CACHE CONTENT DUPLICATION

Marios Kleanthous

University of Cyprus, 2012

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance, and it has become more important in Simultaneous Multithreading processors and Chip-multiprocessors. To cover this memory gap, caches have been the subject of numerous studies aiming to improve their performance as well as their power and area efficiency.

This thesis identifies a new phenomenon in caches that has the potential to improve cache performance and efficiency: the Cache Content Duplication (CCD). CCD occurs when there is a miss for a block in a cache and the entire content of the missed block is already in the cache in a block with a different tag. Caches aware of content-duplication can have lower miss penalty by fetching, on a miss to a duplicate block, directly from the cache instead of accessing lower in the memory hierarchy, and can have lower miss rates by allowing only blocks with unique content to enter a cache.

The usefulness of CCD is also examined at all levels of the memory hierarchy. First, we show that CCD is a frequent phenomenon for instruction caches and that an idealized duplication-detection mechanism for instruction caches has the potential to increase performance of an out-of-order processor, with a 16KB, 8-way, 8 instructions per block instruction cache, often by more than 10% and up to 36%. We also propose CATCH, a hardware mechanism for dynamically detecting CCD for instruction caches. Experimental results for an out-of-order processor show

that a duplication-detection mechanism with a 1.38KB cost captures on average 58% of the CCD's idealized potential.

Second, we examine another case of CCD which we call Text Cloning. Text Cloning can occur when running multiple *copies* of the same binary, Extrinsic Text Cloning, or when running multiple *instances* of the same application in a Virtually Indexed Virtually Tagged cache, Intrinsic Text Cloning. Results show that both Intrinsic Text Cloning and Extrinsic Text Cloning can reduce an application's performance. Specifically, Extrinsic Text Cloning causes up to 11% slowdown on existing platforms. Furthermore, we show that CATCH can benefit performance by eliminating the duplication due to Intrinsic Text Cloning and Extrinsic Text Cloning.

Third, we investigate the potential of CCD for L1 data caches. The results indicate that caches exhibit a high amount of dirty blocks thus making the CCD detection and creating stable correlations between different blocks very difficult. If a block is written, all duplicate relations to that block need to be invalidated. Our analysis also shows that zero runs are very frequent in L1 data caches and, therefore, previously proposed zero detection mechanisms can provide good solutions.

Finally, this thesis considers the CCD phenomenon for Last Level Caches (LLCs). The LLCs are written less frequently (L1 data cache acts as a filter) and have less zero runs because they mostly store evicted cache blocks that have already written with non-zero values. Results indicate that CCD is very frequent for various block granularities, from 4 to 64 bytes, and has potential to improve processors performance or save energy. A new cache design, the Content Duplication Aware Cache, is proposed to detect and eliminate CCD in LLCs. The results indicate that the Content Duplication Aware Cache can improve performance moderately but can reduce Energy Delay product considerably, 10% on average and up to 15% at most, for multiprogram workloads.

CACHE CONTENT DUPLICATION

Marios Kleanthous

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

April, 2012

2012

APPROVAL PAGE

Doctor of Philosophy Dissertation

CACHE CONTENT DUPLICATION

Presented by

Marios Kleanthous

Research Supervisor _____
Yiannakis Sazeides

Committee Member _____
Pedro Trancoso

Committee Member _____
Demetrios Zeinalipour

Committee Member _____
Emre Ozer

Committee Member _____
Andre Seznec

University of Cyprus

April, 2012

ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisor for his support during all the years of my Ph.D. I am deeply grateful for his help, patience, and mostly for his guidance that lead to the completion of this thesis. He taught me how to conduct research and gave me all the tools for a promising future. He has been a great teacher but also a great friend and for that I would like to thank him from the bottom of my heart.

Next, I would like to thank all my Ph.D. committee members for their time and for their punctual comments that helped me improve this thesis. Special thanks also go to all anonymous reviewers that contributed to the development of this thesis over the years.

Also, I would like to thank all my colleagues and friends at the University of Cyprus and Xi-Group for their feedback but also their friendship. Special thanks go to Fanos and Andreas that have been my first colleagues in the lab and still great friends and to Damien for his support and valuable feedback during the last few weeks while preparing my thesis defense.

I am also deeply grateful to my family and especially my parents Michalis and Giannoulla, and my sister Rafaella that stood next to me during all the years of my studies. They have been always there to share the good and the bad moments with me. Also, I would like express my deepest love to my wife, Georgia, for putting up with me all these years and for being always there to support me and encourage me. Without my family and my wife this thesis would have been impossible.

I would also like to thank all my friends outside the University of Cyprus that made my life more interesting during my studies and I believe they will continue to be part of it for many years to come. Especially I would like to thank my best friends Konstantinos, Kyriakos and Telis, and my fishing buddies Giorgos and Christos.

Finally, I would like to dedicated this thesis to all the people that dream for a better world no matter what they face.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The importance of caches and memory hierarchy has increased over time due to the growing gap between processor and memory performance [2]. The memory gap, as shown in Figure 1, has been growing by more than 50% in the last decade and has become more pronounced the last few years with the wider use of Simultaneous Multithreading (SMT) and Chip Multiprocessor (CMP). The applications are also becoming more demanding by exploiting all the computational power provided by the state of the art processors.

The combination of multi-cores and multi-threading is effective in improving processor utilization as long as the memory hierarchy can satisfy all running threads instructions and data needs. Consequently, modern processors devote a large fraction of their real estate for the cache hierarchy and numerous research studies are conducted on how to efficiently share the cache hierarchy among concurrent on-chip threads [3, 4, 5]. These proposals are aimed to overcome various daunting memory related challenges.

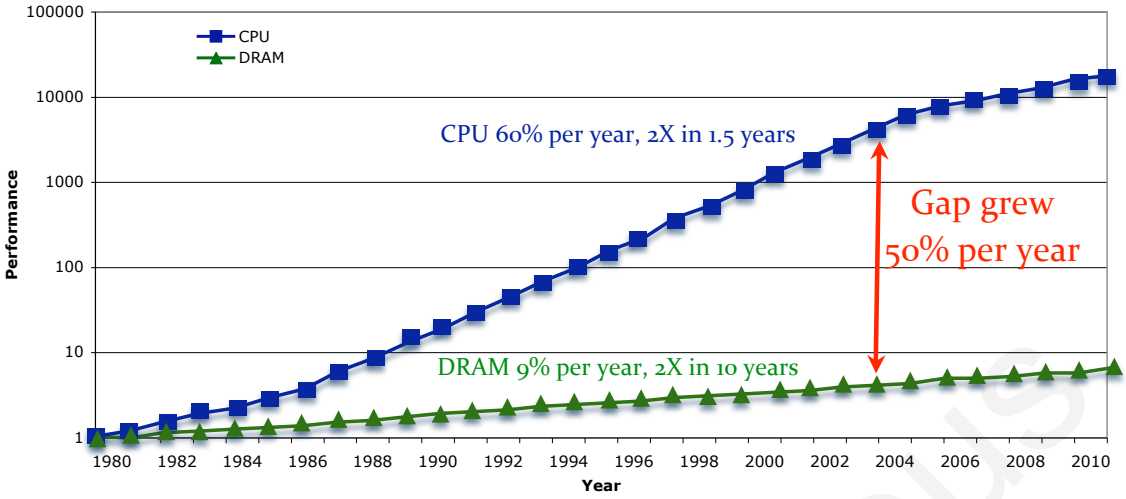

Figure 1: Memory Gap [1]

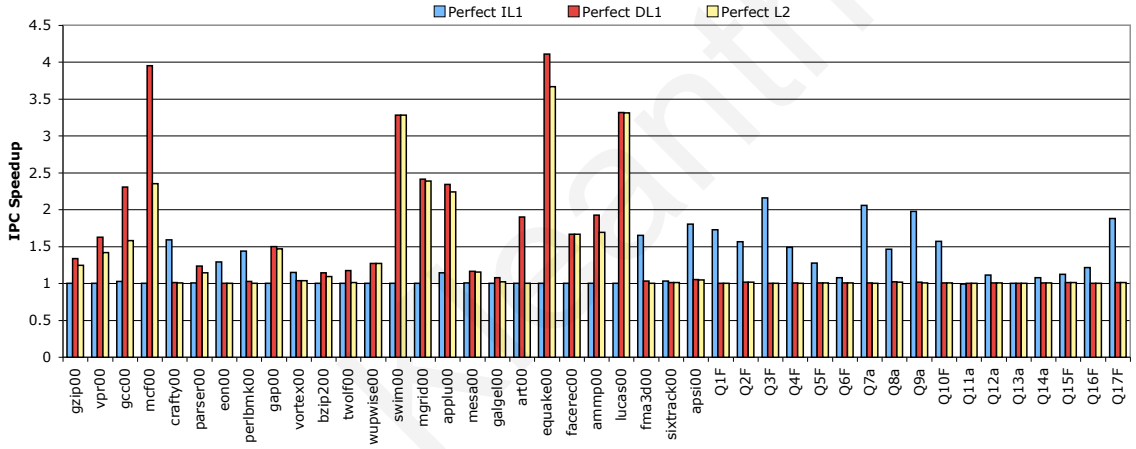

Figure 2: Performance improvement of an out-of-order processor with perfect cache

1.1 Memory Challenges

The performance of caches has always been limited by their size and energy constrains. In order for the caches to be beneficial they have to be small, because this will provide lower latencies, but also need to be energy efficient since they occupy a large portion of the chip.

Adding more levels in the memory hierarchy and bigger caches in the processors is not the answer anymore since both approaches are costly. The Figure 2 shows the performance potential of a high performance processor using various workloads with a perfect L1 instruction, L1 data

and L2 unified cache. The results indicate that there still room to improve the memory gap. We can also observe that different application types to need more instruction cache and others more data and L2 cache. The main challenges of caches are the following:

- **Average Latency**: The ultimate goal is to reduce the average latency as low as possible while maintaining the same or similar cache size. This can be achieved by several mechanisms, like prefetching, victim caches, e.t.c [6, 7]. Reducing misses, by using prefetching for example, helps reduce the average access latency of the cache.

- **Effective Capacity**: Another challenge is to increase the effective capacity of a cache without increasing the physical area. The most common technique to achieve this is by compressing the data in the cache. In this way, more data will fit in the same cache size. A trade off when using compression is to increase the hit latency for larger capacity. The goal though is to achieve a total lower average latency, so the increase in the hit latency will be eventually compensated by the savings due the cache miss reduction.

- **Bandwidth Requirements**: With the use of SMT and CMP processors, requests to the main memory and lower levels of the cache have increased putting pressure to limited off-chip bandwidth. To overcome this challenge several techniques have been proposed, including the use of compressed data through the bus.

- **Power Constraints**: The power constraints have been and will remain to be a primary design constraint for the years to come. The challenge aimed to provide high performing memory hierarchies and low energy has been the subject of several research projects and is still an open issue.

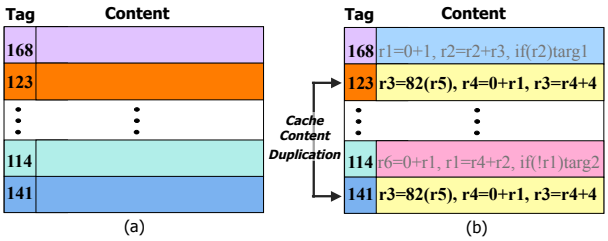

Figure 3: Cache Content Duplication (a) Without CCD, (b) With CCD

1.2 Thesis Contributions

This thesis identifies a new cache property that may influence cache performance: the Cache-Content-Duplication (CCD). This phenomenon occurs when there is a miss for a block in a cache and the content of the missed block resides already in the cache in another block with a different tag. Therefore, CCD is a manifestation of redundancy in the cache content. For example, Fig. 3.a shows an instruction cache where each block is identified by its tag and Fig. 3.b shows an instruction cache which is aware of the block content. This example shows that two different blocks, with tags 123 and 141, have identical content. If block 141 is evicted and later we have a miss on it, the content of 123 can be used without accessing a lower level of the memory hierarchy.

By identifying and exploiting CCD we can address most of the memory challenges mentioned earlier. Removing CCD from caches helps increase the effective cache capacity and therefore improve the cache performance. Also, by identifying CCD and reducing the cache misses will affect the bandwidth since the requests for data reduced. Compressing the data array and power gating its inactive portion can reduce static leakage of the cache.

The main contributions of this thesis are the following:

- **CCD for Instruction Caches**: In Chapter 4 we investigate the potential of CCD in instruction caches. We will characterize the phenomenon of CCD and measure its frequency in various types of instruction caches like block-based caches, trace caches, and normal caches.

We will also discuss which cache parameters might influence CCD and what optimizations can be applied to improve the detection of CCD in caches.

Then we propose two new cache types, the Duplicate-Aware-Cache (DAC) and the Unique-Content-Cache (UCC) that can exploit the CCD phenomenon. Both caches aim to reduce the cache latency on a miss by using a duplicated block in the cache while the UCC cache also increases the effective cache size by allowing unique blocks to enter the cache.

Furthermore we propose CATCH, a hardware mechanism that can dynamically detect CCD, and an investigation of its performance for DAC and UCC instruction caches. The various components of the mechanism are described, and various optimizations are proposed to increase performance and to reduce the cost of the mechanism.

In Chapter 5 we study another case of CCD for instruction caches, the **Extrinsic and Intrinsic Text Cloning**. Text Cloning refers to the phenomenon where identical code, from the same or different applications, coexist simultaneously in an instruction cache. We investigate the effects of Text Cloning on real platforms to quantify the performance degradation due to this type of redundancy. Finally, we discuss possible applications of CATCH mechanism to detect and eliminate this text cloning.

- **CCD for Data Caches**: Chapter 6 investigates the potential of CCD for data caches. We present analysis of duplication in data caches at the block level and discuss why CCD optimizations are hard to apply on data caches.

- **CCD for Last Level Caches**: In Chapter 7 we present a study on Last Level Caches both for single and multiprogram workloads. Specifically we measure the frequency of CCD in a LLC cache at various granularities. We also propose a new cache design, the Content Duplication Aware (CDA) cache uses extra tags to insert more compressed data in the data

array or improves energy efficiency by switching off part of the data array that is not used due to compression.

An optimistic implementation of CDA is evaluated in terms of performance and energy to establish the potential of the proposed approach.

1.3 Main Output of this Thesis

The work during this thesis resulted in the publication of a journal paper, a conference paper, a workshop paper (later published in a special issue journal), three technical reports and two posters.

A technical report [8] described CCD in instructions caches and provided an initial analysis of the phenomenon. An abstract and a poster of this technical report was presented in ACACES 2005 [9].

The effects of detecting duplication at the granularity of valid blocks were presented in ACACES 2006 [10] as a poster. Next there was another technical report [11] that extended the first with performance analysis and provided a detail mechanism to dynamically detect and exploit CCD, CATCH. The same work was later published in DATE [12].

A special case of CCD, the Text Cloning, was presented during WIOSCA 2010 workshop [13]. The paper was later published in LNCS journal. Also, an extension of the DATE paper with more thorough analysis of the effects of CCD and CATCH on other caches optimizations, like prefetching, was accepted for publication in TACO journal in 2011 [14]

Finally, a framework to identify the simulation regions for benchmarks was developed during this thesis and appears as a technical report in [15].

1.4 Other Output from this Work

During this thesis we have also worked on other projects that resulted to publications and a patent.

1.4.1 Improving Branch Prediction by Considering Affectors and Affectees Correlations

Branch prediction has been the subject of several papers. In this work [16, 17] we have investigated the potential of direction-correlations, for both affectors and affectees, to improve branch prediction. The correlations are determined based on data-flow graph information and used to select a subset of history bits that affect a prediction.

The main contribution of this work is the insights that provide on how and why predictors work using history and the possibility of predictors that can efficiently learn correlations that may be non-consecutive from long branch history.

1.4.2 Entry Replacement Within a Data Store

A mechanism to improve the performance of value predictors was developed in [18, 19]. A value predictor's performance relies on the predictability of the program phase. Some program phases are very predictable, for example streaming or stride behavior, while others are completely random. The more load instructions we can keep history of in the value predictor the better performance we will have.

Investigating the behavior of several applications we observed that only a fraction of the load instructions is really predictable and the rest just pollutes the value predictor and evicts useful information.

A replacement predictor was proposed that predicts the predictability of a bigger subset of load instructions by approximating a large direct mapped value predictor without tag matching and

using hashed information. Using this table we effectively profile the behavior of an application, or a certain phase of it, and decide whether the value predictor should be updated or not with a certain load instruction.

The main contribution of this work was a novel replacement policy to select the best values to update a value predictor using a small direct mapped table.

1.5 Thesis Outline

The rest of the thesis is organized as follows: In Chapter 2, a background on memory optimizations and previous work related to cache redundancy is discussed. The Chapter 3 presents the metrics, the simulation environment and the benchmarks that were used in this thesis. In Chapter 4, we discuss our work on Instruction Caches and Chapter 5 presents another case of CCD for instruction. Chapter 6 investigates the potential of CCD in Data Caches. In Chapter 7, an analysis of CCD in LLCs is presented and a mechanism is proposed to exploit CCD. Finally, Chapter 8 provides conclusions and directions for future work.

Chapter 2

Background and Related Work

Since their introduction to the motherboards as fast small off-chip memories and later their integration into the processors as on-chip memories, the caches have improved the processor's performance. This Chapter will provide a background on several memory optimizations that have been proposed with more emphasis on previous work related to cache compression optimizations.

2.1 Memory Hierarchy Optimizations

The caches have been the central to numerous research studies, all aiming to cover this gap more efficiently. 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 [20], predictability [21, 22], and redundancy [23, 24].

The rest of this section categorizes and describes the most important techniques that are applied to improve cache performance and comments on different approaches.

2.1.1 Replacement Policies

The optimal cache replacement policy has been the holy grail in cache optimizations research. The decision for which block to replace from the cache is very critical. The policy has to choose a block that hasn't been used for a long time but also to predict, in a way, that will not be reused in the near future.

Replacement policy's performance always depends on the workloads characteristics so a good replacement policy must be able to adapt to different phases of a program. There are mainly three different approaches that researchers are using to develop or improve a cache replacement policy.

The first approach is the *Optimal Replacement Approximation* [25, 26]. The main goal is to predict which block in the set will be needed further in the future. This can be achieved by keeping a history of the previous replacements and accesses and employing a predictor to indicate the victim. The new block to be inserted in the cache can also be the victim and here the new block is just bypassed, and there are no replacements in the cache.

The second approach is the *Adaptive Replacement Policies* [27, 28, 29]. In this case, researchers attempt to categorize different program phases, such as streaming or good locality or big workloads, and dynamically identify these phases. Once the behavior of the application that is exhibited in the current phase is identified, the appropriate replacement policy can be applied. It is very common that a phase that exercises a big workload is best handled by an MRU policy, while a phase with a good locality will be exploited better using an LRU policy.

Finally, the third approach is the *Global Replacement Policies* [30, 31, 32]. The idea behind this approach is that some of the sets in the cache are very hot while some others are rarely used and can also have empty slots. Using a secondary index the Global Replacement Policies attempt to find an alternative place to insert a block if it doesn't fit in its primary location. Effectively the

Global Replacement Policy attempts to emulate a fully associative cache using a direct cache or a set associative cache. By doing this, we can achieve better utilization without bearing the high cost of accessing a fully associative cache.

2.1.2 Prefetching

Another common optimization is the cache prefetching. The researchers noticed that since the memory hierarchy on a miss is not fast enough to cover the throughput of out-of-order processors then the missed block should be predicted and prefetched before the processor requests it.

To achieve prefetching, future knowledge is required to predict the next access. Since this not possible, even with profiling it cannot be 100% accurate, many have tried to exploit the program regularities to predict the next missed block and prefetch it. We divide the previous work on cache prefetching into three categories.

The first category is the *Compiler assisted prefetching* [33]. The ideal prefetcher will be the one that knows the future and the best way to learn the future is to take a look by profiling. Using profiling and programmer's knowledge the compiler can be armed with the ability to "know" the future and add special instructions that will initiate prefetching when necessary. This approach is very fast and accurate since we have almost perfect knowledge of the program behavior and there is no need for probing and updating predictors. The downside of this technique is that it requires profiling which is extra work and not always possible, for example with legacy code. Also, different program inputs may lead to multiple control flow paths and this will require either more profiling or less accurate prefetching.

Another category of prefetching is the *Fetch streaming* [34, 35]. The use of history predictor is very common in processor designs and has been also used for prefetching. Although sometimes a simple next line predictor can be very efficient most of the times the addresses of the instructions

executing are not continuous. The authors in this category tried to create a chain of dynamic instructions that do not have to be continuous in the static code to use this later for prefetching. Once the sequence of instructions is identified and stored in the predictor the first instruction of the sequence is assigned as the trigger instruction. When the program execution reaches the trigger instruction, a sequence of events is also initiated by the predictor to execute the instructions that were also executed in the past when the trigger was called. This will result to speculative data and instructions to prefetched in the cache.

The last category is the *Runahead Execution* [36, 6, 37] where mechanisms are proposed to speculative run the application and force the misses before they are requested so they will prefetched in cache. By exploiting the extra throughput provided by the SMTs when at least one thread slot is idle, the processor can run a speculative version of the application in parallel with the normal execution but slightly forward few hundred of instructions. If the speculation is correct, useful data will be fetch in both data and instruction caches. A very important detail of the runahead execution is to detect the optimal distance between the runahead version of the application and the normal version. Going too much forward means that unnecessary information will be fetched that may pollute the cache and cause more misses, while going too little forward means that the runahead execution will not be fast enough to fetch the requested information on time.

2.1.3 Compression

Data compression techniques have been around for a long time. People noticed from the beginning of writing that some words are most common than others and if we represent those words with fewer letters then the size of the text gets smaller. The same idea applies when we compress bits and bytes in a computer's memory and using a dictionary we represent the most

common sequences of data that can be from few bits to a whole memory page, with smaller codes aiming to reduce the space needed to store our data.

The data compression was first applied in disks in the early 1990s where the size of a hard disk was at the range of few megabytes to few tens of megabytes. There were several DOS [38] applications to compress files and directories in the disks to increase the free space. Today, disk compression is still used but is mainly applied to increase the effective bandwidth in networks such as the internet or any other network systems. Venti [39] is an example where disk compression is used to save space in a network storage system intended for archival data. Venti hashes the data to detect and eliminated duplicated information stored in magnetic disks. Also, a common application of disk compression is for files that are going to be distributed through the internet or any other local network to save bandwidth.

Another emerging application of compression is in the region of virtualization. Virtual caches might have vast amount duplication because they might be even running the exact image of a system multiple times. This duplication can be eliminated by using mechanisms [40] to detect whole memory pages that contain the same data and merge them. A similar approach is used by the Linux operating system and is called Kernel SamePage Merging (KSM) [41, 42] where a hypervisor is checking all memory pages and merge any duplicated pages, pages that contain the exact same data. In case of a write on such a page the dirty page is copied (copy-on-write). KSM idea was first proposed for virtual memory and then applied to Linux operating system for the first time on version 2.6.32.

Finally, another application of Compression is the cache compression, which is also the most relevant to our work. More specifically in caches the compression is very important to keep them small and fast. By compressing the data we increase the effective cache size without increasing

its physical size that will lead to longer access times and higher energy per access. Also by transferring data in a compressed form between various levels in the memory hierarchy the effective memory bandwidth is also increased.

There are two different approaches in cache compression, the *Dynamic Compression*[43, 23, 44, 45, 46, 47, 48, 49, 50, 51, 23, 52, 53, 54] and the *Static Compression* [55, 56, 57, 24, 58, 59].

Dynamic Compression provides flexibility and adaptivity to the program phase changes. Several researchers adopted this approach and either by building a dictionary through profiling or by dynamically detecting the redundancy they tried to improve the cache performance by increasing the effective cache size. Furthermore, the dynamic compression also provides the option to turn off compression if the overheads become too high on a certain program phase.

On the other hand the *Static Compression* gives more room for transformations and a global view of the application in order reduce the required space even more. The main disadvantages of Static Compression are that it cannot be applied to legacy code either can be turned off during execution.

A more extended discussion on Cache Compression techniques is presented in Section 2.

2.2 Related Work on Compression

This thesis will focus on the redundancy of the memory and cache content, which has also been the subject of several previous works. The main objectives of memory hierarchy redundancy optimizations are to increase the effective memory/cache capacity or to achieve higher bandwidth during the information transfer between the different levels of the memory hierarchy or both. This will result to performance improvement and/or energy reduction.

Previous work on memory hierarchy compression can be separated in two main categories, the *Dynamic Compression* which means the redundancy is detected dynamically and compressed in

any level of the memory hierarchy and the *Static Compression* where the data, either instructions, or information, are compressed before they are loaded in the memory.

2.2.1 Dynamic Compression

A scheme for main memory on-line compression was first proposed by Douglis [43]. This work is targeting the main memory to reduce the I/O between the memory and the disk. Keeping the LRU memory pages compressed the main memory can accommodate more pages, and thus reducing the misses and the need for I/O. The LZRW1 dictionary based algorithm is used for the compression with a 16KB hash table.

Kjelso and Gooch [23] proposed a hardware implementation of the X-Match dictionary compression algorithm for main memory data. Their algorithm requires about 30% more hardware than previously proposed hardware designs but can achieve 2-3 times faster compression and decompression rates.

Lefurgy et al. [44] 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 64K instruction dictionary. The authors observed that the high overhead of software decompression was slowing down the benchmarks, and they show that selective compression can reduce this performance loss.

The high redundancy of a subset of values in data caches was identified in [45]. A Frequent Value Cache (FVC) was proposed to hold the frequent values in compressed form. The FVC is accessed in parallel with a Direct Mapped Cache (DMC) and in case of a cache miss and a hit on FVC the miss was served from there. The FVC is updated only on replacements from the DMC.

Very relevant to our work is [46] 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 in another location. The authors investigated the limits of oracle address correlation, and found it to be significant, but did not propose a mechanism for detecting it.

Molina et al. [47] noticed the value replication in the cache and proposed the use of a decoupled tag array and data array in order to keep unique values in the data array and pointers from the tags to these values. They called this type of a cache a Non Redundant Data Cache.

In the same concept of [47] the authors of [48] proposed the Content-Based Block Caching. The difference from the previous is that is done for a buffer cache and given the advantage of higher latencies in software and memory availability the authors achieve to eliminate most of the limitations that [47] faced on hardware.

Alameldeen and Wood [49] keep information compressed, for both instructions and data, only in L2 cache and can dynamically choose to keep data in uncompressed form when the overhead of compression may cause degradation in performance.

Hallnor and Reinhardt [50] proposed a scheme that can map multiple compressed blocks into a single physical cache block using an Indirect Index Cache. A compressed block will require fewer segments for its data leaving additional storage for other blocks. 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 a higher transfer rate from main memory to cache.

Biswas et al. [51] investigate the phenomenon of data similarity in multi-execution programs. They observed that when multiple instances of the same application are running on a multicore sharing the same L2 cache, their data are usually very similar.

Many researchers have also noticed that there is a big tradeoff of compressing only zeros instead of all possible values. The main idea is that the zero runs in the data are very common and although may not cover a big part of the execution they are much simpler to compress.

Kjelso and Gooch [23] proposed an extension to their hardware implementation X-Match just to zero runs and they called it X-RL.

The zero values were exploited by Villa et al. [52] to save energy. They propose to use an extra bit for every byte stored in the cache that indicates if the block is zero or not. In case of a zero block only that one bit is needed to be read.

Ekman and Stenstrom [53] detected cache blocks that are for zeros using 1 extra bit per block in the Block Size Table (BST). The BST has as many entries as the TLB array and each entry in the BST has pageSize/blockSize bits. In this way, once a zero block arrives in the L2, it's marked it in its corresponding BST entry and they don't cache it at all. Eventually, if that block is referenced again the BST will indicate that the block is all zeros.

The benefit of eliminating zero blocks was also studied by Dusser et al. [54]. The authors propose an extra cache, called Zero-Content cache, to keep track of zeros in the regular cache. Each entry in the ZC cache contains a tag (the sector address) and a number of N bits. N equals to SectorSize/CacheBlockSize and each bit corresponds to one of these blocks. A cache miss followed by a ZC cache hit means that the block is all zeros and an access to lower levels of memory hierarchy is saved.

Finally, the effects of compression to the area and energy were studied by Kim et al. [60]. They proposed the use of two smaller caches, instead of a big one, and smaller block size. All the compressible blocks reside in the first cache while the uncompressed blocks can be in both caches. The authors show that using their scheme can save both and energy and area as compared to a conventional L2 cache with at most 0.5% performance degradation.

2.2.2 Static Compression

Lefurgy et al. [55] 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 when requested for execution.

Benini et al. [56] proposed a dictionary based compression technique for firmware code executed on embedded systems. The aim for this work was to reduce the energy required during execution by compressing the most commonly used instructions to reduce memory accesses. This scheme does not require any processor modification since the instruction decompression is performed on the fly by a hardware module between the memory and processor.

Hines et al. [57] proposed the use of an Instruction Register File (IRF) for holding frequently executed instructions. An integrated compiler/hardware mechanism exploits this to reduce the code size and power requirements and to improve performance.

2.3 Code Compaction

Code compaction methods [24, 58, 59] are used to reduce the executable code size without a need to decompress the compacted code for execution. Code compaction work could be easily labeled as *Static Compression* and included in the previous Section, 2.2.2, but we wanted to differentiate it because it deals only with instructions and employs very special optimizations.

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.

Code compaction transformations have also been proposed to convert "superficially" dissimilar sequences to repeated. 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 code compaction is run-time 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.

2.4 Dynamic VS Static Techniques and Mechanisms

The benefit of static compression comes from the advantage that the compiler and the programmer can have a bigger picture of the application and that enables better transformations and optimizations to eliminate redundancy. On the other hand the Dynamic approach, although limited in the number of transformation, can be adaptive and applicable to legacy code. Especially the adaptivity seems to be a very important factor in cache optimizations since the behavior of an application can change so radically that a specify optimization will degrade, rather than improve, its performance.

Another aspect of compression though is the mechanism that detects and removes redundancy. There are Static and Dynamic mechanisms mostly in the sense of static and dynamic dictionaries that keep the most frequent patterns for compression. The advantage of a static dictionary is that we can achieve maximum compression for a certain set of patterns, but the downside is that those patterns might be not frequent anymore if the application was profiled with different inputs than

the ones are currently used. Using a dynamic mechanism, that can update its dictionary or create other kind of dynamic correlations between duplicated data, will provide more performance in cases where profiling is not efficient or even cannot be done due to legacy code. The downside of such a mechanism is that it needs more logic to detect the redundancy rather than just probing a dictionary of frequent patterns.

In the next section, we will describe our approach and why we choose it based on the previous work and discussion in this chapter.

2.5 Cache Content Duplication (CCD)

We propose Cache Content Duplication (CCD), a phenomenon that appears in the cache and can be exploited by a dynamic hardware mechanism. CCD is defined in Section 1.2.

The key difference of our work from most of the previous effort [43, 23, 44, 45, 49, 50, 52, 55, 56, 57, 53, 54] is that we consider redundancy at the granularity of cache blocks and detect it dynamically using a hardware mechanism instead of comparing arbitrary patterns with profiling aid. Furthermore, most of the previously proposed techniques have static dictionaries or compress only the zero value. We propose the use of a dictionary that is built during execution by dynamically detecting duplicated sequences.

Code compaction [24, 58, 59] and CCD for instruction caches share some similarity since both exploit redundancy in code. However, compaction methods are compiler based whereas the method considered here is dynamic hardware based. The static approach can detect repetition at a coarser scale, for example functions with multiple basic blocks. CCD duplication 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.

Few other works attempted to use dynamic mechanisms and dynamic dictionaries to compress the caches like CCD. One of these works is the address correlation [46] which also exploits the duplication of content at different addresses dynamically. Nonetheless, our work is distinct because: (a) we also consider the duplication of instruction blocks whereas in [46] the focus is individual data values, and (b) we propose hardware mechanisms for detecting and exploiting CCD.

The authors of [47] and [48] propose something similar to our work on Last Level Caches. The approach is the same but the authors of [47] do not provide any performance evaluation of their mechanism and also make many assumptions for the cache to avoid the problem of backward pointers from data to tags when a tag invalidation is needed on data replacements. On the other hand the authors of [48] present a mechanism that is only applicable on buffer caches and cannot be applied in hardware since it requires expensive data structures like very long link lists.

Biswas' work [51] also exploits the CCD phenomenon dynamically as proposed in [12] but only for a specific scenario in which multiple instances of the same application share an L2 cache. Our work extends more by detecting CCD in multiple levels of the memory hierarchy, different granularities and multiprogram workloads with different applications. We also detect CCD within the same application while [51] considers only CCD across multiple instances of an application.

Overall, previous work considered either the compression and compaction of arbitrary length sequences of data or instructions, or the compression at the granularity of individual instructions or values. Our work's major differences from previous are that we combine all the following characteristics:

- We propose a dynamic hardware detection mechanism that detects and creates relations of duplicated content dynamically. This, as opposed to profiling, enables the mechanism

to detect and remove all possible duplication in the cache at the block level and smaller granularities

- The redundancy is detected dynamically without static dictionaries and profiling. The dynamic detection using hash content allows to apply our design to any application and inputs and legacy code

- We detect duplication at block level for various granularities while most of the previous work was focusing on frequent values or whole cache blocks.

- We detect duplication within the same application on a single thread execution and across different applications on a multiprogram execution. This enables the detection of duplication among similar codes and statically linked libraries.

Table 1 summarizes all previously proposed techniques and their characteristics including their hardware and performance overheads.

Table 1: Summary of related work

Article	Granularity	Algorithm	Value range	Perf. Overhead	Area overhead	Cache Level
Dynamic Compression						
Douglis [43]	Memory Pages	LZRW1	16KB hash table	-	38KB + 0.8% of memory size	Main memory
Kjelso [23]	Memory Pages	X-Match	-	100MB/sec Compression, 140MB/sec Decompression	110k gates	Main memory
Lefurgy [44]	Instruction	Frequent instructions	64K instructions	75 cycles/compressed block	Software Decompression	Main memory
Zhang [45]	Word	Frequent values	7 values	0 (Accessed in parallel)	3KB (32KB DMC, 64byte block)	L1 data cache
Sendag [46]	Word	Dynamic Correlation	-	-	-	L1 data cache
Molina [47]	Segments	Dynamic Correlation	-	-	-	L2 cache
Morrey [48]	Segments	Dynamic Correlation	-	-	-	Buffer cache
Alameldeen [49]	Patterns	Frequent Patterns	7 patterns	5 cycles/compressed block	19bits-counter + Decomp. logic	L2 cache
Hallnor [50]	Block segments	LZSS	-	32 cycles compression, 8 cycles decompression	134KB	Main memory and L3 cache
Biswas [51]	Cache blocks	Same virtual address	All possible	0 (normal cache access)	4.28% of the cache size	L2 cache
Kjelso [23]	Memory Pages	X-RL (Addition to X-Match)	Zeros	100MB/sec Compression, 140MB/sec Decompression	0 (Addition to X-Match)	Main memory
Villa [52]	Bytes	Dynamic detection	Zeros	0 cycles	1 bit per byte	L1 data cache
Ekman [53]	Cache blocks	Zero blocks	Zeros	5 cycles	- (1 bit per cache block in the TLB)	L2 cache
Dusser [54]	Cache blocks	Zero blocks	Zeros	0 cycles	80KB	All levels
Kim [60]	Half Cache blocks	Small Memory Values	16KB 4byte values	1 cycle	-53%	L2 cache
Static Compression						
Lefurgy [55]	Instruction sequences	Frequent sequences up to 8 instructions	8192 sequences	-	-	Static code (All levels)
Benini [56]	Instruction	Frequent instructions	255 instructions	1 cycle/compressed instruction	1.2KB	Main memory
Hines [57]	Instruction sequences	Frequent sequences up to 5 instructions	32 sequences	0 (Either IRF or IC are accessed)	128Byte (IRF)	L1 instruction cache

Chapter 3

Methodology

This chapter describes the methodology we used to achieve the goals of this thesis. We define the metrics that will be used to evaluate our ideas and mechanisms, we present the simulation infrastructure that has been used and finally we discuss the choice for the benchmarks and their characteristics.

3.1 Metrics

We will use several metrics to evaluate our limit studies and for the performance evaluation and energy analysis. Different metrics for single program workloads and others for multiprogram workloads will be used. Also, we define a new metric that is related to our work and it's called CCD rate.

The metrics that will be used in this thesis are the following:

- **Accesses per 1K instructions**: A very important metric that we will be using is the Accesses per 1K instructions. We choose this metric to estimate the performance potential of a benchmark during the functional execution. We decided to use Accesses per 1K instructions and especially MissesPer1K instructions over the Miss Ratio because we believe that

it is a more representative performance metric during functional simulations. The reason is that an improvement on a benchmark with very low MissesPer1K instructions will have no effect on performance while the Miss Ratio reduction might be misleading. For example, a benchmark that has only 100 cache accesses and 50% Miss Ratio will result to only 50 misses. Even if we eliminate all 50 misses and decrease the Miss Ratio from 50% to 0% this will have no effect on the performance. On the other hand, the MissesPer1K Instructions metric will immediately indicate the very low miss count that would be an indication for no margins for performance improvement.

- **CCD rate**: The next metric that is also very important for our analysis is the CCD rate. The CCD rate refers to the fraction of misses that are for duplicate-blocks already in the cache in another location. This metric will be used during the limit study analysis and will show if any benchmark has potential to improve its performance if we could avoid the misses that already have a duplicate in the cache.

- **IPC speedup**: As a performance improvement metric for our single program workloads we choose the IPC speedup. The IPC speedup represents the improvement of IPC using our mechanisms relative to the baseline configuration with a cache that is not CCD-aware. To summarize the IPC speedup of many benchmarks the geometric mean is used.

- **Weighted IPC speedup and Harmonic mean**: For the multiprogram workloads we choose the Weighted IPC speedup [61] and Harmonic mean [62]. The Weighted IPC speedup provides the performance improvement in terms of throughput while the Harmonic mean provides an indication of the fairness during execution. The second, the Harmonic mean, will be used as a filter to avoid coming to conclusions for unfair workloads.

- **Dynamic Energy per access**: The dynamic energy per access will be used to measure the effects of our proposed mechanisms in energy consumption. This metric will be an indication of how much the energy consumption is increasing when our mechanisms are employed.

- **Energy Delay product**: Finally, the last metric that will be used in this thesis is Energy Delay product. This metric provides the energy efficiency of a mechanism considering also its performance variation as compared to the baseline. The Energy Delay product will be used to evaluate our mechanisms energy efficiency.

3.2 Simulation Infrastructure

This sections describes the simulator we used and the extensions we made to evaluate the CCD phenomenon and the proposed mechanisms.

3.2.1 Simulator and Extensions

We used the SMTSIM [63] simulator for our experiments. The SMTSIM simulator provides both functional and timing simulations and enables the study of single cores processors, SMT capable processors and chip multiprocessors. The configurations that we will be using are described and explained below.

3.2.2 Single Core Configuration

Table 2 shows the parameters that we used for our single core configuration. We choose these parameters that reflect to a high end single chip processor based on initial version of Intel Core Solo at 1.5Ghz but instead we assume a 2.4Ghz frequency. Based on Core's configuration and

Table 2: Single Core Baseline Configuration

fetch/issue/commit width	4/4/4
INT Issue Queue/FP Issue Queue/ROB	64/64/256
Pipeline Stages	10
L1 instruction cache	16KB 8-way 32B/block, 3 cycles
L1 data cache	16KB 8-way 32B/block, 3 cycles Write-back, Write-allocate
L2 unified cache	2MB 8-way 32B/block, 20 cycles Write-back, No Write-allocate, Non inclusive
Main memory latency	200 cycles
Cond. branch predictor	8KB combining predictor
BTB	1024 entries
RAS	32 entries
Indirect predictor	512 entries

Table 3: Multi Core Baseline Configuration

fetch/issue/commit width	4/4/4
INT Issue Queue/FP Issue Queue/ROB	64/64/256
Pipeline Stages	10
L1 private instruction cache	16KB 8-way 64B/block, 3 cycles
L1 private data cache	16KB 8-way 64B/block, 3 cycles Write-back, Write-allocate
L2 shared unified cache	8MB 16-way 64B/block, 40 cycles Write-back, No Write-allocate, Non inclusive
Main memory latency	200 cycles
Cond. branch predictor	8KB combining predictor
BTB	1024 entries
RAS	32 entries
Indirect predictor	512 entries

with the new frequency we ran the caches on Cacti [64] and the results were the latencies shown in Table 2.

3.2.3 Multi Core Configuration

For the multicore configuration we used a similar configuration to the single core processor but with bigger L2 cache and cache block size. Table 3 shows the core parameters and the cache configuration for the private and shared caches.

3.3 Benchmarks and Characterization

For the workloads we choose to use the SPEC2000 suite and the TPC-H suite. All binaries are run with reference inputs and were compiled for the ALPHA ISA[65] using the Compaq C compiler and -O2 optimization level. Compiler optimizations, such as loop unrolling that increases duplication, are disabled. For the TPC-H suite we are using the postgres database server with SF=0.5 for the data set and the queries used are shown in Table 5.

Below we discuss our choice of regions and benchmarks that will be used throughout this thesis.

3.3.1 Regions

The regions that we choose to simulate were selected based on a framework [15] that was developed during this thesis. The idea of this framework is based on simpoint [66] analysis but instead of doing this automatically we decide to visualize the execution using various application statistics and choose the best regions to simulate. We define the best region as the region that is beyond the initialization phase, and it represents a large portion of the benchmarks execution.

For example by visualizing the access in different locations in memory and separating text segment access and data segment access we can define with high accuracy the region of the application that the most work is done which will be also the most representative regions for this application. More details can be found in [15] where there is a detail analysis and explanation for the choice of SPEC 2000 simulation regions. We followed the same procedure also for the TPC-H benchmarks but are not documented in any article.

Table 4: SPEC 2000 Simulated benchmarks

SPECINT 2000	Skip (10^6)	Execute (10^6)	SPECFP 2000	Skip (10^6)	Execute (10^6)
GZIP	19400	500	WUPWISE	7950	500
VPR	25600	500	SWIM	1150	500
GCC	8400	500	MGRID	150	500
MCF	13400	500	APPLU	2100	500
CRAFTY	950	500	MESA	450	500
PARSER	1000	500	GALGEL	4450	500
EON	26400	500	ART	3150	500
PERLBMK	13800	500	EQUAKE	19300	500
GAP	20900	500	FACEREC	36600	500
VORTEX	18600	500	AMMP	5200	500
BZIP2	43000	500	LUCAS	2650	500
TWOLF	7200	500	FMA3D	10300	500
			SIXTRACK	8200	500
			APSI	1650	500

3.3.2 SPEC 2000

Table 4 show the simulation regions for SPEC2000 benchmarks. These benchmarks will be used mostly to evaluate the data and L2 cache. As shown in Fig. 2 all the benchmarks in this suite, with the exception of CRAFTY, EON, PERLBMK and FMA3D, have very little room for improvement in the instructions cache while they have many data and L2 cache misses that can elevate performance if they are eliminated.

3.3.3 TPC-H

Table 5 shows the simulation regions for TPC-H queries. TPC-H suite exhibits a high miss counts in the instruction cache and very few misses in the data and L2 caches. This is also supported from the results in Fig. 2. So these benchmarks will only be used to evaluate instruction caches.

Table 5: TPC-H Simulated benchmarks

TPC-H	Skip (10^6)	Execute (10^6)
Q1F	200	500
Q2F	50	178
Q3F	200	500
Q4F	50	500
Q5F	1000	500
Q6F	500	500
Q7A	400	500
Q8A	500	500
Q9A	1000	500
Q10F	900	500
Q11A	500	500
Q12A	400	500
Q13A	400	500
Q14A	300	500
Q15F	500	500
Q16F	300	500
Q17F	50	133

To improve figure clarity, sometimes only a subset of these benchmarks will be presented but the results for all benchmarks will be also summarized and discussed. This applies to both SPEC2000 and TPC-H benchmark suites.

3.3.4 Multiprogram Workloads

A representative region should be an amount of **time** or **instructions** that will exercise different phases of a benchmark. The decision between **time** and **instructions** depends on the nature of the experiment. For multiprogram workloads we face a new challenge because it is very difficult to control all applications to run in their representative regions.

This requirement was also challenged in the past. Previous approaches [67, 68] to reduce the problem were by making the workload composition as homogeneous as possible. By doing this, the researchers assume that the experimental results will be more representative.

We have studied and evaluated all previous work, and we believe that a very accurate and efficient way to run a multiprogram workload is by using *Constant Time*. To do this, all applications are run in a single program mode and the total cycles to complete a representative region are noted. When we have multiprogram workloads, the total execution time will be equal to the total cycles of all applications in the workload when they ran alone. Furthermore, we already know for how many dynamic instructions a representative region expands for each application and after the execution of the multiprogram workload we reevaluated and test if any of the applications in the workload have executed more than 10% of instructions outside its representative region. If such a case is detected, then the specific workload composition is excluded from our experiments.

Chapter 4

CCD for Instructions

As a first step toward understanding and exploiting CCD, this work is focused on the content duplication in instruction caches. CCD in instruction caches exists because: (a) high level language programs often contain identical instruction sequences [69] in different segments of a program due to: copy-paste programming practices and reuse of standard libraries and loops in different parts of code, (b) conventions, such as for calls and returns, produce similar sequences, and (c) compiler transformations, such as compiler inlining and macro expansion, lead to duplicated code sequences.

By eliminating this redundancy, we aim to reduce cache misses and therefore improve processor's performance. The main challenge of this approach is to achieve better performance without increasing the energy or area in prohibitive levels. The effects and tradeoffs of detecting and eliminating CCD in instructions will be investigated further in this Chapter.

4.1 How to Detect CCD

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:

1. **regular instruction cache**

2. **basic-block cache** [70], 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. This work considers basic-blocks with a maximum of 4 instructions. A basic-block cache is used in the block-based trace cache proposed in [70]

3. **trace cache** [71] 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 considers traces with a maximum of 8 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 duplicating traces with the same starting PCs and the same content, but with different direction of the last branch [71]

4.1.1 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.

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 occur more likely between blocks that have fewer instructions (smaller cache blocks) and they are basic block aligned. Smaller sequences are more likely to match, and sequences aligned at basic block boundaries are more likely to be identical. To clarify, consider two basic blocks that are identical but reside in two different instructions blocks at different positions. In an instruction cache, the duplication may not be detected because the blocks that contain them are not aligned. Also, the instruction cache blocks may contain other instructions, aside from the duplicated basic blocks that are different.

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

A way to increase the frequency of CCD for regular instruction caches is to consider the duplication between *valid* instructions sent down the pipeline on an instruction cache access, instead of *entire* instruction cache blocks. In [72] a *valid* block is defined as the static consecutive instruction sequence starting from the current PC until: (a) the first predicted-taken conditional branch, or (b)

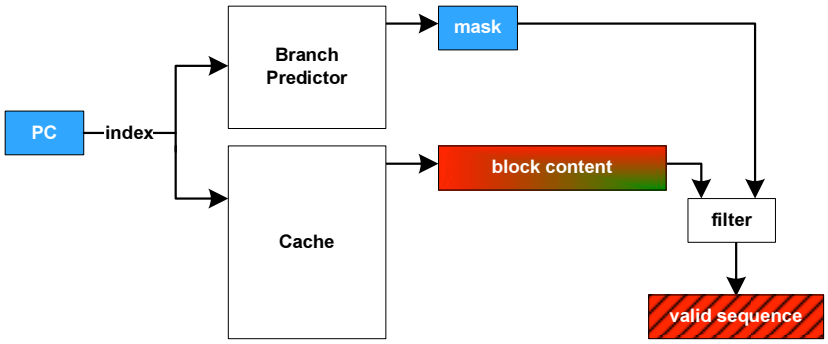

Figure 4: Valid block masked out from a cache block

the first unconditional branch, or (c) a number of instructions equal to fetch bandwidth are read from the cache. A *valid block* is identified by the starting PC and a *bit mask* that can be produced at each cycle using the BTB and the direction predictor [72]. This *mask* indicates the location of the first taken branch in a sequential instruction sequence. A valid block represents, therefore, the predicted instructions that are sent down the pipeline after a cache access, and we will refer to it as a *valid block*. Figure 4 shows how a valid block is build. 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. For other caches, such as a basic block or a trace cache, the valid block is virtually the same with the entire block content.

4.1.2 When to Learn the Cache Content

To detect duplication between cache blocks it is necessary to know the content of blocks already in the cache. This way, when a block misses the cache, it can be detected whether its content is a duplicate with a block already in the cache.

For a regular cache, a basic block cache, and a trace cache the content of a block can be learned by remembering its content when it is inserted in the cache. This is referred to as *learn-on-miss* learn policy. This policy is also sufficient to learn all the content in regular instruction caches when considering duplication of entire cache blocks. However, the *learn-on-miss* is not sufficient to learn all the relevant content in the case of valid blocks because, on a cache miss, an entire cache block is filled in the cache and the missed valid block covers only a subsequence of the entire block. One way to increase the frequency of CCD for valid blocks is to learn both missed valid blocks and valid blocks that are cache hits. This is referred to as *learn on miss and hit* policy. 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.

Another method is to learn on a cache miss the missed valid block content and heuristically learn other valid blocks in the missed block. We refer to this policy as *learn-all-on-miss*. An example heuristic is to build an additional valid block using the remaining instructions in the block after the missed valid block, and treat the next conditional branch to be encountered as taken.

Henceforth, unless indicated otherwise, the *learn-all-on-miss* policy is used for learning 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 4.6.

4.1.3 Which Sequences are Duplicated

Two valid blocks are considered duplicates if each instruction in a block is bit-wise identical in the exact order with its corresponding instruction in the other block. Nonetheless, the duplication

criteria can be relaxed for direct (conditional or unconditional) control transfer instructions by allowing differences in their immediate offset or target fields to increase duplication frequency. This technique is known in code compaction as target abstraction [59]. Section 4.5 discusses, in detail, how using a table that stores small target differences between otherwise identical sequences, facilitates more duplication while maintaining correctness. We note that other abstraction transformations, such as register and constant abstraction [59], can be applied to increase the duplication frequency. However, in this work we focus mainly on duplication detection. For the experimental results, unless stated otherwise, it is assumed that CCD employs target abstraction.

4.2 Code Redundancy Characterization

In this section, we characterize the frequency of duplicated sequences at the granularity of 32-byte blocks (8 instructions) and valid sequences (maximum of 4 instructions) during dynamic execution. The 32-byte blocks are 32 byte aligned, while the valid sequences are built dynamically as explained in Section 4.1.1.

The Figure 5 shows the number of unique blocks, at the granularity of 32 byte blocks, needed to cover a certain amount of dynamic execution when identified by their block tag, TAG, and by their unique content, CONTENT. First, the results show that very few benchmarks have redundancy at the granularity of a whole block. This is expected due to misalignment of code sequences as discussed in Section 4.1.1.

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

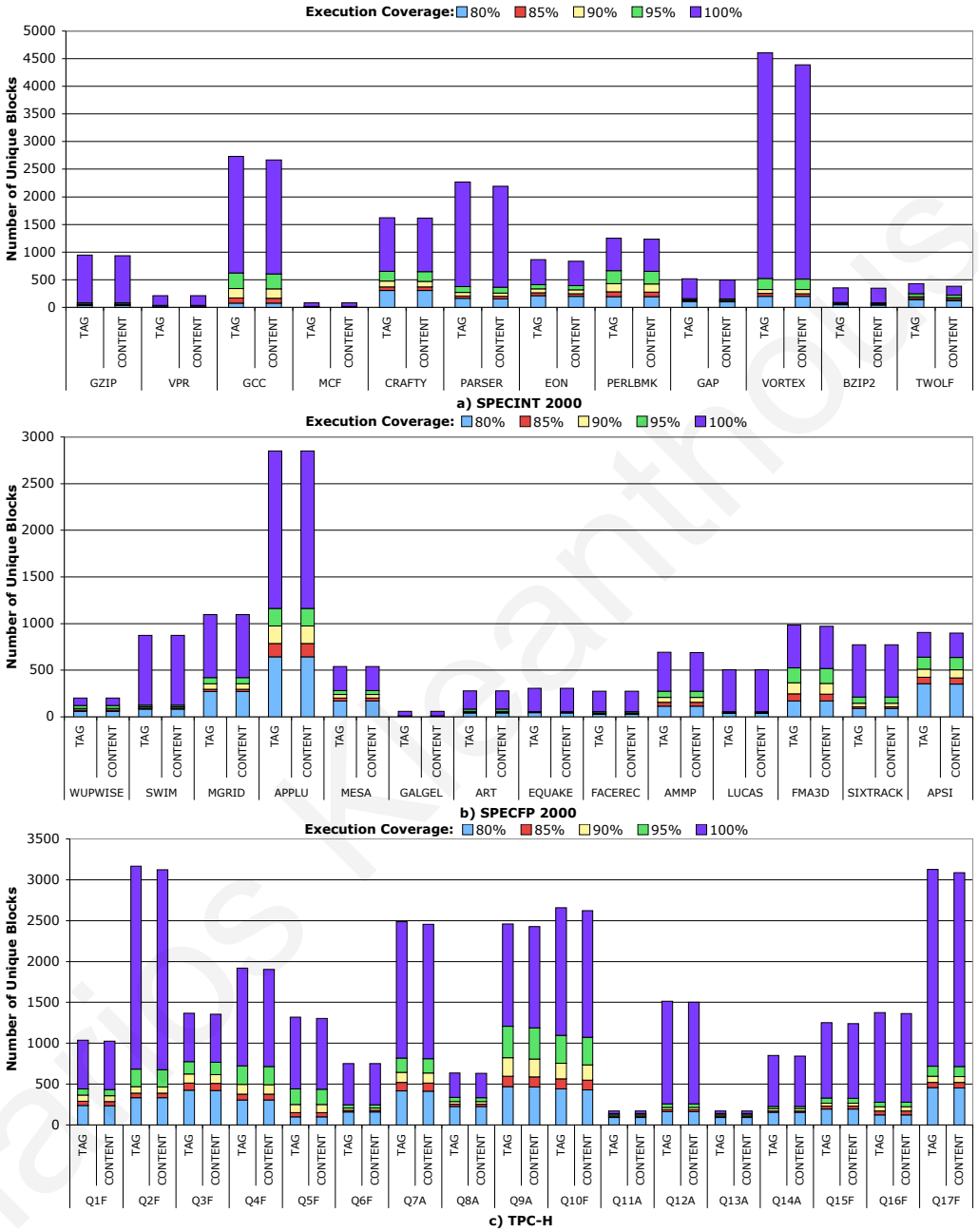

Figure 5: Execution coverage of unique blocks for the a) SPECINT 2000, b) SPECFP 2000 and c) TPC-H benchmarks

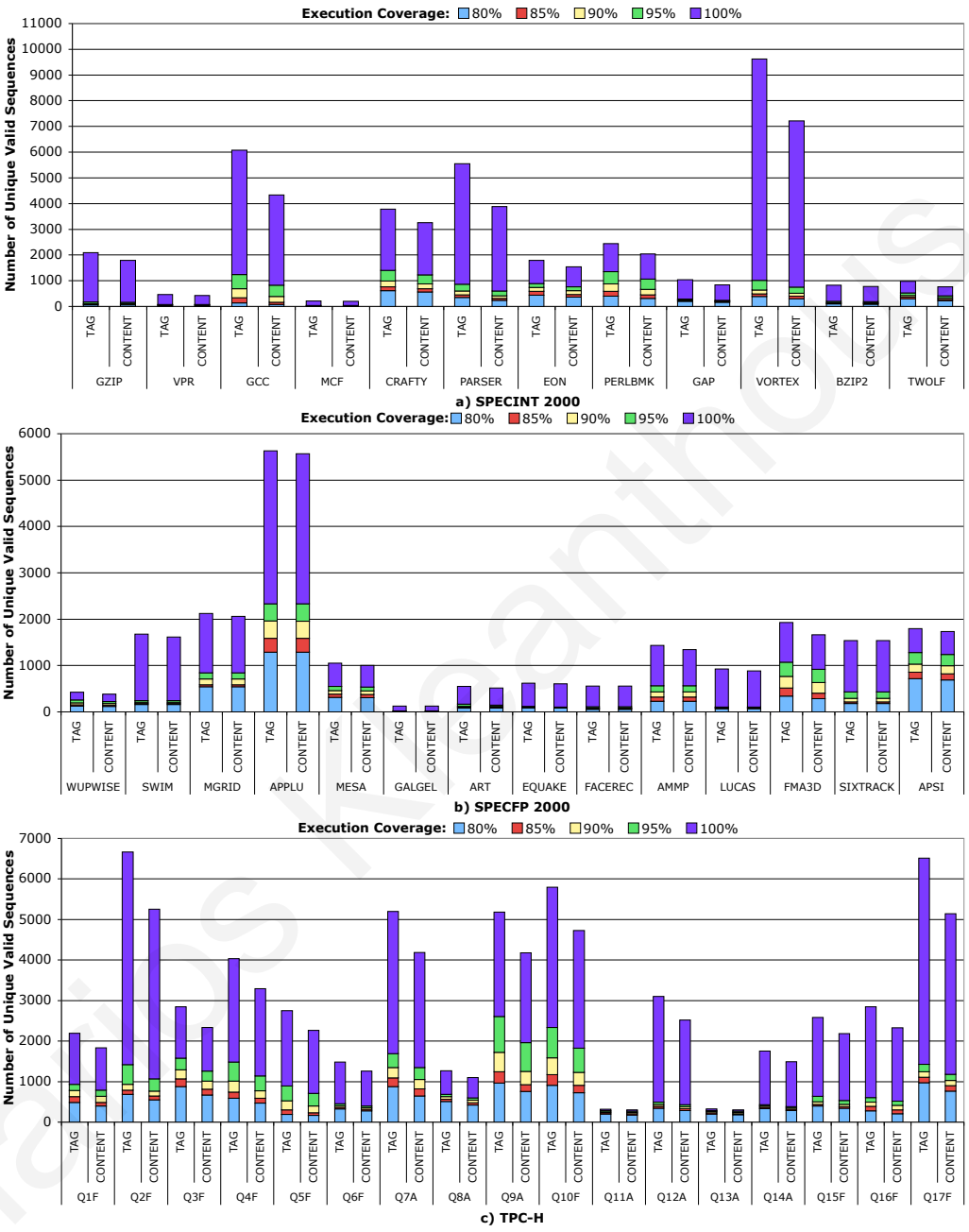

Figure 6: Execution coverage of unique valid blocks for the a) SPECINT 2000, b) SPECFP 2000 and c) TPC-H benchmarks

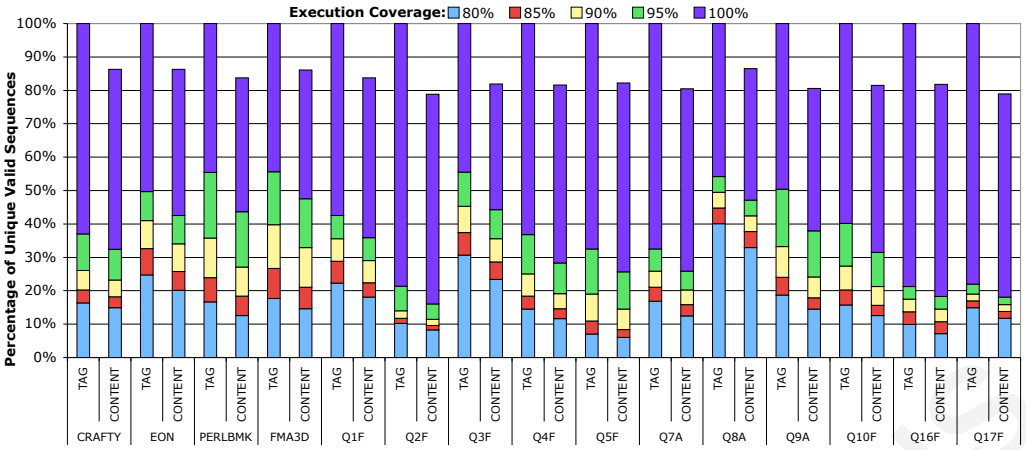

Figure 7: Execution coverage breakdown of unique valid blocks in percentages for a selected subset of benchmarks

Figure 6 shows the number of unique blocks, at the granularity of valid sequences (maximum of 4 instructions), needed to cover a certain amount of dynamic execution. Compared to Figure 5 the results indicate that there is more potential when identifying the redundancy at the granularity of valid sequences than cache blocks. Figure 7 shows the execution breakdown in percentages for few selected benchmarks that have a significant amount of cache pressure. The unique valid sequences identified by their CONTENT are normalized to the total unique valid sequences identified by the sequence TAG. The results clearly show that by removing redundancy, the number of unique valid sequences required for execution is reduced by up to 20% for many benchmarks. For example, the results for Q9A show that we only need 80% of the total unique valid sequences to cover 100% of the execution when we identify them by their CONTENT. Also for the same benchmark, we observe that 50% of the unique valid sequences identified by their TAG are needed to cover 95% of its execution. On the other hand, if we identify the sequences by their CONTENT then only 38% of the total unique valid sequences are required indicating that a significant amount of pressure will be alleviated.

4.3 Limits of Cache-Content-Duplication

The previous section investigated the redundancy during a program's execution. 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 4.1, 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*.

4.3.1 CCD in Instruction Caches for Entire Blocks and Valid Blocks

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

The results show that CCD for entire instruction cache blocks is a rare phenomenon, usually 0-1% of the misses are for duplicated blocks. As it was discussed in Section 4.1 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 effectively lead to identical blocks to appear dissimilar.

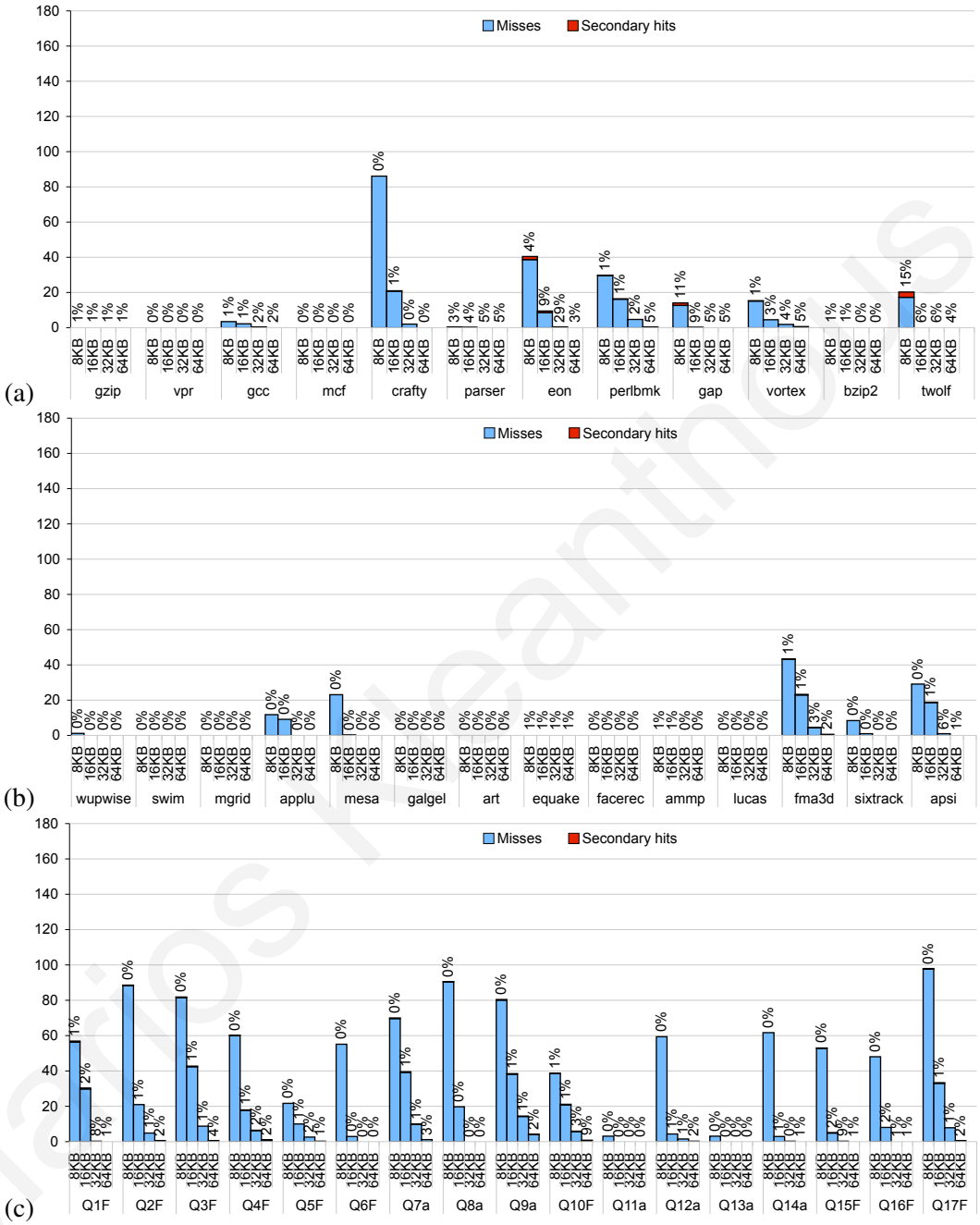

Figure 8: Accesses per 1K instructions. CCD for an 8-way, 8 inst. per block, instruction cache, for entire blocks a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

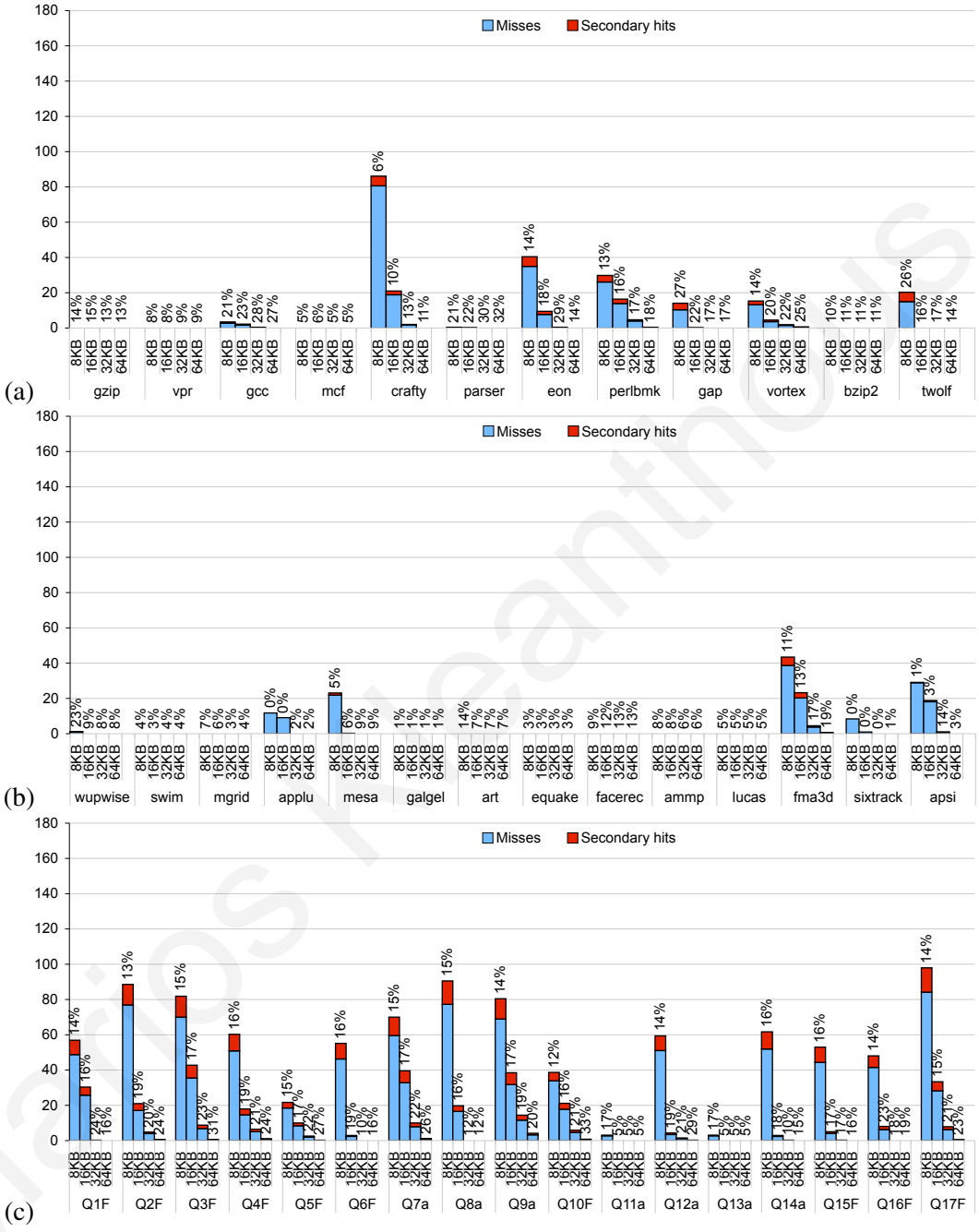

Figure 9: Accesses per 1K instructions. CCD for an 8-way, 8 inst. per block, instruction cache, for valid blocks a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

In Section 4.1 it was suggested that one possible way to overcome the above limitations, and increase CCD rate, is to consider the duplication for valid block.

Figure 9 presents the results for valid blocks. The data show that the CCD rates for valid blocks are often above 15% and therefore more prominent than for entire cache blocks (Figure 8). This increase supports the two claims of Section 4.1 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 Figure 9 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 have a duplicate in the cache.

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

We have also examined the effects of varying associativity on CCD. The frequency and the trends of CCD appear almost the same as with an 8-way cache. The small sensitivity of CCD to associativity may indicate that CCD is not due to conflict misses that can be removed using a victim cache [72]. Section 4.6 presents a more extensive analysis of the associativity effects and 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.

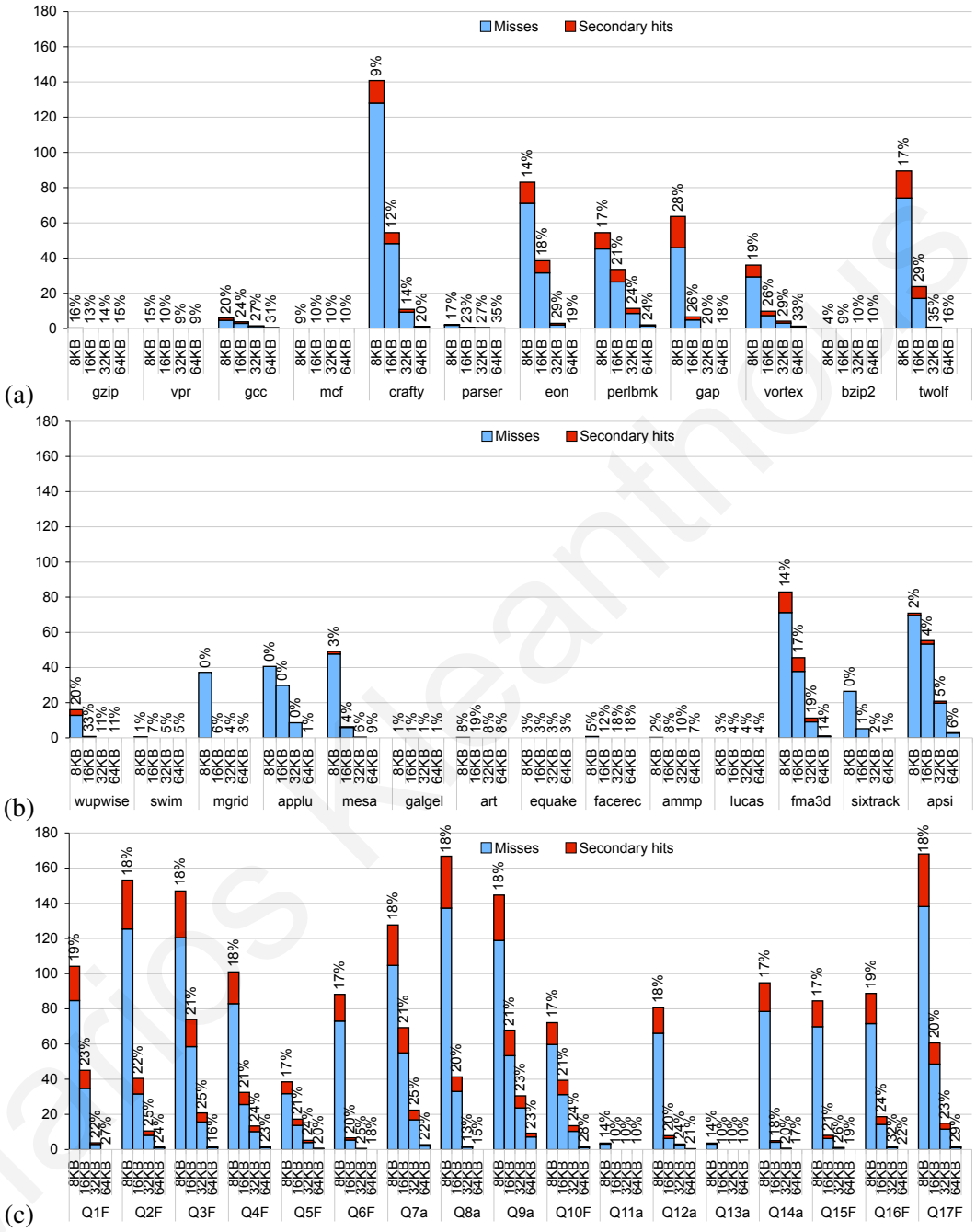

Figure 10: Accesses per 1K instructions. CCD for an 8-way, 4 inst. per block, basic block cache, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

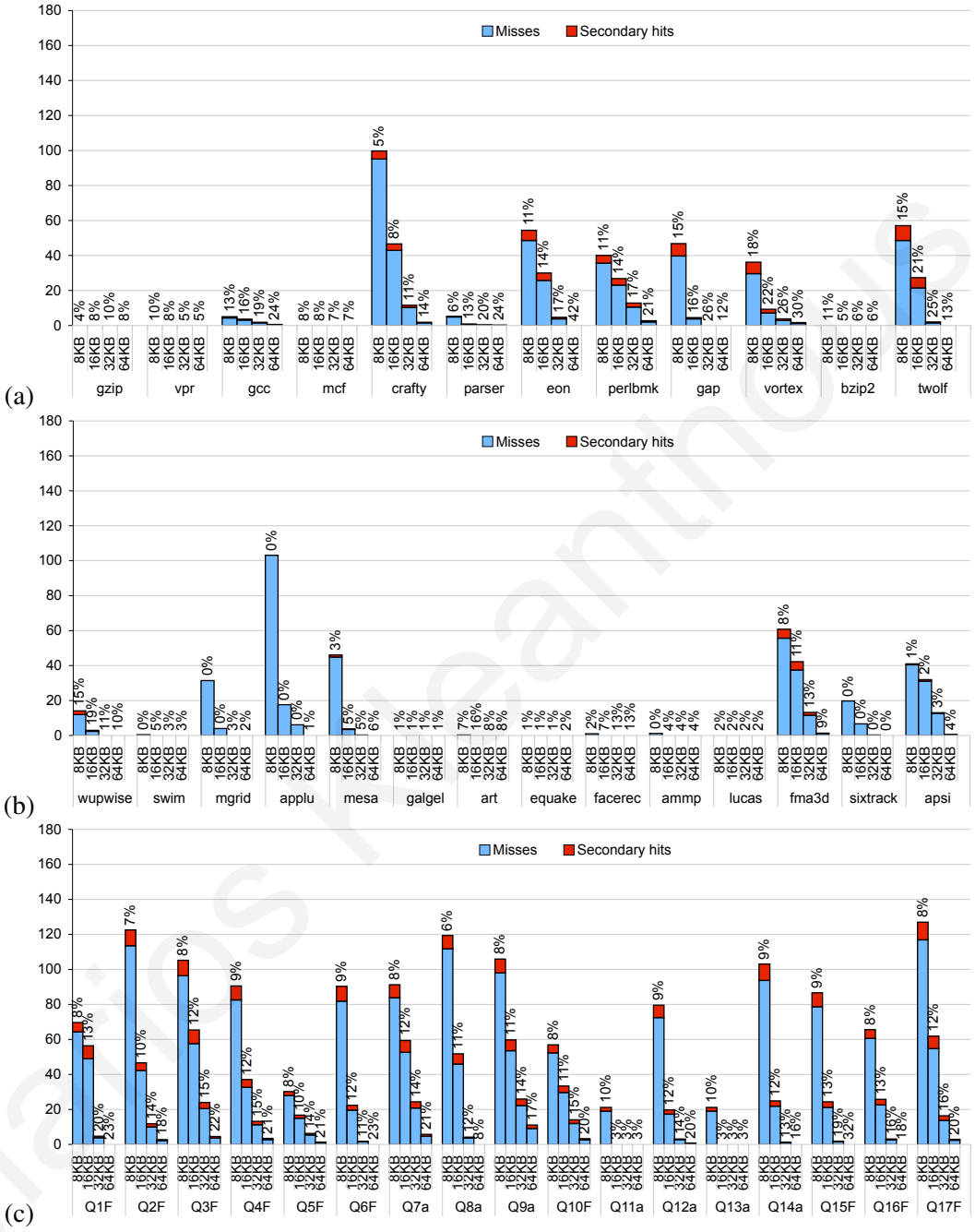

Figure 11: Accesses per 1K instructions. CCD for an 8-way, 8 inst. per block, basic block cache, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

4.3.2 CCD for Basic-Block Caches

Figure 10 presents the breakdown of Accesses per 1K instructions for an 8-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 (Figure 9). The results also show that the duplication rate for several cases is above 20%. 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 higher miss counts (compare total Misses in Figure 9 and Figure 10) and thus more opportunity for missed blocks to be duplicates. For the same reason the SPECINT 2000 and TPC-H benchmarks have higher CCD rates due to more Misses per 1K instructions.

With bigger block size, Figure 11, 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 a higher miss rate.

We have also examined the effects of varying associativity and the CCD trends were very similar with the 8-way basic-block cache. This reinforces the hypothesis that CCD can not be eliminated with a victim cache.

4.3.3 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. Figure 12 presents the breakdown of Accesses per 1K instructions for an 8-way trace cache with 32B (8

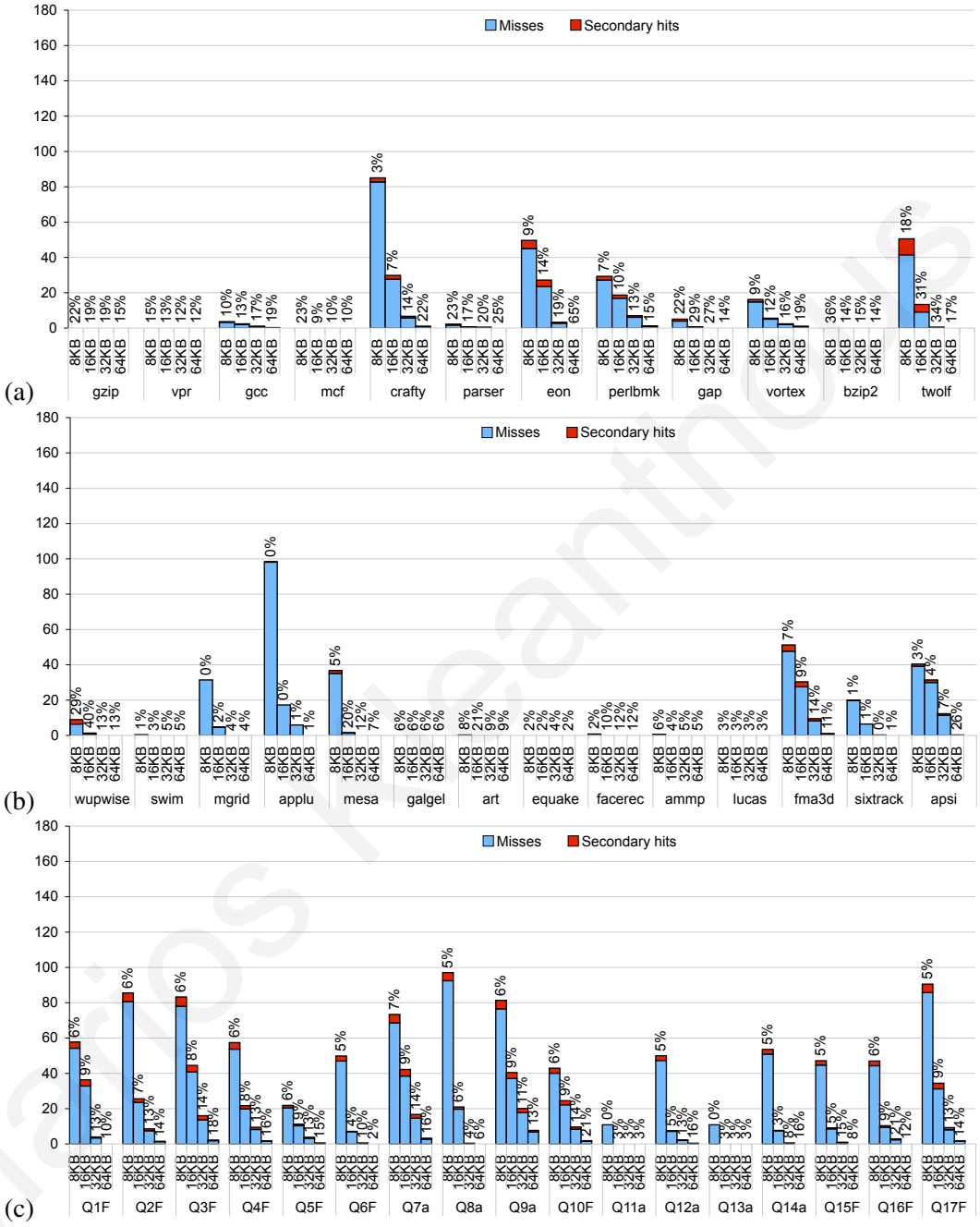

Figure 12: Accesses per 1K instructions. CCD for an 8-way, 8 inst. per block, trace cache, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

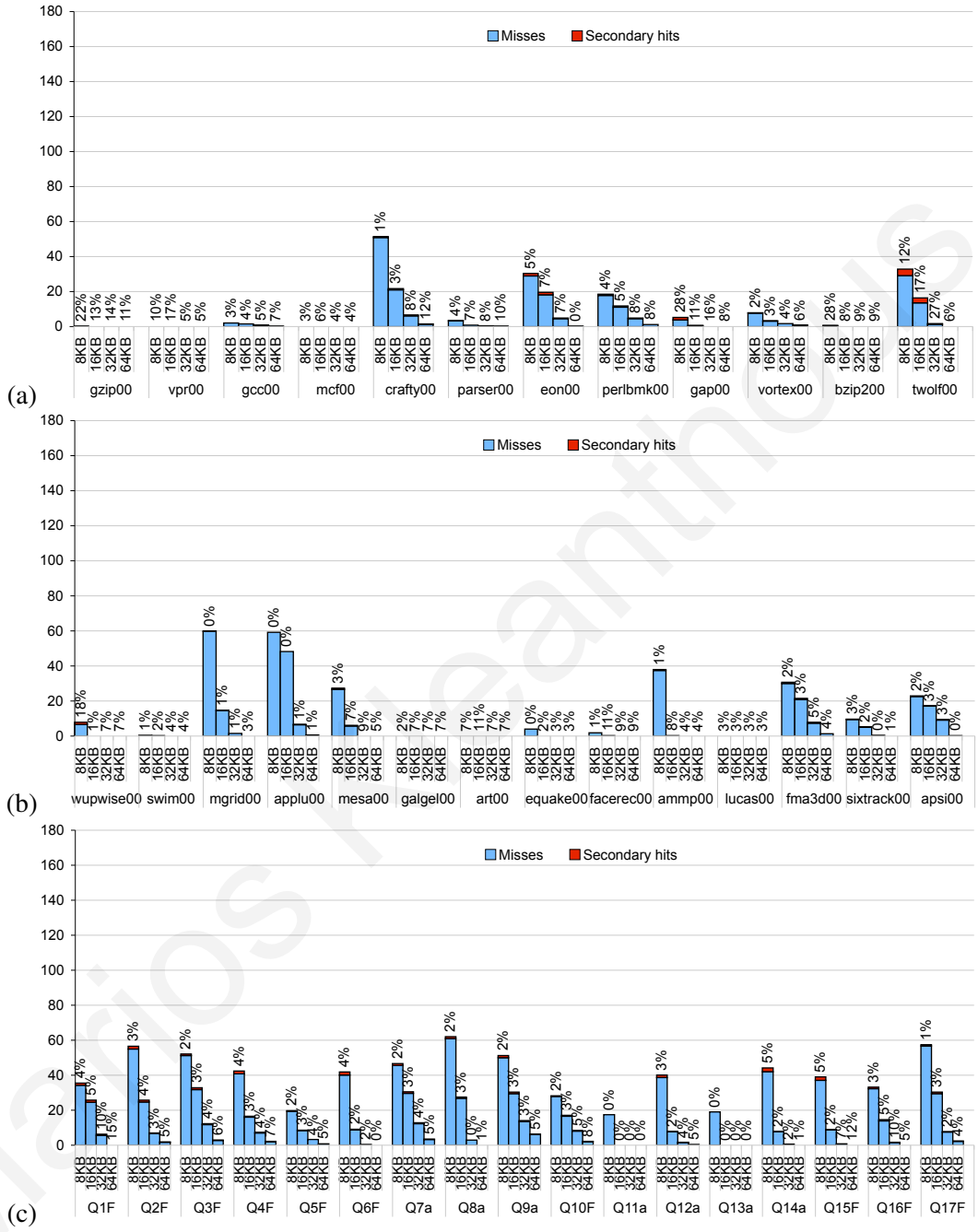

Figure 13: Accesses per 1K instructions. CCD for an 8-way, 16 inst. per block, trace cache, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H

instructions) block sizes. The data show CCD to exist in most benchmarks and often with rates above 15%. Its frequency is comparable to CCD for valid blocks in an instruction cache (Figure 9) but lower than a basic-block cache (Figure 10). The CCD for traces with sixteen instructions, Figure 13, 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.

4.3.4 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 TPC-H benchmarks and lower for SPECFP 2000 benchmarks.

We believe that the observed CCD rates provide a motivation to explore the development and performance of mechanisms that can exploit CCD. The focus of the remaining chapter is on the CCD for regular instruction caches, for valid blocks, because these are the most widely used instruction caches in computing systems.

4.4 CCD Applications: DAC and UCC

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

Cache latency can be reduced through the detection of misses to blocks with a duplicate in the cache and by fetching the block from the cache instead of reading it from lower in the memory hierarchy. We refer to such cache as the Duplicate-Aware-Cache (**DAC**). Therefore, a DAC can reduce the miss penalty of a duplicated miss down to a cache hit. Because the latency of a duplicated miss is likely small, henceforth, we refer to it as a secondary hit (primary hits are those

that hit directly in the cache). All accesses that are neither primary nor secondary hits are misses that need to be serviced from a lower level cache. A DAC cache, when compared to an otherwise identical regular cache, is expected to have as many primary hits as the hits of the regular cache, but have some of the regular cache misses converted to secondary hits. Therefore, in the presence of CCD a DAC can only improve performance. Another benefit of DAC is a reduction in the traffic to lower levels of memory hierarchy because the missed block can be read directly from the L1 Cache and inserted to the correct set. Note that for DAC, in the case of CCD for valid blocks there is no traffic reduction because the entire block is always fetched on a miss. Overall, the amount of improvement from DAC mainly depends on the number of the regular cache misses it converts to secondary hits.

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

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

4.4.1 Limits of the Cache-Content-Duplication

An indication of the performance potential of a DAC, over a regular cache, is given by the fraction of misses that have a duplicate in the cache. These results are shown in Figure 9 for an instruction cache for valid blocks.

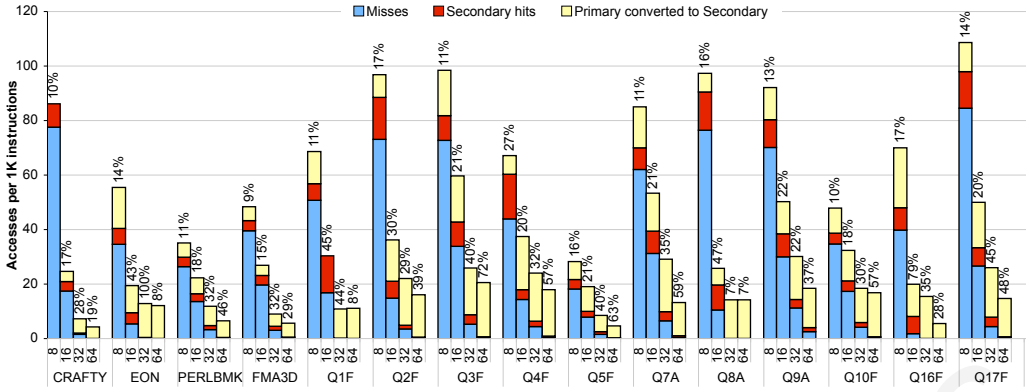

Figure 14: Misses and Secondary hits per 1K instructions breakdown and CCD rates for a UCC 8-way, 8 instructions per block, instruction cache, for valid blocks

To establish the potential of a UCC cache over a regular cache we performed an oracle study with the same assumptions as in Section 4.3. The UCC cache is modeled as a regular cache unless 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.

Figure 14 shows the breakdown of accesses per 1K instructions for a UCC-instruction cache for valid blocks. The graphs show, for comparison purposes, the secondary hits that were initially primary hits, for the respective regular cache, labeled as "Primary converted to Secondary". The graphs also include the numeric values for the CCD-rates that correspond to baseline misses converted to secondary hits (without considering the "Primary converted to Secondary").

A comparison of Figures 9 and 14 reveals that for most benchmarks and cache configurations the CCD rates for UCC caches are higher than their corresponding DAC caches. For example, for a 16KB cache in Figure 14 crafty has 17% CCD rate where its corresponding rate for DAC is 10% (Figure 9). The reason for this increase, is that UCC avoids the insertion of duplicate content in the UCC cache and eliminates capacity and may be conflict misses. This reduces the total number of misses by more than the duplicate misses of DAC.

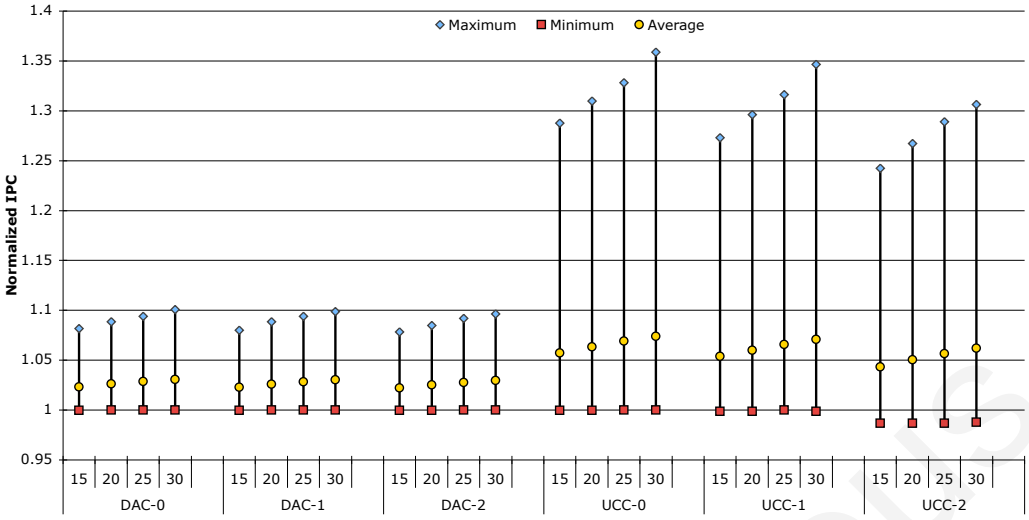

Figure 15: Maximum, minimum and average of the normalized IPC performance of all benchmarks for DAC and UCC for valid blocks. Results are shown for 15, 20, 25 and 30 cycles L2 latencies and 0, 1 and 2 cycles secondary hit latencies

However, the data also show that a UCC cache can have fewer primary hits than a regular cache. For example, for a 16KB cache in Figure 14, crafty has 3.7 access per 1K instructions that were converted from primary hits to secondary hits. Therefore, only 3.5 out of the 7.2 secondary hits per 1K instructions correspond to cache misses converted to secondary hits. The above suggests that a UCC cache, unlike DAC, sometimes may not improve the performance because a decrease in primary hits can offset the benefits of CCD. So, to compare the performance potential of DAC and UCC a study for an out-of-order processor is performed.

4.4.2 Performance Potential of CCD

The Figure 15 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 an oracle CCD detection under the same assumptions as in Sections 4.3 and 4.4.1.

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

The data show both DAC and UCC to have performance potential up to 10% and 36% respectively. The analysis indicates that the TPC-H benchmarks with the highest CCD rates, see Figures 9 and 14, are also the ones with the largest potential while most of the SPEC2000 benchmarks do not benefit from CCD because they have very few misses.

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

The lower UCC performance for 2 cycles secondary hit latency suggests that the performance gains due to the miss reduction of UCC are outweighed by the penalty for having some primary hits converted to secondary hits. Another observation is that, although the limits of CCD rates for DAC and UCC are very similar, as shown in Figures 9 and 14, the results in Figure 15 show that UCC is much better in many configurations. This occurs because in the DAC limit study we

assumed no latency for fetching a block from a lower level in the memory hierarchy and thus in for two consecutive accesses to the same missed block (for different valid blocks), the first would be a secondary hit and the second would be a primary hit. But in a realistic scenario, with a fetch delay, it is possible to have a secondary hit and the next access to the same block to cause a miss because the block is not yet fetched from the L2 cache.

A zero cycle secondary hit latency is possible, but may require more pervasive changes in the processor front-end. This is discussed more extensively in Section 4.5.8. The single cycle secondary hit latency can be achieved by accessing the CCD mechanism and cache in parallel. By the end of the tag array access, assuming 1 cycle, the CCD mechanism will provide an alternative tag-index to access the cache again in case of a miss. Finally, for serially accessing the CCD mechanism after a cache miss a 2 cycle secondary hit latency is required. The first cycle is spend on a tag array access to discover the cache miss, and the second cycle to access the CCD mechanism and provide an alternative tag-index.

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

4.5 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 for instruction caches. We will refer to this mechanism as CATCH. Recall that

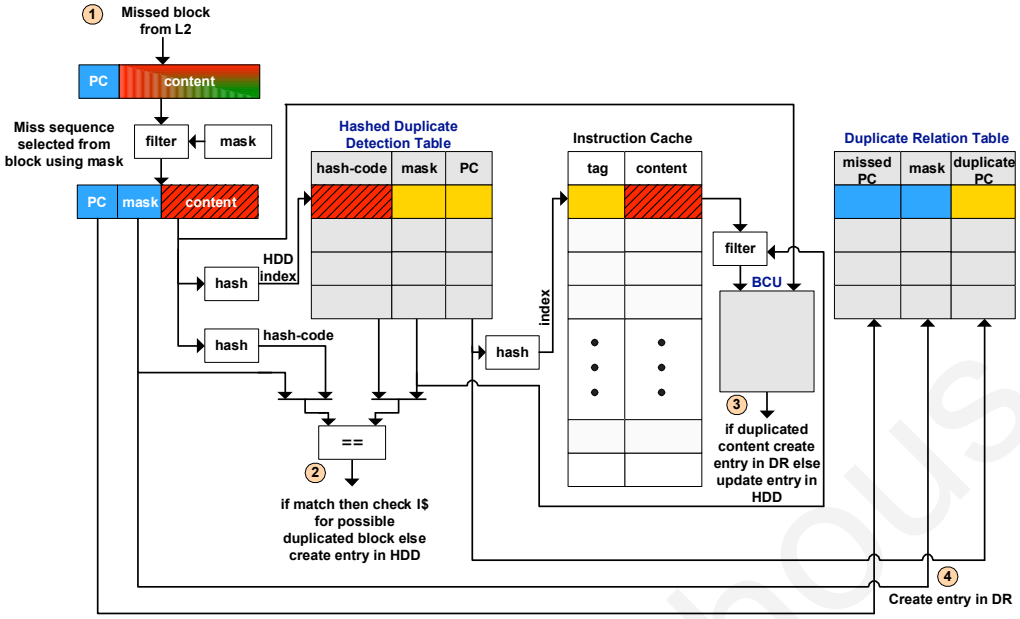

Figure 16: The CATCH flow for a Cache miss, DR miss and HDD hit

valid blocks in instruction caches are identified with their starting PC and a bit mask provided by the branch predictor (see Section 4.1).

The microarchitecture of a cache with a CATCH is shown in Figure 16. It includes the Hashed-Duplicate-Detection table (HDD), the Block Compare Unit (BCU) and the Duplicate-Relation table (DR). The functionality of the different components and their updating policies are the subject of this section.

4.5.1 Hashed-Duplicate-Detection table

The detection of CCD requires a mechanism that, given the content of a block, it provides a starting PC and a mask for a candidate duplicate-block currently in the cache.

The Hashed-Duplicate-Detection table (HDD) provides this functionality. Each entry in the HDD contains a hash-code, which encodes the content of a block, and the corresponding starting PC and mask of the valid block. The use of a hash-code reduces the cost and complexity of detecting duplication but may lead to unnecessary tests for duplication. However, we found that a

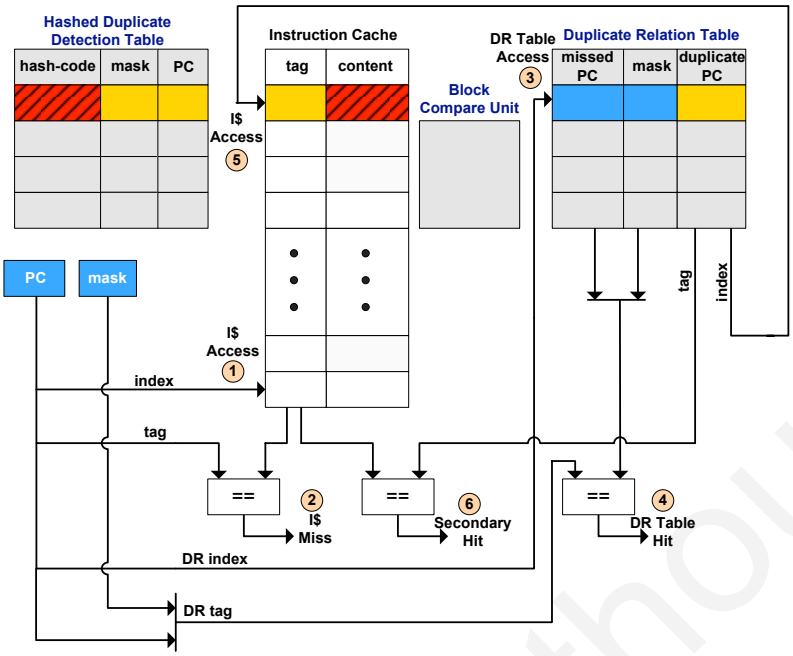

Figure 17: The CATCH flow for a Cache miss, DR hit, Cache hit

simple folding of the valid block content to 16 bits provides very accurate encoding (often 99.9% accurate).

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

When a missed block's hash-code and the hash-code in a valid HDD entry match, we may have content duplication. Here, the cache is accessed using the starting PC found in the HDD to determine whether the two valid blocks are indeed duplicates. The Block Compare Unit (BCU) performs the test for duplication. If the BCU indicates that the blocks are duplicates then an entry is created in the DR. Figure 16 illustrates the sequence of steps in the case of a cache miss that has a duplicate in the cache but not an entry in DR. This process is similar for DAC and UCC.

4.5.2 The Block Compare Unit

When two blocks are signaled by the HDD as possible duplicates, their contents are compared using the Block-Compare Unit (BCU) to detect whether there is indeed duplication. The compare function used in the BCU can be a simple bit-wise 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 4.5.6.

4.5.3 Duplicate-Relation table

The Duplicate-Relation table (DR) contains relations between duplicated blocks detected by CATCH. An entry in the DR is created when a block with a cache miss is fetched from a lower level cache and is found to be a duplicate with a block already in the cache using the HDD table and BCU unit.

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

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

On a cache miss, the DR is accessed with the starting PC and mask of a missed block. When there is a DR hit and the duplicate PC hits in the cache, a secondary hit occurs. In the case of a DAC, the content of the missed valid block will be read and a request in a lower level cache for the *entire missed block* will be initiated in parallel. For a UCC, only the content of the duplicate-block will be read and no miss will be requested from a lower level of the memory hierarchy. Figure 17 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.

4.5.4 Allocating and Updating an HDD and a DR entry

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

1. **Cache miss, DR miss, HDD miss:**

   A valid block is a miss in the cache and no entry in the DR matches its starting PC and mask. The block is fetched from a lower level of memory hierarchy, its content's hash-code is calculated and then HDD is accessed with this hash-code. On a miss a new HDD entry is created.

2. **Cache miss, DR hit, Cache miss, HDD miss:**

   Same as above unless there is a DR hit that leads to a cache access and misses because the duplicate block was evicted. If we miss in the HDD then an entry is allocated and points to the fetched block in the cache.

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

1. **Cache miss, DR miss, HDD hit:**

   A block is a miss in the cache and the DR. The block is fetched from a lower level in the

memory hierarchy and its hash-code is calculated. The HDD is accessed with the hash-code. If we hit in the HDD then the cache is accessed with the duplicate-PC. The two block contents are compared and if they match, a DR entry is created with the missed starting PC and mask, and the duplicate-PC pointed by the HDD. Also, the HDD entry is updated to point to the fetched block in the cache (the implications of not-updating the HDD in this case are discussed in Sections 4.5.6). When the content of the missed block and the one pointed by the HDD do not match in the BCU, we have a case of a false hash-code match. This was found to occur very rarely for hash-codes of 16 bits. When this happens, the HDD entry will be updated to point to the missed block.

2. **Cache miss, DR hit, Cache miss, HDD hit:**

   Same as 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.

4.5.5 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 in an invalidation of the HDD entry. On the other hand, for a UCC the block is first checked for duplication and only if the HDD cannot detect any duplication will the block be inserted in the cache.

4.5.6 Performance Optimizations

This section describes two types of performance optimizations for CATCH. The first optimization is to tolerate simple differences between blocks by using a more advanced compare function in the BCU. The *keep_offset* optimization aims to increase content-duplication by masking out, from the compare process in BCU, the offsets and targets of conditional and unconditional direct branches, and keeping in the DR the offsets and targets of each duplicate block. This aims to convert blocks that contain exactly the same computation into duplicates. This is effectively a hardware implementation of the target abstraction discussed in Section 4.1.3. Two possible caveats of this optimization are the extra cost per DR entry, and that secondary cache reads may need to combine information from the cache and the DR which may make fetching more complicated. The first is considered for the total size of the mechanism calculated in Section 4.5.7 while the second can be accommodated in the valid block masking logic (Figure 4).

Other examples of possible BCU optimizations are to augment the compare function to rearrange source operands of commutative operations and reorder data independent instructions in a block to facilitate content duplication [24]. These and other transformations to be discovered may help uncover even more duplication, but this is to be considered in future work.

The second performance optimization is to filter the updates in the HDD and DR tables by avoiding the insertion of entries that are unlikely to have a significant payoff. 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. [73]. The filtering is accomplished by allowing a table to be updated every n attempts. This policy works because it can prevent rare events from entering the tables, whereas persistently occurring events will eventually make it into the table. For an extended discussion on how this

method works we refer the interested reader to [73]. Based on simulation results for various filtering strategies it was found that the best was to filter only the updates of the HDD and the filter value should be four, i.e. updating the HDD every fourth attempt. Although the DR is not filtered directly, by updating the HDD less frequently, the updates to the DR are indirectly reduced.

The significance of the *keep_offset* and the filtering optimization is investigated in Section 4.6.

4.5.7 Cost Reduction Optimizations

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

Before computing the cost for a DR entry, recall that a DR entry represents logically two full tag-indices. For the Alpha instruction set architecture [65] used in this work, the first tag-index contains 30 bits (28 bits for the address of the first instruction of the missed sequence and 2 bits for the mask, which is the number of valid instructions in the sequence), and the second tag-index contains 28 bits for the address of the first instruction of the duplicate sequence. The second tag-index does not require a mask because it must be the same with its duplicate sequence for a duplicate relation to exist. The non-optimized cost of a DR entry is therefore:

$$(2 * 28 + 2 - \log_2(numberofsetsinDR)) \text{ bits}$$

which is the sum of the two addresses and the length of sequence minus the index of DR.

After some cursory analysis it was observed that usually the 9 leading bits of the starting PC of the missed and duplicate valid block are the same. This reduces the cost of a DR entry by 9 bits if only the entries that satisfy this criterion are inserted into the table.

When the *keep_offset* optimization is employed, the DR should keep a maximum of four direct targets. To reduce the number of bits required by the offsets and direct targets, extra insertion

criterion can be used. Specifically, duplicated relations are inserted when the following are true: (a) valid blocks have at most one control flow instruction and (b) the upper 10 bits of direct targets must be the same with valid block's starting address. Note that for the ISA used in this study target offsets for conditional branches are 16 bits and direct targets are 21 bits. With these criterions in place, the extra cost of the *keep_offset* optimization is 11 bits for each DR entry, for one offset or one target.

Therefore for the DR, the per-entry cost with cost optimizations is:

$$(28 + 19 + 2 - \log_2(numberofsetsinDR)) + 11 \text{ bits.}$$

An HDD entry contains a hash-code, the PC and the mask of the duplicate block. For the limit study we assumed a 32-bit hash-code but further analysis indicates that a 16-bit hash-code causes false-hash-matches very rarely. So, in Section 4.6 we consider the performance with a 16-bit hash-code. Furthermore, we can use the hash-code used for tag-matching the valid blocks to index the HDD. This will reduce the HDD entry by $\log_2$(number of sets of the HDD) bits. Also, the criterion used in DR (the 9 most significant bits of the two tag-indices must be the same) can be used here also. That means we only keep the 21 least significant bits in the HDD and combine them with the 9 most significant bits of the missed valid block to create the index-tag and access the cache.

Therefore for the HDD, the per-entry cost with cost optimization is:

$$16 + 19 + 2 - \log_2(numberofsetsinHDD) \text{ bits.}$$

Finally the replacement policy for HDD and DR is assumed to be tree based pseudo LRU [74] that requires N-1 bits per set, where N is the associativity of the structure. In Section 4.6, we compare the performance with and without the cost optimizations.

4.5.8 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 aside from 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 4.5.

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

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

4.6 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 4.4) 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, for valid blocks, that is 8-way, 8 instructions per block, with a single cycle secondary hit latency in addition to the L1 hit latency and 20 cycles L2 cache latency. In order to make the figures more readable, we show results only for the 15 benchmarks with the higher Misses per 1K that offer more opportunity for performance improvement. We also include the average for the remaining 28 benchmarks (average-other) and the average of all 43 benchmarks (average-all). For the 28 benchmarks not shown, we verify that the worst case degradation is 0.1% for the TWOLF benchmark.

4.6.1 CATCH Performance for DAC and UCC Caches

Figure 18 shows the normalized performance potential captured by DAC and UCC with an oracle CCD detection (same as in Figure 15) and the normalized performance potential captured by DAC and UCC using the CATCH.

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

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

One interesting observation from Figure 18 is that for a benchmark, Q16F, the UCC performance of CATCH is slightly higher than the oracle UCC results. This happens due to the "failure" of a real HDD to maintain the hashed content of all valid blocks in the cache. This results in duplicated content to be inserted in the cache. The data show this duplication to be beneficial to performance.

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

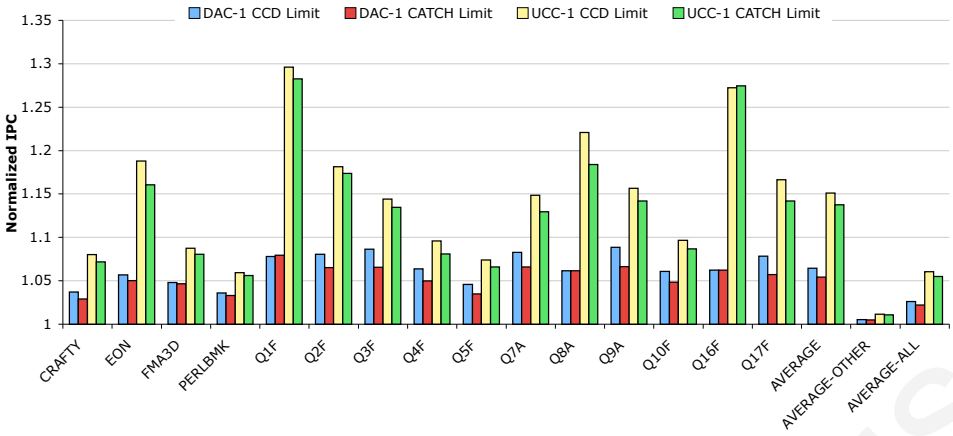

Figure 18: Performance potential captured by oracle detection (limit) and CATCH for DAC and UCC (16KB instruction cache, 20 cycles L2 cache latency)

phenomenon is analyzed later where its effects are more prominent when the size of the CATCH is reduced further.

For the remainder of Section 4.6 we focus on optimizing the performance of the UCC instruction cache due to its higher performance potential compared to DAC.

4.6.2 CATCH Performance

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

Figure 19 shows the performance of CATCH compared to a limit study (CCD Limit) with oracle CCD detection. “CATCH Limit” corresponds to a CATCH implementation with unbounded DR and HDD tables. A design space exploration analysis (APPENDIX A), for various number of entries and associativities, suggests that a 4-way 128 entries DR and an 8-way 128 entries HDD represent a good performing CATCH configuration. This configuration (3.05KB CATCH) can provide an average IPC improvement of 7.5% for the 15 selected benchmarks, and 3% over all 43

benchmarks, which corresponds to 50% of the performance potential of a UCC with oracle CCD detection (Figure 19). Note that this CATCH configuration has 3.05KB state cost and employs all the performance optimizations but none of the cost optimizations.

To reduce the state cost of CATCH we applied the various cost optimizations discussed in Section 4.5.7. This lead to a reduction in CATCH cost to 1.38KB, with negligible performance degradation for few benchmarks, less than 1%, but with an improvement of 0.4% overall and 1.2% over the 15 selected benchmarks as shown in Figure 19. The 1.38KB CATCH can provide 8.7% improvement for the 15 selected benchmarks, and 3.4% over all 43 benchmarks, which corresponds to 58% of the performance of UCC with an oracle CCD detection.

Figure 19 also quantifies the significance of the performance optimizations, discussed earlier in Section 4.5.6, on the 1.38KB CATCH. The results, for the 15 selected benchmarks, show that without filtering (1.38KB CATCH no filter) the performance degrades by 1% on average, without learning an additional valid block on a miss (1.38KB CATCH learn on miss) the degradation is 2% on average and without the target abstraction (1.38KB CATCH no keep offset) the performance benefits are reduced by 1.5%.

An interesting observation is that, sometimes, the smaller 1.38KB CATCH provides better performance than the 3.05KB CATCH. For example, benchmark Q16F shows an increase of 4% with smaller CATCH. Analyzing the benchmark further reveals a reduction in secondary hits due to duplicated blocks that enter the cache. This seemingly undesirable behavior can benefit sometimes performance. Particularly, some of these blocks also contain non-duplicated valid sequences that are referenced in the near future and become cache hits. This suggests that an adaptive filtering mechanism may benefit performance further by exploiting the above phenomenon more effectively. This represents a possible direction for future work.

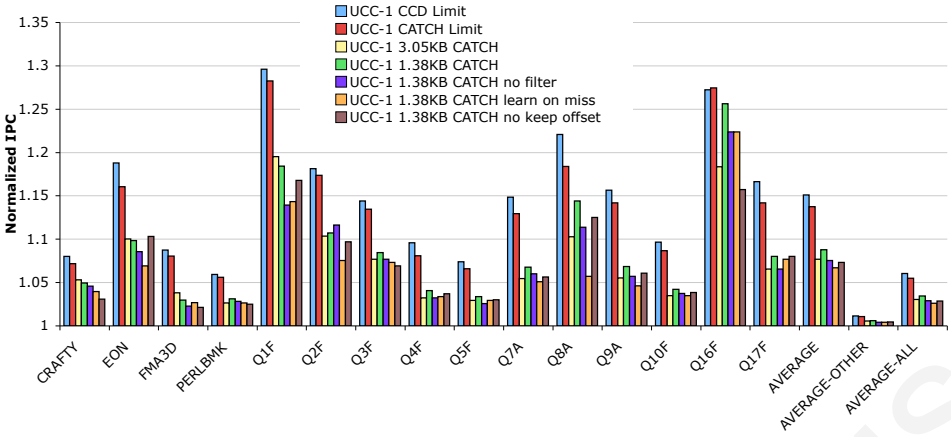

Figure 19: Effects of applying different policies on CATCH performance

4.6.3 Effects of Associativity

The functional simulations showed that the CCD rates are insensitive to associativity and the miss rates were slightly affected. Figure 20 shows the normalized IPC of each baseline cache to the same cache with the addition of CATCH, for example for the 2-way bar the results are for a 2-way cache using CATCH and the baseline is a 2-way cache without CATCH. The results indicate that on average the performance improvement of CATCH is not affected by the associativity. It is interesting that the average for the low miss rate benchmarks, average-other, shows a performance degradation as the associativity increases. This happens because higher associativity means less cache misses and lower potential for the CATCH to improve.

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

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

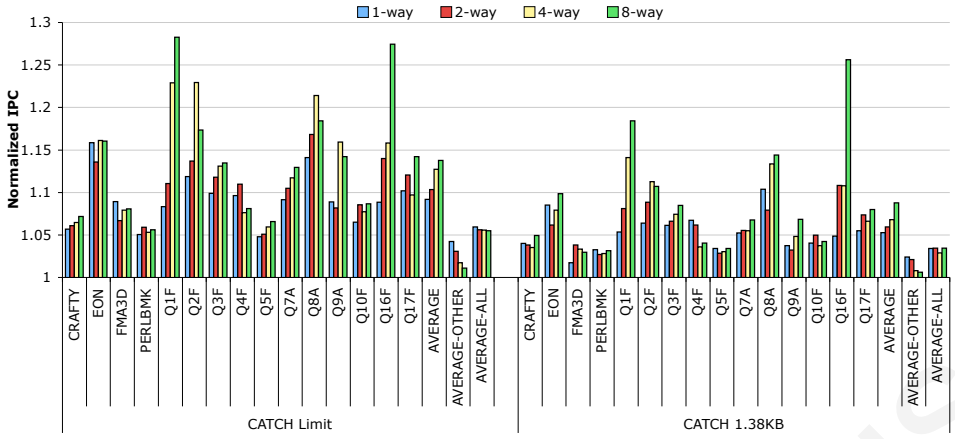

Figure 20: CATCH with various cache associativities

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

The above observations suggest that a balance must be kept between the duplicated blocks allowed in the cache and the associativity of the cache. From our experiments it appears that an 8-way associative cache can solve this problem most of the times and filtering techniques, like the one presented in Section 4.5.6, can almost eliminate the problem.

4.6.4 Effects of Cache Size

Section 4.3 showed that the CCD rates increase as the cache size increases because there is more opportunity to find duplicated blocks when you have more blocks to compare. Figure 21 shows the normalized IPC of each baseline cache to the same cache with the addition of CATCH, for example for the 8KB bar the results are for an 8KB cache using CATCH and the baseline is an 8KB cache without CATCH. The results indicate that on average the performance improvement of

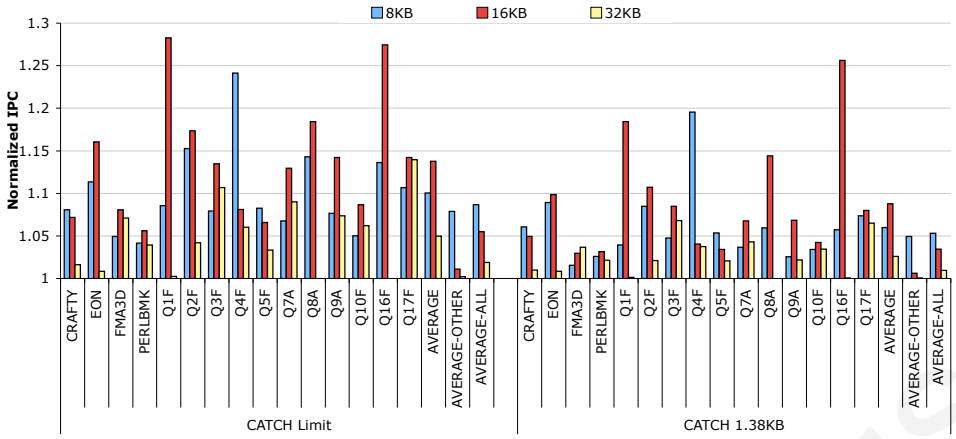

Figure 21: CATCH with various cache sizes

CATCH is reduced as the cache size increases. This is due to the low miss rate that is completely eliminated with a cache bigger than 16KB for most benchmarks. However, for the few high miss rate benchmarks, we can see that the CATCH performs better with a 16KB cache compared to an 8KB but for most of the time a 32KB eliminates all misses and thus any room for improvement.

4.6.5 CATCH vs Victim Cache

An alternative mechanism to reduce cache misses is the victim cache [7]. A victim cache aims to reduce cache misses, due to conflicts in a set, by keeping a fully associative structure and maintaining victim blocks there until they are evicted or needed again from the cache. Figure 22 shows the performance improvement of a regular cache using an 8-entry victim cache, the CATCH with 1.38KB cost, and a combination of the two. When combined, the victim cache is accessed first and the CATCH is used only in case of a victim cache miss.

The data show that for three benchmarks, Q16F, Q8A and Q2F, victim cache is better than the CATCH whereas the CATCH is superior for the others. However, the most important observation is that the performance gain from the combination of CATCH and victim cache is additive. This

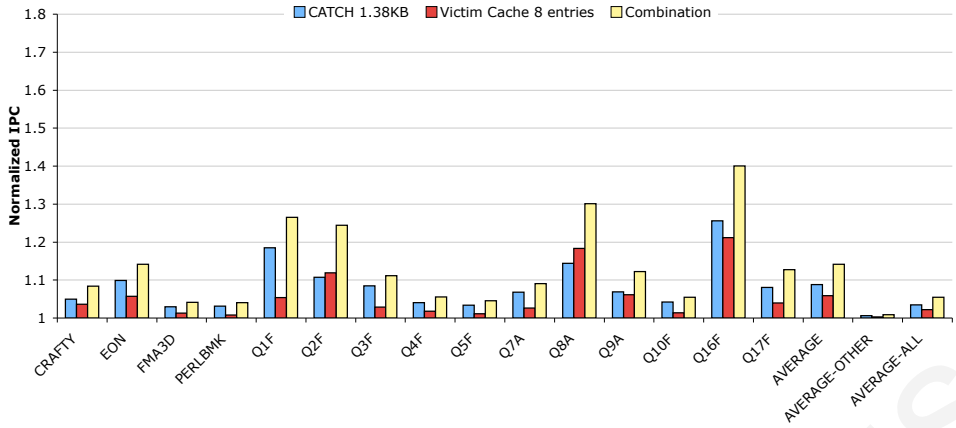

Figure 22: CATCH and 8 entry Victim Cache

indicates that CATCH captures misses that are not conflict misses only in the same set but also across sets.

4.6.6 Effects of Prefetching

Prefetching is another technique to reduce cache misses and improve performance. We have investigated the performance improvement of a simple next-line prefetcher with and without the CATCH. The next-line prefetcher is applied at all cache levels and it was verified that it does not degrade the performance when prefetching data blocks. Figure 23 shows the normalized IPC of the baseline with CATCH, with next-line prefetching and when applying both techniques. The results show that prefetching can significantly improve the performance of a cache but again, as with the victim cache, the performance improvement is additive for CATCH. Furthermore, there is one benchmark, Q16F, that prefetching cannot improve its performance while CATCH can increase its IPC by 25%. This suggests that there are cases where a simple prefetcher cannot predict the program behavior but the redundancy still exists in the cache and can be eliminated using CATCH.

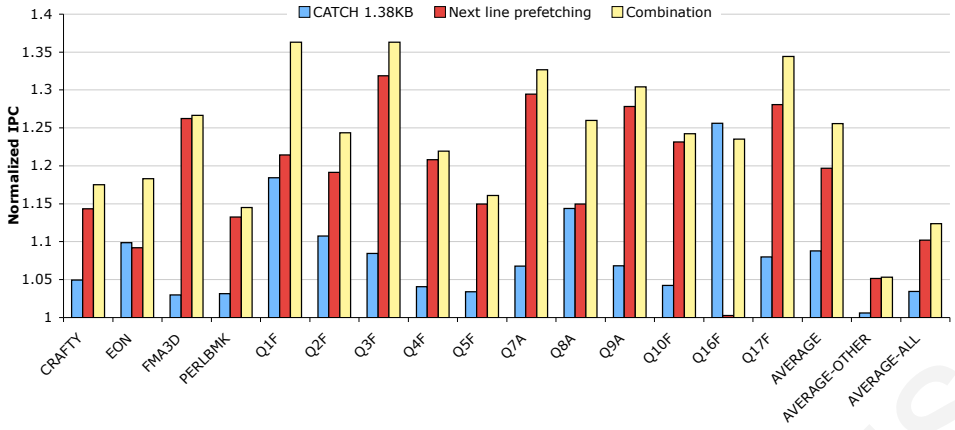

Figure 23: CATCH with next-line prefetching

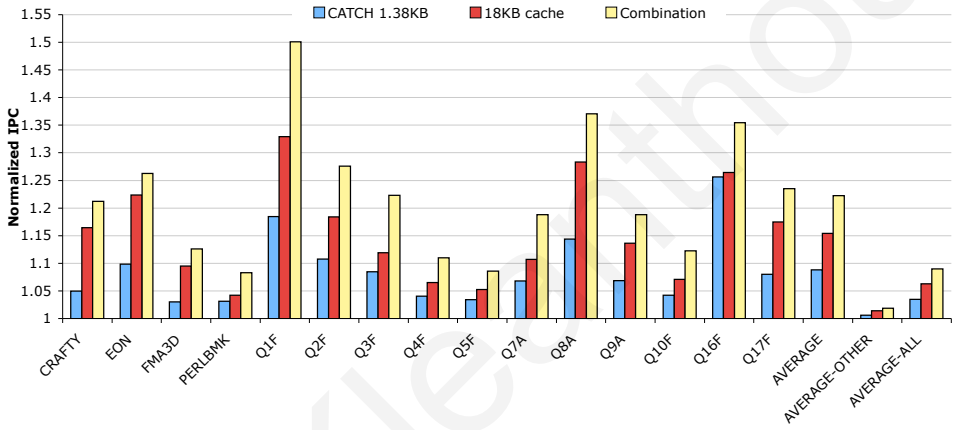

Figure 24: CATCH compared to an 18KB cache

4.6.7 Increasing Cache Size

Another design tradeoff is to consider investing the extra space required by CATCH to increase the cache size. For example 16KB cache + 1.38KB CATCH can roughly correspond to an 18KB cache which has the same design specifications as the 16KB cache + 1 extra way. Figure 24 shows the normalized IPC of the baseline with CATCH, the 9-way 18KB cache and a combination of both the 18KB cache and CATCH. The results indicate that a cache with 2KB extra way provides higher performance than the 1.38KB CATCH. However, it is worth mentioning that even with the extra space there is still room for improvement using CATCH. This is indicated by the extra 7%

Table 6: Energy consumption per access of 16KB 8-way and CATCH

Structure	Dynamic Energy per access(nJ)	Percentage of Cache energy
Cache 16KB 8-way	1.09001	100%
HDD 0.52KB 8-way	0.00479044	0.44%
DR 0.86KB 4-way	0.00380342	0.35%
Block Compare Unit	0.00135553	0.12%

on average improvement that can be achieved using a combination of the 18KB cache and the 1.38KB CATCH compared to the 18KB cache alone.

4.6.8 CATCH Energy Consumption

Table 6 shows the dynamic energy per access, obtained using CACTI [64] of the three structures used to implement 1.38KB CATCH. This is compared with the energy of the 16KB 8-way cache used for the performance evaluation. The results were taken using the exact same array configuration and feature size for all the structures excepts of the number of banks which for HDD and DR where only one bank and for the cache we used four banks.

The table shows that the energy consumption of the HDD corresponds to 0.44% of the cache energy per access while the DR consumption corresponds to 0.35%. The energy consumption of the Block Compare Unit, that compares a maximum of 128 bits, 4 instructions, corresponds to 0.12% of the energy that it is consumed during a cache access.

It's worth noting that the energy consumed by HDD and DR is not proportional to their size as compared to the Cache energy per access. The reason is that Instruction L1 caches are accessed in parallel mode to be fast. That means both the tag and data array are accessed and all the content of a set is read. Although we use the fast mode access for DR and HDD, the content read out as compare to a cache access is much smaller. For example, for HDD we will read 8x16bits from the tag and 8x28 bits from the data array while for the cache we need to read 8x28 bits for the tag and 8x256bits for the data array (all the blocks are read).

Table 7: Cache and CATCH events and units accessed

Event	Units accessed
Cache Hit	Cache, DR
Cache Miss - DR Miss	Cache, DR, L2
Cache Miss - DR Hit - Cache Hit	Cache, DR, Cache
Cache Miss - DR Hit - Cache Miss	Cache, DR, L2
HDD Hit - Cache Hit	HDD, Cache, BCU
HDD Hit - Cache Miss	HDD, Cache
HDD Miss	HDD

Using these energy numbers we developed a first order model to measure the energy delay for all benchmarks with and without CATCH. The following facts were taken into consideration for modeling the CATCH energy:

1. DR table is access on every cache access
2. On a DR hit followed by a cache miss then an extra cache access energy is charged
3. HDD is accessed on every cache miss
4. On an HDD hit an extra energy cache access energy is charged
5. Block Compare Unit is only used on an HDD hit followed by a cache hit

Also, the model takes into consideration the L2 cache access energy for every cache miss and the static energy of Instruction L1 cache and CATCH. Finally, we assume that IL1 cache consumes about 15% [75] and the L2 cache about 30% [76] from the total processors power consumption.

Table 7 shows all the possible cache and CATCH events and the corresponding units that are accessed on each event.

Figure 25 shows the normalized energy delay product of the simulated processor when using CATCH. The results indicate that for the benchmarks that have performance improvement using CATCH, the energy delay product is also improved. On the other hand for some benchmarks, like APPLU that does not have any performance improvement, the energy delay product is slightly

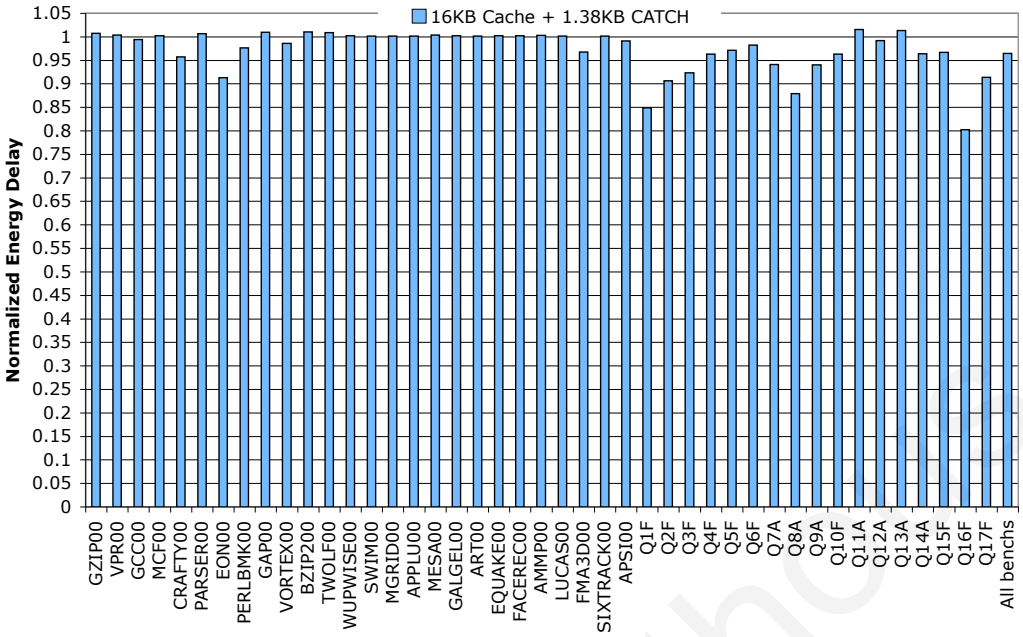

Figure 25: Normalized Energy Delay product when using a 1.38KB CATCH

increased due to CATCH. Overall the results show that the energy overheads of CATCH are very low, and in total we increase the energy delay product by less than 0.04%.

4.7 Chapter Summary

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

We report on the performance of the proposed mechanism with oracle and realistic constraints and investigate the significance of various performance and cost optimizations. Experimental results for a processor with a 16KB, 8-way, 8 instructions per block instruction cache show that a CATCH with 1.38KB cost usually captures 58% on average of the CCD idealized potential.

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

Chapter 5

Extrinsic and Intrinsic Text Cloning

In this chapter we identify Text Cloning as a potential inefficiency in the cache hierarchy of modern multi-core processors. Text Cloning occurs when a processor is storing at any levels of its cache hierarchy the same text multiple times. Text cloning can be wasteful to performance, especially for SMT cores, because processes compete for cache space to store the same instruction blocks simultaneously. There are several causes of text cloning and we divide them into Extrinsic and Intrinsic.

Extrinsic Text Cloning can happen when a user, many users, or middleware, copy a binary and concurrently execute the multiple copies on the same processor. The Operating System cannot detect that these binaries are identical and will map them, during execution, in different physical address space, therefore, creating unnecessary pressure at all cache levels. Such a scenario is very common in Grid Computing job flow where the binary of each submitted job is copied in a temporary directory, a sandbox, with all its input and data.

Intrinsic Text Cloning can happen when an instruction cache is Virtually Indexed/Virtually Tagged and the process identifier (PID) is included in the tag. A simultaneous multithreaded processor , that uses such cache, will map the text of concurrent processes of the same binary to different instruction cache space due to their distinct process identifier. A Virtually Indexed/Virtually Tagged instruction cache is found in the Intel's hyperthreaded (SMT) Netburst microarchitecture [77].

We identify and explain the causes of Text Cloning both, Extrinsic and Intrinsic, and demonstrate experimentally, on real and simulated SMT hardware, the significant performance implications of Text Cloning. We also discuss ways to mitigate the effects of Text Cloning, and we show the potential of CATCH [Chapter 4] to identify and eliminate it.

5.1 Text Cloning: Causes, Implications and Remedies

This section introduces Extrinsic and Intrinsic Text Cloning through discussion about when it can occur, how much it hurts performance and possible methods to avoid it.

5.1.1 Extrinsic Text Cloning

Extrinsic Text Cloning (ETC) can happen due to the user and software practices that result in the execution of multiple copies of the same binary on the same processor. The Operating System is unable to understand that these binaries are clones and will map them in different physical address spaces. Consequently, each process is associated to a different text segment and will eventually create duplication in the shared caches of the processor.

The ETC is common within Grid Computing Systems [78] due to Grid's distributed file system and the middle-ware design. Particularly, typical Grid job flow requires the binary of each submitted job to be copied in a temporary directory, a sandbox, with all its input and data. In the

case that two or more jobs that use the same binary, are submitted to the same multicore or SMT computing node the middle-ware, or even the OS in the Grid computing node itself, is unaware of this duplication.

Another emerging case of ETC is due to virtualized cloud computing where multiple users can run local copies of the same applications that happen to execute on the same physical processor [79].

Furthermore, ETC can happen when an application contains self-modifying code routines. When a process, that shares its physical address space with other processes of the same application, self modifies its code then the memory page that contains the modified code has to be copied in different address. This will result to duplicated blocks that were contained in the copied memory page but remain unaffected from the code self-modifying routine [80].

Finally, a common habit among users is to keep their own copies of same applications in their home directories. This might lead in ETC when two users are logged in the same machine and run the same application, each using their own copy.

5.1.2 Intrinsic Text Cloning

Intrinsic Text Cloning (ITC) is specific to VIVT instruction caches. A VIVT cache uses the Virtual Address to tag match a block. In the case of a shared VIVT cache, the tag also contains the PID of the process to avoid homonym problems. However, each instance of the application will have different PID and this will create synonyms [81] in the instruction cache. ITC is equivalent to the occurrence of synonyms in an instruction cache.

VIVT caches are used for L1 Instruction caches to have lower access latency and lower energy per access by avoiding ITLB translations on every cache access. Cloning in IL1 caches only occurs when the tag of the Virtually Tagged (VT) caches includes also the PID. Single thread cores do

not require keeping the process ID in the tag unless they want to avoid cache flashing after each context switch. For an SMT processor, on the other hand, the PID is essential in the tag of a VT cache because multiple threads coexist in the cache at the same time.

The ITC can happen either when we run multiple copies of the same binary or multiple instances of the same binary. On the first scenario, the reasons are the same as those discussed in Section 5.1.1. The second scenario, multiple instances of the same binary, is very common when running the same application with different inputs, or using applications that by default create a different process for each instance due to lack of multithreading support or other programming reasons. For example, versions of Microsoft Excel and Internet Explorer create a distinct instance each time they are invoked.

Another possible cause of ITC is the service daemons running on servers. Not all these applications are multithread, and create a different process each time a user request the service. A very common category of services that spawn multiple processes is the kernel services.

5.1.3 How Important is ETC and ITC

This Section uses two real processors with 2-way SMT cores, the Intel Pentium 4 (P4) [82] with VIVT 12KB Trace Cache and the Intel i7 [83] with a VIPT 32KB IL1 cache to measure the performance impact of Text Cloning in IL1 cache. We used a synthetic benchmark (see APPENDIX B) that exercises the instruction cache by executing a large basic block of calculations for different basic block sizes. The benchmark has minimal data requirements, only few initial capacity misses, effectively no-conditional branches, and several random indirect unconditional branches to measure only the impact of the instruction references on performance.

We measure the implications of ETC and ITC by performing two experiments for each processor. First, two instances of the same binary are executed in parallel. The OS is aware that both

processes refer to the same binary, and it will load the text only once in the physical address space but it will create two different virtual address spaces, one for each process. This causes ITC only in the P4 with VIVT caches since the address mapping of the threads in the i7 VIPT cache will be the same. For the second experiment, two copies of the same binary are run again in parallel for the SMT execution. This causes the two processes to be mapped in different physical address spaces and hence differently virtual address spaces. This manifests into Text Cloning both for P4 and i7 caches in all levels of the cache hierarchy.

For both experiments, the two processes are forced to run on the same logical core using the taskset command. In this way the two processes will be executed in parallel using one SMT core and share the same IL1 cache.

5.1.3.1 Intel Pentium 4 with a VIVT IL1

Figure 26 shows the results for the Intel P4. The y-axis of the figure shows the SMT speedup compared to running the two processes back to back. The x-axis shows the static instruction footprint of each process. For the VIVT IL1 cache of P4, running either copies or multiple instances of the same binary does not make any difference. In both cases the two processes will be mapped in different virtual address spaces. The evidence for ETC (two copies) and ITC (same binary) are supported by the behavior from 1KB to 12KB instruction footprint. For this sizes the single thread will fit perfectly on the IL1 cache while the SMT executions will suffer with cache misses after the 6KB instruction footprint. In the figure, we can clearly see that the speedup of SMT for both experiments is dropping once the instruction footprint exceeds the 6KB from 80% down to 55% for 12KB.

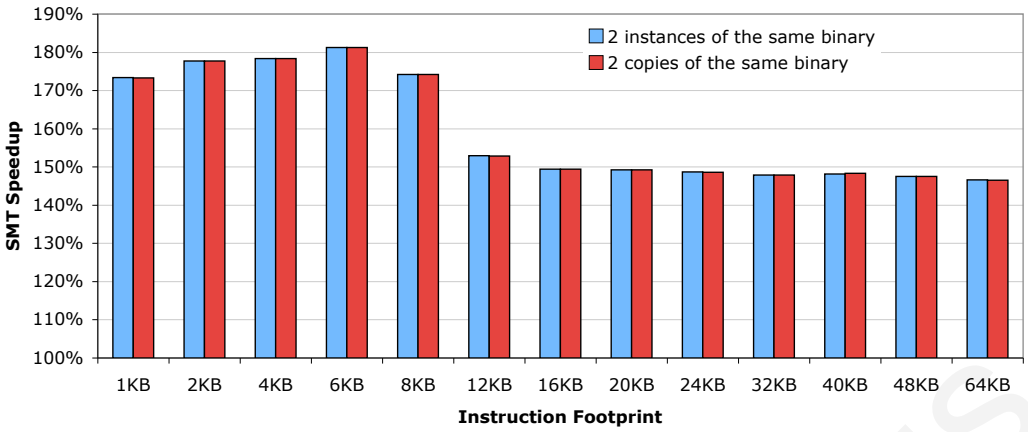

Figure 26: Intrinsic and Extrinsic Text Cloning on Intel Pentium 4

5.1.3.2 Intel i7 with a VIPT IL1

Figure 27 shows the effects of running concurrently the same binary and two copies of the binary on an i7. The trends for i7 are clearly different as compared to P4. Specifically, comparing the two bars in Figure 27 we observe that when running two different copies of the same binary the SMT speedup is reduced when we go beyond the 16KB instruction footprint because now the combined workload of the two copies occupies 32KB in total which barely fits the i7 32KB IL1 cache. This is clearly due to ETC. On the other hand, the runs with the same binary experience no Text Cloning, as opposed to P4. Specifically, with the 16KB instruction footprint the instructions of both processes are mapped in the same physical space and hence are mapped only once in the VIPT IL1 cache of i7. Comparing Figures 26 and 27 we clearly see that ETC can affect both cores while ITC affects only Pentium 4 that uses a VIVT IL1 cache.

Furthermore we have evaluated the effects of ETC using a real application, the SMTSIM simulator with the SPEC2000 benchmarks as inputs. Figure 28 shows the effects of running concurrently two clones of SMTSIM simulator with the same input on an i7. The bottom bar shows the total execution time when there is no cloning, while the top bar indicates the extra overhead

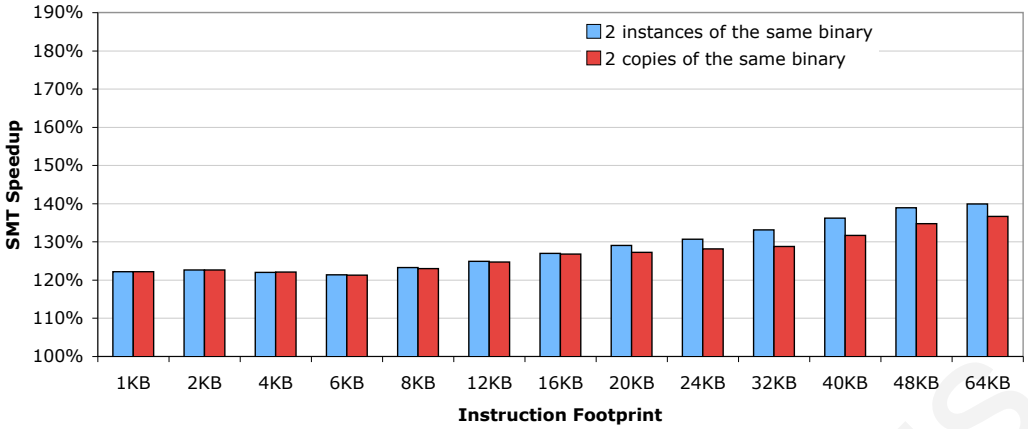

Figure 27: Intrinsic and Extrinsic Text Cloning on Intel i7

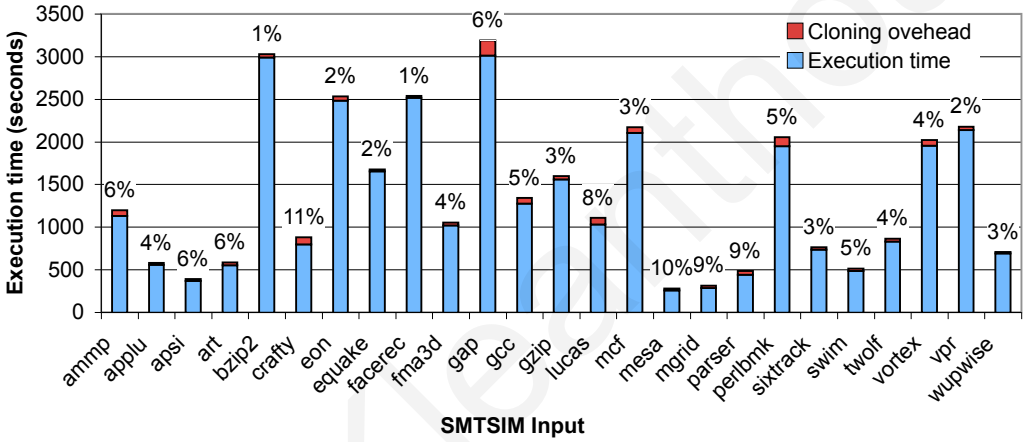

Figure 28: Extrinsic Text Cloning overhead on Intel i7

when ETC is introduced due to cloning. The results show that ETC can increase execution time by up to 11% and most of the times more than 5%.

5.1.4 How to Eliminate ETC and ITC

ETC can be avoided if the OS is enhanced with the ability to detect copies of the same binary and map them at the same physical address space, similar to what linux does with Kernel SamePage Merging [41, 42]. This however can cause security problems since someone can exploit this to inject harmful code in applications that are commonly used among many users.

Another possible solution is to enable the hardware to detect this duplication with hints from the OS or in real time to completely avoid user intervention. At this low level, the detection of cloned text can be more efficient and more secure. Two such mechanisms that have already been proposed are the one in [84] and CATCH in Chapter 4 that with certain modifications can be applied to ETC.

Mohamood et al. [84] proposed a mechanism to detect DLL sharing between different threads that use the same DLLs. The proposed mechanism is based on both VIVT and VIPT caches that are aware of DLL sharing using a bit in the ITLB table that is set with aid of the Operating System. The mechanism described can be used to prevent text cloning but we believe that a simpler mechanism may be sufficient because the granularity of duplication is much bigger in the Text Cloning scenario.

Also this thesis, proposes CATCH, a mechanism that dynamically detects and eliminates duplicated instruction sequences, valid blocks, from the IL1 cache. CATCH is also a candicate to eliminate ETC.

ITC can be avoided by using a VIPT IL1 cache. The VIPT cache requires an access to the ITLB on very cache access to translate the Virtual to Physical address. This costs energy for accessing the ITLB, but also performance because although the Indexing in a VIPT can be done with Virtual address this is not enough to hide the ITLB access and tag matching. This extra translation might increase more than a cycle the IL1 cache access latency. Previous SMT processors, like Intel Pentium 4, kept the L1 Instruction Cache to be VIVT but modern processors, like Intel i7, have a Virtually Indexed/Physically Tagged (VIPT) cache with the extra overhead of the ITLB translation on every IL1 cache access. Therefore, the particular instruction cache configuration may depend on power and performance trade-offs.

Since ITC is the equivalent for synonyms in an instruction cache there has been a lot of work to improve performance or reduce they energy of virtually tagged caches [85, 86].

Also two possible hardware mechanisms for the ITC problem that can detect and eliminate Cache-Content-Duplication dynamically are [84] and CATCH [Chapter 4] which can help to eliminate both ETC and ITC.

5.2 Grid Computing Systems

In this section we will explain in detail how and where Extrinsic Text Cloning manifests in Grid Computing Systems and specifically in EGEE project [87].

5.2.1 Grid Architecture

Figure 29 shows the basic components of EGEE grid system that uses the gLite middleware to submit, schedule, execute and manage users' jobs. The figure shows that this grid computing systems is composed from four basic elements, (a) the User Interface (UI), (b) the Workload Management System (WMS), (c) the Computing Element (CE) and (d) the Worker Node (WN) [88].

The UI provides the tools for the user to submit or cancel his job and to retrieve the output result of the submitted job. Once a job is submitted from a UI it arrives to a WMS. The WMS is responsible for the load balancing of the whole grid infrastructure by keeping records of the balance in each cluster and which clusters are available for execution. Once the WMS chooses the cluster to submit a job it sends the job description in a WMS wrapper script to the appropriate CE of the cluster. The CE is responsible for keeping track of the workload in its own cluster and submits jobs to different WNs that belong to the cluster. Finally the WN is running a job resource manager, for EGEE is Torque/PBS, which executes the WMS job wrapper script that

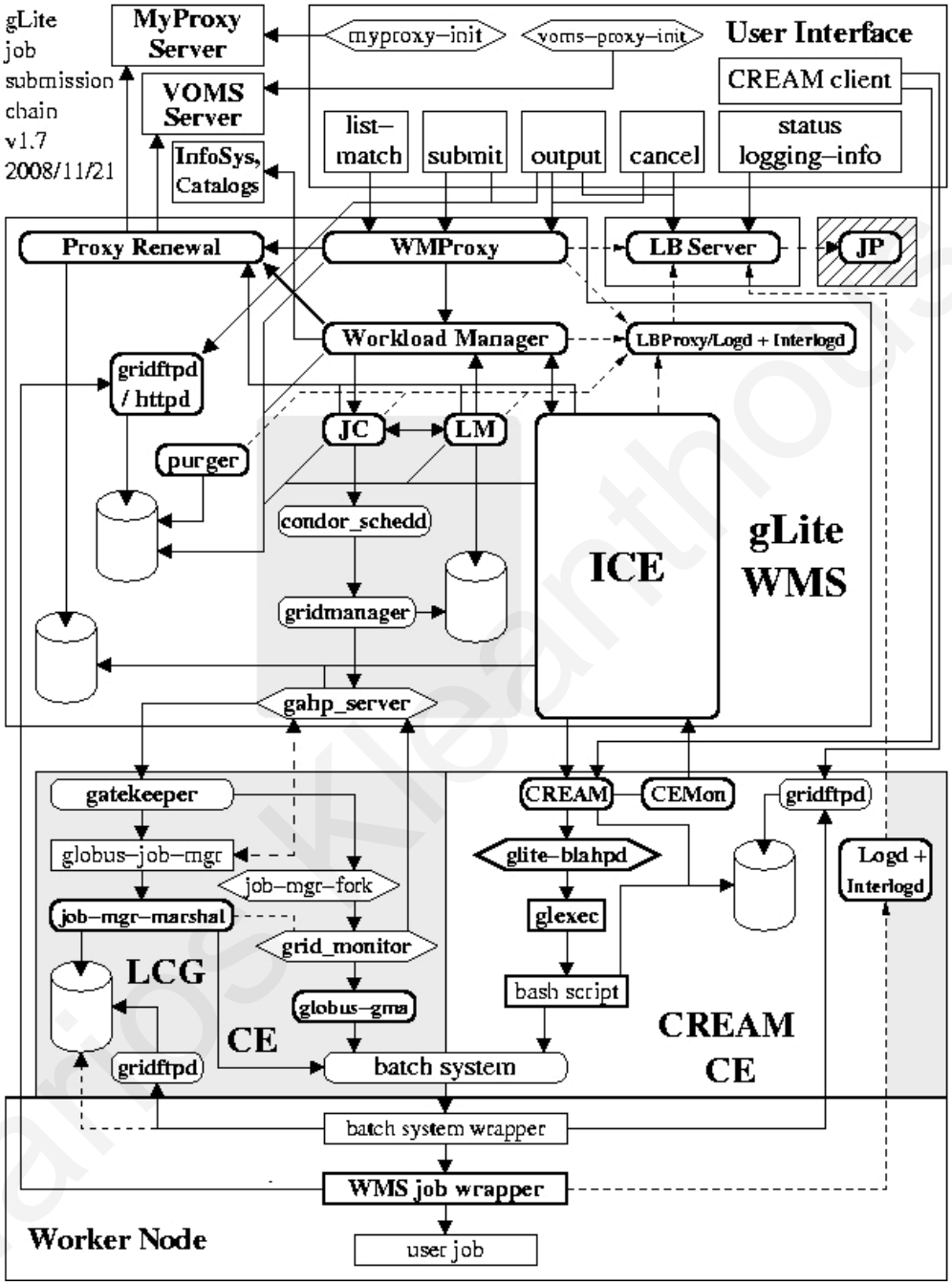

Figure 29: gLite job submission chain
(http://web.infn.it/gLiteWMS/index.php/techdoc/howtosandguides)

setups, downloads and uploads the job's sandbox, executes the job, logs the output and finally cleans up once the job is done.

5.2.2 Extrinsic Text Cloning in Grid

ETC is caused by the very last stage of the grid job flow, at the WN, where the WMS job wrapper creates a different sandbox for each job. This prevents multiple jobs that run on the same worker node, multicore or SMT, to share their binaries but also provides secure execution of the job. The architecture of grid is built to provide abstraction in each level, but also security for the users to run their job without interfering with each other [78].

This approach provides little or no opportunity to the middleware to optimize job submission and execution to share binaries because there is a high risk of compromising security. For example, even if the WMS component is smart enough to group jobs together that use the same binary and submit them to the same CE it would still need to run in different sandboxes to prevent interference between jobs' inputs and outputs and even malicious activity from other users that may try to exploit this hole.

Accordingly to eliminate ETC in grid computing either the OS running on the worker node or hardware support or a co-design of the two is essential.

For example, a service in the OS that compares the new binaries for execution with the binaries already running can be used. This can be done using a table that keeps a content id (e.g. the CRC code) of the text of all running binaries. When a new binary starts executing, its content id is compared with all the running ones and if there is a match the texts are compared for validation. If two texts are identical they can be mapped at the same physical address space. In case of self-modifying code the OS must be aware to split merged texts into different physical address spaces. This technique will require no hardware modifications but requires for the OS to do all the

comparisons and monitoring for self-modifying code or other possibly malicious actions from the users.

Another approach is to have a hardware mechanism detecting text cloning. The granularity of duplication can be chosen statically for each set of binaries or it can change dynamically. For example, only a relation between the PIDs needs to be recorded for two identical binaries. On the other hand, if two binaries are very similar but not identical, for example, an open source simulator that is slightly modified by each user, detection at the granularity of pages or cache blocks is more appropriate. By reducing the detection granularity, the duplication opportunity increases but the number of relations to be recorded increases also. Smaller granularity also provides duplication detection across very different applications and even within the same binary. Furthermore, detecting self-modifying code and invalidating relations is easier in hardware because it can monitor the instructions that write the text segment.

A possible efficient design can be the combination of software and hardware. For example, a co-design where an OS software mechanism provides hints, for the relations and the text cloning granularity, to the hardware mechanism that will validate, create, and detect the duplicate relations. The OS has a broader view of the processes running and can detect if two texts are identical, similar, or very different. This can help the hardware mechanism to adapt the granularity to detect text duplication. Finally, the hardware can detect self-modifying code and invalidate any relations that become invalid.

Provided that Text Cloning is a frequent phenomenon, future work should evaluate and engineer all these options to determine how to best to detect and eliminate it.

5.3 Evaluation Using Simulation

For simulation evaluation of the effects of text cloning we consider only the scenario where multiple copies of the same binary are executed using a VIPT IL1 cache of an SMT core, which corresponds to Extrinsic Text Cloning (ETC).

To evaluate the performance we have used the SMTSIM simulator [63] with a selection of 7 benchmarks of the SPEC2000 suite. The 7 benchmarks selected were 3 with a large instruction workload, fmad3d, crafty and perl, 2 with a medium instruction workload eon and vortex, and 2 with a small instruction workload load, ammp and lucas. This benchmark selection is done to show the potential performance of ETC for different cases of instruction cache pressure. The benchmarks simulation regions, inputs, and compilation and the processor's configuration are described in Chapter 3. Adittionally, when two instances of the same application are executing simultaneously there is a 500 million instructions shift region. The shift region is the difference in dynamic instructions between the two copies of the binary that are executed simultaneously to avoid overlapping program phases. For these shift regions we have verified that there is no overlapping between the simulated regions of the two copies.

5.3.1 Results

Figure 30 shows the Weighted Speedup [89] normalized to the first bar, which is the performance of 2 instances of the same binary running on an SMT processor. For the experiments in Figure 30 all applications are running synchronized, that means they are executing exactly the same program phase. The results show that the performance degradation due to ETC, when running 2 copies of the same binary, is up to 60% for crafty and more than 20% for the other benchmarks. For lucas and ammp that have very little pressure on the instruction cache ETC does not affect the performance.

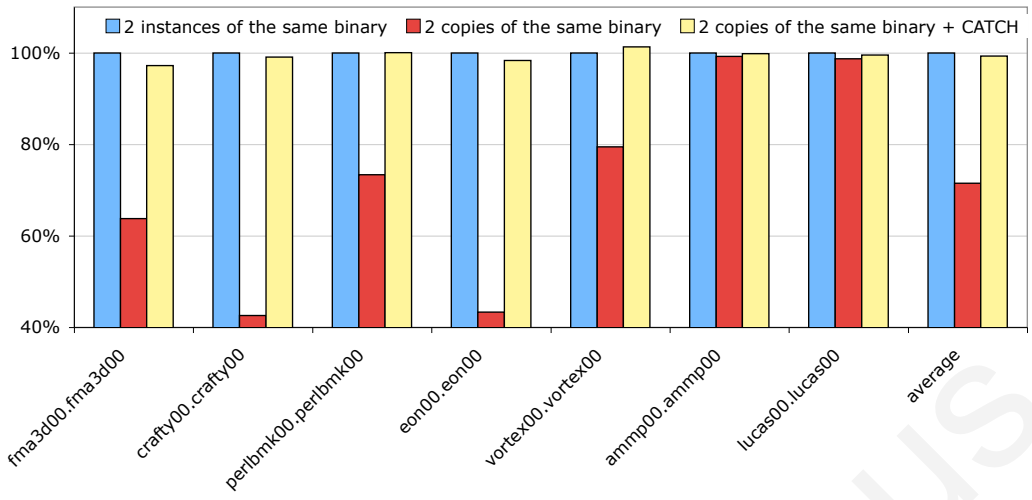

Figure 30: Weighted SpeedUp. Detecting and eliminating ETC with overlapping program phases

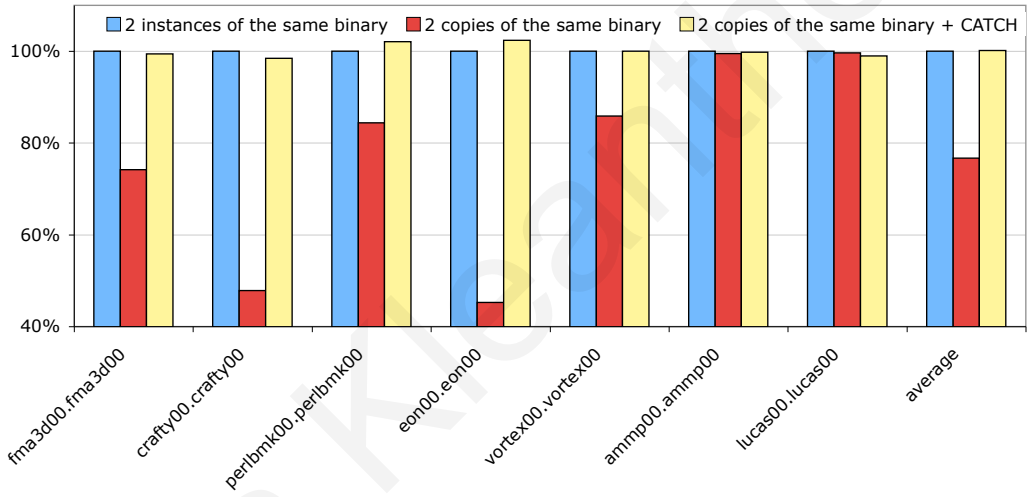

Figure 31: Weighted SpeedUp. Detecting and eliminating ETC with 500 million instructions shift in program phase

Figure 31 shows a more common scenario where the two applications running simultaneously are in different program phase, 500 million instructions shift, in their execution. We have verified that none of the applications are overlapping with their copy during the execution. The results show that the performance degradation is a little less, mainly because by executing a different phase we can avoid some conflict misses. Still the bigger instruction footprint due to ETC can cause 55% slowdown for eon and crafty and about 20% for the other benchmarks. The ammp and lucas are again not affected by ETC due to the very small instruction cache workload.

These results suggest that the use of a hardware mechanism, OS support or a combination of the two will be useful to eliminate the performance degradation due to text cloning. We chose CATCH [Chapter 4] to show how a hardware mechanism can be used to recover performance loss due to Text Cloning.

Figure 30 shows how CATCH can reduce the overhead of cloning. The third bar shows the performance when two copies of the same binary are executing and CATCH is used to detect and eliminate cloning. We can see that when using CATCH the performance degradation is reduced to 0.07% on average. There is even one case, for vortex, that the performance of CATCH is even better compared to the run where we have executed the same binary twice. This is because CATCH detects duplication not only across different binaries, but also within the same binary and thus improving the performance of the single thread execution.

The results in Figure 31 are similar to 30 but this time we can see that CATCH eliminates completely the cloning overheads on average. We would like to note again that CATCH is not for free and each duplication detection is penalized with one extra cycle that corresponds to an extra cache access for the duplicated block. The CATCH mechanism is described in detail in Chapter 4.

We have used CATCH as a case study to show how a hardware mechanism can be applied to eliminate ETC. The results indicate that an Operating System mechanism or a hardware mechanism that is aware of text cloning can be very useful to improve the performance of modern platforms that suffer from ETC, such as the Grid Computing and Cloud Computing Systems .

5.4 Chapter Summary

This chapter analyzes the effects of Extrinsic and Intrinsic Text Cloning (ETC) in caches. Extrinsic text cloning can occur when a binary is copied and executed concurrently multiple times, for example in Grid Computing Systems. In that case the OS is unaware of the Text Cloning and

two or more copies of the same binary will be mapped in different physical addresses. Intrinsic Text Cloning (ITC) can occur in the case of Virtually Index/Virtually Tagged caches where the same text segment is mapped in different virtual address spaces.

We evaluate the effects of ETC and ITC, using two SMT Intel processors, P4 and i7 with a synthetic benchmark. The results indicate that the slowdown in execution due to Text Cloning is significant and a mechanism for detecting and eliminating this overhead can be important.

Simulation based evaluation has shown that the performance overheads of ETC can be completely eliminated using CATCH to detect duplication between instruction sequences. Overall, the analysis suggests the importance of OS and architectural support to eliminate Text Cloning.

Chapter 6

CCD for Data

While Chapters 4 and 5 focused on duplication for instruction, this Chapter will focus on duplication for data. Similar to instructions, by removing redundant data from caches we can increase the effective cache size using mechanisms that will detect and exploit the data duplication. Potential applications of such mechanisms can be to increase the performance by fitting more data in the same space or to reduce the energy by switching off parts of the data array.

In the rest of this Chapter we will characterize the CCD for data caches, investigate the effects of dirty blocks and zero runs and and evaluate its potential and trade offs for various block granularities.

6.1 Data Redundancy Characterization

In this Section we will characterize the redundancy for data blocks during dynamic execution. The results do not include the TPC-H benchmarks in this characterization because they have very small data footprint and do not provide any significant information.

In Figure 32 we present the absolute number of unique blocks (32byte) needed to cover the specific amounts of dynamic execution when they are identified by their block tag, TAG, and by

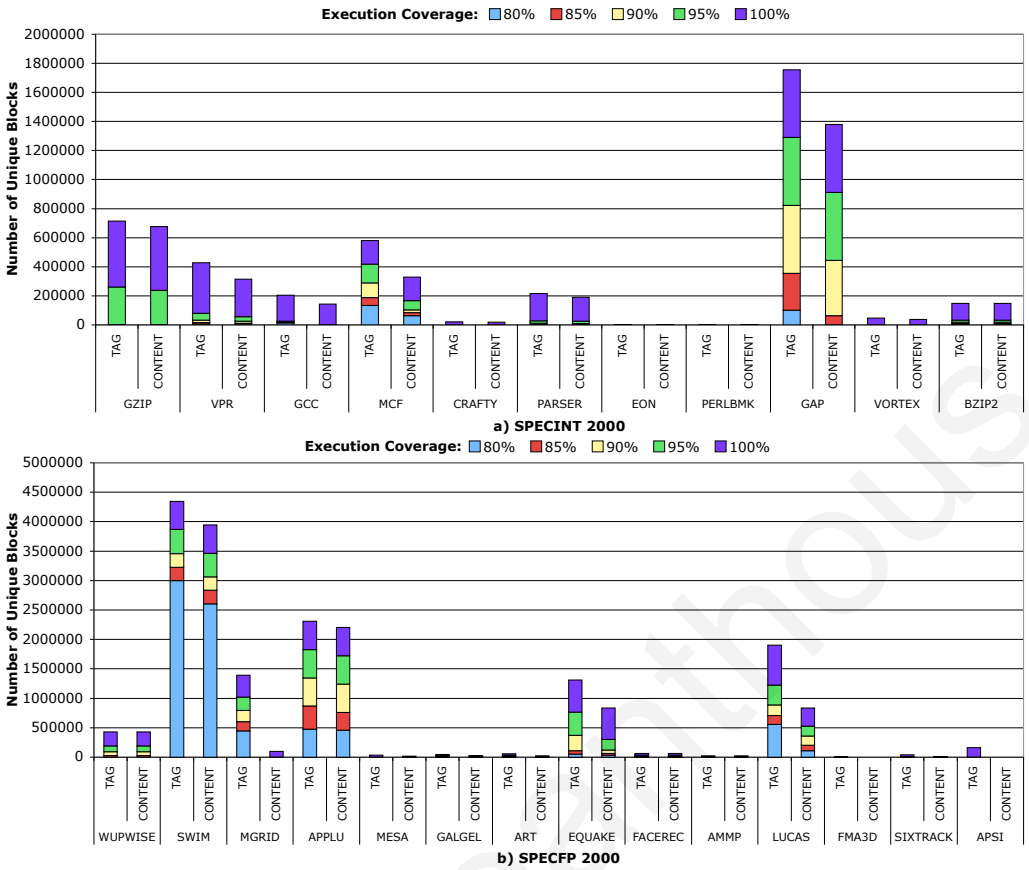

Figure 32: Execution coverage of unique blocks for the a) SPECINT 2000 and b) SPECFP 2000

their unique content, CONTENT. As it appears from the results there are few benchmarks, like MCF, GAP, EQUAKE and LUCAS, with a significant reduction in their unique blocks required when are identified by their CONTENT as opposed to be identified by their TAG. On the other hand we can see several benchmarks that are not actually affected by this phenomenon either because they have very small data footprint, like EON and AMMP for example, or because they have many unique blocks, like APPLU.

Figure 33 presents a normalized execution breakdown of the previous results. The unique blocks identified by their CONTENT are normalized to the total unique blocks identified by the TAG. It is clear that by removing the duplicated blocks the space required for execution is reduced in most of the cases by more than 20% for the SPECINT benchmarks and more than 40% for the

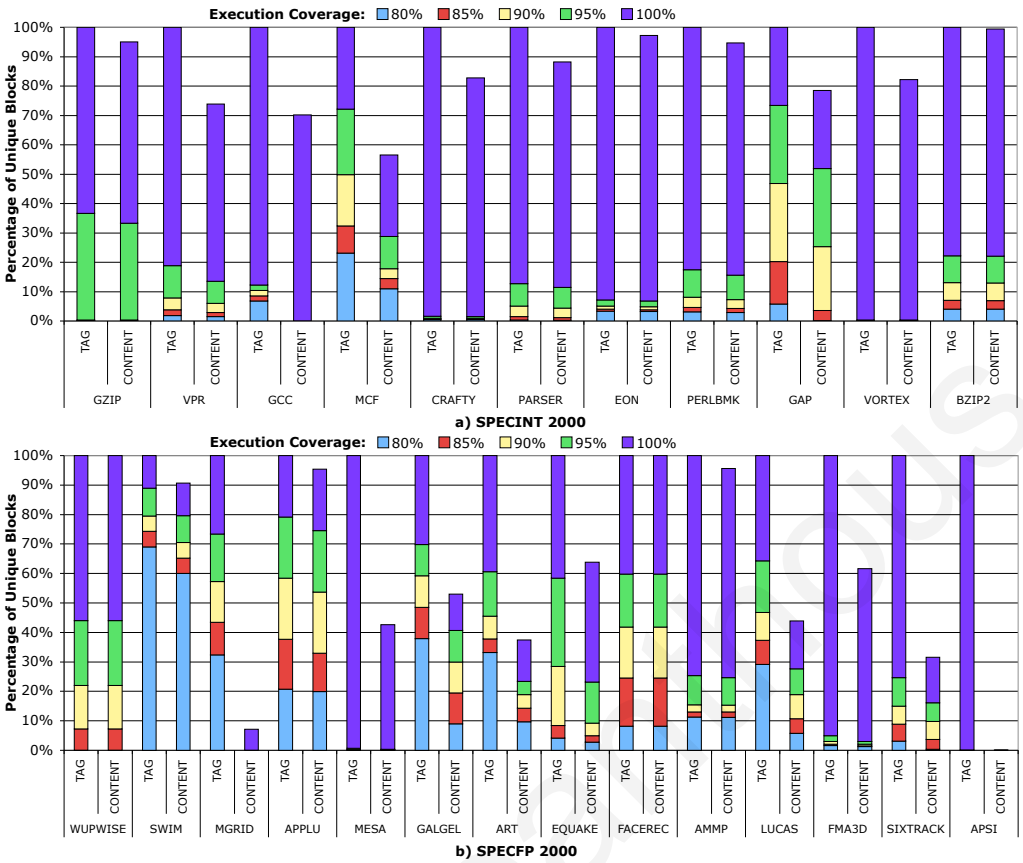

Figure 33: Execution coverage breakdown of unique blocks in percentages for the a) SPECINT 2000 and b) SPECFP 2000

SPECFP benchmarks. For one instance, for apsi benchmark, we can see that the unique blocks required, when identified by their CONTENT, are less than 1%. Of course this benchmark has very few unique blocks but anyhow it's an indication for the potential improvement which we will analyze further, later in this chapter.

Overall the results are encouraging and it seems that duplication in data is also common for many benchmarks.

6.2 Data Duplication Detection

Previous work focus on detecting duplication on data caches is based on frequent patterns, single words, or only zero runs of multiple words, up to a whole block. We noticed that all

approaches had a common challenge, the dirty blocks. Dirty blocks oppose a difficult problem for compression. Blocks that have been previously compressed and fit into a certain cache space now when they are written, and their required size changes, might not fit in their allocated physical space anymore. This will result in more complicated logic to handle this situation.

To overcome this difficulty, previous work proposed to write the dirty, uncompressed, block into a new location. Others that use correlations between many addresses to a single content [47, 51] propose to duplicate the whole dirty block and reinsert it in the cache in another location. Since this problem appears to be a common among almost all works, we decided to investigate more the effects of compressing the dirty blocks in the cache.

Another very important aspect of compression is the choice of the content to compress. Some [53, 54] have noticed that is better just to compress only zero runs instead of various patterns since it makes the compression logic much simpler. This factor will be also investigated further in the following sections.

6.2.1 Compressing Dirty Blocks

Figure 34 shows the compression rate of a benchmark, LUCAS from SPEC2000 suite, for various cache sizes, 8, 16, 32, and 64KB. The granularity that we detect duplication in this figure is for whole blocks, 64byte, and each bar in the figure is the average of 40 snapshots, 10 million instructions each snapshot. The cache is warmed-up for 100 million instructions to avoid any cold effects. We will use LUCAS benchmark only as a first example to introduce the reader to our graph's layout and legend. This type of graph will be used regularly through the rest of this thesis.

The y-axis shows the percentage of required cache size if we remove CCD. For example, the first bar that says "LUCAS ALL 8KB" shows that for lucas we only need about 12% of the total cache for an 8KB cache if we remove all duplication. The next 3 bars, 16KB, 32KB, and 64KB

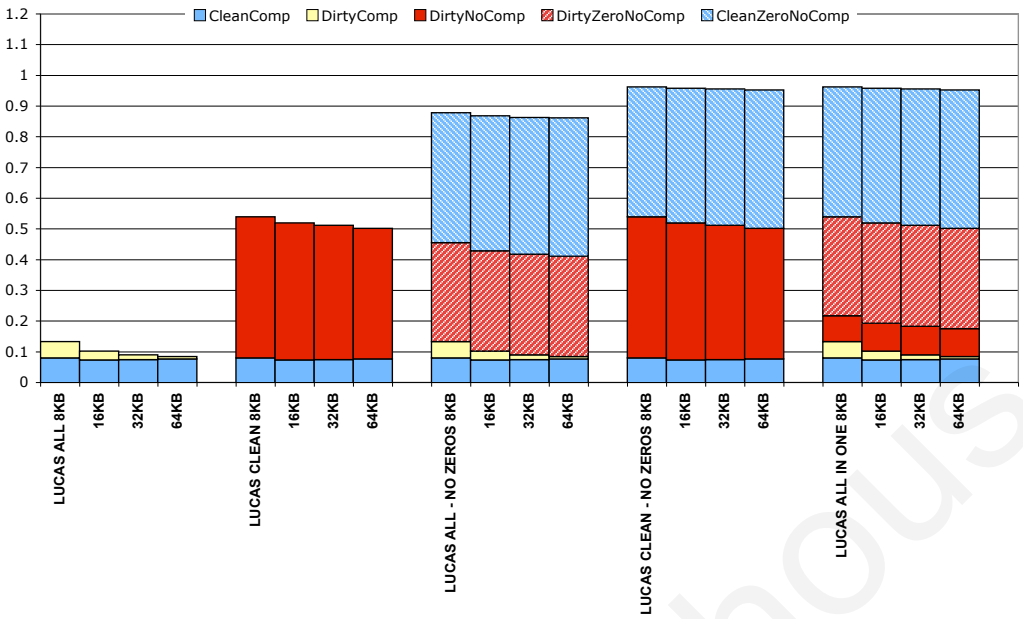

Figure 34: Normalized cache size required after CCD elimination at the granularity of 64byte blocks for benchmark LUCAS

show the cache size required after compression for respective cache sizes. All bars are normalized to their corresponding cache. That means the 10% in the "LUCAS ALL 64KB" indicates that LUCAS will require only 6.4KB of the cache size when running on a system with a 64KB cache. This corresponds to 90% compression rate.

The results in the first group of bars, in Figure 34, indicate that with bigger cache we have more compression potential. As mentioned in Chapter 4 this happens because with bigger cache we have more blocks available and thus the possibility of finding a duplicate block is bigger.

Furthermore, the second group of bars in Figure 34, "LUCAS CLEAN" shows the compression potential of LUCAS if we compress only the clean blocks in the cache. We can see that when compressing only the clean blocks we can achieve an average compression of 50% for LUCAS. This suggests that the dirty blocks might be important to be compressed, and we will investigate this further in Section 6.3.

6.2.2 Compressing Zero Blocks

Figure 34 also shows the effects of compressing the zero blocks in the cache for benchmark LUCAS. The third group of bars in this figure, "LUCAS ALL - NO ZEROS", shows with stripped bars, DirtyNoCompZero and CleanNoCompZero, the extra space that is required if we compress all the blocks except the ones that are all zeros. The results, compared to the first group of bars, shows an increase of about 75-80% in the required cache size when not compressing zero blocks, leaving only a 10-15% compression rate for the rest of the blocks. This suggests that zero blocks are a big contribution to compression for some benchmarks and it is reasonable to look it up for all benchmarks.

Finally, the fourth group in Figure 34 shows only the compression for clean blocks but without including the zero blocks and the last group combines all four previous groups into one showing the required cache size for:

- **CleanComp**: compressed clean blocks,
- **DirtyComp**: compressed dirty blocks,
- **DirtyNoComp**: extra space required by decompressing all dirty blocks except the ones that contain all zeros,
- **DirtyNoCompZero**: extra space required by decompressing dirty blocks that contain all zeros and
- **CleanNoCompZero**: extra space required by decompressing clean blocks that contain all zeros.

This annotation will be used from now on to show the results for all benchmarks in the next sections.

6.3 The Effects of Duplication Granularity for Data Caches

In instruction caches we identified the basic blocks and the valid sequences as an efficient granularity for detecting CCD but in data caches there is no equivalent to valid sequences so the problem of alignment stills exists. For the rest of this section, we will attempt o measure the effects of duplication when considering different granularities at the block level and smaller block segments.

6.3.1 Granularity at the Block Level

Figure 35 shows the results of compression potential of a data cache for several cache sizes from 8KB to 64KB and at the granularity of a whole block, 64byte.

We have already seen, in Section 6.2, that LUCAS can achieve high compression rates, up to 90%, for almost all caches. Unfortunately, things are not so good for the rest of the benchmarks as shown in Figure 35. Most of the benchmarks have 0-5% compression rate, especially in small cache sizes, and a small subset of benchmarks has more than 10% compression rate. These benchmarks are GCC, VORTEX and BZIP from SPECINT and all benchmarks from SPECFP except WUPWISE, SWIM, APPLU and MESA. There are no benchmarks from TPC-H suite that have more than 10% compression when eliminating CCD at the block level.

The above observations suggest that the performance potential for CCD on whole cache blocks is very low and only in few benchmarks. Even for the four best benchmarks where the compression rate is more than 50%, GCC, LUCAS, FMA3D and APSI, we can see that most of the potential is due to zero blocks except for FMA3D.

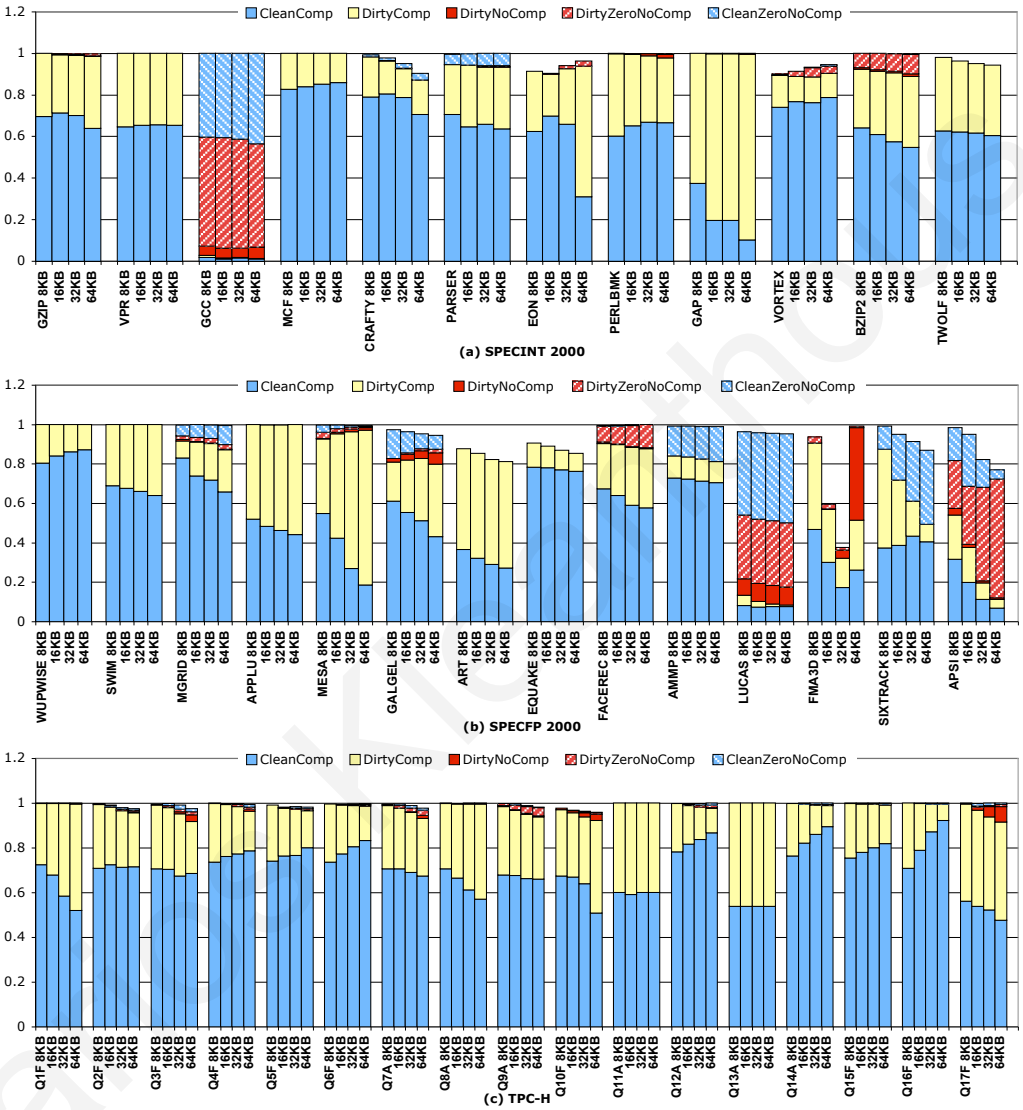

Figure 35: Normalized cache size required after CCD elimination at the granularity of 64byte blocks, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

6.3.2 Granularity at Various Block Segments

To overcome the limitation of the whole block, we considered approximating the valid sequences of instruction caches using smaller segments of the block instead. Each segment is cache block aligned at the original 64byte cache block. For example, for the 32byte segments each cache block is split into two 32byte segments, the 16byte segments splits the 64byte block to 4 segments and so on.

Figures 36, 37, 38 and 39 show the compression rates that can be achieved when splitting the cache block into 32, 16, 8, and 4byte segments. The trend is that with smaller segment size the compression potential is higher. Comparing Figure 36 and 37 with 35 we can see that the compression rates are more significant with more than 20% for all the benchmarks as the duplication granularity gets smaller.

Reducing the duplication granularity even further to 8byte and 4byte segments (Figures 38 and 39) the benefits seem to increase even more with up to 60% compression rates in many benchmarks.

6.4 Chapter Summary

The results in this Chapter indicate that there is a significant improvement as the segment size reduces but also shown that a significant amount of compression is achieved due to dirty blocks and zero runs in L1 data caches.

Compression ratios, for 16byte data segments, are usually around 20% with up to 50% for some benchmarks when compressing only clean blocks. When compressing also dirty blocks the compression can be up to 95% for one benchmark (GCC). Also the contribution of zero blocks appears to be more than 50% of the compression in most cases.

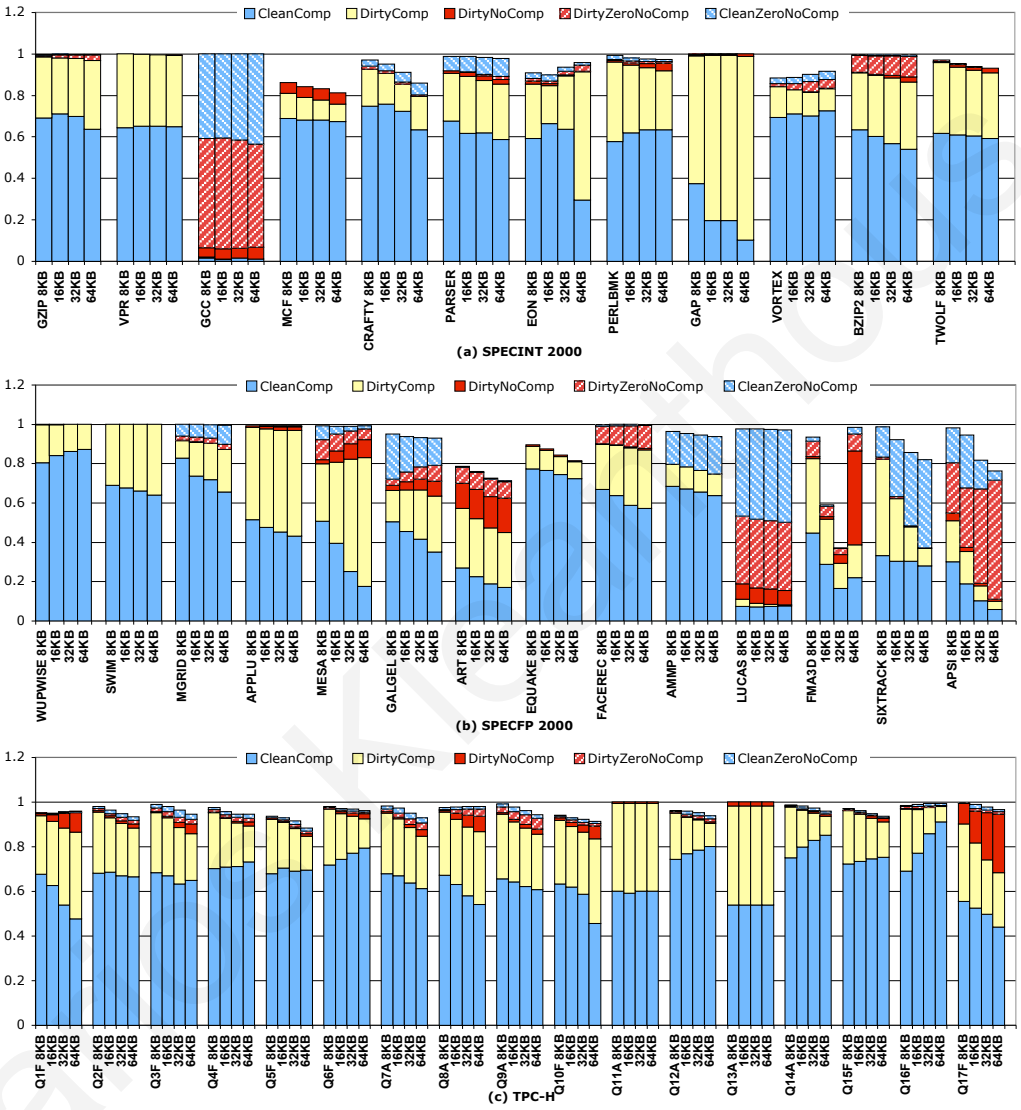

Figure 36: Normalized cache size required after CCD elimination at the granularity of 32byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

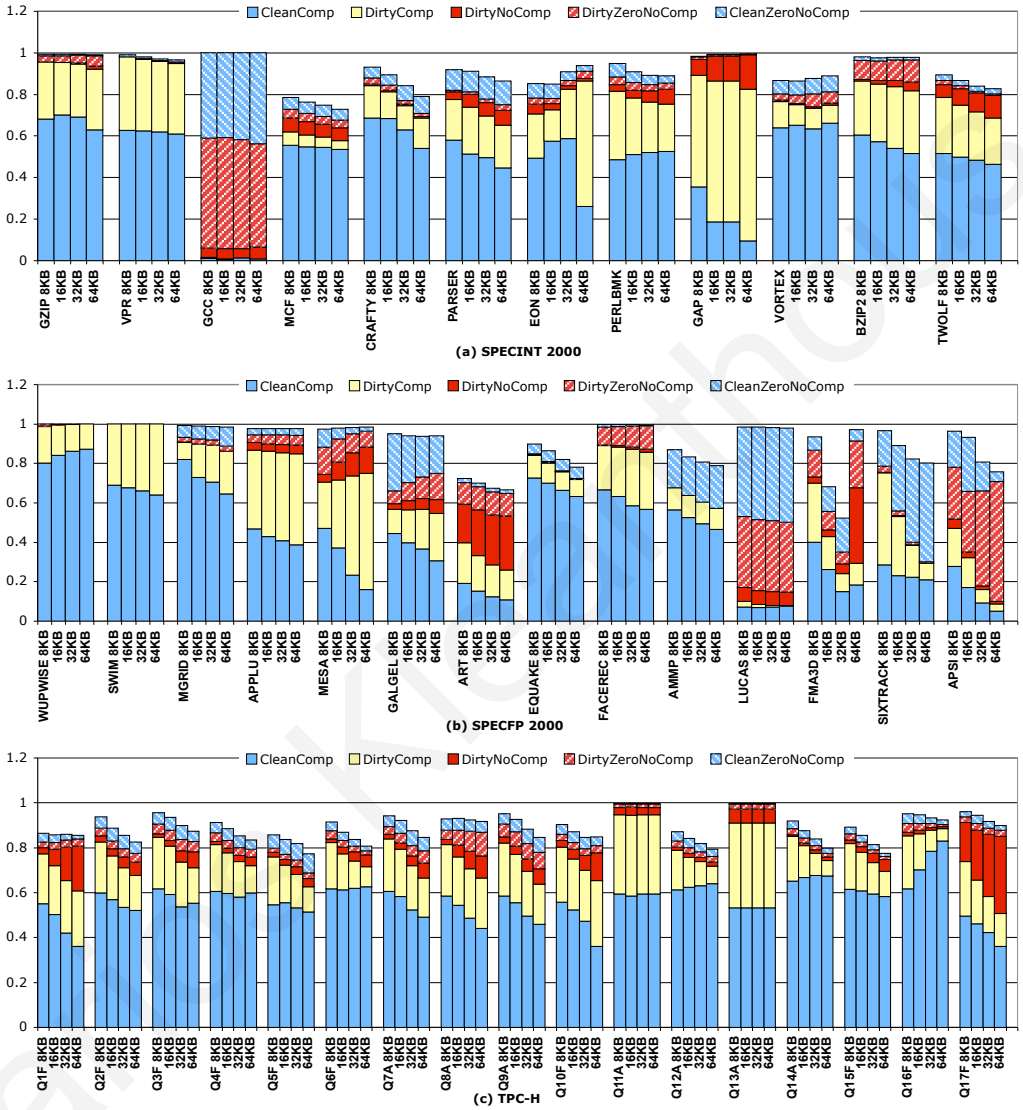

Figure 37: Normalized cache size required after CCD elimination at the granularity of 16byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

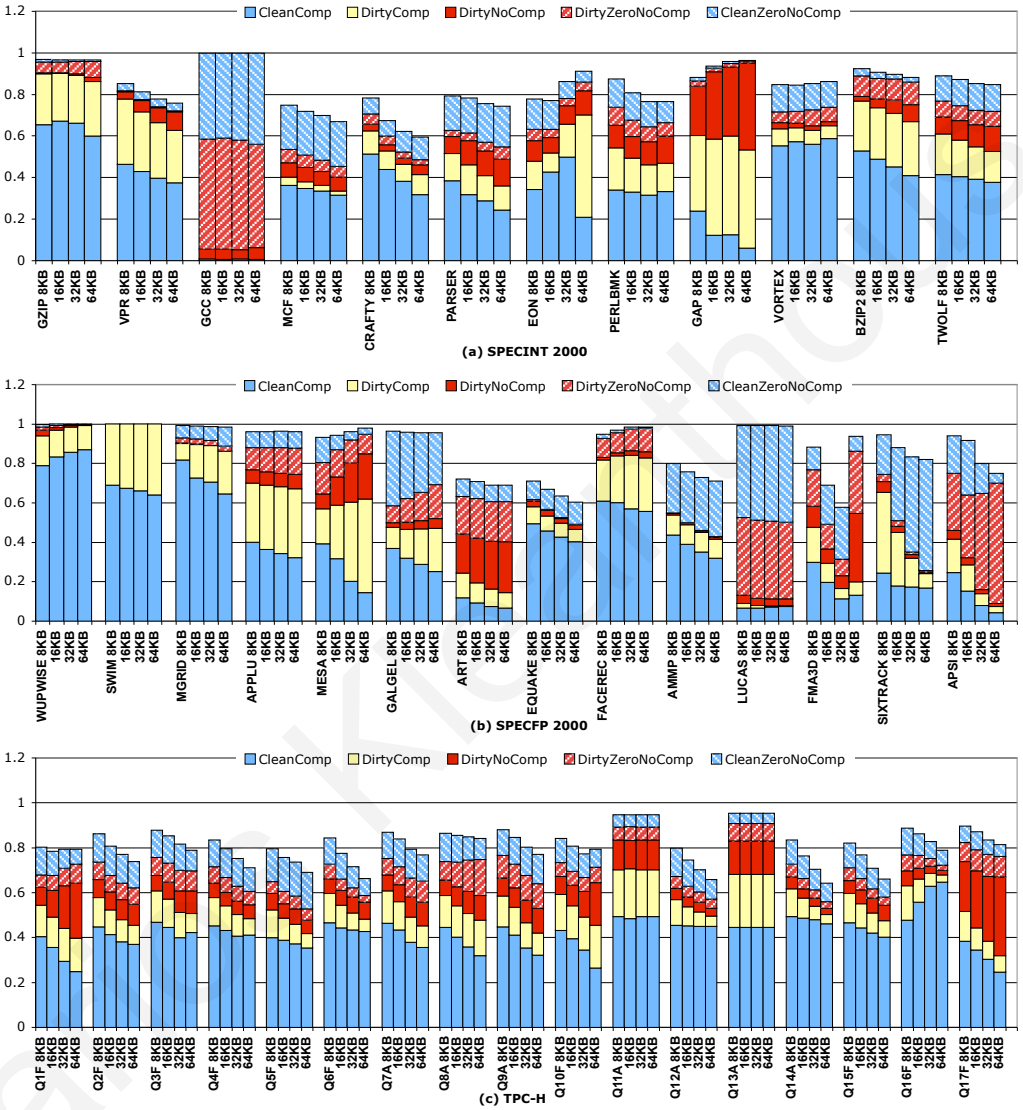

Figure 38: Normalized cache size required after CCD elimination at the granularity of 8byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

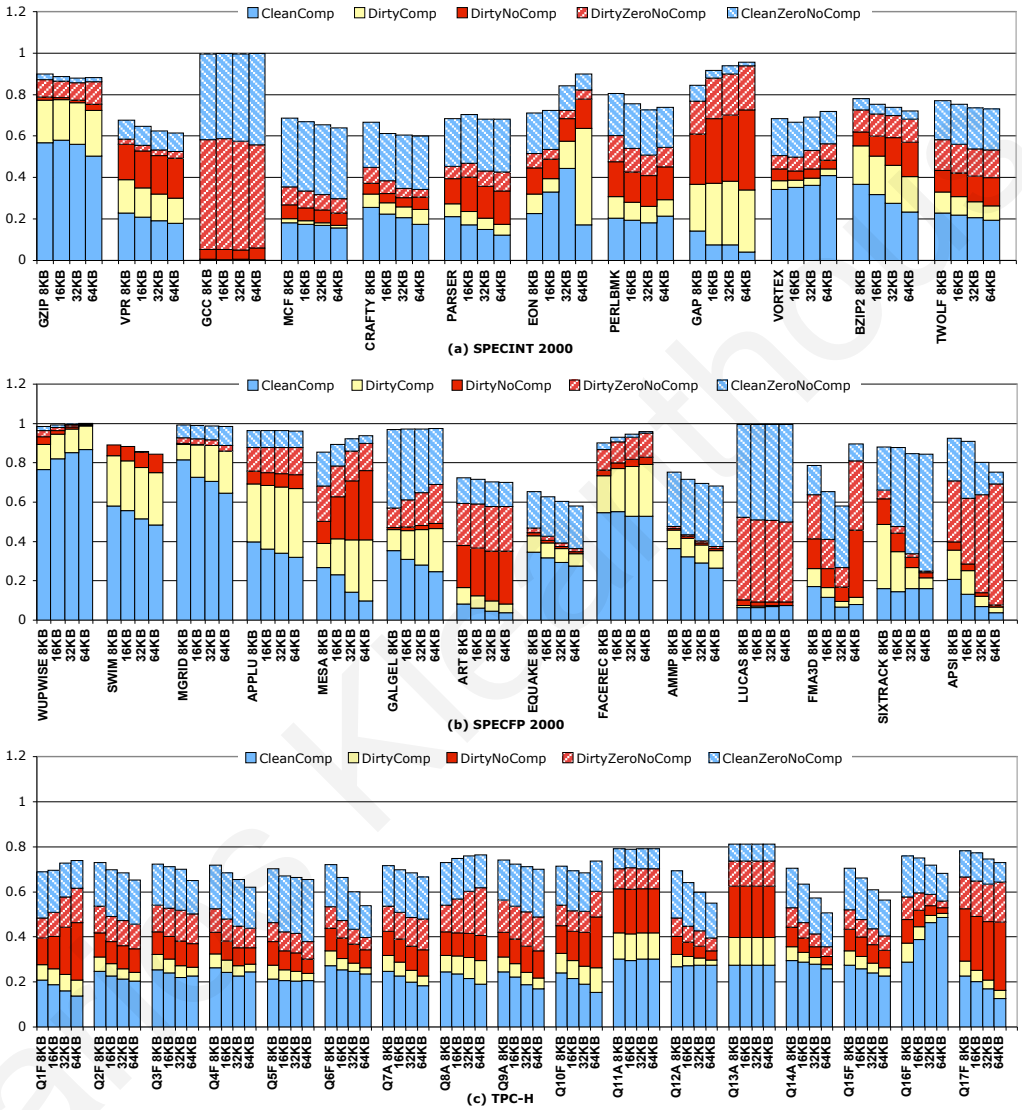

Figure 39: Normalized cache size required after CCD elimination at the granularity of 4byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

The dirty blocks are difficult to handle during compression, as shown in previous work, and especially when a dynamic detection mechanism is used, like the one proposed in the thesis. The main problem with CATCH, or with any similar mechanism, is that when a block is overwritten in the cache all dynamic duplication relations pointing to that block need to be invalidated. Especially for CATCH, since there are no backward pointers, in case of a write in the cache the whole DR table needs to be flushed. L1 data caches are very sensitive to latencies and flushing will increase the average latency which they cannot afford.

The data cache is written very frequently so flushing the DR will be inefficient. We have considered various techniques to do selective invalidation. Initial cost analysis indicates that using either backward points or link lists to be able to invalidate specific relations in the DR is not worth it because it increases the cost of the mechanism. We believe that a dynamic detection mechanism will not be efficient for dirty blocks, and they should not be considered for compression.

Furthermore, we have seen from Figures 35-39 that a large amount of zeros in the cache, with many of them for dirty blocks, contribute up to 50% to the total compression rate. The zero runs have been handled well in the DL1 cache with previously proposed mechanisms.

Finally, the accesses in a DL1 cache are very critical and cannot be hidden by already buffered data like the fetch queue in instruction caches. Any compression mechanism in DL1 will add significant overhead during decompression unless the compressed data are directly accessed like in [45].

Taking into consideration all the above limitations, we believe that a dynamic duplication detection mechanism will be inappropriate for the DL1 cache due to very frequent writes, zero runs, and significant overhead for CCD detection on a secondary hit.

Chapter 7

CCD for Last Level Caches

With wide use of the SMT and Chip Multiprocessors, the need for efficient Last Level Caches (LLCs) has become paramount. The main challenges of large LLCs are to maintain a reasonable latency and energy consumption. A recent study [76] have shown that modern CMPs spend about 30% of their power to LLC and the biggest percentage of it is due to static leakage.

This chapter investigates the potential of CCD in LLCs, for both single and multiprogram workloads to achieve both goals, reducing the latency and energy consumption.

Furthermore, we assume that CCD will be easier to exploit on LLCs, as compared to L1 Data Caches, based on the following observations:

- The higher levels of memory hierarchy work as a filter where most of the writes are hidden by the first level caches and only the last write-backs are visible to the LLCs. This will result in less frequent writes per block in the LLCs, and thus giving more opportunity for compression on clean blocks

- We also assume that the zero blocks will be less frequent in an LLCs since most of the zero runs are caused during the initialization and they are later written with non-zero values

- LLCs can be used for multiprogram workloads on multicores. This will give us the chance to exploit CCD for caches with more pressure.

7.1 Single Program Workloads

First we will investigate the CCD potential in an LLC for single program workloads. The configuration that will be used is described in Table 3 for various LLC cache sizes and the benchmarks are all applications from SPEC2000 and TPC-H suite.

We have to note that we also detect CCD in the instruction blocks contained in LLCs, but these blocks are treated same as data blocks since CCD at unified caches is agnostic of block type.

Figures 40 to 44 show the results of CCD in an LLC cache for various block segments for the cache sizes from 1MB up to 16MB. We refer the reader to Section 6.2.1 where there is a brief explanation of the graphs' annotations. The results indicate that both dirty blocks and zero runs are considerably less as compared to the results for the L1 data cache. More specifically, comparing Figure 42 with Figure 37 we can see that most of the dirty blocks have been converted to clean and the percentage of zero 16byte segments in the LLC cache has been reduced. The most affected benchmark is GCC where we can see that in the DL1 almost 50% on average of the cache during the whole execution contains dirty blocks, while in the LLC the corresponding amount is down to 24% at most for the 8MB cache. Also, the percentage of zero blocks for GCC has been reduced from 95% of the cache to just 22%. Another example is the LUCAS benchmarks in which all dirty blocks have been completely eliminated, although the percentage of zeros remained the same.

The trend is that with the block segment size increasing the compression potential is decreasing, as expected, but it's noteworthy to say that the zero runs are also decreasing with higher rates than the total compression potential. This indicates that zero runs are maybe useful when detecting

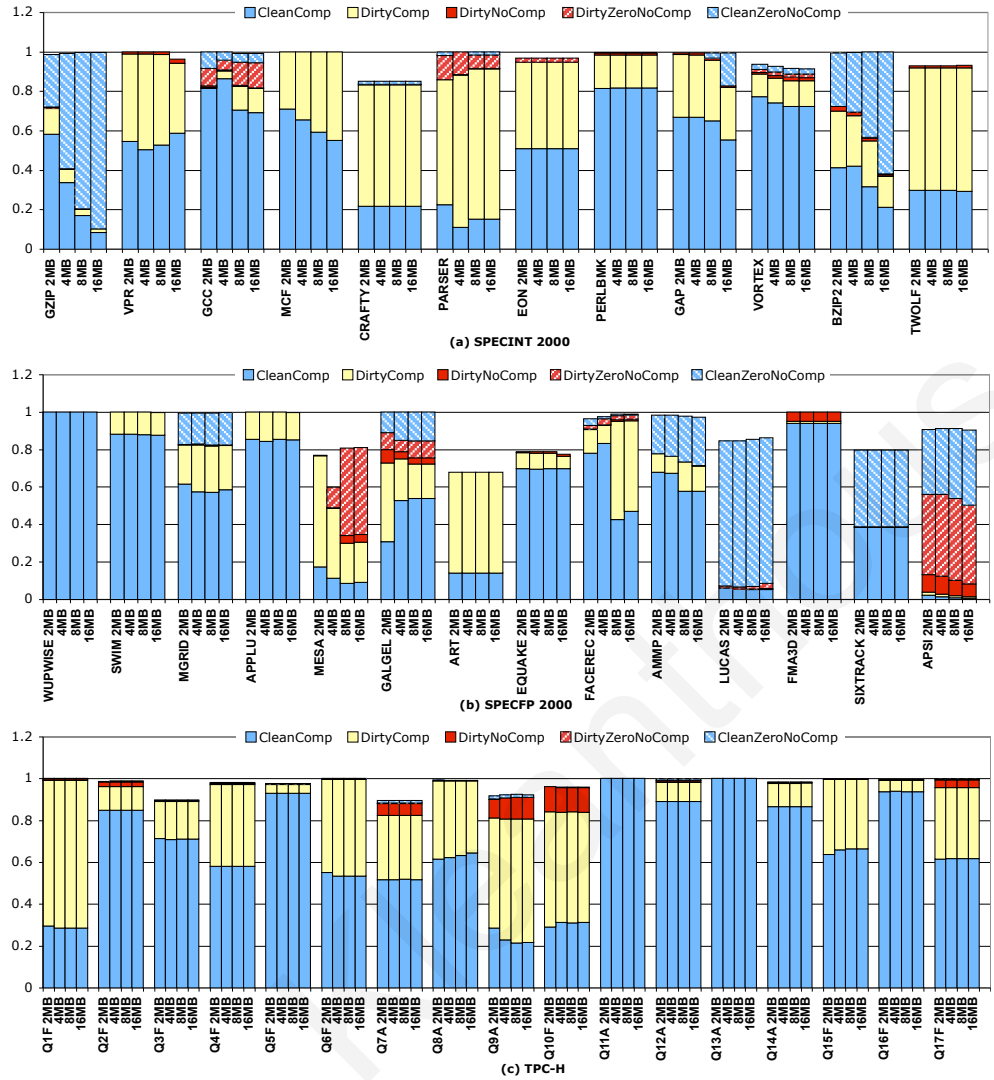

Figure 40: Normalized cache size required after CCD elimination at the granularity of 64byte blocks, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

duplication at the granularity of 4byte or 8byte segments but for 16byte segment size and larger their contributions is reduced substantially.

Also detecting duplication at very small granularities like 4 and 8byte segments means more complicated logic to compose the blocks and more hardware to keep the relations since each segment requires a pointer in its tag to maintain the relation.

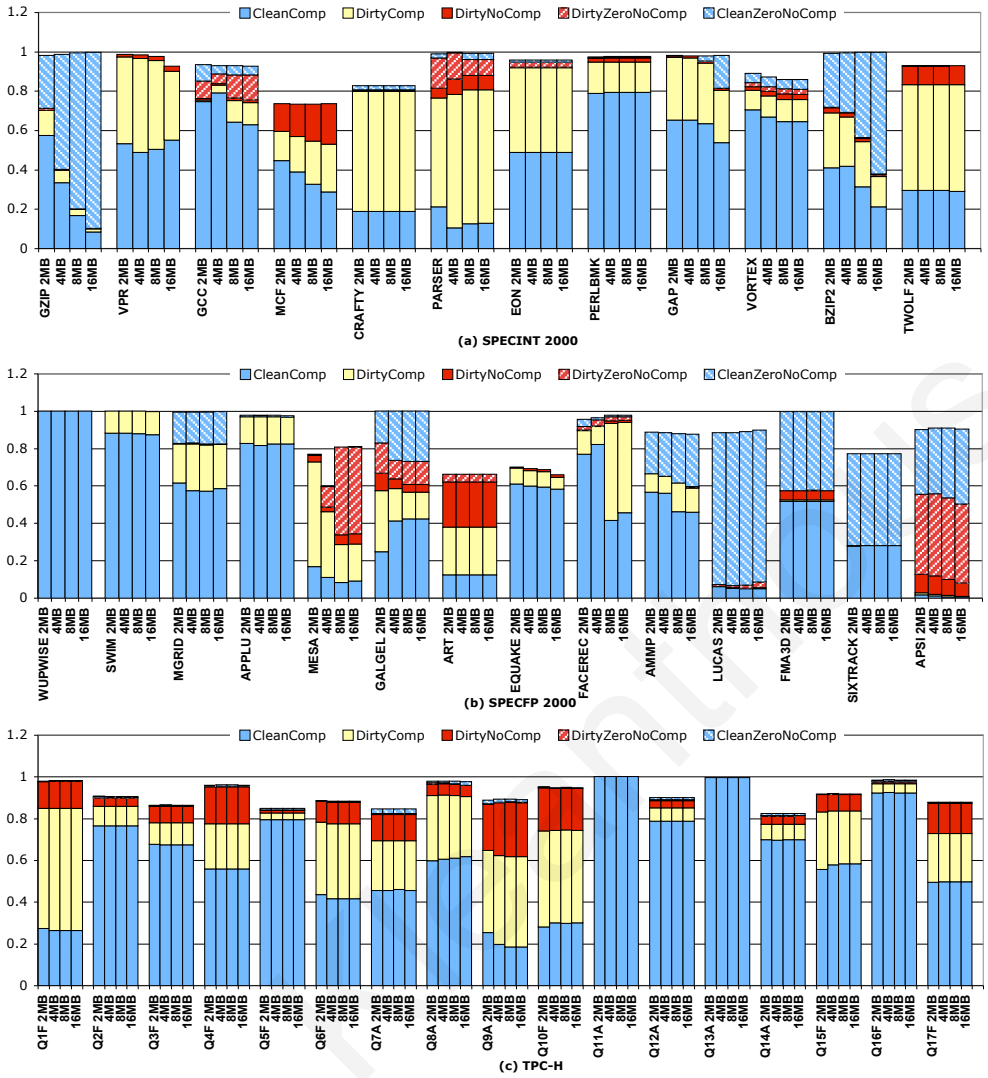

Figure 41: Normalized cache size required after CCD elimination at the granularity of 32byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

For these reasons, we will consider only the duplication for 16byte segments that appear to be more promising regarding both the potential but also the complexity to detect and maintain the duplication at this granularity.

7.2 Multi Program Workloads

Next we will investigate the frequency of CCD in LLCs for multi program workloads for a 2-core Chip Multiprocessor.

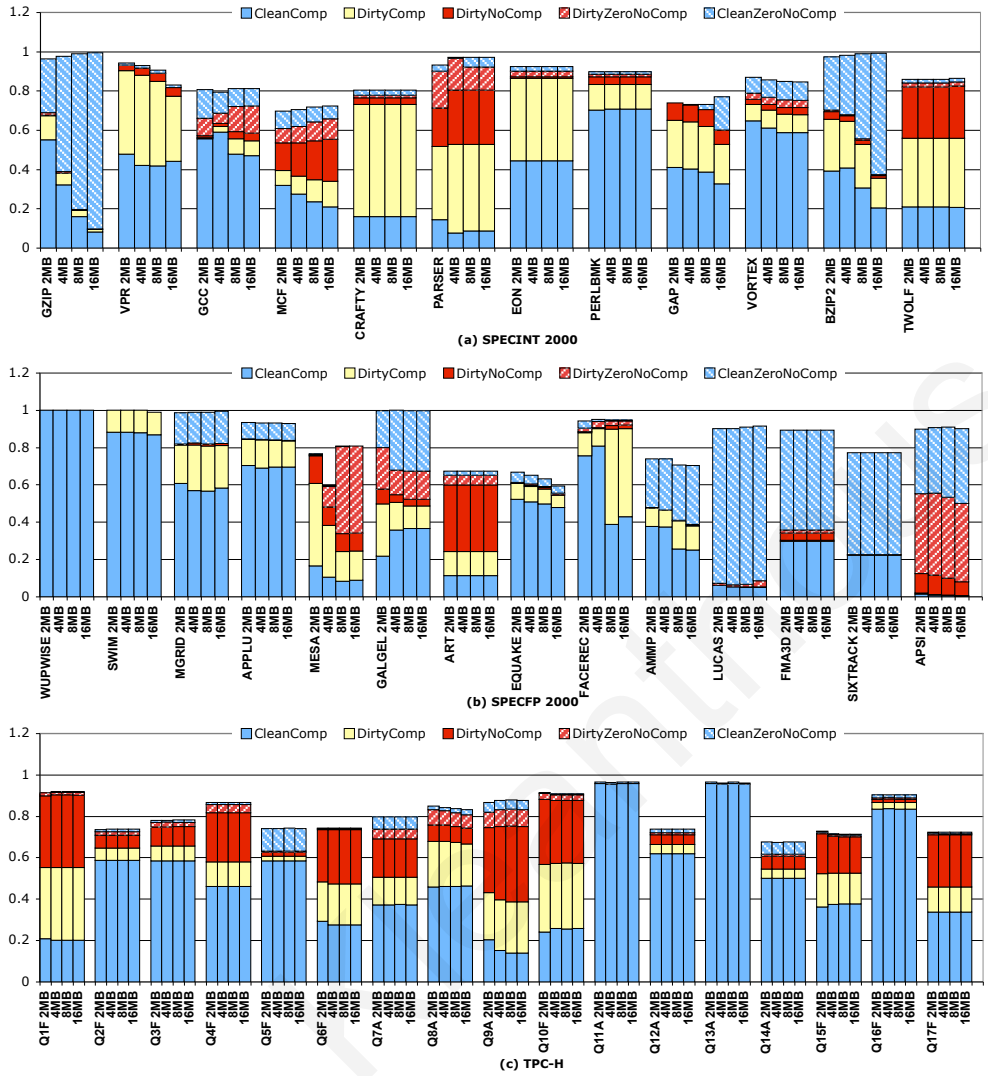

Figure 42: Normalized cache size required after CCD elimination at the granularity of 16byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

Recalling the limit study results of Figure 2, also shown here as Figure 45, we can see that all TPC-H benchmarks have no potential to improve their performance with any LLC cache optimization. Furthermore, from the same figure, we observe that SPEC2000 benchmarks can be classified to 3 categories, low, medium and high, based on their LLC improvement potential. Figure 46 adds more information by providing the LLC Misses Per 1K instructions for several cache size for all SPEC2000 benchmarks.

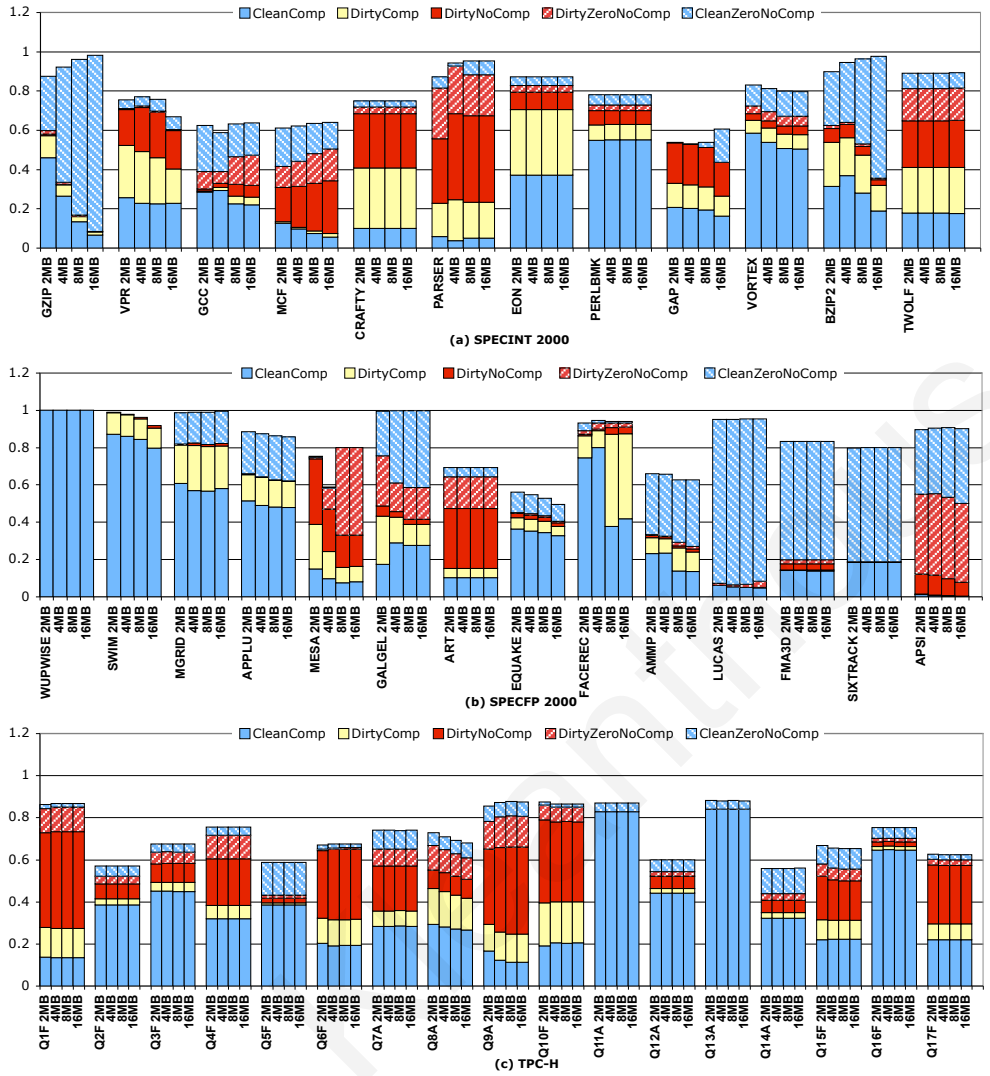

Figure 43: Normalized cache size required after CCD elimination at the granularity of 8byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

Based on the above observations we decided to use a smaller subset of the benchmarks for the rest of our experiments. Table 8 shows our SPEC2000 benchmark classification based on these figures. The benchmarks that do not appear in the table have almost no potential, and we will not be using them for the rest of this Chapter. Benchmark MGRID, although it has high performance potential due to its low IPC, it has much lower Misses Per 1K than the other benchmarks in that category so we have decided to included it in the Medium category. Finally, benchmark SWIM will not be used any further since we have decided to use only 4 benchmarks, from each category.

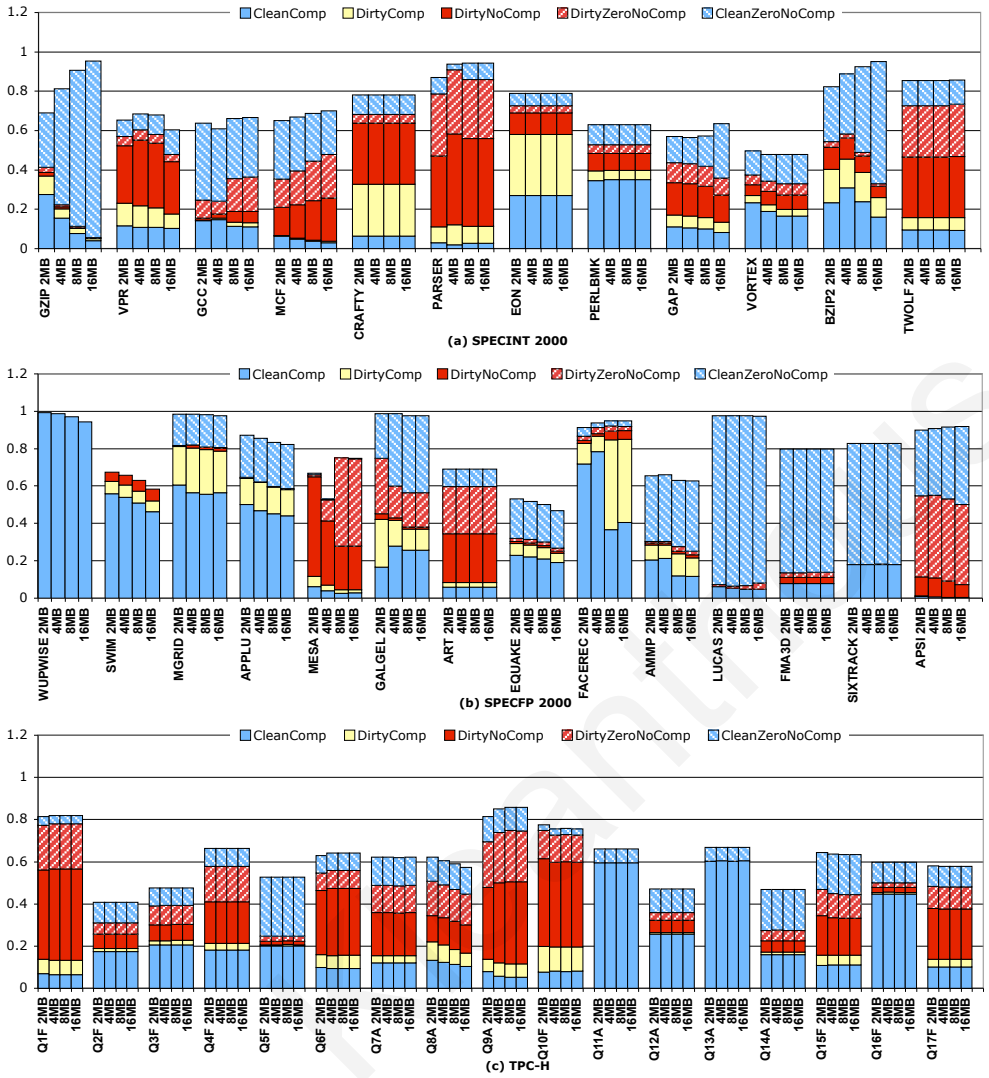

Figure 44: Normalized cache size required after CCD elimination at the granularity of 4byte segments, a) SPECINT 2000, b) SPECFP 2000, c) TPC-H)

Table 8: Benchmark Classification based on their LLC cache pressure and performance potential

High	Medium	Low
MCF	VPR	GZIP
APPLU	MGRID	GCC
EQUAKE	AMMP	WUPWISE
LUCAS	GAP	FACEREC
SWIM		

Figures 47 and 48 shows the results of CCD in a shared LLC cache for 16byte block segments for various cache sizes from 1MB up to 16MB for all the combinations of the 12 selected benchmarks. The first graph, Figure 47 shows all the combinations of High - High (6 combinations),

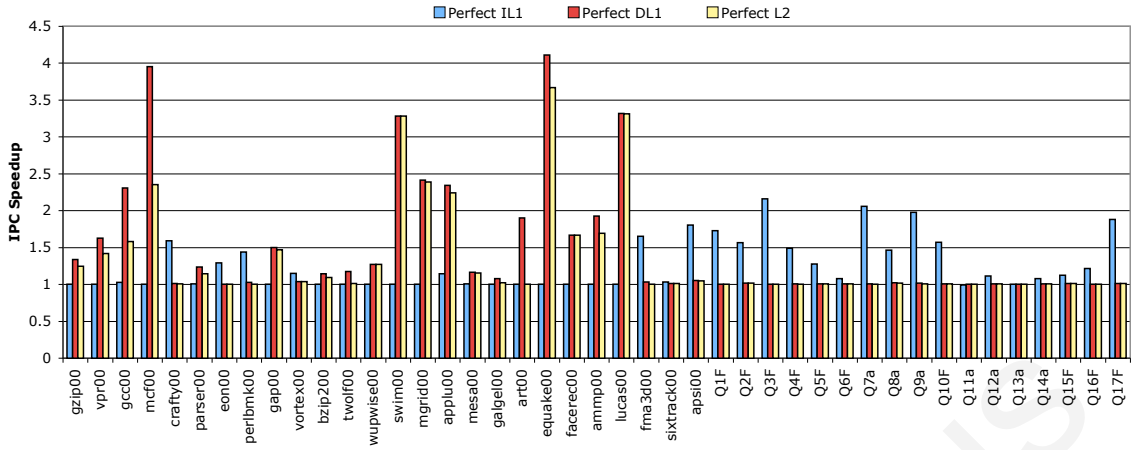

Figure 45: Performance improvement of an out-of-order processor with perfect cache (Same as Figure 2)

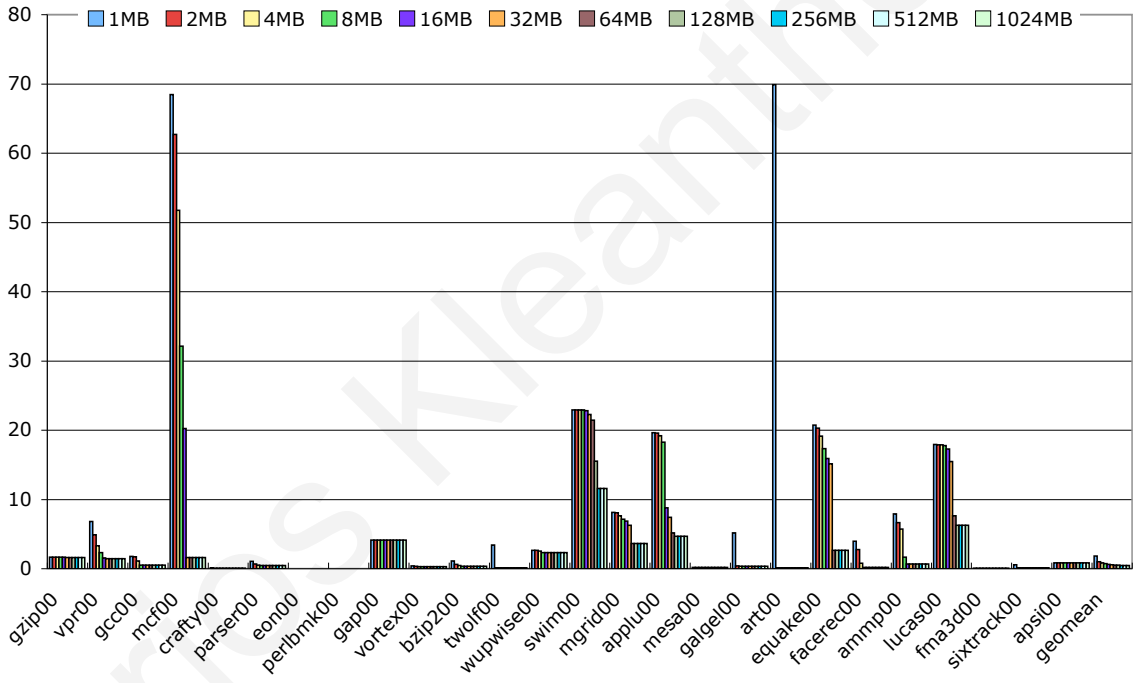

Figure 46: Misses Per 1K instructions for various LLC cache sizes)

Medium - Medium (6 combinations) and Low - Low (6 combinations) benchmarks. The results show a severe reduction in dirty blocks, and especially the duplicated dirty blocks (DirtyNoComp). Furthermore, the duplication to zero blocks appears to be only due to very few benchmarks, such

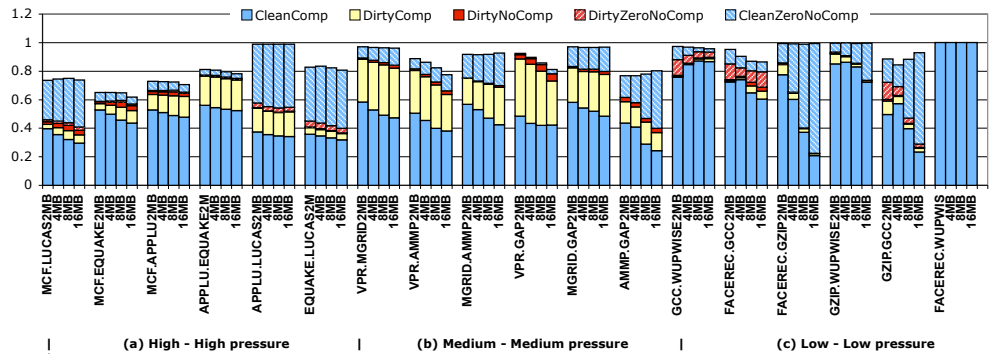

Figure 47: Normalized cache size required after CCD elimination at the granularity of 16byte segments for a) High - High, b) Medium - Medium and c) Low - Low pressure benchmark combinations

as LUCAS, GAP, and GZIP. This suggests we should also consider the duplication for non-zero segments.

Figure 48 shows the results for the rest of the combinations of the 12 benchmarks. We can see that one benchmark with high or medium pressure is enough to make the compression in an LLC cache necessary. The results suggest again that the zero runs are only dominant for the three benchmarks mentioned earlier, and a large amount of compression comes also from non-zero segments for all other combinations. It is important to note that a mechanism tuned to detect duplication regardless the block content will also detect zero segments.

Differentiating the dirty from the clean blocks and the zero from the nonzero at various granularities gives a better insight on how to build an efficient mechanism to detect and exploit this type of duplication. As far as we know, this type of characterization has been published for the first time for the L1 data and Last Level cache.

7.3 Exploiting CCD on Last Level Caches

The results in Section 7.1 suggest that there is more potential for LLC caches compared to L1 Data caches. Also, the high pressure by the multiprogram workloads suggested in Section 7.2 in

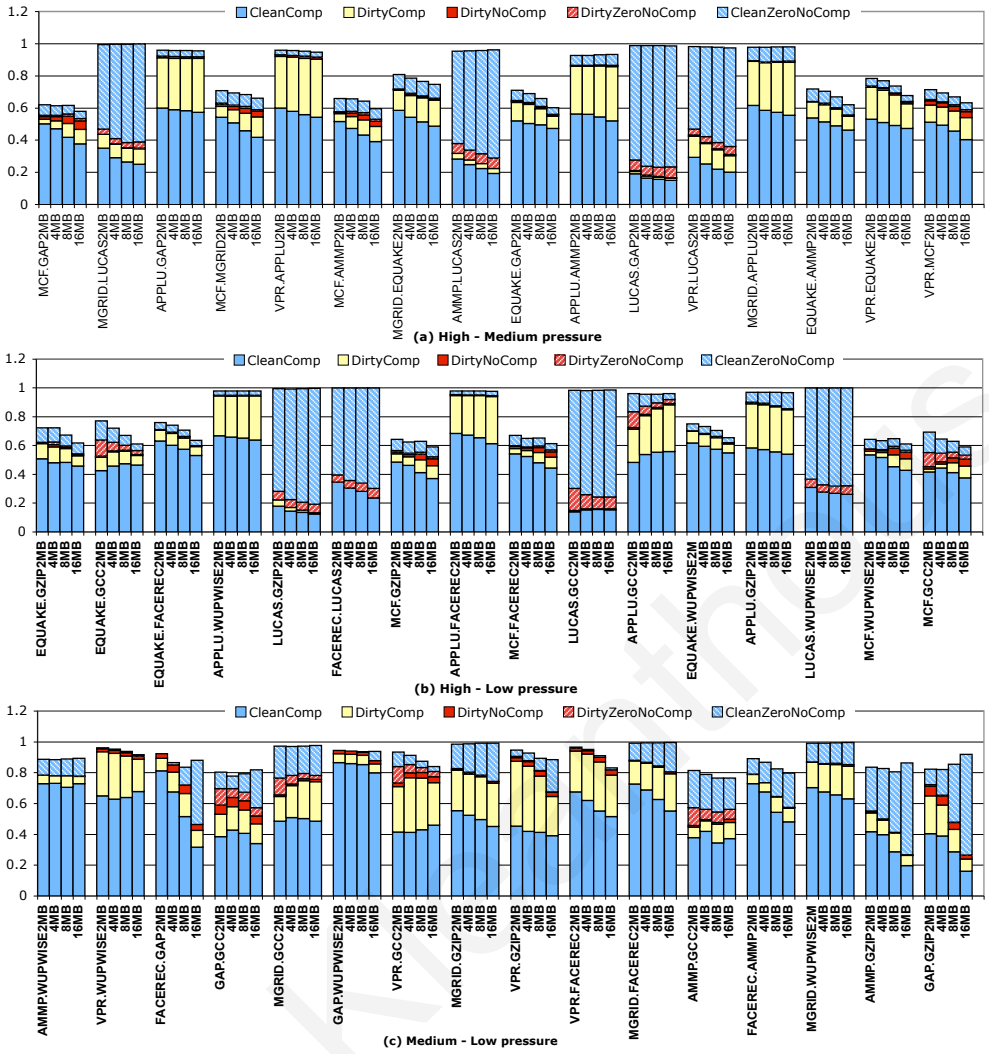

Figure 48: Normalized cache size required after CCD elimination at the granularity of 16byte segments for a) High - Medium, b) High - Low and c) Medium - Low pressure benchmark combinations

combination with the technology trends that lean toward multicores, suggest that it will be more reasonable to investigate the potential of CCD based cache for these workloads.

From the previous results we observed that we have benchmarks that definitely need the extra cache space, but we also have benchmarks that can afford much smaller caches. What we propose is to find a way to decouple the tag array from the data array but at the same time to be able to keep fewer data when possible with the same number of tags. This can happen if the data array is compressed and multiple tags from the tag array are pointing to the same data in the data array.

Such a scenario will give the ability to switch off parts of the data array when it's not used in order to save energy or to have extra tags that can be enabled in the case of program footprints larger than the baseline cache but highly compressible.

Furthermore, previous research indicates that the dirty blocks are hard to be handled in compress caches due to the need of unpredictable space requirements after a write. Also, our results suggest that the contribution of dirty blocks in compression is significant, especially in Last Level Caches. What we propose is to differentiate the dirty and the clean blocks in the compressed data array to make the compression algorithm and cache designs simpler.

The next subsection will propose a new, CCD based, cache design called, Content Duplication Aware (CDA) Cache, and explain in more detail the above assumptions. The aim of this new design is to accommodate the large amount of duplicated relations and relieve high pressure from Last Level Caches.

7.3.1 Content Duplication Aware (CDA) Caches

Figure 49 shows the proposed cache design for the CDA cache. In the figure the Tag and Data array are now decoupled. The indexing and tag matching in the Tag array is done using the block address while the indexing and matching in the Data array is done using the content of the block for Content Based blocks and the block address for the Address Based blocks. These definitions will be explained in more detail in the rest of this section.

The figure shows also all the necessary extensions in each array, Tag and Data:

- **Tag array:** we propose the use of pointers in each Tag to point to their appropriate data. Each tag can have more than one pointer and this depends on the select segment size for compression. For example, if the logical block size of the cache is 64bytes but the physical segment size of the data array is 16bytes then each tag will need 4 pointers to the data array

to build one 64byte logical block. Next, we propose to have extra ways in the Tag array to be able to accommodate more data when an application has a compressible data array and also requires more tags. A more thorough analysis will be presented later in the limit study explaining the benefits of the extra tags.

- **Data array:** we assume the availability of a link list, or a similar structure, for each segment that can keep all the backward pointers to the tags pointing to it. Initially, we assume an infinite size of this list for the limit studies but we explain later several approaches to approximate this solution and provide a feasible implementation. We will refer to this list as Tag List. Aside from the Tag list we also require a single bit to differentiate between Address Based (AB) blocks and Content Based (CB) blocks. We define as AB blocks each segment that is correlated to a dirty tag in the tag array. An AB block can only be correlated to a single tag from the tag array. All other blocks, the CB blocks, are correlated with clean tags from the tag array and can have multiple pointers pointing to them. Finally, we propose the use of a gated data array to allow switching off part of it when the application footprint is compressible and does not require any larger cache. This will enable to switch off part of the cache to save both static and dynamic energy without affecting the performance significantly.

7.3.2 Accessing and Updating a CDA Cache

This subsection will explain in detail each step in accessing and updating the CDA cache. There are three main events that need to be analyzed, the Cache hit, the Cache miss, and the Cache write.

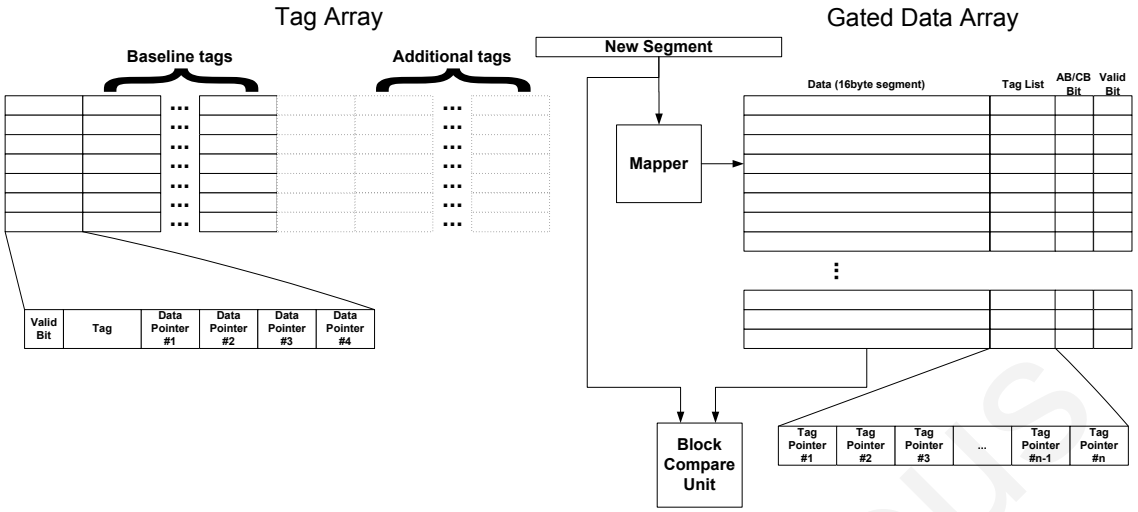

Figure 49: The functional componets of the proposed Content Duplication Aware Cache

- **Cache hit:** On a cache hit there are three steps to access the tag and the data. First, the Tag Array is indexed and tag matched with the address of the block. Assuming we have a hit, then the pointers are extracted and we use each pointer to retrieve the appropriate segment of the logical cache block. Once all segments are gathered, the block is build and send up to the higher level in the memory hierarchy. There is no possibility for a pointer miss in the data array because as we will later explain we assume that if a segment is replaced from the data array then we will invalidate all its correlated tags in the tag array.

- **Cache miss:** On a cache miss the requested block is fetched from lower in the memory hierarchy as usually. Once the new block arrives in the CDA cache, the first step is to index and insert its tag in the tag array with the missed address. We always do this first, such as on tag replacement, the replaced tag might invalidate some or all of its segments in the data array and thus creating more room for the new segments of the missed tag to be inserted.

  In the case of a Valid Tag Replacement, if the block is dirty then it will be pointing to AB segments and thus their content will be written back to the lower level, and all the segments will be invalidated. If the block is clean then, for each segment, we need to index

the data array using the pointer to find the appropriate segment and we remove the tag from each segment's Tag List. If no other tag is pointing to that segment then we invalidate the segment, and it will be sent to the tail of the LRU list to be replaced on the next insertion. **In the case of an Invalid Tag Replacement** then we don't have to do anything since all its appropriate segments were either invalidated or had their Tag Lists updated.

So far the functionality of the CDA cache, as compared to a normal, remains the same with the exception of the need to use the pointers for invalidating the segments or updating their Tag Lists. Once the new tag is inserted in the tag array then we also have to add the new data in the data array. As a first step the 64byte block is split into the appropriate number of segments, depending on their size.

If their is a match and a **Segment Hit** we add the tag to the segment's Tag List and we update the pointer of the tag in the tag array. On the other hand, in the case of a **Segment miss**, we need to insert the segment in the data array and update the pointer in tag array. The insertion of a new segment in the data array will most probably cause the eviction of another segment from the LRU list. If the replaced segment is valid then, using its Tag List, we will track all the tags that point to this block and invalidated them. If any of these tags is dirty then it needs to be written back also all the segments pointed by the invalidated tag, both clean and dirty, need to be updated by removing the invalid tag from their Tag List. If there is an invalid segment replacement there is no action need to be taken.

- **Cache write:** Finally we have the scenario of a cache write in the CDA cache over a **clean** block already in the cache (assuming no write allocate policy). First, we need to index and tag match the tag array with the address. Then we need to extract all the pointers to the segments and copy all data into a new Address Based block in the data array. For each

Content Based segment we have to remove the tag from its Tag List and if no other tag is pointing to that segment then we will also invalidate it and send it to the tail of the LRU list to be replaced on the next insertion. In the case of a write on an already **dirty** block then we only need to update the appropriate segments with the new content.

7.4 Initial Results of a CDA Cache

This section will show the performance potential of a CDA Cache. Our baseline is an 8MB cache as described in Chapter 3. The extensions of the CDA cache will enable the baseline cache to increase its number of ways per set up to 32 and to decrease the required space in the data down to 1MB (while keeping tag array, ways and sets, constant) for both single program and multiprogram workloads. In this study, we assume an infinite size of tag list and the data array is fully associative to avoid any conflicts due to the hash function of the content. These assumptions are made to define the limits of a CDA cache and will be discussed later how they can be implemented.

7.4.1 Single Program Workloads

At first, we show the potential of single program workloads only for the subset selected in Section 7.2. The benchmarks are sorted based on their cache pressure, as in Table 8, from high to low pressure. Figure 50 shows the normalized performance of various cache sizes all normalized to the 8MB cache. For the caches smaller than 8MB we reduce the number of sets and maintain the 16-way set associativity while for caches larger than 8MB we increase the number of ways per set to achieve higher capacity. This is done to be able to compare the baseline results later with the CDA Cache design.

The results indicate that most of the benchmarks are affected by both increasing and decreasing the cache size but there also few of them that are insensitive to these changes. This can happen

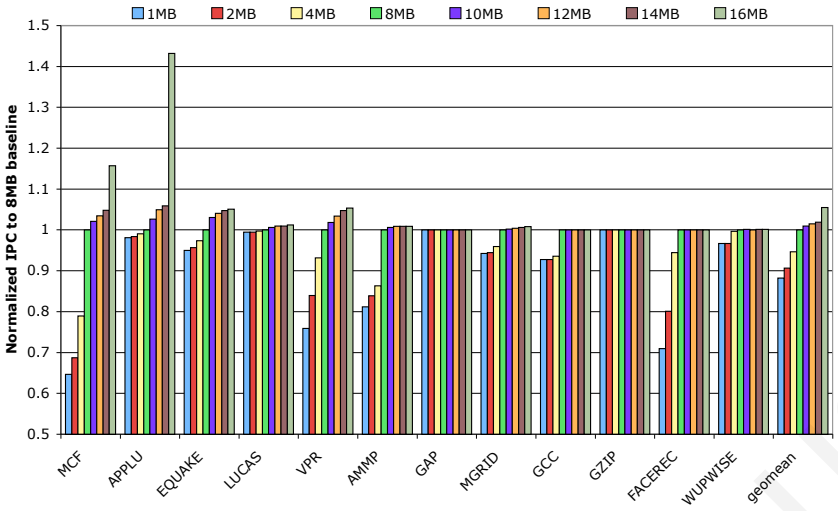

Figure 50: Normalized IPC speedup on the 8MB baseline for various cache sizes

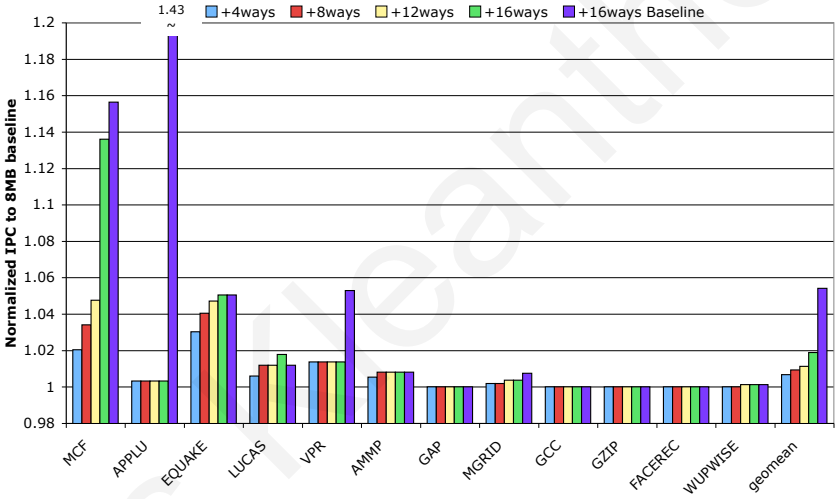

Figure 51: Normalized IPC speedup on the 8MB baseline when increasing tag array

either because the benchmark has a very small footprint so it fits even in the smaller cache or because the benchmark's footprint is extremely large or has a streaming behavior and is causing the same number of misses for all cache sizes. This observation can be validated using Figure 46 where the cache misses are shown for caches from 1MB up to 1024MB. We can clearly see, comparing the two figures, that GAP for example is insensitive for all cache sizes, where LUCAS needs at least 32MB cache for the miss reduction to be noticeable and up to 128MB to fully fit its footprint.

Assuming a CDA cache, with 16byte segment size, as described in the previous section, Figure 51 presents the performance of this cache when increasing the number of available tags but keeping the data array at 8MB and compressed. The results indicate that MCF, EQUAKE, and LUCAS can benefit from the extra tags by improving their performance up to 13%. We have to note here that this performance improvement of this scheme is limited by the tag array associativity. To be exact, a CDA cache with +16 ways will have, in the best case, the same performance as a 32 way 16MB regular cache. The only difference is that the CDA cache will use only 8MB for the data array.

Comparing the results with Figure 50 we can see that the CDA cache can achieve almost the limit for MCF, 13% as opposed to 15% for a regular 32-way 16MB cache, and few other benchmarks. On the other hand, we observe that CDA cache cannot improve the performance of APPLU, although bigger regular cache can improve its performance by 43% with 32-ways and 16MB size. This behavior is explained with the results presented in Figure 42 where it is shown that APPLU has very little compression potential, and thus the information provided by the extra tags cannot be accommodated to the 8MB data array as opposed to MCF where from the same figure it shows 40% compression and that means the 8MB data array can accommodate almost twice the number of tags of the regular cache when compressed.

While increasing the Tag array is a way to achieve performance improvement, power gating and switching off the data array is a way to improve the energy consumption. As we argue at the beginning, some benchmarks need less cache because their data can be compressed to occupy less space. Figure 52 presents CDA cache, when using compression and we can switch off parts of the data array to save energy. The results show that for most of the benchmarks we can switch half the cache without affecting their performance or by affecting it less that it would with a smaller regular cache. For example, if we again look at MCF we can see that the regular 4MB cache

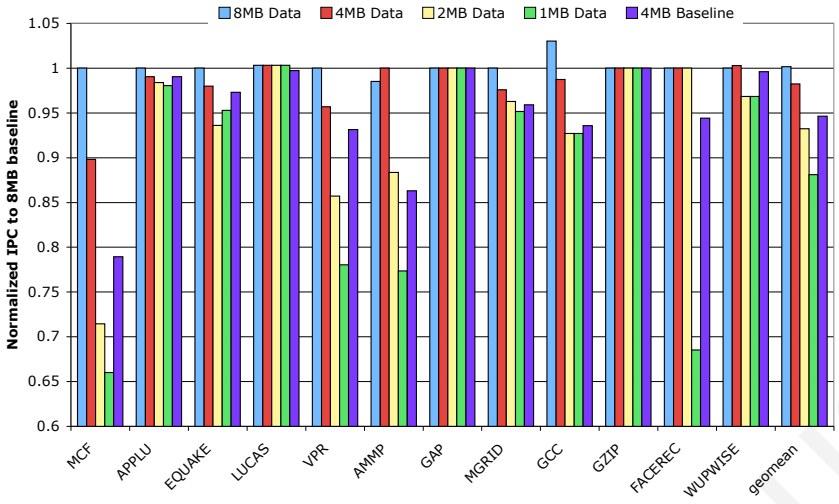

Figure 52: Normalized IPC speedup on the 8MB baseline when decreasing data array

from Figure 50 reduced the performance by more than 20% as compared to the 8MB baseline, while the CDA cache, using a compressed 4MB data array, affects the performance by only 10% as compared to the same baseline.

Overall the results indicate that the CDA cache is worth investigating more. Especially if the CDA cache could dynamically decide to either switch off part of data array to save energy or to switch on the extra ways to improve performance.

7.4.2 Multi Program Workloads

Figures 53 and 54 show the performance of all the multiprogram combinations of the 12 selected benchmarks. We observed that the multiprogram workloads are more sensitive to cache reduction, as expected, compared to the single program workloads and thus giving us more room for improvement

Figures 55 and 56 show the performance achieved by using the CDA cache and increasing the number of ways for better performance. The results show that by increasing only the ways and keeping the same data size, only a small number of benchmarks benefits. The potential is

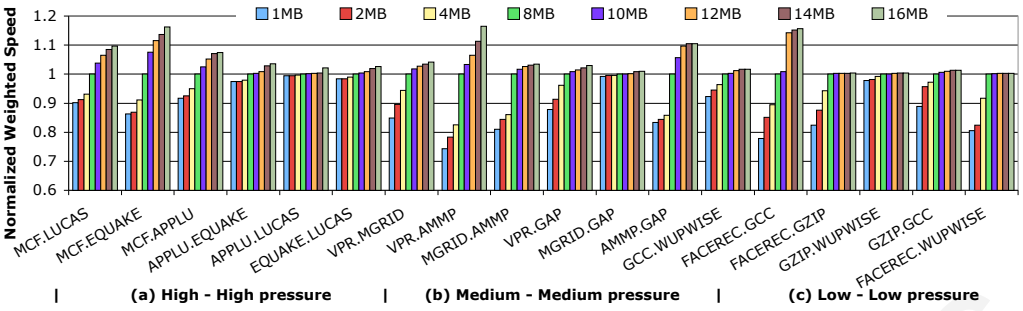

Figure 53: Normalized IPC speedup on the 8MB baseline for various cache sizes for a) High - High, b) Medium - Medium and c) Low - Low pressure benchmark combinations

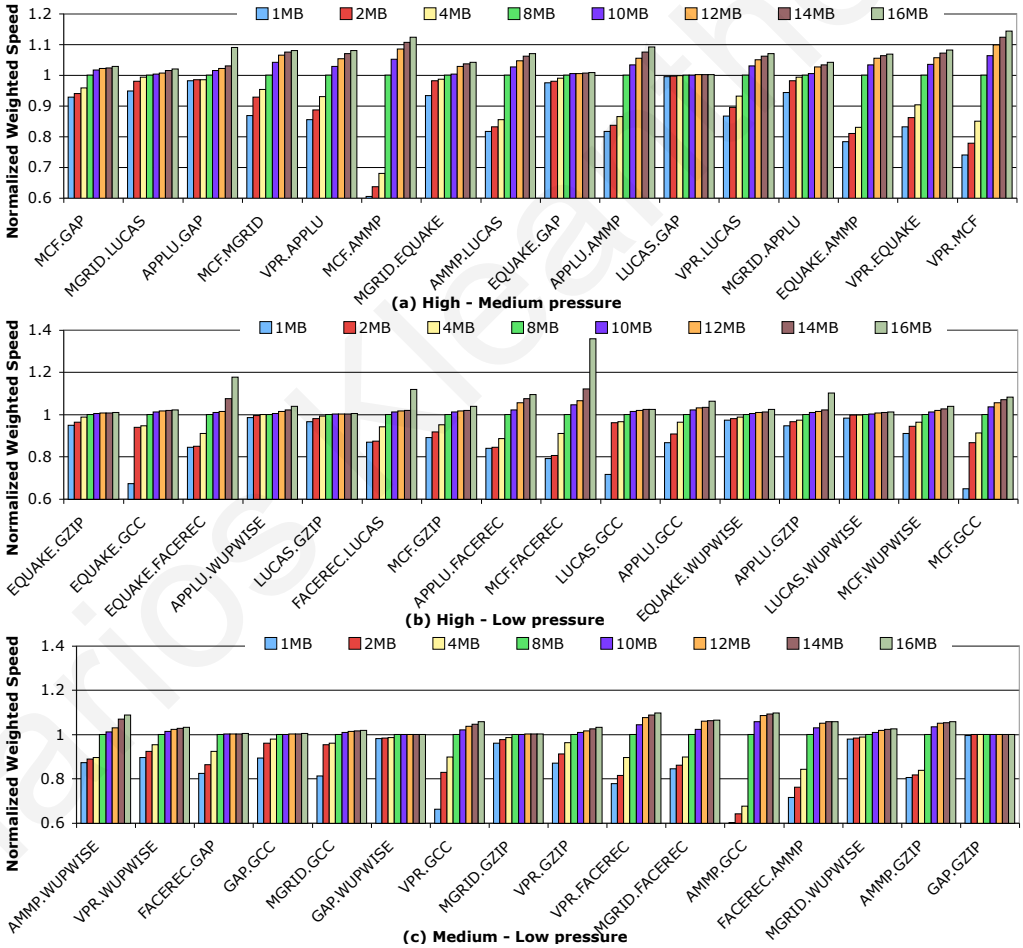

Figure 54: Normalized IPC speedup on the 8MB baseline for various cache sizes for a) High - Medium, b) High - Low and c) Medium - Low pressure benchmark combinations

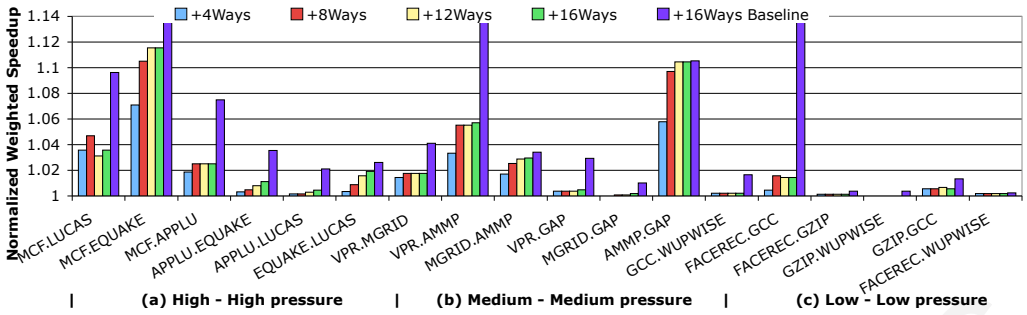

Figure 55: Normalized IPC speedup on the 8MB baseline when increasing tag array for a) High - High, b) Medium - Medium and c) Low - Low pressure benchmark combinations

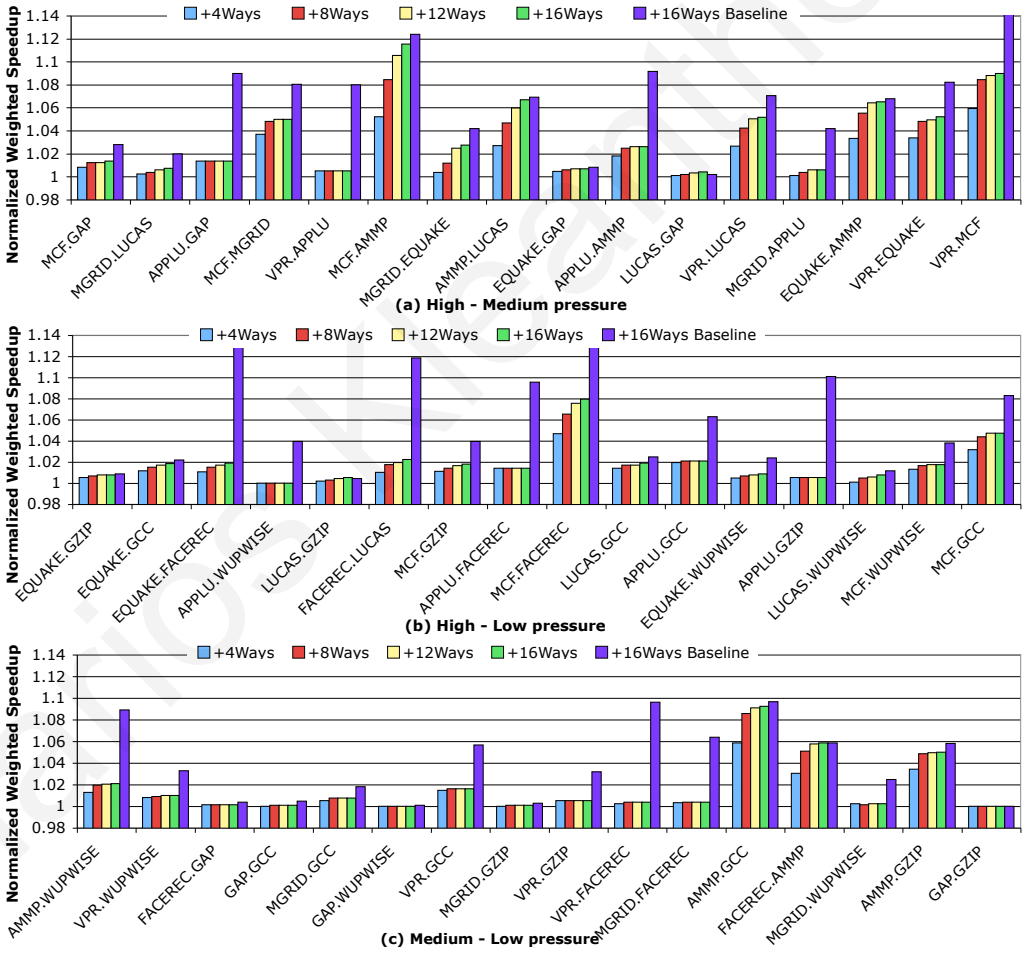

Figure 56: Normalized IPC speedup on the 8MB baseline when decreasing data array for a) High - Medium, b) High - Low and d) Medium - Low pressure benchmark combinations

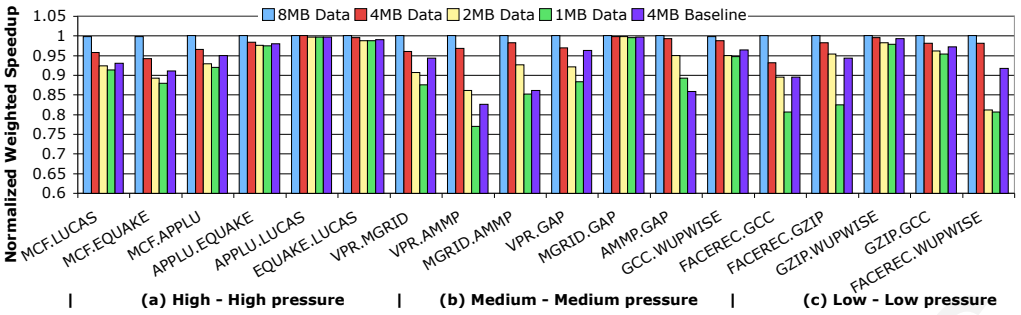

Figure 57: Normalized IPC speedup on the 8MB baseline when decreasing data array for a) High - High, b) Medium - Medium and c) Low - Low pressure benchmark combinations

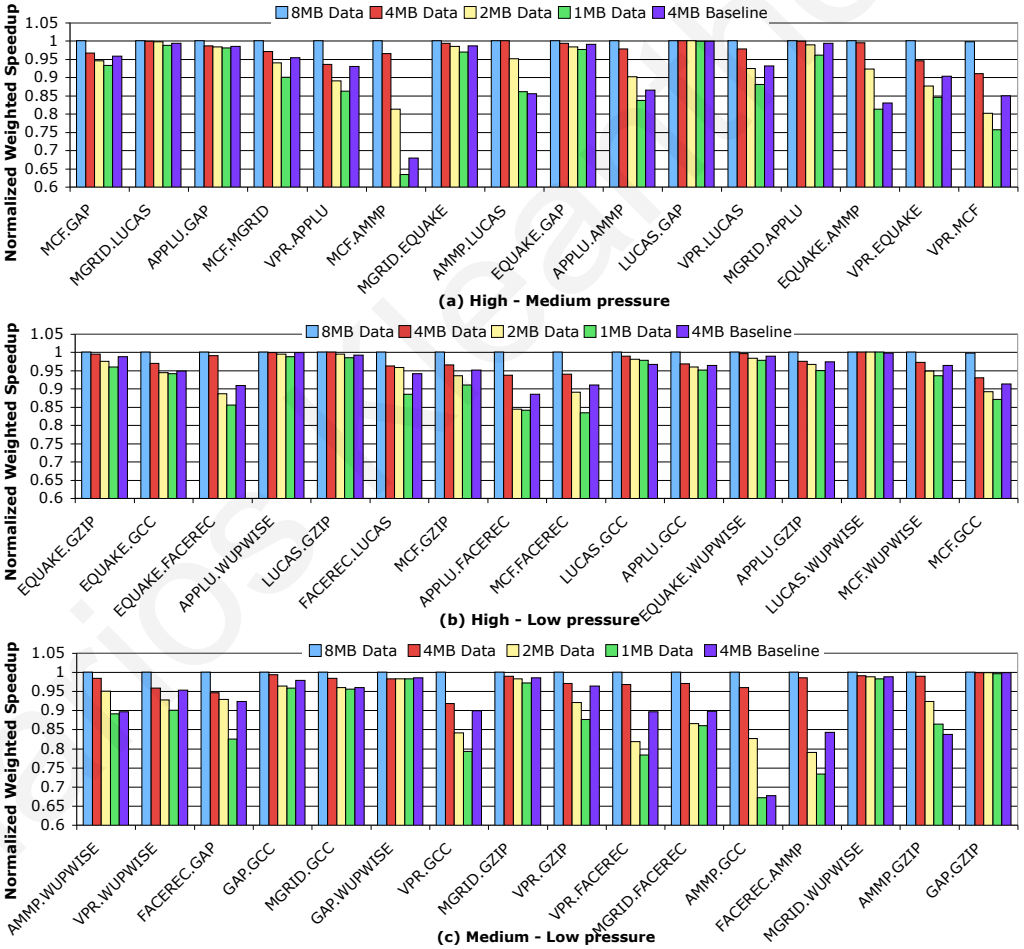

Figure 58: Normalized IPC speedup on the 8MB baseline when decreasing data array for a) High - Medium, b) High - Low and c) Medium - Low pressure benchmark combinations

at best at 12% for few benchmarks. As compared to the 16MB baseline cache we can see that the performance potential of CDA is very limited in several cases but there is also a significant number of benchmarks that the +8way CDA cache is a good tradeoff. For example, AMMM.GAP from Figure 55 shows that the performance of a 16MB regular cache can be approximated with an 8MB +8Ways CDA cache and can be match with +12Ways. Overall the results appear to be encouraging.

Figures 57 and 58 show the performance achieved by using the CDA cache to reduce only the data array and keeping the tag array the same size. By reducing the data array and keeping it in a compressed form we can achieve very close to the original performance, and saving much of static energy from the switched off banks. Comparing the baseline with the CDA results we count 56 benchmarks, out of 66 in total, with an IPC reduction of less than 5% for a 4MB data array. The respective baseline counts only 35 benchmarks with a reduction less than 5% and the rest of them show reductions by more than 15%. This indicates that the compressed design of the CDA cache can handle more data in the 4MB data array than the baseline cache using the extra tags and thus achieve less performance loss.

Overall the conclusion from this limit study is that using the CDA cache to improve performance is hard to achieve for all benchmarks and can be beneficial for only a small subset of them. On the other hand keeping the same tag array and gating the data array might have more potential since it appears from the limits that the performance loss is very small. As compared to the single program workload results, here CDA appears to have more potential due to more pressure in the LLC cache. To investigate further this scenario we will measure the Energy Delay when reducing the CDA cache's data array as compared reducing the size of a regular cache.

7.5 CDA Cache Energy Delay Characterization

This section uses a first order energy model to estimate the performance gains of the CDA cache when reducing the data array. We made the following assumptions based on CACTI analysis:

1. We assume that 10% of the LLC's total energy is consumed in the Tag array while 90% is consumed in the Data array.

2. We assume that 90% of the energy consumed by the Tag array is due to leakage while 10% is due to dynamic energy.

3. We assume that 90% of the energy consumed by the Data array is due to leakage while 10% is due to dynamic energy.

Furthermore, based on [76], we assume that our cache has 1 bank per 1MB, so a total of 8 banks for our baseline cache, and the energy consumption of the LLC cache is 30% of the total core energy.

Based on the above, we produce the Figure 59 where three lines indicate the effects on energy when increasing the execution time, and decreasing the CDA cache data array compared to decreasing the size of a regular cache. The model assumes that from the total energy equation the only parameter that is affected when reducing the data array is the static leakage consumed by the data array while for the regular cache is both the tag and data array static leakage. Furthermore, as we said before, we assume that the relation between the data array size and static leakage is linear. That means when, we cut the data array in half, then the static leakage consumed by the data array is also cut in half. Finally, the model assumes a 1% always better performance of the CDA cache compared to a regular cache with the same size data array. This is done to give an estimation of

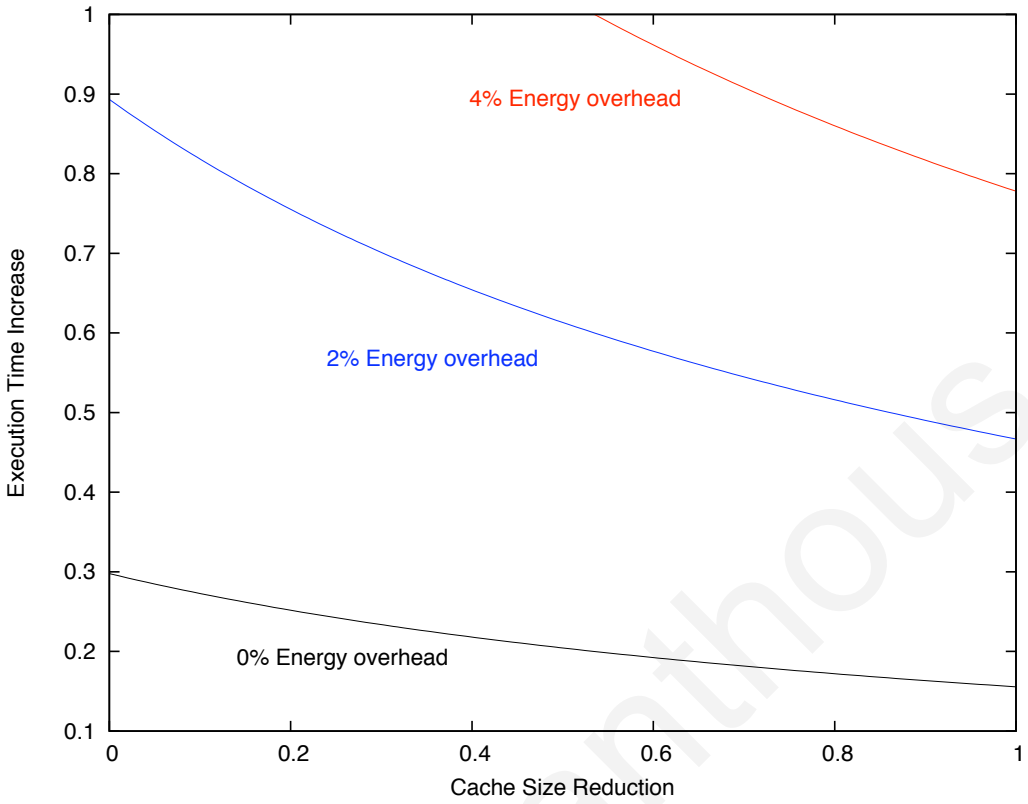

Figure 59: Energy profiling of increasing execution time and decreasing LLC data array

the Energy Delay improvement of the CDA cache assuming its performance will be always better as compared to the regular cache due to the extra tags. In the case, where the performance of CDA is equal to the regular cache then Energy Delay is lower since we always static leakage for the extra tags.

What Figure 59 provides is the limits of the CDA cache based on the performance and cache size reduction trade off assuming that the CDA cache has at least 1% better performance than the regular cache. For example, it shows that by decreasing both CDA and a regular cache size by 50% and increasing the execution time by up to 20% we have always better energy efficiency using the CDA cache because the point is below the 0% Energy overhead line. Any point below the 0% Energy overhead contour indicates that the decreasing only CDA cache data array saves more energy than a decreasing a regular cache's size by the same portion.

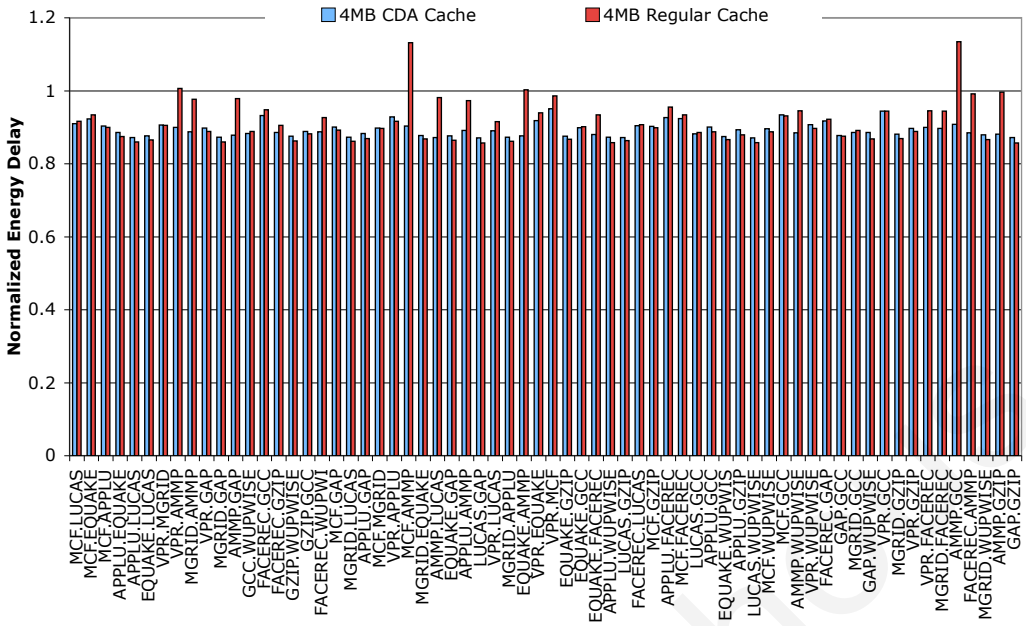

Figure 60: Normalized Energy delay for a 4MB CDA cache (with double the number of tags) and a 4MB regular cache

The results from the previous section indicate that reducing the data array to 4MB, we can maintain the performance in reasonable levels (similar or better to reducing the baseline to 4MB) and increasing around 5% the execution time for most of the benchmarks and 7% on average. Setting this point on the Figure 59 it indicates that for this scenario, and given we have better performance than the 4MB baseline, the CDA cache will save more energy overall.

To backup our model we present the exact results for the Normalized Energy Delay, Figure 60, for all benchmarks when reducing the CDA cache data array to 4MB and when using a 4MB regular cache. The results show that when there is the need for more performance, for example, benchmark MCF.AMMP, the CDA cache can handle the need for more tags by compressing the cache and achieving a total of 10% energy savings, while the regular 4MB cache presents a huge increase in the execution time, about 30% (as shown in Figure 54), which results in 15% increase in the energy consumption. The energy model used here is based on CACTI results and activity factors of a cycle accurate simulation for each benchmark. Overall the results indicate that CDA

cache has the potential to be a more energy efficient cache design as compared to a regular smaller cache.

7.6 Implementation Issues of CDA cache

The CDA cache as described above has significant potential, for saving energy, but also has functional requirements that need to be investigated further and be solved in future work.

The following list describes all the functional requirements and suggests possible solutions:

- **Data pointers to tags**: Our scheme assumes that each tag has the ability to keep a pointer to each segment in the tag array to rebuild its logical block. The pointers will impose a significant overhead to the mechanism regarding the required area. A way to overcome this limitation is by moving the ECC bits from the data array to the tag array. By doing this, the ECC bits can be used for indexing and pointing the data array. Since the ECC bits are always checked and the Tag is read before the data this modification will not affect the latency of the ECC validation. In this way, the overhead will be reduced significantly to only the ECC bits for the extra tags.

- **Indirect data array access**: On a normal cache the data array is accessed by matching the appropriate tag. In the CDA design the Tag array is first needs to be accessed and then the pointers to the data are extracted from the matched tag and are used to indirectly access the segments in the data array. This indirection we assume that will have a very small overhead over the whole caches access procedure but needs to investigated and evaluated in future work.

- **Building cache block**: As opposed to a normal cache, the block in a CDA cache needs to be build from its segments before is send to the higher level in the memory hierarchy. Because cache's physical implementation already involves subarrays, we believe that current cache designs may naturally fit the segmented solution needed by CDA. This requirement is something that needs to be evaluated further, in the future, both in respect of energy but also the delay that is needed to build the blocks.

- **Fully associative data array**: The CDA cache data array is assumed to be a fully associative table in all the experiments so far in order to measure the limits of the mechanism. We have also initial analysis with a 64way set associative data array that shows similar performance as the fully associative data array. As discussed in the previous bullet the ECC bits are used for indexing and pointing to the data array. We believe that a fine tuning in the ECC hashing function by permuting few bits will achieve even better distribution of the segments in the data array.

- **Dirty block**: Dirty blocks have always been a problem for compression designs because when the data are overwritten often require more space than the compressed data [49]. Another problem is with mechanisms that use correlations between duplicated blocks [46]. The problem arises when a data is written with another value and it's no longer duplicated for the two, or more, tags that are pointing to it. Our solution to this is to keep the dirty blocks uncompressed as explained in the 7.3.2. By separating the dirty and clean blocks to Address based and Content based it provides us the ability to handle dirty blocks better and handle the writes with no extra effort since we ensure that only one tag is pointing to a dirty Address based block. The delay and energy overheads of creating the AB blocks need to be investigated further.

- **Tag invalidation on data eviction**: The final problem is the functional requirement to invalidated tags that are pointing to a segment that is being evicted. A segment can be evicted while still valid tags are pointing to it if the data required by the valid tags cannot be accommodated by the compressed data array and the limited number of segments in it. There are few ideas to solve this problem without the need of long tag lists as proposed in the limit study.

  - **Limit number of tags per segment**: The simplest idea to overcome this problem is to limit the number of tags that can point to each segment. By doing this, we can have a limited number of backward tag pointers for each segment. These pointers will be used to invalidate all relevant tags when the segment is evicted. Furthermore when a new tag needs to point to a segment that is already full of tags then the content needs to replicated to another location and the new tag will point there. This approach is simple to implement but will cost in performance since will limit the compression potential of a workload. One way to absorb some of this lost potential is to **keep narrow values in the tag** in the place of the pointers. By having an extra bit in pointer in the tag we can choice that if the content is a pointer to the data array or a narrow value. A narrow value is defined as a value that requires maximum bits as the number of bits available for the pointer. This idea has already been proposed and evaluated by Molina et al. [47]. Another solution is to keep not only small values but all the frequent values, after profiling, encoded using a frequent value table similar to [45].

  - **Bloom filter**[90]: Another idea is to use a bloom filter in front of the tag array and each time a pointer to a segment is updated in a tag the relevant bit in the bloom filter will be set. On a data segment eviction, the bloom filter will be accessed with the

content and will point to valid tags that potentially contain this pointer. Then this tag can be either greedily invalidated or can be checked if indeed contains a pointer to this segment and then invalidated.

- **Link list embedded in the tags**: One last idea is to implement an efficient double linked list in the tags by using two extra tag-pointer for each data pointer. The first will point to the previous tag and the other to the next tag containing using this segment. Also, a backward pointer in the data segments is needed that points to the last tag inserted. In this way when a data segment is replaced, its backward pointer will be used to track the last tag that points to the segment and then the list will be traversed to invalidate all tags that also point to that segment. This list can be accommodated in the data array itself, into unused data segments. This will limit the available space for storing data but, assuming we have a good compression ratio, this loses will be compensated due to the additional tags.

All ideas mentioned here need to be investigated further, both the cost and performance, in future work.

7.7 Chapter Summary

In this chapter we have evaluated the performance potential of a new cache design, the CDA cache, for both single and multiprogram workloads.

The trends indicate that with the granularity that we detect the duplication is decreasing the duplication is increasing. For 16byte data segments granularity for single program workloads the

results indicate usually more than 40% compression ratios and up to 90% for LUCAS benchmark when compressing only clean blocks. When compressing both clean and dirty blocks the compression ratio is usually more than 60% and close to 99% for benchmark APSI.

Furthermore, for the multiprogram workloads the we usually have more than 50% compression and in many cases up to 80% for clean blocks. An interesting result is that the contribution of dirty blocks in compression for multiprogram workloads is much lower and usually less than 5%.

Finally, the CDA design indicates moderate performance improvement, close to 5% when adding 2 extra ways, but it shows that can reduce the Energy Delay product considerably, 10% on average and up to 15% at most, for multiprogram workloads.

The potential appears to be more promising on multiprogram workloads for the Last Level Caches mainly due to the higher pressure. Also, the direction of reducing the data array to save energy appears to be more appealing since it achieves lower Energy Delay and can maintain the performance in a great deal as compared to reducing the size of a regular cache.

The CDA cache design proposed here assumes a limit study implementation, and all the functional requirements mentioned in 7.6 need to be investigated further.

Chapter 8

Conclusions

This thesis defines a new cache phenomenon, the Cache-Content-Duplication (CCD). We have shown that CCD exists for various types of instructions caches, data caches, and Last Level Caches (LLCs). Also applications and mechanisms, to exploit the phenomenon, were proposed. In this Chapter, we will provide a summary of the contributions of this work and directions for future work.

8.1 Contributions

The main contributions of this thesis are:

- **Characterization of redundancy**: We provide a characterization of redundancy for both instructions and data. For the instructions, we have analyzed the redundancy at the granularity of valid sequences while for data we investigate this property at the granularity of 32byte segments.

  This thesis also gives a more thorough analysis of the redundancy in data caches. We investigate what percentage of data, both in L1 data cache but also in Last Level Cache, its

for dirty blocks, and zero runs for various segment and cache sizes. This analysis provides a better understanding of the program behavior and points to two important observations. First, the benefit from compressing dirty blocks is not very important so they can be ignored to make the duplication detection mechanisms simpler. Second, the percentage of duplication due to zero runs drops significantly in Last Level caches as compared to L1 data caches, especially in shared LLCs where multiple applications create more pressure. This suggests that it's worth investigating again the benefit for compressing all values in an LLC, both zero and non-zero.

The results indicate that there is high redundancy in many benchmarks that worth more investigation and proposing mechanisms and applications to remove it for both instructions and data. The outcome for L1 data caches was not very encouraging since there is a significant amount of zero runs that have been already investigated before by others. Even though, the results for LLCs were very promising and were investigated further.

- **CCD limit studies**: We preset limit studies that show the potential of CCD for various instructions cache (regular, basic block and trace caches), for L1 data cache and for Last Level Caches. In all cases the limit studies are applied to a wide design space that includes various cache sizes and duplication detection granularities.

- **Proposed mechanisms and applications**: This thesis proposes two mechanisms that can be applied to detect and eliminate CCD to either improve performance or save energy. The first design is CATCH that is proposed to improve the performance of L1 instruction caches, and the results show that a 1.38KB mechanism can capture about 58% of the potential of the limit study which corresponds to about 5% performance improvement.

Also, the Cache Duplicate Aware (CDA) Cache is proposed that be implemented for Last Level Caches to save energy. Initial analysis shows that a limit study mechanism can achieve close to 10% energy reduction with very little performance loss as compared to the baseline and much better performance as compared to reducing the size of a regular cache in few benchmarks.

We anticipate that the content of this thesis will give more insight for the potential of cache compression and content duplication specifically and inspire people to perform research in this area. The characterization of redundancy presented in this work gives new information on the Cache Content Duplication and the CCD limit study indicates that the potential of the phenomenon is worth investigating more in the future.

8.2 Future Work

This thesis provides several directions for future work. One is to investigate other methods to tolerate block differences and lead to higher CCD frequency. A mechanism for zero cycle secondary hit latency may be also useful to design and evaluate. CCD may also be considered in combination with static code compaction to investigate the synergistic potential of the two approaches.

The Text Cloning phenomenon also needs to be investigated further. An extended analysis of Text Cloning in Grid Computing and Cloud Computing frameworks to determine its frequency and performance implications in a realistic setup will be very beneficial for the developers of Grid Computing systems.

Another important direction of research is to consider transformations of CCD for data caches to increase duplication. For example, data sequences can be approximated as valid sequences,

similar to instructions, by using the compiler's knowledge of structs, tables and other data structures. If we were able, with the assist of compiler, to detect these valid data sequences then we could adapt CCD detection at this granularity and increase the frequency and benefits even further.

Finally, the CDA Cache design needs to be investigated further and all the functional requirements mentioned in Section 7.6 need to be implemented and evaluated.

Overall, all the proposed mechanisms and applications in this thesis can be extended further by using the assist of the compiler or by the operating system to improve the performance of modern processors even further.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, March 1995.

[3] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 319–330.

[4] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 357–368, 2005.

[5] A. Shayesteh, G. Reinman, N. Jouppi, S. Sair, and T. Sherwood, "Dynamically configurable shared cmp helper engines for improved performance," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 70–79, 2005.

[6] I. Ganusov and M. Burtscher, "Future execution: A hardware prefetching technique for chip multiprocessors," in *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 350–360.

[7] 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, pp. 364–373.

[8] M. Kleanthous and Y. Sazeides, "The Duplication of Content in Instruction Caches and its Performance Implications," University of Cyprus, Tech. Rep. TR-CS-01-05, January 2005.

[9] M. Kleanthous and Y. Sazeides, "CATCH: A method for Dynamically Detecting Cache-Content-Duplication," ACACES, July 2005.

[10] M. Kleanthous and Y. Sazeides, "Cache-Content-Duplication for Valid Blocks," ACACES, July 2006.

[11] M. Kleanthous and Y. Sazeides, "Dynamically Detecting Cache-Content-Duplication in Instruction Caches," University of Cyprus, Tech. Rep. TR-CS-03-07, February 2007.

[12] M. Kleanthous and Y. Sazeides, "CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication and its Application to Instruction Caches," in *Proceedings of the 2008 conference on Design, Automation and Test in Europe*, March 2008, pp. 1426–1431.

[13] M. Kleanthous, Y. Sazeides, and M. D. Dikaiakos, "Extrinsic and intrinsic text cloning," in *WIOSCA 2010 (held in conjuction with ISCA 2010)*, ser. Lecture Notes in Computer Science, vol. 6161, 2010, pp. 324–340.

[14] M. Kleanthous and Y. Sazeides, "CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication in Instruction Caches," *Transactions on Architecture and Code Optimization*, (Accepted for publication).

[15] M. Kleanthous and Y. Sazeides, "Simulation Region Analysis for SPEC2000 Benchmarks," University of Cyprus, Tech. Rep. TR-CS-01-12, January 2012.

[16] Y. Sazeides, A. Moustakas, K. Constantinides, and M. Kleanthous, "The Significance of Affectors and Affectees Correlations for Branch Prediction," in *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, January 2008, pp. 243–257.

[17] Y. Sazeides, A. Moustakas, K. Constantinides, and M. Kleanthous, "Improving Branch Prediction by Considering Affectors and Affectees Correlations," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 3, pp. 69–88, 2011.

[18] M. Kleanthous, S. Yehia, Y. Sazeides, and E. Ozer, "A Replacement Policy Based on Dynamic Profiling and Hashed Data," ACACES, July 2007.

[19] M. Kleanthous and S. Yehia, "Entry Replacement Within a Data Store," Patent Number 20080183986, July 2008.

[20] P. J. Denning, "Virtual Memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, September 1970.

[21] J.-L. 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*, November 1991, pp. 176–186.

[22] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 138–147.

[23] M. Kjelso, M. Gooch, and S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor," in *Proceedings of the 22nd EUROMICRO Conference*, September 1996, pp. 423–430.

[24] K. D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999, pp. 139–149.

[25] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 445–454.

[26] M. Kharbutli and Y. Solihin, "Counter-based cache replacement algorithms," in *Proceedings of the International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 61–68.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2007, pp. 381–391.

[28] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proceedings of the 7th International Conference on Supercomputing*. New York, NY, USA: ACM, 1995, pp. 338–347.

[29] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[30] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *Proceedings of the 32nd International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 544–555.

[31] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proceedings of the 27th International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2000, pp. 107–116.

[32] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *Proceedings of the 33rd International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 155–166.

[33] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 182–194.

[34] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 252–263.

[35] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual ACM/IEEE International Symposium on Microarchitecture*, 2008.

[36] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *Proceedings of the 33rd International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 167–178.

[37] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper Threads via Virtual Multithreading," *IEEE Micro*, vol. 24, no. 6, pp. 74–82, 2004.

[38] M. Corporation, *Disk Operating System User's guide (DOS Release 2.10)*. IBM Corporation, 1984.

[39] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Storage," in *Proceedings of the 2002 Conference on File and Storage Technologies*, 2002, pp. 89–101.

[40] C. A. Waldspurger, "Memory Resource Management in VMware ESX server," *SIGOPS Operating Systems Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[41] "Kernel SamePage Merging," http://www.linux-kvm.com/content/using-ksm-kernel-samepage-merging-kvm.

[42] "KVM: Kernel Based Virtual Machine." http://www.linux-kvm.org/.

[43] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Proceedings of 1993 USENIX Conference*, January 1993, pp. 519–529.

[44] 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*, January 2000, pp. 218–228.

[45] 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*, November 2000, pp. 150–159.

[46] R. Sendag, P.-F. Chuang, and D. J. Lilja, "Address Correlation: Exceeding the Limits of Locality," *IEEE Computer Architecture Letters*, vol. 2, no. 1, p. 3, May 2003.

[47] C. Molina, C. Aliagas, M. Garcia, A. Gonzalez, and J. Tubella, "Non Redundant Data Cache," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, August 2003, pp. 274–277.

[48] C. B. Morrey III and D. Grunwald, "Content-Based Block Caching," in *Proceedings of the 23rd IEEE Conference on Mass Storage Systems and Technologies*, May 2006.

[49] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004, pp. 212–223.

[50] E. G. Hallnor and S. K. Reinhardt, "A Compressed Memory Hierarchy using an Indirect Index Cache," in *Proceedings of the 3rd Workshop on Memory Performance Issues: in conjunction with the 31st International Symposium on Computer Architecture*, March 2004, pp. 9–15.

[51] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, "Multi-Execution: Multicore Caching for Data-Similar Executions," in *Proceedings of the 36th International Symposium on Computer Architecture*, June 2009, pp. 164–173.

[52] L. Villa, M. Zhang, and K. Asanović, "Dynamic Zero Compression for Cache Energy Reduction," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, December 2000, pp. 214–220.

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

[54] J. Dusser, T. Piquet, and A. Seznec, "Zero-Content Augmented Caches," in *Proceedings of the 23rd International Conference on Supercomputing*, June 2009, pp. 46–55.

[55] 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*, December 1997, pp. 194–203.

[56] L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Processors," in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, August 1999, pp. 206–211.

[57] 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*, June 2005, pp. 260–271.

[58] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter, "Compiler Techniques for Code Compaction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 378–415, March 2000.

[59] A. Beszedes, R. Ferenc, T. Gyimothy, A. Dolenc, and K. Karsisto, "Survey of Code-Size Reduction Methods," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, pp. 223–267, September 2003.

[60] S. Kim, J. Lee, J. Kim, and S. Hong, "Residue Cache: a Low-Energy Low-Area L2 Cache Architecture via Compression and Partial hits," in *Proceedings of the 44st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 420–429. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155670

[61] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000, pp. 234–244.

[62] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, November 2001, pp. 164–171.

[63] D. M. Tullsen, "Simulation And Modeling Of A Simultaneous Multithreading Processor," in *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996, pp. 819–828.

[64] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, April 2009.

[65] C. C. Compaq, "Alpha Architecture Handbook," October 1998.

[66] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 45–57.

[67] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Representative multiprogram workloads for multithreaded processor simulation," in *Proceedings of the 2007 IEEE International Symposium on Workload Characterization (IISWC)*, September 2007, pp. 193–203.

[68] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernández, and M. Valero, "Fame: Fairly measuring multithreaded architectures," in *Proceedings of the 2007 International Conference on Parallel Architectures and Compilation Techniques*, September 2007, pp. 305–316.

[69] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*, July 2001, pp. 40–56.

[70] B. Black, B. Rychlik, and J. P. Shen, "The Block-based Trace Cache," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 196–207.

[71] E. Rotenberg, S. Bennett, and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 111–120, February 1999.

[72] 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, pp. 333–344.

[73] M. Behar, A. Mendelson, and A. Kolodny, "Trace Cache Sampling Filter," in *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, September 2005, pp. 255–266.

[74] A. Malamy, R. N. Patel, and N. M. Hayes, "Methods and aparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature," United States Patent 5353425, October 1994.

[75] R. Joseph and M. Martonosi, "Run-time Power Estimation in High Performance Microprocessors," in *Proceedings of the 2001 international symposium on Low power electronics and design*, ser. ISLPED '01. New York, NY, USA: ACM, 2001, pp. 135–140. [Online]. Available: http://doi.acm.org/10.1145/383082.383119

[76] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[77] D. Koufaty and D. T. Marr, "Hyper-Threading Technology in the Netburst Microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[78] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid - Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, pp. 200–222, August 2001.

[79] A. W. Services, "Amazon elastic compute cloud: User guide," Tech. Rep. API Version 2009-11-30, 2010.

[80] ARM, "Cortex-A8 Technical Reference Manual," 2007.

[81] A. J. Smith, "Cache Memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, September 1982.

[82] D. Sager, D. P. Group, and I. Corp, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, 2001.

[83] J. Casazza, "First the tick, now the tock: Intel microarchitecture (nehalem)," *Intel Corporation*.

[84] F. Mohamood, M. Ghosh, and H.-H. S. Lee, "DLL-conscious Instruction Fetch Optimization for SMT Processors," *Journal of Systems Architecture*, vol. 54, pp. 1089–1100, 2008.

[85] D. H. Woo, M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee, "Reducing Energy of Virtual Cache Synonym Lookup using Bloom Filters," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 179–189. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176783

[86] X. Qiu, "The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1585–1599, December 2008.

[87] "Enabling Grids for E-sciencE," http://www.eu-egee.org/.

[88] C. Marco, C. Fabio, D. Alvise, C. Antonia, G. Francesco, M. Alessandro, M. Moreno, M. Salvatore, P. Fabrizio, P. Luca, and P. Francesco, "The glite workload management system," in *4th International Conference on Grid and Pervasive Computing*, 2009.

[89] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 234–244, 2000.

[90] B. H. Bloom, "Space/time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

Appendix A

CATCH Design Space Exploration

This appendix presents the design space exploration that we conducted to choose the best size and associativity for HDD and DR tables. The lines in Figure 61 show the IPC for each benchmark with respect to the left y-axis while the bars show the average IPC of all benchmarks for each configuration with respect of the right y-axis.

A.1 HDD Design Space Exploration

Figure 61.a presents the IPC of CATCH for different HDD associativities, and an unbounded DR when the HDD has only 512 entries. The results are for a 16KB UCC cache for valid blocks.

The figure indicates that we can achieve the maximum potential of the mechanism with 8 ways associativity for HDD, and the performance drops rapidly when reducing the associativity.

One interesting observation from Figure 61.a is that sometimes with a lower HDD associativity the performance of CATCH is higher for some benchmarks, for example Q8A. This happens due to the "failure" to maintain the hashed content for all blocks in the cache because of HDD replacements. This results in duplicated content to be inserted in the cache. The data show this duplication to be beneficial to performance because CATCH may "allow" different 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 a different mapping.

Figure 61.b shows the IPC using CATCH with an 8-way HDD for various number of entries and with unbounded and fully associative DR. The data indicate that a 128-entry usually provides performance close to a 512-entry HDD.

Based on these results we decided that the best configuration for HDD should be 128 entries with 8-way associativity.

A.2 DR Design Space Exploration

In the previous section we have concluded that 128-entry, 8-way, HDD provides performance close to CATCH limit. This section explores the effects of associativity and size on the performance of DR.

Using an 8-way set associative HDD with 128 entries, we have measured the performance of DR with 1024 entries and for different associativities. These results are shown in Figure 61.c. It is evident from the data that either a 4-way provides performance close to a fully associative DR.

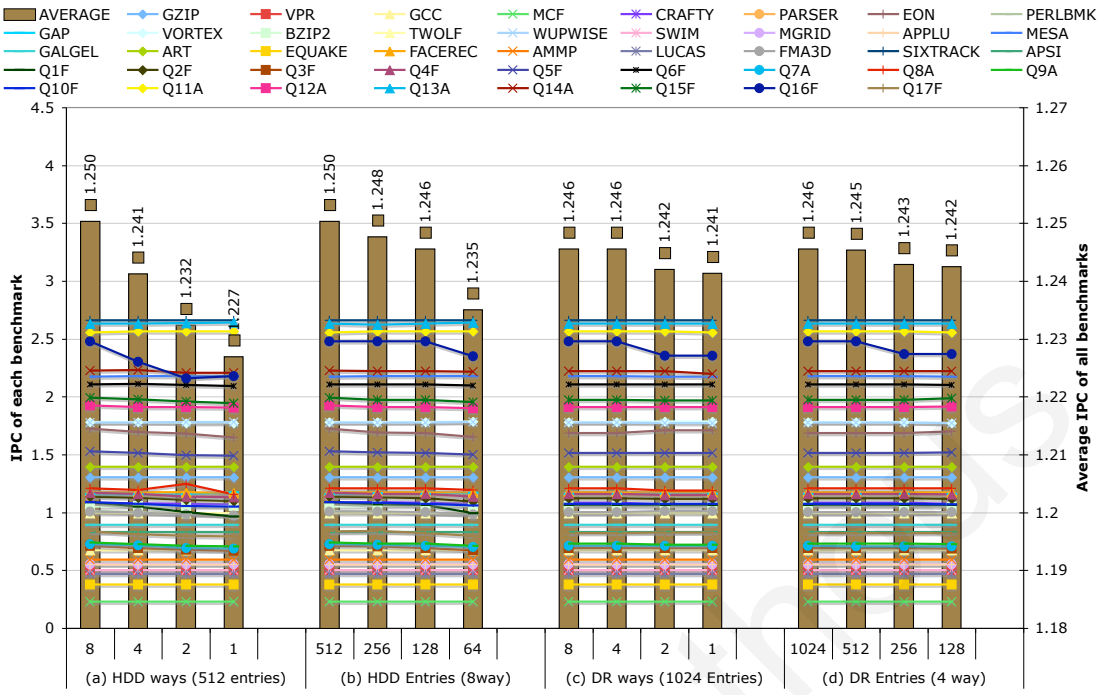

Figure 61: Performance potential of CATCH for a UCC using various sizes and associativity of HDD and DR for 16KB cache for valid blocks

Furthermore, Figure 61.d shows the results for different number of entries in a 4-way associative DR. It appears that we can reduce the required entries in DR down to 128 without loosing much performance.

Overall, the analysis suggests that a 4-way 128-entry DR and an 8-way 128-entry HDD represent a good performing configuration.

Appendix B

Synthetic Benchmark to Exercises Instruction Caches

```
void emptyFunc(){return;}
unsigned long long x = 0;

void oddN()     {
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        return;}
void evenN(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        return;}

void (*functionN[3])() = {&emptyFunc,&oddN,&evenN};

int execFlagN-1 = 1;
void oddN-1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlagN-1 = execFlagN-1 && !(depth == 2);
        int callFunc = gen_rand() & (execFlagN-1);
        functionN[execFlagN-1 + callFunc]();
        functionN[execFlagN-1 + ((callFunc^1) & execFlagN-1)]();
        execFlagN-1 ^= 1;
        return;}

void evenN-1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlagN-1 = execFlagN-1 && !(depth == 2);
        int callFunc = gen_rand() & (execFlagN-1);
        functionN[execFlagN-1 + callFunc]();
        functionN[execFlagN-1 + ((callFunc^1) & execFlagN-1)]();
        execFlagN-1 ^= 1;
        return;}

void (*functionN-1[3])() = {&emptyFunc,&oddN-1,&evenN-1};
.
.
.
void (*function2[3])() = {&emptyFunc,&odd2,&even2};

int execFlag1 = 1;
void odd1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
```

```
        execFlag1 = execFlag1 && !(depth == 1);
        int callFunc = gen_rand() & (execFlag1);
        function2[execFlag1 + callFunc]();
        function2[execFlag1 + ((callFunc^1) & execFlag1)]();
        execFlag1 ^= 1;
        return;}

void even1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlag1 = execFlag1 && !(depth == 1);
        int callFunc = gen_rand() & (execFlag1);
        function2[execFlag1 + callFunc]();
        function2[execFlag1 + ((callFunc^1) & execFlag1)]();
        execFlag1 ^= 1;
        return;}

void (*function1[3])() = {&emptyFunc,&odd1,&even1};

int main(int argc, char* argv[]){
        unsigned long long i = 0;
        unsigned long long k = atoi(argv[1]);
        depth = atoi(argv[2]);
        struct timeval t_start, t_fin;
        gettimeofday(&t_start,NULL);
        for (i = 0; i < k; i++){
        int callFunc = gen_rand() & (0x1);
        function1[callFunc+1]();
            function1[(callFunc ^ 1)+1]();}
        gettimeofday(&t_fin,NULL);
        timeval_subtract(&t_fin,&t_start);
        return 0;}
```

Appendix C

Acronyms

Acronym	Term
AB	Address Based
BCU	Block Compare Unit
BST	Block Size Table
BTB	Branch Target Buffer
CATCH	CCD Detection Hardware Mechanism
CB	Content Based
CCD	Cache Content Duplication
CDA	Content Duplication Aware
CE	Computing Element
CMP	Chip Multiprocessor
DAC	Duplicate Aware Cache
DL1	Data Level 1
DLL	Dynamic Link Library
DMC	Direct Mapped Cache
DR Table	Duplicate Relation Table
ETC	Extrinsic Text Cloning
FVC	Frequent Value Cache
HDD Table	Hashed Duplicate Detection Table
IL1	Instruction Level 1
IPC	Instruction Per Cycle
ITLB	Instruction Translation Lookaside Buffer
IRF	Instruction Register File
ISA	Instruction Set Architecture
ITC	Intrinsic Text Cloning
KSM	Kernel SamePage Merging

Acronym	Term
L1	Level 1
L2	Level 2
LLC	Last Level Cache
LRU	Least Recently Used
MRU	Most Recently Used
PC	Program Counter
PID	Process Identifier
PIPT	Physically Indexed Physically Tagged
OS	Operating System
RAS	Return Address Stack
SMT	Simultaneous Multithreading
TC	Text Cloning
TLB	Translation Lookaside Buffer
VIVT	Virtually Indexed Virtually Tagged
VIPT	Virtually Indexed Physically Tagged
UCC	Unique Content Cache
UI	User Interface
WMS	Workload Management System
WN	Worker Node
ZC	Zero Content