking is outweighed by the decrease in data structure size. The number of blocks in a trace has the next greatest impact and multi-block traces beat single-block traces. The increase in duplication due to multi-block traces is outweighed by the decrease in data structures and auxiliary code. The amount of speculation involved has the least impact as long as the speculation is highly accurate because non-speculative dynamic selection performs as well as speculative, profile-based selection. Holistic memory demand outweighs performance impacts due to context switches and code cache locality. Therefore, it is beneficial to improve the holistic memory demand of path selections in memory-constrained environments.

The results obtained are fairly general because our path selection design choices are comprehensive and are not specific to any platform or workload. We demonstrate that path selection has to be carefully carried out for both holistic memory efficiency and performance. Code cache-oriented traditional approaches are not sufficient. Additionally, we have experimentally selected a path selection strategy for a very common execution environment.

Chapter 4

Code Cache Exit Stub Optimization

We found in Chapter 3 that path selection may have opposing impacts on the different components of memory demand. Therefore, we designed a balanced path selection for holistic memory efficiency and performance. The next research step was to determine, given some path selection strategy, whether it is possible to optimize the components of memory demand further. However, the requirement in this step is that such optimizations should not impact the path selection strategy and should not produce interactions among the different components of memory demand. Exit stubs are a component of memory demand that fit these requirements. First, exit stubs exist only to handle trace exits and no other component of memory demand depends on them. Secondly, we found that exit stubs occupy a major percentage of the code cache and therefore have great potential for optimization.

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

To demonstrate all the challenges to exit stub optimization, the structure of an exit stub is shown in Figure 4.1. An exit stub consists of both code and data. The code is responsible for saving the guest application context, loading the address of exit stub data so that the translator can locate the

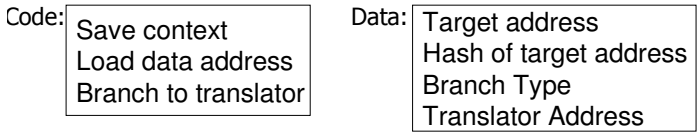


Figure 4.1: Exit stub structure. Exit stubs contain both code and data.

data, and branching to the translator. The data contains arguments to the translator such as the target address of the trace exit. It may also contain arguments such as a hash of the target address to quickly search the code cache directory for the target trace. The arguments also indicate the branch type, for example, whether it is direct or indirect. Finally, the exit stub stores the translator entry address because the branch to the translator has a large offset and is therefore, indirect.

It is challenging to reduce exit stub memory demand by reusing because exit stubs are specialized for the trace exits they handle. While the code is standard across exit stubs, some of the data corresponds to the particular trace exit. The target address of the trace exit is an example of such data. Specialization makes it difficult to reuse exit stubs among trace exits.

Another challenge is presented by the performance optimization measures applied to exit stubs. Performance optimization efforts strive to reduce the dynamic instruction count of exit stub execution by replicating exit stub code. In contrast, factoring exit stub code to a common routine increases the instruction count (details given in Section 4.2.1. Even the storing of the hash of the target address is a measure to reduce dynamic instruction count at the expense of memory. Also, exit stub code and exit stub data for all the exit stubs of a particular trace form two separate groups in the code cache, to improve instruction and data cache efficiency. However, such an arrangement requires extra code to locate the data for each exit stub, as shown in Figure 4.1.

Finally, all data needed by an exit stub are preferably stored within the exit stub, to simplify code cache management. In Figure 4.1, although the translator entry address can be stored once in the code cache, it is replicated to simplify management.

We use several approaches to reduce the exit stub memory footprint. First, we reduce the code size of exit stubs by factoring out common code such as saving the guest application context. Factoring presents a tradeoff between memory efficiency and dynamic instruction count. However, in memory-limited situations, the improvement in memory efficiency reduces flushing activity which

improves performance. The loading of the data address is omitted by storing the exit stub code and data together. Here also, the improved flush activity improves performance. Next, we improve the exit stub data size. The data size is improved by removing derivable data such as the hash of the target address and calculating the hash each time it is needed. The tradeoff reduces memory demand at the expense of increased instruction count. The branch type is omitted by specializing a translator entry for each branch type. This measure increases complexity by increasing the number of translator entry points. These translator entry addresses are stored at a coarser granularity than at per-stub granularity. This measure also increases complexity because the different translator entry addresses are stored at different and arbitrary offsets from each exit stub. We also reduce the exit stub count by identifying sharing opportunities. We even delete or avoid compiling exit stubs, when applicable. This last measure cannot be applied if trace exits ever get unlinked. However, the benefits of such a measure is still worth studying because many executions obey this restriction.

In Section 4.1, we provide an overview of exit stubs. In Section 4.2, we describe the techniques developed for reducing the size of the code cache by optimizing the exit stubs. The experimental evaluation of our techniques is presented in Section 4.3. We summarize in Section 4.4.

4.1 Exit Stubs

Figure 4.2 shows the exit stub code for two different DBTs – Pin and DynamoRIO. In line 1 of Figure 4.2(a), the guest 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 exit stub data. In line 4, the program counter is loaded with the translator handler's address, which is essentially a branch instruction.

Figure 4.2(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.

The exit stub of Pin is applicable to both single-threaded and multi-threaded guest applications.

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

(a) Pin's exit stub code for ARM

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

(b) DynamoRIO's exit stub code for x86

Figure 4.2: Examples of exit stub code from different DBTs

(a) Intermixed stub code and data

(b) Separated stub code and data

Figure 4.3: Two arrangements of stubs for a given trace in the code cache

Pin stores the guest application context at an offset from the stack pointer. Since each thread has its own stack pointer, the saved thread contexts will never overwrite each other. However, DynamoRIO stores `eax` at a predefined memory location. If using a thread-shared cache, the `eax` values for the different threads will overwrite each other. Therefore, DynamoRIO exit stubs do not support thread-shared caches.

Figure 4.3 shows two possible layouts of stubs for a given trace. In Figure 4.3, stubs 1 and 2 are exit stubs corresponding to a single trace. The code and data for each stub may appear together as shown in Figure 4.3(a), or they may be arranged so that all the code in the chain of stubs appear separately from all the data in the chain of stubs, as shown in Figure 4.3(b). The layout in Figure 4.3(b) is better for cache efficiency. As we show in Section 4.2.1, the first layout conserves more space.

Table 4.1: Percentage of code cache consisting of exit stubs.

DBT	**Exit Stub Percentage**
Pin	66.67%
DynamoRIO	62.78%

Table 4.1 shows the space occupied by exit stubs in Pin and DynamoRIO [15]. 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 was obtained using log files generated by the two DBTs. The data for DynamoRIO[9] was calculated from space savings achieved when exit stubs are moved to a separate area of memory.

4.2 Methodology

In this section, we describe our approaches for improving the memory efficiency of DBTs by reducing the space occupied by stubs. In Section 4.2.1, we describe the techniques to reduce exit stub code size and exit stub data size. Section 4.2.2 describes our technique to reduce exit stub count by sharing exit stubs among trace exits. Section 4.2.3 describes our technique to reduce exit stub count by deleting unreachable exit stubs. Section 4.2.4 describes our technique to reduce exit stub count by avoiding compilation of exit stubs when applicable. We show the impact of our optimizations on Pin's exit stub for ARM, wherever applicable.

4.2.1 Exit Stub Size Reduction (R)

In the exit stub size reduction (R) scheme both code and data size are optimized. Code space is saved by identifying the common code in all stubs. We use a common routine for saving the context and remove corresponding instructions from the stubs. However, the program counter has to be saved before entering the common routine. Such a measure is needed because the program counter is the single register that will get modified upon branching to the translator. Also, saving of the program counter enables the translator to locate the exit stub using the saved PC value, the advantage of which is explained in the following paragraphs. Figure 4.4(a) shows the code in the stub after this optimization. Here, we take advantage of the fact that the ARM `st` instruction allows the store

target to be calculated and the store to be executed together. Using the `stm` instruction required the target to be calculated in a separate instruction. Therefore, the size of the exit stub code is reduced by one instruction. However, saving the rest of the guest application context now requires two more instructions to adjust the stack pointer once again and to save the remaining registers using `stm`. Thus, the dynamic instruction count effectively increases. Factoring of common code is a simple technique that has been implemented in some form in systems (e.g., DynamoRIO), already.

To further improve the code size, we adhere to the layout in Figure 4.3(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 Figure 4.3(b).) The stub's start address is known from the program counter saved in the context. Here we see the advantage of saving the program counter. Some instruction and data cache locality is sacrificed in adopting this exit stub layout. The resulting exit stub code is shown in Figure 4.4(b).

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

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

This scheme is applied to all exit stubs corresponding to direct branches and calls which constitute a majority of the exit stubs. Flushing and invalidations can be handled by Scheme R as we

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

(a) Structure of stub after factoring out code to save the guest application context.

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

(b) Structure of exit stub after altering the layout of exit stub code and data.

Figure 4.4: Structure of exit stub code after applying code size optimizations.

```
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
```

Figure 4.5: Algorithm for using target address specific stubs.

have modified only the stub structure. Our mechanism is independent of the flushing strategy and the number of threads being executed by the program.

4.2.2 Target Address Specific Stubs (TAS) or Sharing Exit Stubs

While exit stubs are specialized for the trace exit they handle, many exit stubs are identical because their corresponding trace exits target the same program address. For our scheme, target address specific stub translation (TAS), we ensure that trace exits requesting the same target address use the same exit stub.

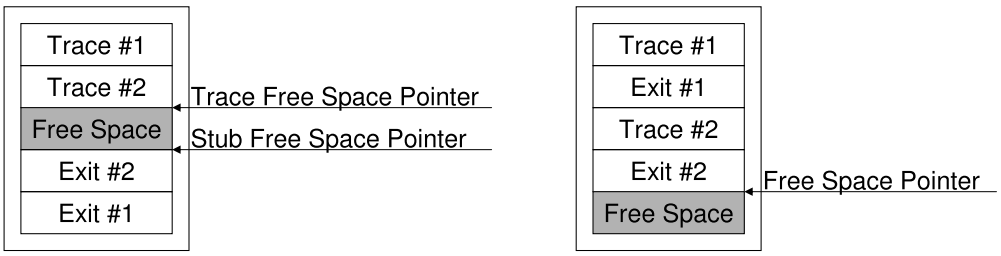
Figure 4.5 shows the algorithm used in this scheme. The target address of each trace exit is examined. The stub corresponding to the target address of each trace exit is searched in line 3. If the required exit stub exists, it is designated as the exit stub for the trace exit in line 4. Otherwise a new 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, we partitioned the code cache into 64KB cache

blocks and reused within cache blocks. These partitions are the *blocks* in line 3 of Figure 4.5. 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 flushing out exit stubs while their trace exits still remain.

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 before the trace being translated is inserted (for example, due to reaching the memory limit), 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.

In this scheme, if traces are ever invalidated, their corresponding exit stubs may still be targeted by other traces. This fact does not present a problem since invalidated traces are not overwritten (to simplify code cache management) and their exit stubs can continue to function. However, exit stub data may contain arguments other than the target address, based on the service being offered by the DBT. For example, the exit stub may contain an encoding of the calling context of the trace and certain functions are invoked based on the calling context. In this case, trace exits sharing the same target address may not have identical exit stubs. Sharing may now have to be carried out at a finer granularity i.e., exit stubs with all matching data are shared. This arrangement will require both the target address and other data to be remembered to determine whether a trace exit can share an exit stub. Alternatively, the exit stub may be split up into shareable and non-shareable parts. However, the memory overhead of splitting up the exit stub into two parts requires adding branch instructions to jump from one part of the stub to the other. Such additions may not lead to net memory gains for a highly optimized exit stub. Additionally, there are some data structure requirements for this technique. The available exit stubs need to be recorded in data structures. However, these data structures are not large because only the exit stubs in the last cache block (where traces are being inserted) need to be remembered. Also, the number of exit stubs to be remembered is of the order



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

(b) Traces and stubs contiguous in code cache

Figure 4.6: Two typical arrangements of traces and exit stubs in code cache

```
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
```

Figure 4.7: Algorithm for deletion of stubs

of the number of traces rather than of the order of the number of trace exits.

4.2.3 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. Figure 4.6 illustrates this concept. Assume the code cache is filled as in Figure 4.6(a) (such an arrangement may be used for better code cache locality). If the branch corresponding to stub 1 is linked and stub 1 is at the top of the stack of exit stubs, stub 1 can be deleted. Additionally, if the stubs are laid out as in Figure 4.3(a), then both the code and data of the stub can be deleted. Here we see another utility of choosing the layout in Figure 4.3(a). The layout in Figure 4.3(a) would only allow us to delete the code. For the code cache arrangement of Figure 4.6, the exit stub at the edge of free space can be deleted if the trace exit corresponding to it gets linked.

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 could have maintained a linked

list of holes formed by storing each node of the linked list in the holes themselves. Such hole management would not require any extra space for data structures. However, filling up these scattered holes with traces will imply that the traces are being stored in an increasingly scattered fashion. Such traces are difficult to manage and are therefore, not pursued.

Figure 4.7 shows our stub deletion algorithm. A code cache has a pointer to the beginning of free space (as shown in Figure 4.6) to determine where the next insertion should occur. If the condition in line 1 of Figure 4.7 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.

There are alternatives to complete stub deletion, such as stub compression and stub retranslation. We did not explore compaction and retranslation of stubs for this dissertation. 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. Retranslation and decompression of stubs on the other hand, creates a sudden demand for free code cache space. This is not desirable because retranslation / decompression of stubs will be needed when cache eviction occurs and cache eviction usually means that there is a shortage of space.

4.2.4 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

```
1  for every trace exit
2      targetaddr = target 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
```

Figure 4.8: Algorithm for avoiding compilation of traces

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

Figure 4.8 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.

This scheme suffers from the same limitation as Scheme D in that individual deletions may not be allowed. Also, this scheme interacts with the linking policy. Lazy linking does not link trace exits at the time of translation. Therefore, a lazy linking DBT will not be able to leverage this scheme. However, this scheme is suitable for proactive linking DBTs.

4.3 Experimental Results

We evaluated the memory efficiency and performance of our proposed techniques. As a baseline, we used Pin [69] running on an ARM architecture. We implemented our solutions by directly modifying the Pin source code. This optimization was included in both the baseline and our modified

versions of Pin.

For the experiments, we ran the SPEC2000 integer suite 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 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 Figure 4.6(a) and lays out stubs as shown in Figure 4.3(b).

All the techniques described in Section 4.2 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.

4.3.1 Memory Efficiency

The first set of experiments focused on the memory improvement of our approaches. Figure 4.9 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.

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 savings 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.

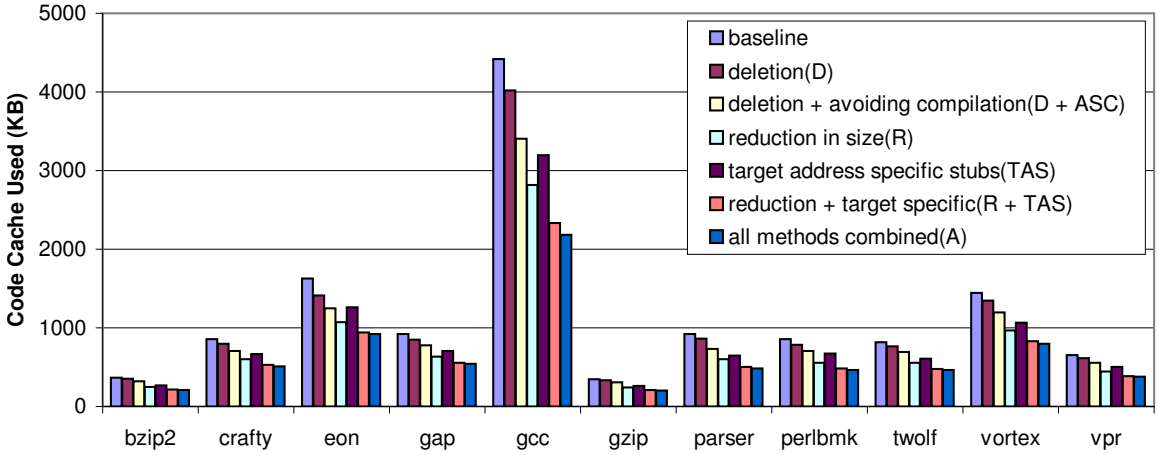


Figure 4.9: Memory usage (reported in kilobytes) of Pin baseline (leftmost bar) and after incorporating our optimizations (rightmost bars).

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.

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 4.2 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%.

4.3.2 Performance Evaluation

In this section, we evaluate the performance of our approaches without placing a memory limit because D and ASC are also being evaluated. Figure 4.10 shows the normalized performance of our schemes with respect to the baseline version. Scheme D has almost the same performance as

Table 4.2: 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%

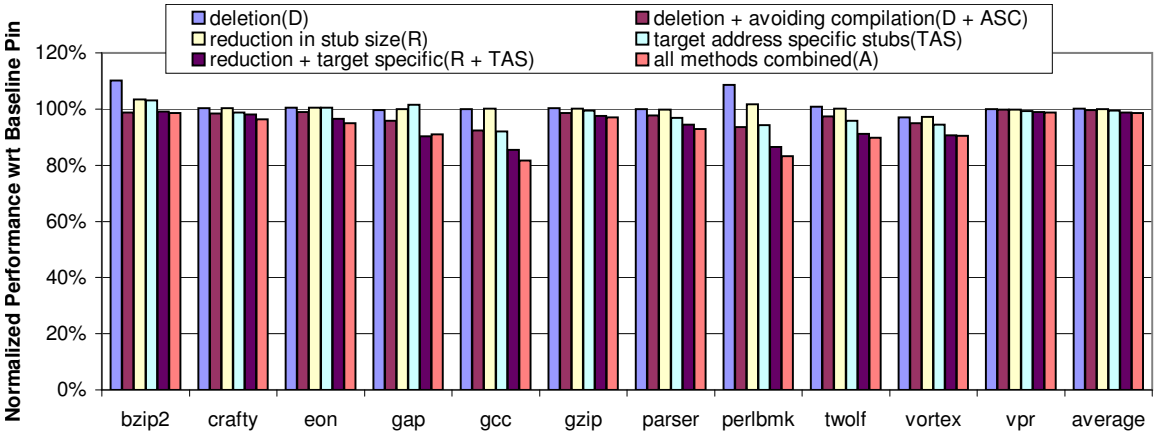


Figure 4.10: Performance of proposed solutions as normalized percentages. (100%) represents the baseline version of Pin and smaller percentages indicate speedups.

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.

4.3.3 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`.

4.4 Summary

In this chapter, 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

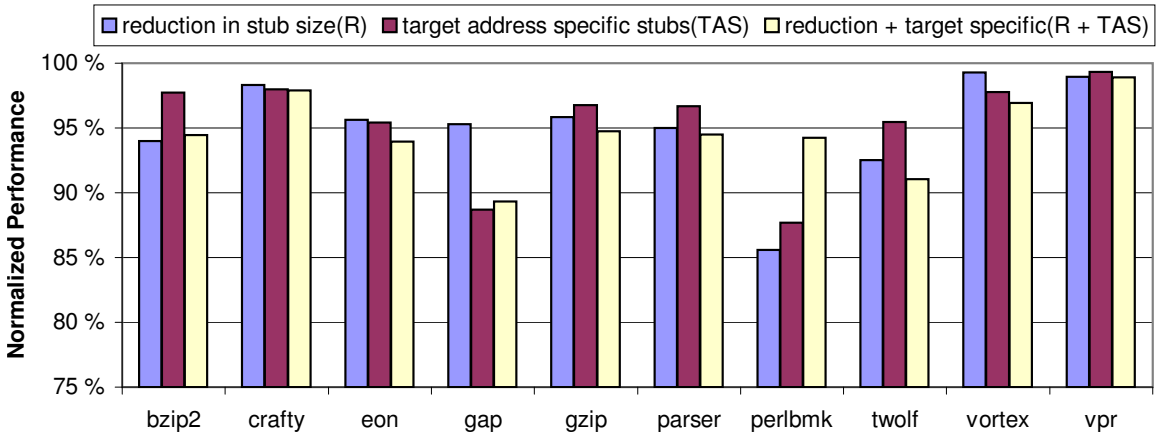


Figure 4.11: Performance of `exit stub reduction` and `target address specific stub translation` with cache limit of 20 blocks (1280 KB).

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.

Chapter 5

Generational Cache Flush

After a translated trace has been executed for the last time, it is dead. Dead traces occupy space in the code cache unless they are evicted. Our experiments show that on the average, 60% of code has a short lifetime of less than 10% of the total execution time. In this chapter and the next, we selectively flush short-lived traces to improve performance in a memory-limited situation. The goal of the research described in these two chapters is to develop efficient methods for dynamically identifying dead traces and evicting them.

There are several challenges to selective flushing. The first challenge in evicting traces is dynamically identifying which traces are dead. Heuristics must be used to approximately determine the lifetime of traces, and the mechanism of profiling for these heuristics has to be efficient because the profiling time will be credited to the total execution time.

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. Code cache management becomes complicated in this scenario.

Previous work eliminated dead traces by allowing them to mature for some time and then monitoring them for execution. If not executed during the monitoring period, traces were presumed to be dead. Meanwhile, the other traces were promoted to long-lived status. In this chapter, we explore this time-based heuristic for an embedded environment.

We also found that lifetime and execution count of traces is strongly related, motivating us to use a heuristic based on the trace execution count (which does not depend on time elapsed) and

to compare the two heuristics. Our experiments also show that a high percentage (about 90%) of long-lived traces have a high execution count while only a small percentage of short-lived traces (about 20%) have a high execution count. Therefore, preserving only long-lived traces is beneficial because such traces occupy a smaller code cache area and yet, cover a greater part of the entire execution.

To manage the code cache, previous work used a generational cache consisting of three parts - nursery, probation cache, and persistent cache-promoting a trace from the nursery to the probation cache and then to the persistent cache as it executes [54]. 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 traces until it is determined whether the trace is long-lived. traces are promoted to the long-lived area based on our heuristics. After several promotions, the temporary area should mostly contain dead traces 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 traces gets promoted to the permanent area. Also, performance may be enhanced by such evictions because the spaces created by eviction will be filled again with traces which are currently live. Packing long-lived traces into a smaller area offers improved instruction locality.

Apart from these optimizations to reduce retranslation time and manage the code cache, we also explored optimizations to reduce the book-keeping overheads of flushes. The first optimization is based on the observation that flushing requires that all incoming links to a trace be removed. Since temporary area flushes are anticipated to be frequent, unlinking may present a considerable overhead. We explored the trade-off of allowing and disallowing links to traces in the temporary code cache. Allowing links speeds up execution in the temporary code cache but slows down flushes. Disallowing links requires the DBT to search the code cache address of each trace to be executed, and also generate the trace if it is not already in the code cache.

Another optimization is deciding whether to promote traces to the permanent area early or late.

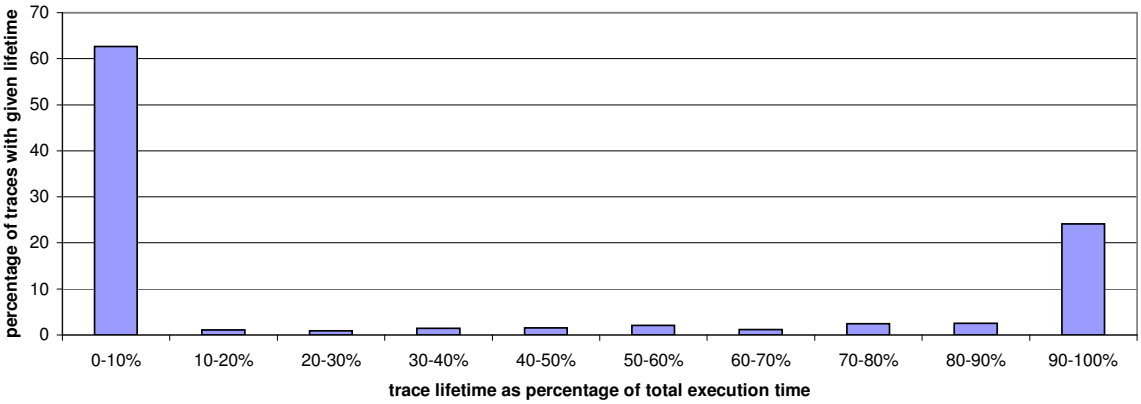


Figure 5.1: Percentages of traces with certain lifetimes. 0-10% and 90-100% are by far the most major categories.

In early promotions, traces are promoted as soon as possible. In late promotions, the decision about each trace in the temporary area is deferred until a flush to determine whether it is eligible for promotion. Early code promotions send long-lived traces to the permanent area as soon as possible and reap more benefits from code cache locality in the permanent area. However, early promotion requires the execution in the code cache to be probed more intensely, possibly slowing it down. We explored the trade-offs of both early and late promotions.

In Section 5.1, we discuss how trace characteristics motivate this work and their relationship to the heuristics used. We describe our code cache organization and code cache management techniques in Section 5.2, the actual implementation in Section 5.3, and experimental results in Section 5.4. We summarize in Section 5.5.

5.1 Profiling Heuristics

In this section, we discuss the trace characteristics that motivated our profiling heuristics. We evaluated the lifetime and execution count characteristics of code regions for the SPEC2000 integer suite [55] and the MiBench embedded benchmark suite [50]. We used Pin 2.0 for XScale [69] to host the execution of these benchmarks on a iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19.

Through experimentation, we found that the majority of traces live for a small fraction of the

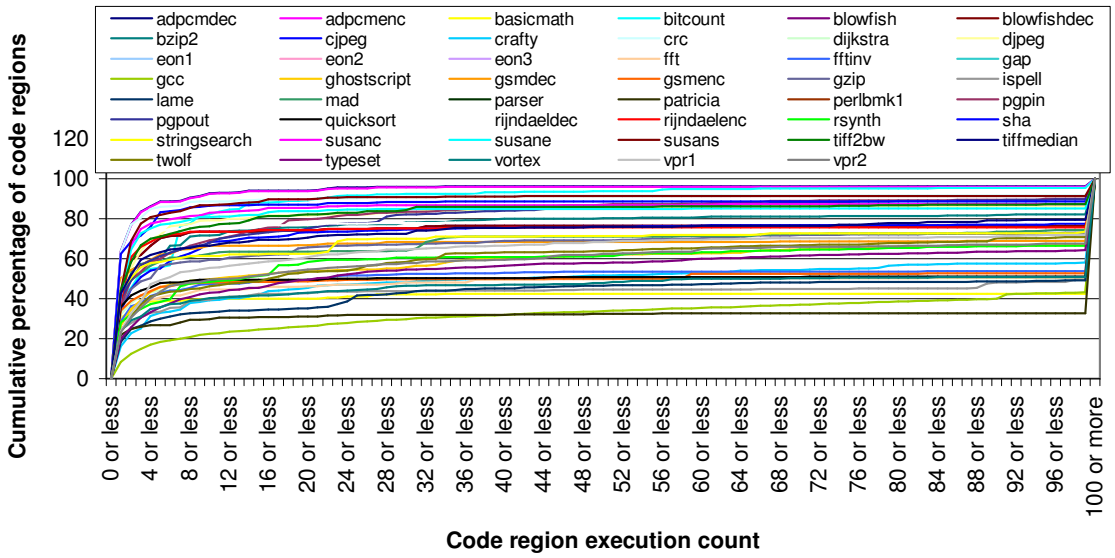


Figure 5.2: Percentages of traces with execution counts within certain thresholds. Most traces have execution counts of five or less.

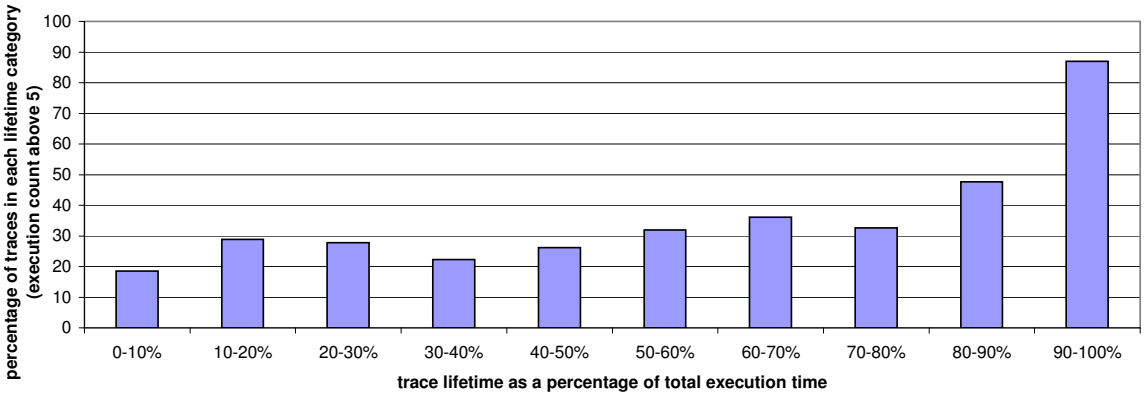


Figure 5.3: Percentages of traces in each lifetime category, which have high execution counts of above five. Most long-lived traces are included while few short-lived traces are included.

total guest application execution time. Figure 5.1 shows the percentage of traces in each lifetime category. The graph shows that on average, 60% of traces live less than 10% of the guest application execution time, motivating us to detect these dead traces and reclaim the space occupied by them.

Intuitively, a trace in the code cache should be allowed to remain in the code cache for 10% of the guest application lifetime and then monitored for execution to determine whether a trace is long-lived. However, it is difficult to determine 10% of the guest application time before a program

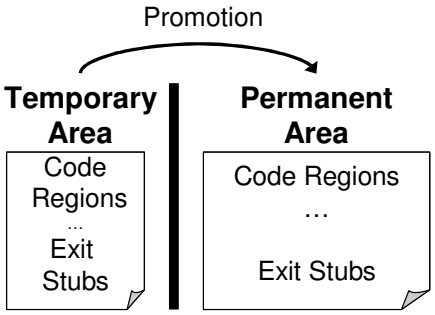


Figure 5.4: Code cache generations. The temporary area is size limited.

terminates. So time intervals between some predetermined events are used to decide when to start monitoring. This observation forms the basis of one of our heuristics, the time-based heuristic.

We also examined the relationship between the execution count of a trace and its lifetime. First, we examined the execution counts of all traces in a guest application. Figure 5.2 shows the percentage of traces that execute a certain number of times. The graph has a step around five for all the benchmarks investigated. The step implies that in the context of embedded applications, five can be considered a high execution count.

Next, we explored the percentage of traces in each lifetime category that has an execution count of more than five. Figure 5.3 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 traces in the 0-10% (20% traces have a high execution count) category. This motivated us to set an execution count threshold of five to determine whether a trace is long-lived. This heuristic is the execution count-based heuristic.

5.2 Code Cache Management

The graph in Figure 5.1 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. In this section we describe our code cache management schemes, which includes our code cache organization and optimizations that we explored. These optimizations include whether to allow links into the temporary code cache and whether to use early or late promotions.

5.2.1 Restructuring the Code Cache

We use a generational code cache divided into the temporary and permanent areas as shown in Figure 5.4. 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 traces and we flush and reuse it. We set a size limit on the temporary code cache to prevent memory expansion due to dead traces and to initiate flushes to reclaim occupied space. Such a generational code cache organizations is a natural outcome of the fact that traces fall into two major lifetime categories: 1) alive less than 10% of the total execution time and 2) alive more than 90% of the total execution time. The generational organization also eliminates fragmentation and improves code cache locality by grouping together traces of the same type.

Memory requirements of code caches are expected to decrease in this scheme as a small percentage of traces are long-lived, making the permanent area small. The temporary area is flushed 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 traces, flushing cache areas and in some cases, retranslating evicted traces, will be incurred.

5.2.2 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 traces 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 to the temporary area may present a considerable overhead. Therefore, we explore disallowing incoming links to traces 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.

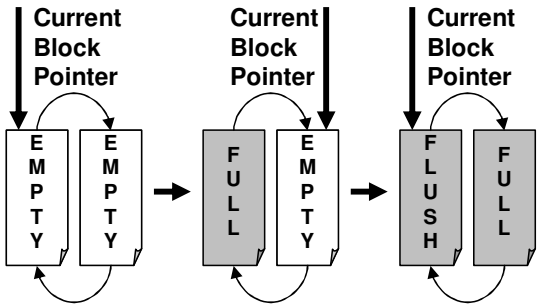


Figure 5.5: Temporary area is divided into cache blocks that form a circular queue and are flushed in FIFO order to provide approximately uniform time to each trace to mature.

5.2.3 Early and Late Promotions

Code promotion can occur early or late. Early code promotions occur as soon as a trace 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 this research.

5.3 Implementation

This section describes the implementation details of the reorganized code cache, the heuristics and the promotion mechanism.

5.3.1 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 increments of 64KB on demand, 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 trace to execute and possibly

complete its lifetime. If the entire temporary area is flushed at once, enough time is not given to the traces that entered the temporary area close to the flush. Therefore, as shown in Figure 5.5, the temporary area is divided into blocks of 64 KB each and one block is flushed at a time in FIFO order.

5.3.2 Profiling

A time-based heuristic and an execution count-based heuristic were motivated in Section 5.1. The time-based heuristic requires that a bit be maintained for each trace to indicate whether it executed while it was being monitored. trace monitoring starts when the cache block containing the trace is next in line to be flushed. In the case of the count-based heuristic, the number of executions of a trace since its translation is tracked by maintaining an integer counter for each trace. We used an execution threshold of five. The monitoring for execution count starts as soon as a trace is inserted into the code cache.

The time-based heuristic aims to determine whether a trace is alive for more than 10% of the total execution time. Since it is difficult to determine how long is 10% of the total execution time, the time-based heuristic approximates by using profile data between two consecutive flushes. However, the time length between two consecutive flushes may not be a good approximation of 10% of the total execution time. Therefore, the time-based heuristic implementation is inaccurate compared to the implementation of the count-based heuristic.

5.3.3 Trace Promotion

trace 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 trace also have to be updated.

When to promote a trace depends on the heuristic. On every execution of a trace, the heuristic value is first updated. Then a decision may be made whether to promote the trace. Figure 5.6 shows the content and location of the instrumentation used to carry out these functions. If links to the trace are disabled, the translator is entered for each execution of the trace. The translator then updates the

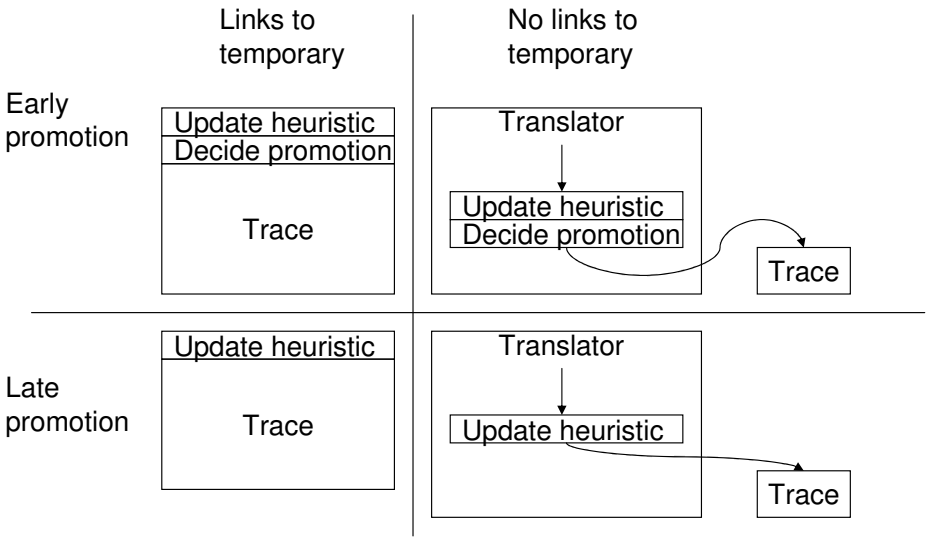


Figure 5.6: The location and content of instrumentation code for each combination.

counter value and decides whether to promote the trace. Therefore, promotions are always early if links are not allowed. Also, instrumentation code does not need to be inserted into the code cache.

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 trace to update the counter value. Late promotion strategy simply requires the heuristic value to be updated every time a trace executes. All decisions based on heuristic values are taken at the point of a flush. However, in the case of early promotions, code to decide whether to promote a trace is also added to the beginning of the trace. Updating and making decisions with the count heuristic required more instrumentation than the time-based heuristic, because in the first case only one bit has to be toggled while in the second case, an integer has to be incremented.

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

There is a performance overhead of updating counters before executing each trace. Also, when

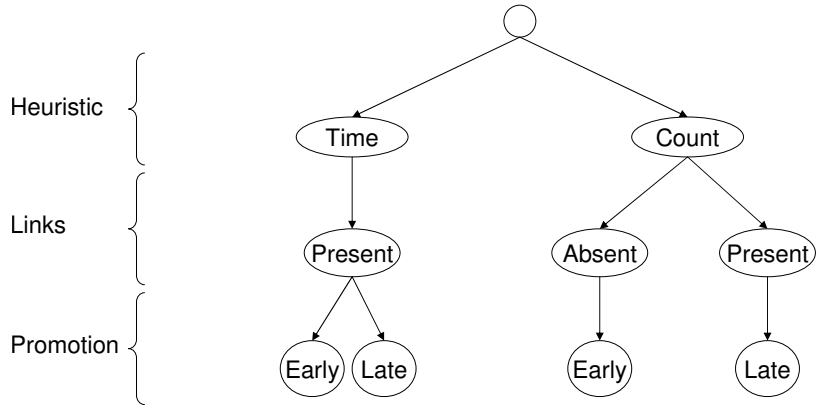


Figure 5.7: The design combinations that we evaluate.

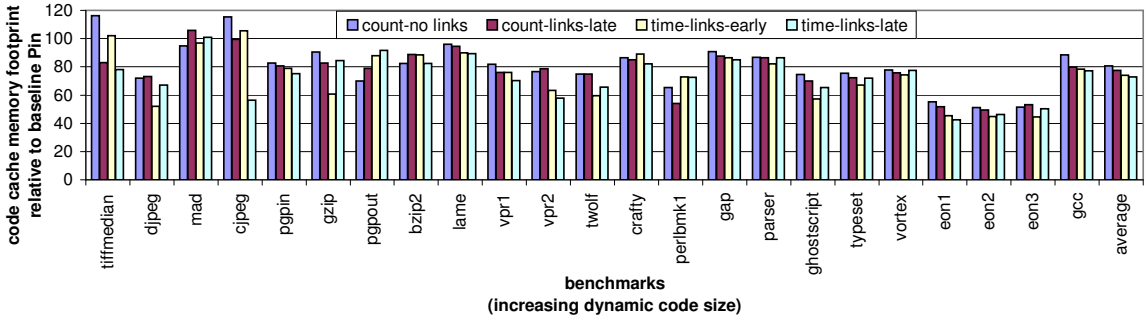


Figure 5.8: Code cache usage as percentage of code cache usage by baseline Pin (benchmarks with code cache sizes greater than or equal to that of `tiffmedian` are shown). Savings for the benchmarks are shown to be 20% on average.

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.

5.4 Experimental Results

This section evaluates the memory efficiency and the performance of our design. We analyze the sources of performance 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. Figure 5.7 depicts the design combinations for which we provide evaluations.

As a baseline, we used Pin 2.0 for the XScale platform [69]. We implemented our solutions by directly modifying the Pin source code. For the experiments, we ran the SPEC2000 integer suite [55] and the MiBench embedded benchmark suite [50] 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.

5.4.1 Memory Efficiency

Figure 5.8 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 5.8 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 traces 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 traces at a time. So late promotion chooses traces from a smaller set and promotes fewer code regions during a flush. The same is true for the time-based heuristic.

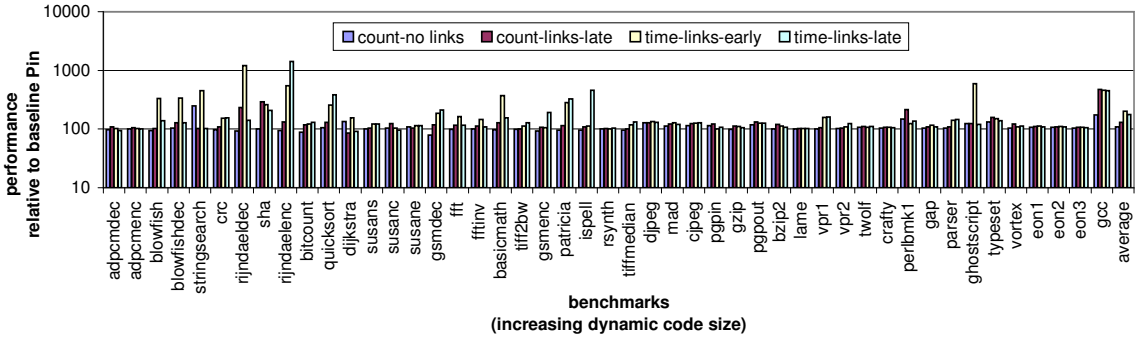


Figure 5.9: Performance with respect to baseline Pin (note the exponential axis). Count heuristic is better in general and the count heuristic with no links is the best.

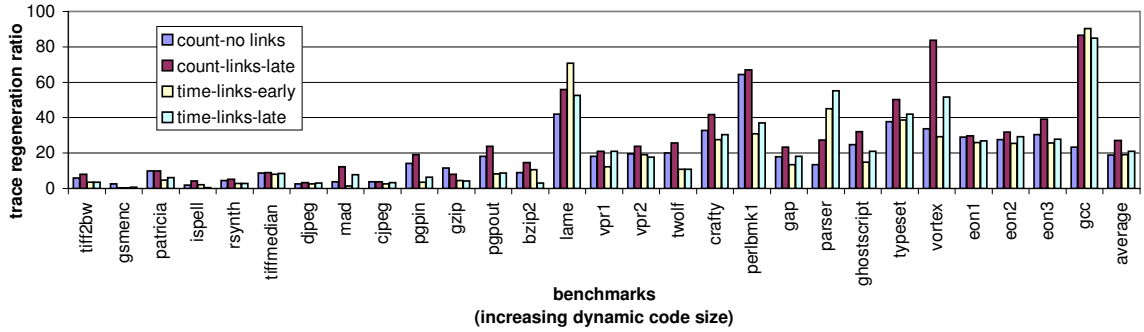


Figure 5.10: Extent of trace retranslation. The time-based heuristic has higher retranslations in most cases, explaining the cause of its worse performance. Benchmarks shown from `tiff2bw` because the ratios are close to zero before it.

5.4.2 Performance Evaluation

Figure 5.9 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 trace is very complicated and occupies considerable space. Executing such code five times for every trace 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 traces and since this scheme promotes the largest number of traces, 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 traces 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 traces and achieves better code locality in the permanent 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 retranslations.

5.4.3 Retranslation Overheads

Trace translation is an expensive task. Therefore, the higher the percentage of trace retranslation, the higher is the performance degradation. Figure 5.10 shows the ratio of the number of code region retranslations to the total number of traces. Although the average percentage of retranslations 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 5.10 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 trace and incurs more retranslation overhead. For the given temporary area size, not enough time elapses between two flushes. A larger temporary area is needed but that will defeat the whole purpose. This is the reason that time-based profiling does not perform well.

5.4.4 Flushing Overheads

Flushes are also responsible for performance degradation. Figure 5.11 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 is due to the fact that the retranslations are lowest in this case.

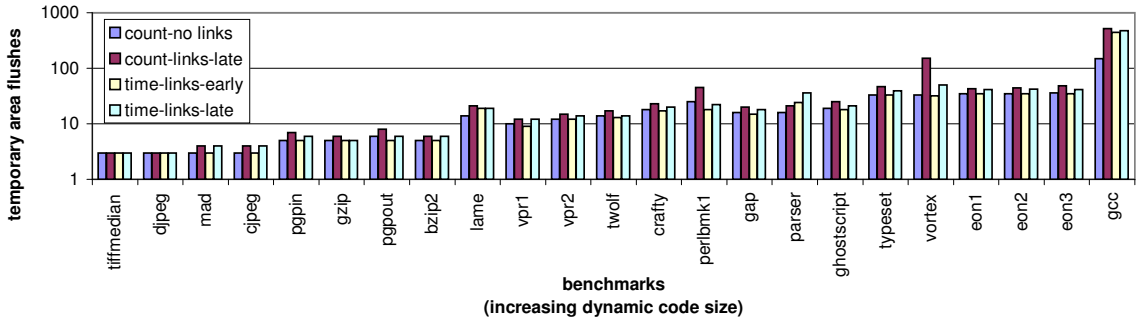


Figure 5.11: Number of temporary code cache flushes. Flushes are influenced by retranslations and also degrade performance. Benchmarks from `tiffmedian` are shown.

5.4.5 Discussion

The results show that the count-based heuristic has better performance than the time-based heuristic and worse memory efficiency. This is due to the fact that the time-based heuristic wrongly eliminates more traces 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, performance degradation due to absence of links continues until trace 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 traces 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 DBT 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 DBT hosted applications can be performance. After some time if the system starts a large graphics application, the focus of the DBT hosted applications can change to memory efficiency.

5.5 Summary

In this chapter 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 for flushing, resulting in different trade-offs. The time-based heuristic achieved better memory efficiency (around 25% code cache reduction on average) but at the cost of more retranslations and degraded performance. The count-based heuristic promoted more traces resulting in less code cache reduction (20% on average), but incurred less retranslations and a lower performance penalty.

The research presented in this chapter was evaluated for single-threaded benchmarks only. However, as mentioned in Chapter 2, there are special considerations for eviction techniques when applied to multi-threaded guest applications. The techniques presented in this chapter can be extended to multi-threaded guest applications. It will be necessary to unlink the traces that are to be evicted from the temporary area, on reaching the high water mark. Therefore, we will have to unlink all traces in all the generations, because we do not know exactly which threads are executing in the traces to be evicted. The impact of such unlinking on performance is unclear. We evaluate flush strategies for multi-threaded guest applications in more depth in the next chapter.

Chapter 6

Unified Cache Flush

In memory-limited situations, the memory demand problem of DBTs is replaced by a performance degradation problem due to flushing. Selectively flushing traces that will not be used in the future, or at least in the near future, can reduce flush overhead. However, selective flushing is challenging. It is difficult to dynamically select which traces to remove because expensive profiling is needed. A by-product of selective flushing is that it complicates code cache management by forming holes (fragmentation) in the code cache. The issue has been further complicated by recent trends towards multicore architectures and multi-threaded programming. Code caches for multi-threaded guest applications are shared by all threads. Thread-shared caches must ensure that no thread is executing in the traces selected for eviction. DBTs fulfill this condition by checking that each thread that was executing in the selected traces exits to the runtime once and is never allowed to return to the selected traces. Such monitoring of threads adds to the complexity of flushing. Another complication is that it is difficult to know which threads were executing in the selected traces in the first place. Due to these challenges, full flush (no selection) has become the standard.

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

We use an approximation of the LRU (least recently used) heuristic to select traces for eviction.

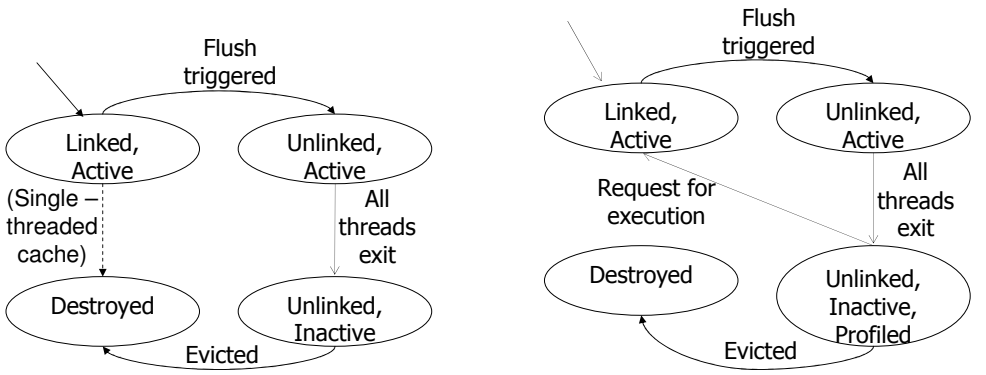
LRU is a popular replacement algorithm and has been successfully applied in many scenarios such as hardware code caches. We group traces that we know have been used least recently, but we do not try to rank within this group. The code cache is treated as a circular buffer and the LRU trace that is closest to the insertion pointer in this circular buffer is evicted. We identify LRU traces by profiling.

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

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

Finally, we solve the problem of managing threads during evictions. Our strategy is to initially select all traces for eviction and use profiling to discard from the eviction set, rather than the other way around. The advantage of selecting all traces for eviction is that we know that all threads need to be monitored and we do not have to identify which threads were executing in the eviction set. We can evict traces after each thread has exited the code cache once. Code regions cannot be re-entered when they belong to the eviction set. traces move in and out of the eviction set. We design data structures and algorithms to efficiently support this mechanism.

There are some fundamental differences between the approaches to solving the selective flushing problem in Chapters 5 and 6. The first difference is that the strategies presented in this chapter are uniformly applicable to both single-threaded and multi-threaded guest applications. The techniques presented in the last chapter could also be extended to multi-threaded guest applications, but



(a) Traditional full flush.

(b) Our proposed selective flush technique.

Figure 6.1: The different states traversed by a cached trace for different flush techniques.

with more difficulty, as discussed in Section 5.5. To facilitate multi-threaded support, we switched from generational code caches to unified code caches. Generational caches are more difficult to use with multi-threaded guest applications because different generations are flushed based on different criteria. Yet, for multi-threaded guest applications, all traces must be unlinked when any generation gets close to a flush. Such unlinking is required because it is not known which threads were executing in the traces to be evicted. All threads are forced out of the code cache to be on the safe side. Alternatively, threads in each generation can be tracked. Tracking can add to the complexity and also degrade performance. Therefore, we used a unified code cache at the expense of fragmentation and greater complexity of code cache management. Secondly, in Chapter 5, we used discrete flush points to evict groups of traces. However, this is eager flushing of traces and evicts many traces prematurely. In this chapter, we do not evict any trace until absolutely necessary. Therefore, we accumulate more profile data at the expense of increased complexity.

We describe our proposed technique in Section 6.1. We discuss design issues in Section 6.2. We evaluate and analyze the performance of our technique in Section 6.3. Finally, we summarize in Section 6.4.

6.1 Selective Flushing

Figure 6.1 shows the conceptual differences between a traditional full flush and our selective flushing technique. In both cases, a trace is initially in the *active* (threads are executing it), linked state. A flush is triggered upon reaching the high water mark. Traces are unlinked although they may continue to be active. When all threads have exited the traces i.e., the traces have become inactive, the two techniques begin to diverge. For a traditional full flush, the traces are immediately evicted. In our technique, traces begin to be profiled for execution. If there is a request for execution of an inactive trace, it is promoted to the active state by dispatching the requesting thread to it. Linking to this trace is again allowed. However, if the trace does not become active, it will eventually get overwritten by new traces. The mechanisms of flush triggering, profiling, promoting and allocating space are described in Section 6.1.1, Section 6.1.2, Section 6.1.3 and Section 6.1.4 respectively. Figure 6.2 shows the successive states of the code cache when our partial flush technique is applied. Figure 6.2(a) is the key for understanding the different code cache states.

6.1.1 Triggering Flushes

A flush is triggered at the high water mark. All traces are unlinked to expedite the exit of threads. In a traditional full flush, all code cache directory entries corresponding to the unlinked traces are discarded, so that these traces cannot be located or re-entered. However, in our technique, there is a possibility of re-entry. Therefore, the code cache directory entries are not discarded. Re-entry must still be disallowed until all threads exit once. Thus, additional tag data structures record whether the trace is old or current. Initially, all traces are in the current state. Upon unlinking, the traces are tagged as old. Old traces cannot be re-entered without first promoting them to the current state. Figure 6.2(b) shows the state of the code cache at a flush trigger point. The code cache contains old traces and some free space.

6.1.2 Profiling Traces

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

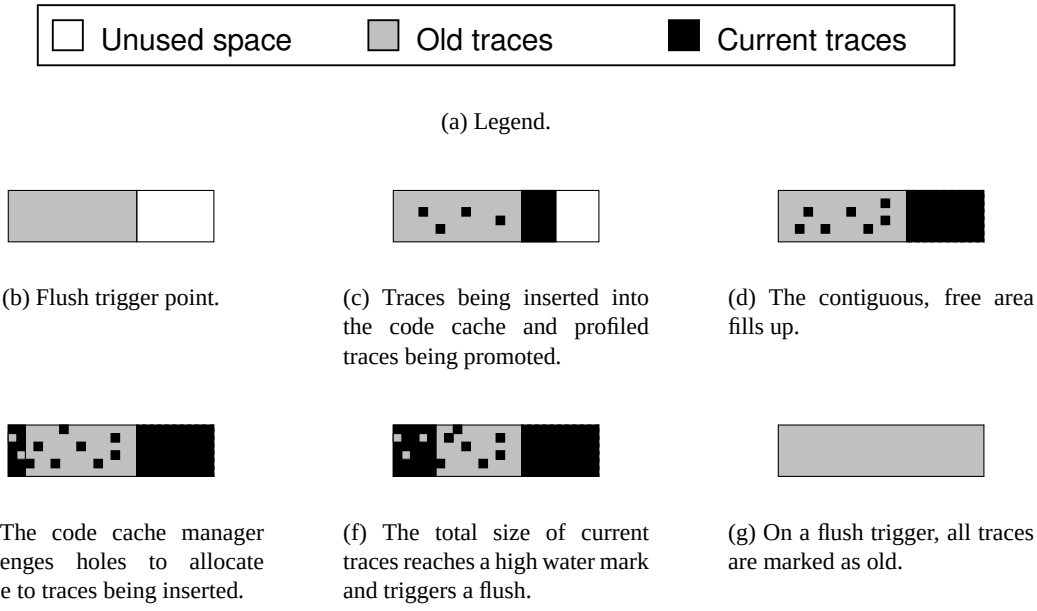
Profiling is enabled by unlinking the traces, which forces the translator to be invoked before every execution of a trace in the LRU set. If the translator determines that the requested trace is already available in the LRU set, the translator activates the trace. Such a strategy makes profiling simple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

6.1.3 Promoting Traces

Promotion of a trace to the active state simply implies changing its tag from old to current, and re-enabling trace linking. Threads can enter promoted traces. However, our promotion is in-place (the trace is not moved from its original position), which gives rise to fragmentation within the code cache. Figure 6.2(c) shows the resulting situation. Traces are being inserted into the free area of the code cache. At the same time, scattered old traces are being promoted to the current state.

6.1.4 Allocating Space to Traces

Traces are inserted into the contiguous, free area of the code cache in Figure 6.2(c). However, upon reaching the state shown in Figure 6.2(d), there is no more contiguous free space in the code



(a) Legend.

(b) Flush trigger point.

(c) Traces being inserted into the code cache and profiled traces being promoted.

(d) The contiguous, free area fills up.

(e) The code cache manager scavenges holes to allocate space to traces being inserted.

(f) The total size of current traces reaches a high water mark and triggers a flush.

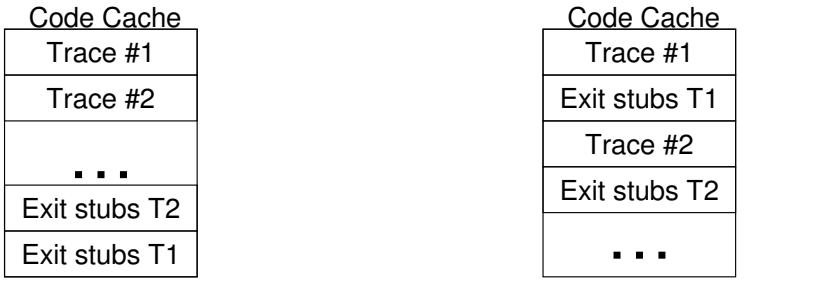
(g) On a flush trigger, all traces are marked as old.

Figure 6.2: The successive states of the code cache when our partial flush technique is applied.

cache to allocate from. From this point forward, the code cache manager must search for suitable free spaces in a fragmented code cache. The code cache manager treats the code cache as a circular buffer. It can assign already free space to a trace. It can also overwrite old traces if they are inactive. The code cache manager overwrites as few traces as possible, to allow profiling for longer periods of time.

The code cache manager may not be able to use all holes as some may not be large enough. Thus, the situation in Figure 6.2(e) results. The traces in unused holes continue to exist in the code cache and may get promoted or overwritten in the future. When the sum of the sizes of the current traces crosses the high water mark, the code cache manager triggers the next flush as shown in Figure 6.2(f). All existing traces are tagged old, as shown in Figure 6.2(g).

Figure 6.2(g) shows that the code cache is full of old traces which are being executed by threads. Therefore, it may seem that the code cache manager is unable to allocate space to new traces at this point, until all threads have exited the code cache once. However, there are really two kinds of old traces now. The first kind is active while the second kind has been inactive since the previous flush



(a) Separated traces and exit stubs.

(b) Contiguous traces and exit stubs.

Figure 6.3: Different arrangements of traces and exit stubs within the code cache. The advantage of separated exit stubs is that they represent infrequent paths. However, separated stubs complicate the code cache manager, which must scavenge holes in both the trace and exit stub areas. We therefore advocate the use of contiguous traces and exit stubs.

point. These inactive traces may be overwritten to allocate space to new traces. To do so, the code cache manager must be able to distinguish between active, old traces and inactive, old traces. To enable such a mechanism, the tag of a trace must have three possible values: 1) current, 2) old and active and 3) old and inactive.

6.2 Design Issues

In this section, we will discuss the design issues we faced and how we resolved them. Section 6.2.1 describes how we arrange traces in the code cache. Section 6.2.2 describes the data structure and algorithm used by the code cache manager to allocate space. Finally, Section 6.2.3 describes two different and popular linking strategies employed by DBTs and how we adapt our partial flushing technique for each of them.

6.2.1 Trace Arrangement

Figure 6.3 shows the different arrangements of traces and their exit stubs within the code cache – separated and contiguous. The separated arrangement in Figure 6.3(a) has been found to exhibit better performance [57]. However, the problem with this arrangement is that traces and their exit

stubs are associated with each other and yet, not co-located. It has to be ensured that when some exit stub is deleted to make space, the corresponding trace is also deleted. As the traces and exit stubs are not co-located, it is more complicated to ensure this condition. However, the arrangement in Figure 6.3(b) does not suffer from this problem. The traces and exit stubs are co-located and can be deleted together. We therefore employ this arrangement, and ensure that any resulting performance degradation is outweighed by the performance improvement of partial flushing by comparing with a baseline that uses the arrangement in Figure 6.3(a).

6.2.2 Managing Fragmentation

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

Figure 6.4 shows the algorithm for finding space using the code cache map. The code cache manager initially holds the value of the first entry in the code cache map i.e., a pointer to the directory entry of the first trace in the code cache. The manager iterates over traces in the code cache map. For each trace it tests whether the trace can be added to the existing hole. If the trace is not deletable or the trace is not contiguous with the existing hole, the trace cannot be added and the code cache manager discards the existing hole as too small. The discarded hole may get filled in a future pass. The manager also tries to start a new hole at or after the current trace, depending on whether or not the trace is deletable. If at any step, the manager finds the hole to be large

enough, it immediately stops. It allocates space from the hole and adjusts the hole. Before exiting, the manager saves the current state of the hole and the current cache map pointer so that it knows where to resume searching.

6.2.3 Trace Linking

The design of our proposed technique interacts with the linking strategy of the runtime. As traces move through the various states shown in Figure 6.1(b), we have to ensure that the linking policy of the runtime is being enforced. To this end, we implement our technique for two different linking policies - lazy and proactive.

For lazy linking, link data structures are discarded upon an unlink and reinstated when the path is again traversed. Lazy linking integrates seamlessly with our technique and no special handling is needed. However, applying our partial flush technique is more complicated in the case of proactive linking. In proactive linking, the code cache entry is the only way to locate branches to be linked. Exit stubs do not duplicate this data to save memory. Also the links are registered with code cache entries only once, when a trace is being translated. We have to ensure that this information is not lost as traces get evicted and retranslated.

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

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

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

Figure 6.4: Algorithm for managing fragmentation. The terms `first`, `last` and `next` should be considered in the spatial (rather than temporal) context.

technique.

6.3 Performance Evaluation

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

We describe our evaluations in the following sections. Section 6.3.1 evaluates and discusses the performance of our proposed technique when applied to single-threaded code caches. Section 6.3.2 explores the performance of our proposed technique when applied to thread-shared code caches.

6.3.1 Single-Threaded Evaluation

For our experimental environment, we used an an iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. The IPAQ has a 200 MHz StrongARM-1110 processor with 64 MB RAM, 16 KB instruction cache and a 8 KB data cache. For the dynamic binary translator, we used Pin [51] for ARM. We implemented and used a lazy linking policy in the runtime for this set of experiments. The runtime uses a code cache limit of 256 KB and triggers a flush when the code cache is 100% full. For our test programs, we used two different benchmark suites: 1) the Mibench [50] suite with large datasets and 2) the SPEC2000 [55] integer suite with test inputs (there was not enough memory on the embedded device to execute larger inputs, even natively). In all these experiments, we are really interested in improving the performance of long-running benchmarks given a fixed

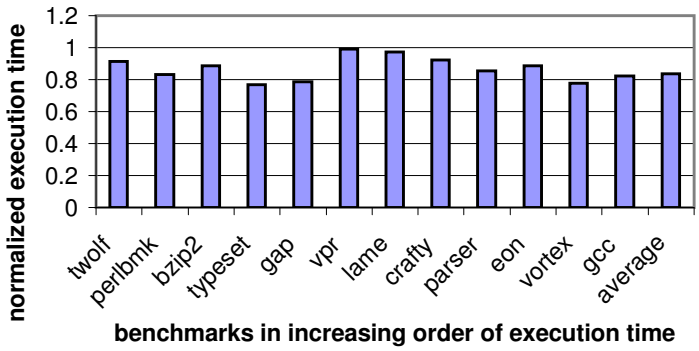


Figure 6.5: Execution time for our proposed partial flush technique with respect to full flush. We reduce execution time by about 17% on the average, for single-threaded benchmarks.

memory budget. Therefore, we did not consider benchmarks with baseline execution times below 100 seconds. This decision eliminated some Mibench and SPEC2000 benchmarks.

Figure 6.5 shows the normalized execution times for the single-threaded benchmarks when our partial flushing technique is applied. All the benchmarks show some improvement in execution time, the average being about 17%. This speedup over full flush shows that the overheads of execution in the unlinked mode and extra book-keeping needed by partial flush are outweighed by the improvements in translation time.

We also studied the source of the speedup by splitting up the total execution time into components. Figure 6.6 shows the fraction of execution time reduction caused by each component. The components on the positive side of each bar contributed to speedup while components on the negative side contributed to slowdown. The effective speedup is calculated by subtracting the total bar height on the negative side from the total bar height on the positive side. The effective speedup in Figure 6.5 and Figure 6.6 may not exactly match because the results in Figure 6.6 are somewhat contaminated by profiling time. We measured the difference between actual execution time and profiled execution time for each benchmark and found that profiled execution time is higher than the actual execution time by 8% on average.

From Figure 6.6 it is clear that the main contributor to speedup is the reduction in translation time, resulting from fewer retranslated traces. The next most important contributor is context switch time, though it contributes to slowdown because there are more context switches compared to full

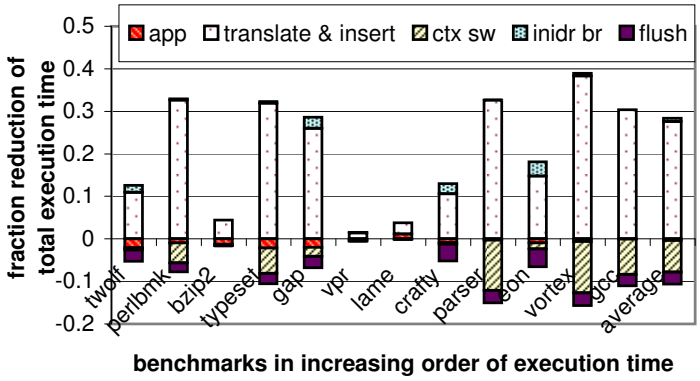


Figure 6.6: Fraction of speedup resulting from each DBT task. Reduction in translation time is the greatest contributor of speedup.

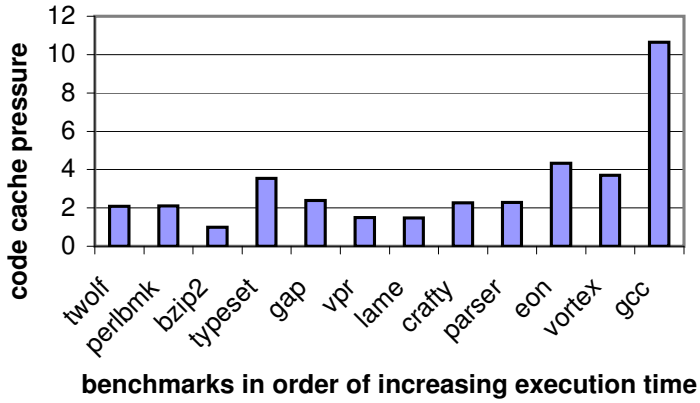


Figure 6.7: Cache pressure for single-threaded benchmarks. Cache pressure is the ratio of the unlimited code cache size to the memory limit (256 KB in this case).

flushing. The reason for having more context switches is that the code cache suffers from fragmentation during partial flush and can accommodate fewer traces compared to full flush. As a result, a trace in a partial flushing system survives through more code cache flushes on an average. Surviving across each flush implies there will be one context switch to promote the trace and there will be one context switch for placing each link to the trace, leading to more context switches overall. It is worth noting that not much extra time is spent in flushing i.e., the book-keeping overhead of our proposed technique is small. Application execution and indirect branch handling time also remain fairly stable.

In order to understand why the reduction in translation time varies among benchmarks, we studied the cache pressure on each of them. The cache pressure is the ratio of the unlimited cache

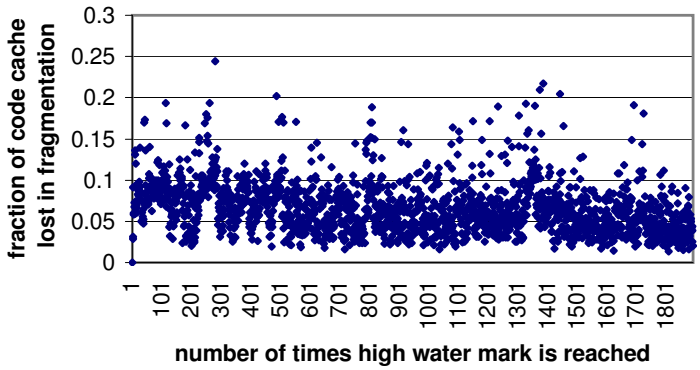


Figure 6.8: Code cache fragmentation for the largest benchmark `gcc`. Fragmentation remains stable and usually below 10%.

size of a benchmark to the memory limit. Figure 6.7 shows that cache pressure varies from about 1(`bzip2`) to 10(`gcc`). A lower cache pressure indicates that fewer retranslations are needed and there is less room for improvement. Indeed `vpr` and `lame` were among the benchmarks with the lowest cache pressure and also registered the least improvement in translation time and the least speedup.

We also measured the impact of fragmentation. Our goal was to ensure that fragmentation does not increase as flushes progress. If fragmentation steadily increases, partial flushing will become useless after a point i.e., our technique will not scale. Figure 6.8 shows the fragmentation in the code cache for `gcc`, the largest benchmark. Fragmentation remains fairly stable across flushes and within 10%, showing that our technique is scalable. Although the amount of memory lost in fragmentation is unusable by the system, it still improves performance compared to full flush.

6.3.2 Multi-Threaded Evaluation

For our experiments on thread-shared code caches, we used an ATOM N270 netbook with a 1.6GHz processor supporting two hardware thread contexts. The processor has a 32KB instruction cache, 24KB data cache with write-back and a 512 KB L2 cache. The memory size is 1 GB. It supports Linux kernel 2.6.24. For the ATOM-based netbook, we used Pin [52, 69] targeting the x86 architecture. The runtime in this case implemented proactive linking. The runtime uses a code cache limit of 512 KB and triggers a flush when the code cache is 70% full (unless otherwise stated). We used

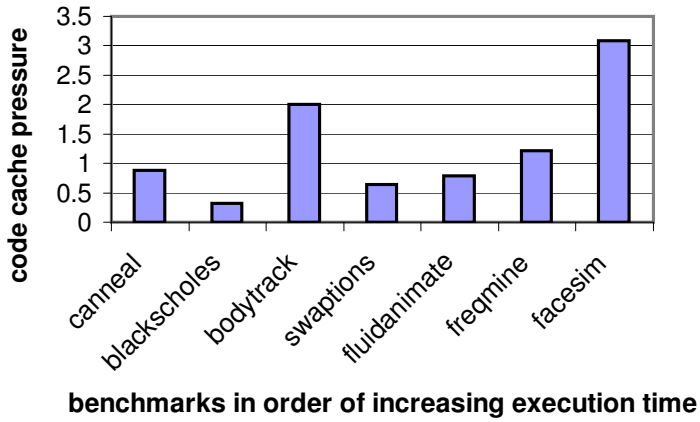


Figure 6.9: Cache pressure for multi-threaded benchmarks. Cache pressure is the ratio of the unlimited code cache size to the memory limit (512 KB in this case).

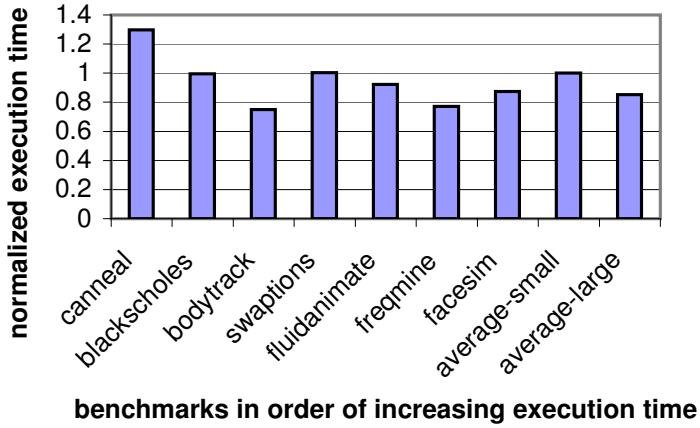


Figure 6.10: Execution time for partial flush applied to thread-shared caches, normalized with respect to full flush. Average speedup is 15%.

the PARSEC [12] suite with native inputs. PARSEC consists of multi-threaded benchmarks and we executed them on the netbook with two threads.

We first explored which of the multi-threaded benchmarks would need flushing activity for the given memory limit. Figure 6.9 shows the ratio of the unlimited cache size of each benchmark to the given memory limit. Benchmarks will undergo flushing only if their cache size crosses the high water mark. Therefore, in our case, if the cache pressure is at least 0.7, we expect flushing to occur. The benchmarks in this category are `canneal`, `bodytrack`, `fluidanimate`, `freqmine` and `facesim`. For the other two benchmarks, `blackscholes` and `swaptions`, we are interested in ensuring that overhead is reasonable rather than obtaining improvements.

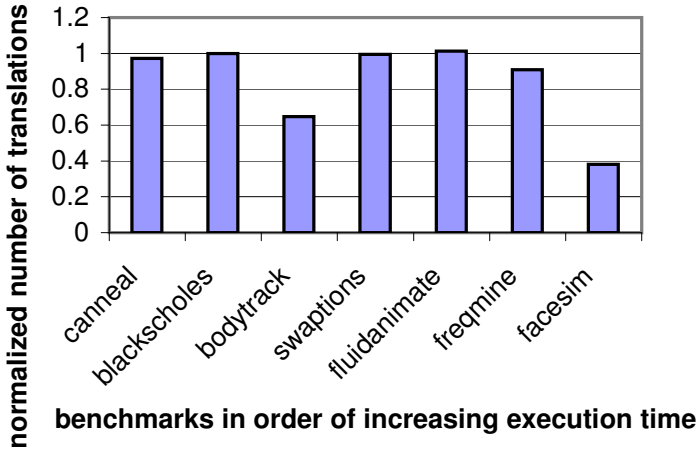


Figure 6.11: The number of trace translations normalized with respect to full flush.

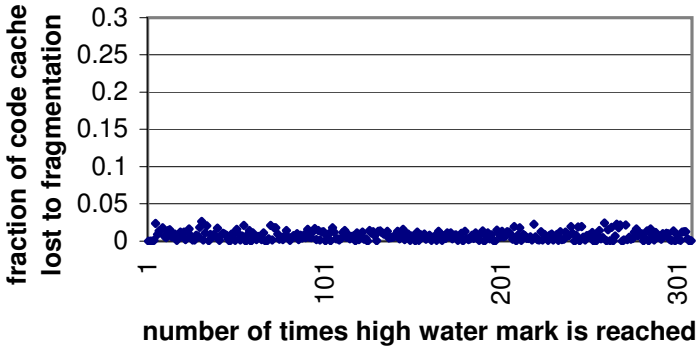


Figure 6.12: Code cache fragmentation for the largest benchmark `facesim`. Fragmentation remains stable and within 3%.

Figure 6.10 shows the normalized execution times for the benchmarks. `Average-small` is the average normalized execution time for the small benchmarks, i.e., the benchmarks which do not undergo flush activity. `Average-large` is the average normalized execution time for the large benchmarks i.e., the benchmarks which do undergo flush activity. On average, the performance improvement for the large benchmarks in 15% while that for the small benchmarks remains the same. Therefore, we have ensured that we get performance improvements for the large benchmarks and do not cause overhead for the small benchmarks. However, we fail to improve performance for `canneal`.

To understand the performance results, we analyzed how many trace translations we have reduced using our technique. Figure 6.11 shows the normalized translation count for partial flush.

`bodytrack`, `freqmine` and `facesim` had the most cache pressure and show the greatest reduction in translations. They also show the best performance improvements, among the large benchmarks. `canneal` and `fluidanimate` have relatively less cache pressure and also show less translation reduction. Not surprisingly, their performance improvements are the lowest among the large benchmarks. `canneal` actually shows slowdown. The slowdown is due to the fact that `canneal` is also the shortest-running benchmark. It is an order of magnitude shorter than `bodytrack`, the next longer benchmark in the large category. Therefore, the overheads due to our technique is more pronounced in `canneal`.

We measured the fragmentation impact for multi-threaded benchmarks. Our goal was to ensure that fragmentation does not increase as code cache flushes progress i.e., our technique is scalable. Figure 6.12 shows the fragmentation for the largest benchmark, `facesim`. The fragmentation is stable, showing that our technique is scalable for thread-shared caches.

6.4 Summary

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

Chapter 7

A Dynamic Binary Translator for Embedded Systems

Chapters 3 through 6 strive to design a DBT for embedded systems using different approaches. In this chapter, the designs are unified to produce a whole system. We select which approaches to combine, and how beneficial it is to combine them. If some approaches cannot be combined seamlessly, we design modified strategies. The unification demonstrates the total improvements we can achieve.

The challenge in producing a unified system is that the intra-flush optimizations were all evaluated for full flush systems supporting single-threaded guest applications. However, the whole system will support partial flush as well as thread-shared code caches for multi-threaded applications. For the inter-flush optimizations, we assumed a particular path selection strategy although that may not be the path selection strategy for the whole system. For generational cache flushing, we did not evaluate with thread-shared caches. Also, we did not consider the data structure sizes for any of the flushing strategies.

To produce the whole system, we select intra-flush optimizations that can be integrated with partial flush and thread-shared code caches without a large amount of effort and yet have shown large benefits. We extend these optimizations to support partial flush and thread-shared caches. For the inter-flush optimizations, we select those optimizations that can apply to both single-threaded and thread-shared code caches. We also accounted for the data structure sizes produced by these techniques.

Section 7.1 discusses the considerations when integrating the path selection approach. Section 7.2 describes the exit stub optimizations incorporated into the full system. Section 7.3 describes the integrated flushing technique. Section 7.4 presents an experimental evaluation of the system and Section 7.5 summarizes.

7.1 Balanced Path Selection

Path selection combines trace selection and link formation. We found dynamic trace selection to be the most beneficial strategy. However, there are several issues in integrating dynamic trace selection with partial flushing and thread-shared code caches. We discuss these issues and their solutions in Section 7.1.1 and Section 7.1.2. For the linking strategy, we found lazy linking to be most beneficial. Lazy linking integrates seamlessly with partial flushing and thread-shared code caches because it only requires some extra data (the branch location) to be stored in the exit stub. Also, the code has to be modified to ensure that links are formed only on demand. These modifications do not conflict with any aspect in the rest of the system and therefore lazy linking does not present any issues.

7.1.1 Interaction between Dynamic Trace Selection and Unified Cache Flush

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

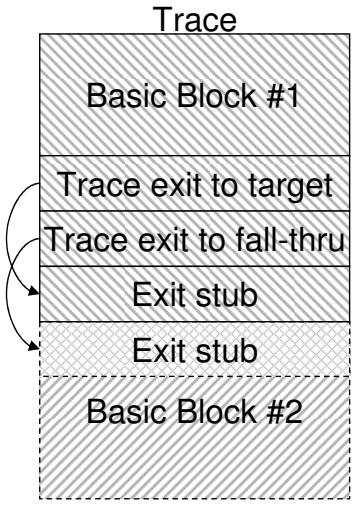


Figure 7.1: Dynamic trace selection and unified cache flushing are integrated into the whole system. Consecutive basic blocks in a trace are separated by exit stubs. Exit stubs corresponding to fall-throughs can be overlaid.

A second problem that occurs with unified cache flush is the trace arrangement strategy. We adopt the trace arrangement strategy shown in Figure 6.3(b) where exit stubs immediately follow the traces. However, for dynamic trace selection, the arrangement in Figure 6.3(a) is more convenient. Dynamically adding basic blocks to the trace maintains the arrangement in Figure 6.3(a). For the arrangement in Figure 6.3(b), each basic block in a trace will be followed by its exit stubs i.e., consecutive basic blocks belonging to the same trace will be separated by exit stubs. Such an arrangement reduces code cache locality.

Also, the space savings reaped from the trace arrangement in Figure 6.3(b) are less than those from the arrangement in Figure 6.3(a). In Figure 6.3(a), both the trace exit and the exit stub corresponding to the fall-through can be overwritten because both of them may border on free space. However, for the arrangement in Figure 6.3(b), the trace exit corresponding to the fall-through will never border on free space and as a result can never be reclaimed. Figure 7.1 shows the trace format resulting from integrating dynamic trace selection and unified cache flushing.

7.1.2 Interaction between Dynamic Trace Selection and Thread-Shared Code Caches

Dynamic trace selection also encounters issues when integrated with a thread-shared code cache. This interaction also arises because the traces are extended one basic block at a time. Two or more threads may strive to construct different traces at the same time. The code cache manager will interleave the basic blocks from the different traces in this situation. If the basic blocks for a trace cannot be allocated contiguously, they must form different traces. Thus, the potential benefit from dynamic trace selection is reduced.

One approach is to not service any other thread in the runtime while one thread is constructing a trace. However, this design may cause deadlock. Therefore, we only check whether we have been able to allocate contiguously or not. We quantify what percentage of traces may encounter this situation to evaluate the potential loss, in Section 7.4.3.

7.2 Exit Stub Optimization

We apply the reduction in exit stub size optimization among the various exit stub optimizations that we explored. We studied the reduction strategy for the baseline system which uses proactive linking. However, our path selection strategy recommends lazy linking. Lazy linking requires larger exit stubs compared to proactive linking. Therefore, we expect the benefits of exit stub optimizations to be reduced when combined with the path selection strategy. Apart from this interaction, reduction in exit stub size integrates seamlessly because it merely recommends a different format for exit stubs. The different format has no interaction with the number of threads executing in the code cache and the flushing technique. Since a large portion of the memory savings was due to the reduction in exit stub size optimization, most of the memory savings realized can be carried over to the whole system implementation.

The shared exit stub optimization is more complicated to apply if there is more specialized information in exit stubs, as explained in Section 4.2.2. Lazy linking is an example where more specialized information such as branch location is stored in the exit stub. Also, sharing of exit stubs

places a restriction on the granularity of flushing, which is difficult to integrate with partial cache flushing. Therefore, we do not apply this optimization in the whole system.

Finally, deleting exit stubs and avoiding compilation of exit stubs carries the restriction that invalidations and partial flushes cannot occur, which conflicts with partial flushing. Also, deleting exit stubs is complicated for thread-shared caches for the same reasons as trace eviction. Therefore, we do not apply these optimizations in the whole system.

7.3 Cache Flush

We incorporated the unified cache flush into the full system. We preferred it to the generational cache flush because it applies to both single-threaded and thread-shared code caches uniformly. Also, we reaped greater benefits using a unified cache flush rather than a generational cache flush. The unified cache flush, however, requires some extra data structures. In the whole system, we must take into account the size of data structures to determine when we reach the memory limit. We discuss the impact of data structures and how we handle them in Section 7.3.1, Section 7.3.2 and Section 7.3.3.

7.3.1 Trace Tagging Data Structures

Each trace needs to be tagged as 1) current, 2) old and active, or 3) old and inactive in an unified cache flushing system. The tag requires extra space. However, allocating extra space for the tag can be avoided by noting that it has only three possible values. Therefore, each tag needs two bits. At the same time, all traces are aligned to word boundaries and a word is four bytes long in the ARM platform. Therefore, every code cache address field in a code cache directory entry will have its last two bits set to zero. We use these two bits to encode the tag of the trace. Therefore, there is no extra space requirements for trace tags.

7.3.2 Code Cache Map

The code cache map enables the code cache manager to scavenge for holes in the code cache. It contains pointers to the code cache directory entries of the traces in increasing order of code cache addresses. The size of the code cache map is included in the total memory demand, along with the code cache size, the code cache directory size and the link data structure size.

7.3.3 Including Data Structure Size in Memory Demand

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

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

7.4 Experimental Results

The goal of our experiments was to determine the potential of combining the proposed techniques. The evaluation of the whole system demonstrates whether all the benefits reaped in the previous chapters are combinable, and if not, what is the reason for losses. We measured the memory efficiency and performance of the combined system to achieve this goal.

For our experimental environment, we used an an iPAQ PocketPC H3835 machine running Intimate Linux kernel 2.4.19. The IPAQ has a 200 MHz StrongARM-1110 processor with 64 MB RAM, 16 KB instruction cache and a 8 KB data cache. For the dynamic binary translator, we used Pin [51] for ARM. The runtime uses a memory demand limit of 512 KB. For our test programs,

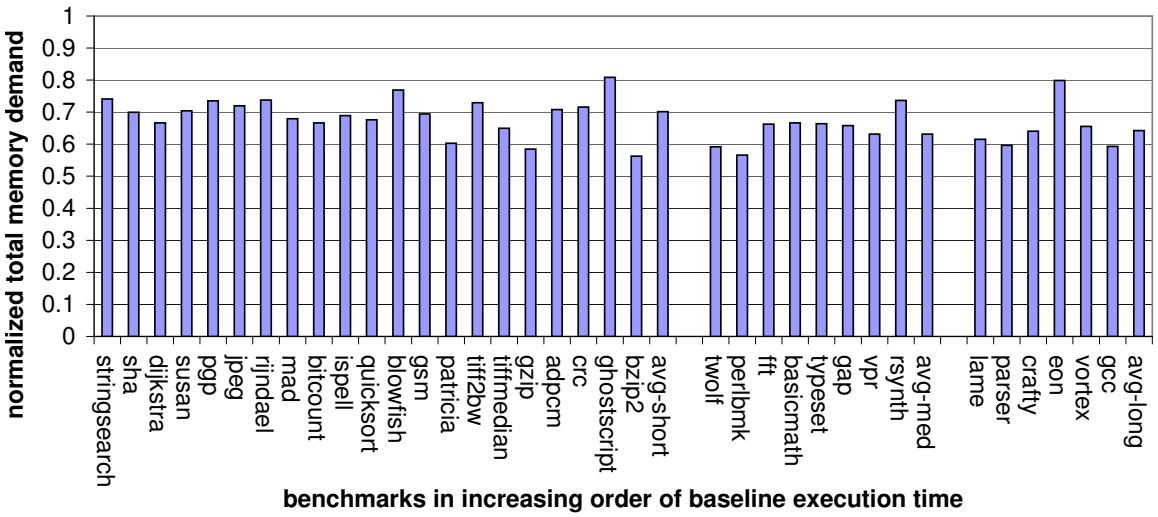


Figure 7.2: Normalized total memory demand of the full system, which combines balanced path selection with reduction in exit stub optimization. Although cache flushing is not used, the data structures required by unified cache flushing are accounted for.

we used two different benchmark suites: 1) the Mibench [50] suite with large datasets and 2) the SPEC2000 [55] integer suite with test inputs (there was not enough memory on the embedded device to execute larger inputs, even natively). We show results for the short (0–100s), medium (100–1000s) and long (1000s–) benchmarks separately.

We present the memory evaluation results in Section 7.4.1 and the results on performance evaluation in Section 7.4.2. The evaluations presented in these sections apply to single-threaded benchmarks. We measure the interaction between dynamic trace selection and thread-shared code caches in Section 7.4.3 to determine whether we can expect similar results as in the case of single-threaded code caches.

7.4.1 Memory Evaluation

We roughly estimated the expected memory savings from our combined techniques first. For the estimation, let us assume the total memory demand of the baseline system to be 1. According to Figure 1.2, the translated code size, auxiliary code size and the data structure size of the baseline system are 23%, 41% and 36% respectively. Therefore, the code cache size of the baseline system is 64% (sum of translated code and auxiliary code sizes). We applied our exit stub optimizations

to the code cache only. For the reduction in exit stub size optimization (which we included in the combined system), we obtained a code cache size reduction of 37%. Therefore, we expect the code cache size to be 40% after this optimization. The total memory demand after this optimization is 76% (sum of code cache and data structures). Dynamic trace selection combined with lazy linking further improved the memory efficiency of the whole system by 20%. Therefore, the total memory demand after combining balanced path selection is 61%, i.e., we expect to improve memory efficiency by 39%.

Figure 7.2 shows the actual normalized memory demand of the whole system. There are memory savings for all the benchmarks, ranging from 20% to 44%. The average memory savings for the short benchmarks, medium benchmarks and the long benchmarks are 30%, 37% and 36% respectively. These numbers show that memory efficiency benefits from our different techniques have been combined because the memory savings of the combined system is better than the memory savings of any of our techniques in isolation. Also, we have produced results quite close to the expected memory efficiency. The actual memory savings is less than the estimated memory savings because 1) memory efficiency was lost in integrating dynamic trace selection, 2) lazy linking requires larger exit stubs than proactive linking, reducing the benefit of exit stub optimizations, and, 3) the code cache map required by unified cache flushing has been incorporated into the memory demand. Figure 7.3 shows the actual memory demands of the benchmarks. The graph uses a logarithmic scale because there is a wide range of memory demand among the benchmarks.

7.4.2 Performance Evaluation

Figure 7.4 shows the normalized execution time of the benchmarks. There are no clear gains for the short benchmarks. However, this is not surprising because short benchmarks do no execute long enough to amortize the overheads of the proposed techniques. For the medium benchmarks, performance gains begin to manifest more clearly. There is an average improvement of 10% in the execution time, with a maximum improvement of 37% in `perlbmk`. For the long category, performance of every benchmark in improved. The average reduction in execution time is 27%, with a maximum execution time reduction of 89% in `parser`. One especially encouraging fact

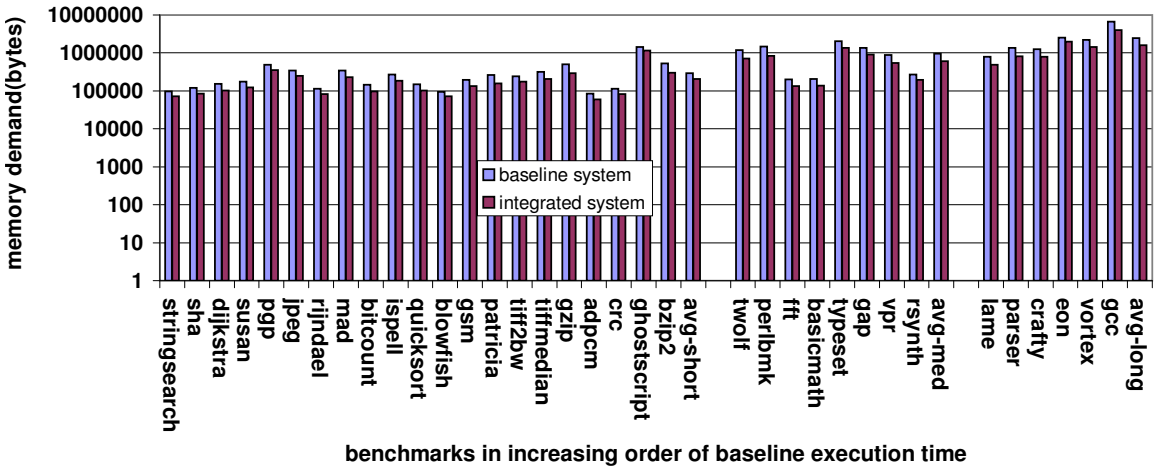


Figure 7.3: Memory demand of the benchmarks, in bytes.

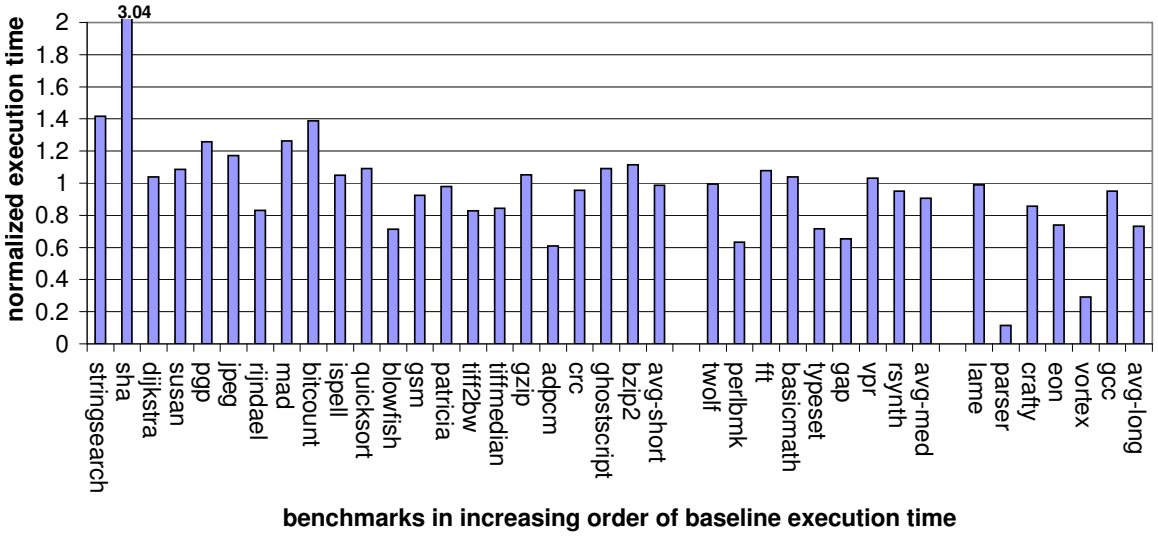


Figure 7.4: Normalized execution time of the full system, which combines balanced path selection with reduction in exit stub size and unified cache flush.

is that, for `gcc`, the longest-running benchmark, a 5% performance improvement occurs in the combined system, while path selection combined with full flushing was unable to improve the performance of `gcc` over the baseline. Similar to Figure 7.3, Figure 7.5 shows the actual execution times of the benchmarks in seconds and the graph uses a logarithmic scale.

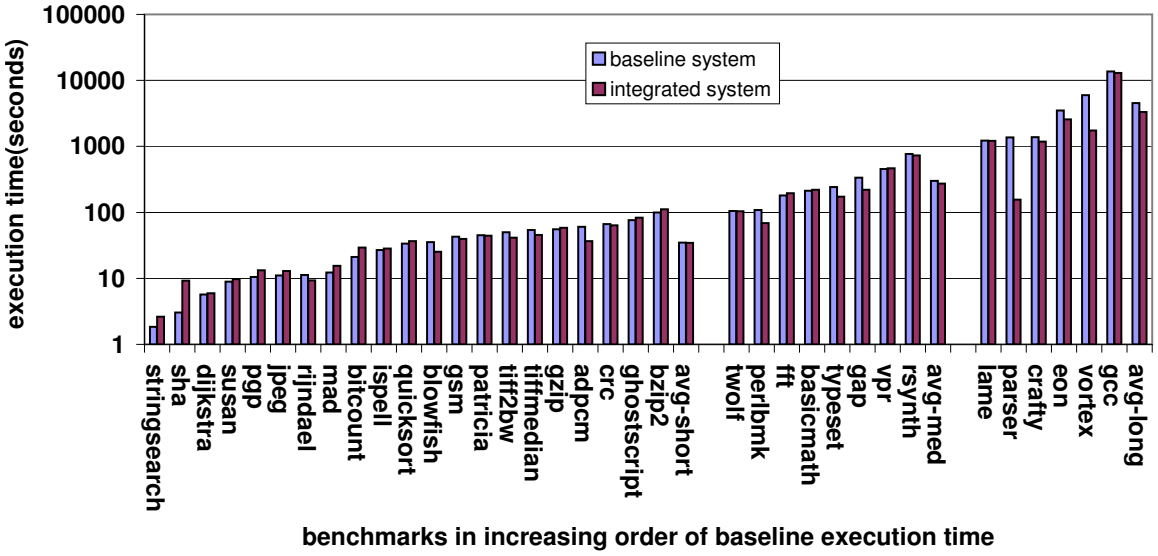


Figure 7.5: Actual execution times of the benchmarks, in seconds.

7.4.3 Interaction between Dynamic Trace Selection and Thread-Shared Code Caches

For our experiment on thread-shared code caches, we used an ATOM N270 netbook with a 1.6GHz processor supporting two hardware thread contexts. The processor has a 32KB instruction cache, 24KB data cache with write-back and a 512 KB L2 cache. The memory size is 1 GB. It supports Linux kernel 2.6.24. For the ATOM-based netbook, we used Pin [52, 69] targeting the x86 architecture. We used the PARSEC [12] suite with native inputs.

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

Figure 7.6 shows the ratio of the number of traces formed when using two threads to the number of traces formed when using a single thread. The increase in the number of traces formed ranges from 0% to 4%, averaging 3%. Therefore, the increase in the number of traces formed is small and

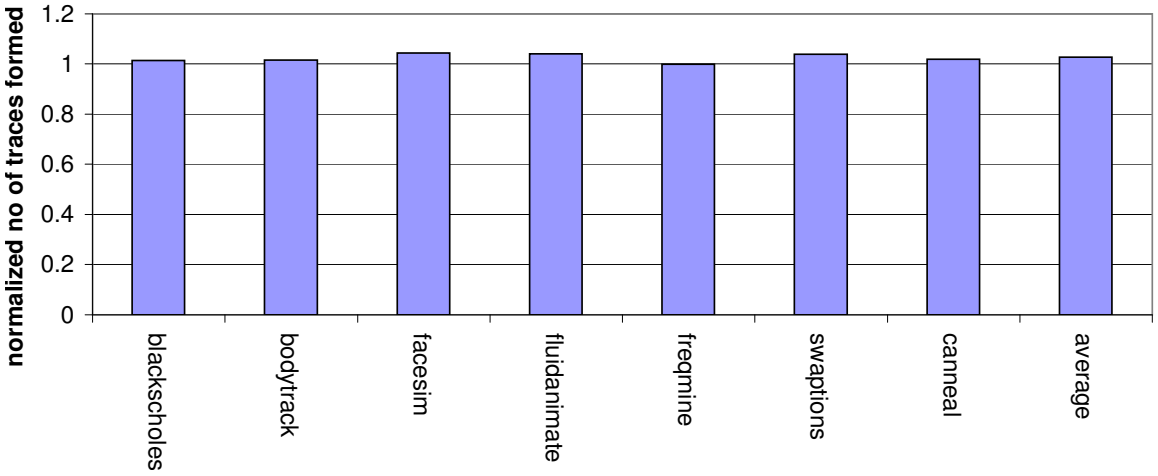


Figure 7.6: Number of traces formed by dynamic trace selection applied to a thread-shared code cache, normalized wrt the number of traces formed by dynamic trace selection applied to a single-threaded code cache.

results similar to single-threaded code caches can be expected for thread-shared code caches.

7.5 Summary

We combined dynamic trace selection, lazy linking, reduction in exit stub size and unified cache flushing into a full system. We identified that the dynamic trace selection strategy and the unified cache flush strategy have some conflicting requirements. We modified the trace arrangement of both the dynamic trace selection and unified cache flushing to address the conflict. We also modified the code cache manager for the unified cache flushing strategy. We identified possible conflicts between dynamic trace selection and thread-shared code caches. We quantified the loss due to the conflict and found it to be small. We also identified conflicts between lazy linking and reduction in exit stub size. We modified the exit stub to support both these optimizations. Finally, we mitigated and incorporated the impact of the data structures required by unified cache flushing into the total memory demand. We evaluated the memory efficiency and performance of the resulting system. The memory savings range from 20% to 44% and average at 30%, 37% and 36% for short, medium and long benchmarks. Performance improvements for the medium and long benchmarks average at 10% and 27% respectively, with maximums of 37% and 89%.

Chapter 8

Related Work

In this chapter we first provide related work on DBT applications in Section 8.1. Then we describe previous research on embedded DBTs in Section 8.2. We discuss previous research related to path selection, exit stub optimization and code cache management in Section 8.3, Section 8.4 and Section 8.5 respectively. We describe other translation techniques in Section 8.6.

8.1 Dynamic Binary Translators

Dynamic binary translators provide software adaptation in various forms [9, 13, 18, 38, 89, 92]. For example, DBTs can enforce security policies [18, 58, 62, 88]. Some DBTs such as Pin [69] and Valgrind [80] support dynamic instrumentation. HDTrans [92] is a lightweight open-source dynamic binary instrumentation framework. Optimizers form another major class of DBT systems. Dynamo [9] optimizes program hot paths and bails out if necessary. Wiggins/Redstone [32] employs program counter sampling to form traces which are then specialized for the Alpha architecture. Mojo [61] is an optimizer that targets Windows NT running on IA-32. Kistler [63] proposes continuous program optimization that involves operating system re-design. ADORE [24] is a research dynamic optimizer that uses the Itanium performance monitors to guide optimization of Itanium Processor Family (IPF) applications. Frequently executed superblocks are identified by the hardware and are optimized by the ADORE software system.

While the host and guest platforms are the same for the DBT systems enumerated in the previ-

ous paragraph, many DBTs target a guest application to a different host platform during execution. Several DBTs provide translation between platforms [10, 37, 41, 86, 96, 101]. Some DBTs virtualize the entire software stack [1]. DBT technology also supports simulators and emulators. In these systems, overhead is lowered by caching native code translations of frequently interpreted regions [11, 28, 70, 94, 97].

Systems such as Strata [57, 65, 89] and Walkabout [27] have a goal of enabling fast production of retargetable dynamic binary translators. Strata targets the SPARC, MIPS, IA-32, ARM, and PISA platforms while Walkabout targets SPARC and IA-32.

8.2 Dynamic Binary Translators for Embedded Systems

Although development of DBTs for embedded systems has been limited compared to general-purpose platforms, there are quite a few embedded DBTs in use. Strata has been targeted to ARM and PISA [6,7,8,76]. Pin provides a dynamic binary instrumenter for ARM [51]. DELI [38] exposes client API for fine-grained control over the guest application execution to provide services such as ISA emulation, software patching and sandboxing on the Lx/ST210 embedded VLIW processor.

Several DBT applications that are especially useful in embedded environments are possible. Examples are dynamic power management [98] and software caching [75, 99, 102]. Software caching manages scratchpad memory during execution by minimizing the cost of loading and evicting instructions from the scratchpad.

8.3 Path Selection

Researchers have found that the code cache has a memory footprint that is 5-10 times the guest application instruction footprint [54]. Also, the data structures are comparable in size to the code cache [59]. The absolute and relative sizes of the code cache and the data structures depend on path selection. However, the previous research on path selection has been from the performance perspective. Previous work has researched both trace selection [40, 56, 57] and linking strategies [9,

18, 38, 61, 69]. Such approaches do not consider the memory demands of path selection or its performance in memory-constrained environments.

For most DBTs, the trace is the unit of choice compared to functions and methods [9, 18, 38, 61, 69, 80, 89]. Indeed, the trace was found to perform well and to be easy to compile [17]. Next-Executing-Tail (NET) is the most popular trace selection algorithm. NET identifies certain instructions as potential trace heads and profiles them until they reach a hotness threshold. NET starts compiling traces at hot trace heads by following the execution direction at every basic block tail, until an end-of-trace condition is reached. While our dynamic trace selection strategy is similar to NET, we conclude that it is not beneficial to deviate from straight line code within a trace although NET may do so. This difference arises because NET was designed with performance as the goal while our goal is memory efficiency leading to good performance in memory-bound situations. There are several variants of NET in use. Mojo [61] uses one threshold for backward-branch targets and a lower threshold for trace exits. BOA [86] maintains counts for each conditional branch that indicate how many times each target is taken in its emulation phase. After the entry point to an instruction sequence is emulated 15 times, a trace is selected by following the target of each conditional branch with the highest count. Hot groups are formed by collating individual paths based on collected branch frequencies. When a hot group entry has been found, a path is selected by following the most likely successors according to the collected branch profile information. Wiggins/Redstone [32] identifies the beginning of a trace by periodically sampling the program counter. From a starting instruction, a trace is selected by adding instrumentation code that determines the most frequent target of each selected branch. ADORE [24] samples registers from the performance monitoring unit of the Intel Itanium 2 in order to detect the four most recently taken branches. When a set of four branches occurs frequently, the corresponding path is selected and linked with other frequent paths to form a trace.

Other DBTs use trace selection strategies that are not so similar to NET. Pin uses straight-line code in its traces. The Pin strategy is same as the threshold-based strategy. Recently, Pin has been used to simulate the last-executed iteration (LEI) strategy to better select traces for optimization [56]. HDTrans [92] uses single-block traces, and elides unconditional branches, but continues

translating the fall through instructions after direct calls.

NET uses interpretation [9] or caches code as basic blocks [18] before it builds a trace. However, we did not use interpretation or basic block caching in our system because DBTs such as Strata have achieved close-to-native performance using full tracing. Indeed, Strata has been implemented on RISC architectures such as SPARC and MIPS. Since ARM is also a RISC architecture, similar results can be expected for ARM.

For the linking strategy, most DBTs use proactive linking [9,18,69]. Some DBTs use a deferred linking policy [38, 61] in which they proactively link the exits of the trace under construction, but link the entries to that trace lazily. Such a policy will still create more unnecessary links than lazy linking. However, deferred linking will reduce some context switches compared to lazy linking. Strata for embedded systems [6, 8] recommends a full proactive linking policy. The reason for recommending a full proactive linking policy may be that when only the code cache size is considered in a memory-constrained environment, proactive linking outperforms both lazy linking and deferred linking.

8.4 Exit Stub Optimization

Hiser et al. study the allocation of trampolines in a separate region (a trampoline pool) in comparison to interleaving them with fragments [57]. They find the technique lowers I-cache pressure. While a trampoline pool is a performance optimization, Strata for ARM also performs memory optimizations on exit stubs [8].

8.5 Code Cache Management

Researchers have found that pure emulation-based VEEs for general-purpose environments have 300x performance overhead compared to native execution [15]. Therefore, code cache usage is common in DBTs.

Partial flushing for single-threaded code caches has been studied. Hazelwood et al [53] found that the retranslation rate for FIFO eviction is similar to that for LRU eviction with less management

overhead and 50% better than the retranslation rate for full flush. Hazelwood et al [54] also found that mid-grained evictions scale better than full-flush and fine-grained evictions, and proposed a generational approach that stores short-lived and long-lived fragments in distinct code caches to prevent premature evictions.

Strata for embedded systems [6, 8] flushes on reaching a memory limit, but does not consider data structure size. Similarly, Pin for the ARM architecture [51] does not consider data structure size when triggering a flush. Valgrind [80] performs FIFO single fragment replacement when its cache fills up. Dynamo [9] flushes single-threaded code caches for performance reasons by full flushing preemptively when a phase change is detected.

Full cache flushing has also been studied for thread-shared caches. However, the flushes are not always performed to reduce memory demand. Flushing is carried out for performance and maintaining cache consistency as well. Thread-shared software code caches [14, 52] emerged as a memory optimization over thread-private caches, at the cost of increased complexity. Bruening et al [16] develop methods for maintaining the consistency of the code cache in the presence of self-modifying code. They also explored expanding a bounded code cache. They double the code cache size when the ratio of retranslated to replaced fragments is above a threshold. But they only scale up the code cache limit adaptively, which may not be suitable in a memory-constrained environment. Pin [52] supports both single-threaded and thread-shared code caches, but only uses a full flush for capacity. Aries [101] uses a single global lock around runtime system code and supports freeing cache space only via forcing all threads out of the cache.

Memory management can be alternatively performed by cached code compression [6, 91, 36]. Compression has a lower retranslation cost compared to eviction but it frees less space at a time. Compression also faces the same challenges as eviction such as selecting which traces to compress, code cache management and thread management. Baiocchi et al [6] used victim compression and pinning to reduce retranslation cost when the binary is in external Flash. Shogan et al [91] compress the program binary prior to execution. The image is decompressed by the dynamic translator.

Memory demand can also be managed by using a client-server [82, 99, 102] architecture for the DBT. Code is stored on a server and downloaded to the embedded system on demand.

Probing into the original binary is an alternative to code caching. The probe-based approach works by dynamically replacing instructions in the original program with trampolines that branch to the instrumentation code. Example probe-based systems include Dyninst [19] and DTrace [22]. The drawback of probe-based systems is that it may not always be possible to replace exactly one instruction in the original binary with a trampoline. Replacing multiple instructions can cause correctness issues because there may be branches to the replaced tail instructions. When such branches are executed, they will transfer control to the middle of a trampoline. This limitation makes fine-grained instrumentation challenging on probe-based systems.

8.6 Other Non-Compile-Time Translation Techniques

Static binary instrumentation was pioneered by ATOM [93], followed by others such as EEL [66], Etch [84], and Morph [100]. Static instrumentation is limited compared to dynamic instrumentation because it is possible to mix code and data in an executable and a static tool may not have enough information to distinguish the two.

Selective dynamic compilation [4, 30, 42, 44, 68, 71, 83] is a staged form of compilation that restricts dynamic compilation to selected portions of code identified by user annotations or source language extensions. In these cases, the static compiler prepares the dynamic compilation process as much as possible by generating templates that are instantiated at run-time by a specialized dynamic compiler.

There are several implementations of offline binary translators that also perform native code optimization [26]. These generate profile data during the initial run via emulation, and perform background translation together with optimization of hot spots based on the profile data. The benefit of the profile-based optimization is only available during subsequent runs of the program and the initial profile-collecting run may suffer from worsened performance. Related techniques include link-time optimization [29, 78].

JIT compilers [2, 3, 23, 31, 39] delay all compilation until execution since they target platform independence. KVM [73] was the first java virtual machine developed by Sun Microsystems for em-

bedded platforms. However KVM is a pure bytecode interpreter and performance suffers as a result. Some just-in-time compilers perform profiling to identify which methods to spend more optimization time on [72]. The Jalapeno Java virtual machine [3, 21, 64] optimizes all code at an initial low level of optimization, embedding profiling information that is used to trigger reoptimization of frequently executed code at higher levels. Standards such as Java card [25], J2ME/CLDC [74, 77] and J2ME/CDC [67] have been built for embedded JVMs. JEPES [87] and Armed E-Bunny [33, 34, 35] are examples of research on embedded JVMs. There have been many research efforts to reduce the memory footprint of embedded applications [5, 45] and JVMs [90].

There are several hardware assists for execution time optimization. The trace cache is a hardware alternative that can be extended to do superscalar-like optimization off the critical path [43, 85]. Software code caches are also coupled with hardware support for hardware virtualization [20, 95] and instruction set compatibility [37, 38, 41].

Chapter 9

Merits and Future Work

This dissertation explored approaches to reducing the memory demand of DBTs targeting embedded systems. These approaches reduce the pressure on the memory subsystem, a resource which is constrained in embedded systems. For DBTs that handle the problem of memory demand by placing a limit on the memory usage, the memory demand problem manifests itself as a performance degradation problem. Our memory optimizations help reduce this performance degradation by reducing the number of flushes. Additionally, we designed strategies to selectively preserve traces and their corresponding exit stubs and data structures across flushes to reduce the performance overhead further.

We designed two kinds of approaches - intra-flush and inter-flush. Intra-flush approaches apply in the time interval between two consecutive flushes. Inter-flush approaches are applied at flush points. For the intra-flush optimizations, we found that the absolute and relative memory demand of translated code, auxiliary code and data structures depends on the path selection (trace selection and linking) strategy used by the DBT. The performance of the DBT also depends on the path selection strategy. Therefore, we evaluated a comprehensive set of path selection strategies to propose a strategy for both holistic memory efficiency and performance. We also found that the data structure size has been ignored in previous research, and incorporating data structure size into the memory demand leads us to better designs. We then recognized that for any given path selection strategy, the exit stub size can be further reduced without impacting any of the other memory demand components. We designed several optimizations to reduce the exit stub code size, exit stub

data size and exit stub count. Next, we explored two different flushing strategies in the inter-flush category. We were motivated to explore flushing strategies based on our finding that there are two major categories of code in terms of lifetime - traces that live less than 10% of the total execution time and traces that live more than 90% of the total execution time. Our goal was to preserve the latter across flush boundaries while discarding the former. We explored generational cache flush in which the code cache was divided into two generations to contain the two different kinds of code. We developed a time-based heuristic and an execution count-based heuristic to classify traces. We also designed a unified cache flushing strategy. We used a pseudo-LRU heuristic to classify traces. We developed this strategy to be uniformly applicable across single-threaded and multi-threaded guest applications. We also combined all these approaches into a single system. We chose the path selection, exit stub optimization and flushing strategy that we found to be the most effective, to integrate into the system. During integration, we found several conflicts among the strategies and devised techniques to handle them.

We evaluated our techniques on two different embedded platforms. We used a single-core processor for our evaluations on single-threaded guest applications. We used a processor supporting two hardware thread contexts for our evaluations on multi-threaded applications. We performed all our optimizations by direct, source-code modification of an industrial-strength DBT. We used three different sets of benchmarks (including embedded and multi-threaded benchmarks) to serve as our guest application programs.

We found that path selection can improve memory efficiency of the whole system by 20% on the average and performance by 5-20%. We found that exit stub optimizations can reduce the code cache size from 37-44% and the performance by 5-6%. The generational cache flush technique used a size-limited, temporary generation and an unlimited permanent generation. It reduced the code cache size (sum of the temporary and permanent generation sizes) by 20-25%, without any significant performance degradation. Unified cache flushing placed a size limit on the entire code cache and improved the performance of single-threaded benchmarks by 17% and that of multi-threaded benchmarks by 15%. The combined system exhibited memory savings from 30%, 37%, and 36% for short, medium and long benchmarks respectively and performance improvements of

10% and 27% for medium and long benchmarks respectively.

9.1 Merits of the Dissertation

The benefits of DBT-based applications are recognized widely. However, most of the development for DBTs has targeted general-purpose and server platforms. Since these platforms are not constrained in memory, the focus of DBT design has been traditionally on performance. We explored a memory-oriented design for DBTs, so as to make them more suitable for embedded platforms. We considered a very general, translation-based DBT and examined every aspect that contributed to the memory demand, to propose a memory-oriented design. We found that a memory-oriented design not only reduces memory pressure, but also improves performance. We expect our techniques to be applicable to all translation-based DBT systems operating in memory constraints.

We showed that path selection plays an important role in the memory efficiency of a DBT and proposed a path selection strategy to address memory efficiency as well as performance. We also showed that it is important to incorporate data structures into the total memory demand and to place a limit on the total memory demand rather than on the code cache only. We demonstrated the size impact of exit stubs across several DBTs and exploited opportunities to reduce their size. We showed that it is beneficial to give preference to reduced memory demand than to dynamic instruction count, especially for infrequently executed code such as exit stubs. We also contributed two flushing techniques. We showed execution count to be an effective heuristic for predicting trace lifetime. We also showed LRU to be effective in flushing. Finally, we designed selective flushing to be applicable to both single-threaded and multi-threaded guest applications.

The research contained in this dissertation has a broad impact because embedded systems permeate every aspect of our lives. Many of these embedded systems could benefit from DBT-based services. For example, developers of complex embedded applications such as streaming and gaming want to detect memory leaks and race conditions. Dynamic ISA translation can prove useful because there are several different embedded architectures. Even dynamic optimization is important because there is a proliferation in the number of microarchitecture implementations corresponding

to embedded architectures e.g., ARM. Interestingly, even Java-based applications that are long-running, performance-critical or require quick response can use this research. Such applications frequently use code caching to support JIT compilation or ahead-of-time compilation. Digital TV boxes, global positioning systems, and mobile phones are some devices which use such applications.

9.2 Future Work

Several future directions of research on DBTs for embedded systems are possible. These directions are discussed in the following sections.

9.2.1 Power-Efficient Designs

While memory is an important consideration for embedded systems, power is also a first-order goal in embedded system optimization. Power consumption by guest applications can be managed by DBTs by predicting the active and idle times of the guest application. DBTs can further optimize for power by taking advantage of accelerators that are available on some embedded platforms. These accelerators typically perform a small set of common functions and consume less power than a full-featured processors. However, it is difficult for the static compiler to identify code regions that can be seamlessly fitted on these accelerators. Dynamic compilation can identify these code regions more effectively because they have more up-to-date, path-specific information.

Apart from power optimization for guest applications, it is also important to focus on the power consumption of the DBT itself. Similar to memory, the DBT competes with the guest application for the battery resource. Therefore, the DBT also has to be optimized for power.

9.2.2 Client-Server DBT Architecture

Much of the DBT overhead on an embedded system can be offloaded to a server connected to an embedded device. Functions such as translation can be offloaded. Alternatively, data such as the contents of the code cache, can also be offloaded. Offloading of cached code can form an alternative

to eviction and lower the costs of retranslation. However, now the DBT will be competing with the guest application for network bandwidth as well. Also, many embedded devices being mobile, servers may not always be available or server connections may change.

9.2.3 Cache locking

Embedded architectures often provide instructions to lock cache lines into instruction and data caches. Locking in cache lines ensures that there are no cache misses on the contents that are locked in. Fewer cache misses not only implies better performance, but also lower power consumption as the traffic to the memory is reduced. DBTs are very well positioned to determine code pieces that should be locked into the instruction cache. They can adapt with phase changes to lock in the hot code corresponding to each phase. DBTs can even perform data optimizations by locking in frequently accessed data into data caches.

9.2.4 Dual Instruction Sets

Embedded platforms often provides two different instruction set - one for performance and one for better code density. For example, the ARM platform provides both the ARM and THUMB instruction sets. The ARM instruction set has better performance while the THUMB instruction set has better code density. Better code density may also lead to better cache performance because more instructions fit into a cache line, which in turn leads to lower power consumption. DBTs can adaptively translate guest applications to ARM or THUMB depending on the remaining battery power in the system, to prolong the battery life.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, San Jose, California, USA, 2006.

[2] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 280–290, Montreal, Quebec, Canada, 1998.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, Minneapolis, Minnesota, United States, 2000.

[4] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. *ACM SIGPLAN Notices*, 31(5):159, 1996.

[5] David F. Bacon, Perry Cheng, and David Grove. Garbage collection for embedded systems. In *4th ACM International Conference on Embedded Software*, pages 125–136, Pisa, Italy, 2004.

[6] Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. Fragment cache management for dynamic binary translators in embedded systems

with scratchpad. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 75–84, Salzburg, Austria, 2007.

[7] José A. Baiocchi and Bruce R. Childers. Heterogeneous code cache: using scratchpad and main memory in dynamic binary translators. In *46th Annual Design Automation Conference*, pages 744–749, San Francisco, California, 2009.

[8] José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Reducing pressure in bounded DBT code caches. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 109–118, Atlanta, GA, USA, 2008.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.

[10] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 191, Washington, DC, USA, 2003.

[11] R.C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, page 24, 1995.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Parallel Architectures and Compilation Techniques*, October 2008.

[13] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[14] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *4th Int'l Symposium on Code Generation and Optimization*, pages 28–38, Manhattan, New York, NY, March 2006.

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

[16] Derek Bruening and Saman Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *International Symposium on Code Generation and Optimization*, pages 74–85, San Jose, California, 2005.

[17] Derek Bruening and Evelyn Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pages 13–20, 2000.

[18] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, California, 2003.

[19] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[20] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[21] M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, VC Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.

[22] B.M. Cantrill, M.W. Shapiro, A.H. Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, pages 15–28, 2004.

[23] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160, Portland, Oregon, United States, 1989.

[24] H. Chen, J. Lu, W.C. Hsu, and P.C. Yew. Continuous adaptive object-code re-optimization framework. *Advances in Computer Systems Architecture*, pages 241–255, 2004.

[25] Z. Chen. *Java card technology for smart cards: architecture and programmer's guide*. 2000.

[26] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18:56–64, 1998.

[27] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Walkabout - a retargetable static binary translation framework. Technical report, 2002.

[28] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, Nashville, Tennessee, United States, 1994.

[29] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large windows/nt applications. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 80–89, Paris, France, 1996.

[30] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, St. Petersburg Beach, Florida, United States, 1996.

[31] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[32] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An on-line program specializer. In *Proceedings of the IEEE Hot Chips XI Conference*, 1999.

[33] 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.

[34] Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Nadia Tawbi, Hamdi Yahyaoui, and Sami Zhioua. A dynamic compiler for embedded java virtual machines. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 100–106, 2004.

[35] Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Hamdi Yahyaoui, and Sami Zhioua. A synergy between efficient interpretation and fast selective dynamic compilation for the acceleration of embedded Java virtual machines. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 107–113, 2004.

[36] Saumya Debray and William Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, Berlin, Germany, 2002.

[37] 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, San Francisco, California, March 2003.

[38] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In *35th International Symposium on Microarchitecture*, pages 257–268, Istanbul, Turkey, 2002.

[39] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on*

Principles of programming languages, pages 297–302, Salt Lake City, Utah, United States, 1984.

[40] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 202–211, Cambridge, Massachusetts, United States, 2000.

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

[42] D.R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 160–170, 1996.

[43] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 173–181, Dallas, Texas, United States, 1998.

[44] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 293–304, Atlanta, Georgia, United States, 1999.

[45] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for java embedded devices. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 230–238, Chicago, IL, 2005.

[46] 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.

[47] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. Code lifetime based memory reduction for virtual execution environments. In *6th Workshop on Optimizations for DSP and Embedded Systems (ODES)*, Boston, MA, March 2008.

[48] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. Balancing memory and performance through selective flushing of software code caches. In *International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES)*, Scottsdale, AZ, October 2010.

[49] Apala Guha, Kim hazelwood, and Mary Lou Soffa. DBT path selection for holistic memory efficiency and performance. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 145–156, Pittsburgh, Pennsylvania, USA, 2010.

[50] 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 *Workshop on Workload Characterization*, pages 3–14, 2001.

[51] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the ARM architecture. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 261–270, Seoul, Korea, 2006.

[52] Kim Hazelwood, Greg Lueck, and Robert Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 20–29, Dublin, Ireland, 2009.

[53] Kim Hazelwood and Mike D. Smith. Code cache management schemes for dynamic optimizers. In *6th Workshop on Interaction between Compilers and Computer Architectures*, pages 102–110, February 2002.

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

[55] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 2000.

[56] David J. Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *38th International Symposium on Microarchitecture*, pages 141–154, Barcelona, Spain, November 2005.

[57] 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, Ottawa, Ontario, Canada, 2006.

[58] 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.

[59] Vijay Janapareddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *International Symposium on Code Generation and Optimization*, pages 74–88, San Jose, California, 2007.

[60] Lizy Kurian John. More on finding a single number to indicate overall performance of a benchmark suite. *SIGARCH Comput. Archit. News*, 32(1):3–8, 2004.

[61] Wen ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gilles. Mojo: A dynamic optimization system. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

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

[63] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[64] C.J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software-Practice and Experience*, 31(8):717–738, 2001.

[65] Naveen Kumar, Bruce R. Childers, Daniel Williams, Jack W. Davidson, and Mary Lou Soffa. Compile-time planning for overhead reduction in software dynamic translators. *Int. J. Parallel Program.*, 33(2):103–114, 2005.

[66] James R. Larus and Eric Schnarr. Eel: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, La Jolla, California, United States, 1995.

[67] M. Laukkanen. Java on Handheld Devices: Comparing J2ME CDC to Java 1.1 and Java 2. *CiteSeer, NEC Research Inst*, 2001.

[68] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 137–148, Philadelphia, Pennsylvania, United States, 1996.

[69] 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.

[70] Peter S. Magnusson, Fredrik Dahlgren, Hkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrm, and Bengt Werner. Simics/sun4m: A virtual workstation, 1998.

[71] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 281–292, 1999.

[72] Sun Microsystems. *The Java HotSpot Performance Engine Architecture. White Paper.* 1999.

[73] Sun Microsystems. *KVM Porting Guide. White Paper.* 2003.

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

[75] Jason E. Miller and Anant Agarwal. Software-based instruction caching for embedded processors. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 293–302, San Jose, California, USA, 2006.

[76] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of DBT for the ARM architecture. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 147–156, Dublin, Ireland, 2009.

[77] J.W. Muchow. *Core J2ME technology and MIDP*. 2001.

[78] R. Muth, S.K. Debray, S. Watterson, and K. De Bosschere. Alto: A link-time optimizer for the Compaq Alpha. *Software: Practice and Experience*, 31(1):67–101, 2000.

[79] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, page 74. ACM, 2007.

[80] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, San Diego, California, USA, 2007.

[81] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer, 2005.

[82] J. Palm, H. Lee, A. Diwan, and J. E. B. Moss. When to use a compilation service? In *Conference on Languages, Compilers, and Tools for Embedded Systems*, Berlin, Germany, 2002.

[83] M. Poletto, D.R. Engler, and M.F. Kaashoek. Tcc: A system for fast, flexible, and high-level dynamic code generation. *ACM SIGPLAN Notices*, 32(5):121, 1997.

[84] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, 1997.

[85] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Paris, France, 1996.

[86] Sumedh Sathaye, Paul Ledak, Jay Leblanc, Stephen Kosonocky, Michael Gschwind, Jason Fritts, Arthur Bright, Erik Altman, and Craig Agricola. Boa: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 2–11, 1999.

[87] 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.

[88] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 209–218, 2002.

[89] 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, San Francisco, California, March 2003.

[90] Nik Shaylor, Douglas N. Simon, and William R. Bush. A java virtual machine architecture for very small devices. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 34–41, San Diego, California, USA, 2003.

[91] Stacey Shogan and Bruce R. Childers. Compact binaries with code compression in a software dynamic translator. In *Design, Automation and Test in Europe*, page 21052, Paris, France, 2004.

[92] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 175–185, Ottawa, Ontario, Canada, 2006.

[93] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, Orlando, Florida, United States, 1994.

[94] P. Stears. Emulating the x86 and DOS/Windows in RISC environments. In *Proceedings of the Microprocessor Forum*, 1994.

[95] E. Traut. Building the virtual PC. *Byte*, 22(11):51–2, 1997.

[96] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 41–51, 2000.

[97] Emmett Witchel and Mendel Rosenblum. Embra: fast and flexible machine simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 68–79, Philadelphia, Pennsylvania, United States, 1996.

[98] Q. Wu, M. Martonosi, D. W. Clark, V. Janapareddi, 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, Barcelona, Spain, 2005.

[99] L. Zhang and C. Krintz. Adaptive unloading for resource-constrained VMs. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, Washington, DC, 2004.

[100] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 15–26, Saint Malo, France, 1997.

[101] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. *Computer*, 33:47–52, 2000.

[102] 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, Chicago, IL, USA, 2005.