CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication and its Application to Instruction Caches

Marios Kleanthous and Yiannakis Sazeides
Department of Computer Science, University of Cyprus

Abstract

Cache-Content-Duplication (CCD) occurs when there is a miss for a block in a cache and the entire content of the missed block is already in the cache in a block with a different tag. Caches aware of content-duplication can have lower miss rates by allowing only blocks with unique content to enter a cache. This work examines the potential of CCD for instruction caches. We show that CCD is a frequent phenomenon and that an idealized duplication-detection mechanism for instruction caches has the potential to increase performance of an out-of-order processor, with a 2-way eight instruction per block 16KB instruction cache, often by more than 5% and up to 20%. This work also proposes CATCH, a hardware based mechanism for dynamically detecting CCD. Experimental results for an out-of-order processor show that a CATCH with a 2.32KB cost usually captures 60% or more of the CCD's idealized potential.

1 Introduction

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance [12]. Caches, consequently, have been central to numerous research studies. Several techniques have been proposed to improve various aspects of caches by reducing their miss rates, size, latency and energy. Most of these techniques attempt to exploit different types of properties of memory addresses and data.

This work identifies a new cache property that may influence cache performance: the Cache-Content-Duplication (CCD). This phenomenon occurs when there is a miss for a block in a cache and the content of the missed block resides already in the cache in another block with a different tag. For example, Fig. 1.a shows an instruction cache where each block is identified by its tag and Fig. 1.b shows an instruction cache which is aware of the block content. This example shows that two different blocks, with tags 103 and 115, have identical content. If block 115 is evicted and later

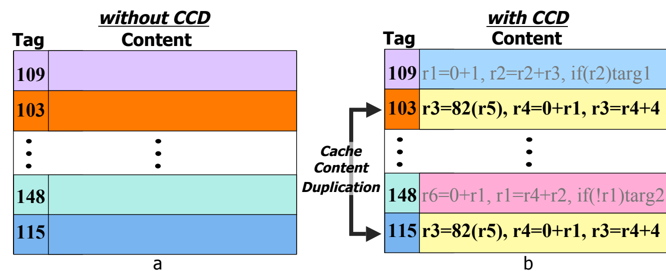

Figure 1. Cache Content Duplication

we have a miss on it, the content of 103 can be used without accessing a lower level of the memory hierarchy.

What mainly distinguishes CCD from previous work is that it exploits cache content redundancy at the granularity of cache blocks instead of considering the compression of patterns in the cache content or the elimination of redundant memory content irrespective of its cache placement. An example of CCD based optimization is the Unique-Content-Cache (UCC) that can lower the miss ratio by allowing only blocks with unique content to enter the cache.

As a first step toward understanding and exploiting CCD this work is focused on the content duplication in instruction caches. CCD in instruction caches exists because: (a) high level language programs often contain identical instruction sequences [9] in different segments of a program due to: copy-paste programming practices and reuse of standard libraries and loops in different parts of code, (b) conventions, such as for calls and returns, produce similar sequences, (c) compiler transformations, such as compiler inlining and macro expansion, lead to duplicated code sequences.

The main contributions of this work are, the phenomenon of Cache-Content-Duplication, Unique-Content-Cache a new cache type that exploits CCD, CATCH a hardware mechanism that can dynamically detect CCD, an investigation of its performance for a UCC cache, and several optimizations to increase the CATCH's cost-efficiency.

In Section 2, previous work is discussed. Section 3, discusses CCD detection. Section 4 presents the simulation environment. In Section 5, CATCH is introduced and different optimizations to improve its cost-efficiency are discussed. Section 6 evaluates the performance of CATCH. Section 7 provides conclusions and directions for future work.

978-3-9810801-3-1/DATE08 © 2008 EDAA

2 Related work

The redundancy of the memory and cache content has been the subject of several previous papers. The main objectives were to increase the effective memory/cache capacity and to achieve higher bandwidth during information transferring between different levels of the memory hierarchy.

A scheme for main memory on-line compression was first proposed by Douglis [7]. The compression cache proposed allows both software and hardware based compression using different compression algorithms. Lefurgy et al. [10] explored the idea of keeping compressed code in instruction memories of embedded processors. Based on static analysis, common sequences of instructions are assigned unique codes. These codes are stored in instruction memory and are expanded to their original form after being read. The high redundancy of a subset of values in data caches was identified in [13]. A frequent-value cache was proposed to hold the frequent values in compressed form.

Very relevant to our work is [11] that introduces the notion of address correlation: two different addresses are correlated when at the same time they contain the same value. Nonetheless, our work is distinct because: (a) we consider the duplication of instruction blocks whereas in [11] the focus is individual data values, and (b) we propose a hardware mechanism for detecting and exploiting CCD.

CCD work can also borrow many concepts from research in the area of code compaction [2, 5, 6]. The main cost of code compaction is runtime overhead due to the extra instructions executed to steer the control flow to/from unique copies of repeated sequences and to transform dissimilar sequences to similar. This overhead, however, can be offset by a possible reduction in instruction cache misses. Both code compaction and CCD aim to detect and exploit redundancy in code. However, compaction methods are compiler based whereas the method considered here is dynamic hardware based. Code compaction typically reduces code size and cache misses, at the expense of increasing the dynamic instruction count. CCD, on the other hand, aims to reduce execution time using extra hardware, instead of extra instructions, to minimize/eliminate the penalty for misses on duplicated sequences. Furthermore, CCD may be the only way to exploit duplication in legacy code where there is no opportunity for re-optimization.

Previous work considered either the compression and compaction of arbitrary length sequences of data or instructions, or the compression at the granularity of individual instructions or values. Approaching redundancy in terms of cache blocks enables new memory hierarchy optimizations but requires mechanisms for detecting block level redundancy. The organization and performance of these novel memory optimizations and of the duplication detection mechanisms are the main issues examined in this work.

3 Cache-Content-Duplication

3.1 What is the cache content considered for duplication

One important parameter than can influence the frequency of CCD is the cache content that is considered for duplication. By CCD's definition this is an entire cache block. It is expected that CCD will occur more likely between blocks that have fewer instructions (smaller cache blocks) and are basic block aligned. Smaller sequences are more likely to match, and sequences aligned at basic block boundaries are more likely to be identical. To clarify, consider two basic blocks that are identical. In an instruction cache, the duplication may not be detected because the blocks that contain them are not aligned, and/or because the instruction cache block may contain other instructions, in addition to the duplicated basic block, that are different.

A way to increase the frequency of CCD for instruction caches is to consider the duplication between *valid* instructions sent down the pipeline on an instruction cache access, instead of *entire* instruction cache blocks. In [4] a *valid* block is defined as the static consecutive instruction sequence starting from the current PC until: (a) the first predicted-taken conditional branch, or (b) the first unconditional branch, or (c) a number of instructions equal to fetch bandwidth are read from the cache. A *valid block* is identified by the starting PC and a *bit mask* that can be produced at each cycle using the BTB and the direction predictor [4]. This *mask* indicates the location of the first taken branch in a sequential instruction sequence. Valid blocks have properties that make them more amenable to CCD. They are usually basic block aligned and their size roughly corresponds to a basic block.

3.2 When to learn the cache content

To detect duplication between valid blocks, it is necessary to know the content of blocks already in the cache. This way, when a block misses the cache, it can be detected whether or not its content is duplicate with a block already in the cache.

The content of a valid block can be learned by remembering its content when it is inserted in the cache. This is referred to as *learn-on-miss* learn policy. However, the *learn-on-miss* is not sufficient to learn all the relevant content in a cache because, on a cache miss, an entire cache block is filled in the cache and the missed valid block covers only a subsequence of the entire block. Another method is to learn on a cache miss the missed valid block content and heuristically learn other valid blocks in the missed block. We refer to this policy as *learn-all-on-miss*. An example heuristic is to build an additional valid block using the remaining instructions in the block after the missed valid block,

benchmark	Skip (millions)	Simulate (millions)
gcc95	0	177
go95	0	133
perl95	0	40
vortex00	100	100
mesa00	350	100
basicmath	0	100

Table 1. Simulated benchmarks

fetch/issue/commit width	4/4/4
Issue Queue/LSQ/ROB	64/32/64
Stages	14
L1 instruction cache	16KB 2-way 32B/block, 1 cycle
L1 data cache	32KB 2-way 64B/block, 2 cycles
L2 unified cache	2MB 8-way 128B/block, 20 cycles
Main memory latency	200 cycles
Cond. branch predictor	8KB combining predictor
BTB/RAS/Ind. predictor	1024/32/512 entries

Table 2. Processor Configuration

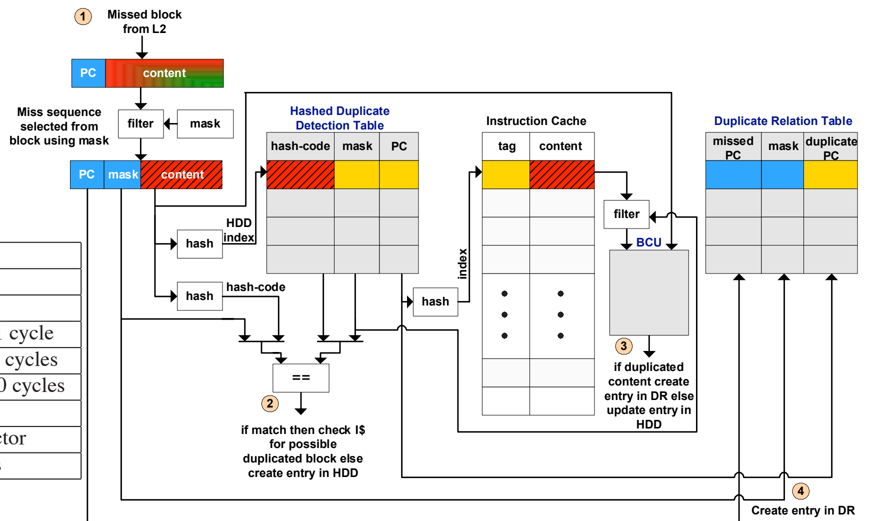

Figure 2. Cache miss, DR miss and HDD hit

and treat the next conditional branch to be encountered as taken. Henceforth, unless indicated otherwise, the *learn-all-on-miss* policy is used for learning valid blocks.

3.3 Which sequences are duplicated

Two valid blocks are considered duplicates if each instruction in a block is bitwise identical in the exact order with its corresponding instruction in the other block. Nonetheless, the duplication criteria can be relaxed for direct (conditional or unconditional) control transfer instructions by allowing differences in their immediate offset or target fields in order to increase duplication frequency. This technique is known in the area of code compaction as target abstraction [2]. Section 5 discusses, in detail, how using a table that stores small target differences between otherwise identical sequences, facilitates more duplication while maintaining correctness. We note that other abstraction transformations, such as register and constant abstraction [2], can be applied to increase the duplication frequency. However, in this work we focus mainly on duplication detection. For the experimental results, unless stated otherwise, it is assumed that CCD employs target abstraction.

4 Experimental Framework

The experiments in this paper were performed using benchmarks from the SPEC95, SPEC2000 and MiBench1.0 suites with train or reference inputs. All benchmarks are compiled with gcc 2.7 with -O3 optimizations for the PISA ISA [3]. We report results for the following six benchmarks: *gcc95*, *go95*, *perl95*, *vortex00*, *mesa00* and *basicmath*. These benchmarks were selected because they have the largest miss rates for the cache configurations we considered and, therefore, more likely to benefit from the techniques proposed in this work. Table 1 shows the dynamic instructions skipped and simulated for these benchmarks.

We assess and compare the performance impact of the different techniques using a modified sim-out-of-order [3] simulator with the configuration listed in Table 2. The instruction size is 4 bytes and therefore a 32B cache block contains 8 instructions.

5 Unique-Content-Cache and CATCH

CCD can be used to reduce misses by allowing only blocks with unique content to enter the cache. We refer to such cache as the Unique-Content-Cache (UCC). A UCC can reduce capacity misses because it allows a smaller number of blocks to enter a cache. A UCC needs to be also duplicate-aware to detect misses to duplicated blocks.

A hardware implementation of a UCC requires a mechanism for detecting and remembering duplicate relations. Specifically, this mechanism given the starting PC and mask of a valid block that caused a cache miss, should return whether there is a duplicate in the cache and also the starting PC of the duplicated block. This section presents a method for dynamically detecting CCD. We will refer to this mechanism as CATCH. Recall that valid blocks are identified with their starting PC and a bit mask provided by the branch predictor (see Section 3). The microarchitecture of a cache with a CATCH is shown in Fig. 2. It includes the Duplicate-Relation cache (DR), the Hashed-Duplicate-Detection cache (HDD) and the Block Compare Unit (BCU). The functionality of the different components and their updating policies is the subject of this section.

5.1 Duplicate-Relation cache

The Duplicate-Relation cache (DR) contains duplication-relations detected by the CATCH. Each DR entry contains a starting PC and a mask of a missed valid block and the starting PC of its duplicate valid

block. The use of a PC and a mask is sufficient to prevent false duplicate relations. Once a duplicated relation is established it is assumed to be always correct (in the case of self-modifying code or page remapping the DR may need to be flushed to ensure correctness).

DR can be either virtually or physically tagged. A virtually tagged DR can be used in combination with a virtually tagged cache or by keeping virtual tags in the HDD. A virtually tagged DR in combination with a physically tagged cache may add an extra penalty for translating the tag using the Instruction Translation Look-aside buffer (ITLB) each time we access the cache for a secondary hit (secondary hit is a cache hit to a duplicate sequence using CATCH). On the other hand, using a physically tagged DR will eliminate this overhead but the DR may need to be flushed each time we have a page remapping. However, page remapping is a very rare phenomenon. For our experiments, we used a physically tagged DR with a physically tagged cache.

On a cache miss, the DR is accessed with the starting PC and mask of a missed block. When there is a DR hit and the duplicate PC hits in the cache, a secondary-hit occurs. The content of the duplicate-block will be read and no miss will be requested from a lower level of the memory hierarchy.

5.2 Hashed-Duplicate-Detection cache

An entry in the DR is created when a block with a cache miss is fetched from a lower level cache and is found to be a duplicate with a block already in the cache. Therefore, the detection of CCD requires a mechanism that given the content of a block, it provides a starting PC and a mask for a candidate duplicate-block currently in the cache.

This functionality is provided by the Hashed-Duplicate-Detection cache (HDD). Each entry in the HDD contains a hash-code, which encodes the content of a block, and the corresponding starting PC and mask of the valid block. The use of a hash-code reduces the cost and complexity of detecting duplication but may lead to unnecessary tests for duplication. Nevertheless, we found that a simple folding of the block content to 16 bits provides very accurate encodings (often 99.9% accurate).

The HDD is indexed using a hash of the content of a missed block after it is fetched from the lower-level of memory hierarchy. When a missed block's hash-code and the hash-code in a valid HDD entry match, we may have content duplication. In this case, the cache is accessed using the starting PC found in the HDD to determine whether the two valid blocks are indeed duplicates. The test for duplication is performed by the Block Compare Unit (BCU). If the BCU indicates that the blocks are duplicates then an entry is created in the DR. Fig. 2 illustrates the sequence of steps in the case of a cache miss that has a duplicate in the cache but not an entry in DR.

5.3 The Block Compare Unit

When two blocks are signaled by the HDD as possible duplicates, their contents are compared using the Block-Compare Unit (BCU) to detect whether there is indeed duplication. The compare function used in the BCU can be a simple bitwise comparison of the instructions in the two blocks. BCU optimizations that use more advanced compare functions to tolerate differences in the targets of branches are considered and discussed in Section 5.5.

5.4 Allocating and Updating an HDD and a DR entry

An HDD entry is allocated when a block is both a cache and an HDD miss. There are two different scenarios for allocating an HDD entry:

1. Cache miss, DR miss, HDD miss: The block is fetched from a lower level of memory hierarchy, its content's hash-code is calculated and then HDD is accessed with this hash-code. On a miss a new HDD entry is created.
2. Cache miss, DR hit, Cache miss, HDD miss: Same as above except that there is a DR hit that leads to a cache access and misses because the duplicate block was evicted. If we miss in the HDD then an entry is allocated and points to the fetched block in the cache.

There are also two cases for updating an HDD entry and allocating or updating a DR entry:

1. Cache miss, DR miss, HDD hit: The block is fetched from a lower level in the memory hierarchy and its hash-code is calculated. The HDD is accessed with the hash-code. If we hit in the HDD then the cache is accessed with the duplicate-PC. The two block contents are compared and if they match, a DR entry is created with the missed starting PC and mask, and the duplicate-PC pointed by the HDD. When the content of the missed block and the one pointed by the HDD do not match in the BCU, we have a case of a false hash-code match. This was found to occur very rarely for hash-codes of 16 bits. When this happens, the HDD entry will be updated to point to the missed block.
2. Cache miss, DR hit, Cache miss, HDD hit: Same as above except: (a) there is a DR hit that leads to a cache access that does not hit and (b) if the HDD points to a truly duplicate block then the DR entry will be updated with the duplicate starting PC pointed by the HDD.

5.5 Performance Optimizations

The first optimization is to tolerate simple differences between blocks by using a more advanced compare function in the BCU. The *keep_offset_in_dr* optimization aims to increase content-duplication by masking out, from the compare process in BCU, the offsets and targets of conditional

and unconditional direct branches, and keeping in the DR the offsets and targets of each duplicate block. This aims to convert blocks that contain exactly the same computation into duplicates. This is effectively a hardware implementation of the target abstraction discussed in Section 3.3.

The second performance optimization is to filter the updates in the HDD and DR tables by avoiding the insertion of entries that are unlikely to have a significant pay-off. A successful implementation of updating filtering can be conducive in reducing the table sizes and/or improve their performance. CATCH employs a simple but effective filtering scheme proposed by Behar et al. [1]. The filtering is accomplished by allowing a table to be updated every *n* attempts. This policy works because it can prevent rare events from entering the tables, whereas persistently occurring events will eventually make it into the table. For an extended discussion on how this method works we refer the interested reader to [1]. Based on simulation results, not shown here, it was found that the best strategy was to filter only the updates of the HDD and the filter value should be four, i.e. updating the HDD every fourth attempt. Although the DR is not filtered directly, by updating the HDD less frequently, the updates to the DR are indirectly reduced.

The significance of the *keep_offset_in_dr* and the filtering optimizations is investigated in Section 6.

5.6 Cost Reduction Optimizations

This Section describes several optimizations to reduce the amount of state required by the HDD and DR caches. A 2-way, 32B block, 16KB instruction cache with four instructions maximum valid block length is assumed.

A DR entry represents logically two full tag-indices. For the PISA instruction set architecture used in this work [3], the first tag-index contains 30 bits (28 bits for the starting PC and 2 bits for the number of instructions in the valid block) and the second tag-index contains 28 bits for the starting PC of the duplicate valid block. After some cursory analysis it was observed that usually the 9 leading bits of the starting PC of the missed and duplicate valid block are the same. This reduces the cost of a DR entry by 9 bits if only the entries that satisfy this criterion are inserted into the table. When the *keep_offset_in_dr* optimization is employed, the DR should keep a maximum of four direct targets. To reduce the number of bits required by the offsets and direct targets, extra insertion criterion can be used. Specifically, duplicated relations are inserted when all of the following are true: valid blocks have at most one control flow instruction and the upper 10 bits of direct targets must be the same with valid block's starting address. Note that for the ISA used in this study target offsets for conditional branches are 16 bits and direct targets are 26 bits. With these criterion in place, the extra cost of the keep_offset_in_dr optimization is 16 bits for each DR entry, for an offset or a target.

Therefore, for the DR the per-entry cost with cost optimization is (28+19+2-$\log_2$(number of sets in DR))+16 bits.

An HDD entry contains a hash-code and the PC and mask of the duplicate block. A 16-bit hash-code causes false-hash-matches very rarely. Also, the criterion used in DR (the 9 most significant bits of the two tag-indices must be the same) can be used here also. That means we only keep the 21 least significant bits in the HDD and combine them with the 9 most significant bits of the missed valid block to create the index-tag and access the cache.

Therefore, for the HDD the per-entry cost with cost optimization is 16 + 19 + 2 - $\log_2$(number of sets of the HDD) bits. In Section 6, we compare the performance with and without the cost optimizations.

6. Performance evaluation of CATCH

In this Section we evaluate the performance of the CATCH mechanism to detect CCD. The analysis is focused on the performance of a 16KB instruction cache that is 2-way 32B per block, with a single cycle secondary hit latency (secondary hit is a cache hit to a duplicate sequence) in addition to the L1 hit latency.

6.1 CATCH performance

Fig. 3 shows the performance of CATCH compared to a limit study (CCD Limit) with oracle CCD detection. "CATCH Limit" corresponds to a CATCH implementation with unbounded DR and HDD tables. Analysis, not shown here, suggests that a 4-way 256 entries DR and a 2-way 128 entries HDD represent a good performing CATCH configuration. This configuration (4.56KB perf_optim) can provide an average IPC improvement of 4% which corresponds to 61% of the performance potential of a UCC with oracle CCD detection (Fig. 3). Note that this CATCH configuration has 4.56KB state cost and employs all the performance optimizations but none of the cost optimizations. To reduce the state cost of CATCH we applied the various cost optimizations discussed in Section 5.6. This lead to a reduction in CATCH cost to 2.32KB but with negligible performance degradation (on the average 1% compared to the CCD Limit).

Fig. 3 also quantifies the significance of the performance optimizations, discussed earlier, on the 2.32KB CATCH. The results show that without filtering the performance is degraded by 10%, without learning an additional valid block on a miss the degradation is 19%, and without the target abstraction the performance benefits are reduced by 34%.

We would like to note that we have also experimented with the 2.32KB version of CATCH with all SPEC and MiBench benchmarks that had minimal cache misses and established that CATCH did not degrade their performance.

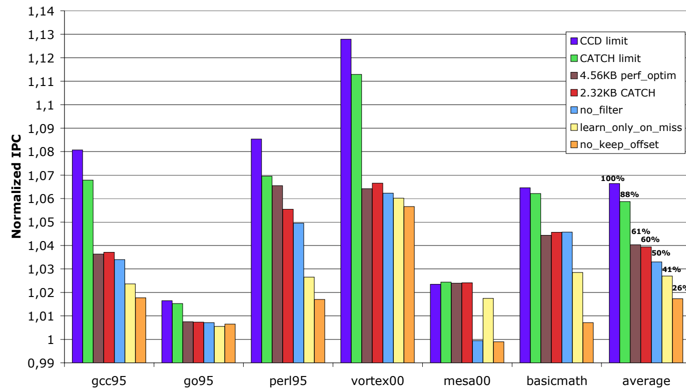

Figure 3. Effects of applying different policies on CATCH performance

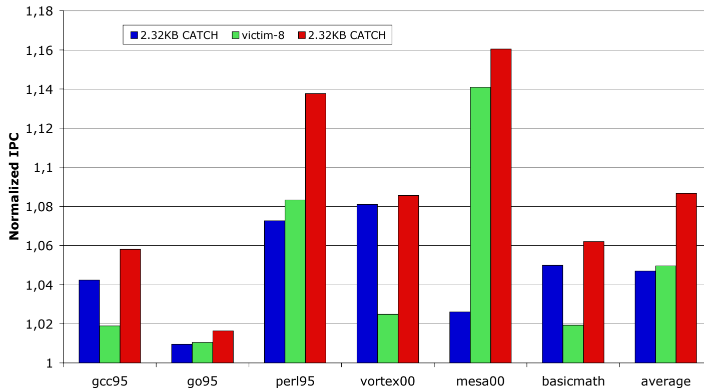

Figure 4. CATCH and 8 entry Victim Cache

6.2 CATCH vs Victim cache

An alternative mechanism to reduce cache misses is the victim cache [8]. Fig. 4 shows the performance improvement of a regular cache using an 8-entry victim cache, the CATCH with 2.32KB cost, and a combination of the two. In the combination, the victim cache is accessed first and the CATCH is used only in case of a victim cache miss.

The data show that in some benchmarks, gcc95, vortex00 and basicmath, CATCH is better than the victim cache whereas the victim cache is superior for the others. However, the most important observation is that the performance gain from the combination of CATCH and victim cache is additive. This indicates that CATCH captures misses that are not conflict misses in the same set but across sets.

7 Conclusions and Future Work

This work introduces the notion of CCD and proposes CATCH, a hardware mechanism for dynamically detecting CCD. The paper evaluates the performance of CATCH for the UCC cache architecture that exploits CCD. Experimental results for a processor with a 2-way 8-instruction per block 16KB instruction cache show that a CATCH with 2.32KB cost usually captures 60% or more of the CCD idealized potential. Experimental results comparing CATCH with victim cache show that CATCH can capture misses that are not due to conflict misses in the same set. Thus the performance gain of the two mechanisms is additive.

There are several directions for future work. One is to investigate other methods to tolerate block differences and lead to higher CCD frequency. A mechanism for zero cycle secondary hit latency may be also useful to design and evaluate. Another important direction of research is to consider CCD for data caches and other levels in the memory hierarchy and for multicores. CCD may also be considered in combination with static code compaction to investigate the synergistic potential of the two approaches. CCD must also be evaluated with other types of workloads, including database applications. Finally, timing and power complexity issues of CATCH need to be investigated in more detail.

References

[1] M. Behar, A. Mendelson, and A. Kolodny. Trace Cache Sampling Filter. In *PACT*, September 2005.

[2] A. Beszedes, R. Ferenc, T. Gyimuthy, A. Dolenc, and K. Karsisto. Survey of Code-Size Reduction Methods. *ACMCS*, 35(3), September 2003.

[3] D. Burger and T. Austin. The SimpleScalar tool set: Version 2.0. Technical Report 1342, University of Wisconsin-Madison, June 1997.

[4] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *ISCA*, June 1995.

[5] K. D. Cooper and N. McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *PLDI*, May 1999.

[6] S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler Techniques for Code Compaction. *ACM TOPLAS*, 22(2), March 2000.

[7] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *USENIX*, January 1993.

[8] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ISCA*, June 1990.

[9] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS*, July 2001.

[10] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. In *MICRO*, December 1997.

[11] R. Sendag, P. Chuang, and D. Lilja. Address Correlation: Exceeding the Limits of Locality. *Computer Architecture Letters*, 2(1), May 2003.

[12] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *Computer Architecture News*, 23(1), March 1995.

[13] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-Centric Data Cache Design. In *ASPLOS*, November 2000.