Dynamically Detecting Cache-Content-Duplication in Instruction Caches

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

February 2007
TR-07-03

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 smaller 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 various types of 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 for instruction caches. Experimental results for an out-of-order processor show that a duplication-detection mechanism 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 [25]. 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 [11], predictability [2, 21], and redundancy [17, 9].

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 entire 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 the cache is not aware of the block content. 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, to have identical content. If block 115 is evicted and we have a subsequently miss on it, the content of 103 can be used without accessing a lower level of the memory hierarchy.

CCD can happen when cache blocks with different tags have exactly the same content. Therefore, CCD is a manifestation of redundancy in the cache content. There have been numerous previous papers that attempt to exploit cache

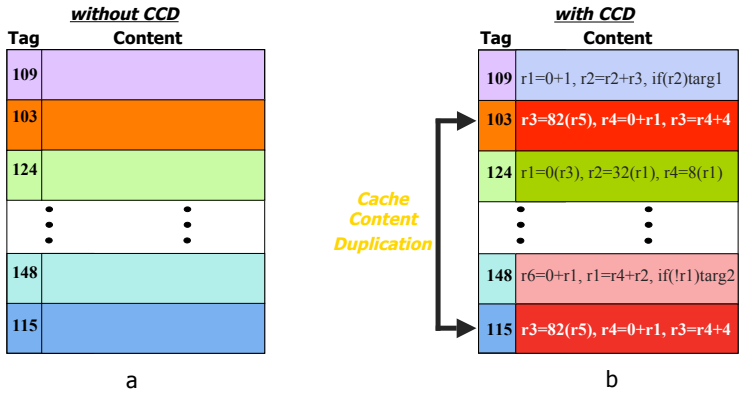

Figure 1. Cache Content Duplication

redundancy. 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. Consequently, CCD can enable new hardware mechanisms for memory hierarchy optimization. Examples of CCD based optimizations are: (a) the Duplicate-Aware-Cache (DAC) 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 from lower in the memory hierarchy, and (b) the Unique-Content-Cache (UCC) that can lower the miss ratio by allowing only blocks with unique content to enter the cache.

CCD may occur at any level of memory hierarchy for both data and instructions. However, an initial analysis, performed for L1 data caches and unified L2 caches, revealed that CCD for data is mainly due to zero blocks. Existing techniques may handle zero blocks in L2 effectively [13], or capture duplication in data L1 at the granularity of values, both for zeros and non-zeros, more efficiently [26]. On the other hand, the CCD in instruction caches will be for non-zero values and its frequency may be significant, because: (a) high level language programs often contain identical instruction sequences [18] in different segments of a program due to: copy-paste programming practices and reuse of standard library 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. Therefore, as a first step toward understanding and exploiting CCD this work is focused on the content duplication in instruction caches. In particular, the main contributions of this work are:

- the phenomenon of Cache-Content-Duplication (CCD),
- 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 limits for instruction, basic-block and trace caches, and
- 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 a 2-way eight instruction per block 16KB instruction cache show that a CATCH with a 2.32KB cost usually captures 60% or more of the CCD's idealized potential

The paper is organized as follows: Section 2 discusses previous work on cache redundancy. Section 3 discusses issues related to CCD detection. Section 4 presents the simulation environment. Section 5 examines the limits of CCD for various types of instruction caches. Section 6 considers two possible applications of CCD and investigates their performance potential. In Section 7, we introduce CATCH and discuss different optimizations to improve its cost-efficiency. Section 8 evaluates the performance of CATCH under realistic constraints. Section 9, 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 of these proposals were to increase the effective memory/cache capacity and to achieve higher bandwidth during transfers of information between different levels of the memory hierarchy. Some of these papers are discussed below.

A scheme for main memory on-line compression was first proposed by Douglis [12]. The compression cache proposed allows both software-based and hardware-based compression using different compression algorithms. Benini et al. [4] 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 and Gooch [17] proposed a hardware implementation of the X-Match dictionary compression algorithm for main memory data. Lefurgy et al. [19] 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. [20] 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.

Alameldeen and Wood [1] 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 [14] 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 [22] 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. [15] 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 high redundancy of a limited subset of values in data caches was identified in [26]. A frequent-value cache was proposed to hold in compressed form the frequent values.

Very relevant to our work is the paper by Sendag et al. [24] that 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 [24] 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 the research in the area of code compaction [5, 9, 10]. Code compaction methods are used to reduce the executable code size without a need to decompress the compacted code to execute it. The main idea behind most compaction techniques is to have the compiler back-end identify repeated sequences in a program and eliminate the repetition by either *cross-jumping* or *procedural abstraction*. Cross-jumping replaces all instances of a repeated sequence with a jump to a new location that contains a single copy of the repeated sequence. Procedural abstraction is used to convert a repeated sequence to a procedure and replace the repeated sequences with calls to this procedure. Control flow dominance criteria are used to decide which of the two methods is applied in each case of repetition [5].

Code compaction transformations have also been proposed to convert "superficially" dissimilar sequences to repeated

[9]. For example two sequences can perform exactly the same computation using different registers. These differences can be eliminated using move instructions to rename registers prior and after executing a compacted repeated sequence.

The main cost of 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.

Previous code compaction work and this work share similarities but also differences. Both approaches aim to detect and exploit redundancy in instruction sequences. However, the compaction methods are compiler based where 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 re-optimization. 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, the key distinct feature of our work is that we consider redundancy at the granularity of cache blocks and detect it dynamically using a hardware mechanism. Previous work considered the redundancy, compression and compaction of arbitrary length sequences of data or instructions, or considered 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 for Instruction Caches

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. The discussion is concerned with the following types of instruction caches:

(i) regular instruction cache,

(ii) a basic-block cache [6], where blocks are divided on the boundaries of control flow instructions and identified by their starting address. A basic-block is a sequence of instructions where only the first instruction is an entry and only the last instruction is an exit. Consequently, all instructions in a basic-block get executed as long as we enter the block. The cache block, in a basic-block cache, contains either an entire basic-block or a partial basic-block when it is larger than a cache block. A basic-block cache is used in the block-based trace cache proposed in [6], and

(iii) a trace cache [23] with the following trace termination criteria: (a) the maximum number of instruction has been reached, (b) the maximum number of basic blocks has been reached, (c) the last instruction is an indirect jump or a system call, and (d) a basic block, other than the first in the trace, that is larger than the remaining space in the trace. A trace is identified with a 33-bit value. The 28 most significant bits of the trace-id correspond to the address of the first instruction in the trace. The five least significant bits of the trace-id represent the direction of three conditional branches and the number of basic blocks in the trace. This report considered traces with a maximum of eight instructions. When the last instruction of the trace is a conditional branch then the direction of the branch is not recorded in the trace-id. This prevents trace duplication with the same starting program counter and the same content, but with different direction of the last branch [23].

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 because the contents of a cache block have different semantics depending on the instruction cache type.

For an instruction cache a block always contains a block size number of instructions starting from the block address, whereas for a basic block and a trace cache the block may contain (a) less than block size instructions, and (b) the block starting address usually corresponds to the beginning of a basic block.

It is expected that CCD will be more likely between blocks that have fewer instructions 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.

According to the above qualitative discussion, it is expected that CCD will be more common for a basic block cache (one small block and aligned), less prevalent for a trace cache (many blocks but aligned) and even lower for an instruction cache (many blocks that are non-aligned).

One way to increase the frequency of CCD for regular instruction caches is to consider the duplication between *valid* instruction sequences sent down the pipeline on an I$ access, instead of *entire* instruction cache blocks. In [8] a *valid* sequence is defined as the static consecutive instruction sequence starting from the current PC until: the first conditional branch that is predicted taken, or the first unconditional branch, or fetch bandwidth number of instructions are read from the cache. A valid block is identified by the starting PC and a bit mask. The mask size is equal to the cache fetch bandwidth. The bit mask can be produced each cycle, in a pipeline, using the BTB and the direction predictor [8]. The mask indicates the location of the first taken branch in a sequential instruction sequence. A valid block represents, therefore, the predicted instructions that are send down the pipeline after a cache access, and we will refer to it henceforth as a *valid block*.

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. The distinction between an *entire* cache block and a *valid* block is only applicable to regular instruction caches, since for basic block cache and trace cache the valid block is virtually always the same as the entire block content.

Section 5 considers the CCD limits for all of the above instructions caches and block types. Section 8 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 new block comes in the cache, it can be detected whether or not it's content is duplicate with a block already in the cache.

For a basic-block cache and a trace cache the content in all cache blocks can be learned by remembering the content of the blocks when inserted in the cache. This is referred to as *learn-on-miss* learn policy. This policy is also sufficient to learn all the content in regular instruction caches when considering duplication of entire cache blocks. However, for valid blocks 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.

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. Benchmarks simulated

fetch/issue/commit	4/4/4
Queue/LSQ/ROB	64/32/64
Stages	14
ALU/Data Cache Ports	4/2
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 latency
Main memory latency	200 cycles
Cond. branch predictor	8KB combining predictor
BTB	1024 entries
RAS	32 entries
Indirect predictor	512 entries

Table 2. Out of Order Processor Configuration

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. However, this policy can be inefficient since it may learn the same valid block multiple times. Furthermore, to learn the valid blocks in a cache block may require multiple block accesses, with some CCD potential lost in the intervening time. An alternative method, that may increase CCD frequency, 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, for learning entire blocks the *learn-on-miss* policy is used, and for learning valid blocks the *learn-all-on-miss* policy is employed that learns the missed blocks and an additional valid block, as described above. The importance of the learn strategy on CCD for valid blocks is investigated in Section 8.

3.3 What are the criteria for two sequences to be classified as duplicates

Two blocks, sequences of instructions, are considered duplicates if each instruction in one 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 fields with immediate offsets or targets in order to increase duplication frequency. This technique is known in the area of code compaction as target abstraction[5]. Section 7 discusses, in more detail, how the use of 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 [5], can be applied to increase the duplication frequency, but 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 instruction set architecture [7]. For almost all of the experiments 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 across the cache configurations we considered and, therefore, more likely to benefit from the techniques proposed in this report. Table 1 shows the dynamic instructions skipped and simulated for these benchmarks.

We assess and compare the performance impact of the different techniques using both functional (modified sim-fast [7]) and performance (modified sim-out-of-order [7]) simulators. The functional simulator is used for quick characterization

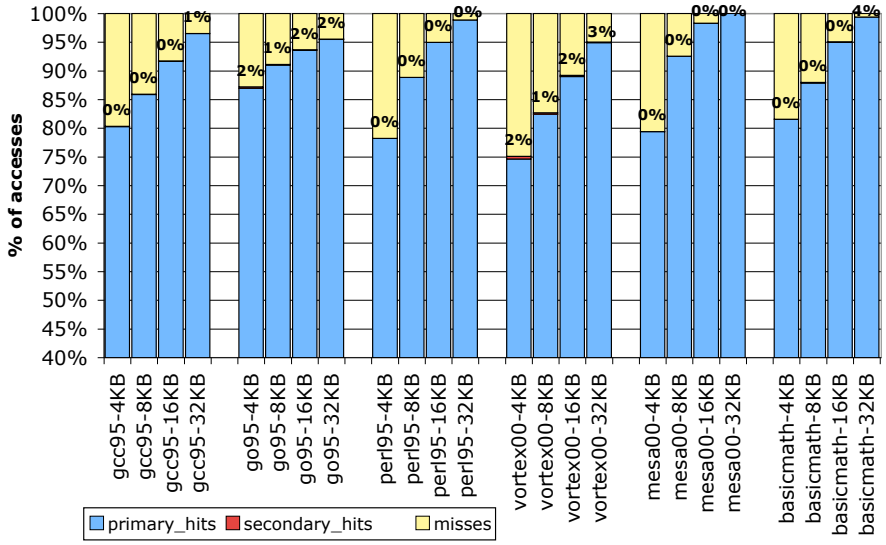

Figure 2. CCD for a 2-way, 8 instructions per block, instruction cache, for entire blocks

of the design space in Sections 5 and 6. The performance simulator is used in all other cases to model an out-of-order processor with the configuration listed in Table 2. The performance metrics used in this study are instructions per cycle (IPC), hit/miss rates and the CCD (or duplication) rate of each benchmark. The CCD rate refers to the fraction of misses that are for duplicate-blocks.

5 Limits of Cache-Content-Duplication

In this section we establish the CCD limits for an instruction cache, a basic-block cache and a trace cache using a functional simulator. 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. For these experiments the cache operation remained unaffected by secondary-hits, i.e. the missed block is always fetched and inserted in the cache.

5.1 CCD in Instruction Caches for Entire and Valid Blocks

Fig. 2 shows the breakdown of accesses into hits (primary-hits), secondary-hits (hits on duplicated blocks) and misses with a 2-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 on each bar.

The results show that CCD for entire instruction cache blocks is a rare phenomenon, never more than 4% 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.

In Section 3 it was suggested that one possible way to overcome the above limitations, and increase CCD rate, is to consider the duplication for valid block. Fig. 3 presents the CCD for valid blocks. The data show that the CCD rates for valid blocks is often above 15% and therefore more prominent than for entire cache blocks (Fig. 2). This increase

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

The general trend in Fig. 3 is that with increasing cache size the amount 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. This occurs because with a larger cache, it is more likely for a missed valid block to already have a duplicate in the cache.

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 a 2-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 [8]. Section 8 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, for regular instruction caches we consider CCD for valid blocks because it has higher potential.

5.2 CCD for Basic-Block Caches

Fig. 4 presents the breakdown of accesses for a 2-way 16B (4 instructions) block basic-block cache. The data show clearly that across all benchmarks CCD is more prevalent with a basic-block cache as compared to an instruction cache (Fig. 3). The results show that the duplication rate for several cases is above 25% and can be as high as 42%. The trend with increasing cache size is higher CCD rates.

The increased occurrence of CCD for a basic block cache is because smaller and aligned sequences are checked for duplication. Also, on a miss we only fetch one basic block. Consequently, basic block caches have low hit rates (compare primary hits in Fig. 3 and Fig. 4) and thus more opportunity for missed blocks to be duplicates.

With bigger block size (results not shown) the frequency of CCD remains at the same levels. This mainly occurs because the typical basic-block size is 4-5 instructions. This suggests that for a basic-block cache larger block size may result in block fragmentation and higher miss rates. This was confirmed by experimental data that show, for equal size basic-block caches, larger block usually meant higher miss rate.

We have also examined the effects of varying associativity, the CCD trends were very similar with the 2-way basic-block cache. This reinforces the hypothesis that CCD can not be eliminated with a victim cache.

5.3 CCD for Trace Caches

The high frequency of CCD for basic-block caches suggests the possibility of CCD in trace caches since each trace contains one or more dynamically consecutive basic blocks. Fig. 5 presents the breakdown of accesses for a 2-way set-associative trace cache with 32B (8 instructions) block sizes. The data show CCD to exist in every benchmark, often with rates above 8% and up to 25%. Its frequency is comparable to CCD for valid blocks in an instruction cache (Fig. 3) but lower than a basic-block cache (Fig. 4). The CCD for traces with sixteen instructions (results not shown) is less, since longer sequences are more difficult to match, but still significant. The CCD behavior was found to be insensitive to the degree of associativity of a trace-cache.

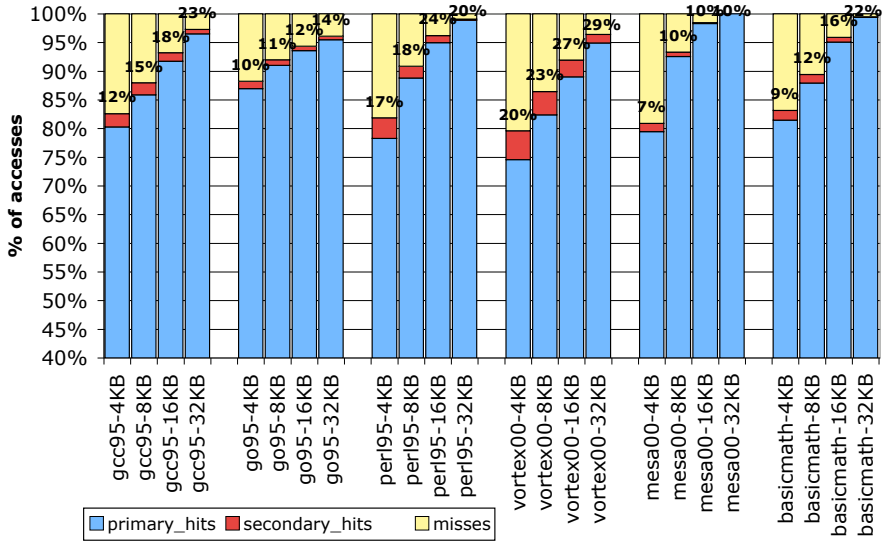

Figure 3. CCD for a 2-way, 8 instructions per block, instruction cache, for valid blocks

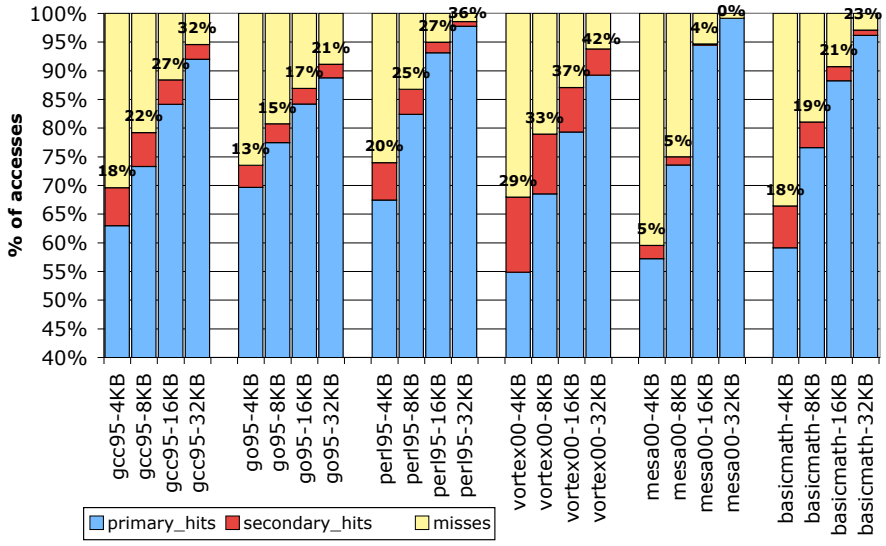

Figure 4. CCD for a 2-way, 4 instructions per block, basic block cache

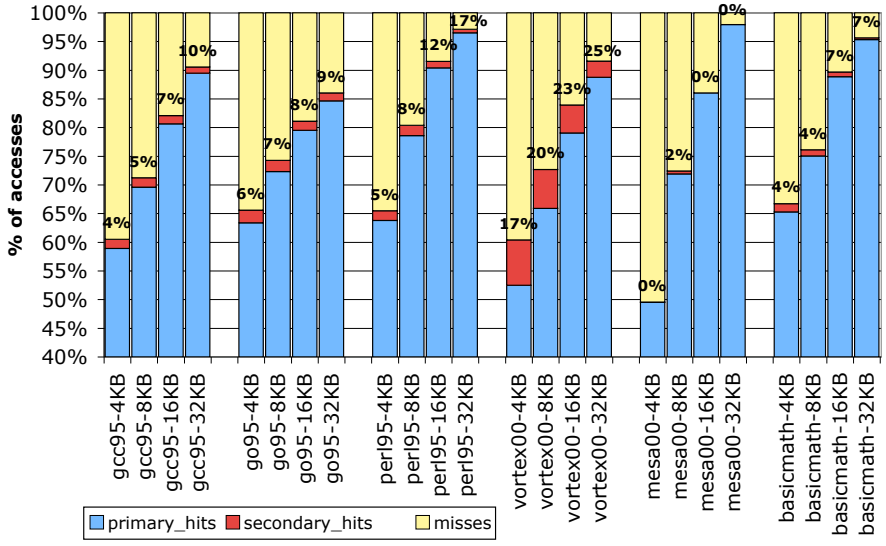

Figure 5. CCD for a 2-way, 8 instructions per block, trace cache

5.4 Overall Observations

Overall, the experimental results in this section suggest that CCD exists across benchmarks, for different cache types and configurations. The data indicate that with increasing cache size the relative importance of CCD also increases. The behavior across benchmarks varies, with higher rates for gcc95, perl95 and vortex00 and lower for go95, mesa00 and basicmath. We believe that the degree of the observed CCD provides a motivation to explore the development and performance of mechanisms that can exploit CCD.

6 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. We refer to such cache as the Duplicate-Aware-Cache (**DAC**). Latency can be reduced by fetching the block from the cache instead of reading it from lower in the memory hierarchy. 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. Note that 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.

CCD can also be used to reduce misses by allowing only blocks with unique content to enter a cache. We refer to such cache as the Unique-Content-Cache (**UCC**). A UCC can reduce 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 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 and misses.

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

6.1 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 were shown in Fig. 3, 4 and 5 for an instruction cache, a basic-block cache and a trace cache respectively.

To establish the potential of a UCC cache over a regular cache we performed an oracle study with the same assumptions as in Section 5. The UCC cache is modeled as a regular cache except when there is a miss that has a duplicate block in the cache - i.e. 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.

Fig. 6 shows the breakdown of memory accesses for a UCC-instruction cache for valid blocks, Fig. 7 the breakdown for a UCC-basic-block cache, and Fig. 8 the breakdown for a UCC-trace cache. For comparison purposes, the graphs

show, using error-bars, the accesses that were hits for the respective regular cache. Also, included in the graphs are numeric values for the CCD-rate, that do not correspond to baseline primary hits, over the baseline misses.

A comparison of Figs. 3–5 with Figs. 6–8, 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 DAC trace-cache *vortex00* has CCD rates 23% where its corresponding rates for UCC are 35%. The reason for this increase is that by avoiding the insertion of duplicate content in a UCC cache a lot of capacity and may be conflict misses are eliminated,. Consequently, reducing the total number of misses by more than the duplicate misses observed in a DAC.

The data also show, however, that a UCC cache can have fewer primary hits than a regular cache. For example, for a 16KB UCC instruction cache for valid blocks, *vortex00* has 13% secondary hits and only 79% primary hits (the baseline had 89% hit ratio). Therefore, only 3% out of the 13% secondary hits correspond to cache misses that were converted to secondary hits. The remaining 10% of accesses correspond to primary hits in the regular cache that were converted to UCC secondary hits. The above shows that, unlike DAC, a UCC cache 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.

The focus of the remaining paper is on the CCD for regular instruction caches because these are the most widely used instruction caches in computing systems.

6.2 Performance Potential of CCD

Fig. 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 5 and 6.1.

Results are presented for secondary hit latencies of 0, 1, 2 and 3 cycles (denoted in the graph as DAC-0, DAC-1, DAC-2 and DAC-3 or UCC-0, UCC-1, UCC-2 and UCC-3 respectively) and for various L2 cache latencies (10, 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 CDD with increasing latency to lower levels of memory hierarchy. All other processor parameters are as in Table 2.

The data show both DAC and UCC to have performance potential up to 15% and 20% respectively. The benchmarks *gcc95*, *perl95* and *vortex00* with the highest CCD rates, see Figs. 3 and 6, are also the ones with the largest potential. Benchmark *mesa00* does not benefit from CCD because it has very few misses for duplicated blocks. 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 be detrimental. For example with UCC-3 latency many configurations, for *go95* and *perl95*, suffer a performance degradation. The results also reveal that the potential of UCC is higher than DAC only when the secondary hit latency is less than three cycles.

The low UCC performance, for three 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. Finally, although the limits of CCD rates for DAC and UCC are very similar, as shown in figures 3 and 6, the results in Fig. 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 in the case of two consecutive accesses to the same missed block (for different valid blocks), the first will be a secondary hit and the second will be a primary hit. But in a realistic scenario with non-zero fetch latency it is possible to have secondary hit and the next access to cause a miss because the block is not yet available.

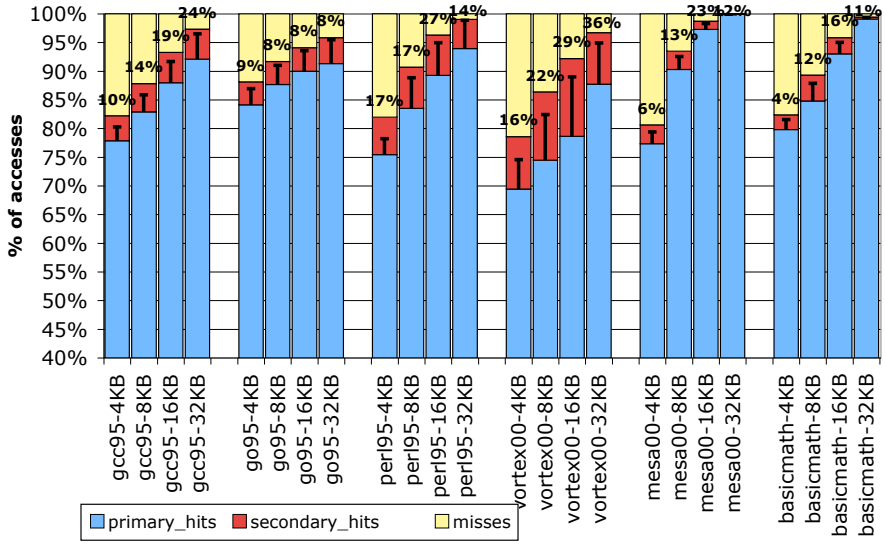

Figure 6. CCD for a 2-way, 8 instructions per block, instruction cache with UCC, for valid blocks

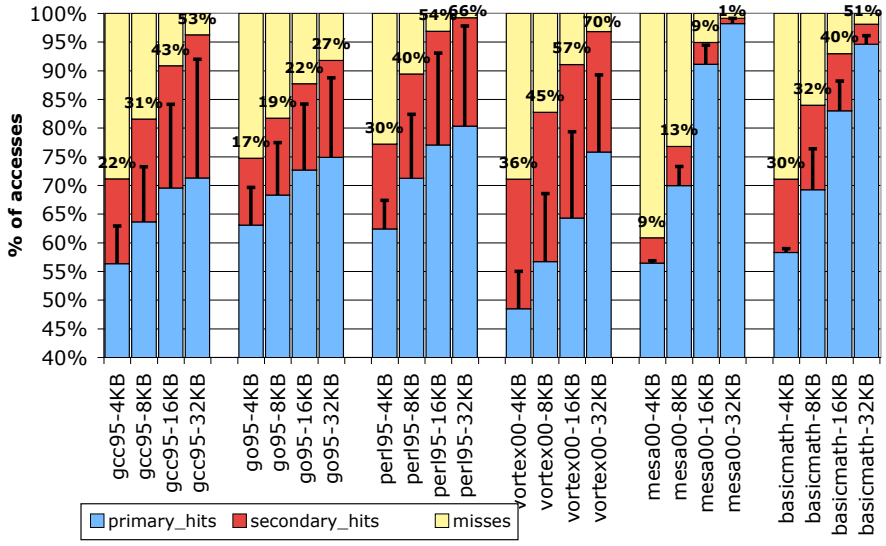

Figure 7. CCD for a 2-way, 4 instructions per block, basic block cache with UCC

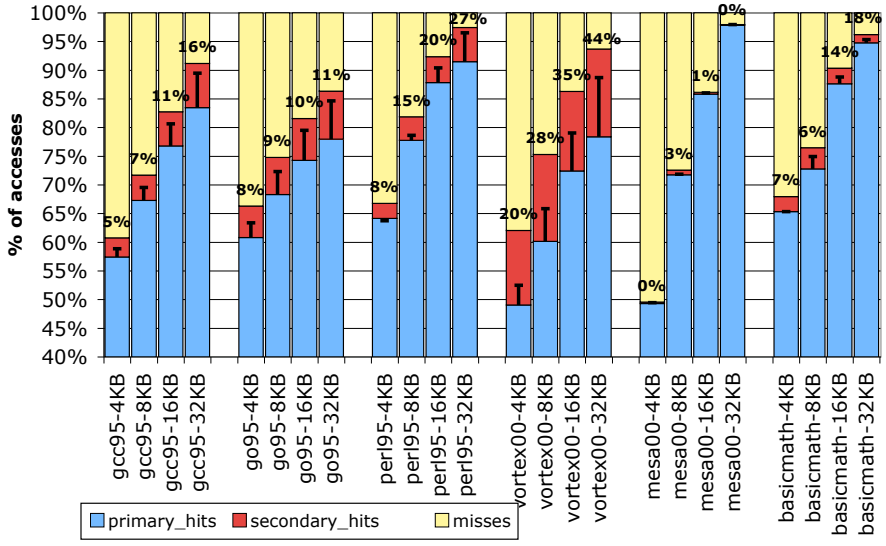

Figure 8. CCD for a 2-way, 4 instructions per block, trace cache with UCC

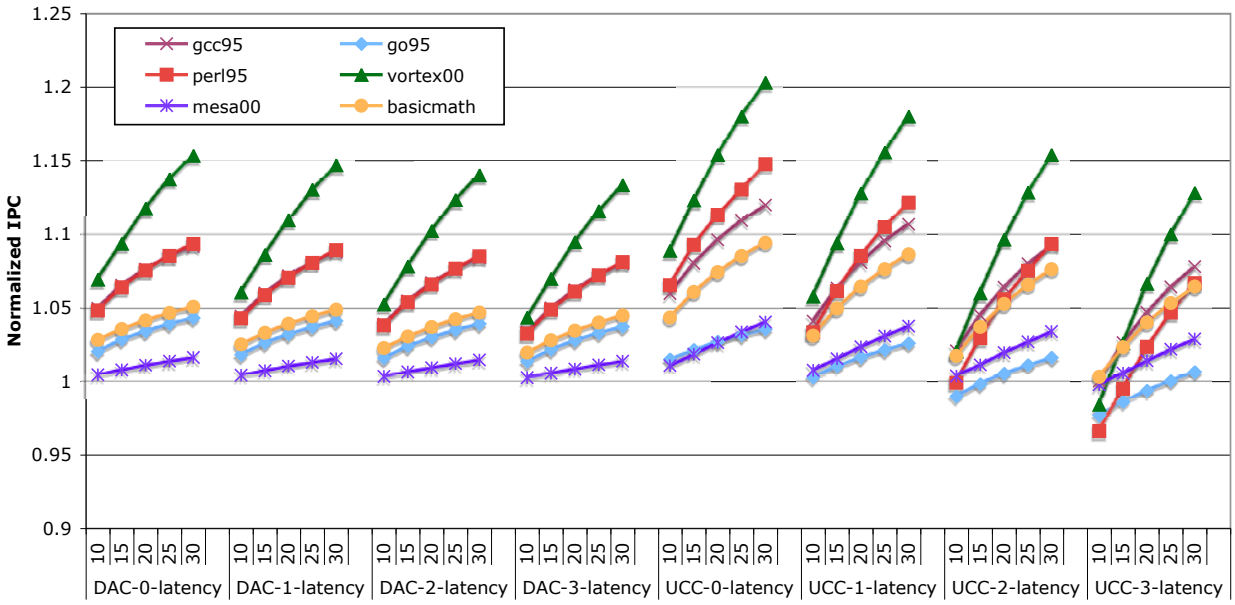

Figure 9. Performance potential of DAC and UCC for a 2-way, 16KB, 8 instructions per block, instruction cache, for valid blocks (Normalized IPC)

The performance potential analysis suggests that, for good performance and long duplicated hit latency, the DAC with two or three cycle latency is a good compromise but for better performance with non-zero CCD detection, UCC with one cycle latency is better.

In the case of a single cycle latency cache, a single cycle secondary hit latency can be achieved by accessing the CCD mechanism and cache in parallel. By the end of the cache access cycle the CCD mechanism will provide an alternative tag-index to access the cache in case of a miss. 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 7.8.

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

7 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 Fig. 10. 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.

7.1 The Duplicate-Relation cache (DR)

The duplicate-relation cache (DR) contains block 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

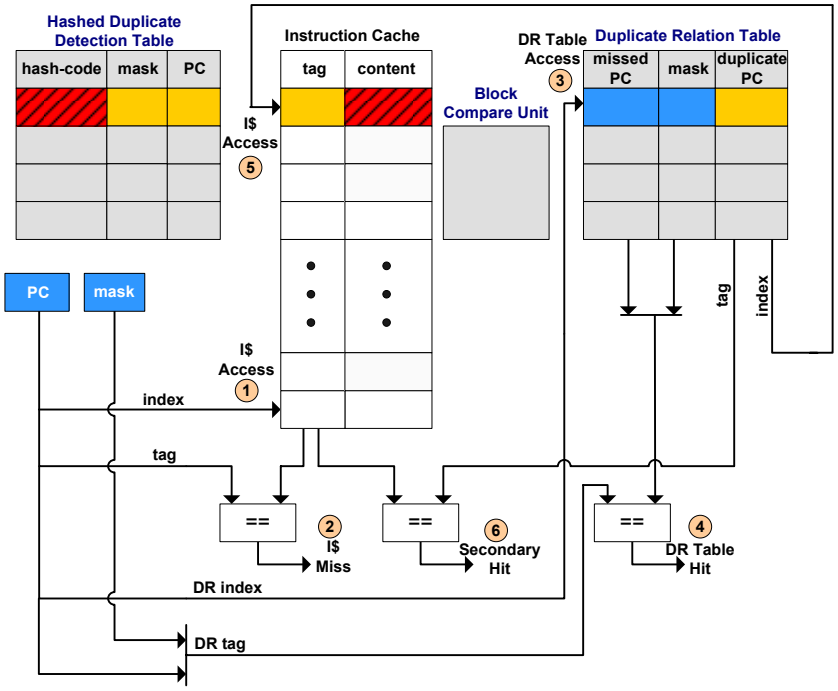

Figure 10. The CATCH and the flow for a Cache miss, DR hit, Cache hit

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 tagged 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. On the other hand, using a physically tagged DR will eliminate this overhead but we may need to flush the DR 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 valid 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. Fig. 10 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.

7.2 The Hashed-Duplicate-Detection cache (HDD)

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

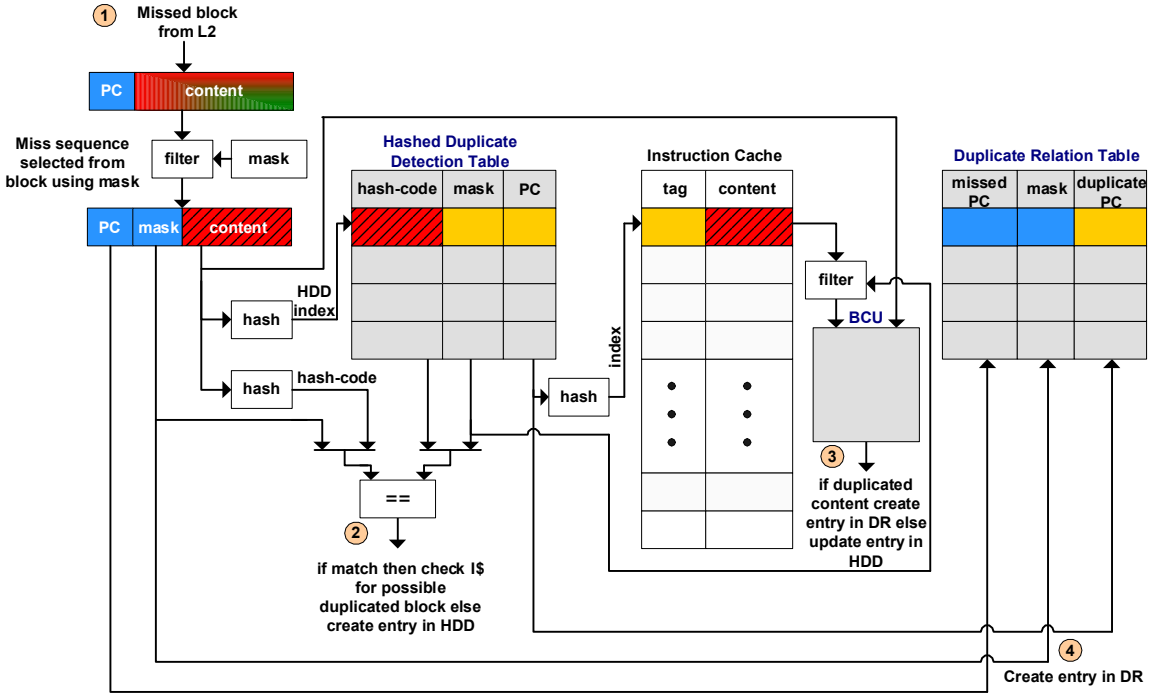

Figure 11. The flow with CATCH for a Cache miss, DR miss and HDD hit

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. 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 that case, the cache is accessed using the starting PC found in the HDD to determine whether the two blocks are indeed duplicates. The testing 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. 11 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 a DAC and UCC.

7.3 The Block Compare Unit (BCU)

When two blocks are signaled by the HDD as possible duplicates, their contents are compared using the 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 7.6.

7.4 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:
   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 lever of memory hierarchy and the hash-code of its content is calculated. The HDD is accessed with this hash-code and 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 that misses because the duplicate block was evicted. If we miss in the HDD then an entry is allocated and points to the newly fetched block in the cache.

There 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, and if we hit in the HDD then the cache is accessed with the duplicate-PC in the HDD. The two blocks contents are then compared in the BCU and if there is duplication 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 newly fetched block in the cache (the implications of not-updating the HDD in this case are discussed later in Sections 7.6). 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 with 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.

7.5 The use of CATCH in DAC and UCC

A DAC and a UCC can use the CATCH, as described above, 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 to 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 can not detect any duplication will the block be inserted in the cache.

7.6 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_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 for each duplicate block its offsets and targets. This is aimed to convert blocks that contain exactly the same computation to duplicates. This is effectively a hardware implementation of the target abstraction discussed in Section 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 extra cost is considered to the total size of the mechanism calculated in Section 7.7. The fetch complication 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 [9]. These and other transformations to be discovered may help uncover even more duplication.

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. [3]. 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 a more extended discussion on the working of this method we refer the interested reader to [3]. 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 it indirectly reduces the updates to the DR.

The significance of the *keep_offset_in_dr* and the filtering optimizations is investigated in Section 8.

7.7 Cost Reduction Optimizations

This Section describes several optimizations for reducing the amount of state required by the HDD and DR caches assuming a 2-way, 32B block, 16KB instruction cache with four instructions maximum valid sequence length.

Before computing the cost for a DR entry, recall that a DR entry represents logically two full tag-indices. For the PISA instruction set architecture used in this work [7], the first tag-index contains 30 bits (28 bits for the address of the first instruction of the missed sequence, 2 bits for the number of valid instructions in the sequence, which must be the same with its duplicate sequence for a duplicate relation to exist) and the second tag-index contains 28 bits for the address of the first instruction of the duplicate sequence. The non-optimized cost of a DR entry is therefore (2x28 + 2 - $\log_2$(number of sets of the DR)) which is the sum of the two addresses and the length of sequence minus the index of DR.

The DR entry cost can be reduced by introducing criteria for when to insert a duplicate relation in the DR. The following criterion was reached after some cursory analysis: do not insert a relation in the DR unless the most significant 9 bits of the starting address are the same for the two tag-indices. This reduces the cost of a DR entry by 9 bits.

When the *keep_offset_in_dr* optimization is employed, the DR should have space for four direct targets (it is possible to have four consecutive not taken branches). To reduce the number of bits required by the offsets and direct targets extra insertion criterion can be applied. These criteria are to insert duplicated relations only 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 these criterion in place, the extra cost of the keep_offset_in_dr optimization is 16 bits for each DR entry (16 bits for an offset or a target). Note that for the ISA used in this study target offsets for conditional branches are 16 bits and direct targets are 26 bits. Therefore, for the DR the per-entry cost without cost optimization is (2x28 + 2 - $\log_2$(number of sets of the DR)) + 3x16 + 26 bits and with is (28 + 19 + 2 - $\log_2$(number of sets of the 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. In Section 8 we consider the performance with a 16 bit hash-code. Furthermore, we can use the hash-code used for tag-matching the sequences in order to index the HDD. This will reduce the HDD entry by $\log_2$(number of sets of the HDD) bits. Finally the criterion used in DR (do to not insert a relation in the DR unless the 9 most significant bits of the two tag-indices are the same) can be used here also. That means we only keep the 21 least significant bits in the HDD and combine 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 without cost optimization is 16 + 28 + 2 - $\log_2$(number of sets of the HDD) bits and with is 16 + 19 + 2 - $\log_2$(number of sets of the HDD) bits. In Section 8, we compare the performance with and without the cost optimizations.

7.8 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 7.

The above discussion is by no mean complete, nonetheless, it suggests that a zero cycle latency duplicate hit may be feasible.

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. Future work should evaluate the proposed mechanism using a more realistic setting.

8. 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 6) it can be captured by a more feasible to implement hardware configuration of CATCH. The analysis is focused on the performance of a 16KB instruction cache that is 2-way 32B per block, with single cycle secondary hit latency and 20 cycles L2 cache latency.

8.1 CATCH performance for DAC and UCC Caches

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

Overall, from the data is evident that CATCH can capture often more than 50% of the potential limit of DAC and more than 85% of the potential limit of UCC.

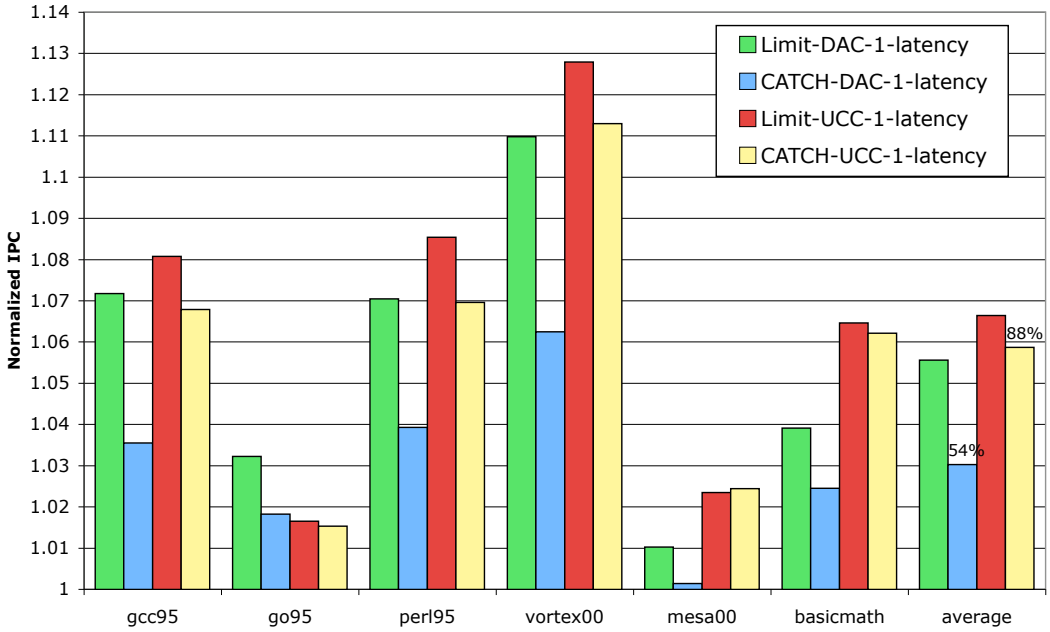

Figure 12. Performance potential captured by (a) oracle detection (limit) and (b) CATCH for DAC and UCC (20 cycles L2 cache latency)

The lower CATCH potential is due to the optimism in the oracle study that allowed to serve 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. UCC can still benefit from the first detection of a relation because it does not insert the duplicate block in the cache.

One interesting observation from Fig. 12 is that for mesa00 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. Our hypothesis for 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 repeatedly is evicted due to conflicts. On the other hand, a UCC based on 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 better performance as compared to the oracle UCC.

For the remaining of Section 8 we focus on optimizing the performance of the 16KB UCC instruction cache due to its higher performance potential.

8.2 CATCH performance with real constraints

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 significance of the various performance and cost optimizations.

Analysis, not shown here, suggests that a 4-way 256 DR and a 2-way 128 HDD represent a good performing CATCH configuration. This configuration can provide an average IPC improvement of 4% which corresponds to 60% of the performance potential of a UCC with oracle CCD detection (Fig. 13). Note that this CATCH configuration has 4.56KB state cost and employs all the performance optimizations but none of the cost optimizations.

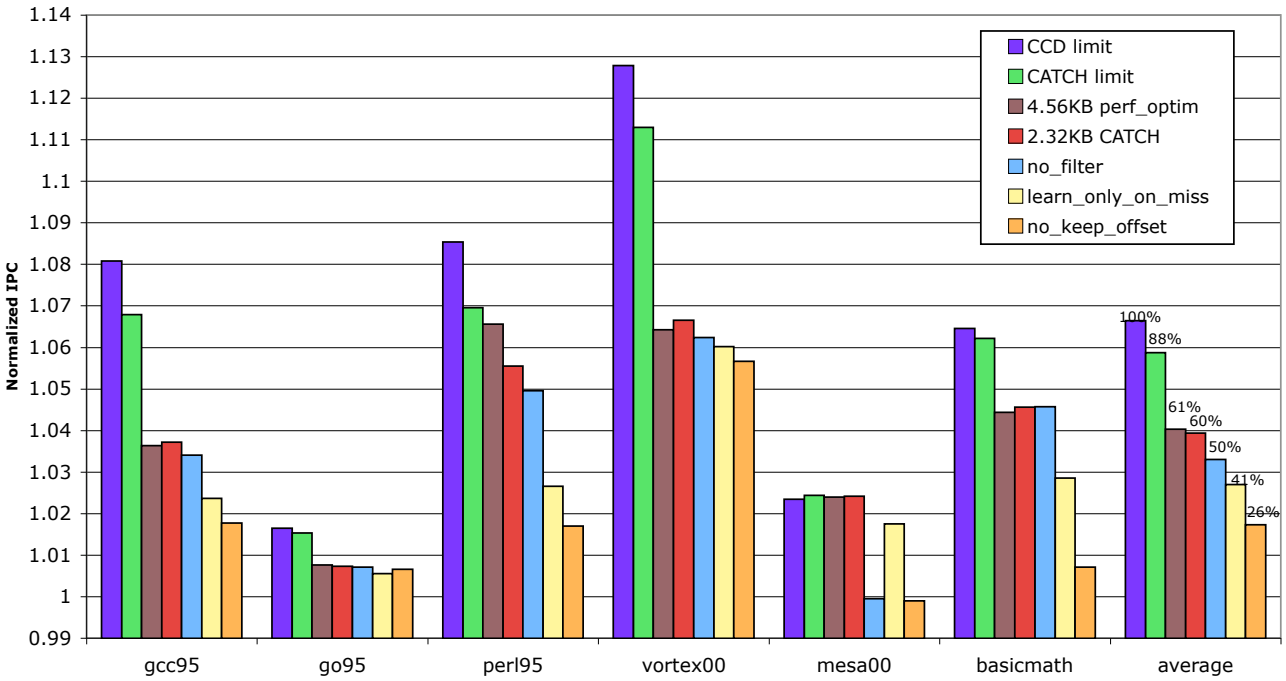

Figure 13. Effects of applying different policies on CATCH performance

To reduce the state cost of CATCH we applied the various cost optimizations discussed in Section 7.7. This lead to a reduction in CATCH cost to 2.32KB but with negligible performance degradation (on the average 0.1%).

Fig. 13 also quantifies the significance of three different performance optimizations, discussed earlier in the paper, on the 2.32KB engineered version of CATCH. The results show that without filtering the performance is degraded by 10%, without learning an additional valid sequences on a miss the degradation is 19%, and without the target abstraction the performance benefits are reduced by 34%.

We like to note that we have also experiment with the 2.32KB engineered version of CATCH with more than twenty SPEC and MiBench benchmarks that had minimal cache misses and established that CATCH did not degrade their performance.

8.3 CATCH vs Victim cache

An alternative mechanism to reduce cache misses is the victim cache [16]. 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. Fig. 14 show the performance improvement of regular cache using an 8 entry victim cache, using the CATCH with 2.32KB cost, and a combination of the two. In the combination the victim cache is accessed first and only in a case of a victim miss the CATCH is used.

The data show that in some benchmarks, gcc95, vortex00 and basicmath, CATCH is better than the victim cache while for the others victim cache is superior. However, the more important observation from this graph 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.

9 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 two cache architectures that exploit CCD: the Duplicate-

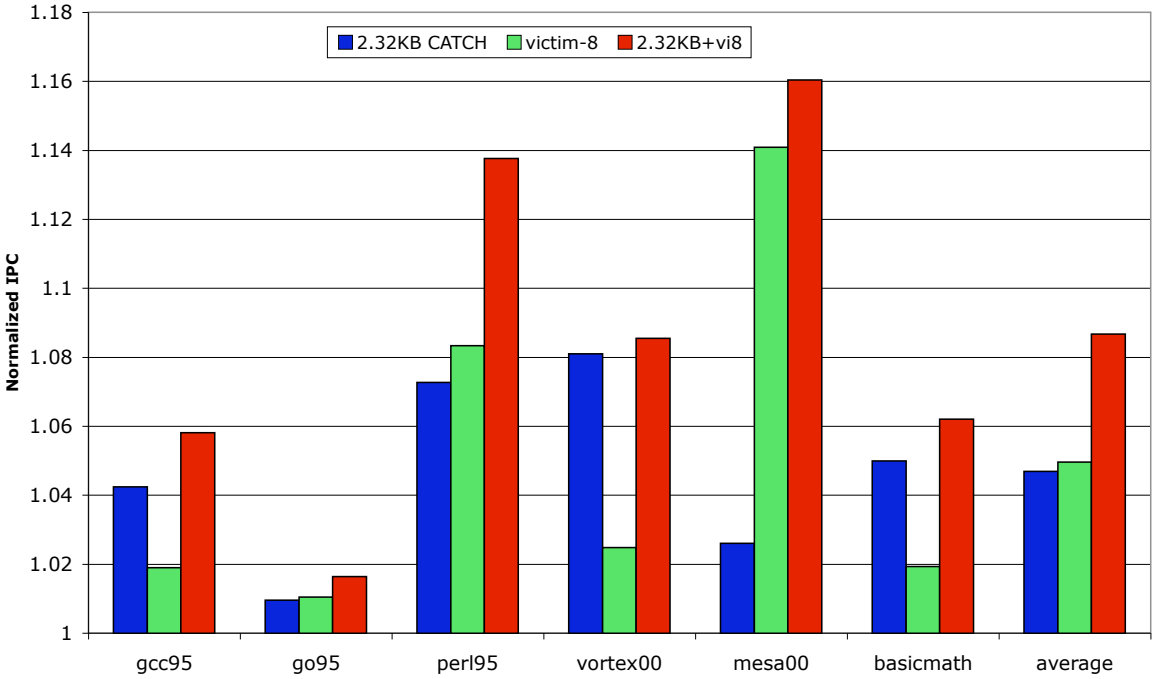

Figure 14. CATCH compared to an 8 entry Victim Cache

Aware-Cache and the Unique-Content-Cache.

The paper 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 an out-of-order processor with a 2-way eight instruction per block 16KB instruction cache show that a duplication-detection mechanism with a 2.32KB cost usually captures 60% or more of the Cache-Content-Duplication 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 mechanism is additive.

This work points to several direction of future work. One is to investigate other methods that tolerate block differences that can lead to higher CCD frequency. A mechanism for zero cycle duplicated hit latency may be also useful to evaluate and design. One other 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] R. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 212–223, June 2004.

[2] J. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, November 1991.

[3] M. Behar, A. Mendelson, and A. Kolodny. Trace Cache Sampling Filter. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 255–266, September 2005.

[4] L. Benini, D. Bruni, A. Macii, and E. Macii. Selective Instruction Compression for Memory Energy Reduction in Embedded Processors. In *International Symposium on Low Power Electronics and Design*, pages 206–211, August 1999.

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

[6] B. Black, B. Rychlik, and J. P. Shen. The Block-based Trace Cache. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 196–107, May 1999.

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

[8] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

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

[10] S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2), March 2000.

[11] P. J. Denning. Virtual Memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, September 1970.

[12] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *In Proceedings of 1993 Winter USENIX Conference*, pages 519–529, January 1993.

[13] M. Ekman and P. Stenstrom. A Robust Memory Compression Scheme. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[14] E. G. Hallnor and S. K. Reinhardt. A Compressed Memory Hierarchy using an Indirect Index Cache. Technical Report CSE-TR-488-04, University of Michigan, March 2004.

[15] S. Hines, J. Green, G. Tyson, and D. Whalley. Improving Program Efficiency by Packing Instructions into Registers. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 260–271, June 2005.

[16] N. P. Jouppi. 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*, June 1990.

[17] M. Kjelso and S. J. M. Gooch. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd EUROMICRO Conference*, pages 423–430, September 1996.

[18] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *8th International Symposium on Static Analysis*, pages 40–56, July 2001.

[19] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 194–203, December 1997.

[20] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing Code Size with Run-time Decompression. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 218–227, January 2000.

[21] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[22] M. A. Postiff and T. Mudge. Smart Register File for High-Performance Microprocessors. Technical Report CSE-TR-403-99, University of Michigan, June 1999.

[23] E. Rotenberg, S. Bennett, and J. E. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers*, 48(2):111–120, February 1999.

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

[25] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[26] Y. Zhang, J. Yang, and R. Gupta. 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*, pages 150–159, November 2000.