DBT Path Selection for Holistic Memory Efficiency and Performance

Apala Guha   Kim Hazelwood   Mary Lou Soffa
Department of Computer Science
University of Virginia

Abstract

Dynamic binary translators (DBTs) provide powerful platforms for building dynamic program monitoring and adaptation tools. DBTs, however, have high memory demands because they cache translated code and auxiliary code to a software code cache and must also maintain data structures to support the code cache. The high memory demands make it difficult for memory-constrained embedded systems to take advantage of DBT-based tools. Previous research on DBT memory management focused on the translated code and auxiliary code only. However, we found that data structures are comparable to the code cache in size. We show that the translated code size, auxiliary code size and the data structure size interact in a complex manner, depending on the path selection (trace selection and link formation) strategy. Therefore, holistic memory efficiency (comprising translated code, auxiliary code and data structures) cannot be improved by focusing on the code cache only. In this paper, we use path selection for improving holistic memory efficiency which in turn impacts performance in memory-constrained environments. Although there has been previous research on path selection, such research only considered performance in memory-unconstrained environments.

The challenge for holistic memory efficiency is that the path selection strategy results in complex interactions between the memory demand components. Also, individual aspects of path selection and the holistic memory efficiency may impact performance in complex ways. We explore these interactions to motivate path selection targeting holistic memory demand. We enumerate all the aspects involved in a path selection design and evaluate a comprehensive set of approaches for each aspect. Finally, we propose a path selection strategy that reduces memory demands by 20% and at the same time improves performance by 5-20% compared to an industrial-strength DBT.

Categories and Subject Descriptors D.3 [*Processors*]: Code generation, Compilers, Incremental compilers, Optimizations, Runtime environments

General Terms Design, Experimentation, Measurement, Performance

Keywords Dynamic binary translation, Memory management, Embedded systems, Virtual machines, Path selection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-910-7/10/03...$10.00

1. Introduction

The capability of DBTs to monitor and translate the guest application instruction stream can be leveraged for many uses such as runtime security [6, 20, 23], dynamic optimization [3] and dynamic instrumentation [25]. While these uses are important across all platforms, some DBT uses are particularly important in the embedded context. For example, DBTs can manage power by exploiting system calls to configure hardware units according to the needs of the guest application. DBTs can also complement static compilers for scratchpad memory management. However, it is necessary to minimize the memory impact of DBTs because embedded systems such as PDAs have small memories (an order of magnitude smaller than general purpose systems) while embedded applications are increasingly becoming larger and more complex. Additionally, PDAs are OS-based and support multitasking, both of which increase the burden on the memory system. The specific functionality targeted by the DBT may have memory requirements of its own. For example, runtime security intended to protect certain memory regions must store instrumentation code for every memory load and store. Therefore, it is necessary to optimize the core DBT memory footprint to enable a wide variety of applications on PDAs.

The three sources of DBT memory footprint are 1) translated code, 2) auxiliary code, and 3) data structures. DBTs modify and translate code (for example, inserting security checks) and cache the translations in a software code cache to amortize the translation overhead. Control has to be transferred back and forth between the translator and the code cache for creating new translations and for locating existing translations. DBTs cache auxiliary code with translated code to facilitate the control transfers, adding to the code cache size. DBTs use data structures to keep track of the code cache contents. DBTs patch translated control transfer instructions (CTIs) to point directly to translated code segments and reduce transfers to the translator, thus forming paths. Data structures also keep track of these *links* between translated code segments.

The sizes of the memory demand components of a DBT depend on path selection which denotes the way that code is selected for each translation and the way translated code is linked. The various aspects of the path selection strategy as well as the holistic memory efficiency impact the performance in memory-constrained environments. Therefore, our goal is to design a path selection strategy that holistically optimizes all three memory sources without degrading performance. For example, an aspect of path selection is whether code is translated speculatively or non-speculatively. While speculation may improve performance by reducing control transfers to the translator, it may use code cache and data structure space for code that will never be executed. The increased space usage may require the code cache to be flushed (to reclaim space) more frequently, offsetting the performance benefit of speculation. We discuss path selection in more detail in the following sections.

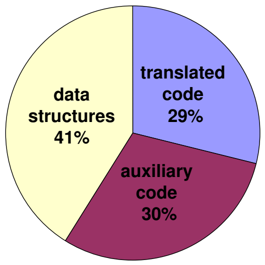


Figure 1. Memory distribution among translated code, auxiliary code, and data structures. The results are averages taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin on ARM [25].

It is challenging to design a path selection strategy for improved holistic memory efficiency because there are 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 the improved memory efficiency may improve performance.

We explore these complex interactions in this paper to motivate the design of path selection strategies targeting holistic memory demand. We enumerate all the aspects involved in a path selection design and evaluate a comprehensive set of approaches for each aspect. Finally, we propose a path selection strategy for both memory efficiency and performance in memory-constrained environments. The proposed path selection strategy specifies 1) the granularity of a translation, 2) whether code should be speculatively selected for translation, 3) how informed the speculation (if used) should be, 4) when a translated region should be terminated, and 5) how translated code should be linked.

Path selection exposes complex interactions between the components that comprise the memory demand. For example, fine-grained translations (such as single basic blocks) increase the number of translated units and the amount of data structures required for bookkeeping. However, fine-grained translations promote code reuse (and reduce code duplication) if the code appears in multiple program paths. Fine-grained translations also give rise to more points at which control may need to be transferred to the translator, and therefore increase the amount of auxiliary code. Similarly, the linking policy also impacts the holistic memory demand in a complex manner. Early (*proactive*) linking of translated code may create links that are never used, and allocate data structures for recording such links. Late (on-demand or *lazy*) linking, however, does not allocate unnecessary data structures but requires more functionality in auxiliary code, increasing the auxiliary code size.

Path selection also impacts performance in a complex manner. For example, lazy linking will increase the number of control transfers between the code cache and the translator, which may degrade performance. At the same time, in memory-constrained environments, performance is impacted by the holistic memory demand because the code cache must be flushed to free memory space on reaching the memory limit, leading to retranslations which create performance degradation. Therefore, the memory savings offered by lazy linking offsets the context switch overhead in memory-constrained environments.

Previous work on DBT memory management has not investigated path selection and has focused only on the memory demands of translated code and auxiliary code [1, 2, 11, 12, 16]. However, such approaches are not sufficient because they ignore the data structures which are comparable to the code cache in size [21]. As

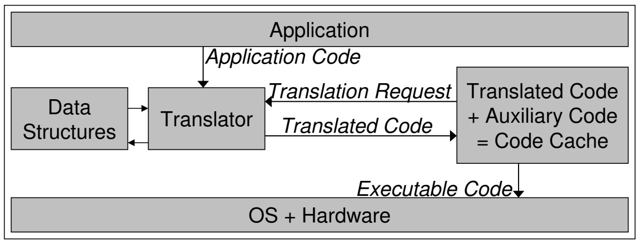


Figure 2. Block diagram of a typical translation-based DBT. The translator is the core of the DBT. It caches its translations and uses data structures to manage the translations.

shown in Figure 1, we experimentally found that on average, translated code and auxiliary code constitute 59% of the total memory demand while data structures constitute 41%, in Pin, an industrial-strength DBT. Similar data on the code cache components has been found across different DBTs [2, 8]. We also found that factoring in data structure sizes impacts the relative memory demand of path selection strategies. Previous approaches placed the memory limit on the code cache only [1, 2, 12, 16], leading to inaccurate interpretation of performance results. Additionally, previous work has researched both code selection [10, 18, 19] and linking strategies [3, 6, 9, 22, 25] but from the performance perspective only. Such approaches do not consider the memory demands of path selection or its performance in memory-constrained environments.

We make the following contributions in this paper:

- We provide conceptual and experimental motivation for considering the holistic memory demands of DBT path selection.
- We enumerate the design axes involved in path selection and evaluate a comprehensive set of choices for each axis.
- We enumerate and experimentally evaluate the tradeoffs among the memory demand components.
- We evaluate whether performance improvement due to better holistic memory efficiency outweighs the performance degradation due to individual aspects of path selection.
- We propose a path selection strategy to improve holistic memory demand as well as performance.

We provide background on DBTs in Section 2. We describe the various aspects of path selection and their implications in Section 3. We evaluate path selection strategies and propose a strategy for holistic memory efficiency and performance in Section 4. Finally, we present related work in Section 5 and conclude in Section 6.

2. Background

Figure 2 is a simplified diagram of a translation-based DBT. The core of the DBT is a translator responsible for translating the guest application code dynamically. The translator caches translated code which executes natively from the software code cache. Code is translated into program traces containing one or more basic blocks. These traces have a single entry and one or more exits, depending on the number of basic blocks in the trace.

Code translation is performed on demand and requests have to be generated for translations of new code. There are repeated context switches between the translator (for translation of new code) and the code cache (for execution of translated code). The context switch involves saving and restoring state. DBTs create and cache *exit stubs* to facilitate context switches. Exit stubs are constructs that translated control transfer instructions (CTIs) initially target. Exit stubs are the main constituent of auxiliary code, consisting of cached code that is not part of the guest application code.

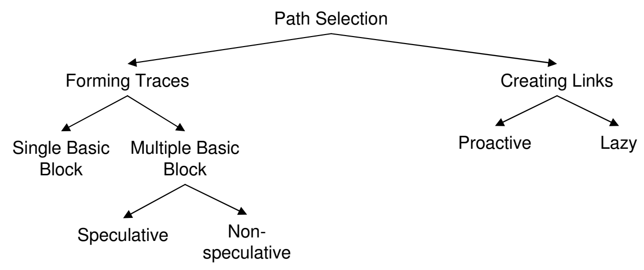
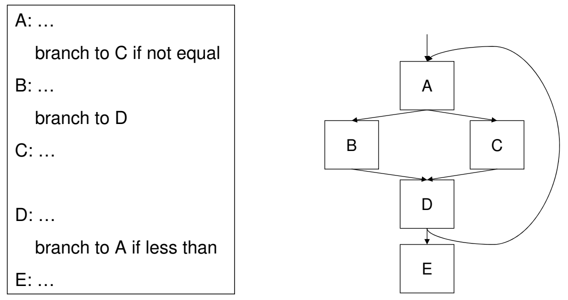


Figure 3. Topology of the path selection choices.



```
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 4. Section 3 describes the different path selection strategies by applying them to this code snippet.

Although exit stubs are crucial for correct functionality, there is a performance penalty to context switch to the translator every time a CTI is executed. Thus, CTIs are patched to directly point to their target code if available in the code cache in a process known as *linking*. Linking is possible only for a *direct* CTI, i.e., a CTI whose target does not change during the application execution.

DBTs also use data structures, as shown in Figure 2. The main data structure is a code cache directory, which stores an entry for each cached trace. Each entry contains the original program address and the corresponding code cache address of a trace. The translator searches the directory for an existing translation in the code cache before translating code at a requested program address. Data structures also record how traces are linked, since CTIs may need to be unlinked if the target trace is ever removed from the code cache (flushed). The link data structure contains the source and and corresponding exit stub addresses of the link. Lists of incoming and outgoing links of a trace are associated with its code cache directory entry to facilitate removal of a trace and the corresponding modification of incoming edges.

3. Path Selection

Path selection determines how code is selected to form a trace and how traces are linked to each other. Figure 3 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 CTI 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 CTI 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 4(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 4(b) shows the control flow graph corresponding to Figure 4(a).

3.1 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 5 shows two possible trace formations when the translator attempts to translate A from Figure 4. Figure 5(a) shows a single-block trace starting at A. Figure 5(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 5(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 5(b), there is only one off-trace branch for A, while in Figure 5(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 5, 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.

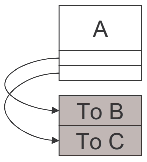
In summary, single-block traces reduce code duplication but increase the proportion of auxiliary code and data structures. Regarding performance, single-block traces suffer from lower code cache locality and higher context switches.

3.2 Multi-Block Trace Selection and Termination

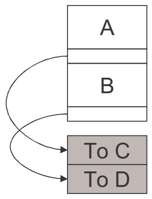
Figure 5(b) shows that the translator chose to translate B to extend the trace starting at A. Several other choices are possible for a multi-block trace. In this section we discuss strategies for forming a multi-block trace.

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 by speculating the next basic block. In Figure 6(a), we execute A and find that B is the next basic block to execute. We append B to A. In Figure 6(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.

We use two strategies for speculative trace selection. In the first strategy, we use data gathered in an offline profiling run to

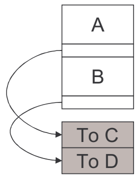


(a) Single-block trace.

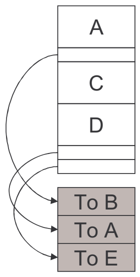


(b) Multi-block trace.

Figure 5. 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 CTI targets.



(a) Extending trace non-speculatively.



(b) Extending trace speculatively.

Figure 6. Extending a trace. White areas are parts of the trace while gray areas are exit stubs.

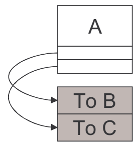
Basic Block Type	**Average No. of CTIs pointing to Basic Block**
Fall throughs of conditional CTIs	0.082
Targets of conditional CTIs	1.429
Targets of unconditional CTIs	3.219

Table 1. Table showing the average number of CTIs pointing to different types of basic blocks. The results are averages taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin [25].

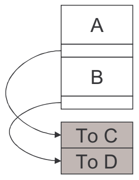
speculate which way a branch will go. We speculate about a CTI only if it shows a particular bias for at least 90% of its executions in the profiling run. In the second strategy, we translate a contiguous stream of code until the trace size reaches a certain threshold size or it encounters an indirect or direct, unconditional CTI. Such a strategy is equivalent to speculating that no conditional CTI will be taken. This strategy would, however, speculate that B follows A and trace the path AB. The speculative strategy using profiling is highly informed, while the second strategy uses minimum information.

There are more context switches in forming non-speculative traces as these are translated one basic block at a time. However, if the speculation is incorrect, there will be wasted space for translated code and data structures.

Trace Termination. After translating each basic block, the translator must determine whether to extend the trace further. 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 produce the trace in Figure 7(a). However, if B always follows A, we should produce the trace in Figure 7(b).



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



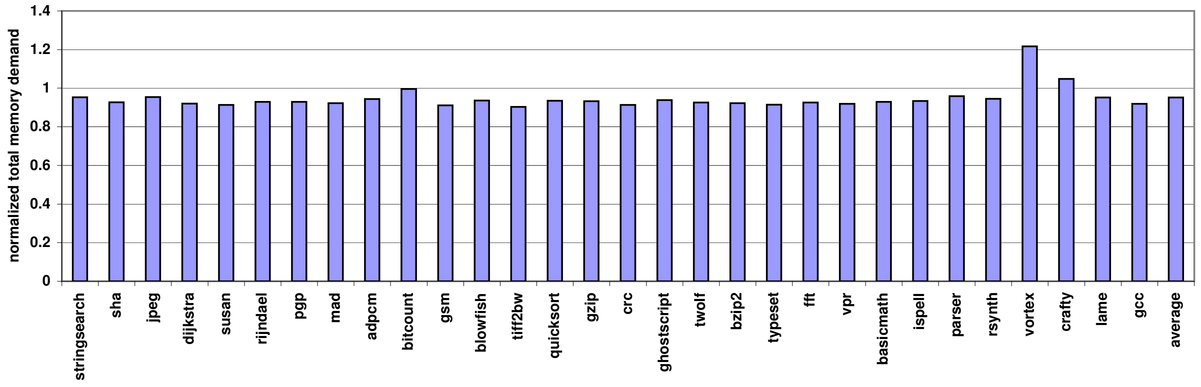
(b) Terminating trace when the next basic block, B, appears only on this path.

Figure 7. Termination of a trace based on the number of different paths the next basic block appears in. White areas are parts of the trace while gray areas are exit stubs.

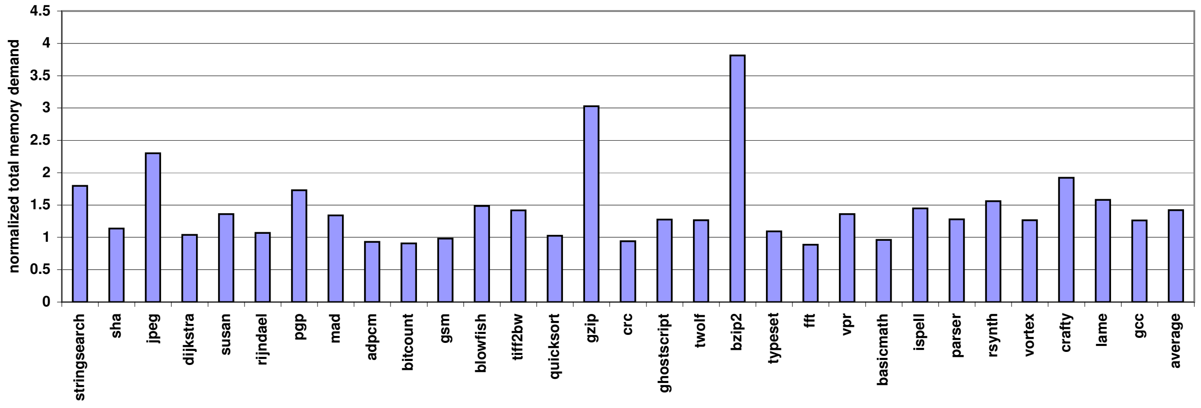
We experimentally explore the trace termination condition here because the number of program paths executing a basic block is independent of other aspects of path selection. We hypothesized that the number of program paths in which the next basic block appears (i.e., the number of CTIs that point to the next basic block) is correlated with the type of the CTI ending the last basic block translated. Indeed, as shown in Table 1, we found that if the next basic block is a fall-through of the direct, conditional CTI ending the last basic block, it is rarely targeted by CTIs, while if the next basic block is a target of a direct CTI ending the last basic block, it is usually pointed to by other CTIs also. Therefore, we terminate traces based on CTI type i.e., whether it is a not taken direct conditional CTI, taken direct conditional CTI or a direct, unconditional CTI.

To confirm our hypothesis, we measured the DBT memory requirements when 1) not taken direct conditional CTIs are *elided* to the trace (Figure 8(a)), 2) taken direct conditional CTIs are elided to the trace (Figure 8(b)), and 3) direct unconditional CTIs are elided to the trace (Figure 8(c)). We then compared the results with single-block traces (essentially, not eliding any CTI). If eliding any of the above categories of CTIs has better memory efficiency than the baseline, then it is considered worth eliding. Figure 8(a) shows the results of eliding not taken conditional CTIs. Memory efficiency improves in almost all the benchmarks with an average 5% improvement. Figure 8(b) shows the results of eliding taken conditional CTIs. Memory efficiency degrades 41% on average. Figure 8(c) shows the results of eliding unconditional CTIs. Memory efficiency degrades 14% on average. Although more CTIs point to targets of unconditional, direct CTIs, the degradation from eliding unconditional, direct CTIs is smaller because conditional, direct CTIs are higher in number. Therefore, it is beneficial to elide only direct conditional CTIs that are not taken, which matches the data in Table 1.

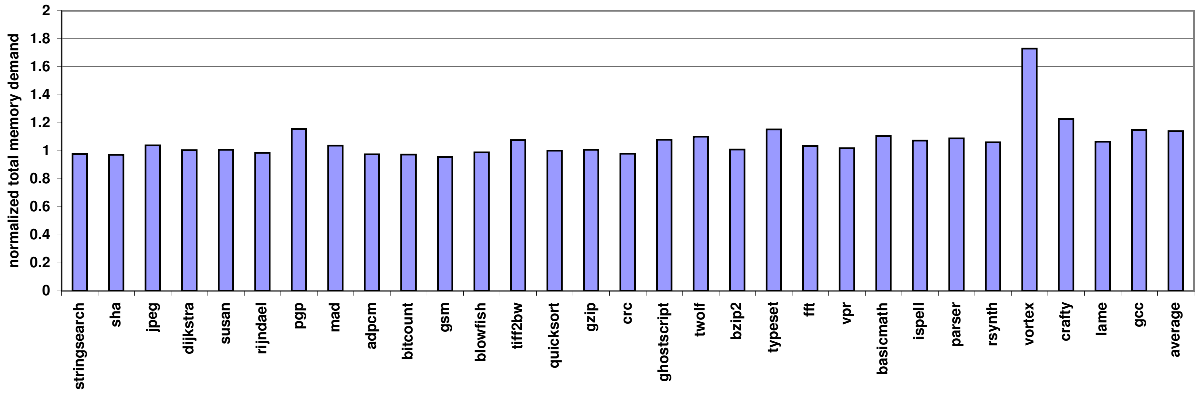
These evaluations modify the trace selection strategies such that 1) non-speculative trace selection terminates traces at taken, direct CTIs, and, 2) speculative, profile-based trace selection terminates traces at direct CTIs predicted to be taken. All of these trace selection strategies continue to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



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

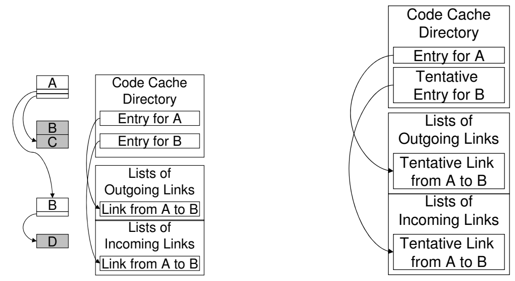


(b) Normalized memory requirements of benchmarks when *taken conditional* control transfers are elided. Memory efficiency degrades 41% on average.



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

Figure 8. Normalized memory demand of terminating traces at different types of CTIs. The baseline is single-block traces. The results are taken over the SPEC2000 integer and MiBench embedded benchmark suite, hosted by Pin [25].



(a) Proactive linking applied to the trace being translated when the branch target is cached.

(b) Proactive linking applied to the trace being translated when the branch target is not cached.

Figure 9. Proactive linking of a trace being translated.

3.3 Link Formation

Links can be formed proactively or lazily. Proactive linking places the link as soon as the source and target traces are cached. When a trace is translated, a **proactive linking** configuration examines each of the trace's outgoing branches. If the branch targets are already in the cache, the branches are immediately linked to the targets. For example, as shown in Figure 9(a), basic block A is being translated into a single-block trace. One branch in trace A needs to link to target basic block B. The target B is already in the code cache. Therefore, the branch is immediately linked to B. The link is registered with the code cache directory entries of its source A and target B. However, all such registered links may not be traversed. If the target B is not already cached, as shown in Figure 9(b), proactive linking registers a tentative code cache directory entry for B (if not already registered) and then registers the tentative link with the directory entries corresponding to A and B. It is therefore ensured that, whenever B is translated, all tentative links to it will be immediately put in place. Tentative data structures occupy space and may never get associated with a translation. Also, the usage of tentative entries implies that there is a time gap between recording the entries and the actual linking. So, at the time of linking, checking has to be done to determine whether the source trace for the link still exists or not. To check whether the source trace still exists, each trace is given a identifier which is unique over the entire execution. The source trace can be verified in the code cache directory using the identifier. Therefore, proactive linking needs extra memory for trace identifiers.

Lazy linking creates a link only when the corresponding path executes for the first time. For lazy linking, link entries are created when the link is needed. Therefore, lazy linking does not need tentative entries or trace identifiers. Since link entries are not created beforehand, the exit stub has to tell the translator the location of the CTI that is requesting to be linked. Since the exit stub stores the CTI location, it is larger compared to the exit stubs used by proactive linking. Thus, the auxiliary code size is larger for lazy linking than for proactive linking.

In summary, lazy linking needs less data structure space than proactive linking. However, lazy linking results 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.

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 2. Trace selection strategies and their descriptions.

4. 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 4.1 followed by results in Section 4.2.

4.1 Experimental Setup

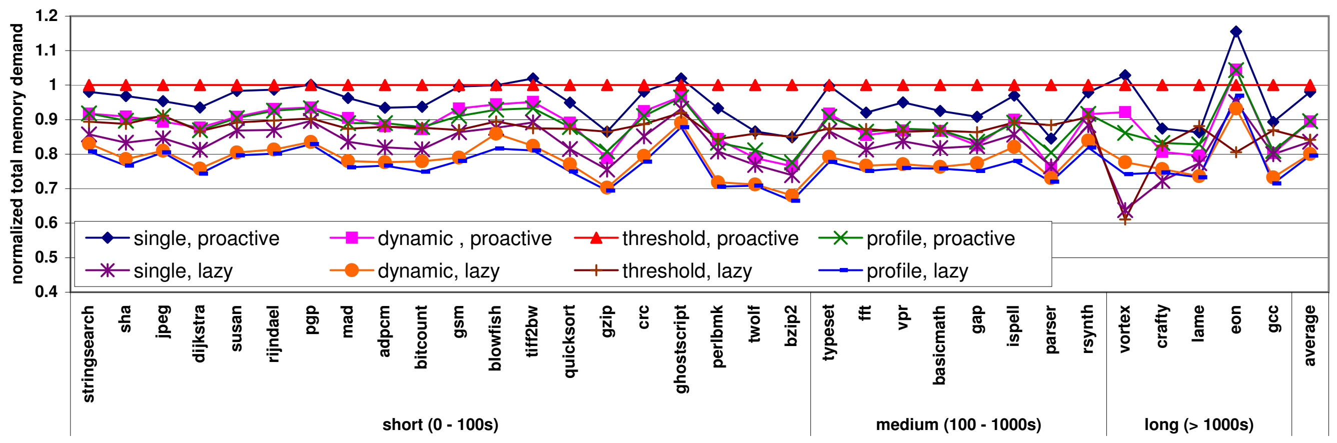
We evaluated both the memory and performance effects of the different path selection strategies. For memory effects, we measured the sum of the space occupied by the translated code, auxiliary code and the data structures. For performance, we measured the execution times of applications hosted by a DBT. In one of the performance experiments, 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 choose a baseline path selection strategy for comparison, which selected a contiguous chunk of code until the trace reached a size threshold or an unconditional or indirect CTI. For linking, the baseline used proactive linking. This trace selection strategy is one of our two speculative strategies. We chose this baseline because it is used by Pin [25], a production-quality DBT.

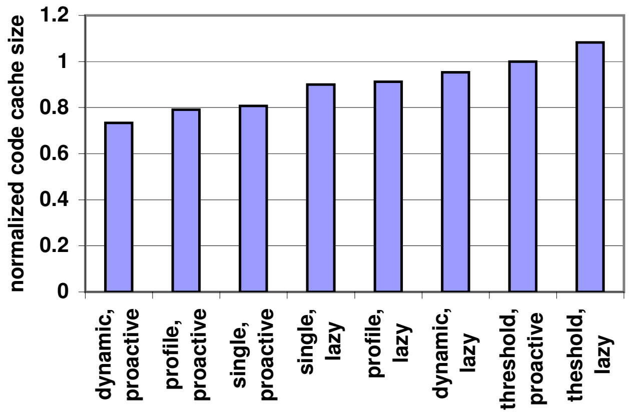
We used Pin for XScale [14] as our DBT. We implemented our strategies by directly modifying the Pin source code. However, our findings apply to other DBTs because the relative proportions of translated code, auxiliary code, and data structures are reported to be similar [2, 8]. Although Pin is generally 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 [17] and MiBench embedded benchmark [13] 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.

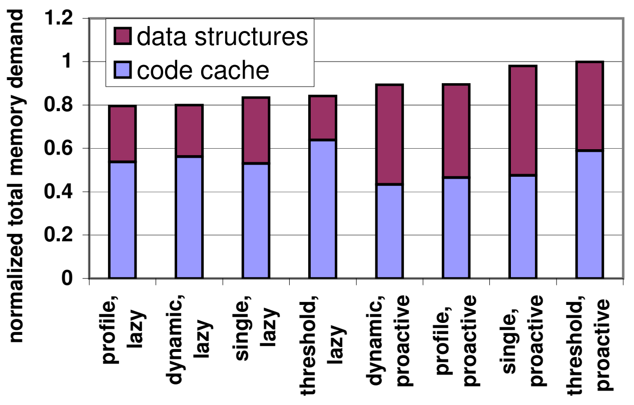
We divide the benchmarks into three groups - short-running, medium-length and long-running, according to their baseline execution times. The short-running benchmarks have execution times of less than 100 seconds. The medium-length benchmarks execute between 100 to 1000 seconds. The long-running benchmarks 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



(a) Normalized memory demand of the benchmarks, with the benchmarks being arranged in increasing order of 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 10. Normalized memory demands of different path selection strategies, with `threshold-based selection, proactive linking` as the baseline.

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.

4.2 Memory and Performance Evaluation

We compare the memory efficiency and performance of our path selection strategies in this section. We present the results on memory efficiency in Section 4.2.1 followed by the results on performance in Section 4.2.2. We discuss the results and propose a path selection strategy in Section 4.2.3. In the graphs, where applicable, the benchmarks are arranged in increasing order of baseline execution time. We use the nomenclature shown in Table 2 for the path selection strategies. Each of the path selection strategies will be combined with both lazy and proactive linking.

4.2.1 Memory Efficiency

Figure 10 shows the normalized memory demands of the benchmarks. Figure 10(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 10(b) and Figure 10(c). Figure 10(b) shows how the memory efficiency would rank the strategies if we consider the code cache only, while Figure 10(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 10(b) and Figure 10(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 small 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 is highly accurate speculation because neither waste space. 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 10(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.

4.2.2 Performance

The normalized performance of the short-running, medium-length and long-running benchmarks are shown in Figure 11(a), Figure 11(b) and Figure 11(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 12 shows that similar results were obtained with a 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 13, 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.

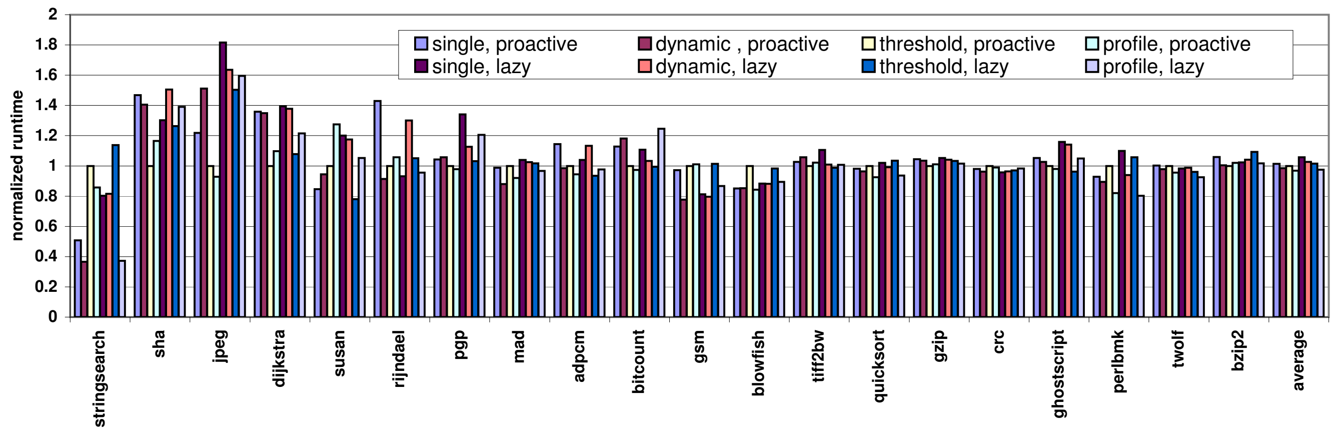
The following is a summary of the performance evaluation:

- 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 13) 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.

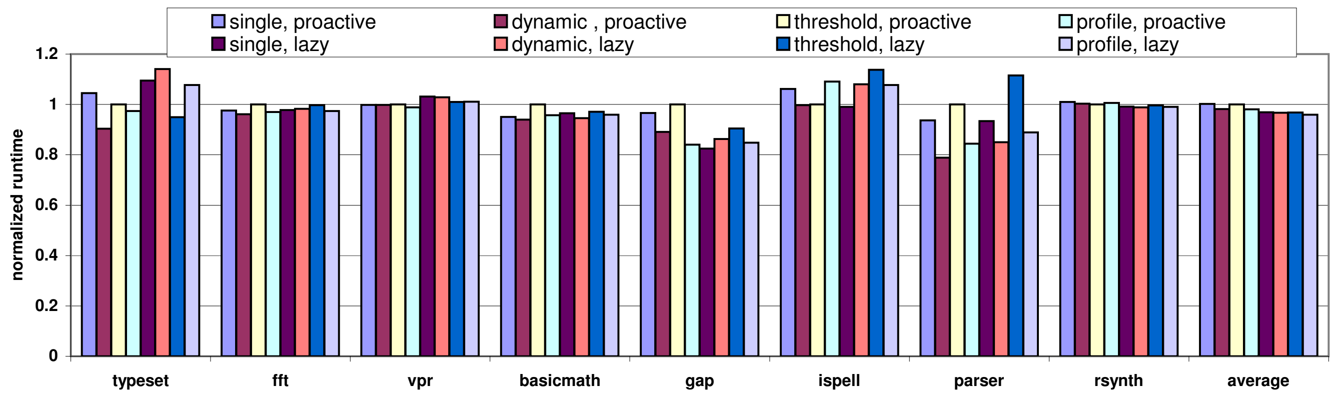
4.2.3 Discussion

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.

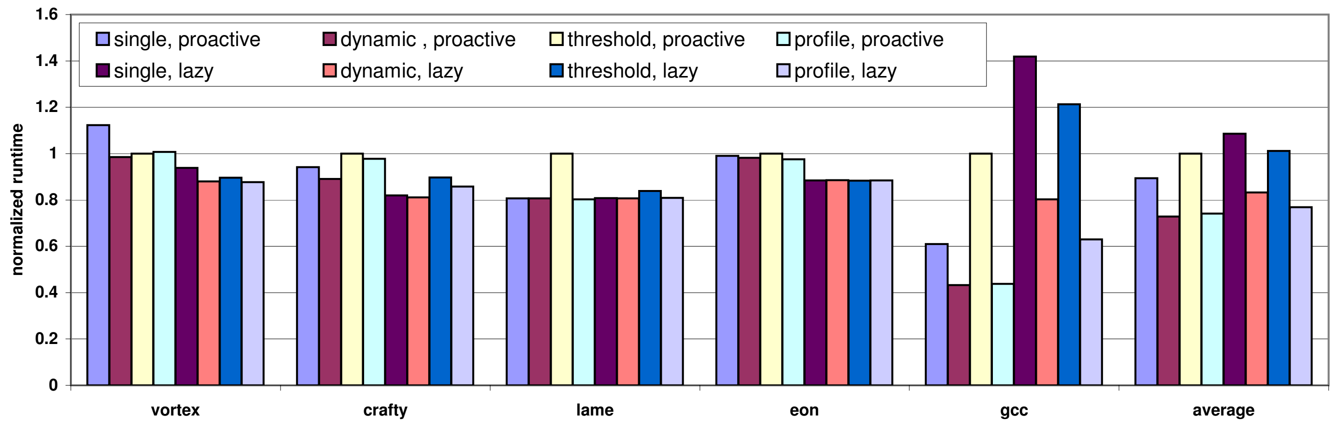
We also found that the linking strategy has the most effect on memory efficiency followed by the number of basic blocks in the



(a) Normalized performance of short-running benchmarks.

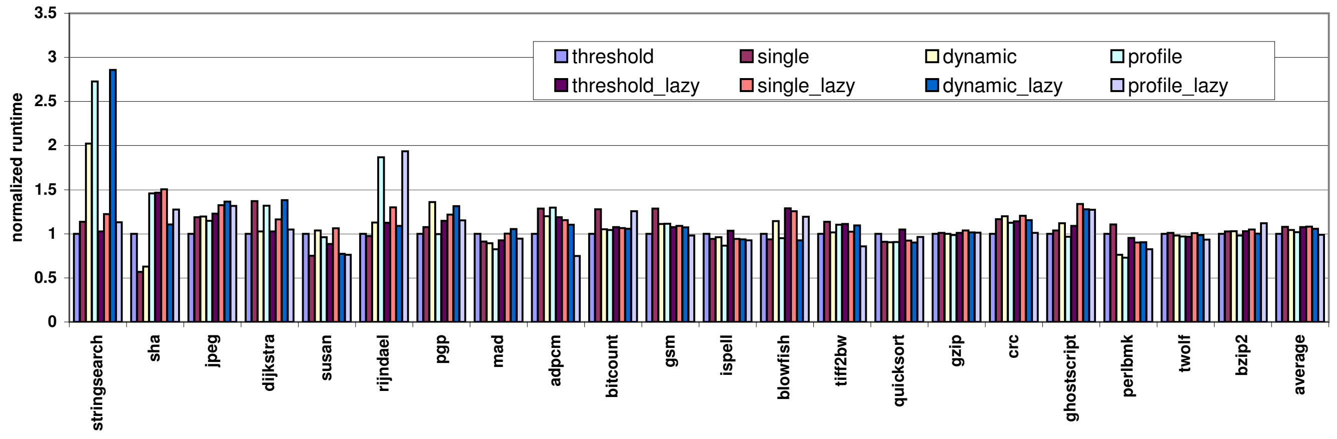


(b) Normalized performance of medium-length benchmarks.

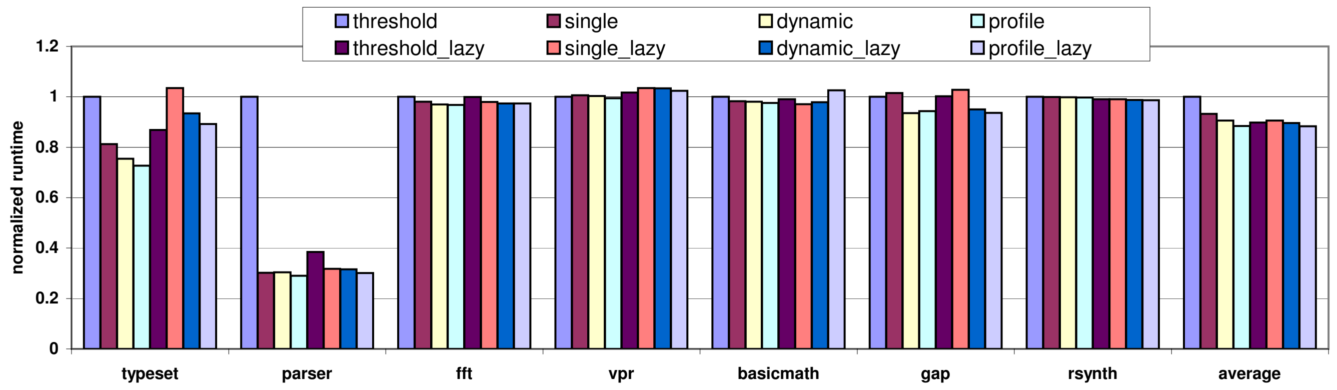


(c) Normalized performance of long-running benchmarks.

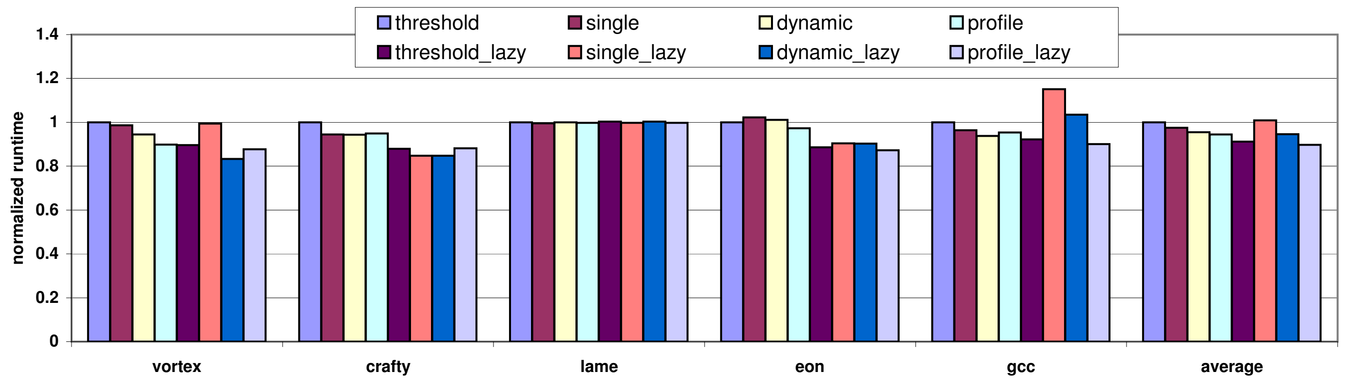
Figure 11. Normalized performance of different path selection strategies for the different benchmark categories, with `threshold-based selection, proactive linking` as the baseline.



(a) Normalized performance of short-running benchmarks.



(b) Normalized performance of medium-length benchmarks.



(c) Normalized performance of long-running benchmarks.

Figure 12. Normalized performance of different path selection strategies, with `threshold-based selection, proactive linking` as the baseline and 512 KB as the uniform memory limit.

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.

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%.

5. Related Work

Several DBTs have been developed for general-purpose machines, including DynamoRIO [3, 6], Strata [19, 20, 28], Valgrind [27] and Pin [25]. These DBTs provide features such as optimization [3], instrumentation [25] and security [6, 20, 23]. Most DBTs support general-purpose computing platforms. Pin [25], DELI [9] and Strata [1, 2, 26] are DBTs that support embedded platforms.

For most DBTs, the trace is the unit of choice compared to functions and methods [3, 6, 9, 22, 25, 27, 28]. Indeed, the trace was found to perform well and to be easy to compile [5]. Therefore, we chose traces as the unit of compilation in this paper.

Next-Executing-Tail (NET) [10] 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. Pin [14, 25] also uses straight-line code. However Pin's strategy is really the threshold-based strategy that we use in this paper. We found that just placing a threshold on the size of a trace is not sufficient. A trace needs to be terminated depending on the CTI type.

NET uses interpretation [3] or caches code as basic blocks [6] before it builds a trace. However, we did not use interpretation or basic block caching in our system because DBTs such as Strata [19, 24] 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.

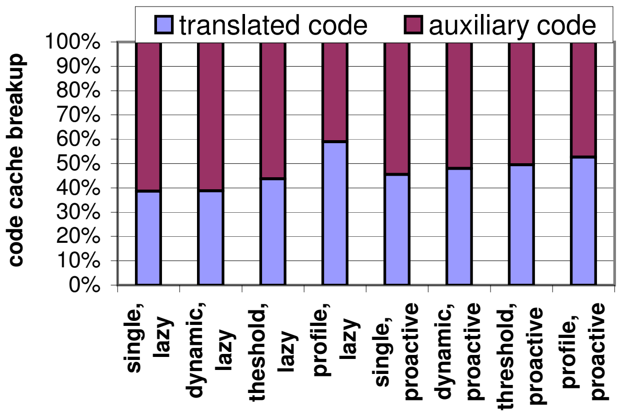


Figure 13. The division between translation code and auxiliary code within the code cache.

For the linking strategy, most DBTs use proactive linking [3, 6, 25]. Some DBTs use a deferred linking policy [9, 22] 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. Surprisingly, however, Strata for embedded systems [1, 2] 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. However, we found that lazy linking is the best option in a memory-constrained situation.

Apart from trace selection and linking strategies, memory management policies have been studied before. However, either memory management was done for performance or a holistic view of memory usage was not taken. For example, Dynamo [3] triggers a cache flush when the rate of trace generation becomes too high, to improve performance. Strata for embedded systems [1, 2] flushes on reaching a memory limit, but does not consider data structure size. Similarly, Pin for the ARM architecture [14] does not consider data structure size when triggering a flush. DynamoRIO [4, 7] uses thread-shared software code caches to reduce code expansion and also dynamically detects the working set size. But they manage the code cache only for consistency events such as self-modifying code and not for capacity. Also, they only scale up the code cache limit adaptively, which may not be suitable in a memory-constrained environment. Code cache eviction schemes have been studied before [15, 16, 22]. Our trace selection strategies for improved memory efficiency are meant to complement such strategies.

6. Conclusions

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 because 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 because the increase in duplication 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.

References

[1] J. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser, and J. 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.

[2] J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. 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.

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

[4] D. Bruening and S. 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.

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

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, California, 2003.

[7] 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.

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

[9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *35th International Symposium on Microarchitecture*, pages 257–268, Istanbul, Turkey, 2002.

[10] E. Duesterwald and V. 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.

[11] A. Guha, K. Hazelwood, and M. L. 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.

[12] A. Guha, K. Hazelwood, and M. L. 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.

[13] 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.

[14] K. Hazelwood and A. 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.

[15] K. Hazelwood, G. Lueck, and R. 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.

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

[17] J. L. Henning. Spec cpu2000: Measuring CPU performance in the new millennium. *Computer*, 2000.

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

[19] 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.

[20] 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.

[21] V. Janapareddi, D. Connors, R. Cohn, and M. 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.

[22] W. ke Chen, S. Lerner, R. Chaiken, and D. Gilles. Mojo: A dynamic optimization system. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

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

[24] N. Kumar, B. R. Childers, D. Williams, J. W. Davidson, and M. L. Soffa. Compile-time planning for overhead reduction in software dynamic translators. *Int. J. Parallel Program.*, 33(2):103–114, 2005.

[25] 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.

[26] R. W. Moore, J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. 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.

[27] N. Nethercote and J. 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.

[28] 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.