UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Characterizing Cache-Content-Duplication and Its Applications to Instruction Caches

Marios Kleanthous

Supervising Professor

Yiannakis Sazeides

This thesis is submitted to the Graduate Faculty of University of Cyprus in partial fulfillment of the requirements for the Degree of Master of Science at Computer Science Department

June 2006

Acknowledges

I would like to thank my family for the support gave me during my studies. I am also grateful to my supervising professor Yiannakis Sazeides for his guidance during the last two years of my studies which gave me the opportunity to write this thesis. Furthermore, I would like to express my thanks to the members of the Xi Computer Architecture Group and especially Panayiotis Charalambous and Theofanis Konstantinou for their help, feedback and comments. At last I am deeply grateful to my fiance Georgia for her moral support during the last two years of my studies.

Summary

Cache-content-duplication (CCD) occurs when there is a miss for a block in a cache and the required block of data resides already in the cache but under 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 from 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 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 cache-content-duplication, and considers its application to instruction caches. Experimental results for an out-of-order processor show that a duplication-detection mechanism with a 2.14KB cost captures usually 60% or more of the cache-content-duplication idealized potential.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance [23]. 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, cache size required, 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], and redundancy [1], [14], [15], [17], [19], [18], [20], [24], [22], [10], [5], [13], [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 for which the required block of data reside already in the cache but under a different tag as shown in Fig. 1.1.

CCD is a manifestation of redundancy in cache content. 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. And this enables 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 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 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 zeroes and non-zeroes, more efficiently [24]. Consequently, as a first step toward understanding and exploiting CCD this work is focused on the

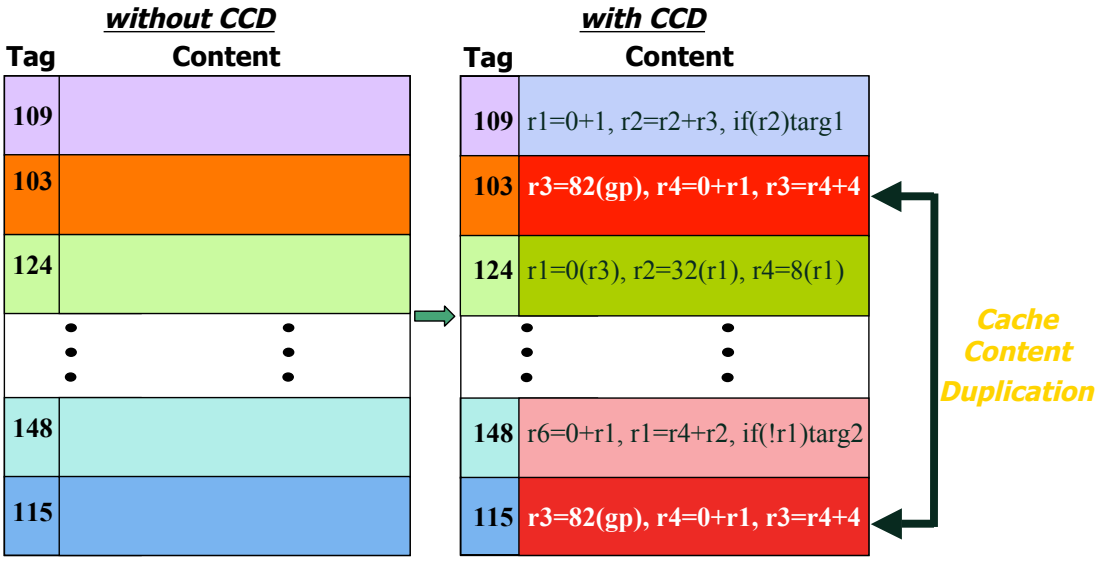

Figure 1.1. Cache Content Duplication

content duplication in instruction caches. The frequency of CCD in instruction caches may be significant, because: (a) high level language programs often contain identical instruction sequences 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 and (d) instruction blocks have less variance, than data blocks, because usually their content is not modified.

This work makes the following contributions related to CCD:

1. Introduces the phenomenon of cache-content-duplication
2. Present various policies and methods to increase CCD frequency
3. Characterizes the CCD frequency for instruction, basic-block and trace caches, assuming oracle CCD detection
4. Presents two new cache types - the duplicate-aware-cache and the unique-content-cache - that can exploit the phenomenon, and performs an evaluation of their potential with oracle CCD detection
5. Introduces CATCH, a hardware mechanism that can dynamically detect cache-content-duplication and investigates its performance for instruction caches. The experimental analysis for an out-of-order processor with a 2-way eight instruction per block 16KB instruction cache show that

CATCH with a 2.14KB cost captures usually 60% or more of the cache-content-duplication idealized potential

6. Presents several optimizations to increase the CATCH's accuracy and/or cost-efficiency

Chapter 2, discusses previous work on cache redundancy. In Chapter 3, we discuss the CCD detection policies and techniques to increase CCD. Chapter 4 presents the simulation environment. Chapter 5, examines the limits of CCD for various types of instruction caches. Chapter 6 considers two possible applications of CCD and investigates their performance potential. In Chapter 7, we introduce Catch, and discuss different policies and optimizations to improve its performance and cost-efficiency. Chapter 8 evaluates the performance of the mechanism under realistic constraints when applied to an instruction cache. Chapter 9, provides conclusion and direction for future work.

Chapter 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 the most relevant of these papers are discussed below.

A scheme for main memory on-line compression was first proposed by Douglis [12]. The compression cache proposed should allow both software- and hardware-based compression using different compression algorithms. Benini et al. [4] proposed a dictionary based compression technique for instruction caches. The scheme does not require any modification of the processor since the instructions are decompressed outside the core and it always executes full-size instructions. Kjelso and Gooch [17] proposed a hardware implementation of the X-Match dictionary compression algorithm for main memory data. Lefurgy et al. [18] 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. [19] 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 an instruction dictionary. Due to the high replication of instructions and the small size of the dictionary, the amount of memory required to store the static code in main memory is significantly reduced.

The high redundancy of a limited subset of values in data caches was identified in [24]. A dictionary-based approach was proposed where a special cache is used to hold in compressed form the frequent values.

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 [20] 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.

Very relevant to our work is the paper by Sendag et al. [22] 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 investigate the limits of oracle address correlation, which is found to be significant, but do 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 [22] 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 [10], [5], [9]. 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 thesis 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. CCD is applicable to both instructions and data. Finally, it is possible, but beyond 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.

Chapter 3

Cache-Content-Duplication for Instruction Caches

CCD occurs when there is a miss in a cache and the content of the missed block is already in the cache but under a different tag. High level programming practices, low level assembly conventions and compiler transformations can lead to identical code sequences in different parts of an executable. Depending on run-time program behavior and the policies/methods used to detect duplication, such redundancy may result in CCD. 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 [21] 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 [21]. This prevents trace duplication with the same starting program counter and the same content, but with different direction of the last branch.

3.1 What is the cache content considered for duplication

One important parameter than can affect the frequency of CCD is the cache content that is considered for duplication. By CCD's definition this is an entire cache block. However, the contents of a cache block have a different meaning 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 discussion, in qualitative terms, it is expected that CCD will be more common for a basic block cache, less prevalent for a trace cache and even lower for an instruction cache.

One way to increase the frequency of CCD for regular a instruction caches, is to consider the duplication between *valid* instruction sequences send 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 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. Starting Section 8 the paper focuses on CCD for valid blocks in regular instruction caches.

3.2 When to learn the cache content

To detect duplication between cache blocks is useful to know the content of blocks in the cache. This way when a new block comes in the cache it can be detected if its content is duplicated 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 missed 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 know all the 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.

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, thus some CCD potential may be lost in the intervening time. An alternative method, that increases CCD frequency and is also efficient, 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 in combination with the heuristic that builds 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 in [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.

Chapter 4

Experimental Framework

The experiments in this paper were performed using benchmarks from the SPEC95, SPEC2000 and MiBench, version 1.0, 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 4.2 shows the dynamic instructions skipped and simulated for these benchmarks. Nevertheless, for some experiments (Section 8) the performance is reported for all the benchmarks that we were able to compile using the simplescalar compiler.

We assess and compare the performance impact of the different techniques using both functional (modified sim-fast [7]) and timing (modified sim-out-of-order [7]) simulators. The functional simulator is used for quick characterization of the design space in Sections 5 and 6. The timing simulator is used in all other cases to model an out-of-order processor with the configuration listed in Table 4.1. 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.

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 4.1. Out of Order Processor Configuration

benchmark	Skip (millions)	Simulate (millions)
gcc95	0	177
go95	0	133
perl95	0	40
vortex00	100	100
mesa00	350	100
basicmath	0	100
compress95	0	442
ijpeg95	0	129
li95	0	202
m88ksim95	0	241
perl95	0	40
vortex95	0	101
ammp00	50	100
art00	2000	100
bzip200	315	100
equake00	1300	100
gcc00	700	100
gzip00	300	100
mcf00	2000	100
parser00	400	100

Table 4.2. Benchmarks simulated

Chapter 5

Limits of Cache-Content-Duplication

In this Chapter we present the limits of CCD for an instruction cache, a basic-block cache and a trace cache. The various types of instruction caches represent potential application of the 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 duplicate-miss. For these experiments the cache operation remained unaffected by duplicated misses, i.e. the missed block is always fetched and inserted in the cache. For detecting and learning CCD we use the default policies presented in Chapter 3

5.1 CCD for Instruction Caches and Valid Blocks

Fig. 5.1 shows the breakdown of accesses into hits, duplicate misses and misses with a 2-way, 32B block instruction cache for various cache sizes. The graph also shows the CCD rate, secondary misses/total misses, using a label on each bar.

The results reveal that CCD is a rare phenomenon for fetch blocks, never more than 4% of the misses are for duplicated blocks. 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.

One possible way to overcome the above limitations, and increase CCD rate, is to consider the duplication in the valid block (the instruction sequence send down the pipe after a miss is serviced, instead

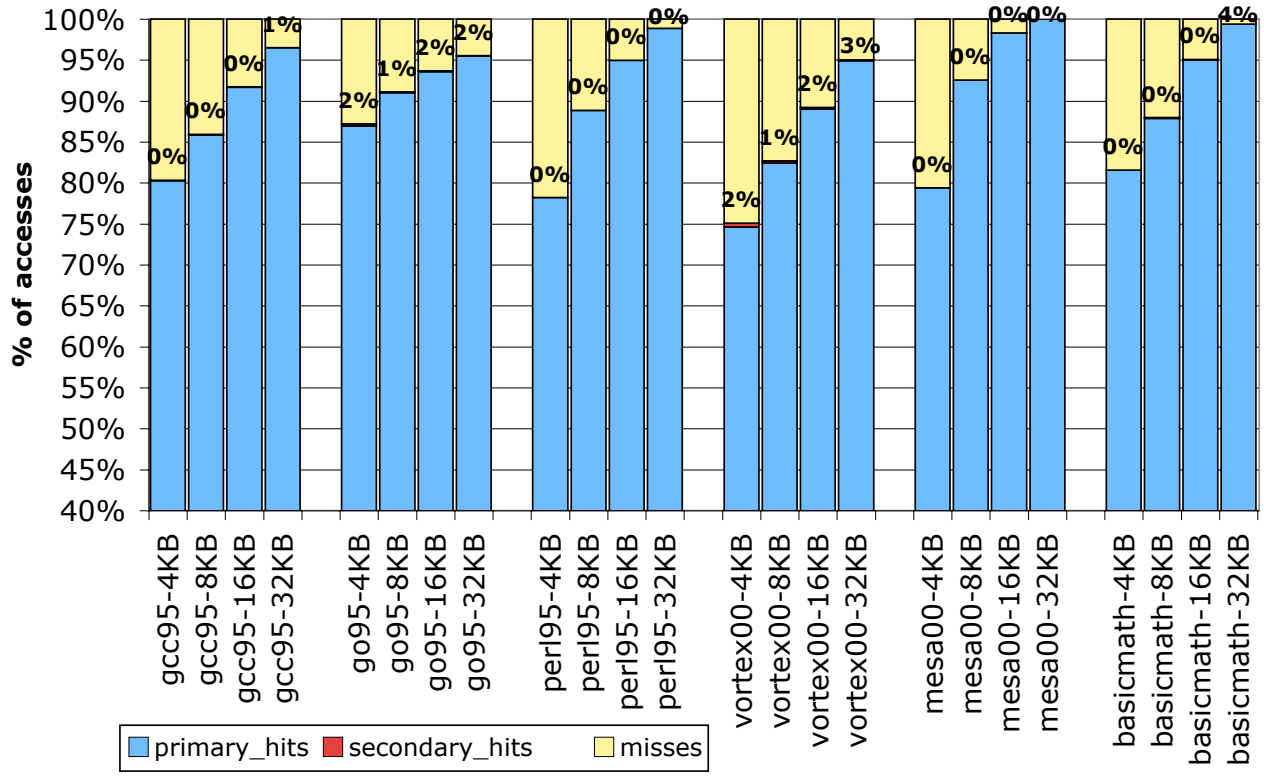

Figure 5.1. CCD for a 2-way, 8 instructions per block, instruction cache

of the duplication of all the instructions in missed block, Chapter 3)

Fig. 5.2 presents the CCD for valid blocks, specifically, it shows how often on a cache miss the required valid block is already in the cache. The data show that the CCD rates for valid blocks is often above 15% and therefore more prominent than for entire missed blocks (see Fig. 5.1). This increase is due to the following:(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 with increasing cache size is that 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 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 (Fig 5.3). 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 short distance conflict misses that can be removed using a victim cache [8]. In Chapter 8 we consider the combined performance of an instruction cache with a victim cache and the combination of an instruction cache with a victim cache and a CCD detection mechanism and the results shows that the two approaches, CATCH and victim cache, are

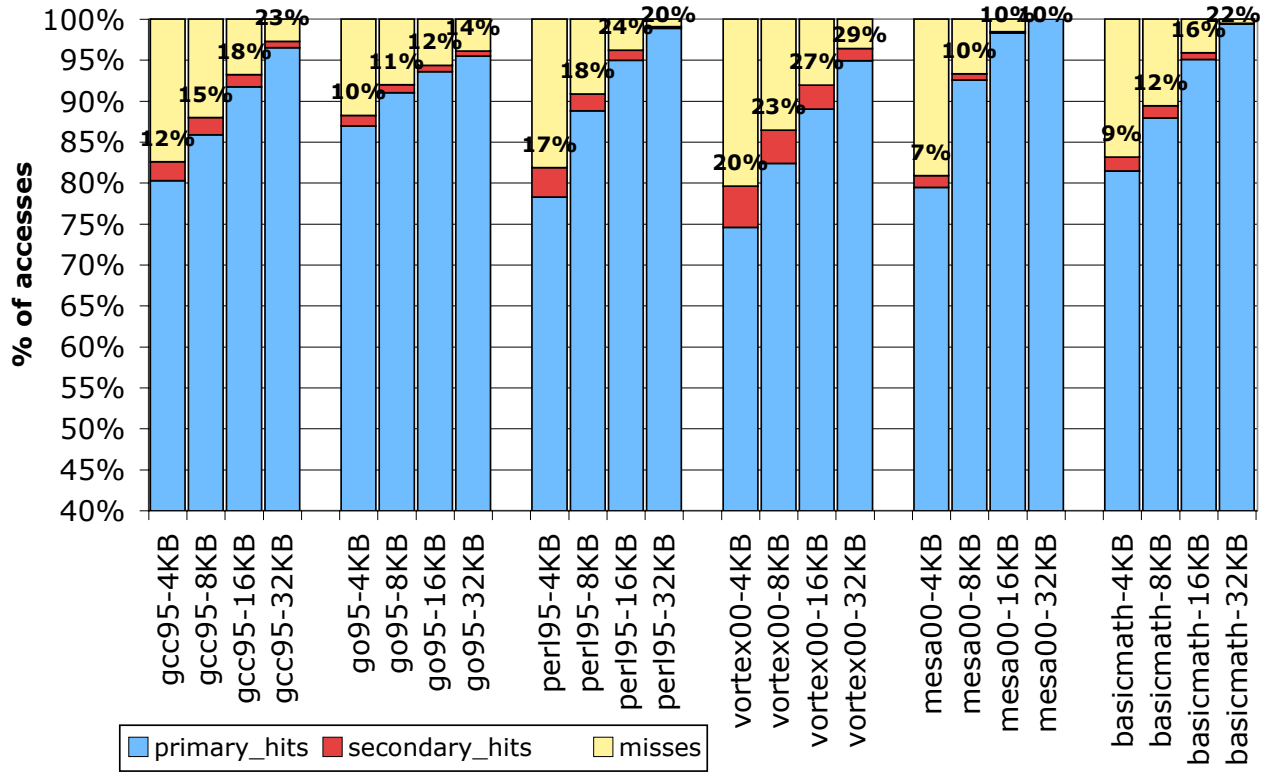

Figure 5.2. CCD for a 2-way, 8 instructions per block, instruction cache, for valid blocks

orthogonal. Henceforth, for instruction caches we only consider CCD for valid blocks which appears more promising.

5.2 CCD for Basic-Block Caches

Fig. 5.4 presents the breakdown of accesses for a 2-way 16B 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 for valid blocks (see Fig. 5.2). 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 on a miss we only fetch one basic block. Consequently, basic block caches have high miss rates (compare primary hits in Fig. 5.2 and Fig. 5.4) and thus more opportunity for missed blocks to be duplicates. In contrast, with an instruction cache on a miss we fetch the entire missed block which may contain one or more basic blocks. Therefore, valid and missed block contents are the same for a basic block cache whereas for an instruction cache the valid is smaller than the missed block.

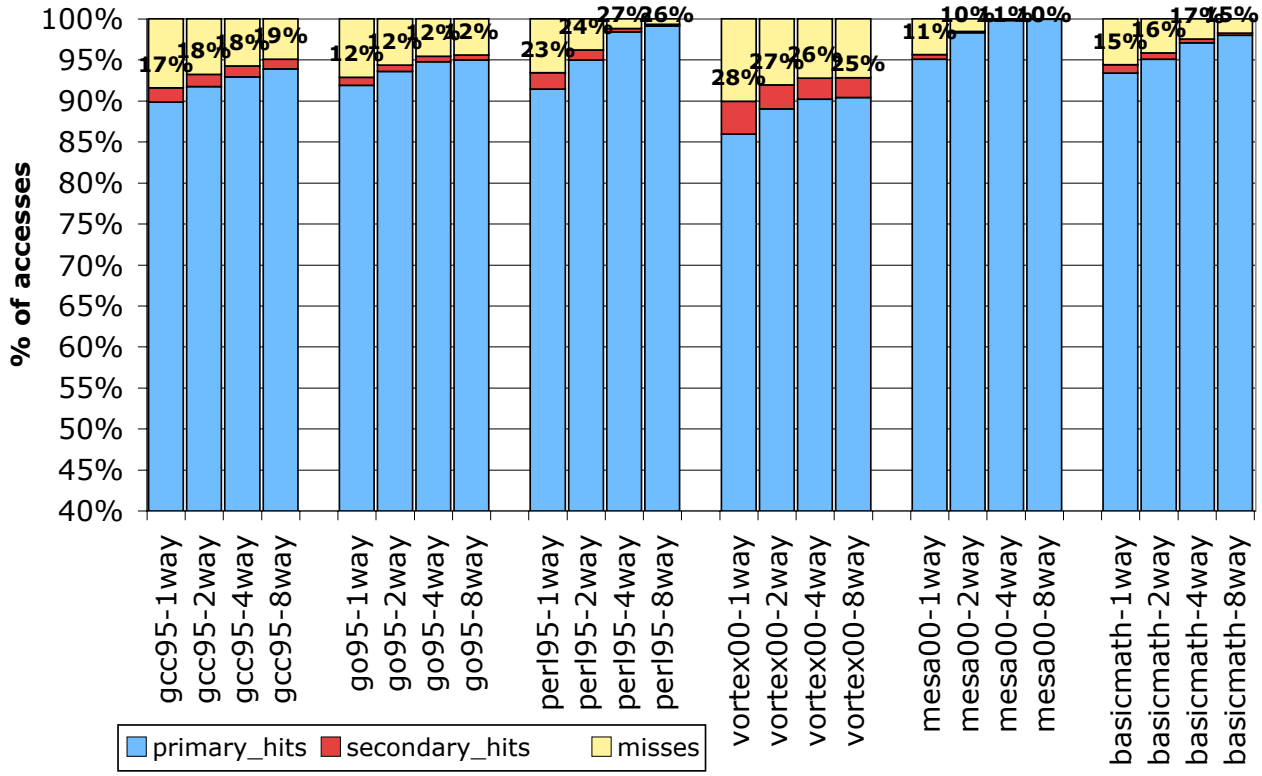

Figure 5.3. CCD for a 16KB, 8 instructions per block, instruction cache, for valid blocks

With bigger block size (Fig. 5.5) 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 meant usually 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 where each trace contains one or more dynamically consecutive basic blocks. Fig. 5.6 presents the breakdown of accesses for a 2-way set-associative trace cache with 32B (eight 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 (see Fig. 5.2) but lower

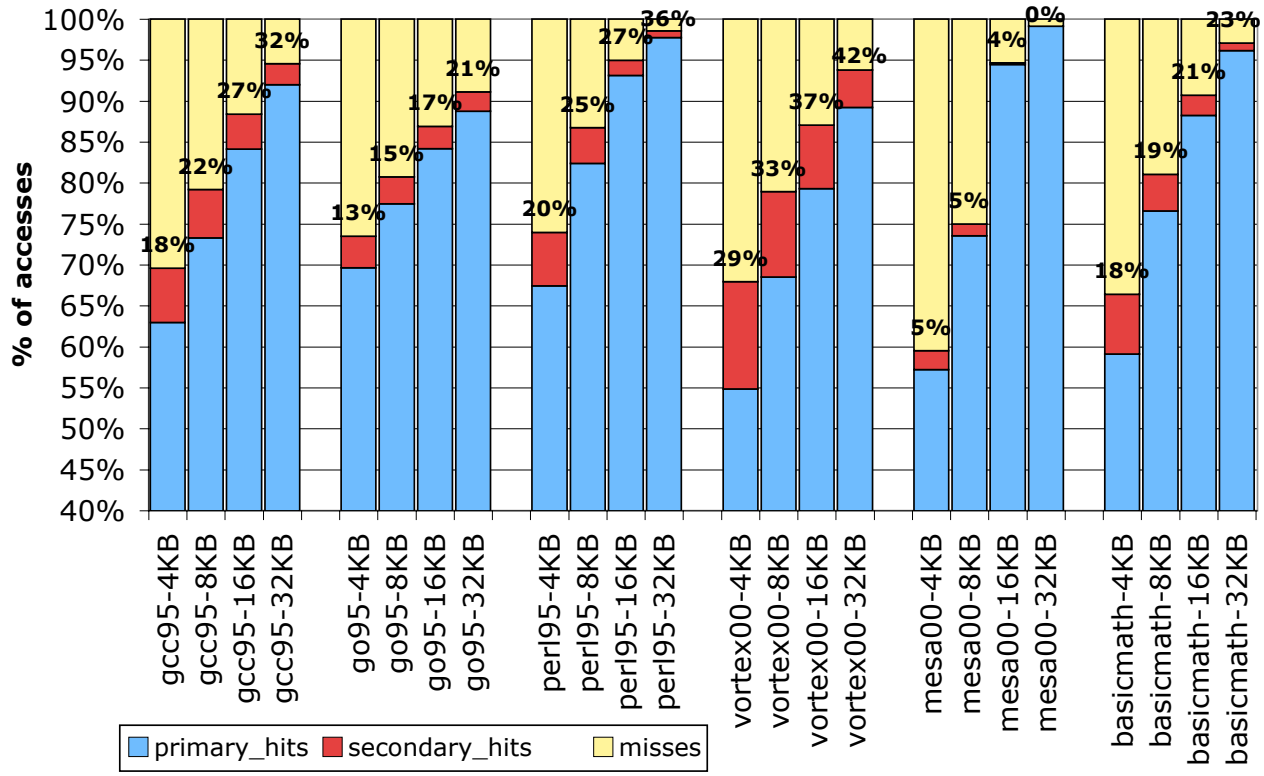

Figure 5.4. CCD for a 2-way, 4 instructions per block, basic block cache

than a basic-block cache (see Fig. 5.4). The CCD for traces with sixteen instructions (Fig. 5.7) is less, because 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.

Note that, similar to basic block caches, for trace-caches in most cases there is no distinction between a missed block and a valid block because on a cache accesses the entire trace line is send down the pipeline.

5.4 Overall Observations

Overall, the experimental results in this Chapter 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. Finally, CCD is more common with valid instruction cache blocks, basic-block caches and trace caches. We believe that the degree of the observed CCD provides a motivation to explore the performance and development of mechanisms that can exploit CCD.

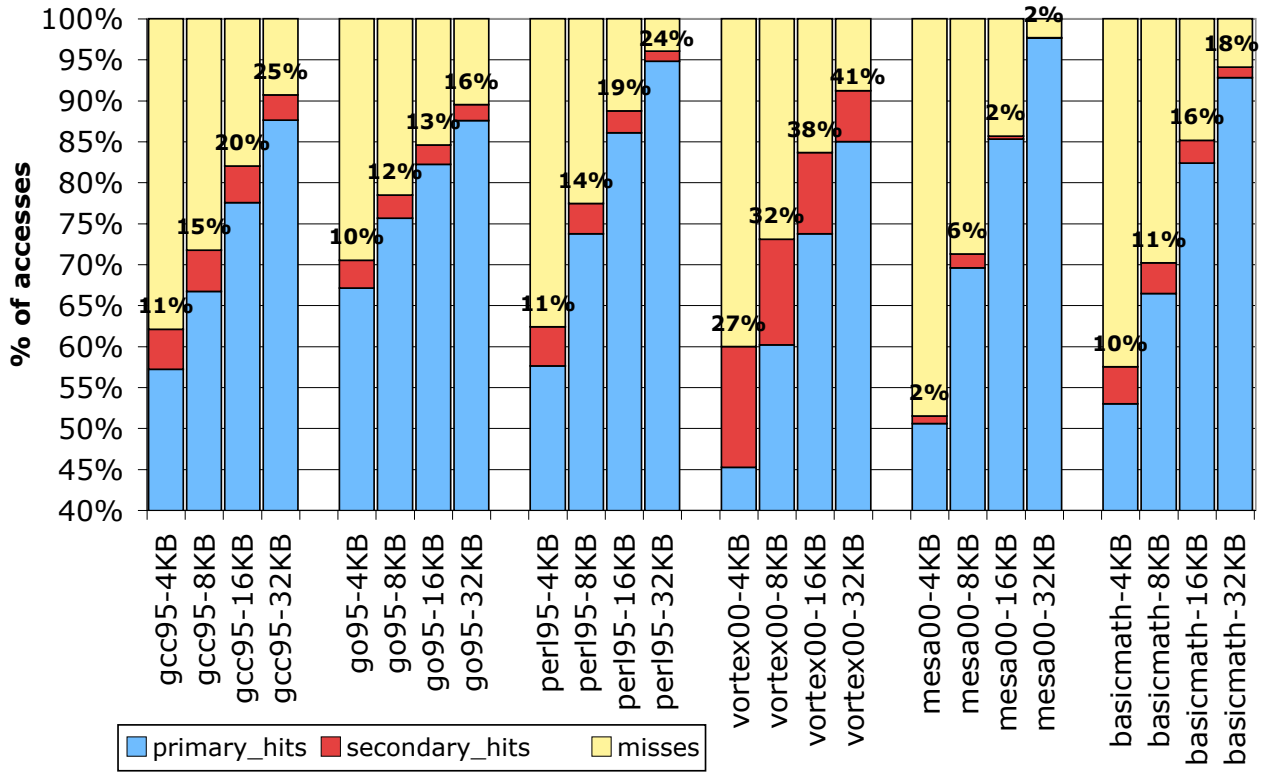

Figure 5.5. CCD for a 2-way, 8 instructions per block, basic block cache

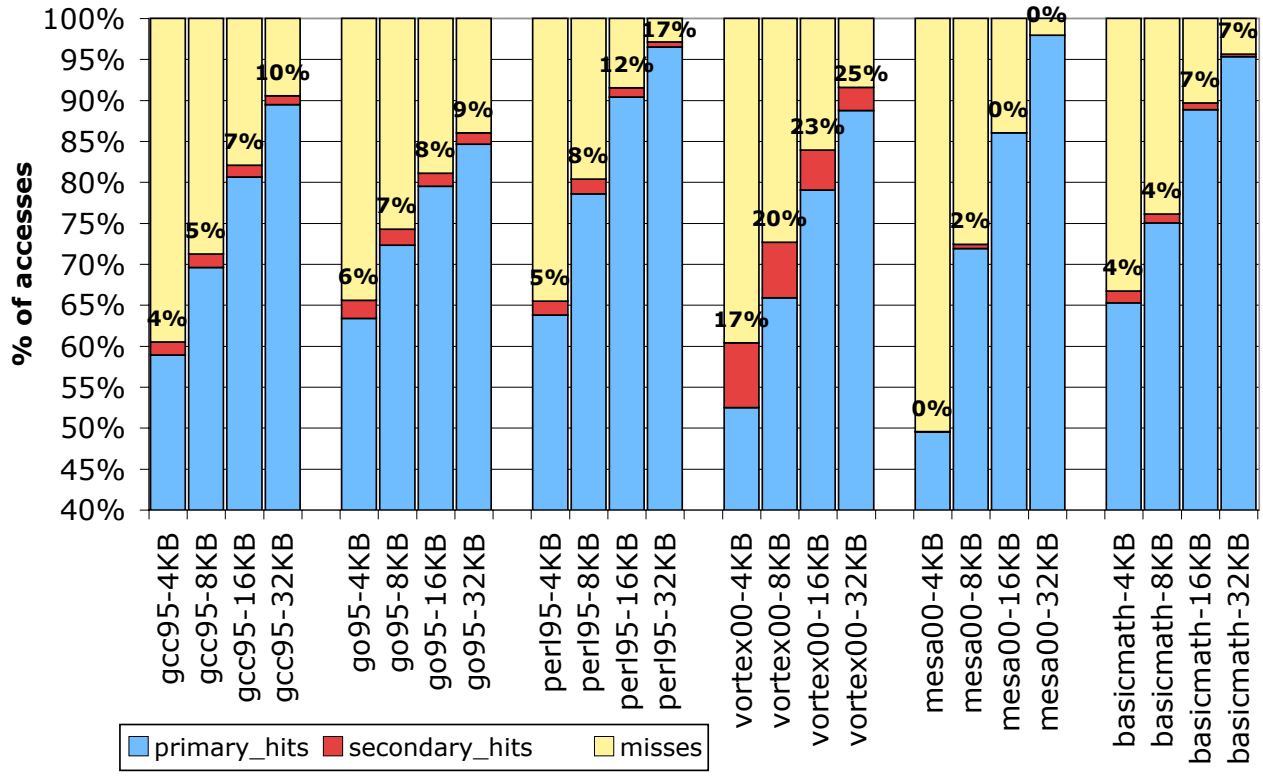

Figure 5.6. CCD for a 2-way, 8 instructions per block, trace cache

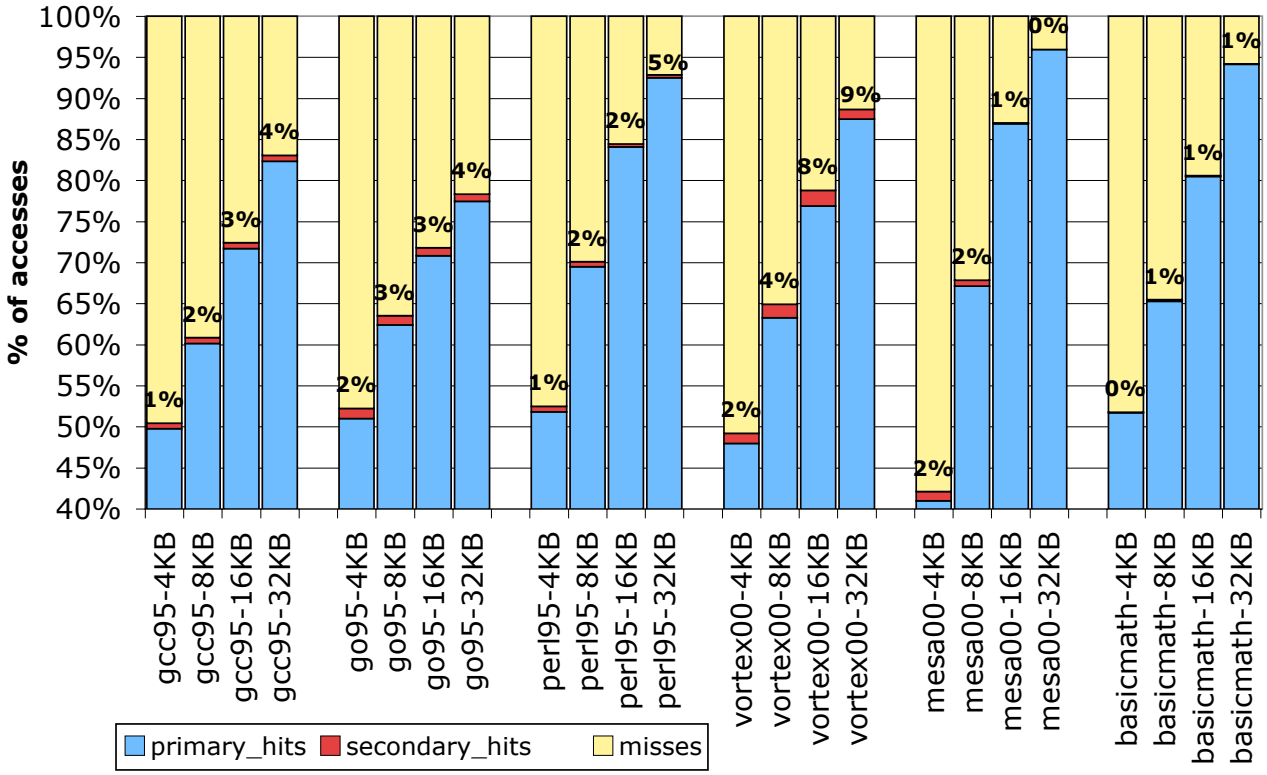

Figure 5.7. CCD for a 2-way, 16 instructions per block, trace cache

Chapter 6

CCD Applications: DAC and UCC

This Chapter describes two possible memory hierarchy enhancements based on CCD that can be useful for reducing 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 in the case of instruction cache, basic block cache and trace cache, because, in case of a duplicate, the content of the missed block can be read directly from the Level one cache. This benefit is lost in the case of CCD detection for valid blocks since we don't know the whole content of the missed block but just a part of it, the duplicate valid block, so the amount of improvement of a DAC for valid blocks 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 conflict 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.

Both DAC and UCC are applicable to instruction, basic-block and trace caches. However, a DAC for valid blocks has a subtle difference: a miss for a duplicated valid block requires the fetch of the missed block, from lower in the memory hierarchy, since the valid block is a subset of the missed block. This means that for valid blocks the benefit of DAC comes from executing the duplicated valid block while serving in parallel the miss to the original block.

More details about a possible implementation and policies of the DAC and UCC are discussed in Chapter 7. Below we investigate experimentally the performance limits of these two cache applications.

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. 5.2, 5.4 and 5.6 for the instruction cache valid blocks, basic-block cache and trace cache respectively.

To establish the potential of a **UCC** cache over a regular cache we performed the following study. A 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. 6.1 shows the breakdown of memory accesses for a UCC-instruction cache, Fig. 6.2 shows the breakdown of memory accesses for a UCC-instruction cache for valid blocks, Fig. 6.3 the breakdown for a UCC- basic-block cache, and Fig. 6.3 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, this is defined as miss reduction due to secondary hits, that do not correspond to baseline primary hits, over the baseline misses.

A comparison of Figs. 5.1–5.6 with Figs. 6.1–6.4, 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 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 positive CCD rate may not be sufficient to ensure that a UCC cache can improve the performance, because a decrease in primary hits can offset the benefits of CCD. Consequently, to establish a performance potential with UCC we need performance with a timing simulator for a specific pipeline.

6.2 Performance Potential of CCD

Fig. 6.5 presents the breakdown of Level one instruction cache (il1) accesses for the out-of-order processor. Benchmarks *gcc95*, *go95*, and *vortex00* appear to have the highest CCD rates (miss reduction due to secondary hits over baseline misses). Results show that, for DAC, primary hits are very close to baseline primary hits because DAC only affects the miss latencies due to secondary hits and not the hit-miss behavior of the cache. On the other hand, for UCC, that prevents duplicate block to enter the cache, a bigger portion of primary hits is converted to secondary hits.

Fig. 6.6 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. Data are presented for an instruction cache for valid blocks due to its wider use both in general and embedded computing systems. For the same reason the focus of the remaining report will be on instruction caches for valid blocks.

The results are 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 miss 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 miss 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 4.1. The duplication is detected ideally with the same assumptions as in Chapter 5.

The data show both DAC and UCC to have performance potential. The benchmarks *gcc95*, *perl95* and *vortex00* with the highest CCD rates, see Figs. 5.2 and 6.2, 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 miss 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 outweighted by the penalty for having some primary hits converted to secondary hits. Finally, although the limits of DAC and UCC are very similar as show in figures 5.2 and 6.2 the results in Fig. 6.6 show that UCC is much better in many configurations. This is because in the limit study of DAC we assume no latency for fetching a block from a lower level in the memory hierarchy. In 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 with a fetch latency it is possible to have secondary hit and the second access to cause a miss because the block is not yet available.

The performance potential analysis suggests that, for good performance and room for duplicated hit latency, the DAC with two or three cycle latency is a good compromise but for better performance with still some room for detecting CCD, UCC with at most one cycle latency is better.

A single cycle latency can be achieved in the case of a single cycle latency cache access by assuming that the mechanism and cache will be accessed in parallel. By the end of the cache access cycle the mechanism will provide an alternative tag-index to access the cache for a secondary hit. The single cycle latency will be charged in order to have a second cache access, a secondary hit.

A zero cycle latency is possible to be achieved, but may requires more pervasive changes, and is discussed extensively in Chapter 7, Section 7.8.

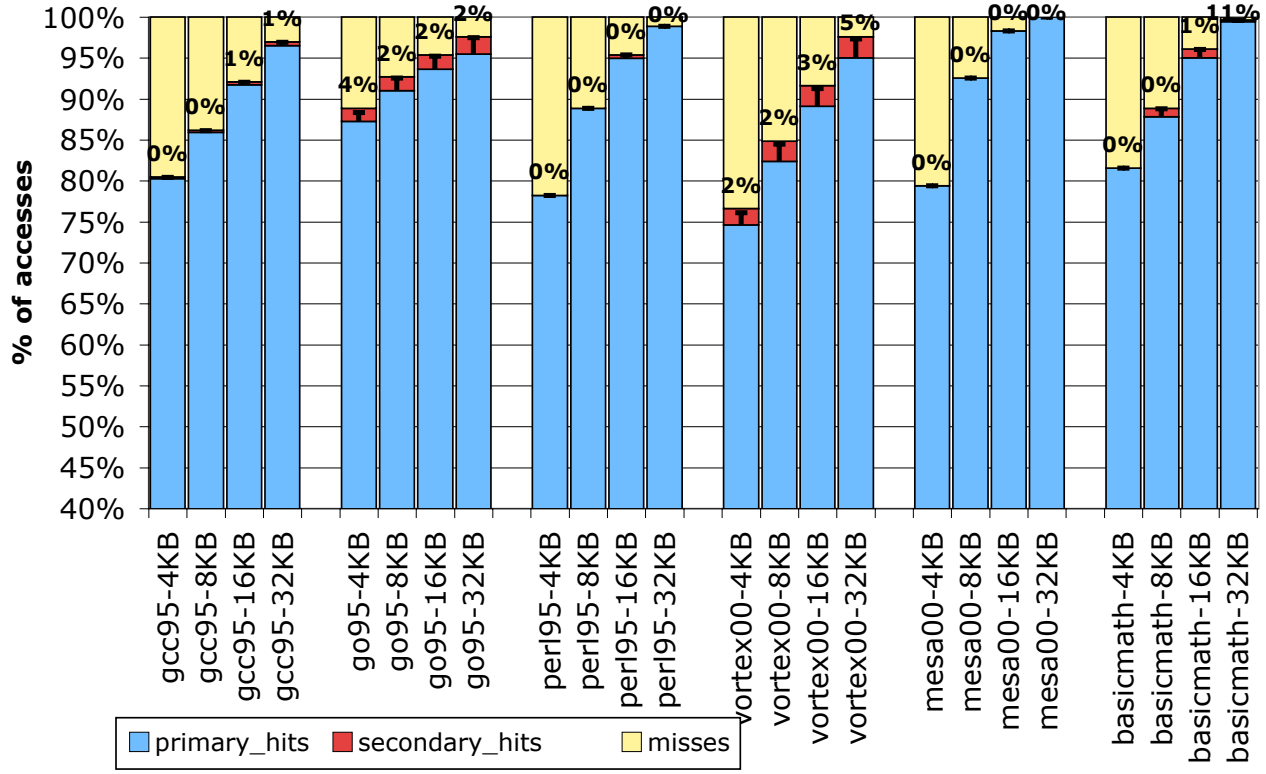

Figure 6.1. CCD for a 2-way, 8 instructions per block, instruction cache with UCC

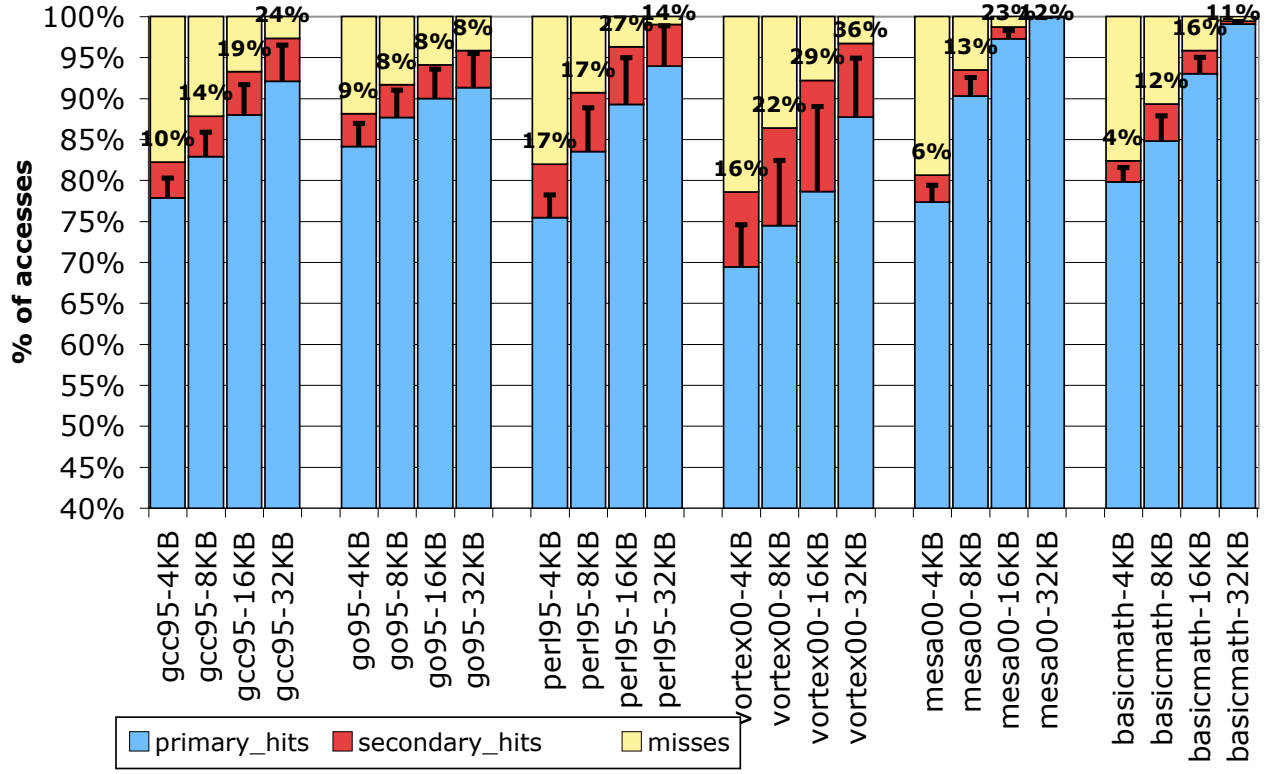

Figure 6.2. CCD for a 2-way, 8 instructions per block, instruction cache with UCC, for valid blocks

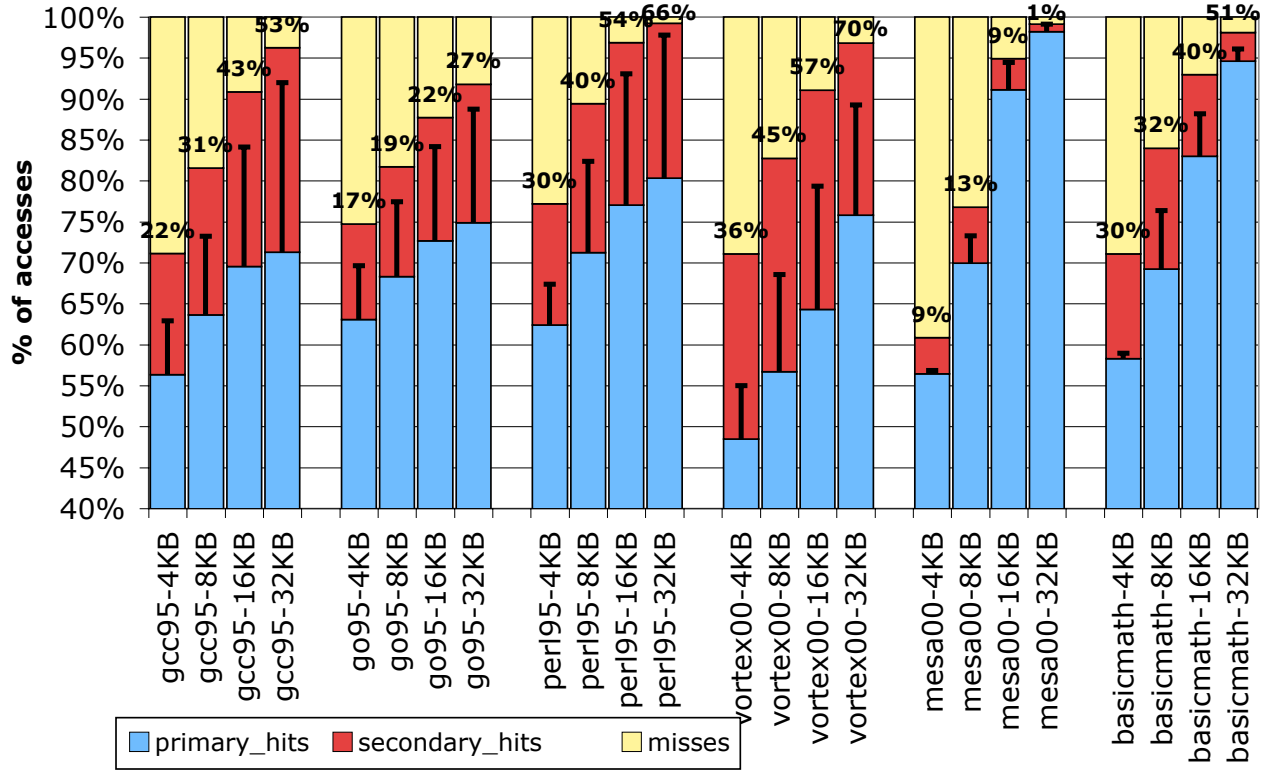

Figure 6.3. CCD for a 2-way, 4 instructions per block, basic block cache with UCC

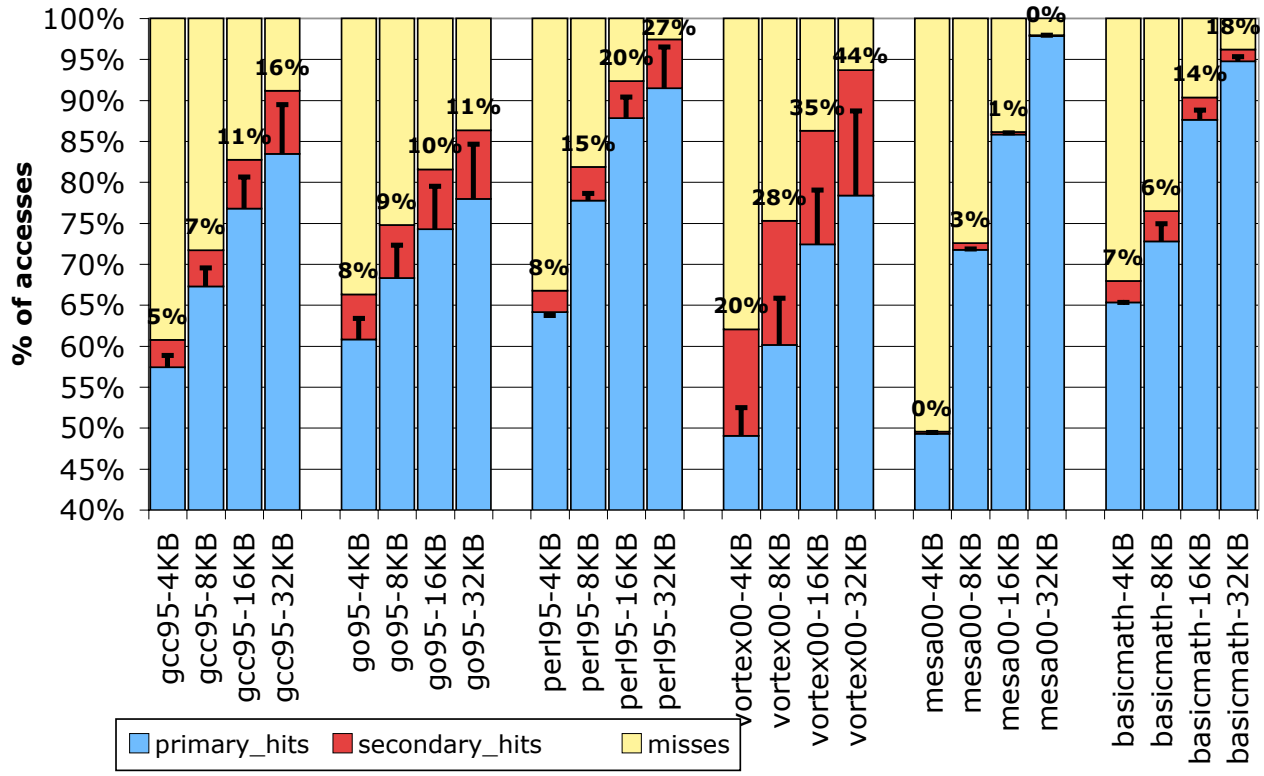

Figure 6.4. CCD for a 2-way, 4 instructions per block, trace cache with UCC

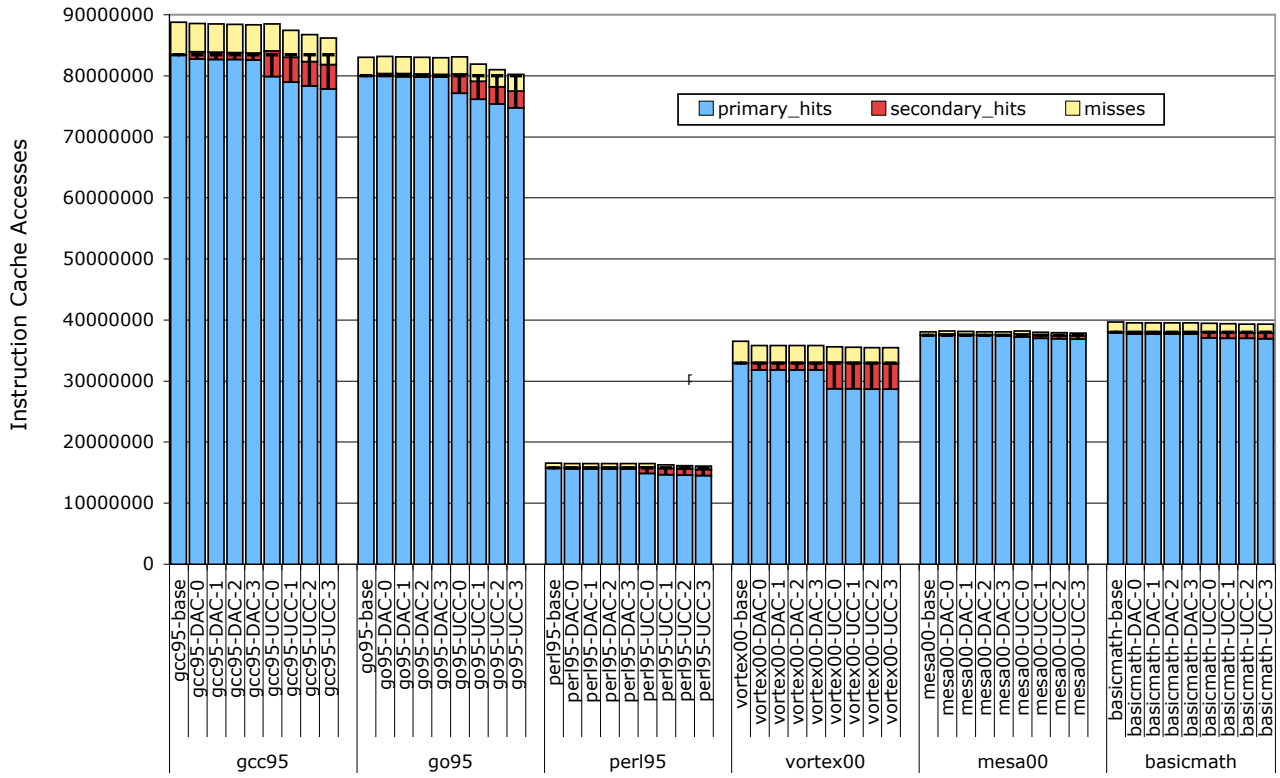

Figure 6.5. IL1 accesses distribution for 20 cycles L2 for a 2-way, 16KB, 8 instructions per block, instruction cache, for valid blocks latency

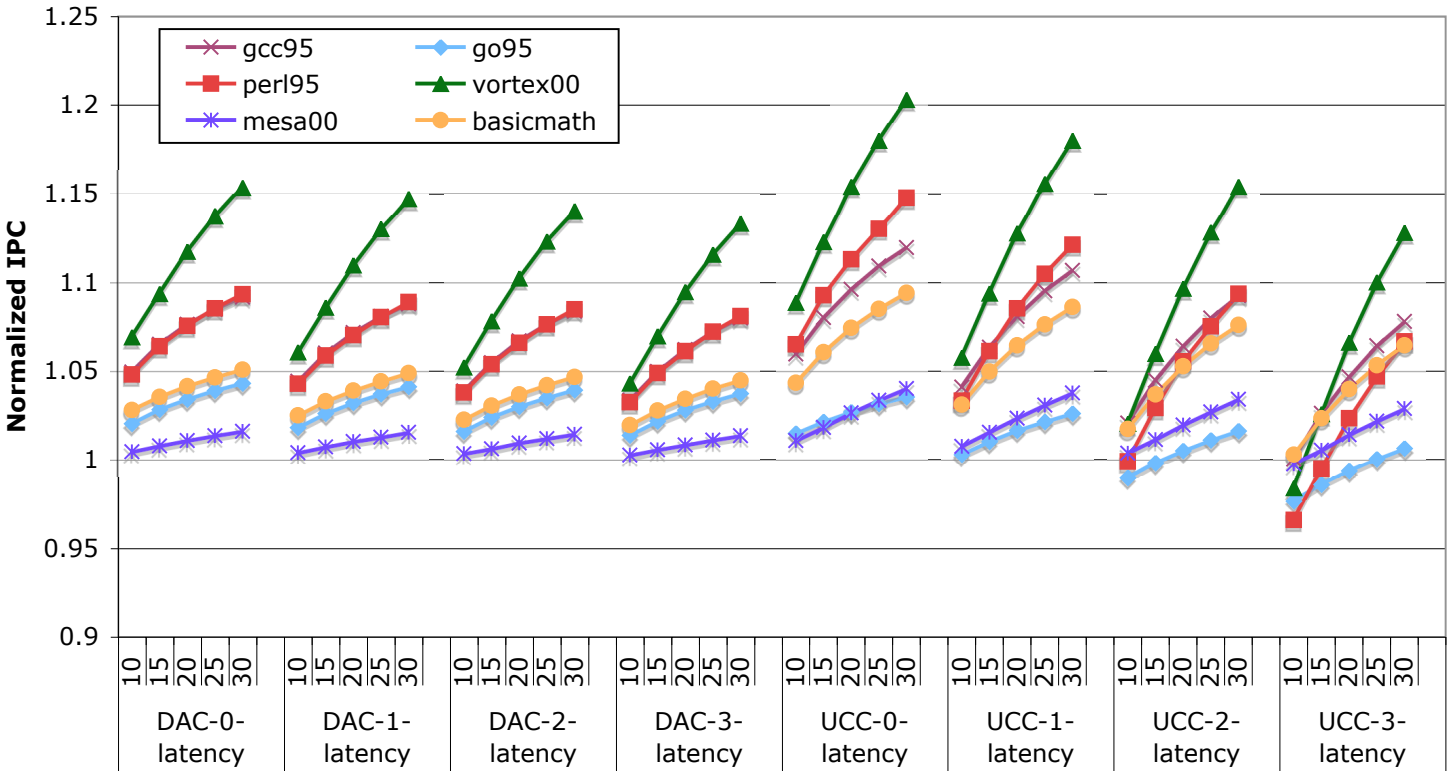

Figure 6.6. Performance potential of DAC and UCC for a 2-way, 16KB, 8 instructions per block, instruction cache, for valid blocks (Normalized IPC)

Chapter 7

CATCH: A method for dynamically detecting CCD

A practical dynamic implementation of a DAC or a UCC requires a hardware mechanism for detecting cache-content-duplication. 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 Chapter presents a method for dynamically detecting CCD. We will refer to this mechanism as CATCH.

Recall that for instruction caches duplication is detected for valid blocks (see Chapter 3) and thus a bit mask provided by the branch predictor, in addition to the starting PC, is used to identify a missed valid block.

The architecture of CATCH is shown in Fig. 7.1. 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 Chapter.

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 to be always correct (in the case of self-modifying code or page remapping the DR may need to be flashed to ensure correctness).

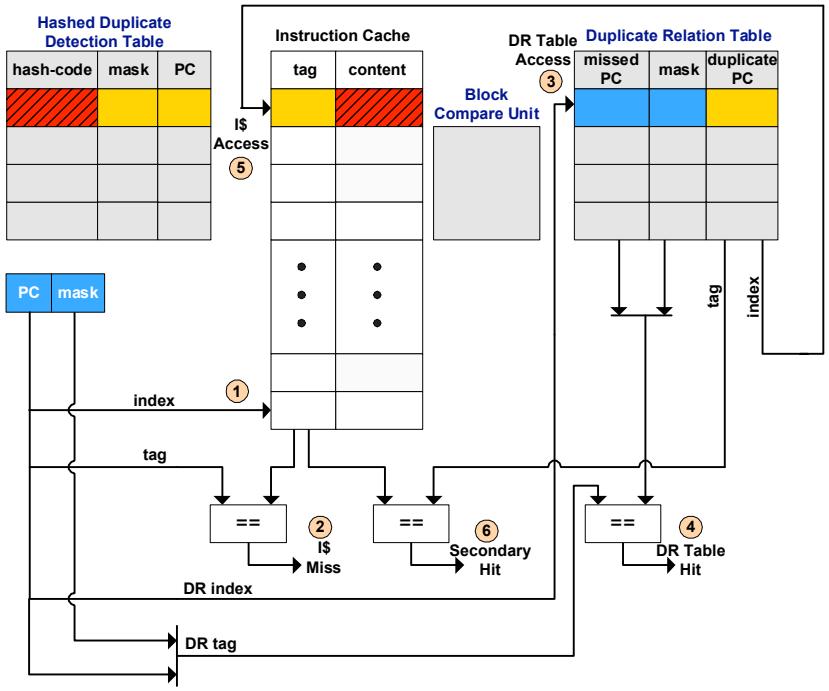

Figure 7.1. The CATCH and the flow for a Cache miss, DR hit, Cache hit

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 will add the extra penalty of 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 a need of flashing the DR each time we have a page remapping is necessary. Considering that a page remapping is a very rare phenomenon, during our experiments we decided to use a physically tagged DR with a physically tagged cache to avoid the overhead translation through the iTLB.

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 *complete missed block* will be initiated in parallel. For a UCC, the content of the duplicate-block will only be read and no miss will be requested from a lower level of the memory hierarchy. Fig. 7.1 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.

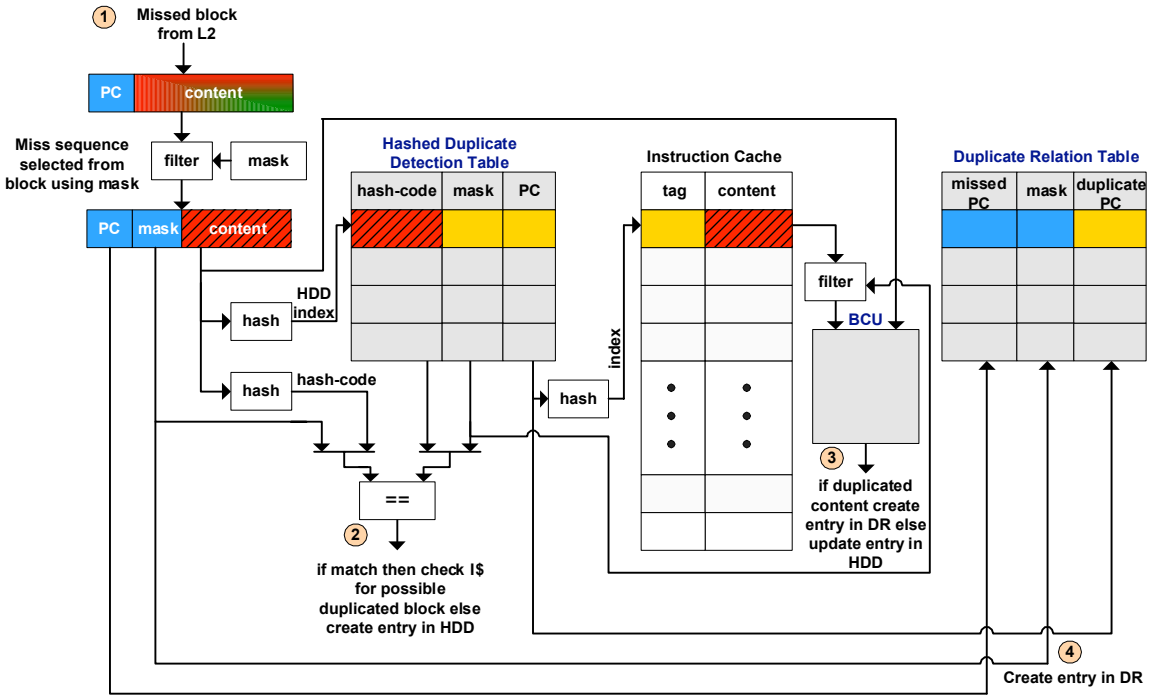

Figure 7.2. The flow with CATCH for a Cache miss, DR miss and HDD hit

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 into 32 bits provides very accurate encodings (often 99.9% accurate).

The HDD is indexed using a hash of the content of a missed block after 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 BCU. If the BCU indicates that the blocks are duplicates then an entry is created in the DR. Fig. 7.2 illustrates the sequence of steps in the case of a cache miss that has a duplicate in the cache but not an entry in DR. 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 are discussed in this Chapter, 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. The different scenarios for allocating an HDD entry are the following:

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 its content hash-code 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.

The different cases for updating an HDD entry and allocating or updating a DR entry are the following:

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 this Chapter, 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 we found it to occur very rarely for hash-codes with 16 to 32 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 misses for duplicated block and read the missed block directly from the cache as long as the block is in the cache. The key difference of CATCH for the DAC and the UCC is the following. 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 optimizations to improve the performance of CATCH. These optimizations can be divided into two general classes: Policy optimizations, aimed to improve performance through different update policies for the HDD and DR, and BCU optimizations, concerned with more advanced compare functions to facilitate more duplication. Below we describe two policy optimizations:

1. The keep_last optimization updates the HDD to point to the cache location where the most recent missed block is inserted. This is performed each time we have an HDD hit and the block

pointed by the HDD is a duplicate with the missed block. A performance comparison (data not shown) of the keep_last versus keep_old, indicates that the keep_last has better performance for a DAC but worse for a UCC and the difference for both cases is very small. Keep_last optimization keeps the logic simpler since we always have to update the HDD with the missed valid block.

2. The lru_update_on_duplicate_probe optimization updates the LRU state of a block in the cache each time we probe it. The cache is probed when there is a hit in the HDD or a hit in the DR. Updating the LRU indicates that it helps performance because useful blocks with duplicate content are kept for longer time in the cache. Also, we believe that updating on probe is the more natural choice since it requires no special handling by the cache.

The second class of performance optimizations is useful for increasing the potential of BCU to detect duplicate blocks that otherwise appear to be different. Below we mention such optimizations.

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 but at different program locations. 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.

Other examples of BCU optimizations, not evaluated in this work, is to consider more advanced compare function that could 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 default optimization configuration for all experimental results presented from the beginning assumes keep_last, keep_offset_in_dr and lru_update_on_duplicate_probe (for both DAC and UCC).

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 ISA, PISA, used in this work [7], a 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 28 bits for the address of the first instruction of the duplicate sequence. The non-optimized cost of a DR entry is therefore (2x28 - $\log_2$(number of sets of the DR)) + 2 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 to 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. For the ISA consider in this study this means 4x26=104 extra bits for each DR entry because the target is 26 bits and our fetch width is four instructions and that means it is possible to have four consecutive not taken branches. To reduce the number 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 each offset or target).

Therefore, for the DR the per-entry cost without cost optimization is (2x28 - $\log_2$(number of sets of the DR)) + 2 + 4x26 bits and with is (28 + 21 - $\log_2$(number of sets of the DR)) + 2 + 16 bits.

An HDD entry contains a hash-code and the PC and mask of the duplicate block. The hash-code assumed so far is 32 bits. This size of hash-code produced very rarely false-hash-matches and this is an indication that we can reduce the HDD cost without sacrificing performance. In Chapter 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 DR) bits. Finally the same criterion used in DR, do to not insert a relation in the DR unless the following are the same for the two tag-indices most significant 9 bits of the starting address, can be used also here. 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 32 + 28 + 2 - $\log_2$(number of sets of the DR) bits and with is 16 + 21 + 2 - $\log_2$(number of sets of the DR) bits. In Chapter 8, we compare the performance with and without the cost optimizations.

7.8 Pipelining Issues

To incorporate successfully a CATCH in a pipeline we have to consider pipeline 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 in a case of miss we can access the cache right away 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* which is used for control flow sequencing and the *fetch-PC* which 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 limited for duplication for the case of taken branches. In practice it is possible that duplication may start following a not taken branch or even after a non-taken branch. If desired to capture the CCD for these cases it may be necessary to update the BTB even with no-taken branches or non-branch instructions.

Another related issue are duplicated sequences that start after returning from a routine. A BTB, in general, is unable to predict the duplication after a return because a routine may be called from multiple sites. To predict a duplicated target for a return address the RAS must be modified to contain in each entry the actual return address and the duplicated. It is the responsibility of a call instructions to push in the RAS both the sequence and fetch-PCs in a RAS entry. Consequently, call instructions can have two duplicated-PCs associated with them. The duplicated-PC for the call target and the duplicate-PC for the continuation after the target.

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.

Chapter 8

Performance evaluation of CATCH

The results in the previous Chapters assumed oracle CCD detection. In this Chapter we evaluate the performance of the CATCH mechanism to detect CCD. First, we determine the performance of CATCH with large and fully-associative DR and HDD. Then, we introduce various constraints to the size, associativity and information per entry, to establish how much of the oracle performance it can be captured by successively more feasible to implement hardware configuration of CATCH.

8.1 CATCH performance for DAC and UCC Caches with large HDD and DR

Fig. 8.1 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. 6.6).

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.

One interesting observation from Fig. 8.1 is that sometimes the performance of CATCH is higher than the potential results of the limit study. 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

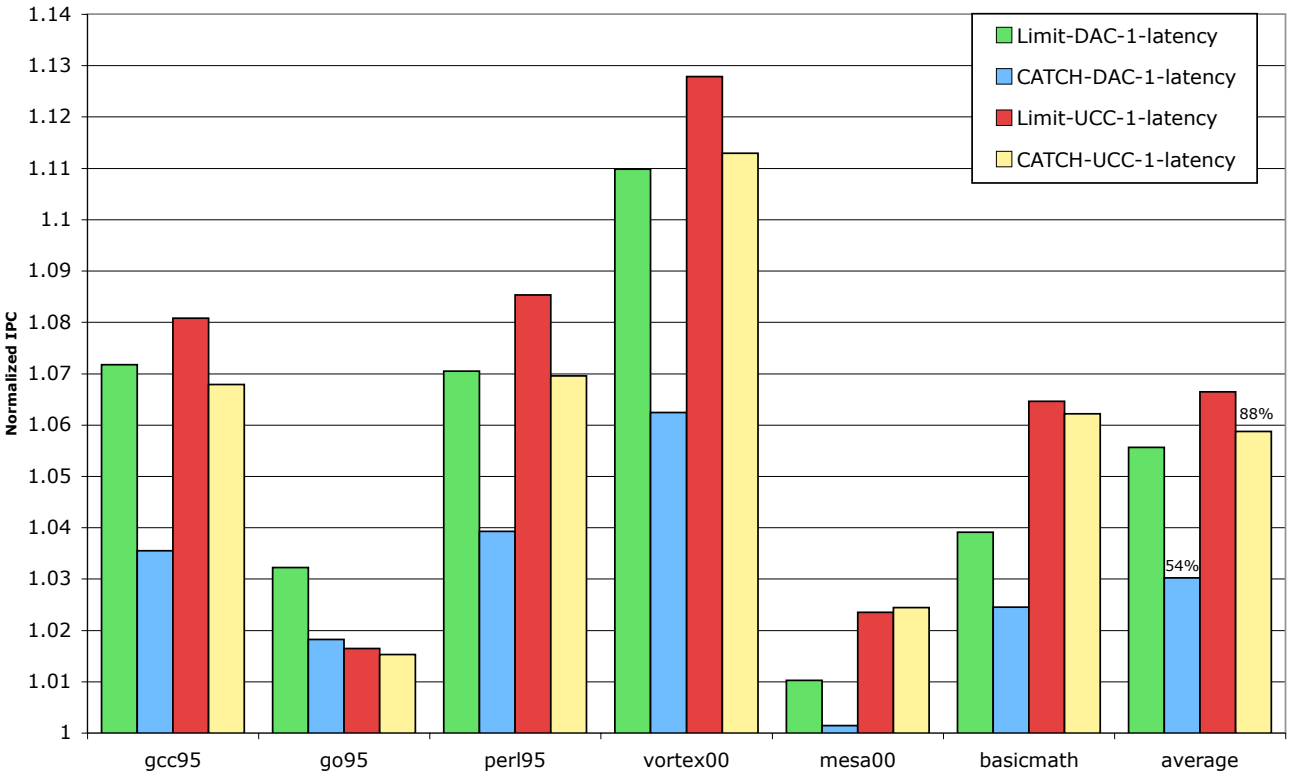

Figure 8.1. Performance potential of CATCH for DAC and UCC for 20 cycles L2 latency

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.

The low potential of DAC is due to the fact that in the oracle observation all the duplicates were detect even if this was the first time that the relation between two blocks appeared but for the real mechanism we already pay the penalty of L2 miss the first time the relation is created. As we stated in Chapter 6 the only gain of DAC is the miss latency savings. On the other hand, the UCC application, even using the real mechanism, will prevent duplicates to enter the cache from the first time the duplication is detected and by this way may not save miss latency on the first time, but might save conflict misses.

For the remaining of Chapter 8 we focus on optimizing the performance of a 16KB UCC instruction cache that is 2-way 32B per block, with zero cycle secondary hit latency and 20 cycles L2 cache miss latency. Finally we will give the performance of a single cycle secondary hit latency CATCH in combination with a victim cache.

8.2 CATCH performance for a UCC Cache with limited entries and associativity for HDD and DR

The results shown in the previous Section were using unbounded HDD and DR. This Section examines the effect on performance of fewer entries in HDD and DR and smaller associativity.

8.2.1 Smaller HDD and DR

Fig. 8.2 presents the normalized performance of CATCH for different HDD sizes, and an unbounded DR when the HDD is fully associative. The results are normalized with respect to IPC of a 16KB instruction cache without CCD detection. The results indicate that we can achieve the maximum potential of the mechanism with 512 HDD entries and an almost optimal performance with 128 HDD entries.

Fig. 8.3 presents the performance of CATCH with 128 entry HDD and different sizes of DR when both the HDD and DR are fully associative. The results are normalized to the IPC of a 16KB instruction cache without CCD detection. The data indicate that for most benchmarks a 512 entry DR achieves optimal performance, and that a 256 entry DR is a reasonable compromise.

To have a better view of the effects of DR and HDD size on performance, we run experiments with several combinations of DR and HDD sizes. The results showed that a very good compromise for all benchmarks is an HDD with 128 entries and a DR with 256 entries.

8.2.2 Reducing HDD and DR Associativity

The previous Section have concluded that with fully-associative structures a 128 entry HDD and a 256 entry DR provide performance close to oracle. This Section explores the effects of associativity on the performance of the HDD and DR.

Fig. 8.4 shows the IPC of CATCH over a regular instruction cache for different degrees of associativity in HDD and a fully associative DR. The data indicate that a 8-way set associative HDD provides in most cases performance close to a fully-associative HDD. Also, the performance of a 2-way HDD is comparable and, therefore, a 2-way HDD may be a good design choice.

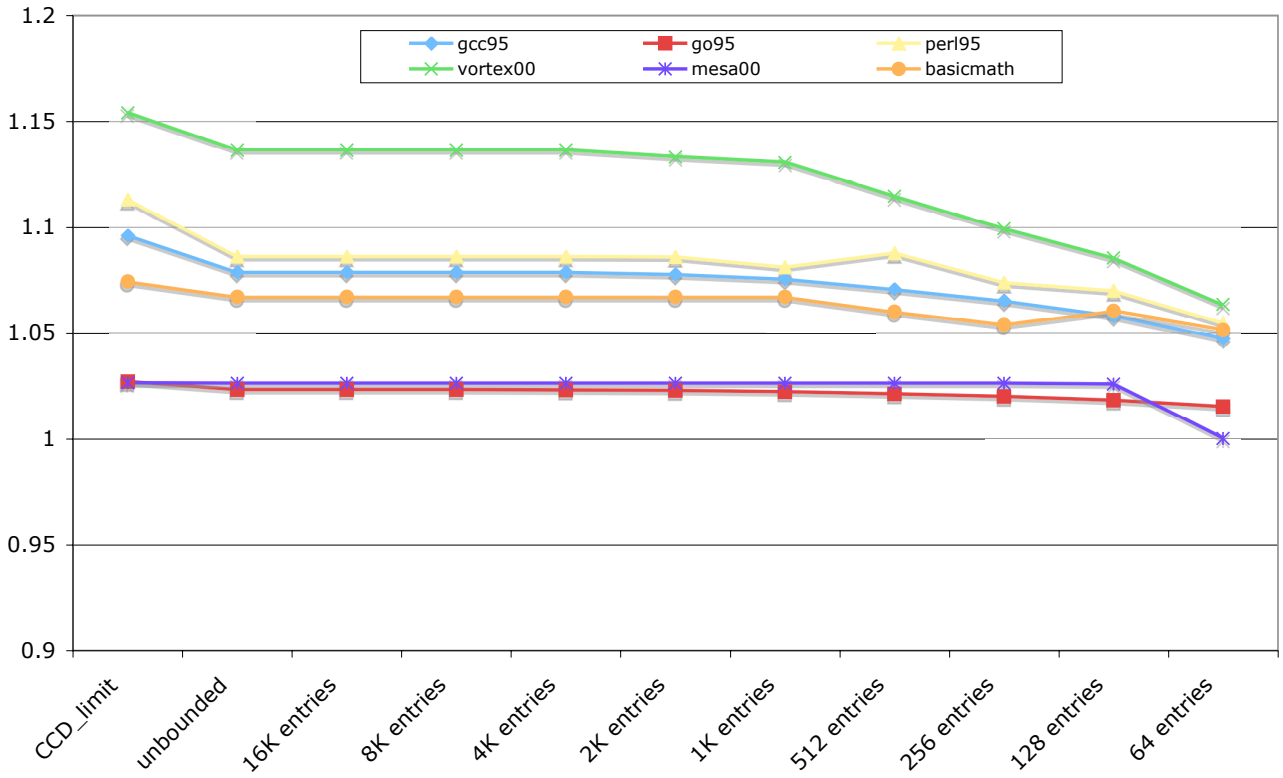

Figure 8.2. Performance potential of CATCH for a UCC using various sizes of HDD

Using a 2-way set associative HDD with 128 entries, we have measured the performance of DR with 256 entries but for different associativity. These results are shown in Fig. 8.5. It is evident from the data that either a 2-way or 4-way provides performance close to a fully associative DR, except for benchmark vortex00.

Overall, the analysis suggests that a 4-way 256 DR and a 2-way 128 HDD represent a good performing configuration. The data show that this configuration can provide an average IPC improvement of 4% which corresponds to 49% of the performance potential of a UCC with oracle CCD detection. Note that this CATCH configuration has 5.5KB cost.

8.3 The effects of cost optimizations and filtering

In order to reduce the total cost of CATCH we applied various cost optimizations discussed in Section 7.7. Fig. 8.6 shows the normalized IPC of an engineered version of CATCH with 2.14KB cost over a regular instruction cache. The results indicate that the cost optimizations reduce the performance insignificantly, almost 0.1%, but reduce the cost of the mechanism more than half.

Using a small 2-way associative HDD is difficult to maintain all the useful information about the

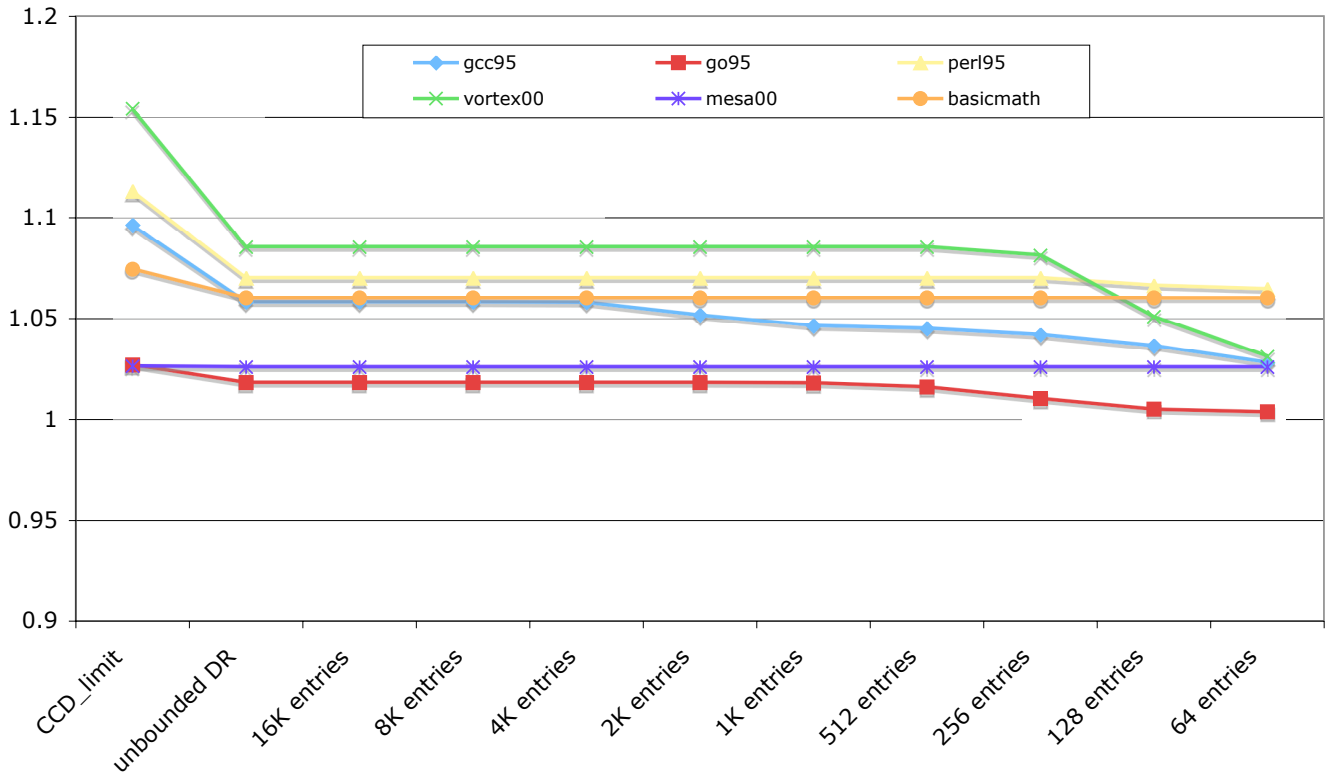

Figure 8.3. Performance potential of CATCH for a UCC with 128 entries in HDD and various sizes of DR

cache content. In order to increase the performance we apply a simple filtering technique [3] that allows a fraction of potential HDD entries to be created. Fig. 8.7 present the results of the HDD filtering, with value 4, using normalized IPC over a scheme without filtering. Filtering 4 means that one of every four potential HDD entries will be actually created in the HDD.

The data indicate that the performance improvement for some benchmarks is significant, for example mesa00, while for others may get marginally worse depending on the filtering values, for example basicmath. We came to the conclusion that a small filtering value of four, is beneficial for the HDD. Filtering four found to be the best compromise for this benchmark set and improves the performance by 20%.

Note that on this point, with cost optimizations and filtering, the cost of CATCH is 2.14KB and can capture on average 67% of the potential of oracle CCD detection assuming a zero cycle secondary hit latency.

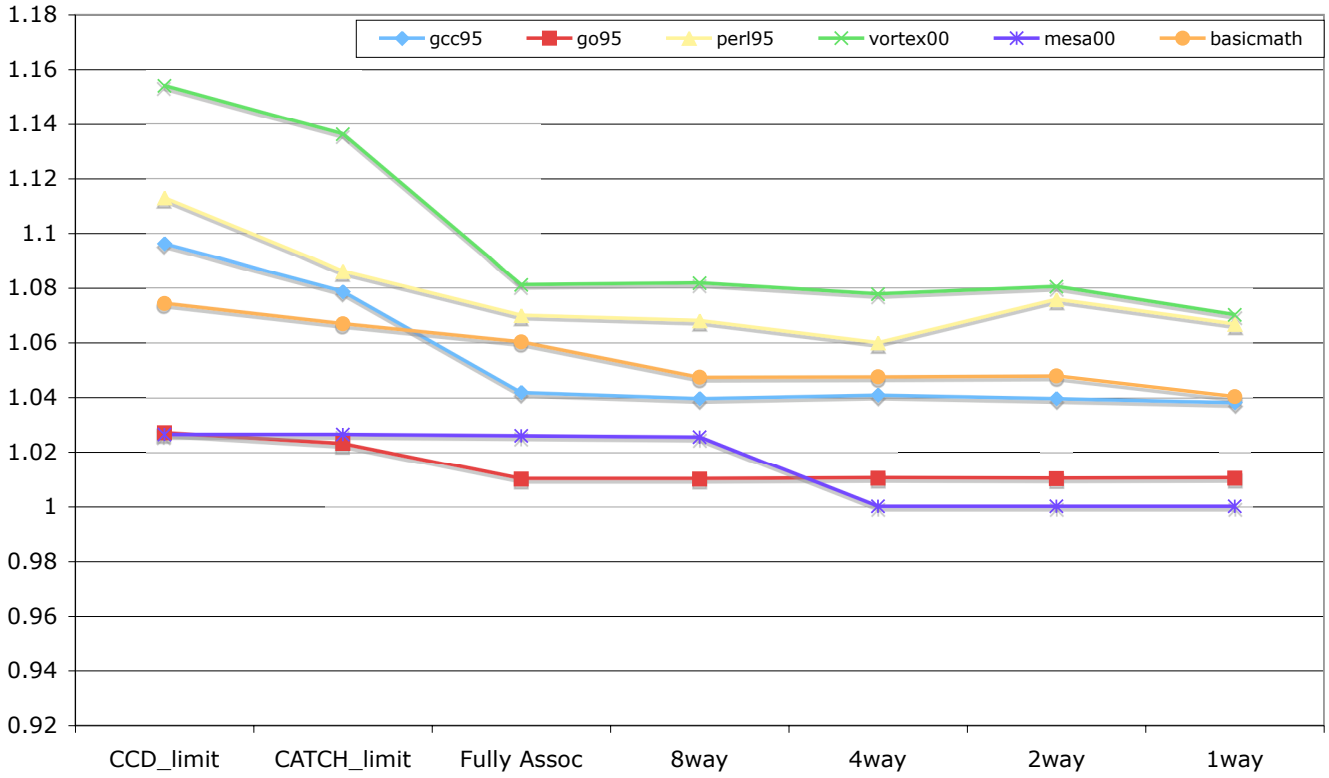

Figure 8.4. Effects of associativity on the HDD with a fully associative DR (128 entries HDD, 256 entries DR)

8.4 The significance of the various optimizations

The results shown so far assumed the performance optimizations introduced in Section 7.6 and zero cycle latency for CATCH. In this Section we discuss the performance implications of various optimizations for a UCC instruction cache based on CATCH with a 4-way 256 entry DR and a 2-way 128 entry HDD with a single cycle latency for CATCH.

Fig. 8.8 shows, for the complete benchmark set considered in this study, the performance for (a) CCD limit study, (b) CATCH with unbounded size, (c) CATCH with realistic size and all performance optimizations, (d) CATCH with realistic size and without any performance optimizations and (e) CATCH with realistic size and all performance and cost optimizations. The average results are only for benchmarks that have potential for improvements. These are the benchmarks between gcc95 and basicmath on the x-axis of Fig. 8.8. The average results are labeled with a percentage that represents the amount of performance potential captured. The other benchmarks shown in Fig. 8.8 have very few misses with a 16KB instruction cache and are mainly shown to demonstrate that CATCH is not detrimental to them.

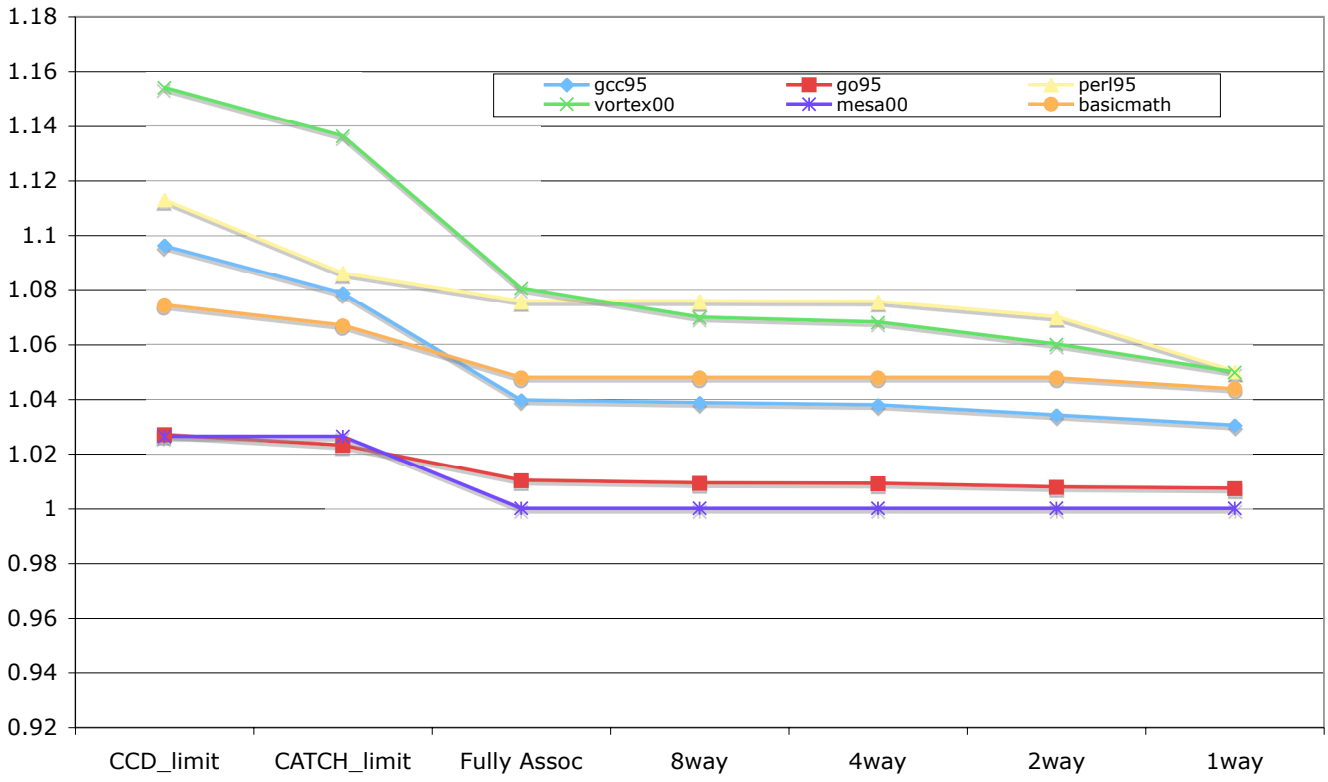

Figure 8.5. Effects of associativity on the DR with a 2-way HDD (128 entries HDD, 256 entries DR)

The no-optimization represents a configuration with no optimizations, and on the average achieves 22% of the performance limit.

Finally, the configuration that combines all performance and cost optimizations provides 60% of the potential. This configuration has only slightly lower performance as compared to the configuration with no cost optimizations (60% vs 61%).

The state cost for optimizations, no_optimizations and cost configurations, obtained based on the discussion in Section 7.7, is 5.5KB, 2.22KB and 2.14KB respectively. This indicates that the 2.14KB option represents a cost-efficient implementation of the CATCH with good performance.

8.5 CATCH vs Victim cache

An alternative mechanism to reduce cache misses is the victim cache [16]. A victim cache targets 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 8.9 show the performance improvement of regular cache using an 8 entry victim cache, using CATCH of 2.14KB cost and finally a combination of these two. In the combination model the victim cache is accessed first

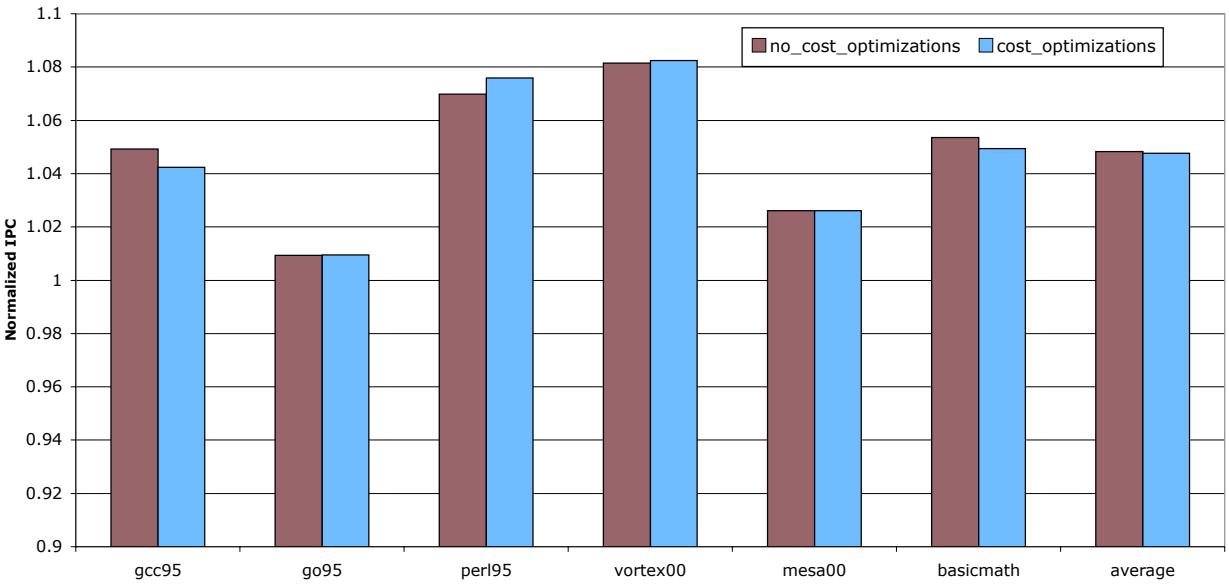

Figure 8.6. Effects of cost optimizations on CATCH

and only in a case of a miss the secondary tag-index, if exists, provided by the CATCH is used for a secondary cache access.

The data show that in some benchmarks,gcc95, vortex00 and basicmath, CATCH is better than victim cache while for the others victim cache is better. The 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 only conflict misses in a set but also exploits another behavior of the programs, the cache content duplication, that appears to happen across sets.

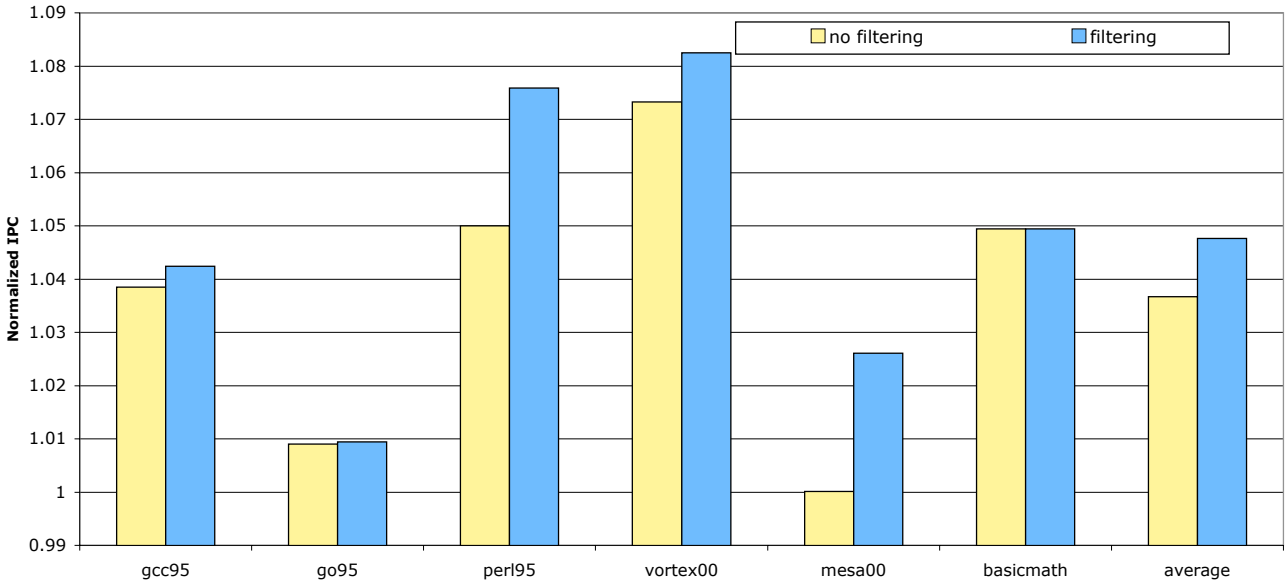

Figure 8.7. Effects of HDD filtering on CATCH

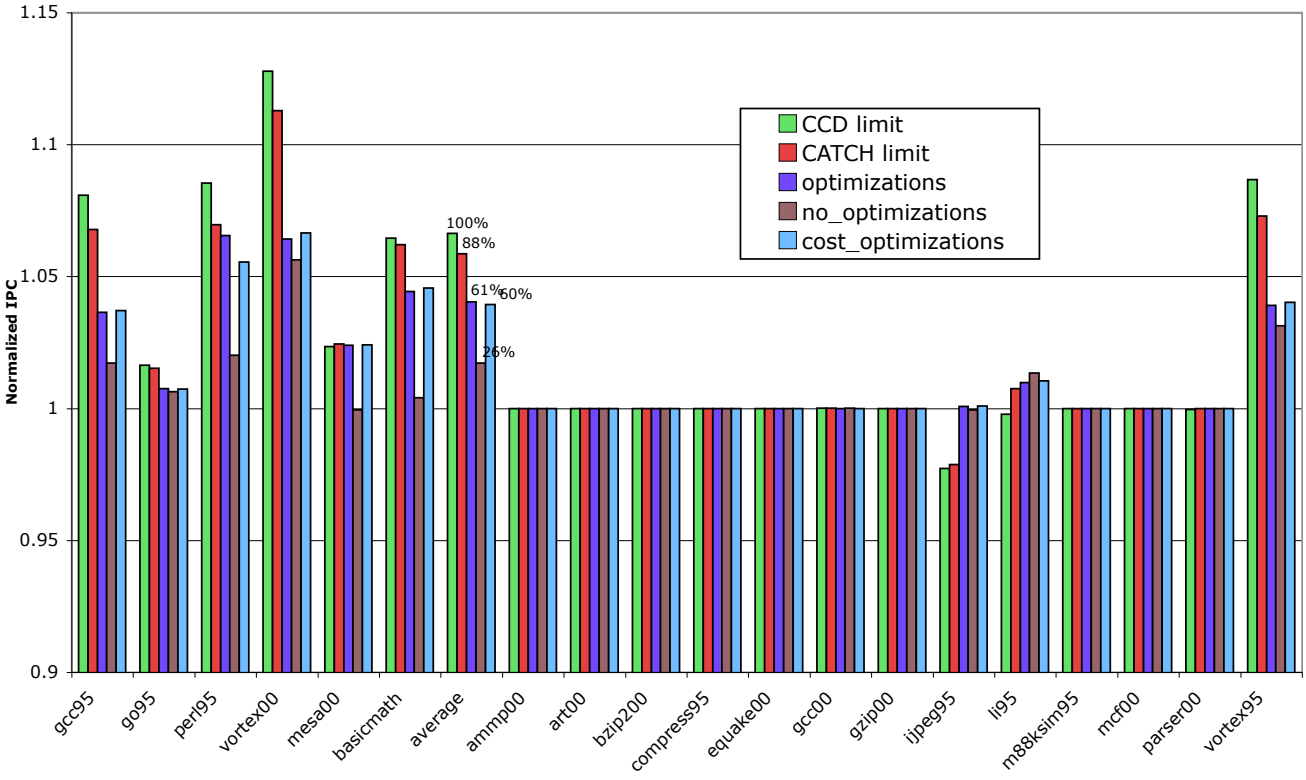

Figure 8.8. Effects of performance optimizations (4-way 256 entries DR, 2-way 128 entries HDD)

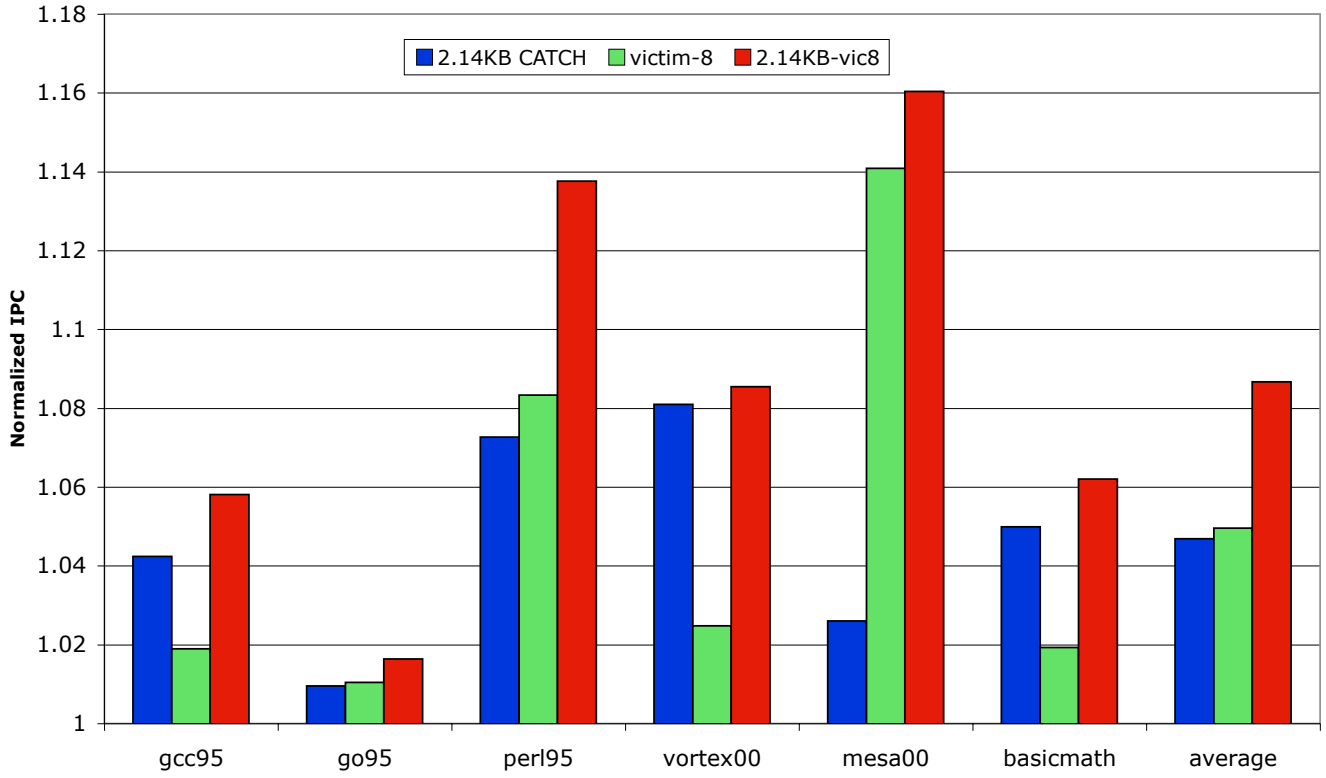

Figure 8.9. CATCH compared to an 8 entry victim cache

Chapter 9

Conclusions and Future Work

This thesis introduces the notion of cache-content- duplication and proposes CATCH, a mechanism for dynamically detecting cache-content-duplication. The thesis evaluates the performance of CATCH for two cache architectures that exploit cache-content-duplication: the duplicate-aware-cache and the unique-content-cache.

The thesis 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 2.14KB cost captures usually 60% or more of the cache-content-duplication idealized potential.

Experimental results comparing CATCH with victim cache show that CATCH can capture misses not only due to conflicts in the same set as victim cache is able to do. The performance gain of the two mechanism is additive, indicating that CATCH can also be used in a processor with victim cache.

The thesis points to several direction of future work. One is to investigate other policy optimizations that can lead to better CCD behavior. One area of work is to investigate the potential of more advanced compare functions for the BCU that consider the semantics of individual instructions and the dependences between instructions. The mechanism for zero cycle duplicated hit latency must also be evaluated. One other important direction of research is to consider CCD for multicores, and a more thorough investigation and tuning for data caches and for different levels in the memory hierarchy. CCD may also be considered in combination with static code compaction with the help of compiler techniques. CCD must also be evaluated with a bigger set of benchmarks, that consists of DBMS applications and other applications with large footprint. Finally, design and power complexity issues of CATCH need to be investigated.

Appendix A

Additional Results

In order to better understand the behavior of CCD for valid sequences we have made one more experiment. Figure A.1 shows the distribution of duplicate valid instruction sequences during the execution of all benchmarks and also the average. The results show that usually the single instruction sequences are dominant but there are some cases like ijpeg95 where 75% of all duplicate sequences contain four instructions. The average shows that 45% of all duplicates during the execution of the whole benchmark suite are for a single instruction valid sequences. The data suggest that further optimizations can be consider to improve the performance by exploiting such behaviors. For example, a filter can be applied to remove the single instruction valid sequences in order to reduce the cost of the mechanism if they are found to be bad for performance.

Furthermore, during the design exploration we considered different algorithms for learning the sequences as mentioned in Section 3.2. The results of the various algorithms considered are shown in figure A.2. The results show that learn_all_on_miss is slightly worse than the learn_on_miss_hit (the algorithms of these policies are described in Section 3.2). Considering the complexity of the learn_on_miss_hit policy we decided that is a good trade-off to use the learn_all_on_miss policy and avoid all the complexity but loose some performance.

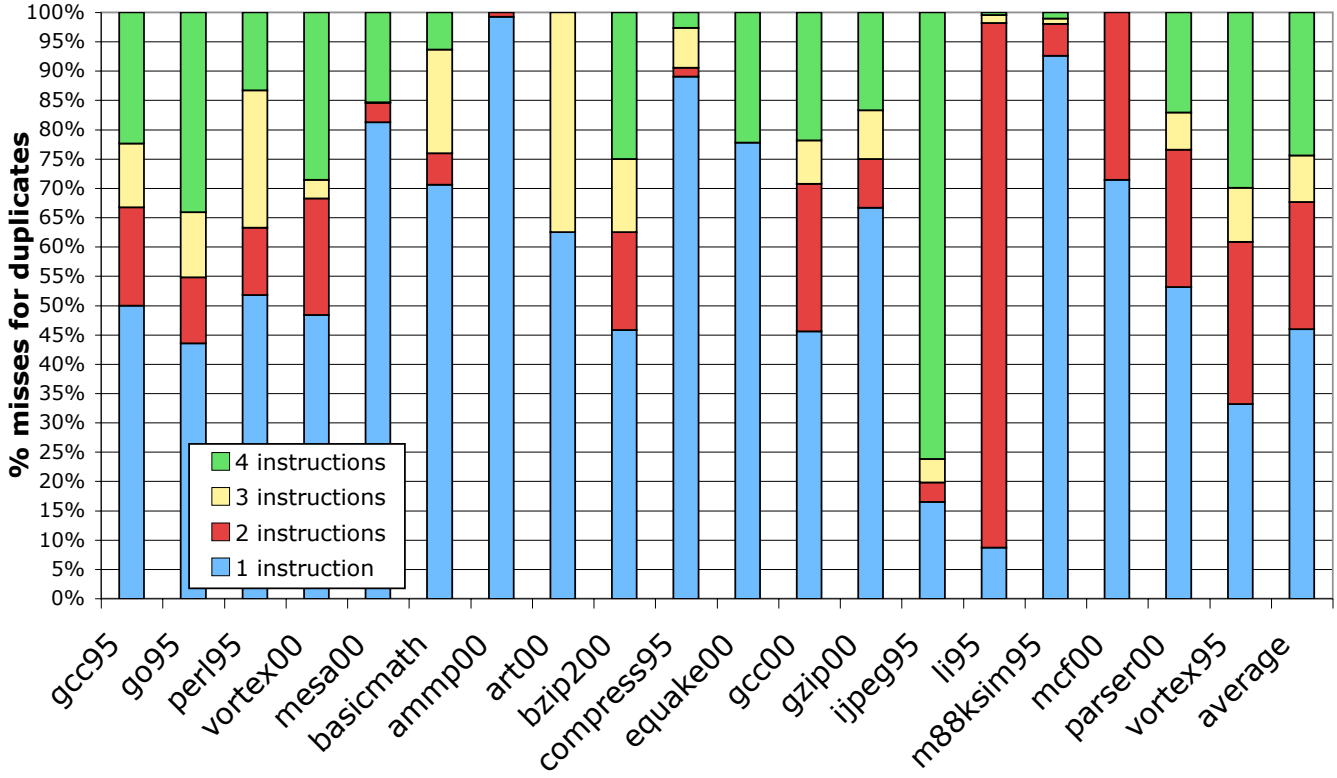

Figure A.1. Breakdown of duplicate valid instruction sequences

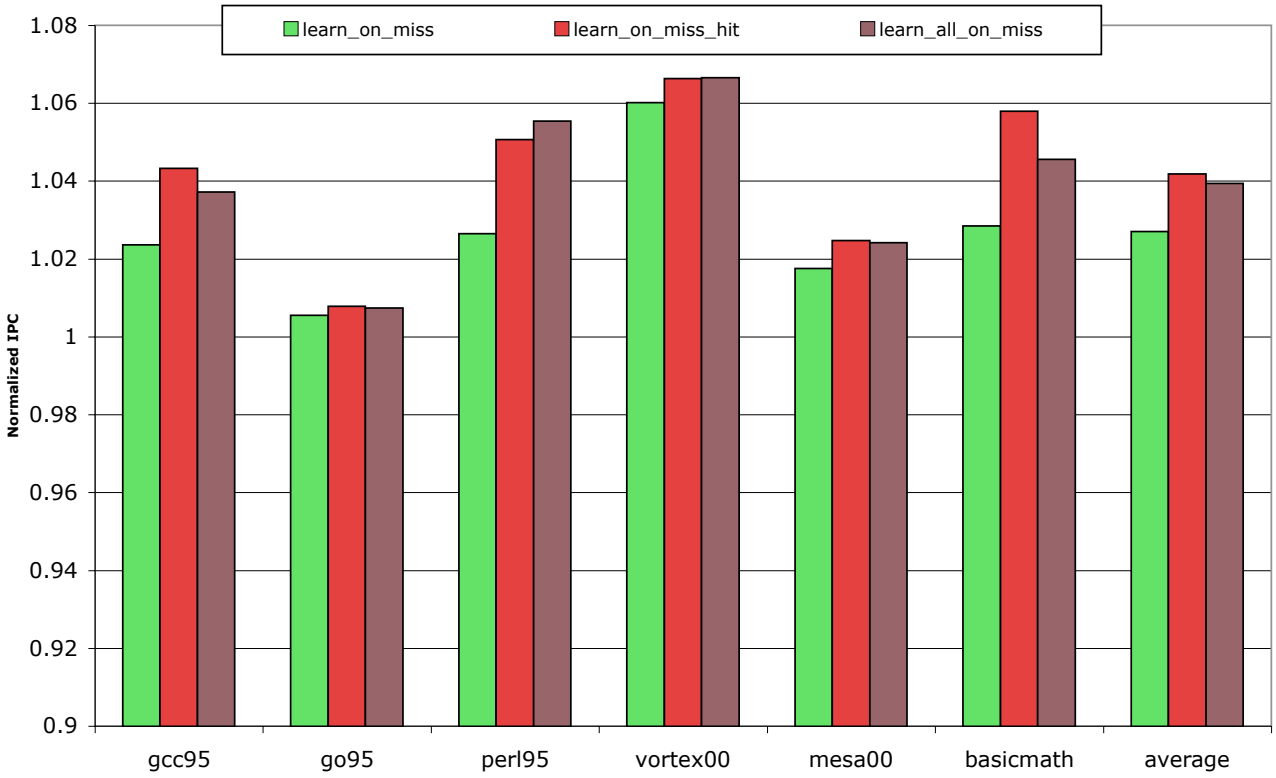

Figure A.2. Learning techniques of CCD

Bibliography

[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. Enhacning 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] 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.

[19] 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.

[20] M. A. Postiff and T. Mudge. Smart register file for high-performance microprocessors. Technical Report CSE-TR-403-99, University of Michigan, June 1999.

[21] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2):111–120, February 1999.

[22] R. Sendag, P. Chuang, and D. Lilja. Address correlation: Exceeding the limits of locality. *Computer Architecture Letters*, 2(1), May 2003.

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

[24] 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.