Heterogeneous Code Cache: Using Scratchpad and Main Memory in Dynamic Binary Translators

José A. Baiocchi
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
baiocchi@cs.pitt.edu

Bruce R. Childers
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
childers@cs.pitt.edu

ABSTRACT

Dynamic binary translation (DBT) can be used to address important issues in embedded systems. DBT systems store translated code in a software-managed code cache. Unlike general-purpose systems, embedded systems often have specialized memory resources, such as a fast scratchpad memory, that can be used to mitigate DBT performance overhead. This paper presents the *Heterogeneous Code Cache* (HCC), a code cache split among scratchpad and main memory. We explore several HCC management policies and show that, on average, an HCC outperforms a code cache allocated only to scratchpad or only to main memory.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Incremental compilers, Interpreters, Optimization, Run-time environments*

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Dynamic Binary Translation, Scratchpad, Software Caching

1. INTRODUCTION

Embedded systems continue to increase their capabilities to support more demanding applications: multimedia, image recognition, advanced online signal processing, etc. Their evolution creates new challenges (e.g., security, reliability, etc.), while traditional constraints (e.g., performance, memory, real-time, energy) are still important. Many of these challenges have been successfully addressed in general-purpose systems by Dynamic Binary Translation (DBT). DBT is a technology that allows control over the execution of a program by processing every instruction before it is executed. DBT uses include virtualization [1], optimization [5], software caching [13] and code compression [8].

Despite the potential, DBT use in embedded systems has been limited due to performance and memory constraints. To ensure good performance, a dynamic binary translator stores translated code in a software-controlled memory buffer, called the *code cache* (CC). The CC is usually placed in main memory and large enough to capture an application's working set. An unconstrained CC minimizes DBT's *base performance overhead* – i.e., overhead in the absence of complex code transformations – to 4% on average [12]. An unconstrained CC may grow to several megabytes in size for desktop applications [10] and to hundreds of kilobytes for embedded applications [3].

This paper addresses the problem of allocating and managing a CC in an embedded system with scratchpad memory (SPM). SPM is a small on-chip SRAM with low energy consumption [6]. It is fast but must be explicitly controlled by software. We propose a new approach, the *Heterogeneous Code Cache* (HCC), that splits the CC among multiple memory levels. Unlike a traditional multi-level memory hierarchy, objects in the lower levels of the HCC are not replicated in the higher levels. The goal is to exploit the fast but small SPM, while keeping the low miss rate of a large CC. We study a two-level HCC with one level in SPM and the other in main memory. Our contributions include:

- A new code cache organization (HCC) that uses both scratchpad and main memory;
- Several new HCC management policies, including novel "SPM-aware" policies, that place the most recently translated code in the scratchpad memory;
- A new heuristic to adaptively resize the HCC to minimize translation overhead while constraining HCC size;
- A thorough evaluation of our techniques with a dynamic binary translator in a simulated embedded system.

The rest of the paper is organized as follows: Section 2 describes the framework for our work. Section 3 describes an HCC organization and several solutions to design issues that impact performance. Section 4 evaluates the proposed techniques, finds those that work best and compares them to traditional CC approaches. Section 5 describes related work and Section 6 concludes.

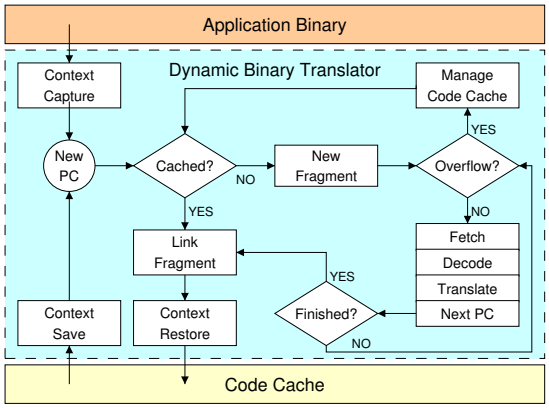

Figure 1: DBT overview

2. FRAMEWORK

We initially describe the class of embedded system targeted by our research and the operation of a typical DBT.

2.1 Target System

The techniques in this paper target embedded systems-on-chip (SoC) with a heterogeneous memory system. We assume a SoC that has a processor with L1 instruction and data caches, application-specific integrated circuits, SPM (on-chip SRAM), ROM (on-chip NOR Flash), controllers for external memories and off-chip I/O channels. External (off-chip) memories include SDRAM and NAND Flash. SDRAM is used as main memory and holds application code and data during program execution. NAND Flash is accessed through a file system and holds user files, including application binaries. System code, including the boot program and OS, is kept in ROM. SPM is *directly mapped* in the address space and has a fast access latency (1 cycle).

A dynamic binary translator can be part of the SoC's system code. It accesses application binaries from external NAND Flash. When an application is initiated, the translator loads the static data into main memory. The code is loaded on demand as execution progresses using DBT, which provides a form of *software instruction caching* [13, 4].

2.2 Dynamic Binary Translation

A high-level view of the DBT process is illustrated in Figure 1. The translator must ensure that all untranslated application instructions are loaded, examined and possibly modified prior to their execution. To do so, it must be invoked whenever new application code is requested. New code is requested when execution begins and every time a control transfer instruction (CTI) invokes an untranslated application address. Each CTI is replaced with an exit stub, called a *trampoline*, that "re-enters" the translator.

To safely re-enter the translator, the application's context must be saved to free registers for use by the translator, i.e., a *context switch* is done. The translator checks if there is translated code in the code cache (CC) for the requested application address. If so, the application context is restored and the corresponding translated code is executed. Otherwise, a new set of translated instructions, called a *fragment*, is built and saved in the CC. When translation stops, the context is restored and the new fragment is executed.

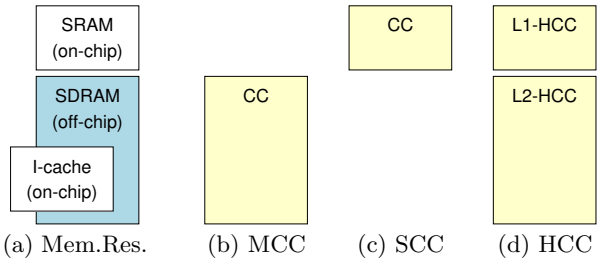

Figure 2: Code Cache Allocation Alternatives

To avoid unnecessary context switches, *fragment linking* overwrites trampolines to jump to their target fragments [5]. Indirect CTIs may not always have the same target, so they are replaced by an *Indirect Branch Translation Cache* (IBTC) lookup. The IBTC maps application addresses targeted by indirect CTIs to their translated addresses [12].

The CC is progressively filled with new fragments as execution progresses. If it is too small to hold the application's working set, a CC overflow occurs. This event is handled by a *code cache management* routine, which makes room in the CC for new fragments [10, 11, 7, 4].

3. HETEROGENEOUS CODE CACHE

In embedded systems where memory resources have different access latencies and sizes, we propose to split the CC among those resources. This arrangement introduces several CC design issues that affect performance. In this section, we describe the issues and propose solutions to address them.

3.1 Resource Allocation

We consider an embedded system with a scratchpad (on-chip SRAM) and main memory (off-chip SDRAM) serviced by an instruction cache (I-cache), as shown in Figure 2(a). The first design decision is where to allocate the CC.

One choice, shown in Figure 2(b), places the CC in main memory (MCC), so it can be relatively large to fully capture the application's translated working code set. A large CC leads to a minimal miss rate and avoids costly Flash memory reads to fetch binary code for retranslation. Another choice, shown in Figure 2(c), places the CC in SPM (SCC). The advantage to this choice is that translated instructions can be fetched in a single cycle, with the SPM serving as a *software instruction cache* [13]. The disadvantage is that the CC will be small (constrained to the SPM size), potentially leading to a high miss rate and additional Flash memory reads due to non-compulsory misses. Neither choice fully utilizes the on-chip memory resources – the first one does not use the SPM and the second one does not use the I-cache.

Our approach, the *Heterogeneous Code Cache* (HCC) exploits both SPM and I-cache. It aims to get the capacity benefit of a large CC in main memory and the latency benefit of a CC in SPM. A two-level HCC, shown in Figure 2(d), has its first level (L1-HCC) in SPM and its second level (L2-HCC) in main memory. The two levels share the lookup tables for CC metadata (translator data about fragments and trampolines). SPM is directly mapped in the address space, so the levels may not be contiguous. SPM use may help to improve the I-cache miss rate due to less potential conflicts.

3.2 Eviction Policies

When the capacity of the HCC is exhausted and space is needed for newly translated code, an *eviction policy* is used to determine what code should be removed from the HCC to make room for new code. In general, evicting one or more fragments requires updating the CC metadata, reverting links from non-evicted fragments to evicted fragments back into trampolines (*unlinking*), and invalidating IBTC entries. An effective CC eviction policy must balance the miss rate, the frequency of evictions and the cost of unlinking [11]. We propose three eviction policies, derived from general-purpose ones, that address specific HCC challenges:

FLUSH is the simplest eviction policy. It discards the whole contents of the CC on an overflow [5]. The CC is refilled when translation is restarted after an overflow. This scheme may increase the HCC miss rate, but eviction management is greatly simplified. The CC metadata can be simply discarded and unlinking is not necessary. For the HCC, FLUSH initially emits code into L1-HCC. When it becomes full, L2-HCC is filled. When both levels are full, all fragments are evicted and translation resumes in L1-HCC.

FIFO is a fine-grained eviction policy. It treats the CC as a circular buffer, evicting only the *least recently created* (LRC) fragment when space is needed. FIFO has a similar or better CC miss rate than more complex policies (e.g., LRU, LFU), but does not suffer from fragmentation and has low management overhead [10]. FIFO requires unlinking only *backward links* that target the evicted fragment. FIFO for an HCC, like FLUSH, starts filling L1-HCC and continues with L2-HCC. However, when both levels are full, only the LRC fragment in L1-HCC is evicted (rather than evicting all the code). Translation resumes in L1-HCC after an eviction. When all fragments in L1-HCC have been replaced, the fragments in L2-HCC are evicted next. The effect of this policy is that the combined levels of the HCC form a circular buffer.

Segmented FIFO divides the HCC into several *segments* of the same size. To handle CC overflows, whole segments are evicted in FIFO order [11]. This policy tries to get both FIFO's low miss rate and FLUSH's low management overhead. Evicting a group of fragments together makes more space available at once, which reduces the frequency of evictions. With Segmented FIFO only *inter-segment links* must be unlinked on eviction; *intra-segment links* do not need to be processed. We use the same segment size in both HCC levels. First, segments in L1-HCC are filled. When the L1-HCC is full, segments in L2-HCC are in turn filled. After both levels are full, the first segment in L1-HCC is flushed when space is needed. When the code in all L1-HCC segments has been replaced, eviction continues at L2-HCC.

The choice of eviction policy affects the layout of the translated code. One reason is that space must be reserved for the trampolines created when unlinking. Another reason is that when the next translated fragment does not fit in the remaining space of a CC segment, it is stored instead into the next segment. This leaves unused space at the end of the CC segments. A third reason is retranslation. A finer eviction granularity may cause a fragment to be found in the CC, but with a coarser granularity that same fragment would be retranslated at a different location after being previously evicted. These differences affect both SPM usage (depending on fragment execution frequency) and I-cache effectiveness (due to mapping conflicts).

3.3 SPM-aware Fragment Placement

The HCC policies described to this point allow new fragments to be stored into the SPM only after the L2-HCC has been completely filled or replaced. Increasing the size of the L2-HCC reduces the chances for (first-time or retranslated) fragments to be assigned to the L1-HCC, i.e., the benefit of the SPM is reduced. To address this problem, we develop a set of new "SPM-aware" management policies. These policies force the DBT to place new fragments only into the SPM.

When all new fragments are put into the L1-HCC (SPM), they have to be moved to L2-HCC (main memory) when the L1-HCC overflows. Code relocation requires the capability to fix any links that are associated with a relocated fragment. We explore both single fragment and group relocations. Relocating fragments requires redirecting links and updating IBTC entries.

Our SPM-aware policies ensure that the SPM holds the most recently translated code. The first three policies use the eviction granularity for relocation. The fourth policy relocates one fragment at a time but uses Segmented FIFO for eviction. The policies are:

FLUSH@L1: Code is initially translated into L1-HCC. When it becomes full, *all fragments* in L1-HCC are relocated to L2-HCC and the DBT starts to fill L1-HCC again. When both levels are full, i.e., there is no space in L2-HCC to hold the contents of L1-HCC, all code in both levels is evicted.

FIFO@L1: Code is initially translated into L1-HCC and when it becomes full, fragments in L1-HCC are moved one at a time to L2-HCC in FIFO order. When L2-HCC is full, fragments are evicted from it in FIFO order. From the eviction point of view, the effect is the same as basic FIFO.

Segmented FIFO@L1: Code is initially translated into L1-HCC segments. When L1-HCC becomes full, its least recently filled segment is relocated to L2-HCC. When L2-HCC is full, its least recently added segment is evicted. From the eviction point of view, the overall effect is the same of the Segmented FIFO policy.

FIFO / Segmented FIFO: This hybrid policy combines single-fragment relocation with segmented eviction. Fragments are translated into L1-HCC and moved to L2-HCC one at a time in FIFO order. However, L2-HCC is divided in segments, which are evicted in FIFO order.

3.4 Retranslation-aware Resizing

When the HCC is too small to fully capture the translated code working set, some fragments are repeatedly evicted and retranslated, leading to excessive DBT overhead. To avoid this problem, we use a *retranslation-aware resizing heuristic*. On an overflow, the heuristic is used to decide whether to increase L2-HCC capacity (i.e., "adding" more main memory to it) or to evict code.

To make this decision, the translator monitors code that has been previously seen using the *Translation History Table* (THT). The THT records the application address (PC) of every fragment. If a PC is not found in the THT, it is recorded and the fragment is classified as "first-time"; otherwise, the fragment is classified as "retranslated". L2-HCC expansion is chosen when L2-HCC (or the segment to evict) contains more retranslated fragments than first-time fragments. The fragments that caused the expansion are marked to force eviction the next time. This strategy reduces the likelihood of keeping unneeded fragments in the HCC.

Parameter	Configuration
Fetch queue	4 entries
Branch predictor	not taken, 2 cycle mispred.penalty
Fetch/decode width	1 instr./cycle
Issue	1 instr./cycle, in order
Functional units	1 IALU, 1 IMULT, 1 FPALU, 1 FPMULT
RUU capacity	4 entries
Issue/commit width	1 instr./cycle
Load/store queue	4 entries
Memory page size	4 Kbytes
TLBs	64 entries, 2-way, 10 cycle miss
L1 D-cache	8 Kbytes, 4-way, LRU, 1 cycle
L1 I-cache	4 Kbytes, 4-way, LRU, 1 cycle
Scratchpad memory	4 Kbytes, 1 cycle
Bus width	4 bytes
Main memory latency	10 cycles first chunk, 4 cycles rest
Flash page size	512 Kbytes
Flash latency	3000 cycles first chunk, 10 cycles rest

Table 1: SimpleScalar Configuration

4. EXPERIMENTAL EVALUATION

In this section we evaluate our techniques, choose those that work best, and compare the HCC to a CC that uses only SPM or only main memory.

4.1 Methodology

We implemented our techniques in the Strata DBT [14], retargeted to run on SimpleScalar/PISA [2]. We extended the out-of-order simulator with ROM, SPM and Flash, which can have different sizes and latencies. The simulator reports the number of cycles spent executing code fetched from each kind of memory.

The simulator was configured as shown in Table 1. The resources are modeled after the ARM926EJ-S processor, used in smart phones, digital cameras, etc. PISA is an instruction set similar to MIPS, but with a 64-bit encoding to facilitate experimentation (e.g., instructions have a 16-bit annotation field and 8-bit register fields) [2]. To account for the long instructions, we doubled the size of the SPM and I-cache for the simulations but refer to the *effective size* (e.g., a 4KB SPM/I-cache is simulated with 8KB). We run programs from MiBench [9], a benchmark suite representative of embedded applications, with their large input data sets.[1]

Strata was configured to create *Dynamic Basic Blocks* (DBB), i.e., to stop fragment formation whenever a CTI is found. This configuration minimizes code duplication and speculative translation of code that may never be executed. To ensure a small translated code footprint, we enabled a set of techniques described in [3]: shadow link register trampolines, out-of-line shared IBTC lookup with shared target register copies, and prologue elimination.

4.2 Eviction Policies

To understand which eviction policy is most appropriate for the HCC, we measured their impact on program performance. Our evaluation is done with an HCC that has two levels: a 4K L1-HCC (SPM) and a L2-HCC (main memory) with an initial size of 16K. Capacity is added in 2K increments to L2-HCC using our resizing heuristic. We evaluate DBT performance with an HCC in the absence of complex code transformations. We compare FLUSH, Segmented FIFO with 2K segments (2K-Segs) and FIFO. The slowdown for the benchmarks relative to native execution with these policies is shown in Figure 3. Our result charts show slowdown bars split into three sections: the bottom section is time spent executing code from main memory (main memory time), the middle section is time spent executing code from SPM (SPM time), and the top section is the time spent executing Strata (translation time). We report these times as a slowdown relative to native execution to facilitate comparisons. *Native execution* means running a binary fully loaded into main memory (with a 4K I-cache), without the use of DBT.

HCC resizing prevents the programs from thrashing, so translation time is generally small and does not vary much across policies, with a few exceptions (*lame*, *pgp*, *typeset*). For instance, *typeset* has slowdowns of 1.17x (FLUSH), 1.19x (2K-Segs) and 1.14x (FIFO). Its translation times are: 0.22x (FLUSH), 0.15x (2K-Segs) and 0.24x (FIFO).

For some programs, SPM usage is insignificant (e.g., *adpcm*, *crc*, *rijndael*, *sha*). In programs with significant SPM usage, an important trend arises: more frequent use of SPM helps performance. For instance, *basicmath* has slowdowns of 1.35x (FLUSH), 1.04x (2K-Segs) and 1.29x (FIFO). Its translation times are 0.17x (FLUSH, 2K-Segs) and 0.16x (FIFO) and its SPM times are 0.07x (FLUSH), 0.11x (2K-Segs) and 0.06x (FIFO). *tiffdither* has speedups of 1.25x with FLUSH and 1.10x with 2K-Segs, but a 1.04x slowdown with FIFO. Its SPM times are 0.19x (FLUSH), 0.06x (2K-Segs) and 0.03x (FIFO), and its translation time is 0.03x.

Using the SPM reduces pressure on the I-cache, which helps to improve performance. For instance, *ghostscript* has speedups: 1.16x (FLUSH), 1.15x (2K-Segs), 1.14x (FIFO). Its translation and SPM times are respectively 0.06x and 0.02x for all policies. However, the native execution has 15% I-cache miss rate, which is reduced to 9-10%.

4.3 SPM-aware Policies

Our SPM-aware policies attempt to improve SPM usage by keeping the *most recently translated code* in SPM. Although they do not guarantee that the *most frequently executed* code is assigned to the SPM, they improve performance for several benchmarks. Figure 4 shows the slowdown relative to native execution for the benchmarks with the SPM-aware policies.

For some benchmarks where no evictions occur (*adpcm*, *bitcount*, *blowfish*, *crc*, *susan.smoothing*), the SPM is heavily used, but performance is practically unaffected since the I-cache miss rate is low. For other benchmarks without evictions (*dijkstra*, *qsort*, *rijndael*, *stringsearch*) there are significant improvements. For instance, the speedup of *qsort* is increased from 1.25x to 1.54x with all policies; *stringsearch*'s slowdowns of 1.11x (FLUSH), 1.12x (2K-Segs, FIFO) are improved to speedups of 1.32x (FLUSH@L1), 1.37x (2K-Segs@L1) and 1.43x (FIFO@L1, FIFO/2K-Segs).

The SPM-aware policies increase *ghostscript*'s speedups to 2.70x (FLUSH@L1), 1.22x (2K-Segs@L1), 2.78x (FIFO@L1) and 2.56x (FIFO/2K-Segs). For *basicmath*, its 1.35x slowdown with FLUSH is reduced to 1.03x with FLUSH@L1, and its 1.29x slowdown with FIFO turns into a 1.04x speedup with FIFO@L1. However, *basicmath*'s 1.04x slowdown with 2K-Segs is increased to 1.20x with 2K-Segs@L1.

ispell is not helped. Its 1.28x speedup with FLUSH turns into a 1.25x slowdown with FLUSH@L1. Its 1.23x slowdown with 2K-Segs is increased to 1.28x with 2K-Segs@L1. Its 1.15x speedup with FIFO is reduced to 1.10x with FIFO@L1.

[1] We could not compile `mad`, `rsynth`, and `sphinx`.

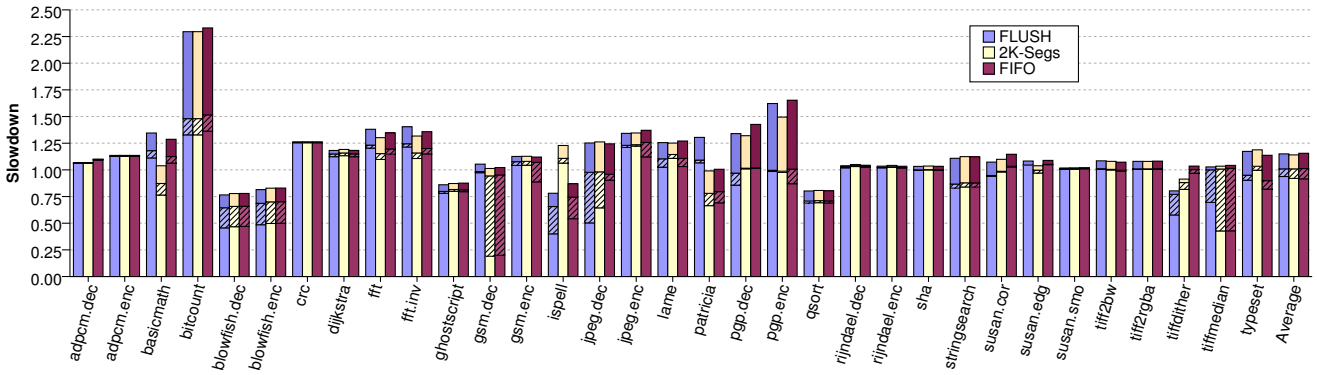

Figure 3: Slowdown relative to native execution with FLUSH, Segmented FIFO and FIFO eviction policies

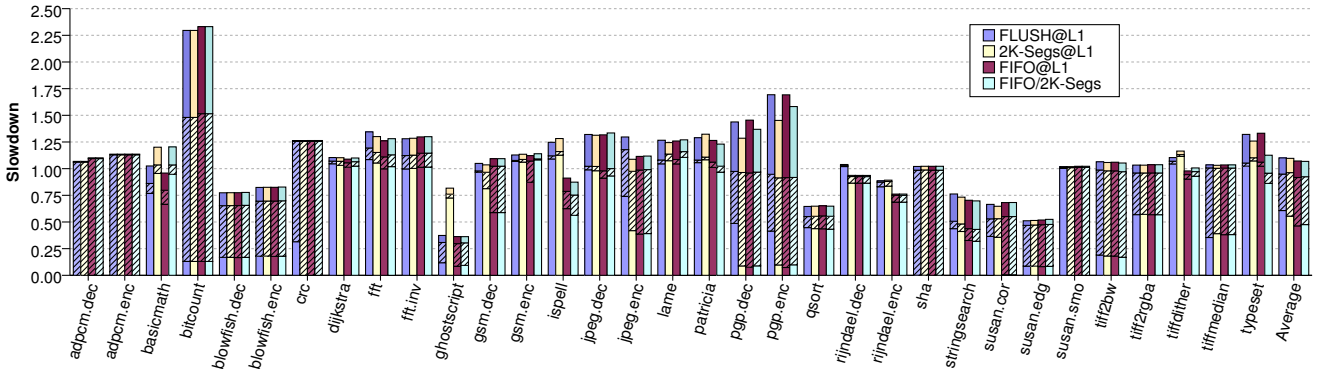

Figure 4: HCC slowdown relative to native execution with SPM-aware policies

However, with FIFO/2K-Segs, it has the same 1.15x speedup than with FIFO. In this case, the most recently translated code is not the most frequently used, and the SPM-unaware policies casually capture code with higher execution frequency in the SPM.

Interestingly, the average performance of all benchmarks with FIFO@L1 and FIFO/2K-Segs is the same: 1.07x slowdown. The average slowdown with FIFO is 1.16x. The average slowdowns with FLUSH and 2K-Segs are also improved by putting all new fragments in the SPM: from 1.15x (FLUSH) to 1.10x (FLUSH@L1) and from 1.14x (2K-Segs) to 1.10x (2K-Segs@L1).

In conclusion, the SPM-aware HCC management policies should be used instead of their SPM-unaware counterparts.

4.4 Comparison to Traditional Code Caches

We compared an HCC managed with an SPM-aware policy to a CC that uses only SPM and a CC that uses only main memory. The performance results for these three alternatives are shown in Figure 5.

SCC:FLUSH allocates the CC to the 4K SPM and handles overflows with FLUSH, without resizing. In most cases, the translated code does not fit in the small SPM, leading to unacceptable slowdowns[2]. For instance, *stringsearch* has a 15.09x slowdown, spent mostly in translation – SPM time is just 0.45x. With HCC:FLUSH@L1, it has a 1.32x speedup. Another example is *typeset*, which has a 43.93x slowdown with SCC:FLUSH, but its SPM time is just 0.61x. With HCC:FLUSH@L1, its slowdown is only 1.32x. When the translated code working set fits in SPM, SCC outperforms the techniques that execute code in main memory. For instance, *crc* has slowdowns of 1.09x with SCC:FLUSH and 1.26x with HCC:FLUSH@L1 and MCC:FLUSH.

MCC:FLUSH initially allocates a 16K CC to main memory and uses our resizing heuristic to adaptively increase its size (2K each time). FLUSH is used for evictions. In most cases, the SPM-aware techniques outperform MCC:FLUSH thanks to their taking better advantage of the SPM. For instance, *fft* with MCC:FLUSH has a 1.48x slowdown, but only a 1.34x with HCC:FLUSH@L1. For *ghostscript*, MCC:FLUSH does reasonably well: 1.15x speedup. With HCC:FLUSH@L1, it does much better: 2.70x speedup.

On average, the HCC outperforms both SCC and MCC. HCC:FLUSH@L1 has an average slowdown of 1.10x, while MCC:FLUSH slowdown is 1.23x. SCC:FLUSH has the worst average slowdown, 11.41x, because SCC is too small.

Only some benchmarks need the CC in main memory to grow. Table 2 shows the final CC size for benchmarks where L2-HCC grows at least by 4K. HCC size includes the 4K SPM. Although no growth limit is set when enabling the resizing heuristic, the amount of main memory needed for DBT is much less than for native execution. For instance, the final CC size for *ghostscript* is less than 8% the size of the binary's code segment.

From these results, we conclude that the HCC is a good choice for a SoC with heterogeneous memory resources since it can fully use those resources to minimize DBT overhead.

[2] *ispell* did not run to completion.

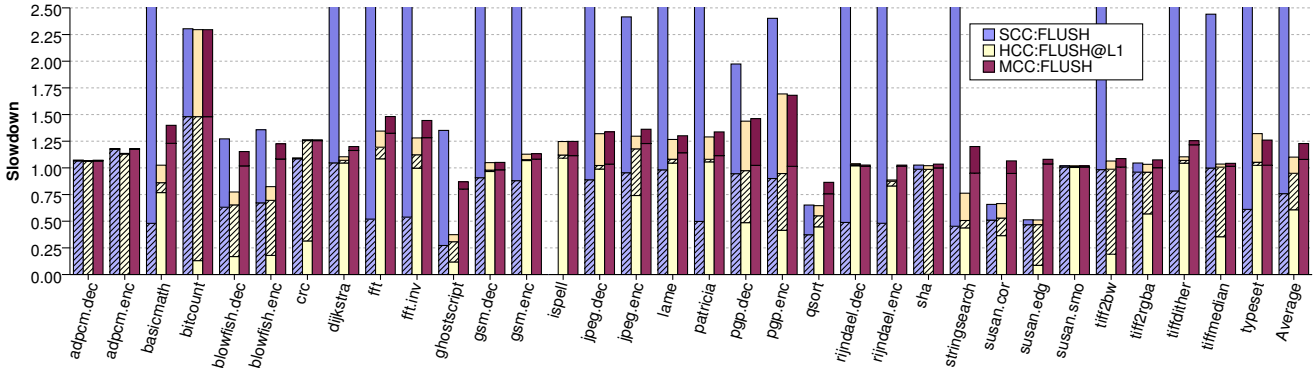

Figure 5: SCC, HCC and MCC slowdown relative to native execution with FLUSH

Benchmark	Binary	HCC:FLUSH@L1	MCC:FLUSH
ghostscript	888K	60K (6.76%)	64K (7.21%)
gsm.enc	78K	28K (35.90%)	28K (35.90%)
ispell	106K	28K (26.42%)	28K (26.42%)
lame	158K	70K (44.30%)	68K (43.04%)
patricia	58K	28K (48.28%)	26K (44.83%)
pgp.dec	228K	30K (13.16%)	30K (13.16%)
pgp.enc	228K	28K (12.28%)	26K (11.40%)
typeset	544K	80K (14.71%)	86K (15.81%)

Table 2: Final code cache size

5. RELATED WORK

CC management has been extensively studied in general-purpose DBT systems. [5] proposed using FLUSH on overflows or preemptively when a phase change is detected. [10] found that FIFO's miss rate is similar to LRU's with less management overhead and 50% better than the miss rate of FLUSH. [11] found that mid-grained evictions (Segmented FIFO) scale better than FLUSH and FIFO, and proposed a generational approach that stores short-lived and long-lived fragments in disctinct CCs to prevent premature evictions.

[7] explored expanding a bounded CC. [7] doubles the CC size when the ratio of retranslated to replaced fragments is above a threshold. To avoid exponential growth, our resizing approach expands the HCC only by a constant amount.

[13] developed software instruction caching for processors with SPM and no I-cache. [13] uses a static binary rewriter to form cache blocks. [4] showed that DBT can also treat the SPM as a software I-cache. [4] used victim compression and pinning to reduce retranslation cost when the binary is in external Flash. The CC in [13, 4] is only in SPM.

6. CONCLUSION

This paper presents the Heterogeneous Code Cache (HCC), which is a CC split among SPM and main memory. Several HCC design issues and management policies are evaluated. Our results show that an HCC outperforms a CC that uses only SPM or only main memory. The best performance is achieved when treating SPM and main memory distinctly.

Our techniques make DBT more feasible in embedded systems. HCC can be combined with compelling DBT uses, such as compressing infrequently used code and decompressing it on-demand [8]. Future HCC enhancements include promoting frequently used code from main memory to SPM.

7. ACKNOWLEDGEMENTS

This work was supported by NSF under grants CCF-0811295, CCF-0811352, CNS-0702236, CNS-0720483 and CNS-0551492.

8. REFERENCES

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] 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, Architecture, and Synthesis for Embedded Systems*, 2008.

[4] J. A. 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*, 2007.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Conference on Programming Language Design and Implementation*, 2000.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *International Conference on Hardware/Software Codesign*, 2002.

[7] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *International Symposium on Code Generation and Optimization*, 2005.

[8] S. Debray and W. Evans. Profile-guided code compression. In *Conference on Programming Language Design and Implementation*, 2002.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop on Workload Characterization*, 2001.

[10] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *Workshop on Interaction between Compilers and Computer Architectures*, 2002.

[11] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Trans. on Architecture and Code Optimization*, 3(3):263–294, 2006.

[12] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *International Symposium on Code Generation and Optimization*, 2007.

[13] J. E. Miller and A. Agarwal. Software-based instruction caching for embedded processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[14] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization*, 2003.