CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication in Instruction Caches

MARIOS KLEANTHOUS and YIANNAKIS SAZEIDES, University of Cyprus

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 penalty by fetching, on a miss to a duplicate block, directly from the cache instead of accessing lower in the memory hierarchy, and 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 16KB, 8-way, 8 instructions per block instruction cache, often by more than 10% and up to 36%.

This work also proposes CATCH, a hardware mechanism for dynamically detecting CCD for instruction caches. Experimental results for an out-of-order processor show that a duplication-detection mechanism with a 1.38KB cost captures on average 58% of the CCD's idealized potential.

Categories and Subject Descriptors: B.3.2 [Design Styles]: Cache Memories

General Terms: Design, Performance

Additional Key Words and Phrases: Cache optimizations, cache compression, cache content duplication

ACM Reference Format:

Kleanthous, M. and Sazeides, Y. 2011. CATCH: A mechanism for dynamically detecting cache-content-duplication in instruction caches. *ACM Trans. Architect. Code Optim.* 8, 3, Article 11 (October 2011), 27 pages.

DOI = 10.1145/2019608.2019610 <http://doi.acm.org/10.1145/2019608.2019610>

1. INTRODUCTION

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance [Wulf and McKee 1995]. 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, such as locality [Denning 1970],

We would like to thank the University of Cyprus, Intel and HiPEAC for the research grants that support this work. This work also falls under the Cyprus Research Promotion Foundations Framework Programme for Research, Technological Development and Innovation 2009-10 (DESMI 2009-10), co-funded by the Republic of Cyprus and the European Regional Development Fund, and specifically under Grant ΤΠΕ/ΠΑΗΡΟ/0609(BIE)/09.

This article is an extension of Kleanthous and Sazeides [2008]. Additional material provided: analysis for a larger set of benchmarks; offline analysis of code redundancy in the applications; more thorough analysis of the CCD mechanisms and their optimizations; Comparison of CATCH with prefetching and 16+2KB cache; power analysis of CATCH.

Authors' address: M. Kleanthous and Y. Sazeides, University of Cyprus, Department of Computer Science, 75 Kallipoleos Street, P.O. Box. 20537, CY-1678 Nicosia, Cyprus; email: mklean@cs.ucy.ac.cy.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 1544-3566/2011/10-ART11 \$10.00

DOI 10.1145/2019608.2019610 <http://doi.acm.org/10.1145/2019608.2019610>

Fig. 1. Cache Content Duplication (a) Without CCD, (b) With CCD.

predictability [Baer and Chen 1991; Lipasti et al. 1996], and redundancy [Kjelso et al. 1996; Cooper and McIntosh 1999].

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. Therefore, CCD is a manifestation of redundancy in the cache content. For example, Figure 1(a) shows an instruction cache where each block is identified by its tag and Figure 1(b) shows an instruction cache which is aware of the block content. This example shows that two different blocks, with tags 123 and 141, have identical content. If block 141 is evicted and later we have a miss on it, the content of 123 can be used without accessing a lower level of the memory hierarchy.

CCD may occur at any level of memory hierarchy for both data and instructions. However, 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 high-level language programs often contain identical instruction sequences [Komondoor and Horwitz 2001] 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; and (c) compiler transformations, such as compiler inlining and macro expansion, lead to duplicated code sequences.

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. This enables new memory hierarchy optimizations such as (a) the Duplicate-Aware-Cache that can reduce the miss penalty by identifying misses on blocks with duplicated content and fetching the duplicate block already in the cache instead of fetching it from lower in the memory hierarchy, and (b) the Unique-Content-Cache that can lower the miss ratio by allowing only blocks with unique content to enter the cache.

The main contributions of this article are the following.

- (1) We present the phenomenon of Cache-Content-Duplication (CCD).
- (2) We introduce two new cache types, the Duplicate-Aware-Cache (DAC) and the Unique-Content-Cache (UCC), that can exploit the CCD phenomenon and a performance evaluation of their potential for instruction caches.
- (3) We introduce CATCH, a hardware mechanism that can dynamically detect CCD, and an investigation of its performance for DAC and UCC instruction caches. The experimental analysis for an out-of-order processor with an 16KB, 8-way, 8 instructions per block (32Byte block) instruction cache show that a CATCH with a 1.38KB cost captures on average 58% of the CCD's idealized potential.

The rest of the article is organized as follows. In Section 2, previous work is discussed. Section 3 discusses issues related to CCD detection. Section 4 presents the simulation environment. In Section 5 an offline analysis is presented of the frequency and coverage of duplicated instruction sequences. Section 6 examines the limits of CCD. Section 7 considers two possible applications of CCD and investigates their performance potential. In Section 8, CATCH is introduced and different optimizations to improve its cost-efficiency are discussed. Section 9 evaluates the performance of CATCH. Section 10 provides conclusions and directions for future work.

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 online compression was first proposed by Douglass [1993]. The cache proposed allows both software and hardware based compression using different algorithms. Benini et al. [1999] proposed a dictionary based compression technique for instruction caches. This scheme does not require any processor modification since the instructions are decompressed outside the core. Kjelso et al. [1996] proposed a hardware implementation of the X-Match dictionary compression algorithm for main memory data. Lefurgy et al. [1997] 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. Lefurgy et al. [2000] studied the concept of keeping compressed code in main memory and “software decompressing” on a cache miss. More specifically, frequently used instructions, in the original code, are replaced by pointers to an entry in a small instruction dictionary. The high redundancy of a subset of values in data caches was identified in Zhang et al. [2000]. A frequent-value cache was proposed to hold the frequent values in compressed form.

Alameldeen and Wood [2004] keep information compressed, for both instructions and data, only in level-2 cache and can dynamically choose to keep data in uncompressed form when the overhead of compression may cause degradation in performance. Hallnor and Reinhardt [2004] proposed a scheme that can map multiple compressed blocks into a single physical cache block using an indirect-index cache. This scheme maintains compressed data both in main memory and on-chip and enables the data to travel through the bus in compressed form. Therefore, this approach offers both extra space on main memory and cache, and higher transfer rates from main memory to cache.

Postiff and Mudge [1999] proposed smart-register-files aiming to solve the aliasing problem of more than one registers referring to the same datum, either address or data. Hines et al. [2005] proposed the use of an instruction register file to hold frequently executed instructions. An integrated compiler/hardware mechanism exploits this to reduce area and power.

The key difference of our work from previous effort is that we consider redundancy at the granularity of cache blocks and detect it dynamically using a hardware mechanism instead of comparing arbitrary patterns with profiling aid.

Very relevant to our work is Sendag et al. [2003], which introduces the notion of address correlation: two different addresses are correlated when at the same time they contain the same value. Address correlation can improve performance if on a cache miss the correlated address is found in the cache. The authors investigated the limits of oracle address correlation, and found it to be significant, but did not propose a mechanism for detecting it. The concept of CCD is similar with address correlation

because it also exploits the duplication of content at different addresses. Nonetheless, our work is distinct because: (a) we consider the duplication of instruction blocks whereas in Sendag et al. [2003] the focus is individual data values, and (b) we propose a hardware mechanism for detecting and exploiting CCD.

Biswas et al. [2009] investigate the phenomenon of data similarity in multiexecution programs. They observed that when multiple instances of the same application are running on a multicore sharing the same L2-cache, their data are usually very similar. Biswas's work exploits the CCD phenomenon as proposed in Kleanthous and Sazeides [2008] but only for a specific scenario in which multiple instances of the same application share an L2-cache.

Previous code compaction work [Beszedes et al. 2003; Cooper and McIntosh 1999; Debray et al. 2000] and our work share similarities but also differences. 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. The static approach can detect repetition at a coarser scale, for example functions with multiple basic blocks. CCD detection window is limited to at most a cache block at a time. 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 reoptimization. Finally, it is possible, but outside the scope of this work, to consider schemes that combine static code compaction with dynamic CCD since they are in some respects complementary.

Overall, 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.

This article extends Kleanthous and Sazeides [2008] by providing an offline analysis for code redundancy, a more comprehensive analysis of the CCD mechanisms and optimizations, and results with a larger set of benchmarks.

3. CACHE-CONTENT-DUPLICATION

CCD occurs when there is a miss in a cache and the entire content of the missed block is already in the cache in another block with a different tag. This section discusses key issues that can influence the CCD frequency in instruction caches.

3.1. What Is the Cache Content Considered for Duplication

One important parameter that can influence the frequency of CCD is the cache content that is considered for duplication. For an instruction cache, a block always contains a block size number of instructions starting from the block address. 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 but reside in two different instruction blocks at different positions. In an instruction cache, the duplication may not be detected because the blocks that contain them are not aligned. Also the instruction cache blocks may contain other instructions, in addition to the duplicated basic blocks, which 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

Fig. 2. Valid block masked out from a cache block.

access, instead of entire instruction cache blocks. In Conte et al. [1995] 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 [Conte et al. 1995]. This *mask* indicates the location of the first taken branch in a sequential instruction sequence. A valid block represents, therefore, the predicted instructions that are sent down the pipeline after a cache access, and we will refer to it as a valid block. Figure 2 shows how a valid block is built. 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.

Sections 5 and 6 consider the CCD limits for both cache blocks and valid blocks, while Section 9 focuses on CCD for valid blocks in regular instruction caches.

3.2. When to Learn the Cache Content

To detect duplication between cache 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 a duplicate with a block already in the cache.

To learn a whole cache block it is sufficient to remember its content when it is inserted in the cache. This is referred to as *learn-on-miss* policy. However, the *learn-on-miss* is not sufficient to learn all the relevant content in the case of valid blocks 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. One way to increase the frequency of CCD for valid blocks is to learn both missed valid blocks and valid blocks that are cache hits. This is referred to as *learn on miss and hit* policy. Furthermore, to learn the valid blocks in a cache block may require multiple block accesses, with some CCD potential lost in the intervening time. 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, and treat the next conditional branch to be encountered as taken. Henceforth, unless indicated otherwise, the *learn-all-on-miss* policy is used. The importance of the *learn* strategy on CCD for valid blocks is investigated in Section 9.

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

Table I. Simulated Benchmarks

SPECINT 2000	Skip (10 ⁶)	Execute (10 ⁶)	SPECFP 2000	Skip (10 ⁶)	Execute (10 ⁶)	TPC-H	Skip (10 ⁶)	Execute (10 ⁶)
gzip	19400	500	wupwise	7950	500	Q1F	200	500
vpr	25600	500	swim	1150	500	Q2F	50	178
gcc	8400	500	mgrid	150	500	Q3F	200	500
mcf	13400	500	applu	2100	500	Q4F	50	500
crafty	950	500	mesa	450	500	Q5F	1000	500
parser	1000	500	galgel	4450	500	Q6F	500	500
eon	26400	500	art	3150	500	Q7a	400	500
perlbnk	13800	500	equake	19300	500	Q8a	500	500
gap	20900	500	facerec	36600	500	Q9a	1000	500
vortex	18600	500	ammp	5200	500	Q10F	900	500
bzip2	43000	500	lucas	2650	500	Q11a	500	500
twolf	7200	500	fma3d	10300	500	Q12a	400	500
			sixtrack	8200	500	Q13a	400	500
			apsi	1650	500	Q14a	300	500
						Q15F	500	500
						Q16F	300	500
						Q17F	50	133

Table II. Processor Configuration

fetch/issue/commit width	4/4/4
INT Issue Queue/FP Issue Queue/ROB	64/64/256
Pipeline Stages	10
L1 instruction cache	16KB 8-way 32B/block, 3 cycles
L1 data cache	16KB 8-way 32B/block, 3 cycles
L2 unified cache	2MB 8-way 32B/block, 20 cycles
Main memory latency	200 cycles

fields in order to increase duplication frequency. This technique is known in the area of code compaction as target abstraction [Beszedes et al. 2003]. Section 8 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 [Beszedes et al. 2003], 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 article are performed using all benchmarks from the SPEC2000 and TPC-H suites with reference inputs. The binaries are compiled for the ALPHA ISA [Compaq 1998] using the Compaq C compiler and -O2 optimization level. Compiler optimizations that increase duplication, such as loop unrolling, are disabled. Table I shows the dynamic instructions skipped and executed for each benchmark. These regions were selected using a simpoint like tool [Sherwood et al. 2002].

We assess and compare the performance impact of the different techniques with both functional and timing simulations using a modified SMTSIM simulator [Tullsen 1996]. The functional simulator is used for quick characterization of the design space in Sections 5, 6 and 7. The performance simulator is used in all other cases to model an out-of-order processor with the configuration listed in Table II. The performance metrics used in this study are Accesses per 1K instructions, instructions per cycle (IPC) and the CCD rate of each benchmark. The CCD rate refers to the fraction of misses that are for duplicate-blocks.

Fig. 3. Execution coverage of unique blocks for the a) SPECINT 2000, b) SPECFP 2000 and c) TPC-H benchmarks.

5. CODE REDUNDANCY CHARACTERIZATION

In this section we characterize the frequency of duplicated sequences at the granularity of 32-byte blocks (8 instructions) and valid sequences (maximum of 4 instructions) during dynamic execution. The 32-byte blocks are 32 byte aligned, while the valid sequences are built dynamically, as explained in Section 3.1.

Figure 3 shows the number of unique blocks, at the granularity of 32-byte blocks, needed to cover a certain amount of dynamic execution when identified by their block

tag, TAG, and by their unique content, CONTENT. First, the results show that very few benchmarks have redundancy at the granularity of a whole block. This is expected due to misalignment of code sequences as discussed in Section 3.1.

Another interesting observation is that many benchmarks need less than 500 of 32-byte blocks for their execution. A 16KB cache contains 512 of 32-byte blocks. That means detecting and eliminating redundancy for these benchmarks will not make much difference for a cache equal or bigger than 16KB unless of course there are many set conflicts between those blocks.

Figure 4 shows the number of unique blocks, at the granularity of valid sequences (maximum of 4 instructions), needed to cover a certain amount of dynamic execution. Compared to Figure 3, the results indicate that there is much more potential when identifying the redundancy at the granularity of valid sequences than cache blocks. Figure 5 shows the execution breakdown in percentages for few selected benchmarks that have significant amount of cache pressure. The unique valid sequences identified by their CONTENT are normalized to the total unique valid sequences identified by the sequence TAG. The results clearly show that by removing redundancy the number of unique valid sequences required for execution is reduced by up to 20% for many benchmarks. For example, the results for Q9a show that we need only 80% of the total unique valid sequences to cover 100% of the execution when we identify them by their CONTENT. Also for the same benchmark, we observe that 50% of the unique valid sequences identified by their TAG are needed to cover 95% of its execution. On the other hand, if we identify the sequences by their CONTENT then only 38% of the total unique valid sequences are required indicating that a significant amount of pressure will be alleviated.

6. LIMITS OF CACHE-CONTENT-DUPLICATION

The previous section investigated the redundancy during a program's execution. This section will establish the Cache-Content-Duplication (CCD) limits for an instruction cache by investigating how often two duplicated blocks, or valid sequences, happen to be in the cache at the same time.

The results are obtained assuming oracle CCD detection: complete knowledge of all blocks in a cache and ability to detect any possible duplication of a missed block with a block already in the cache. The oracle CCD detection uses the default policies, presented in Section 3, for detecting and learning CCD. The CCD is determined by checking on each miss if the missed block content is identical with a block already in the cache. This is referred to as a *secondary-hit*.

Figure 6 shows the breakdown of Misses and Secondary hits per 1K instructions with an 8-way, 32B (8 instructions) block, instruction cache for various cache sizes when considering duplication of entire blocks. The graph also shows the CCD rate, secondary-hits/total misses, using a label above each bar. The results are split into three graphs, (a) SPECINT 2000, (b) SPECFP 2000 and (c) TPC-H benchmarks.

The results show that CCD for entire instruction cache blocks is a rare phenomenon, usually 0-1% of the misses are for duplicated blocks. As it was discussed in Section 3, one of the main reason for the low duplication rates is that instructions are placed in the instruction cache based on their block address and duplicated sequences may not start at the same relative address within different cache blocks. Furthermore, an instruction cache block may contain instructions that never get executed, for example instructions before a branch target or after an always taken control flow instruction, and this may lead to effectively identical blocks to appear dissimilar.

Figure 7 presents the CCD for valid blocks of a subset of 15 benchmarks with the highest Misses per 1K instructions. The data show that the CCD rates for valid blocks is often above 15% and therefore more prominent than for entire cache blocks (Figure 6).

Fig. 4. Execution coverage of unique valid blocks for the a) SPECINT 2000, b) SPECFP 2000 and c) TPC-H benchmarks.

This increase supports the two claims of Section 3 that: (a) two valid blocks that are shorter than cache block size are more likely to match, and (b) valid blocks starting at a different position in two cache blocks can be detected as duplicates.

The general trend in Figure 7 is that with increasing cache size the number of duplicate valid misses decrease because larger caches have fewer misses, but the CCD rates increase. This suggests that the relative importance of duplicates misses increases.

Fig. 5. Execution coverage breakdown of unique valid blocks in percentages for a selected subset of benchmarks.

This occurs because with a larger cache, it is more likely for a missed valid block to already have a duplicate in the cache.

Furthermore, the data show that SPECINT 2000 and TPC-H benchmarks have higher CCD rates. This is mainly due to the higher misses per 1K of these benchmarks that offer more opportunity for duplication detection.

We have also examined the effects of varying associativity on CCD (results not shown). The frequency and the trends of CCD appear almost the same as with an 8-way cache. The small sensitivity of CCD to associativity may indicate that CCD is not due to conflict misses that can be removed using a victim cache [Jouppi 1990]. Section 9 compares the performance of an instruction cache with a victim cache against an instruction cache that combines a victim cache and a CCD mechanism and reveals that victim caching and CCD are orthogonal.

Henceforth, we consider CCD only for valid blocks because it has higher potential.

7. CCD APPLICATIONS: DAC AND UCC

This section describes two possible memory hierarchy enhancements based on CCD that can reduce cache latency and cache miss rate.

Cache latency can be reduced through the detection of misses to blocks with a duplicate in the cache and by fetching the block from the cache instead of reading it from lower in the memory hierarchy. We refer to such cache as the Duplicate-Aware-Cache (DAC). Therefore, a DAC can reduce the miss penalty of a duplicated miss down to a cache hit. Because the latency of a duplicated miss is likely small, henceforth, we refer to it as a secondary hit (primary hits are those that hit directly in the cache). All accesses that are neither primary nor secondary hits are misses that need to be serviced from a lower level cache. A DAC cache, when compared to an otherwise identical regular cache, is expected to have as many primary hits as the hits of the regular cache, but have some of the regular cache misses converted to secondary hits. Therefore, in the presence of CCD a DAC can only improve performance. Another benefit of DAC is a reduction in the traffic to lower levels of memory hierarchy because the missed block can be read directly from the L1 Cache and inserted to the correct set. Note that for DAC, in the case of CCD for valid blocks there is no traffic reduction because the entire block is always fetched on a miss. Overall, the amount of improvement from DAC mainly depends on the number of the regular cache misses it converts to secondary hits.

Fig. 6. Misses and Secondary hits per 1K instructions breakdown and CCD rates for an 8, 16, 32 and 64KB, 8-way, 8 instructions per block, instruction cache, for entire blocks. a) SPECINT 2000, b) SPECFP 2000, c) TPC-H.

CCD can also be used to reduce misses by detecting misses to duplicated blocks and allowing only blocks with unique content to enter a cache. We refer to such cache as the Unique-Content-Cache (*UCC*). A UCC, when compared to an otherwise identical regular cache of same size, is expected to convert some hits of the regular cache to secondary hits and misses, but also have a large number of misses converted to primary and secondary hits. The performance of a UCC will be superior over a conventional cache if the savings due to the conversion of misses to primary and secondary hits outweigh the penalty of having some primary hits turned into secondary hits or misses.

Next we investigate experimentally the performance limits of DAC and UCC.

Fig. 7. Misses and Secondary hits per 1K instructions breakdown and CCD rates for an 8, 16, 32 and 64KB, 8-way, 8 instructions per block, instruction cache, for valid blocks.

Fig. 8. Misses and Secondary hits per 1K instructions breakdown and CCD rates for a UCC 8-way, 8 instructions per block, instruction cache, for valid blocks.

7.1. Limits of the Cache-Content-Duplication

An indication of the performance potential of a DAC, over a regular cache, is given by the fraction of misses that have a duplicate in the cache. These results are shown in Figure 7 for an instruction cache for valid blocks.

To establish the potential of a UCC cache over a regular cache we performed an oracle study with the same assumptions as in Section 6. The UCC cache is modeled as a regular cache except when there is a miss that has a duplicate block in the cache, that is, a secondary hit. When this occurs, the duplicate content is used without fetching the missed block from the lower levels of memory hierarchy and without inserting it in the cache.

Figure 8 shows the breakdown of accesses per 1K instructions for a UCC-instruction cache for valid blocks. For comparison purposes, the graphs show the secondary hits that were initial hits, for the respective regular cache, labeled as “Primary converted to Secondary.” Also included in the graphs are numeric values for the CCD-rate that correspond to baseline misses converted to secondary hits (without considering the “Primary converted to Secondary.”)

A comparison of Figures 7 and 8 reveals that for most benchmarks and cache configurations the CCD rates for UCC caches are higher than their corresponding DAC caches. For example, for a 16KB cache in Figure 8 crafty has 17% CCD rate where its corresponding rate for DAC is 10% (Figure 7). The reason for this increase, is that UCC

Fig. 9. Maximum, minimum and average of the normalized IPC performance of all benchmarks for DAC and UCC for valid blocks. Results are shown for 15, 20, 25 and 30 cycles L2 latencies and 0, 1 and 2 cycles secondary hit latencies.

avoids the insertion of duplicate content in the UCC cache and eliminates capacity and may be conflict misses. This reduces the total number of misses by more than the duplicate misses of DAC.

However, the data also show that a UCC cache can have fewer primary hits than a regular cache. For example, for a 16KB cache in Figure 8, crafty has 3.7 access per 1K instructions that were converted from primary hits to secondary hits. Therefore, only 3.5 out of the 7.2 secondary hits per 1K instructions correspond to cache misses converted to secondary hits. This result suggests that a UCC cache, unlike DAC, in some cases may not improve the performance because a decrease in primary hits can offset the benefits of CCD. Consequently, to compare the performance potential of DAC and UCC a study for an out-of-order processor is performed.

7.2. Performance Potential of CCD

Figure 9 shows the performance potential of DAC and UCC in terms of normalized IPC for 16KB DAC and UCC instruction caches over a 16KB regular instruction cache. Note that in these experiments we assume oracle CCD detection under the same assumptions as in Sections 6 and 7.1.

Results are presented for secondary hit latencies of 0, 1, and 2 cycles (denoted in the graph as DAC-0, DAC-1 and DAC-2 or UCC-0, UCC-1 and UCC-2 respectively) and for various L2 cache latencies (15, 20, 25 and 30 cycles). The various secondary hit latencies are aimed to reveal how critical is to detect quickly duplication after a miss. The different L2 cache latencies are useful to examine the importance of CCD with increasing latency to lower levels of memory hierarchy. All other processor parameters are as in Table II. The middle point in each line shows the average IPC improvement of all benchmarks while the top and bottom point show the maximum and minimum IPC improvement for each configuration.

The data show both DAC and UCC to have performance potential up to 10% and 36% respectively. Analysis not shown here indicates that the TPC-H benchmarks with the highest CCD rates, see Figures 7 and 8, are also the ones with the largest potential while most of the SPEC2000 benchmarks do not benefit from CCD because they have very few misses.

The potential improves with increasing L2 cache latency for both DAC and UCC. The DAC performance is rather insensitive to secondary hit latency, however, for UCC the effects of secondary hit latency can degrade performance. For example with UCC-2 latency and 15 cycles L2 latency, gap suffers a performance degradation of 2% compared to the baseline. The secondary hit latency effects are reduced as the L2 latency increases. As shown for the same benchmark, for 30 cycles L2 latency, the performance degradation is reduced to 1%. For UCC-0 and UCC-1 there is no performance degradation.

The lower UCC performance for 2 cycles secondary hit latency suggests that the performance gains due to the miss reduction of UCC are outweighed by the penalty for having some primary hits converted to secondary hits. Another observation is that, although the limits of CCD rates for DAC and UCC are very similar, as shown in Figures 7 and 8, the results in Figure 9 show that UCC is much better in many configurations. This occurs because in the DAC limit study we assumed no latency for fetching a block from a lower level in the memory hierarchy and thus in the case of two consecutive accesses to the same missed block (for different valid blocks), the first would be a secondary hit and the second would be a primary hit. But in a realistic scenario, with fetch latency, it is possible to have a secondary hit and the next access to the same block to cause a miss because the block is not yet fetched from the L2 cache.

A zero cycle secondary hit latency is possible, but may require more pervasive changes in the processor front-end. This is discussed more extensively in Section 8.8. The single cycle secondary hit latency can be achieved by accessing the CCD mechanism and cache in parallel. By the end of the tag array access, assuming 1 cycle, the CCD mechanism will provide an alternative tag-index to access the cache again in case of a miss. Finally, for a serial access of cache and the CCD mechanism after a cache miss, a 2 cycle secondary hit latency is required. The first cycle is spend on a tag array access to discover the cache miss, and the second cycle to access the CCD mechanism and provide an alternative tag-index.

Overall, the CCD performance potential results are encouraging and thus in the next section we propose and evaluate CATCH, a hardware mechanism that can dynamically detect CCD for DAC and UCC caches.

8. CATCH: A METHOD FOR DYNAMICALLY DETECTING CCD

A hardware implementation of a DAC or a UCC instruction cache 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 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 in instruction caches 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 Figure 10. It includes the Hashed-Duplicate-Detection table (HDD), the Block Compare Unit (BCU) and the Duplicate-Relation table (DR). The functionality of the different components and their updating policies are the subject of this section.

8.1. Hashed-Duplicate-Detection Table

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.

The Hashed-Duplicate-Detection table (HDD) provides this functionality. 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

Fig. 10. The CATCH flow for a Cache miss, DR miss, and HDD hit.

duplication. Nevertheless, we found that a simple folding of the valid block content to 16 bits provides very accurate encoding (often 99.9% accurate).

The HDD is indexed using a hash of the content of a missed block after it is fetched from a lower level of the memory hierarchy. For better performance this hash can be different from the one used for producing the hash-code for a block.

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 Block Compare Unit (BCU) performs the test for duplication. If the BCU indicates that the blocks are duplicates then an entry is created in the DR. Figure 10 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. This process is similar for DAC and UCC.

8.2. 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 8.6.

8.3. Duplicate-Relation Table

The Duplicate-Relation table (DR) contains relations between duplicated blocks detected by CATCH. 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 using the HDD table and BCU unit.

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

Fig. 11. The CATCH flow for a Cache miss, DR hit, Cache hit.

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. The design for virtually tagged caches is outside the scope of this paper.

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. In the case of a DAC, the content of the missed valid block will be read and a request in a lower level cache for the entire missed block will be initiated in parallel. For a UCC, only the content of the duplicate-block will be read and no miss will be requested from a lower level of the memory hierarchy. Figure 11 illustrates the sequence of steps in the case of a cache miss that has an entry in the DR and a duplicate in the cache.

8.4. Allocating and Updating an HDD and a DR Entry

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

- (1) Cache miss, DR miss, HDD miss. A valid block is a miss in the cache and no entry in the DR matches its starting PC and mask. 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 (1) 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. A block is a miss in the cache and the DR. 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. Also the HDD entry is updated to point to the fetched block in the cache (the implications of not-updating the HDD in this case are discussed in Sections 8.6). 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 (1) 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.

8.5. The Use of CATCH in DAC and UCC

A DAC and a UCC can use the CATCH, as described, previously, to detect a miss for a duplicated block and read the missed block directly from the cache, as long as the block is in the cache. However, there is a key difference in how CATCH is used for a DAC and a UCC. In a DAC, when accessing the HDD, the block will be first inserted in the cache and then it will be checked for duplication because there is a risk to evict its duplicate from the cache and this will result in an invalidation of the HDD entry. On the other hand, for a UCC the block is first checked for duplication and only if the HDD cannot detect any duplication will the block be inserted in the cache.

8.6. Performance Optimizations

This section describes two types of performance optimizations for CATCH. The first optimization is to tolerate simple differences between blocks by using a more advanced compare function in the BCU. The *keep_offset* 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. Two possible caveats of this optimization is the extra cost per DR entry, and that secondary cache reads may need to combine information from the cache and the DR which may make fetching more complicated. The first is considered for the total size of the mechanism calculated in Section 8.7 while the second can be accommodated in the valid block masking logic (Figure 2) and is not considered further in this paper.

Other examples of BCU optimizations, not evaluated in this work, is to augment the compare function to rearrange source operands of commutative operations and reorder data independent instructions in a block to facilitate content duplication [Cooper and

McIntosh 1999]. These and other transformations to be discovered may help uncover even more duplication but this is to be considered in future work.

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. [2005]. 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 Behar et al. [2005]. 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* and the filtering optimizations is investigated in Section 9.

8.7. Cost Reduction Optimizations

This section describes several optimizations to reduce the amount of state required by the HDD and DR caches. A 16KB, 8-way, 8 instructions per block instruction cache with four instructions maximum valid block length is assumed.

Before computing the cost for a DR entry, recall that a DR entry represents logically two full tag-indices. For the Alpha instruction set architecture [Compaq 1998] used in this work, the first tag-index contains 30 bits (28 bits for the address of the first instruction of the missed sequence and 2 bits for the mask, which is the number of valid instructions in the sequence), and the second tag-index contains 28 bits for the address of the first instruction of the duplicate sequence. The second tag-index does not require a mask because it must be the same with its duplicate sequence for a duplicate relation to exist. The nonoptimized cost of a DR entry is therefore $(2 * 28 + 2 - \log_2(\text{number of sets in DR}))$ which is the sum of the two addresses and the length of sequence minus the index of DR.

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* 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 the following are true: (a) valid blocks have at most one control flow instruction and (b) 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 21 bits. With these criteria in place, the extra cost of the *keep_offset* optimization is 11 bits for each DR entry, for one offset or one target.

Therefore, for the DR the per-entry cost with cost optimizations is $(28 + 19 + 2 - \log_2(\text{number of sets in DR})) + 11$ bits.

An HDD entry contains a hash-code, the PC and the mask of the duplicate block. For the limit study we assumed a 32-bit hash-code, but further analysis indicates that a 16-bit hash-code causes false-hash-matches very rarely. So, in Section 9 we consider the performance with a 16-bit hash-code. Furthermore, we can use the hash-code used for tag-matching the valid blocks in order to index the HDD. This will reduce the HDD

entry by $\log_2(\text{number of sets of the HDD})$ bits. 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 $16 + 19 + 2 - \log_2(\text{number of sets in HDD})$ bits.

Finally the replacement policy for HDD and DR is assumed to be tree based pseudo LRU [Malamy et al. 1994] that requires $N-1$ bits per set, where N is the associativity of the structure. In Section 9, we compare the performance with and without the cost optimizations.

8.8. Pipelining Issues

To incorporate a CATCH in a pipeline successfully, we have to consider timing issues. Some of these issues are discussed below.

The latency overhead for a duplicated hit is the total time required to access the DR with the missed block address plus the latency for a cache access to read the duplicated block. The DR latency component can be hidden if we access in parallel the cache and the DR so that as soon as a miss is detected we access the cache with the duplicated-PC.

A method that can provide zero duplicated hit latency is to maintain two program counters (PC) in a processor. The *sequence-PC* is used for control flow sequencing, and the *fetch-PC* is used for accessing the cache for fetching instructions. When a program starts the two PCs contain the same address. As long as a program has no duplication the two PCs will point to the same address. In the case of CCD, the sequence-PC should sequence as if there was no duplication but the fetch-PC should be made to point to the duplicate location. This can be accomplished by integrating the function of the DR in the BTB table. The BTB is normally used to store and predict targets of taken branches. To accommodate their new functionality, BTB entries should be extended to contain a duplicated-PC field in addition to the target of a branch. When this field is not valid the fetch-PC takes the address of the sequence-PC. However, when a predicted taken branch has a valid duplicated-PC the sequence-PC will take the normal branch target from the BTB but the fetch-PC will be updated with the duplicated-PC. A duplicated-PC is inserted in the BTB when the instruction sequence at the target of a taken branch is detected to be duplicated with another sequence starting at the duplicated-PC. The detection can be accomplished using an HDD as discussed earlier in Section 8.

This qualitative discussion suggests that a zero cycle detection mechanism may be feasible but its implementation details need to be considered further in future work.

One other important concern is the CATCH update latency. After a cache miss the newly fetched valid block must be checked for duplication. This means that the HDD must be accessed and if a possible duplicate exists, it must be compared using BCU and update the HDD and DR accordingly. A possible implementation of the mechanism can use a temporary buffer to keep the missed valid block and proceed with the updating process during the next cache miss. The L2 or main memory miss latency will provide enough time to compare the blocks and update the DR and HDD. In this work we assume optimistically that the updating of HDD or DR can be done in a single cycle in parallel with the testing and updating process.

9. PERFORMANCE EVALUATION OF CATCH

In this section we evaluate the performance of the CATCH mechanism to detect CCD. First, we determine the performance of CATCH with unbounded DR and HDD tables. Then, we introduce various constraints to the size, associativity and information per entry, to establish how much of the oracle performance (Section 7) it can be

Fig. 12. Performance potential captured by oracle detection (limit) and CATCH for DAC and UCC (16KB instruction cache, 20 cycles L2 cache latency).

captured by a more feasible to implement hardware configuration of CATCH. The analysis is focused on the performance of a 16KB instruction cache, for valid blocks, that is 8-way 8 instructions per block, with a single cycle secondary hit latency in addition to the L1 hit latency and 20 cycles L2 cache latency. In order to make the figures more readable, we show results only for the 15 benchmarks with the higher Misses per 1K that offer more opportunity for performance improvement. We also include the average for the remaining 28 benchmarks (average-other) and the average of all 43 benchmarks (average-all). For the 28 benchmarks not shown we verify that the worst case degradation is 0.1% for the twolf benchmark.

9.1. CATCH Performance for DAC and UCC Caches

Figure 12 shows the normalized performance potential captured by DAC and UCC with oracle CCD detection (same as in Figure 9) and the normalized performance potential captured by DAC and UCC using the CATCH.

Overall, from the data is evident that CATCH can capture 84% of the potential limit of DAC and more than 91% of the potential limit of UCC on average. This suggests that the CATCH design is very efficient.

The lower CATCH potential is due to the optimism in the oracle study that allowed serving a miss from the duplicate block the first time a relation is detected. In a real scheme this is not possible since the relation needs first to be detected and inserted in the DR and only afterwards may be useful for a secondary hit. Nonetheless, the data show that UCC suffers a smaller degradation because UCC can still benefit from the first detection of a relation by not inserting the duplicate block in the cache.

One interesting observation from Figure 12 is that for a benchmark, Q16F, the UCC performance of CATCH is slightly higher than the oracle UCC results. This happens due to the “failure” of a real HDD to maintain the hashed content of all valid blocks in the cache. This results in duplicated content to be inserted in the cache. The data show this duplication to be beneficial to performance.

The cause of this behavior is that with an oracle UCC no content duplication is possible and a given block content may be mapped to sets where the block is repeatedly evicted due to conflicts. On the other hand, a UCC with CATCH may “allow” multiple concurrent mappings of a block-content in the cache. If one of these mappings is to a set with fewer conflict misses, then all the duplicates pointing to that block may have

Fig. 13. Effects of applying different policies on CATCH performance.

better performance compared to the oracle UCC. This phenomenon is analyzed later where its effects are more prominent when the size of the CATCH is reduced further.

For the remainder of Section 9 we focus on optimizing the performance of the UCC instruction cache due to its higher performance potential compared to DAC.

9.2. CATCH Performance

The previous section presented the performance of CATCH with unbounded DR and HDD tables. This section will discuss the performance implications when using a CATCH with small size, set-associative DR and HDD tables. Some experiments will also help uncover the significance of the various performance and cost optimizations.

Figure 13 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 128 entries DR and an 8-way 128 entries HDD represent a good performing CATCH configuration. This configuration (3.05KB CATCH) can provide an average IPC improvement of 7.5% for the 15 selected benchmarks and 3% over all 43 benchmarks, which corresponds to 50% of the performance potential of a UCC with oracle CCD detection (Figure 13). Note that this CATCH configuration has 3.05KB 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 8.7. This led to a reduction in CATCH cost to 1.38KB, with negligible performance degradation for few benchmarks, less than 1%, but with an improvement of 0.4% overall and 1.2% over the 15 selected benchmarks as shown in Figure 13. The 1.38KB CATCH can provide 8.7% improvement for the 15 selected benchmarks and 3.4% over all 43 benchmarks, which corresponds to 58% of the performance of UCC with oracle CCD detection.

Figure 13 also quantifies the significance of the performance optimizations, discussed earlier in Section 8.6, on the 1.38KB CATCH. The results, for the 15 selected benchmarks, show that without filtering (1.38KB CATCH no filter) the performance degrades by 1% on average, without learning an additional valid block on a miss (1.38KB CATCH learn on miss) the degradation is 2% on average and without the target abstraction (1.38KB CATCH no keep offset) the performance benefits are reduced by 1.5%.

An interesting observation is that, sometimes, the smaller 1.38KB CATCH provides better performance than the 3.05KB CATCH. For example, benchmark Q16F shows

Fig. 14. CATCH with various cache associativities.

an increase of 4% with smaller CATCH. Analyzing the benchmark further reveals a reduction in secondary hits due to blocks with duplication that enter the cache. This seemingly undesirable behavior can benefit sometimes performance. In particular, some of these blocks also contain nonduplicated valid sequences that are referenced in the near future that become now cache hits. This suggests that an adaptive filtering mechanism may benefit performance further by exploiting this phenomenon more effectively. This represents a possible direction for future work.

9.3. Effects of Associativity

The functional simulations showed that the CCD rates are insensitive to associativity and the miss rates were slightly affected. Figure 14 shows the normalized IPC of each baseline cache to the same cache with the addition of CATCH, for example for the 2-way bar the results are for a 2-way cache using CATCH and the baseline is a 2-way cache without CATCH. The results indicate that on average the performance improvement of CATCH is not affected by the associativity. It is interesting that the average for the low miss rate benchmarks, average-other, shows a performance degradation as the associativity increases. This happens because higher associativity means less cache misses and lower potential for the CATCH to improve.

Nevertheless there are some cases, like benchmark Q16F, where increasing the associativity improves CATCH performance. We analyzed this behavior and found two possible scenarios that the associativity affects the performance of CATCH.

First, due to the CATCH algorithm, on every secondary hit access the LRU of the duplicated block is updated. In the case of a very hot duplicated block, its LRU will be updated constantly and effectively remains in the MRU position. This can reduce the associativity and the performance potential of CATCH.

Second, we observed that some duplicated blocks have large distance between their accesses. In the case of high miss rate benchmarks, a shallow LRU stack will maintain the duplicated block enough time in the cache to be accessed by the DR on a secondary hit. Deeper LRU will favor the block with secondary hits to remain longer in the cache without affecting significantly the performance of the cache.

The above observations suggest that a balance must be kept between the duplicated blocks allowed in the cache and the associativity of the cache. From our experiments it appears that an 8-way associative cache can solve this problem most of the times and

Fig. 15. CATCH with various cache sizes.

filtering techniques, like the one presented in Section 8.6, can almost eliminate the problem.

9.4. Effects of Cache Size

Section 6 showed that the CCD rates increase as the cache size increases because there is more opportunity to find duplicated blocks when you have more blocks to compare. Figure 15 shows the normalized IPC of each baseline cache to the same cache with the addition of CATCH, for example for the 8KB bar the results are for an 8KB cache using CATCH and the baseline is an 8KB cache without CATCH. The results indicate that on average the performance improvement of CATCH is reduced as the cache size increases. This is due to the low miss rate for the majority of the benchmarks that can be completely eliminated with a cache bigger than 16KB. However, for the few high miss rate benchmarks, we can see that the CATCH performs better with a 16KB cache compared to an 8KB but for most of the time a 32KB eliminates all misses and thus any room for improvement.

9.5. CATCH vs Victim Cache

An alternative mechanism to reduce cache misses is the victim cache [Jouppi 1990]. A victim cache aims to reduce cache misses, due to conflicts in a set, by keeping a fully associative structure and maintaining victim blocks there until they are evicted or needed again from the cache. Figure 16 shows the performance improvement of a regular cache using an 8-entry victim cache, the CATCH with 1.38KB 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 for three benchmarks, Q16F, Q8a, and Q2F, victim cache is better than the CATCH, whereas the CATCH 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 only in the same set but also across sets.

9.6. Effects of Prefetching

Prefetching is another technique to reduce cache misses and improve performance. We have investigated the performance improvement of a simple next-line prefetcher with and without the CATCH. The next-line prefetcher is applied at all cache levels

Fig. 16. CATCH and 8 entry Victim Cache.

Fig. 17. CATCH with next-line prefetching.

and it was verified that it does not degrade the performance when prefetching data blocks. Figure 17 shows the normalized IPC of the baseline with CATCH, with next-line prefetching and when applying both techniques. The results show that prefetching can significantly improve the performance of a cache but again, as with the victim cache, the performance improvement is additive for CATCH. Furthermore, there is one benchmark, Q16F, that prefetching is unable to improve its performance while CATCH can increase its IPC by 25%. This suggests that there are cases where a simple prefetcher can not predict the program behavior but the redundancy still exists in the cache and can be eliminated using CATCH.

9.7. Increasing Cache Size

Another design trade-off is to consider investing the extra space required by CATCH to increase the cache size. For example 16KB cache + 1.38KB CATCH can roughly correspond to an 18KB cache which has the same design specifications as the 16KB cache + 1 extra way. Figure 18 shows the normalized IPC of the baseline with CATCH, the 9-way 18KB cache and a combination of both the 18KB cache and CATCH. The results indicate that a cache with 2KB extra way provides higher performance than

Fig. 18. CATCH compared to an 18KB cache.

Table III. Energy Consumption per Access of 16KB 8-way and CATCH

Structure	Dynamic Energy per access(nJ)	Percentage of Cache energy
Cache 16KB 8-way	1.09001	100%
HDD 0.52KB 8-way	0.00380342	0.35%
DR 0.86KB 4-way	0.00479044	0.44%
Block Compare Unit	0.000135553	0.01%

the 1.38KB CATCH. However, it is worth mentioning that even with the extra space there is still room for improvement using CATCH. This is indicated by the extra 7% on average improvement that can be achieved using a combination of the 18KB cache and the 1.38KB CATCH compared to the 18KB cache alone.

9.8. CATCH Energy Consumption

Table III shows the dynamic energy per access, obtained using CACTI [Muralimanohar et al. 2009] of the three structures used to implement 1.38KB CATCH. This is compared with the energy of the 16KB 8-way cache used for the performance evaluation. The table shows that the energy consumption of the HDD corresponds to 0.35% of the cache energy per access while the DR consumption corresponds to 0.44%. The energy consumption of Block Compare Unit, that compares a maximum of 128 bits, 4 instructions, corresponds to 0.01% of the energy that it is consumed during a cache access. In total, the 1.38KB CATCH consumption is less than 0.8% of the cache energy consumption.

Furthermore, CATCH is accessed only when there is a cache access, the HDD is accessed only on a miss, while the DR is accessed on each cache access. The BCU is only used on a cache miss that follows an HDD miss. Therefore, the total power consumed by CATCH is less than 1% of the cache energy consumption.

10. CONCLUSIONS AND FUTURE WORK

This work introduces the notion of CCD and proposes CATCH, a hardware mechanism for dynamically detecting CCD and evaluates the performance of CATCH for two cache architectures that exploit CCD: the Duplicate-Aware-Cache and the Unique-Content-Cache.

The article reports on the performance of the proposed mechanism with oracle and realistic constraints and investigates the significance of various performance and cost optimizations. Experimental results for a processor with a 16KB, 8-way, 8 instructions

per block instruction cache show that a CATCH with 1.38KB cost usually captures 58% on average of the CCD idealized potential.

Experimental results comparing CATCH with victim cache show that CATCH can capture misses that are not due to conflicts in the same set. Thus the performance gain of the two mechanisms is additive.

This article provides 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. CCD may also be considered in combination with static code compaction to investigate the synergistic potential of the two approaches. Another important direction of research is to consider CCD for data caches and other levels in the memory hierarchy and for shared caches used in multicores and SMT processors.

ACKNOWLEDGMENTS

We would like to thank the High Performance Computing Systems Laboratory at the University of Cyprus for supporting us and providing access to the Grid resources to run our experiments. We would also like to thank all the anonymous reviewers whose guidance and punctual comments have helped us improve the final version.

REFERENCES

- ALAMELDEEN, A. R. AND WOOD, D. A. 2004. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st International Symposium on Computer Architecture*. 212–223.
- BAER, J.-L. AND CHEN, T.-F. 1991. An Effective On-chip preloading scheme to Reduce Data Access Penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 176–186.
- BEHAR, M., MENDELSON, A., AND KOLODNY, A. 2005. Trace cache sampling filter. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 255–266.
- BENINI, L., MACHI, A., MACHI, E., AND PONCINO, M. 1999. Selective instruction compression for memory energy reduction in embedded processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 206–211.
- BESZEDES, A., FERENC, R., GYIMOTHY, T., DOLENC, A., AND KARSISTO, K. 2003. Survey of code-size reduction methods. *ACM Comput. Surv.* 35, 3, 223–267.
- BISWAS, S., FRANKLIN, D., SAVAGE, A., DIXON, R., SHERWOOD, T., AND CHONG, F. T. 2009. Multi-execution: Multicore caching for data-similar executions. In *Proceedings of the 36th International Symposium on Computer Architecture*. 164–173.
- COMPAQ, C. C. 1998. *Alpha Architecture Handbook*.
- CONTE, T. M., MENEZES, K. N., MILLS, P. M., AND PATEL, B. A. 1995. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd International Symposium on Computer Architecture*. 333–344.
- COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 139–149.
- DEBRAY, S. K., EVANS, W., MUTH, R., AND SUTTER, B. D. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* 22, 2, 378–415.
- DENNING, P. J. 1970. Virtual memory. *ACM Comput. Surv.* 2, 3, 153–189.
- DOUGLIS, F. 1993. The Compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Conference*. 519–529.
- HALLNOR, E. G. AND REINHARDT, S. K. 2004. A compressed memory hierarchy using an indirect index cache. In *Proceedings of the 3rd Workshop on Memory Performance Issues: in Conjunction with the 31st International Symposium on Computer Architecture*. 9–15.
- HINES, S., GREEN, J., TYSON, G., AND WHALLEY, D. 2005. Improving program efficiency by packing instructions into registers. In *Proceedings of the 32nd International Symposium on Computer Architecture*. 260–271.
- JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*. 364–373.

- KJELSO, M., GOOCH, M., AND JONES, S. 1996. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd EUROMICRO Conference*. 423–430.
- KLEANTHOUS, M. AND SAZEIDES, Y. 2008. CATCH: A mechanism for dynamically detecting cache-content-duplication and its application to instruction caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1426–1431.
- KOMONDOOR, R. AND HORWITZ, S. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*. 40–56.
- LEFURGY, C., BIRD, P., CHEN, I.-C., AND MUDGE, T. 1997. Improving code density using compression techniques. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. 194–203.
- LEFURGY, C., PICCININI, E., AND MUDGE, T. 2000. Reducing code size with run-time decompression. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*. 218–228.
- LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 138–147.
- MALAMY, A., PATEL, R. N., AND HAYES, N. M. 1994. Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. United States Patent 5353425.
- MURALIMANOVAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2009. CACTI 6.0: A tool to model large caches. Tech. rep. HPL-2009-85, HP Laboratories.
- POSTIFF, M. A. AND MUDGE, T. 1999. Smart register file for high-performance microprocessors. Tech. rep. CSE-TR-403-99, University of Michigan.
- SENDAG, R., CHUANG, P.-F., AND LILJA, D. J. 2003. Address correlation: Exceeding the limits of locality. *IEEE Comput. Archit. Lett.* 2, 1, 3.
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.
- TULLSEN, D. M. 1996. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*. 819–828.
- WULF, W. A. AND MCKEE, S. A. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Comput. Archit. News* 23, 1, 20–24.
- ZHANG, Y., YANG, J., AND GUPTA, R. 2000. Frequent value locality and value-centric data cache design. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. 150–159.

Received March 2010; revised February 2011; accepted June 2011