Haakon Dybdahl

Architectural Techniques to Improve Cache Utilization

Department of Computer and Information Science
Norwegian University of Science and Technology
N-7491 Trondheim, Norway

NTNU Trondheim
Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap
Doktor ingeniøravhandling 2007:77

ISBN 978-82-471-1707-1 (electronic)
ISBN 978-82-471-1691-3 (printed)

ISSN 1503-8181 (Doktoravhandlinger ved NTNU)

Abstract

The cache is a memory area where recently accessed data is stored for fast access. The size of the cache has grown rapidly during the last decades to compensate for the increasingly slower main memory. The state-of-the-art memory hierarchy has multiple levels of cache with the last-level being the largest and slowest. The area of the chip dedicated to the last-level cache is substantial, but the performance of the cache is quite low. Typically 50% of the data in the cache will not be accessed before the data is removed. This thesis takes three approaches to improve the utilization of the cache.

The first approach extends the cache functionality to include write-backs for *destructive-read* DRAM. Destructive-read DRAM is faster than conventional DRAM, but the content is lost on read operations. The contribution is two schemes for using the cache as a write-back buffer instead of using a dedicated write-back buffer.

In the second approach several different schemes are proposed that improve control of which data that are kept in the last-level cache. The first scheme bypasses memory accesses that are transient. A heuristic classifies accesses as transient or non-transient based on a run-time analysis of the instructions that load the data. This heuristic is more precise than earlier proposed schemes intended for first-level caches. In the second scheme the references to each cache block are counted. This information is used in an extension to the conventional *least-recently-used* (LRU) replacement policy so cache blocks that are accessed frequently are protected from eviction by cache block which are less frequently accessed. The scheme is fairly simple as it requires no additional information from the processor and only extends the cache block with an additional counter per block. The scheme is efficient in a chip multiprocessor (CMP) architecture with shared last-level cache since it can stop a single processor from filling the cache with unnecessary blocks. This protects needed cache blocks for the other processors.

In the third approach the last-level cache is dynamically partitioned in a CMP. This is evaluated for a conventional shared cache architecture and for a novel non-uniform cache architecture (NUCA) with concurrent private and shared last-level cache. Novel mechanisms for determining the best partition sizes for each processor are described and used in these schemes. These schemes outperform conventional partitioning by an LRU replacement policy which partitions the cache more arbitrary.

A mechanism is proposed for monitoring performance of an alternative cache scheme in order to activate it only when it improves performance. The tags for a limited number of sets in the cache are duplicated and the alternative scheme is run on these tags. This mechanism can stabilize schemes that are not robust and hence increase the overall performance.

The different schemes are simulated by an extended version of SimpleScalar. The SPEC2000 benchmark suite is used as workload. The performance gains of the new schemes are shown relative to conventional architectures, state-of-the-art techniques and compared to upper-bound non-implementable approaches.

Contents

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree *PhD*.

The work herein was performed at and funded by the Department of Computer and Information Science, NTNU, under the supervision of Professor Lasse Natvig.

The thesis consists of two parts. The first part is introduction, background and methodology for the work done as well as a summary of the papers and final conclusions.

The second part is the main contribution, presented as a collection of six research papers which are verbatim copies of the previously published versions.

Acknowledgments

There are many people to whom I am very grateful for their help and encouragement while undertaking the work described in this thesis.

- Firstly, I would like to thank my advisor Professor Lasse Natvig for support and guidance and for bringing me back to NTNU. I am especially grateful to Natvig for allowing me to pursue my own ideas to a large extent, but still keeping me on track.
- Thanks to Professor Per Stenström for sharing his broad knowledge of research in an exceptionally motivating and intelligible manner. When I am facing some difficulties I often question myself "How would Stenström approach this (research) problem?". The visit to Chalmers University of Technology autumn 2005 was extremely satisfying.
- Thanks to Professor Mads Nygård and Dr. Gaute Myklebust who have acted as my co-advisors. They were especially involved in the early phase of the work and helped to focus and adjust the research plan through honest and straightforward opinions.

- I would like to thank all the people at the Computer Architecture and Design Group at NTNU. Especially, I would like to thank Marius Grannæs and Magnus Jahre for marvelous computer architecture discussions that have lead to noble research ideas.
- Thanks to Dr. Per Gunnar Kjeldsberg for his positive attitude, knowledge and effort in our collaboration.

Last but not least, I would like to thank all my friends and colleagues and my family for love and support through three decades - and most important of all, thanks to Tine for your love and support and Mari for all the joy.

Haakon Dybdahl
March 27, 2007

Chapter 1

Introduction

> "Never be afraid to try something new. Remember, amateurs built the ark; professionals built the Titanic." — *Unknown*

This chapter introduces the thesis, with the main introduction given in Section 1.1. Section 1.2 presents the main research question that has been identified and explored. Section 1.3 gives an introduction to the methodology used in the research. A summery of the contributions is found in Section 1.4. The roles of the different co-authors for the included papers are described in Section 1.5. An outline of the thesis is given in Section 1.6.

1.1 Introduction

The *cache* [53, 114] system is crucial to feed the processors with enough data and instructions to process. The access latency of the main memory is several hundred times slower than the clock period of the processor [33]. The cache bridges this difference in latency between processor and main memory. This difference has been growing exponentially for many years and was predicted to become a bottleneck decades ago [32,117].

On high performance processors the cache is co-located with the processor core(s) on the same chip to minimize the cache access time. The cache fills up a significant area of the chip, often as much as 50% (for example see Power 4 [39]). The cache power usage is also considerable and is expected to increase with denser technologies due to increasing leakage currents [89] and increasing cache size. Typically 50% of all blocks in a cache are dead [100] which means these blocks will not be accessed again before they are evicted. By improving which data that are kept in cache, the performance of the cache can be higher without increasing cache size.

Even with so much of the processor chip area being used for cache, the processor cores are still suffering due to the slow memory system. Applications with low locality in the memory access patterns have low performance limited by the memory system. Other techniques are used to further compensate for slow memory accesses and low cache performance such as multi-level caches, instruction and data prefetching in both hardware and software, out-of-order execution, speculative loads, lock-up free caches (i.e. a cache miss not blocking subsequent accesses [61]), data prediction, run ahead execution, multi-threaded execution and memory-request reordering at the controller level (a description of these techniques can be found in the book by Hennessy and Patterson [33]). All these extra techniques increase chip area , increase design complexity and require a large amount of power. Today, design complexity and power usage are the most visible bottlenecks for further single-thread performance development. However, the underlying fundamental problem is slow memory access. The number of cores in a processor is growing in order to exploit more parallelism and hence improve performance.

Methods for improving the speed of the main memory reduce the negative effect of cache misses. Main memory is usually built with DRAM technology which is rather slow. Hwang et al. proposed a scheme for reducing the latency of the DRAM memory by not conserving the data content on read operations [38]. We call this *destructive-read DRAM* in this thesis. Dedicated write-back buffers were proposed to conserve the data content.

Improved cache performance can reduce the need for some of the other techniques used to compensate for the slow main memory. This can in turn decrease power and area requirements and lead to better computational performance. Techniques for improving cache performance should not increase the cache size as this will increase power consumption, latency and area requirements.

1.2 Research Question

The main research question identified and explored by this thesis is:

How can the cache be used more efficiently to increase system performance?

This thesis is an exploratory study that proposes and evaluates several different novel schemes to bridge the processor-memory performance gap by improving cache utilization. The focus is improving the last-level cache since this is the largest cache. The main research question is decomposed into the following sub-questions which are answered in the six papers. Background and motivation for the sub-questions are found in Chapter 2 and 3.

1. Can the proposed dedicated write-back buffers for destructive-read DRAM be eliminated by using the last-level cache as a write-back buffer? Will this cause

contention due to extra write-backs? Will the extra write-backs increase power consumption?

2. Can the locality in the last-level cache be improved by bypassing some of the cache blocks based on classifying the instructions that load the data?

3. Can the locality in the last-level cache be improved by augmenting the least recently used (LRU) replacement policy to include frequency of use?

4. Can the sharing of the last-level cache for chip multiprocessors (CMPs) be improved by dynamically controlling the partition sizes?

5. Can earlier proposed non-uniform cache architectures (NUCAs) be improved by controlling the partition sizes?

6. How can alternative cache schemes efficiently be enabled when they perform well and disabled when they do not improve performance?

This thesis does not consider alternating or profiling the program code to improve cache utilization. All schemes execute in run-time. In this way the schemes can be used with existing software and compilers.

1.3 Methodological Approach

There are different methods for analyzing behavior of computers which represent different abstraction levels. The most common methods are mathematical models, simulation at different abstraction levels and implementation of prototypes in hardware. Often, when new architectures are being studied, a bottom up approach is used. That is, logical and mathematical models are used to understand basic concepts. Example of this is cache miss rates and classification of cache misses. Simulation is done for exploration and confirmation of performance. Prototypes are closer to "hard evidence", allow larger benchmarks to be run and involve routing and layout. They are however much more expensive and complex to develop.

The new schemes presented in this thesis are extensions to conventional architectures and schemes. In the papers, the algorithms and logic of the newly proposed schemes are first presented. They are simulated and compared against the performance of conventional schemes, other state-of-the-art schemes and to theoretical non-implementable schemes. Only relative numbers based on comparison to other schemes are presented. More technical details about methodology are given in Section 4.1.

1.4 Contributions

The main contributions of the work are the different schemes and the ideas presented and evaluated in the papers:

1. *Using the cache as a write-back buffer for destructive-read DRAM.* While previous attempts to use a destructive-read DRAM macro have used a dedicated write-back buffer to conserve memory content, we use the cache of the computer system to perform this write-back task. A novel memory architecture is presented and evaluated in terms of computation performance. Two different strategies for writing data back to DRAM are considered. The power consumption of using the cache as a write-back buffer with destructive-read DRAM is evaluated.

2. Improving utilization of cache memory by bypassing memory accesses that are transient. While previous work has studied bypassing the first-level cache, this is the first work to consider *block bypassing in the last-level cache*. The classification is based on a run-time analysis of the instructions that load the data. A novel heuristic is presented which is based on a feedback-loop. It is compared to earlier bypassing schemes. This new approach is more stable and hence more usable. The heuristic is not only used for block bypassing, it is also used for *early miss determination*. Early miss determination implies that accesses that are predicted to miss in the last-level cache can speculatively be sent earlier to main memory, i.e. without waiting for miss confirmation from the last-level cache. This reduces latency for accesses that miss in the last-level cache.

3. Augmenting the *cache LRU replacement policy with frequency of use*. The scheme is evaluated for a CMP with shared cache where one processor can evict cache blocks belonging to other processors. The cache blocks that are frequently accessed receive improved protection from eviction compared to cache blocks that are less frequently accessed. This extension to the replacement policy stops applications that install a lot of unneeded cache blocks into cache from evicting needed cache blocks for other processors.

4. A mechanism called *switching* which turns on and off schemes depending on their performance. The tags for a small number of sets in the cache are duplicated and the alternative scheme is run on these tags. This mechanism can stabilize schemes that are not robust and hence increase overall performance. This switching is based on a novel technique which monitors the performance of the alternative scheme by running it on what is coined as *shadow tags*. This switching is applied to existing schemes for improving last-level cache performance and improves the stability of these schemes.

5. *Dynamically adjusting the last-level cache partition sizes for CMPs*. A novel mechanism to determine the gain of increasing the cache size per processor is described. The idea of *shadow tags* is used to monitor the number of cache misses

that would have been avoided if the cache size was increased. This, in addition to existing techniques for monitoring the number of cache misses increased by reducing the cache size, enable a novel mechanism for improving partitioning of the shared last-level cache in CMPs. This scheme is presented and evaluated in such a context. The cache partition sizes are tuned for maximum total performance for all processors, i.e. the total number of cache misses is minimized.

6. A non-uniform cache architecture (NUCA) with *concurrent reconfigurable private and shared last-level caches*. The private cache is faster than the shared cache, and by dynamically controlling the size of the partitions, a more optimal configuration is chosen compared to earlier proposed schemes. The size of the shared cache is balanced against the different private cache sizes. The new novel architecture is presented and evaluated for a CMP configuration.

Another contribution is the initialization of the research on cache systems and CMPs in the computer architecture group under supervision of Prof. Natvig. This includes using SimpleScalar as a tool and extending this to evaluate new ideas with SPEC2000 benchmarks. In collaboration with Marius Grannæs a framework for running experiments on clusters of workstations was made. The next PhD students will continue down the same road with related research problems even though the simulator might be phased out.

Most of the work in this thesis was conducted in a very successful collaboration with Professor Per Stenström at Chalmers University of Technology.

1.5 Roles of the Co-authors

I was the principal author of the papers, but the papers could not have been written without the help from the co-authors. During the research and writing of each paper, regular meetings were performed with the co-authors. The idea, focus of the research, contribution of the research, the experiments, evaluation of the experiments, general methodology, related work, results, conclusions, presentation and reviews have been typical topics for discussion.

More specific information about the different roles of the authors for each paper:

- *Paper I: Cache Write-Back Schemes for Embedded Destructive-Read DRAM.* The initial idea came up while studying existing DRAM macros and their latency, density and technology. Grannæs and Natvig had the role as advisers during the whole process and helped on planning, presentation and quality assurance.
- *Paper II: Destructive-Read in Embedded DRAM, Impact on Power Consumption.* Grannæs, Natvig and Kjeldsberg had the role as advisors during the whole process. Kjeldsberg was involved with the very details of the power models of the processor and memory that were simulated. Grannæs made some of the general scripts used for analyzing simulation data.

- *Paper III: Enhancing Lower Level Cache Performance by Early Miss Determination and Block Bypassing.* Stenström came up with the initial idea. I developed the idea further under his supervision. Stenström contributed a lot to the presentation in the paper, especially the introduction.
- *Paper IV: An LRU based Replacement Algorithm Augmented with Frequency of Access in Shared Chip Multiprocessor Caches.* The ideas came up as a result of trying to simplify the scheme from the previous paper and at the same time making a more adaptive scheme which was Stenström's idea. Stenström and Natvig helped on focusing the work and with the presentation and acted as advisors. Some of the code was written in co-operation with Grannæs as a general framework for CMP experiments.
- *Paper V: A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors.* The idea of this work came when writing the previous paper. Stenström and Natvig helped focus the work, highlight the contributions and the presentation of the work and acted as advisors.
- *Paper VI: An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors.* The idea of this work came partly from a discussion with Magnus Jahre and partly from writing the previous paper. Stenström helped on highlighting the contributions, focusing the research and improving the presentation of the work and acted as an advisor while the work was conducted.

1.6 Thesis Outline

This thesis is a *collection of papers*. The outline is as follows:

Part I - Context

Chapter 1 gives an introduction to the thesis.

Chapter 2 motivates for improving the memory system.

Chapter 3 relates the new presented schemes to existing schemes.

Chapter 4 gives an introduction to the methodology used.

Chapter 5 summarizes the papers.

Chapter 6 concludes the thesis, suggests possible avenues for future work and gives a short outlook on the challenges facing the research field.

References contains references for the first part of the thesis.

Index is an index for the first part of the thesis.

Part II - Papers

This part presents the main research contribution of this thesis. A collection of six research papers is given, with the papers being verbatim copies of the originally published versions.

Chapter 2

Background

> "Life can only be understood backwards, but it must be lived forwards." — *Soren Kierkegaard*

2.1 Introduction

This chapter explains why improving cache performance has become an important research area during the last few years:

- The performance per watt has become more important due to restrictions on power consumption. Smaller caches require less power than larger caches.
- Cache misses are becoming more expensive because the speed of the processor is increasing much faster than the speed of the main memory.
- With the introduction of chip multiprocessors (CMPs), transistors can be used for new cores to improve performance instead of increasing the cache size or adding more functional units. Smaller caches allow more cores.
- Increasing cost of communication with denser technologies. A larger cache is slower than a smaller cache. This difference in speed becomes larger with denser technologies.

The bottom line is that techniques that improve cache performance without increasing cache size will become more important with denser future technologies.

It is increasingly difficult to extract more parallelism from a single thread which has caused the development to slow down for single-threaded performance. Increasing the number of cores to increase parallelism is the motivation behind the industry moving

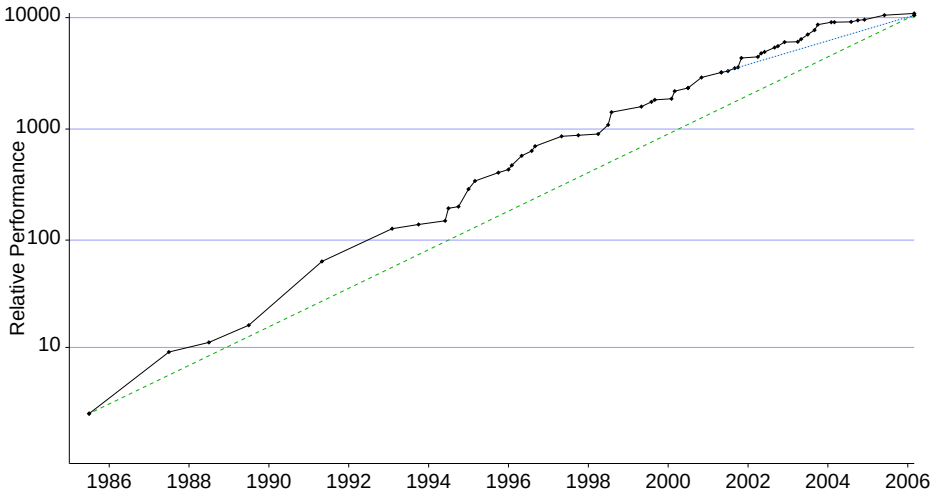


Figure 2.1: The development of single thread performance [111]. ©Fredrik Warg, 2006. Republished with permission.

to chip-multiprocessors (CMPs) architectures. This raises new questions such as what architecture is most suitable for the last-level cache.

Section 2.2 explains the term *the process-memory performance gap* and why cache performance is increasingly more critical for sustained performance growth. Section 2.3 shows that performance per watt is increasingly more important with future processors due to power limitations. Section 2.4 discusses how shrinking technology (feature size) increases latency, but improves transistor switch time. CMPs are discussed in Section 2.5.

2.2 The Processor-Memory Performance Gap

The growth in performance for high-performance processor has followed almost an exponential development during the last decades as shown in Figure 2.1 with a growth rate of 40-60% per year. However, the speed of the main memory has followed a much lower growth rate of about 7% per year as shown in the Figure 2.2. This has lead to what is referred to as the processor-memory performance gap which has been growing with about 40-50% each year since 1980. One driving force behind the exponential growth of processor performance is what is often referred to as *Moore's law* [73]. The most popular formulation of Moore's law is that the number of transistor on a chip is doubled every 18 months. This is due to the progress in manufacturing chips with denser technologies.

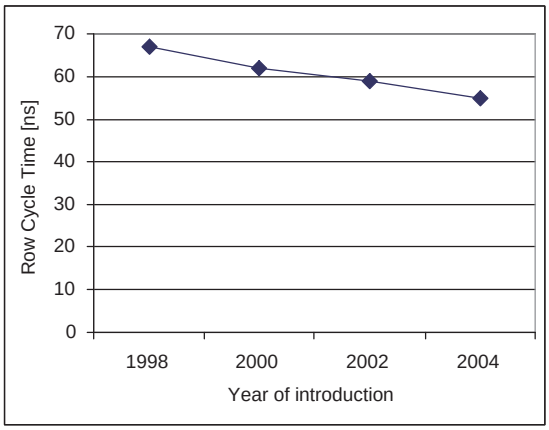


Figure 2.2: DRAM row cycle time scaling trend for commodity DRAM [108]

The growth of the number of transistors per chip is shown in Figure 2.3. Even though it has not always followed the 18 months prognosis, the growth has been quite steady between 18 and 24 months. A different popular interpretation of Moore's law is that the performance of computers is doubled every 18 month. However, this performance is not only related to the increasingly number of transistors that fit into a single chip, it is also highly related to the increased clock frequency of chips with denser technologies. The increase in frequencies is shown in Figure 2.4. Increasing clock frequencies for processors result in higher performance[1]. Increased number of transistors per chip has enabled processor architectures with higher parallelism. However frequencies have stopped following an exponential growth due to power (see Section 2.3) and latency issues (see Section 2.4) and therefore the growth rate of single thread performance is starting to slow down (see Figure 2.1). Another factor is that it has become harder to exploit more parallelism in sequential programs. Even though theoretical hundreds of instructions can be executed in parallel for some applications ([33]:page 240-259), it is increasingly hard to exploit more than four instructions in parallel. Some of the limiting factors for instruction-level-parallelism (ILP) are imperfect branch predictors, caches and data dependencies. This has lead to the development of parallel processors or CMPs (see Section 2.5).

[1] If there are bottlenecks outside the processor such as hard-disk or if the application is bound by main memory it is not true.

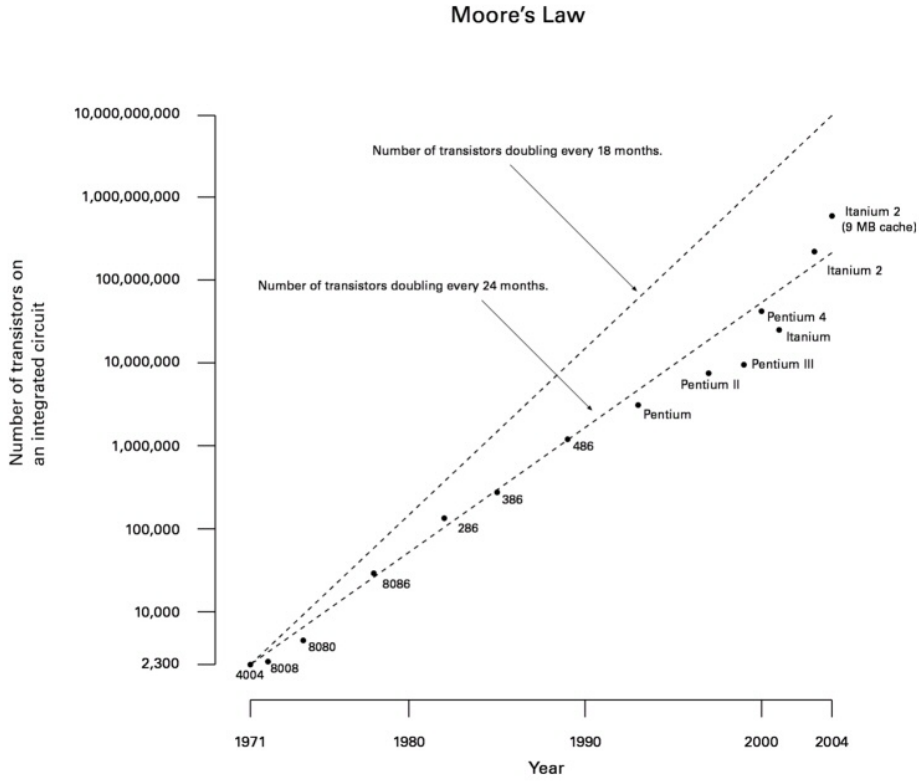


Figure 2.3: The number of transistors in microprocessors [113]. The figure was released as public domain by its author.

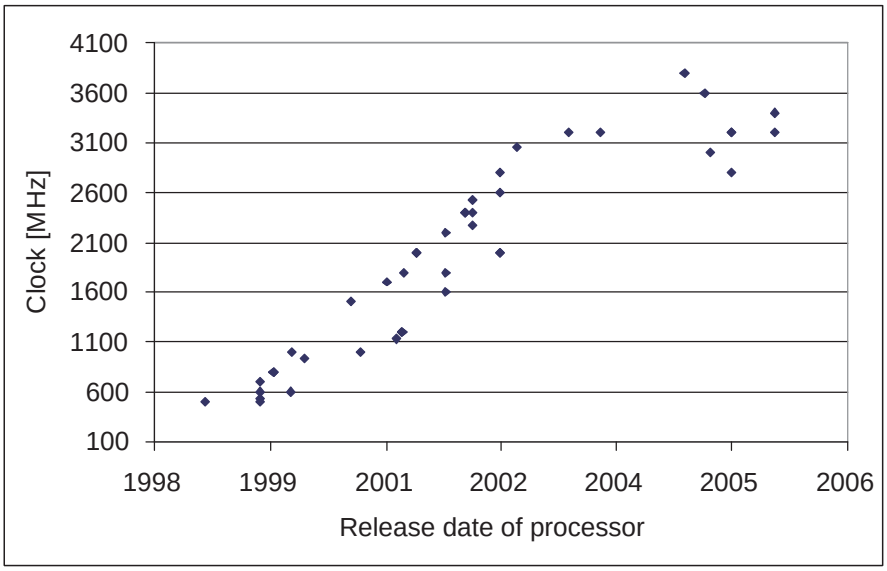


Figure 2.4: Frequency trend of microprocessors. The graph is based on numbers from A. Batto [10].

2.3 Power Restrictions

The techniques for compensating for the slow main memory increase design complexity and number of transistors required. The cache sizes have grown exponentially as shown in Figure 2.5. This growth is not necessarily because the problems the processors are solving are growing exponentially[2], but because of the growing problem with the processor-memory performance gap. A larger cache will in most cases reduce the number of cache misses. Around year 2003 the power consumption of high performance processors reached practical limits. The high power consumption was caused by denser technologies that enabled more transistors that switched faster. The dynamic power consumption is proportional to the number of transistors and the clock frequency[3]. Additionally, with denser technologies (denser than 130 nm) leakage current (static power) has become significant. However, with denser technologies the voltage can be turned down slightly and this reduces both the static and dynamic power consumption, but does not compensate fully for effect of the increase in frequency and number of transistors. The power consumption trends for x86 processors are shown in Figure 2.6. At one point the power consumption became a major bottleneck for performance (about 130 watt);

[2]In some cases this might not be the case such as scientific computing where problems are scaled by computing power.

[3]This is a simplification because it depends on the share of transistors that are switching, the square of the supply voltage and a lot of other factors.

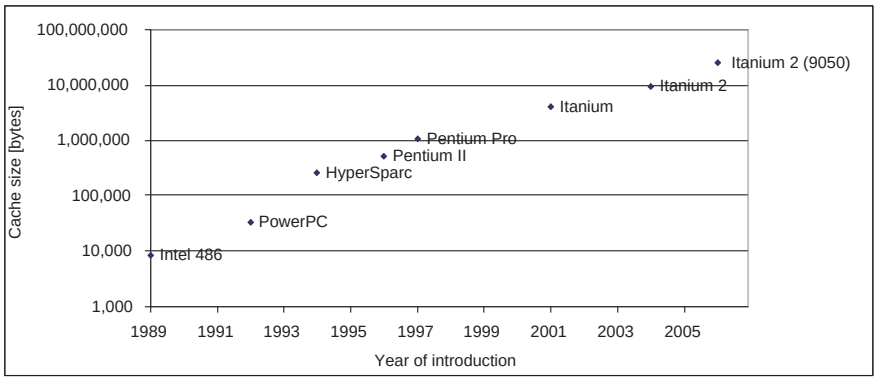


Figure 2.5: Trend for largest last-level cache as function of year of introduction. The graph is based on numbers from Ken Polsson's web page about microprocessors [85] and information found at Intel.com.

The power that can be supplied to a chip has not grown with the same rate as the power consumption has increased [89]. High power consumption has several disadvantages such as high cost for electricity, cooling, chip packaging and heavier and more expensive batteries in mobile applications. This means that each transistor used has to be justified since it might increases the power budget, i.e. the power vs. performance becomes more important. Large caches have high leakage currents, and if caches can be made smaller with equal performance (i.e. with fewer transistors) power can be saved and used for other purposes. Increasing the utilization of caches is the main subject for this thesis.

2.4 Communication Latency and Technology

The performance of transistors and wires in chips are influenced by technology. A general trend is that transistors perform better with denser technologies while wires perform worse [33]. Transistors switch faster because they become smaller. Signal propagation becomes slower due to increased resistance and capacity per unit length of wire. The consequence for modern microprocessors is that a signal uses several clock cycles to travel from one edge of the die to the opposite edge.

With increased cost of communication, cache accesses in large caches will become slower. Worst case cache access time is for data with the longest distance to the requesting processor while data that are closest to the processor will have the lowest access time. Conventional caches have only one access time which is the worst-case access time. Cache architectures that have different access time for different cache blocks within the same level of cache are referred to as *non-uniform cache architecture* (NUCA). Smaller

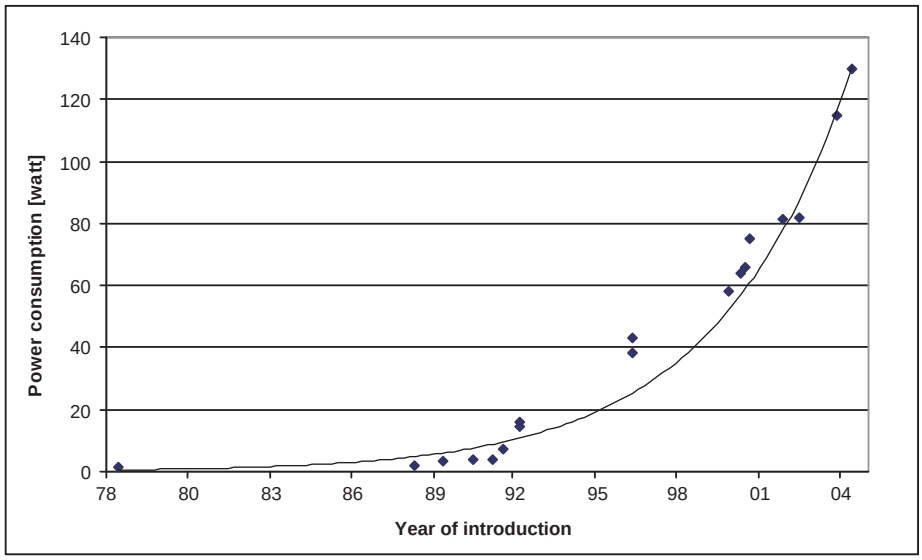
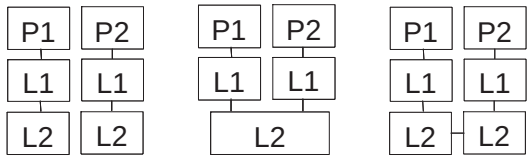


Figure 2.6: Trends for maximum power consumption [watt] for x86 processors as function of the year of introduction. The trend line is exponential. The graph is based on numbers from Batto [10].

caches are faster than larger caches, and even faster with denser technology. This is true for both NUCA and conventional cache architectures. Improving the cache performance (e.g. reducing miss rate) without increasing the area requirements (i.e. number of transistors) will give more payback with denser technology because the smaller cache will be increasingly faster than the larger cache.

2.5 Chip Multiprocessor

CMP is the new de-facto standard for high-performance computing. The number of processor cores in the chip is increased in order to exploit more thread-level parallelism and the frequency is turned down to decrease power consumption. A lower frequency not only saves power, but also reduces the processor-memory performance gap and hence balances the architecture to some extent. This architecture is called *chip multiprocessor* (CMP) architecture in this thesis. Several of the papers in this thesis propose techniques for CMP architectures. The last-level cache in a CMP can be private, shared or a hybrid as shown in Figure 2.7. The private cache can be faster than a shared cache and its content is not altered by other processors. A shared cache can utilize the cache space better since an application that works on a large data set will use a larger amount of the cache space when run in parallel with a an application that works on a small data set. A

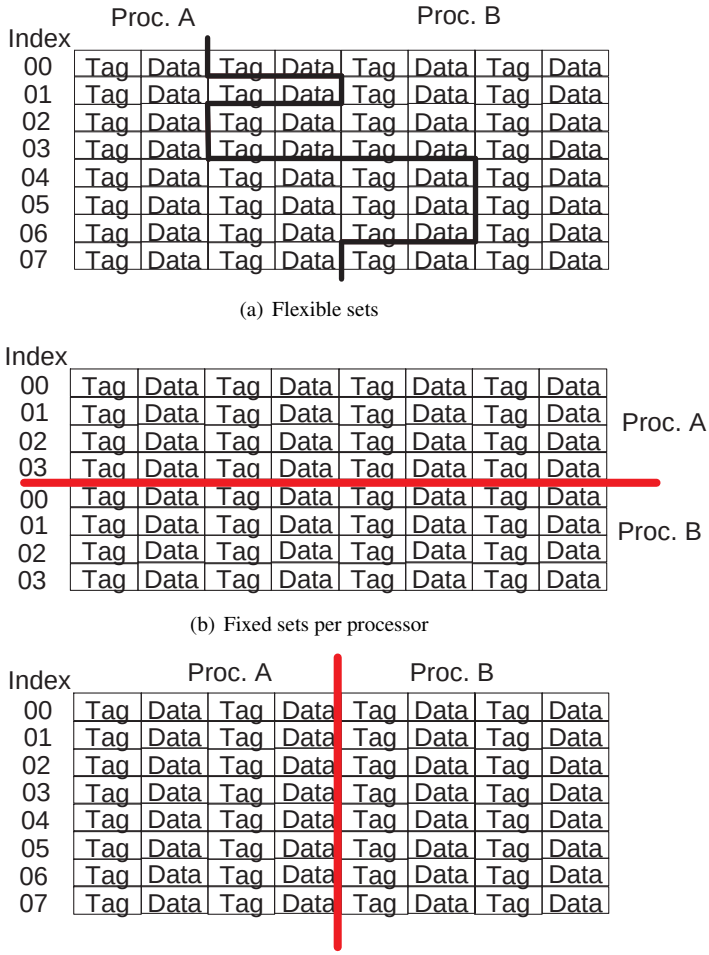


(a) Private (b) Shared (c) Hybrid

Figure 2.7: Different last-level cache organization for CMPs. L1 is first-level cache, L2 is the second and last-level cache and Px is processor cores.

hybrid cache architecture combines the advantages of private and shared caches. Each processor has a local fast cache, but can also use the cache space of the other processors. Mechanisms for this are described in Section 3.6.3.

A private last-level cache can be partitioned so each processor has a fixed cache space as shown in Figure 2.8(b). Changing the partition size is complex because the hashing function that maps memory addresses to sets has to be modified and the cache blocks have to be relocated or invalidated. The cache can be partitioned with a fixed number of cache blocks per set as shown in Figure 2.8(c). It is simpler to repartition the cache since there is no change to which set an address maps to. In state-of-the-art CMPs with shared cache, the LRU-replacement policy partitions the cache as shown in Figure 2.8(a). Each set can contain different number of cache blocks from different processors. Paper V proposes an improved replacement policy that partitions the cache more optimal. Processors that can utilize more cache space get more space and processors that fill the cache with unneeded cache blocks get less cache space.



(a) Flexible sets

(b) Fixed sets per processor

(c) Fixed partitioned sets

Figure 2.8: Three different ways of partitioning a cache. In Figure a) each set is shared while in Figure b) each set belongs to only one processor. In Figure c) each set is shared equally between the two processors. The effective associativity for each processor is different between b) and c).

Chapter 3

Improving Memory Latency

> "Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible." *— A. W. Burks, H. H. Goldstine, and J. von Neumann, 1946 [15]*

This chapter gives a brief introduction to the components of the memory access latency for the memory system of a computer. The proposed new schemes in the papers for improving cache performance are introduced and related to existing techniques.

Section 3.1 defines the different components of the memory access latency and presents an equation for the total latency. Section 3.2 relates the included papers in this thesis to the different memory components. Sections 3.3, 3.4, 3.5 and 3.6 consider reducing the miss rate, reducing off-chip bandwidth requirements, reducing miss latency and hit latency for the last-level cache, respectively. Finally, Section 3.7 considers main memory latency.

3.1 Memory Latency Components

The memory system with the involved latencies for a two core CMP with a multi-level [6, 109] cache system is shown in Figure 3.1. In this example the last-level cache is the level-two cache.

For cache hits in the first-level cache, the latency is the communication time for round-trip to the cache and the hit time for the cache. If there is a hit in the last-level cache, the latency is the (round-trip) communication time (both between processor and first-level

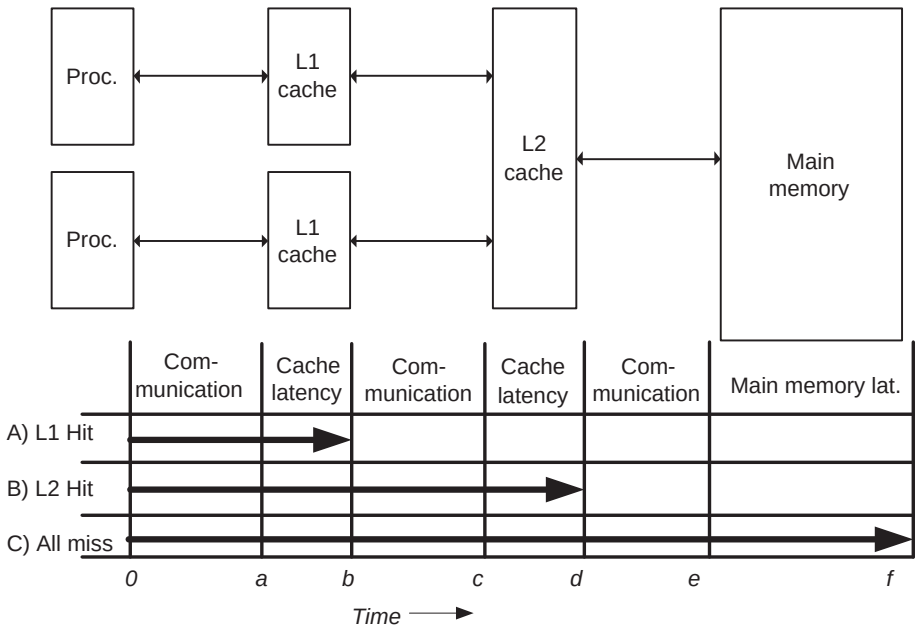


Figure 3.1: Logical sketch of a memory system and the latencies for accesses that hit in the first-level cache (L1), hit in the last-level cache (L2) and miss in both L1 and L2.

cache, and between first-level and last-level cache), miss time for the first-level cache and hit time for the last-level cache. For misses in all levels of cache, latency includes miss times for all caches, (round-trip) communication time (which also include off-chip communication) and latency of main memory.

The first-level cache typically has a latency of 2-3 clock cycles and a capacity of about 16-64 kBytes for Intel and AMD processors [8]. On-chip last-level caches have a magnitude larger storage capacity and latency. Latency is typically in the range of 8-30 clock cycles and capacity is up to tens of megabytes with current high-end processors. The increased latency of the larger memory blocks is caused by the distance to the memory block due to its larger size and the look up time in the larger memory block.

In order to make an expression for the average latency of the memory system some parameters have to be introduced:

- $miss\_lat_x$ is the time used to determine that a cache block is not in cache x.
- hit_lat_x is the time used to retrieve the cache block from cache x.
- $com\_lat_{x,y}$ is the round-trip communication latency from x to y. This parameter also covers the transmission delay of a cache block.
- $miss\_rate_x$ is the share of the accesses that result in a cache miss for cache x.
- hit_rate_x is $1 - miss\_rate_x$.
- $average\_access\_latency_x$ is the average memory access latency time for unit x.

The average latency (L) for a memory request is:

$$\begin{aligned} L = \quad & com\_lat_{processor,L1} + hit\_rate_{L1} * hit\_lat_{L1} + miss\_rate_{L1} * \\ & (miss\_lat_{L1} + com\_lat_{L1,L2} + hit\_rate_{L2} * hit\_lat_{L2} + miss\_rate_{L2} * \\ & (miss\_lat_{L2} + com\_lat_{L2,main\_memory} + \\ & average\_access\_latency_{main\_memory})) \end{aligned} \tag{3.1}$$

3.2 The Focus of Each Paper

The purpose of the work in this thesis is to reduce L without increasing power and area requirements significantly.

There are two ways to do this, either reduce average hit time or reduce miss penalty for each unit. Miss penalty is the product of the number of memory accesses that causes misses and the miss latency. This thesis uses several approaches to reduce the average memory latency (L). Figure 3.2 shows which fields the different papers investigate graphically. More specific, the different papers look at improving L by affecting the following parameters:

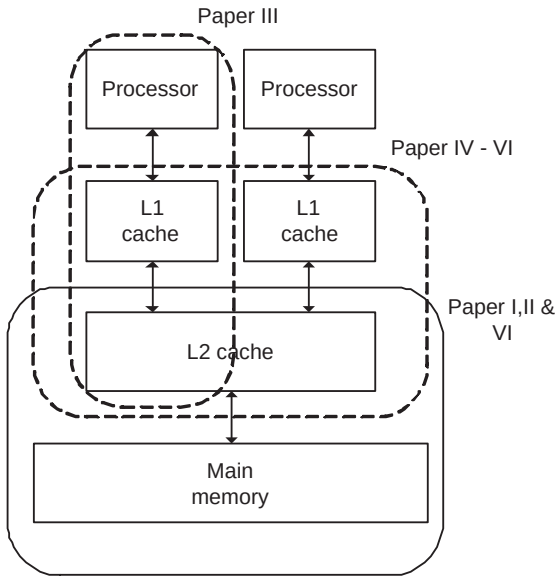


Figure 3.2: The research focus for the different papers.

1. Reduce the $miss\_rate$ for the last-level cache. Paper III, IV, V and VI.
2. Reduce the com_lat between the last-level on-chip cache and to the off-chip memory. Paper IV.
3. Reduce the $miss\_latency$ for the last-level cache. Paper III, V and VI.
4. Reduce the $hit\_latency$ for the last-level cache. Paper V and VI.
5. Reduce the $average\_access\_lat$ for main memory. Paper I and II.

The next sections introduce the techniques presented in the papers and look at relevant existing techniques. The categorization of techniques is inspired by the book written by Hennessy and Patterson [33]. However, while they look at all different cache techniques, this thesis is focused on techniques for the last-level cache. Description of general techniques mentioned later in this section without references can be found in the same book.

3.3 Reducing the Miss Rate for Last-level Cache

The cache is based on the assumption that the processor often requires the same data (same addresses) repeatedly or that the processor requests data that is close to the previous accesses in the address space. This was coined *temporal locality* and *spatial locality*

respectively by A. J. Smith [94].

Cache misses can be categorized based on their cause [33] [34]:

- Capacity misses. The cache can only contain a limited amount of data. If the program has a too large memory footprint, data is evicted before they are reaccessed.
- Conflict misses. Too many memory accesses are competing for the same set in the cache and end evicting each others cache blocks.
- Coherence misses. In a system with multiple cores, one processor can invalidate cache blocks in private caches for other processors through the coherence protocol and hence later cause cache misses.
- Cold misses. The first time the cache block is required it has to be loaded into the cache and hence causes a miss. This is also referred to as *compulsory misses* or *cold start*.

3.3.1 Reducing the Number of Capacity Misses

Increasing the size of the last-level cache improves the performance due to fewer capacity misses in most cases but is expensive in terms of chip area and power requirements. A larger cache is also slower.

Static memory (SRAM) and *dynamic memory* (DRAM) are two different memory technologies widely used in computers. SRAM can be made very fast, but requires four or six transistors to store one bit of data. DRAM is slower, but requires only one transistor and a capacitor to store one bit. The density of DRAM is thus much higher and modules are available with gigabits of data. These modules are inexpensive and are therefore used as main memory in computers. SRAM is the technology used for most caches.

The HP PA-8800 RISC processor [42] has increased the last-level cache size by only storing the tags on the processor chip and storing the data in a separate tightly connected DRAM chip. Since DRAM can store data more densely than SRAM, the chip area requirements are not increased, but the cache latency is increased. This is because DRAM is in general slower than SRAM due to technology reasons.

As the technology for merging DRAM and logic on the same chip has been improved, on-chip caches can be made of DRAM which is denser than SRAM. This increases the cache size without increasing area requirements as used in IBM's Blue Gene/L [79]. The DRAM will however not be as fast as SRAM.

A different way to increase the amount of data that can be stored in a multiple-level cache system is to use an *exclusive cache hierarchy* as described by Jouppi and Wilton [46]. This means that data in the higher-level caches are not stored in the lower-level cache, and hence the size of the cache is the sum of the cache sizes for all levels of cache. For

CMPs with shared last-level cache, the difference in capacity between an exclusive and inclusive cache hierarchy can be significant since there are many first-level caches. For an inclusive cache hierarchy, the storage capacity is only the size of the last-level cache as the other levels of cache only have redundant information.

Compressing the data stored in the cache can increase the effective size of the cache [3,65,118,122], but this increases latency as cache blocks have to be decompressed when needed. Compression and decompression also requires additional logic and increases complexity. The performance will depend on how compressible the cache blocks are and hence it will be less predictable than a conventional cache.

Chishti et al. proposed a scheme for increasing the usage of the cache space for CMPs with "private" caches. Cache blocks evicted for one processor can be stored in the cache storage for other processors if free space is available [23].

In Paper V we try to improve the utilization of the cache space by partitioning more optimally than the arbitrarily partitioning performed by the LRU replacement policy found in CMP with a shared last-level cache. We predict which processors that can best utilize the cache space and partition the last-level cache for maximum utilization. Suh et al. described a way of partitioning a cache for multi-threaded systems by estimating the best partition sizes [102, 103]. They counted the hits in the LRU position of the cache to predict the number of extra misses that would occur if the cache size was decreased. A heuristic used this number combined with the number of hits in the second LRU position to estimate the number of cache misses that are avoided if the cache size is increased. In paper V we extended this scheme with shadow tags and counters for estimating the number of hits that would be avoided by increasing the cache size. The method presented in Paper V increases the precision of the predictions and hence the performance is more robust and higher. Evaluation of cache partitioning in shared CMP caches has also been studied earlier by Kim, Chandra and Solihin [55] for a two-core CMP where a trial and error heuristic was used. Trial and fail as a partitioning method does not scale well with increasing number of cores since the solution space grows fast.

3.3.2 Reducing the Number of Conflict Misses

Associativity can be increased in order to reduce the number of conflict misses. However, the associativity is already quite high in modern last-level caches (e.g. 16-way in Sun's Niagara), and increasing it further will have diminishing results on reducing the number of misses while increasing complexity, latency, chip area and power consumption.

Using victim caches is one inexpensive way to reduce the number of conflict misses without increasing the associativity [45]. The victim cache holds recently evicted lines, typically less than the last six evicted and is fully associative. This improves performance of a direct mapped first-level cache, but is less suitable for last-level caches since

these caches are already highly associative and the number of sets is high. Bypassing has also been used to reduce conflict misses by using a bypass buffer in parallel with a direct mapped cache [41, 43, 71]. However, direct mapped caches are not commonly used as last-level caches in state-of-the-art high performance microprocessors and this reduces the usefulness for these techniques in this context. A similar approach is to split the cache into two independent caches, one larger cache with cache blocks that are predicted to have temporal locality and one smaller cache for cache blocks with unknown or predicted low temporal locality. In this way cache blocks with long reuse distance do not displace cache blocks with shorter reuse distance. Rivers and Davidson show that a smaller streaming (i.e. a FIFO) cache can be used to decide when data has short enough reuse distance to be inserted into a larger cache [87]. John and Subramanian use an annex cache [41], where blocks are moved from the annex cache into the main cache after several hits.

3.3.3 Reducing the Number of Coherence Misses

Shared memory multiprocessors use caches to reduce latency and reduce the number of accesses to main memory to avoid memory contention and communication contention. Memory contention occurs when the memory system can not handle any more requests in parallel due to limited number of DRAM chips and other bottlenecks. Communication contention is caused by the limited bandwidth in and out of the processor chip. However, when caches are used on shared memory a cache coherence scheme must be used to ensure sequential consistency (i.e. that all processors have the same view of the shared memory). There are a wide range of cache coherence schemes (see for example Stenström's survey [99] or Dubois et al. work [28]). Coherence misses are caused by parallel programs that share and modify the same data structures and use a write-invalidate protocol. If one processor modifies a cache block that is also present in some other private cache, the data is invalidated in the other processor(s) cache. There are different ways to reduce the number of misses caused by this invalidation such as changing the hardware scheme to update the data rather than invalidate it. The programmer or compiler can also reduce the number of misses by improving how the processors share the data.

Recently there is a significant interest on NUCA schemes for CMPs which address coherence and sharing of data [11, 19, 23, 54, 121]. Important aspects of these works are cache coherence protocols and duplication of shared cache blocks to speed up access latencies.

The coherence problem relates mostly to private caches, usually the first-level or the two first-levels of cache. This thesis focuses on the last-level cache and does not consider coherence problems.

3.3.4 Reducing the Number of Cold Misses

The number of cold misses can be reduced by prefetching data. Prefetching can be done by software or hardware and reduces the number of cold misses. Vanderwiel and Lilja made a survey about prefetching techniques [106].

The simplest prefetching method is sequential [45, 81, 91, 93], i.e. when requesting a cache block, the immediate following cache block is also fetched. More advanced prefetching methods store information about memory access pattern in dedicated hardware tables [7, 35, 44, 48, 56, 63, 96] in order to prefetch more complex access patterns.

Prefetching can be software controlled. In this case the program issues a special instruction which causes prefetching [17]. These instructions can be made by the programmer or by the compiler and must be non-blocking (i.e. the program should continue running even if the retrieval of the data has not finished).

A more advanced software approach is to use a helper thread that run a special program for prefetching data for the main program [27]. However, the helper thread requires resources and can degrade performance due to resource conflicts. Warg has looked into the major problems with thread-level speculation for run-time parallelization of programs [111].

New techniques for prefetching cache blocks could have been part of this work. However, a different PhD student at our group focuses on prefetching.

3.3.5 Exploiting More Locality

The programmer or compiler can improve the miss rate by accessing data in a pattern so data are kept in cache and hence fewer cache misses are created. Exchanging *for loops* in matrices is one examples of such an optimization. Blocking is another optimization technique where the program is operating on submatrices or blocks instead of entire row or columns so the working data fits into the cache [64]. I do not consider software or compiler techniques in this thesis.

The replacement policy decides which cache blocks that are candidates for eviction, and can in most cases be improved [101]. The most widely used policy for processor cache is the least recently used (LRU) replacement policy. Wang et al. looked at compiler techniques for improving the replacement policy, and included a limited study with prefetching [110]. In general there are several problems with a compiler approach. (1) The program has to be written for a specific hardware platform which makes the code less portable. (2) In CMP configurations the compiler will not have the information about the other applications that are sharing the same cache in contrast to a run-time implementation.

A mechanism for protecting cache blocks within a set was described by Chiou et al. [22]. Their proposal was to control which blocks that can be replaced in a set by software in order to reduce the number of conflicts. The scheme was intended for a multi-threaded core with a single cache.

Kharbutli and Solihin [52] use a counter to predict when cache blocks have no future temporal locality and should be evicted. Two algorithms were presented. One is based on access interval: For each cache block, it records the number of accesses to the set where the block resides, between consecutive accesses to that block. The other is based on live time: For each cache block, it records the number of accesses to itself during the interval in which the block resides continuously in the cache. A table is used to store information about earlier access patterns.

Paper IV presents a scheme for an augmented LRU replacement policy so that cache blocks that are accessed frequently are given more protection than infrequently accessed cache blocks. This scheme is different from Kharbutli and Solihin's scheme since in their scheme the cache blocks are categorized as either having temporal locality or not (when counter reaches maximum value), while in our scheme the cache blocks are compared. Another difference is that our scheme considers frequency and receny of use, while they predict when cache block loose their temporal locality based on repeating behavior patterns. In our paper we evaluate such a scheme for a CMP with shared last-level cache. One problem with a conventional LRU replacement policy in a shared cache is that one processor that works on a very large data set which does not fit into the cache, also evicts data for the other processors that otherwise would have fit into the cache. In Paper IV we show that using frequency as well as recency improves system performance by limiting the number of cache blocks one processor can evict for other processors; Cache blocks that are frequently accessed are better protected from eviction.

A different approach for exploiting more locality is to not store data in cache which are predicted to have low locality. Bypassing has earlier been studied for first-level caches. It has been used to reduce the required memory bandwidth by assuming that a single reference can be loaded by the processor instead of a cache block [21, 105]. This can also reduce the miss rate since fewer cache blocks are replaced. However this benefit has only been shown for selected applications and an increase in miss rate is shown for the median performance of applications [105]. The overhead associated with fetching an entire cache block compared to a single word has diminished with introduction of burst transfer, wider buses and pipelined memory accesses. Different approaches for finding memory accesses to bypass are: static by compiler [21, 105], dynamic based on instruction address [105] and dynamic based on memory access pattern [43]. Using memory access patterns require tables with a size proportional to the size of memory. Other work has looked into algorithms for placing data into different caches based on historical performance [47, 50, 88, 116].

In Paper III a heuristic for finding cache blocks with low locality is described for the last-level cache. Previous studies [1, 82] have shown that a few load instructions are

responsible for most of the cache misses, called *delinquent loads*. Our approach is to base the prediction of which blocks to be bypassed by detecting such loads and record them at run-time. However, we validate the correctness of the prediction and change it by also monitoring whether we erroneously bypass a block with a reuse distance shorter than the last-level cache but longer than the first-level cache by monitoring the reuse at the last-level cache. While this basic approach uses some of the components from the dynamic scheme proposed by Tyson et al. [105], we found that the Tyson scheme increased the miss rate for many applications. By bypassing cache blocks in the last-level cache fetched by these instructions (i.e. install the cache blocks only in the higher level caches), the cache space is utilized better by keeping cache blocks with higher locality.

3.3.6 Scratch-pad Memory

Scratch-pad memory is a small and fast memory area directly controlled by the processor. The miss-rate in cache depends on access patterns and replacement policies and is hardware controlled. Unlike the first-level cache, the program can control which data that are stored in the scratch-pad memory. The scratch-pad memory can consist of several bank which works independently. The direct memory access (DMA) can fill one bank while the processor is working on the data in the second bank.

Banakar et al. [9] have studied the efficiency of scratch-pad memories and found that performance is higher, power consumption is lower and chip area usage is lower compared to conventional caches. Scratch-pad memories can not be used on conventional applications without profiling, recompiling or reprogramming. It is therefore not pursued in this thesis. It is used in embedded systems where the programmer knows exactly which hardware the program will run on and where the application and hardware can be simple. It is used in game consoles such as Sony Playstation 2 and 3 [60].

3.4 Reducing Off-chip Communication Latency

The off-chip communication latency is significant because of the high signal delay (long wires and RC-delay) and transmission delay (limited width of the bus and limited number of IO-pins). Previously the signal delay was the main concern because transmission delay took less time. However, with the growing number of processors in a single chip, and a much lower growth in IO bandwidth, bandwidth is becoming an issue. For CMPs this results in contention for the communication channel.

One efficient solution is to remove the need to go off-chip by merging processor and main memory on the same chip. This concept is discussed in Section 3.7.2. The rest of this section assumes a more conventional architecture with external main memory (DRAM) chips.

There are techniques for improving this latency. The external bus can be made wider which can increase the bandwidth. However, the number of pins on the chip is limited and growing slower than the processor performance (see Section 6.3). In some cases the bus can be made shorter to reduce latency. Different signaling techniques can also reduce latency and improve bandwidth, but this section considers more architectural approaches. There has been a lot of research on the interface between the processor and the DRAM chip/macro which speeds up serial accesses (e.g. fast page mode/burst mode). Also by locking the output buffer of the DRAM macro, subsequent serial accesses within the same column is fast (no need to lookup column). Split transactions can increase utilization of the bus.

One architectural technique to reduce this latency is to reduce the number of accesses going over the bus. This will reduce the contention of the bus and hence reduce the latency. Techniques for reducing the number of cache misses in the last-level cache will decrease the number of off-chip accesses.

A different approach is to utilize *compression*. Compression of the IO data increases the amount of data that can be transferred per time unit [5, 29, 58]. For large data volume the latency can be reduced because of higher bandwidth even though compression adds a new component to the latency. Ekman and Stenström [29] proposed a scheme where simulated performance is only degraded by 0.2% while 30% of the memory resources are freed up.

Dirty cache blocks have to be written back to DRAM while clean cache blocks can just overwritten without write-back. In the scheme presented in paper IV, blocks that are dirty are less likely to be evicted than blocks that are clean. Even though bandwidth requirement is only slightly reduced, the principle is interesting.

Merging write buffers can reduce the number of blocks that are sent off-chip. This technique collects several modifications to a cache block before it is written back and hence reduces bandwidth requirements.

Reducing the block size can also reduce the amount of data that are transferred over the bus. However, too small cache blocks can increase miss rates and hence decrease the performance. Smaller cache blocks combined with a high precision prefetcher can reduce bandwidth requirements without decreasing performance. Not loading whole cache blocks on cache miss if the rest of data in the cache block are not predicted to be accessed is another technique [105] which is based on the same principle. Since most processors use prefetchers, increasing the accuracy of these prefetchers will decrease bandwidth requirements.

3.5 Reducing the Miss Latency for the Last-level Cache

3.5.1 Critical Word First and Early Restart

In configuration with large cache blocks critical word first means that the actual requested word is loaded before the rest of the block and sent to the processor so it can continue. Early restart is based on the same idea, but data is read in-order. The processor can however restart when the data requested is loaded, which can be earlier than when the whole block is loaded [33].

3.5.2 Priority to Read Misses Over Write Misses

Since the processor can continue in cases with write misses (if there are write buffers), giving priority to read misses over write misses can reduce the stall time for the processor and hence the effective miss latency [33]. The write buffer writes to the lower level cache/main memory when there are available resources. The read operations have to check the write buffer on requests to ensure that the latest value is read.

3.5.3 Non-blocking Caches

Non-blocking caches [61,95] allows several independent accesses to proceed in the presence of waiting misses. This reduces the miss latency for some of the accesses that do not need to wait for the first access to complete before execution is started.

3.5.4 Increasing Parallelism

The effect of the latency can be reduced by increasing parallelism. This can be done by the programmer, the compiler or in run-time by the hardware. (Simultaneous) multithreading is one level of such parallelism [104]. Independent instructions and independent loop iteration are other examples [20] .

3.5.5 Early Miss Determination

Memik et al. [72] proposed different techniques for predicting misses in lower level caches. In this way the request can bypass some of the cache levels (since the cache block is not predicted to be in these caches) and avoid some of the latency. We use the same principle in Paper III; the miss time is reduced by introducing a fast heuristic that is able to predict if data is found in the last-level cache. Since this logic is faster than the miss confirmation of the last-level cache, the total latency is reduced. This heuristic is

different than Memik et al.'s heuristic. In our work we show that the same heuristic that is used for bypassing can successfully be used for early miss determination.

3.6 Reducing the Hit Latency for the Last-level Cache

3.6.1 Making the Cache Simpler

Basically a cache hit can be made faster by making the cache simpler. Making the cache smaller improves the hit time as well. First-level cache is often split into instruction and data cache to improve hit latency. It makes each of the two caches smaller and faster. The interconnect between the processor and the first-level cache becomes simpler since the units connected are divided between the two caches. Handling several requests in parallel is simpler with two caches than with a single cache, and hence the architecture of the cache can be simpler and faster.

3.6.2 Reduce Associativity

One problem with a highly associative cache is the relative high hit latency because the cache block has to be selected from one out of several possible cache blocks. Since the last-level cache usually is highly associative, there is some potential for decreasing hit time. Further the power consumption is increased since there is more logic working in parallel to find a possible match. Way prediction [16, 18, 40, 51] is an approach to achieve both high-performance and low-power consumption. First the most recently used (MRU) block in a set it checked for a hit, and if this is a hit, it is faster compared to a conventional set-associative cache. The power consumption is low because only one comparison is required. However, if the hit is not in the MRU block the other blocks have to be checked for a tag match as well, and the cache becomes slower than a conventional cache.

A different approach is pseudo-associative caches proposed by Agarwal and Pudar [2]. It works like a direct mapped cache, but on misses a second location is checked as well (e.g. by switching the highest bit of the address). Effectively as a 2-way associative cache, but with only a single comparator. The access time depends on "way-position" of the hit. Hits in the first position that is checked are as fast as a direct mapped cache, otherwise they are slower and even slower than a 2-way associative cache.

3.6.3 NUCA Caches

Recently there has been research on non-uniform cache architectures [11]. The basic idea is that the latency of the data depends on where the data is stored. For CMPs this

means that data that is located physically closer to the each processor has lower latency.

Spilling evicted cache blocks to a neighboring cache has been described by Chishti et al. [23] and Chang and Sohi [19]. Before a cache block is evicted it is installed into a random neighboring cache. Local cache hits are faster than hits to the neighboring cache. One problem with this solution is *pollution*, i.e. that a processor can evict dead cache blocks into the neighboring cache and cause eviction of needed cache blocks. They did not consider putting constraints on the sharing (i.e. installing cache blocks in the cache for other processors) or other methods for protection from pollution. No mechanism was described for optimizing partition sizes, i.e. the cache space was arbitrary partitioned. We extend their work by insertion of constraints for cache usage, dividing the cache space in private and shared partitions and mechanisms for finding the best partition sizes in Paper VI. Each processor can use a limited number of blocks in its own cache and a limited number of the shared cache blocks for each set. As our results show, the extensions improve performance significantly. Zhang and Asanovic [121] evaluate a combination of shared and private caches. The ultimate goal is to have the size of the shared cache available to all processors with the speed of the local cache. They do this by replicating cache blocks so that it can be close to several processors. Their solution is intended for parallel programs working on the same data set.

3.7 Reducing the Access Time of Main Memory

3.7.1 Latency of DRAM Memory

In DRAM, data is stored in capacitors as shown in Figure 3.3. One transistor is associated with each capacitor and is used to transfer charges to and from the capacitor. A read or write operation involves accessing all the capacitors in a specific line by enabling the word line. The charges from the capacitors are transferred to the sense amplifiers through the bit lines. The sense amplifiers interpret the charge as a digital signal, recharge the capacitor and submit digital signals to the column decoders. The number of bit lines are usually much higher than the number of IO-pins. The column decoder connects the required bits to the input/output channel.

There are many word lines and bit lines in the DRAM physically laid out in parallel. This causes a significant capacitance between the lines. Additionally the lines are relative long and thin. The result is a significant RC-delay. Communication between the DRAM and the processor usually goes off-chip and the length of the wires causes a large communication latency. Additionally the address decoding and column decoding cause further latency.

Due to the leakage current in the capacitors, they have to be refreshed periodically. This is done by reading the content since reading involves writing back the content and thus

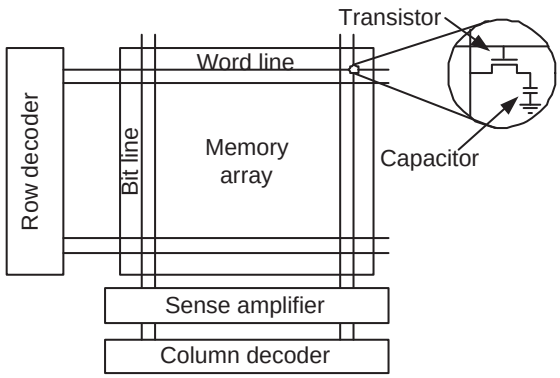


Figure 3.3: Logical drawing of a DRAM macro. Inside the circle a single DRAM cell is drawn.

recharging the capacitors. This can cause contention to the DRAM and hence cause latency.

3.7.2 Eliminating Off-chip Communication Latency

Off-chip communication can be eliminated by moving the DRAM into the same chip (die) as the processor. The on-chip bus can be much wider and shorter and is less likely to be a bottleneck. Several projects have researched into architectures with merged memory and processors [26, 30, 49, 57, 59, 67, 80, 84, 119] and some systems have been commercialized [31, 97].

Paper I and II assume an architecture with merged DRAM and processor, and hence this off-chip communication is eliminated.

3.7.3 Reducing Address Decoding Latency

Figure 3.4 illustrates the inner workings of a DRAM chip. Usually there is a latch in the row-address decoder. This latch is used because the row-address and column-address share pins on the package. This reduces the number of pins and hence the cost of the chip. The column-address is required after the row-address (it is required after the data is read out of the memory area). However, this latch introduces some latency. By eliminating this latch the address can reach the row-decoder faster. This can be done by adding more pins to the chip and by not sharing the address lines. This is easier and less expensive when the DRAM and processors are merged into the same chip.

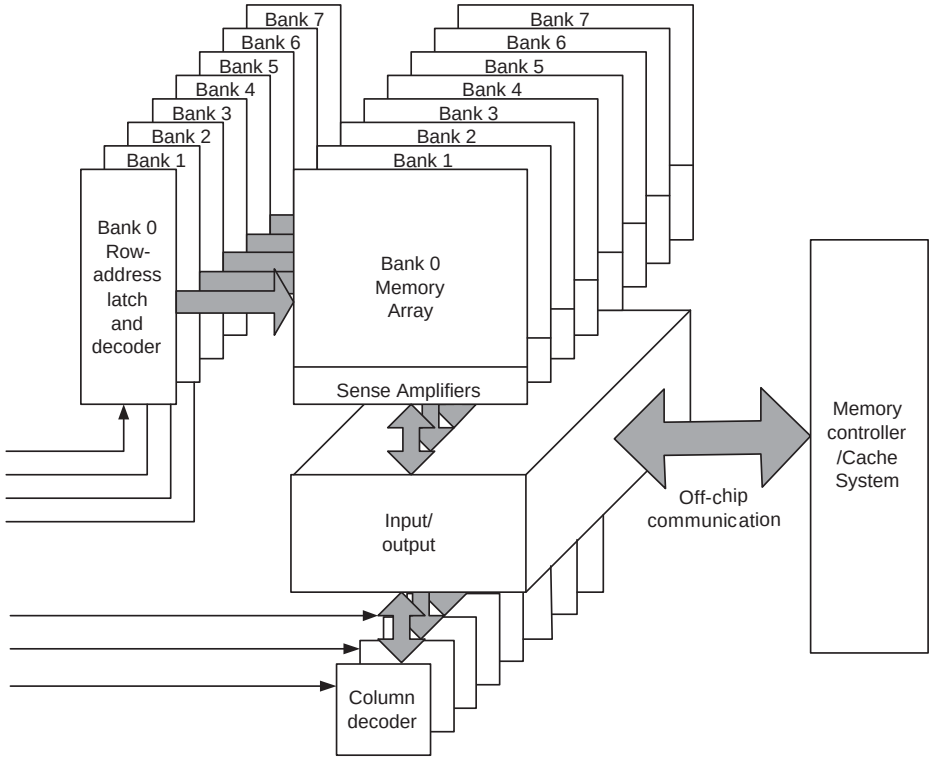


Figure 3.4: A logical sketch of a DRAM design for a memory chip. The chip consists of several memory arrays. This reduces the size of each array and hence the latency.

Paper I and II assume an architecture with merged DRAM and processor, and the address is sent in parallel without splitting it into two bus transactions and without this latch.

Most of the latency for the address decoding is caused by the logic for selecting which word line to activate. One approach to reduce this latency is to remove the row decoder and have one address line per word line [75]. This approach increases the number of pins and does not fit with a conventional architecture.

3.7.4 Word-Line Activation Latency

This delay is caused by capacitance created by the number of transistors that are connected to the word lines. Reducing the size of the memory bank reduces the number of transistors per word line, but it does increase the overhead area (and cost) for the logic. Figure 3.4 shows that there is an array of memory arrays inside a chip and hence balanc-

ing the latency and overhead cost is already implemented in conventional chips. Other techniques used includes different signaling voltages and other metals or thickness of the word line wire. It is beyond the scope of this thesis to look into this in more detail.

3.7.5 Bit-Line Sensing Latency

Since the capacitors that store the charges are small and the bit lines are relative long, the RC-delay is significant. Bit lines are densely laid out and capacity between the lines also adds to the RC-delay, see Figure 3.3. Additionally data is lost after the charge has been used to drive the bit line and the capacitor has to be charged again.

In Paper I and Paper II the latency of the bit line sensing is reduced by not recharging the capacitor. This technology is called destructive-read DRAM and was proposed by Hwang et al. [38]. In their work a write-back buffer ensured that data were written back to the capacitors. Our approach in Paper I and II with destructive-read DRAM requires no additional write-back buffer or expensive logic for write-back of data. Instead the data is written back later from the cache. We find that no new congestion is created and the power consumption is not increased due to the extra write-back communication.

Direct sensing is a similar technique for reading data faster out of the bit line with additional logic for write back. However, this technique increased chip area by 14% [75].

Chapter 4

Methodology

4.1 The Simulator

In all the included papers simulations are used to study the proposed techniques. These simulations are run on a simulator based on SimpleScalar [4] written by Todd Austin at University of Wisconsin. This simulator was chosen because the source code is open and it is one of the most popular simulators, if not the most popular [120]. Originally it simulates only a single processor. For multiprocessor simulations there are all kinds of simulators available such as:

- RSIM [36] by Hughes, Pai, Ranganathan and Adve, Rice University, 2002.
- Talisman [12] by Bedichek, Massachusetts Institute of Technology, 1995.
- Tango [25] by Davis, Goldschmidt and Hennessy, Stanford University, 1990
- Augmint [76] by Nguyen, Michael, Sharma and Torrella, University of Illinois at Urbana-Champaign/University of Rochester, 1996.
- MINT [107] by Veenstra and Fowler, University of Rochester/University of Copenhagen, 1994
- M5 [13], by Binkert, Dreslinski, Hsu, Lim, Saidi and Reinhardt, University of Michigan, 2006
- GEMS [69] by Martin, Sorin, Beckmann, Marty, Xu, Alameldeen, Moore, Hill and Wood, University of Pennsylvania/Duke University/University of Wisconsin-Madison, 2005
- Simics [66] by Magnusson, Christensson, Eskilson, Forsgren, Hallberg, Hogberg, Larsson, Moestedt and Werner, Virtutech AB, 2002

- Multiprocessor enhancements of the SimpleScalar [68] by Manjikian, Queen's University, 2001

Even though, I decided to extend SimpleScalar myself to support new architectures such as CMP. The rationale behind this is that there was no simulator for CMP easy available with the features needed for my research when I started, and I already had experience with SimpleScalar. SimpleScalar comes with cross-compilers and pre-built benchmarks.

Wattch [14] is an extension to SimpleScalar written by Brooks, Tiwari and Martonosi and is used to predict dynamic power consumption in Paper II. Hotleakage [92] was written by are Skadron, Stan, Huang, Velusamy, Sankaranarayanan and Tarjan and used for static power estimation purpose. Cacti [115] written by Wilton and Jouppi has been used to estimate cache latency.

The actual configuration of the computer systems simulated is different for each paper. E.g. the first two papers consider a simple processor core while the others consider a memory latency tolerant processor.

4.2 Platform for Running Simulations

In order to speedup simulations, several simulations were run in parallel. Two clusters located at the NTNU Gløshaugen were used for running simulations. They are based on Ganglia [70] monitoring system and NPACI Rocks [83] for installation and maintenance. The *Norgrid* cluster [78] has 64 computer nodes and is operated by ITEA at NTNU. The *Clustis2* [24] cluster has 26 nodes and is operated by the institute (IDI/NTNU). Each node has 3.4 GHz Intel P4 processor, 1GB DRAM and 36GB SCSI disk and dual Gbit network interface card.

During the experiments each computer runs a specific combination of workloads and configurations. Since the SPEC2000 benchmark suit consists of many applications, the parallelization was trivial. Python scripts were used to initialize the experiments and parse the results.

The experience with using clusters shows that it is efficient. However, at times there might be competition for using the resources creating long queues. Scripts for automatic generation of workloads were efficient both for saving time and for avoiding errors. We found that it was important to be able to simply re-run jobs that were canceled or sent to dead nodes.

4.3 Generation of Workloads

There are a lot of values in the results from the simulations and the most important is instructions-per-clock cycle (IPC). However, the simulator is far from accurate. It would have been too imprecise to state that this new scheme results in a specific IPC value and then compared this value to measured values from real hardware. Instead the same benchmarks are simulated with a conventional configuration. The performance of the new scheme is compared to the simulated conventional architecture and this ratio is highlighted in the papers. The potential of the proposed techniques are shown by comparing the performance to oracle schemes, optimal schemes and to conventional schemes. The oracle schemes have perfect knowledge about the future and take the correct decisions and uses the same mechanisms as proposed in the new scheme. The optimal scheme is a non-implementable scheme and often simpler than an oracle scheme. For example an oracle cache scheme for cache replacement policy knows exactly which cache blocks to evict based on future knowledge, while an optimal cache scheme has only hits (no misses).

The basis for the workloads is the SPEC2000 [98] benchmark suite. While the first two papers used a reduced dataset (*lgred*) made for fast simulation of the workload, the rest of the papers use the reference dataset. However, only part of this dataset is simulated as it takes years to simulate the whole dataset. First some hundred millions instructions are forwarded to warm up the cache and instantiate the benchmarks, then some hundred million clock cycles are simulated.

For CMP workloads, random combinations of different applications from the benchmarks suite are run on the processors in parallel. The start of the interval for each application is chosen randomly. Each paper describes how this workload is composed and used. No workload with shared memory is included in the study.

It is assumed that the relative performance of the new scheme for these simulated intervals are representative for the benchmark applications. SMARTS (Sampling Microarchitecture Simulation framework) [112] is based on the same assumption, but their methodology is intended for studying the processor core and includes capturing processor states. Since SMARTS is capable of restarting from hard-disk images, they can pick simulation intervals from the whole program behavior, while we limit the simulation intervals to the first part of the program. Simulating last-level cache is different since the functional warming is much longer and hence saving state to disk is not suitable. Changes to the cache schemes would then require generation of new states which is extremely computation exhaustive. SimPoint [90] written by Sherwood et al. at University of California is a different approach where program phase behavior is analyzed. The goal is to find the simulations interval(s) that represents the whole program by analyzing behavior of the program. Modification to the last-level cache can change the phase behavior of the benchmarks and we chose not to use this approach.

Chapter 5

Paper Summaries

"In the middle of difficulty lies opportunity." — *Albert Einstein*

This chapter gives a summary of the research documented in this thesis based on the abstracts from the papers. Retrospective comments are given for each paper.

5.1 Paper I: Cache Write-Back Schemes for Embedded Destructive-Read DRAM

5.1.1 Details

Authors of the paper: Haakon Dybdahl, Marius Grannæs and Lasse Natvig

Presented at ARCS'06: Architecture of Computing Systems (ARCS), 13-16 March, Frankfurt/Main, Germany, 2006

Published in LCNS: Lecture Notes in Computer Science (LNCS) 3894, Springer 2006. Pages 145-159

5.1.2 Summary

Much of the chip area and power consumption in a modern processor are caused by mechanisms that compensate for slow main memory such as caches, out-oforder execution and prefetching. In this work we attack this problem by utilizing a new DRAM macro that is faster than conventional DRAM macros. The macro made by Hwang et. al enables faster random access to data, but does not conserve data in the DRAM cells after

reading. Hwang et. al. included a large write-back buffer in their prototype for conserving data and hiding all write-backs. We eliminate this buffer by utilizing the already existing cache in processor designs at the cost of potential memory bank congestions. The modified cache conserves data by writing data back to DRAM.We have studied the impact of different write-back schemes from cache to DRAM and looked at different performance issues in this context such as number of independent DRAM banks, writeback buffers and latency of DRAM. A theoretical scheme with free write-backs for data conversation is studied, and we show that our implementable schemes do not create significant congestion due to write-backs. Our baseline architecture for evaluation is a low-power processor with small caches and embedded DRAM. Our first conclusion is that the size of the cache can be highly reduced without degrading performance when utilizing our write-back schemes with destructive-read DRAM compared to conventional DRAM. Secondly, the large write-back buffer can be omitted when destructive-read DRAM is used with a processor with cache.

5.1.3 Retrospective

While this work looked at fitting the whole memory inside the chip with the processor, a different approach could have been to look at using destructive-read DRAM as on-chip cache. This would have been closer to current conventional architectures. It would also been interesting to study a new instruction set architecture (ISA) with special instructions for destructive-read which are faster than conventional read instructions.

There has not been much published on destructive-read DRAM lately. The only exception is one patent issued on September 2005 [74] for a write-back buffer for destructive-read DRAM that is also based on destructive-read DRAM.

Embedded DRAM (eDRAM) on the other hand, is widely used on game consoles even though it is still in an early development stage compared to SRAM and DRAM. Large memories on chip in eDRAM have several advantages compared to SRAM such as smaller area, lower power consumption and lower soft error rate [77]. It is available in 55 nm technology.

5.2 Paper II: Destructive-Read in Embedded DRAM, Impact on Power Consumption

5.2.1 Details

Authors of the paper: Haakon Dybdahl, Per Gunnar Kjeldsberg, Marius Grannæs, Lasse Natvig

To be published in Journal of Embedded Computing, IOS Press, Vol 2 Issue 2, 2006
Scheduled for November 2006

5.2.2 Summary

This paper explores power consumption for destructive-read embedded DRAM. Destructive-read DRAM is based on conventional DRAM design, but with sense amplifiers optimized for lower latency. This speed increase is achieved by not conserving the content of the DRAM cell after a read operation. Random access time to DRAM was reduced from 6 ns to 3 ns in a prototype made by Hwang et. al. A write-back buffer was used to conserve data. We have proposed a new scheme for write-back using the usually smaller cache instead of a large additional write-back buffer. Write-back is performed whenever a cache line is replaced. This increases bus and DRAM bank activity compared to a conventional architecture which again increases power consumption. On the other hand computational performance is improved through faster DRAM accesses. Simulation of a CPU, DRAM and a 2 kbytes cache show that the power consumption increased by 3% while the performance increased by 14% for the applications in the SPEC2000 benchmark. With a 16 kbytes cache the power consumption increased by 0.5% while performance increased by 4.5%.

5.2.3 Retrospective

The complexity and uncertainty are high for power simulations of computers. There are a lot of low-levels techniques to improve power consumption and it depends on the technology as well. However, there is a correlation between performance and power consumption at an architectural level. Simple techniques that increase performance can in general reduce power consumption per instruction executed. I did not pursue power simulations in the rest of the papers. I focused more on performance and on estimation of chip area.

5.3 Paper III: Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination

5.3.1 Details

Authors of the paper: Haakon Dybdahl and Per Stenström

Presented at ACSAC'06: Asia-Pacific Computer Systems Architecture Conference, Shanghai, China, September 6-8th, 2006

Published in LCNS: Lecture Notes in Computer Science (LNCS) 4186, Springer 2006. Pages 52-66

5.3.2 Summary

While bypassing algorithms have been applied to the first-level cache, we study for the first time their effectiveness for the last-level caches for which miss penalties are significantly higher and where algorithm complexity is not constrained by the speed of the pipeline. Our algorithm monitors the reuse behavior of blocks that are touched by delinquent loads and re-classify them on-the-fly. Blocks classified as bypassed are only installed in the level-1 cache.We leverage the algorithm to early send out a miss request for loads expected to request blocks classified to be bypassed. Such requests are sent to memory directly without tag checks at intermediary levels in the cache hierarchy. Overall, we find that we can robustly reduce the miss rate by 23% and improve IPC with 14% on average for memory bound SPEC2000 applications without degrading performance of the other SPEC2000 applications.

5.3.3 Retrospective

We did some limited work (not published) on using this scheme for a CMP, but ended up with creating the scheme in Paper IV instead. We believe that the bypassing scheme should be suitable for CMPs as well, but at a higher communication cost. Each processor has to send the address of the instruction that fetched the data to the last-level cache. In the following papers we increased last-level cache performance without the need for any new information to be submitted to the last-level cache.

In Figure 3, fifth row, "Cache miss caused by not bypassing", the sign of values should have been positive not negative.

5.4 Paper IV: An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches

5.4.1 Details

Authors of the paper: Haakon Dybdahl, Per Stenström and Lasse Natvig

Presented at MEDEA Workshop (held in conjunction with PACT Conference), September 16-20, 2006 Seattle, WA

Published at IEEE Xplore To be published in ACM SIGARCH Computer Architecture News

5.4.2 Summary

This paper proposes a new replacement algorithm to protect cache lines with potential future reuse from being evicted. In contrast to the recency based approaches used in the past (LRU for example), our algorithm also uses the notion of frequency of access. Instead of evicting the least recently used block, our algorithm identifies among a set of LRU blocks the one that is also least-frequently-used (according to a heuristic) and chooses that as a victim. We have implemented this replacement algorithm in a detailed simulation model of a chip multiprocessor system driven by SPEC2000 benchmarks. We have found that the new scheme improves performance for memory intensive applications. Moreover, as compared to other attempts, our replacement algorithm provides robust improvements across all benchmarks. We have also extended an earlier proposed scheme by Wong and Baer so it is switched off when performance is not improved. Our results show that this makes the scheme much more suitable for CMP configurations.

5.4.3 Retrospective

Qureshi, Lynch, Mutlu and Patt independently made something similar to *shadow tags* [86] which was published at ISCA June 2006. They came to the same conclusion that the number of sets required can be reduced and used statistical methods to motivate for the scheme. It is interesting to see that two separate research groups invented almost identical mechanisms. The selection methods for finding which cache blocks to sample are however different. It would have been interesting to compare the different selection methods as the assumption for the methods are different. While we have made the assumption that the sets with the lowest index are representative for the whole cache, Qureshi et al. believe that accesses are more randomly distributed. However, this is only a minor issue for our paper since the selection algorithm with shadow tags works almost perfectly and there is not much room for improvement.

There are many other approaches to improve the replacement policy, and more advanced methods are regarded as future work.

5.5 Paper V: A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors

5.5.1 Details

Authors of the paper: Haakon Dybdahl, Per Stenström and Lasse Natvig

To be presented at HiPC'06: IEEE International Conference on High Performance Computing, Bangalore, India, December 18-21, 2006 . **Received best paper award**. Two papers were selected for best paper awards out of 335 submitted papers

To be published in Springer, Lecture Notes in Computer Science (LNCS)

5.5.2 Summary

Chip multiprocessors (CMPs) usually employ shared, last-level caches to use on-chip memory resources effectively. Unfortunately, conventional replacement policies applied to shared caches fail to partition memory resources among cores to achieve an optimal execution throughput. This paper presents a novel replacement policy that dynamically estimates how many misses would be eliminated if one more block per set would be allocated to a certain processor taking into account the extra misses for some other processor. Our implementation makes novel use of shadow tags for the estimation. We show that it can yield 50% higher execution throughput on a 4-way CMP and in contrast to previously proposed schemes, we did not observe any noticeable degradation of performance for any application in the SPEC2000 we used.

5.5.3 Retrospective

Including shared memory workloads is regarded as future work. It requires a simulator with cache coherence and different workloads. The scheme must be extended to support this, but there should not be any major obstacles for doing this.

5.6 Paper VI: An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors

5.6.1 Details

Authors of the paper: Haakon Dybdahl, Per Stenström

To be presented at HPCA-13: International Symposium on High-Performance Computer Architecture-13, Phoenix, Arizona, February 10-14, 2007

To be published in conference proceedings and uploaded to IEEE Xplore

5.6.2 Summary

The significant speed-gap between processor and memory and the limited chip memory bandwidth make last-level cache performance crucial for future chip multiprocessors. To use the capacity of shared last-level caches efficiently and to allow for a short access time, proposed non-uniform cache architectures (NUCAs) are organized into per-core partitions. If a core runs out of cache space, blocks are typically relocated to nearby partitions, thus managing the cache as a shared cache. This uncontrolled sharing of all resources may unfortunately result in pollution that degrades performance. We propose a novel non-uniform cache architecture in which the amount of cache space that can be shared among the cores is controlled dynamically. The adaptive scheme estimates, continuously, the effect of increasing/decreasing the shared partition size on the overall performance. We show that our scheme outperforms a private and shared cache organization as well as a hybrid NUCA organization in which blocks in a local partition can spill over to neighbor core partitions.

5.6.3 Retrospective

Shared memory workloads for the presented scheme is regarded as future work. The scheme can also be described/implemented at a lower abstraction level and with more details.

It could be interesting to simulate the scheme with a higher number of processor.

Chapter 6

Concluding Remarks

> "Predicting the future, as we all know, is risky. Predicting the evolution of new technology is downright hazardous." — *Leon Cooper*

This chapter gives some final conclusions in section 6.1 and outline some possible avenues for future work in section 6.2. Section 6.3 presents a personal view on some important challenges facing the field of memory systems in the near future.

6.1 Conclusions

The sustained exponential growth of processor performance is now facing several obstacles. First of all power consumption has reached an unpractical high level. Secondly, the processor-memory performance gap is huge and problematic.

In state-of-the-art processors a large portion of the chip area is used by the cache. This cache uses a large share of the power, and this share will grow with denser technology due to higher leakage currents. Improving cache performance is therefore important both to reduce the effect of the processor-memory performance gap, but also to limit power consumption. This thesis proposes technical improvements for the last-level cache which is the largest cache in the processor. The improvements can be classified as: Technique for reducing the latency of DRAM by using the cache as a write-back buffer, techniques for improving locality in the last-level cache, techniques for improving utilization of cache space for CMPs, technique for switching between cache schemes and a novel cache architecture for CMPs.

The techniques were studied by simulation. However, the simulations are done with a limited number of benchmarks and only for parts of the benchmarks. The numbers for performance, area requirements and power consumption for the new schemes have

some uncertainty. Also the efficiency of the technique for destructive-read DRAM depends on specific technology. There is an unlimited number of sensitivity analysis that can be done when simulating computer architecture. Exact numbers for improvements depend on the configuration parameters such as memory latency and processor speed. Even with all this uncertainty and the limitations of the simulations, we believe there is enough evidence for the conclusions draw in the different papers. We have focused on understanding under which circumstances a proposed scheme works and when it does not work. The techniques are compared to theoretical upper limits and to other proposed state-of-the-art techniques. In general we can conclude that these new techniques look promising for applications with large data set, and at the same time they are robust for a mix of different applications.

None of the proposed techniques are revolutionary, but are contributions to existing architectures. Even if none of the techniques will find their way directly into a high performance processor, they can inspire other researchers and be the first steps toward new and improved techniques for next generation processors.

6.2 Future Work

Shadow tags are used to measure performance for running an alternative configuration in parallel with an existing configuration. The mechanism is inexpensive and simple. We have shown in several papers that storing information about what would have been the consequence of having a different cache block in cache or using a different configuration are efficient to make adaptive architectures. This idea is used for feed-back loop for classification of instructions for bypassing and balancing cache sizes. The same idea can be applied to different problems that require an adaptive solution.

There has been some work on heterogeneous processor CMP architectures [62]. However, there has not been much research on heterogeneous cache/interconnect CMP architectures. Utilizing shadow tags for finding best match of application and core is possible future work.

In the future the expected number of processor cores in a single chip is expected to grow. The organization of the last-level cache will be influenced by this development. NUCA caches is one example of possible future architecture, but more researched is needed in this area.

Also in Chapter 5 and in each paper there are some hints about topics that could be investigated further.

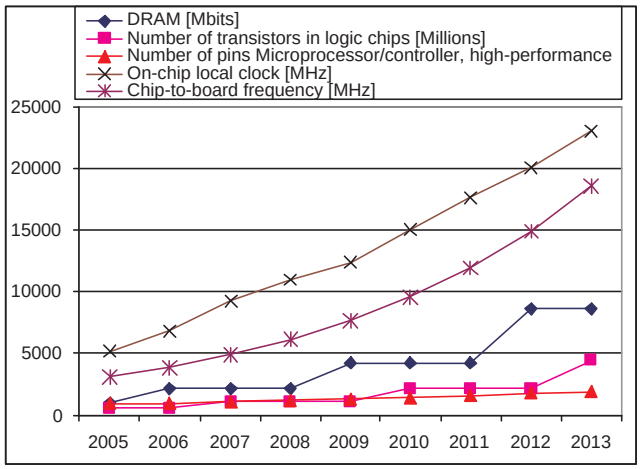


Figure 6.1: Future trends in number of pins, bandwidth and number of transistors per chip based on numbers from ITRS [89].

6.3 Outlook

The International Technology Roadmap for Semiconductors (ITRS) has a lot of prognosis about the outlook for technology scaling [89]. Figure 6.1 contains the prognosis for selected features for future chips. One issue with future processors is the bandwidth bottleneck. As the figure shows, the prognosis is that the internal frequency will grow with the same pace as the chip-to-board frequency. If we compare the growth of transistors with the growth of IO pins there are two different growth rates. For 70 nm technology there are about 1 million transistor per IO pin while there are 2.5 and 6 millions transistor per IO pin for 50 nm and 35 nm technology respectively [37], i.e. each transistor will have much less IO bandwidth. Merging memory and processors on the same chip might solve some of this problem since the IO bandwidth requirement is reduced. This will raise new problems and new challenges and call for new architectures.

Latency (in terms of clock cycles) of sending data across the chip increases as the density of the chip increases for a fixed chip (die) size. The difference in latency between main memory and the processor is going to continue to grow as well. This makes cache systems, register usage and scratch pad memories even more important. The problem will involve the programmer and algorithm designer to a much greater extent than it is today. The Cell processor is an existing example where the processor has high performance but requires complicated parallel programming to work efficiently.

One emerging obstacle is the increasing static leakage current. Denser technologies have a higher leakage current. Transistors therefore consume power without switching as long as they have supply voltage. Powering down part of the chip that is not needed at

the moment can increase efficiency. This requires that the computer knows how urgent it has to solve tasks. How much of the cache must be power up to respond quickly enough? How many processors should be powered up? These are possible new tasks to be handled by the operating system.

The instruction set architecture has until now been seen as the contract between the programmer and the hardware architect. With new architectures with increasing number of processors this view might change and a more complex interface between the programmer and the architect might become necessary.

With today's technology several hundred processor cores can fit in a single chip. Theoretically such a processor chip can have a very high performance. There are though several problems such as how to feed the cores with enough data to process. There is no simple answer to how the the interconnect should function. Programming models and languages for these cores are not mature. Today such a computer has to be programmed by very skilled programmers and even these programmers have difficulties in utilizing all cores.

Bibliography

[1] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of load/store instruction latencies. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 139–152, 1993.

[2] A. Agarwal and S. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. Technical report, Cambridge, MA, USA, 1992.

[3] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 212, 2004.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. IEEE Comp., V35I2, 2002.

[5] B. Abali B, H. Franke, X. Shen, D. Poff, and B. Smith. Performance of hardware compressed main memory. Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA), pages 73-81, 2001.

[6] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, 1988.

[7] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[8] BalusC server, CPU info. http://balusc.xs4all.nl/srv/har-cpu.html, 2006.

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[10] Amos Batto. A better upgrade, not a faster throw-away: An activist guide to minimizing the social and environmental impact of computers and reforming the industry. http://www.ciber-runa.net/guide/BetterUpgrade–ActivistGuide.pdf.

[11] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, 2004.

[12] Robert C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 14–24, 1995.

[13] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, 2000.

[15] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946). Perspectives on the computer revolution, 1989, pages 39–48.

[16] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. *HPCA*, 00:244, 1996.

[17] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, 1991.

[18] J. H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS system/370. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 208–213, 1987.

[19] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. ISCA, 2006.

[20] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Water, , and Wen mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. Proceedings of the 18th Annual Int'l Symposium on Computer Architecture(ISCA), 1991.

[21] Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. *Proceeding of the 22th Annual Hawaii Inter. Conf. on System Sciences*, 1989.

[22] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. Proceedings of Design Automation Conference, Los Angeles, 2000.

[23] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. ISCA, 2005.

[24] Clustis2 web page at NTNU. http://clustis2.idi.ntnu.no/.

[25] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using Tango. In *Multiprocessor performance measurement and evaluation*, pages 141–149, 1995.

[26] Jeff Draper, Chang Woo Kang, Ihn Kim, Gokhan Daglikoca, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, and Chun Chen. The architecture of the DIVA processing-in-memory chip. Proc. 16th ACM Int'l Conf. Supercomputing, pages:14-25, June 2002.

[27] Michel Dubois. Fighting the memory wall with assisted execution. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 168–180, 2004.

[28] Michel Dubois, Jonas Skeppstedt, and Per Strenström. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, 1995.

[29] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. *SIGARCH Comput. Archit. News*, 33(2):74–85, 2005.

[30] D. G. Elliott, W. Martin Snelgrove, and Michael Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In Custom Integrated Circuits Conference, Boston, MA, pages:30.6.1-30.6.4, May 1992.

[31] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory; the Terasys massively parallel PIM array. IEEE Computer, pages:23-31, April 1995.

[32] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, 1991.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[34] Mark D Hill. Aspects of cache memory and instruction buffer performance. Ph.D. dissertation. University of California, Berkeley, 1987.

[35] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag correlating prefetchers. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 317, 2003.

[36] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, 2002.

[37] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future CMPs. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[38] C-L. Hwang, T. Kirihata, and M Wordernan et.al. A 2.9ns random access cycle embedded DRAM with a destructive-read architecture. VLSI Circuits, Digest of Technical Papers, IEEE Symposium on, p:174-175, 2002.

[39] Power 4 processor design. http://www.research.ibm.com/power4/, 2006.

[40] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 273–275, 1999.

[41] L.K John and A. Subramanian. Design and performance evaluation of a cache assist to implement selective caching. *Proc. of Intl. Conf. on Comp. Design, pp. 510-518*, 1997.

[42] David J. C. Johnson. HP's mako processor. Fort Collins Microprocessor Lab, http://ftp.parisc-linux.org/docs/whitepapers/mako_mpf_2001.pdf.

[43] T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W Hwu. Run-time cache bypassing. *IEEE Trans. on Comput. V48I12*, 1999.

[44] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, 1997.

[45] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ISCA 1990: 364-373*, 1990.

[46] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in two-level on-chip caching. In *ISCA*, pages 34–45, 1994.

[47] M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *Proc. of Computing Frontiers*, 2004.

[48] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: an application-driven study. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 195–206, 2002.

[49] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Patnaik, and J. Torellas. FlexRAM: Towards an advanced intelligent memory system. International Conference on Computer Design, October 1999.

[50] M. Karlsson and E. Hagersten. Timestamp-based selective cache allocation. *High Performance Memory Systems, Springer-Verlag, 2003*, 1990.

[51] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 131–139, 1989.

[52] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. ICCD, 2005.

[53] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. IRE Transactions on Electronic Computers 11, 2 223-235, 1962.

[54] Changkyu Kim, Doug Burger, and Stephen W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. IEEE Micro, vol. 23, no. 6, pp. 99-107, 2003.

[55] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2004.

[56] Sunil Kim and Alexander V. Veidenbaum. Stride-directed prefetching for secondary caches. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, page 314, 1997.

[57] G. Kirsch. Active memory: Micron's Yukon. Parallel and Distributed Processing Symposium, Proceedings. International, pages:11, April 2003.

[58] M. Kjelso, M. Gooch, and S. Jones S. Design and performance of a main memory hardware data compressor. In Proceedings of the 22nd Euromicro Conference, pages 423-429, 1996.

[59] P.M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and logic chip for massively parallel systems. IEEE, Advanced Research in VLSI, Proceedings. Sixteenth Conference on, pages:4-16, March 1995.

[60] Kevin Krewell. Cell moves into the limelight. Microprocessor Reports, 14th Feb., 2005.

[61] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, 1981.

[62] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings*

of the 15th international conference on Parallel architectures and compilation techniques, pages 23–32, New York, NY, USA, 2006. ACM Press.

[63] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 144–154, 2001.

[64] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, 1991.

[65] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 184, 1999.

[66] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[67] K. Mai, T. Paaske, N. Jayasena, W R. Ho, Dally, and M. Horowitz. Smart Memories: A modular reconfigurable architecture. ISCA, June 2000.

[68] Naraig Manjikian. Multiprocessor enhancements of the SimpleScalar tool set. *SIGARCH Comput. Archit. News*, 29(1):8–15, 2001.

[69] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[70] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. In *Proceedings on Parallel Computing*, 2004.

[71] Scott McFarling. Cache replacement with dynamic exclusion. ISCA, 1992.

[72] Gokhan Memik, Glenn Reinman, and William H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 307, 2003.

[73] Gordon E. Moore. Cramming more components onto integrated circuits. Electronics Magazine, 1965.

[74] Seiji Munetoh, Toshiaki K. Kirihata, Chorng-Lii Hwang, and Brian L. Ji. Destructive-read random access memory system buffered with destructive-read memory cache. United States Patent 6948028, 2005.

[75] T. Nagai, K. Numata, M. Ogihara, M. Shimizu, K. Imai, T. Hara, M. Yoshida, Y. Saito, Y. Asao, S. Sawada, and S. Fujii. A 17-ns 4-Mb CMOS DRAM. Solid-State Circuits, IEEE Journal of , vol.26, no.11pp.1538-1543, 1991.

[76] Anthony-Trung Nguyen, Maged Michael, Arun Sharma, and Josep Torrella. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, page 486, 1996.

[77] NEC: eDRAM advantages over embedded SRAM. http://www.necelam.com/edram90/edramadvantages.html, 2006.

[78] Introduction to Norgrid, Wiki at NTNU. http://norgrid.ntnu.no/norgrid/NorgridIntroduction.

[79] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski. Blue Gene/L compute chip: Memory and ethernet subsystem. IBM Journal of Research and Development, Mar-May, 2005.

[80] M. Oskin, F.T. Chong, and T. Sherwood. Active Pages: A model of computation for intelligent memory. International Symposium on Computer Architecture, Barcelona, Spain, 1998.

[81] Subbarao Palacharla and R.E. Kessler. Evaluating stream buffers as a secondary cache replacement. In Proceedings of the 21th Annual International Symposium on Computer Architecture, 1994.

[82] Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. Static identification of delinquent loads. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, pages 303–314, 2004.

[83] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. NPACI rocks: Tools and techniques for easily deploying manageable linux clusters. *cluster*, 00:258, 2001.

[84] D. Patterson, T. Anderson, and K. Yelick. A case for intelligent DRAM: IRAM. Presented at Hot Chips VIII, Palo Alto CA, pages:18-20, August 1996.

[85] Ken Polsen. Web pages: Chronology of microprocessors. http://www.islandnet.com/~kpolsson/micropro/.

[86] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for MLP-aware cache replacement. *SIGARCH Comput. Archit. News*, 34(2):167–178, 2006.

[87] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP, Vol. 1*, pages 154–163, 1996.

[88] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS '98*, 1998.

[89] International Technology Roadmap for Semiconductors. http://public.itrs.net/, 2005.

[90] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[91] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[92] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 2–13, 2003.

[93] Alan Jay Smith. Sequential program prefetching in memory hierarchies. IEEE transactions on computers, vol 11, No 12, pp.7-21, 1978.

[94] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[95] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 53–62, 1991.

[96] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 171–182, 2002.

[97] ASC9: 90 nm CMOS process technology, the industry's first practical 90 nm, embedded DRAM process. CX-News, vol 34, http://www.sony.net/Products/SC-HP/cx_news/vol34/, November 2003.

[98] SPEC benchmark suite. Information available at http://www.spec.org/.

[99] Per Stenström. A survey of cache coherence schemes for multiprocessors. IEEE Computer, Vol 23, No 6, 1990.

[100] Per Stenström. Chip-multiprocessing and beyond. Keynote speech at HPCA-12, Feb 15, Austin, Texas, 2006.

[101] Rabin A. Sugumar and Santosh G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 24–35, 1993.

[102] G. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. IASTED Int. Conf. on Parallel and Distributed Computing Systems, 2001.

[103] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. HPCA, 2002.

[104] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In 22nd Annual International Symposium on Computer Architecture (ISCA), 1995.

[105] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to cache management. *Proc. of the 28th Annual Symp. on Microarchitecure*, 1995.

[106] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[107] Jack E. Veenstra and Robert J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *MASCOTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 201–207, 1994.

[108] David Tawei Wang. Modern DRAM memory systems: Performance analysis and a high performance, power-constrained DRAM scheduling algorithm. Doctor of Philosophy Thesis, 2005.

[109] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 140–148, 1989.

[110] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the compiler to improve cache replacement decisions. PACT, 2002.

[111] Fredrik Warg. Techniques to reduce thread-level speculation overhead. PhD thesis, Chalmers University of Technology, 2006.

[112] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe. SimFlex: Statistical sampling of computer system simulation. Micro, IEEE, vol.26, no.4pp. 18- 31, July-Aug., 2006.

[113] Wikipedia: Moore's Law. http://en.wikipedia.org/wiki/Moore's_law.

[114] M. Wilkes. Slave memories and dynamic storage allocation. IEEE Transactions on Electronic Computers, EC14(2):270–271, 1965.

[115] S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model, 1996.

[116] Wayne A. Wong and Jean-Loup Baer. Modified LRU policies for improving second-level cache behavior. *HPCA*, 2000.

[117] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[118] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *In Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–265, 2000.

[119] L.V. Yerosheva, S.K. Kuntz, J.B. Brockman, and P.M. Kogge. A microserver view of HTMT. Parallel and Distributed Processing Symposium, Proceedings 15th International, pages:10, April 2001.

[120] Joshua J. Yi and David J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, 2006.

[121] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. ISCA, 2005.

[122] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. *SIGPLAN Not.*, 35(11):150–159, 2000.

Index

The Papers

The papers

This appendix contains the collection of the six papers which are verbatim copies of the originally published versions. The titles of the papers and the page reference are shown in the table below.

Cache Write-Back Schemes for Embedded Destructive-Read DRAM.

Haakon Dybdahl, Marius Grannæs and Lasse Natvig

Presented at Architecture of Computing Systems (ARCS), Frankfurt/Main, Germany, 2006.

Lecture Notes in Computer Science (LNCS) 3894
Springer 2006.
Pages 145-159.

Cache Write-Back Schemes for Embedded Destructive-Read DRAM

Haakon Dybdahl and Marius Grannæs and Lasse Natvig

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering
Norwegian University of Science and Technology
{dybdahl, grannas, lasse} @idi.ntnu.no

Abstract. This paper proposes two new write-back schemes for caches used in conjunction with destructive-read DRAM. Destructive-read DRAM is based on conventional DRAM design, but with sense amplifiers optimized for faster data access. This speed increase is achieved by *not* conserving the content of the DRAM cell after a read. Data is written back later, or the data will be lost. A prototype of the memory was made by Hwang et al. where random access time to DRAM was reduced from 6 ns to 3 ns. This paper proposes two new schemes for write-backs with the advantages of smaller caches and/or higher performance. The simulation of one of the scheme performs on average 13.5% better than systems with the conventional DRAM for systems with small caches. Cache size can be reduced with 75% by using these new schemes compared to conventional systems while achieving the same performance.

1 Introduction

Reducing the latency gap between main memory and *central processing units* (CPUs) has been a main objective for decades. This research has resulted in various mechanisms such as caches, out-of-order scheduling, prefetchers and simultaneous multithreading. These mechanisms consume a substantial amount of power and are thus an emerging challenge for designing high performance CPUs.

Main memory in most computers is accessed through off-chip buses that connect the CPU and the memory chips. This memory has a much higher internal bandwidth than what is available through these buses, as off-chip buses are limited by the number of pins on the chips as well as distance and capacitance. In *processing in memory* (PIM) architectures main memory is *embedded* together with the CPU on the same chip[1]. In PIM architectures the internal bandwidth of main memory is available to the CPU as there are no limiting off-chip buses. Although PIM benefits from the higher bandwidth, latency is still a factor.

Embedded DRAM is in general slower than CPUs, and many transistors are needed for the necessary caches. Caches made from *static random access memory* (SRAM) normally use 6 transistors to store one bit of data. As feature size shrinks, the static power consumption increases due to higher leakage currents[9]. Energy consumption by caches will therefore increase as technology gets denser. A new type of embedded DRAM has been prototyped by Hwang et al. [7] which enables lower latencies. This is implemented by modifying the sense amplifiers to perform read operations in two phases; first reading the data and then later writing data back. Reading a cell thus destroys

[1] This is also referred to as *intelligent memory*.

Type	Access Time
PC133 SDRAM	71.4nS
DDR266 DRAM	58.8nS
RL DRAM	25.0nS
130nm eDRAM	12.0nS
90 nm eDRAM Dense	8.0nS
90 nm eDRAM Fast	4.5nS
DDR SRAM	3.3nS
embedded SRAM	2.0nS

Table 1. Random cycle time for various memories [8].

its content and thus data must be put in a write-back buffer. The prototype required a write-back buffer with data size of 25% of the DRAM banks. Dybdahl et. al have studied the impact on power consumption by utilizing destructive-read DRAM[3]. In this paper, performance of two new schemes for writing data back to main memory are examined. These schemes require less buffer size than Hwang's model. The buffer works as a fast cache, and total performance is improved.

1.1 Related Work

Several projects have done research into merging processors and memory ([5, 4, 16, 17, 15, 11, 19, 2, 12]). Most of these projects do not research into DRAM design, but presume a conventional design. The C*RAM project[4] is an exception that integrated small processing elements into the sense amplifiers and utilized the parallelism available at that level. A scaled down prototype was made. It was a SIMD computer with single bit processors. This architecture was mainly suitable for problems with high data locality because of limited communication between the single bit processing elements.

Many other projects use SIMD architecture to utilize the extra bandwidth: the IRAM project [18], Yukon [12], Terasys [5] and Execube [14]. The Mitsubishi M32R/D [16] chip uses the bandwidth to increase the number of bits in the data bus between main memory and cache. The use of FPGA technology and independent processors have also been proposed to utilize the bandwidth ([15, 2]). During the last years, embedded DRAM has become more common, and more chips of this type are in mass production in graphics or network processors such as Sony's Playstation 2, EZchip's NP-1c network processor[6] and Nintendo's GameCube.

2 Embedded Destructive-Read DRAM

2.1 Embedded Memory

DRAM memory is cheap, dense and consumes little power compared to other technologies; therefore it is the main choice for main memory. However, the process of merging main memory and CPU is not trivial as DRAM uses capacitors to store data (see figure 1). DRAM fabrication technology is optimized for these analogue circuits. Logic circuits on the other hand are optimized for speed and power distribution. Therefore embedded DRAM is not as dense (bits per area) as pure DRAM chips (typically 50% less bits per area), but this is improving fast with newer technology.

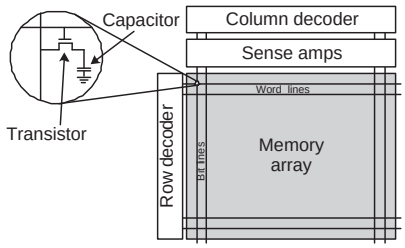


Fig. 1. Inside DRAM cell.

Different embedded DRAM designs have different latencies. Reducing the size of the memory array reduces the length of the lines and thus reduces latency. Each memory bank has extra circuitry such as sense amplifiers, and reducing the size of the memory array increases this overhead and lowers density. Table 1 contains random access latency time for various memory technologies. By reducing the size of each memory bank, a smaller unit is activated during an access. This reduces power consumption, but the static leakage from the added transistors will limit the minimum practical size. Off-chip buses consume substantial power, and embedding DRAM eliminates this consumption.

DRAM is not as fast as SRAM due to the construction of the cell and the way data are accessed. In SRAM the data is already represented as logic signal voltages, whereas in DRAM the capacitors have to be read and decoded into logic signal voltages. SRAM is therefore more suited for use in caches. However, as chips get denser, leakage power increases and transistors consume more power without switching.This is called static power consumption. Future caches will consume increasingly more power. DRAM will use less static power as they have only one transistor per data cell.

2.2 Destructive-Read DRAM

Destructive-read DRAM [7] is a modified version of conventional DRAM. A memory bank with conventional DRAM is shown in figure 1. The row decoder is the first component activated in a read access. It enables one *word line* and causes all transistors in that row to be activated. These transistors connect the capacitors in the memory array to the *sense amplifiers* through *bit lines*. The sense amplifiers work in three phases as shown in figure 2a. In the first phase the charge from the capacitor drives the sense amplifier into a logic state. In the second phase that logic state is locked. In figure 3a the locking works as a buffer. From this buffer the data is both sent to the processor and written back to memory. In the final phase the bit lines are pre-charged so they are ready for the next access. Destructive-read DRAM memory works differently. The read operation of conventional DRAM (see figure 2a) is split into two cycles (see figure 2b and c) The destructive-read DRAM does not lock the data after reading (as shown in figure 3b). Instead the data are sent directly out of the chip, in this case to the cache memory. Since data is not sent back to memory, data is destroyed after reading. Data is conserved by writing it back to DRAM after use as shown in 2c. However, write-back can be done later in contrast to conventional DRAM where read and write-back are a single operation.

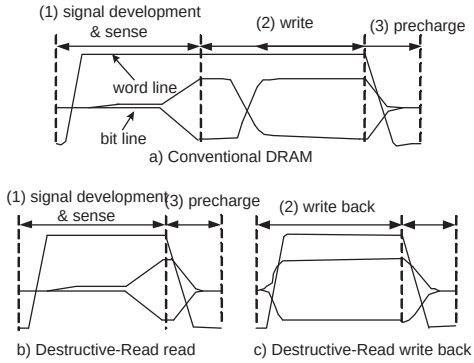


Fig. 2. Conceptual waveform diagrams of conventional DRAM architecture vs. destructive-read [7].

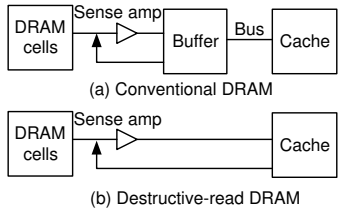


Fig. 3. Conceptual view of DRAM.

Hwang et al. made a prototype where random access time to DRAM was reduced from 6 ns (conventional read) to 3 ns (destructive-read). The prototype had four independent memory banks and large write back buffer (WBB) that was the same size as one memory bank. The WBB was made of SRAM. The purpose of the WWB was to hide write-backs, not to reduce latency time. The WBB could write-back to several banks simultaneously and required significant chip area. Later a new scheme was made where the WBB was replaced with destructive-read DRAM[10]. The design guaranteed that write-backs never conflicted with read operations.

2.3 New Write-back Schemes

Two new schemes for write-back will now be presented. The size of the WBB, or cache, is dramatically reduced without degrading performance. With these schemes the cache operates at a higher frequency than the DRAM banks.

The first scheme is the *delayed write-back scheme*. It can be compared to a cache that always has dirty cache lines. This implies that all data that are read into the cache have to be written back when replaced. A different approach is to write back data immediately after reading and is

called *immediate write-back scheme*. The differences between conventional DRAM and destructive-read DRAM with the immediate write-back scheme can be clarified by examining the steps in a read operation. For conventional DRAM a read operation is completed with the bit line in the DRAM not being changed. Also there is only one access on the memory bus. For destructive-read with immediate write-back scheme, the data is first transfered to the cache and then written back to main memory. Two accesses are executed on the bus to perform one read operation. One intuitive idea might be to insert a buffer inside the conventional DRAM macro so data becomes available earlier. One important factor is that the DRAM is embedded. Insertion of extra latches for each DRAM bank will require substantial chip area as these latches require many transistors. Each independent memory bank seen from the processor can have several sub banks. In this case the sense amplifiers have to drive both the extra latch and data to the cache and therefore will have to be more powerful. By centralizing these latches fewer are needed at the cost of extra (on-chip) bus traffic. This enables buffering of write-backs for subsequent accesses which will improve performance. In a system with non-embedded DRAM, the situation is different as bus traffic is slow, limited and energy expensive.

In the delayed write-back scheme data in the cache has to be written back to make space before a read operation can start. If the data to be written and the data to be read belong to different DRAM banks, the two operations can be done in parallel. The advantage with this scheme is that data is only written back to DRAM once. With the immediate write-back scheme, data might be written back to DRAM twice. First, the data is written back right after reading. Then, if the data is modified, it is written a second time when it is thrown out of the cache.

As an example to illustrate the difference between the two write-back schemes, a simple program is executed with the two different write-back schemes (see figure 4). The program is executed on hardware with the following properties: There is only one DRAM bank, and a read or write operation to DRAM takes 3 clock cycles. Each read cycle is illustrated with *L[address]* and each write cycle with *S[address]*. The read operations are destructive, the content of the loaded addresses are erased in DRAM. The data cache has two cache lines and each line can store one word. The cache has a 1 cycle latency and is *not* write-through. The cache is initialized with unmodified cache lines for addresses x and y. The example shows the difference in access patterns (number refers to lines in figure 4):

1. Address 0 is loaded into the cache and address x is thrown out. This line is clean and there is no need for a write-back.
2. Address 1 is about to be loaded into cache, but the bus is busy with the write-back from the previous instruction and this has to finish before loading can start.
3. The result of an addition is written in address 0. Since this address is in the cache, it is a cache hit, and no activity on the bus is needed. The write-back from the previous instruction starts as well.
4. Data from address 2 is loaded into the cache. However, before any DRAM accesses can start, the write-back from instruction in line 2 has to finish (2 clock cycles). Then, the data in the cache has to be written back since it has become dirty (address 0 and address 2 map to the same cache line). Finally the load operation can start.
5. In the delayed write-back scheme data in cache is always treated as dirty. Therefore before loading data for address 0, data in the cache has to be written back.
6. Same as line 5.
7. Cache hit, no activity on the system bus.
8. Data in the cache has to be written back before loading can start.

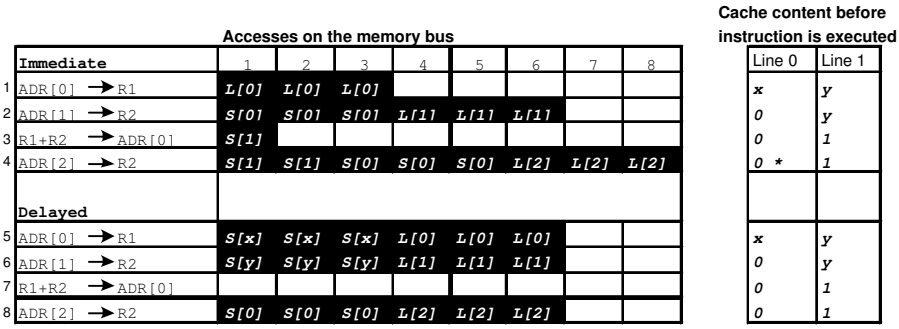


Fig. 4. Example of execution with the two different write-back schemes. DRAM bus activity is shown with *L[address]* for reading and *S[address]* for writing. The addresses that are kept in the cache(i.e. the state of the cache) before the instruction is executed are shown to the right. The cache is initialized with addresses *X* and *Y* which are not address *0, 1* or *2*. Addresses *0, 2* map into the first cache line, while address *1* maps into the second cache line. * indicates a modified cache line. In all cases the delayed write-back scheme has to write data back to DRAM on replacement, so cache lines can always be considered to be dirty.

In the delayed write-back scheme data for address 0 is only written once, while in the immediate write-back scheme it is written twice for address 0. A load instruction with the delayed write-back scheme takes 3 or 6 cycles; if the data in the cache line that is replaced is on the same memory bank as the data that is loaded, it takes 6 cycles. Otherwise, when the data in the cache line and the data to be read are on different banks, it takes 3 cycles. Load instructions with the immediate write-back scheme takes 3 to 9 cycles. The first 3 clock cycles might be needed to wait for the bus to become available due to earlier background writing operations. 3 additional cycles are needed when the data in the cache line is dirty and have to be written back to the same memory bank. Finally, 3 cycles are always needed for reading data.

Which of the two schemes has the highest performance? With the immediate write-back scheme the result from a load operation is available after only 3 cycles when the cache line is clean. The strength of the delayed write-back scheme is the reduced traffic on the memory bus. In cases where data in the cache is modified, the number of transactions on the bus is reduced to only one.

The advantage of the immediate write-back scheme depends on unmodified cache lines while the advantage of the delayed write-back scheme depends on a modified cache lines. Smaller caches will have a lower ratio of modified cache lines that are replaced because data are swapped out before they are written to, while larger caches will have a higher ratio of modified cache lines. The ratio of modified cache lines that are replaced also depends on the program.

3 Simulation

The purpose of our simulation is to study the performance of different write-back schemes with the destructive-read memory and compare this to conventional DRAM. The simulator is based on SimpleScalar version 3 [1]. SimpleScalar is extended to simulate a configurable number of DRAM

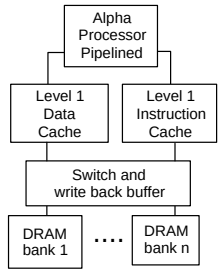


Fig. 5. The simulated computer.

banks and a configurable stand alone write-back buffer in addition to the two write-back schemes and Hwang's original scheme.

A logical sketch of the simulated computer is shown in figure 5. The target is a computer with embedded memory and only one level cache. The processor is simple to save area and power. The configuration for the baseline of the simulations are:

- Cycle-true simulation.
- Alpha processor with a five stage pipeline running at 1 GHz.
- Single issue, no branch prediction buffer, no translation look-aside buffer, in-order execution, single decode, single commit width, single ALU.
- Two independent caches, one instruction cache and one data cache each with 64 bytes cache lines, two ways set associative, and each total cache size of one kbytes. Latency is 1 ns.
- Four independent memory banks with simulation of congestion. Memory bus width is the same as cache line width (64 bytes).
- Latency of DRAM is 6 ns for a read operation. For destructive-read, this is 3 ns for reading and 3 ns for writing. DRAM Refresh is not simulated as it is presumed to have little effect on the result. Total access time for a cache miss includes 1 ns for cache plus access time for DRAM.
- In simulations of Hwang's original scheme, the latency of the memory system is always 3 ns (write-backs are perfectly hidden in the large write-back buffer).
- A write-back buffer is implemented for each memory bank capable of storing one cache line.

SPEC2000 applications were used as benchmark with *lgred* (large reduced input dataset)[13] as the data set. One of the 26 applications found in the *SPEC2000* did not work with the simulator (*vortex* application). In order to reduce computation time experiments that return average values are based on a subset of the applications (*gzip, gcc, crafty, mcf, swim, mgrid* and *equake*). Sampling shows that the subset represents the total average values within +/- 2%.

Four different configurations were simulated:

- *Conventional* represents the conventional DRAM scheme. Access latency is double the latency of destructive-read DRAM, but no write-back is required.
- *Immediate* is the immediate write-back scheme. Data is written back immediately or put in the write-back buffer if enabled.

```
/* Code for read experiment */
for(x=0;x<30000;x++) {
    y=y+data[x];
    z=z+data[x];
}
/* Code for read/write experiment */
for(x=0;x<30000;x++) {
    data[x]=data[x]+y;
    data[x]=data[x]+z;
}
```

Fig. 6. Source code for the initial experiment.

- *Delay* is the delayed write-back scheme. The cache behaves like a normal cache, but the lines are always written back on replacement.
- *No cost* represents an ideal DRAM, combining the speed of destructive-read and the data integrity of conventional DRAM. The intention is to study the performance degradation imposed by the extra write-backs for conserving data.
- *Hwang* represents the original scheme from Hwang.

4 Results

4.1 Initial experiment

An initial experiment was run to verify the predictions regarding the performance of the two write-back schemes. The experiment has two test programs, one that reads data and one that reads and writes data into a data structure as shown in figure 6. To reduce the effect of instruction cache misses, the experiment was run with a very large instruction cache. The data cache was limited (128 bytes) in the same way as in the example. There was only one memory bank with 10 ns latency. The latency was set high so the effect of the memory latency becomes dominant and less influenced by the CPU performance. This configuration does not reflect a real computer, but is used to illustrate the differences between the two write-back schemes. The immediate write-back scheme should suit the read experiment as the second line in the loop can execute while write-back from the first instruction in the loop is executed in the background. The delayed write-back scheme should suit the read/write experiment as the number of write-backs to DRAM is reduced compared to the immediate write-back scheme. The results from the experiment are summarized in figure 7 and are according to predictions.

4.2 Baseline

Simulation of the different write-back schemes for the baseline architecture is shown in figure 8. Average values are shown to the right. Compared to conventional DRAM, the delayed write-back scheme is 13.2% faster, immediate write-back scheme is 13.5% faster, no cost write-back scheme is 14.4% faster and Hwang's original write-back scheme is 12.1% faster. The buffer size of Hwang's original scheme was 25% of the DRAM size. For the simulated applications the size of the buffer

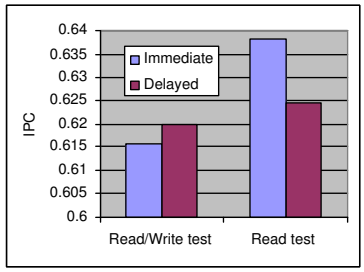


Fig. 7. Results from the initial experiment.

is in the range of 170k-5714kbyte, depending on necessary memory size. The other schemes are simulated with 1kbyte cache. By comparing the no cost scheme with the other two destructive schemes, it is found that only 0.02% and 0.05% of the performance is lost due to write-backs for immediate and delayed write-back schemes respectively.

4.3 Cache size

The IPC for different cache sizes are shown in figure 9. The immediate write-back scheme is slightly better than delayed write-back scheme for small caches. For larger caches they perform more or less equal. In order to understand this advantage, the ratio of the number of accesses to the DRAM subsystems for the two schemes is shown in figure 10. In the delayed write-back scheme, data are not written back immediately. For modified data (by CPU), the total number of accesses is reduced compared to immediate write-back scheme where data are written twice in this case. Larger caches improve the probability of data being modified before written back (replaced). The advantage of the immediate write-back scheme is that data is available earlier in cases where there is a conflict between writing and reading data. Even though the two models are different the performance is similar except for small caches where the immediate write-back scheme is better.

4.4 Latency and number of DRAM banks

Simulation of different DRAM-latencies is shown in figure 11. As latency increases, performance degrades due to conflicts between reading and writing. Increasing the bank count would reduce the probability of congestion. The effect of increasing (or decreasing) the number of DRAM banks is shown in figure 12.

4.5 Write-back Buffer size

The write-back buffer is complementary to the cache and each memory bank has its own small fully associative write-back buffer. They are important for performance of the delayed write-back scheme as shown in figure 13. In this scheme data has to be written back when the cache line is replaced. Without a buffer the processor has to wait for both operations to finish before data becomes available. In the immediate write-back scheme this buffer is less important.

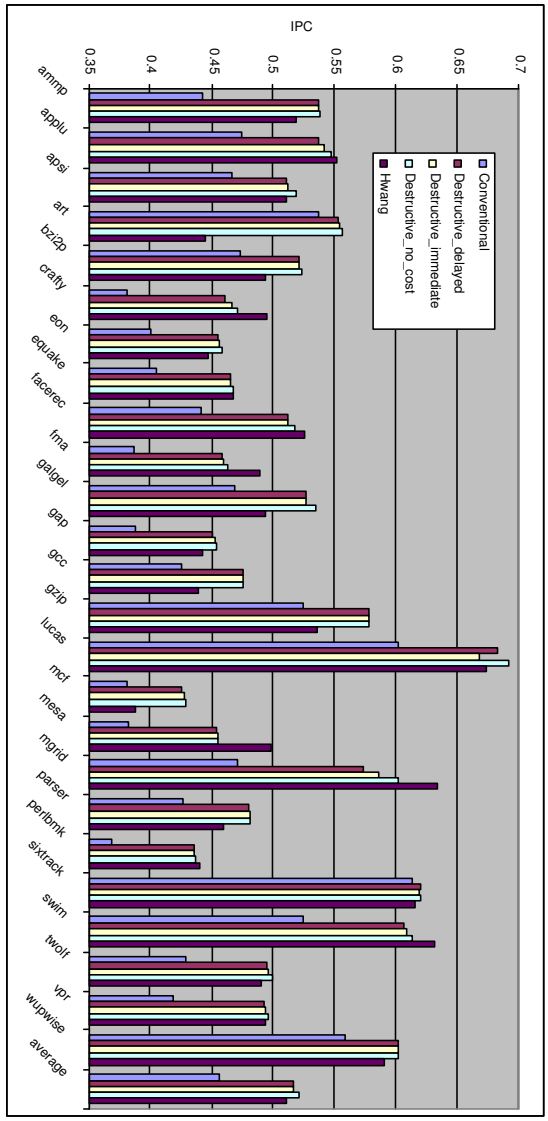


Fig. 8. Performance of baseline configuration for different SPEC2000 benchmarks

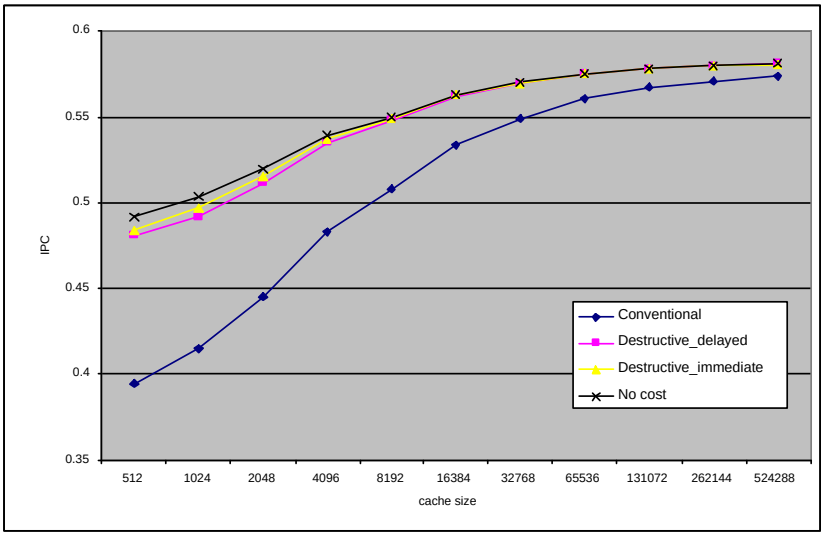


Fig. 9. Average IPC as a function of cache size. Smaller caches favour the immediate scheme. By comparing the write-back schemes with the no cost scheme it can be seen that write backs are better hidden with larger caches.

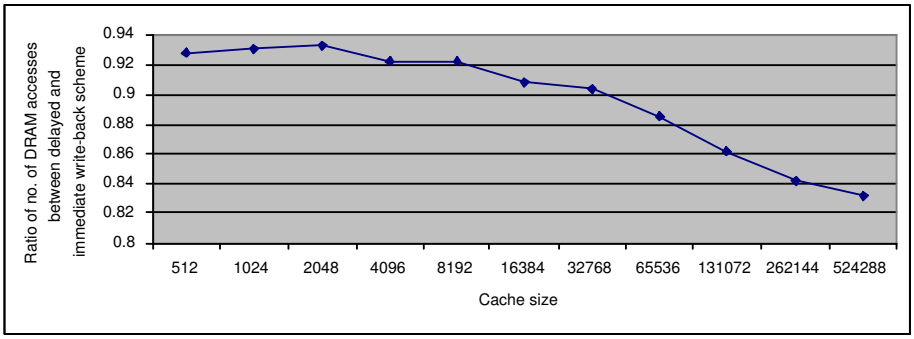


Fig. 10. Comparison of the total number of accesses to DRAM from caches for the two different write-back schemes. For larger caches more data are modified before they are written back to DRAM, and this gives the delayed write-back scheme an advantage for large caches.

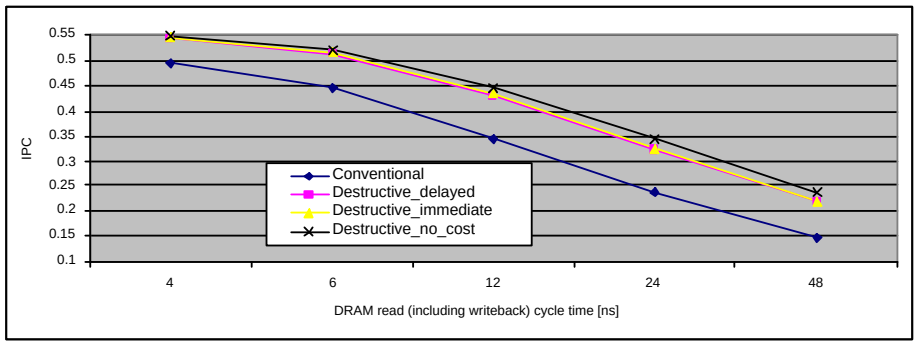


Fig. 11. Average IPC as a function of latency. Write-backs become blocking as latency is increased. The values on the x-axis are non-linear, the first value is increased from 3 to 4 ns to match the cycle time of the CPU.

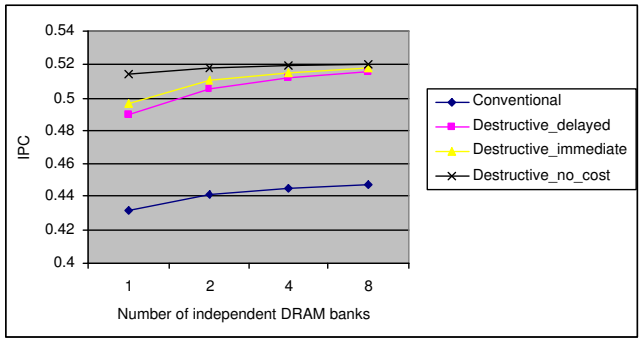


Fig. 12. Average IPC as a function of the number of DRAM banks. For a small number of DRAM banks, write-backs blocks performance.

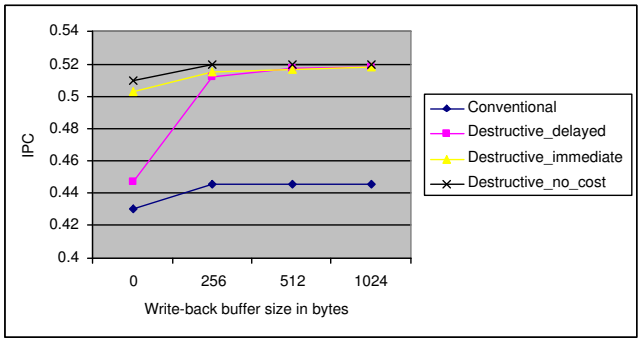


Fig. 13. Average IPC as function of number of buffer size. By turning off the write-back buffer, the delayed write-back model has to wait for the data in the cache to be written back before a new line can be loaded in cases where these two line are mapped to the same memory bank.

5 Discussion

The simulated results support our predictions regarding cache size and write-back schemes. In systems with a relatively large cache, delayed write-back is the preferred scheme due to less traffic on the DRAM buss, while for smaller caches, immediate write-back results in slightly increased performance due to data being available earlier.

The buffer size of Hwang's original scheme was 25% of the DRAM size. For the simulated applications this buffer will be in the range of 170k-5714kbyte, and this is outperformed with the new schemes with a faster 1kbyte cache.

The simulations show that the process of writing back data is hidden quite well (less than 0.05% of IPC is lost due to write-backs in the baseline scheme), this is true for both write-back schemes. This is shown to be connected to the number of independent memory banks and the write-back buffers for each memory bank. More banks reduces the probability of congestion. Write-back buffers change the delayed write-back scheme to first read data before the existing cache line is written, and therefore data becomes available earlier and performance is increased. For longer memory latencies, congestion is more likely as a write operation has to finish before a read operation can start.

By comparing the performance of different configurations it can be seen that by replacing conventional DRAM with destructive-read DRAM, cache sizes can be reduced without degrading performance. The savings in cache size is shown in figure 14. Typically for the configurations that are simulated, the cache size can be reduced with a factor four. Reducing cache size has a positive impact on power consumption and less chip area is needed.

The bus between DRAM and cache has to run at double the speed with destructive-read DRAM compared to conventional DRAM. Since the bus is on-chip this should result in only slightly higher power consumption. DRAM contributes to just a small portion to the total power consumption in most computers (not including off-chip buses).

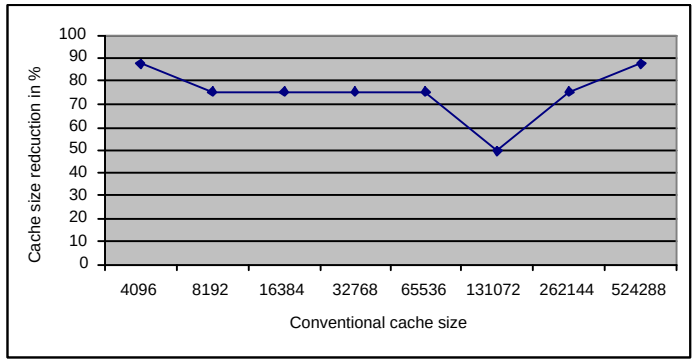


Fig. 14. Cache size saving in % by using destructive-read DRAM compared to conventional DRAM for equal or better performance. This figure is based on figure 9.

6 Conclusion

Two different schemes for write-back to destructive-read DRAM have been analyzed and simulated. In configuration with no write-back buffer or small caches (about 512 bytes) the immediate write-back scheme outperforms the delayed write-back scheme. For configurations with larger caches and longer cache lines the performance of the two write-back schemes are almost identical. Both schemes hide write-back operations well (less than 0.05% of IPC is lost).

Configuration with destructive-read DRAM performs better than conventional DRAM. In terms of IPC, performance is increased by 13.5%.

In general, systems with destructive-read DRAM can perform equal or better than systems with conventional DRAM and four times larger caches. Compared to Hwang's configuration the cache size can be reduced by several magnitudes without degrading performance.

The reduction in cache size reduces both dynamic and static power consumption as well as system size. In PIM design, chip area made available by reducing cache size can be used to increase the number of processors or memory size.

Bibliography

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. IEEE Computer, Volume 35, Issue 2, February 2002.

[2] J. Draper, C. W. Kang, I. Kim, G. Daglikoca, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, and C. Chen. The architecture of the DIVA processing-in-memory chip. Proc. 16th ACM Int'l Conf. Supercomputing, p:14-25, 2002.

[3] H. Dybdahl, P.G. Kjeldsberg, and M. Grannæs. Destructive-read in embedded dram, impact on power consumption. *Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures, Tentative Publication Date: December 2005.*

[4] D. G. Elliott, W. Martin Snelgrove, and Michael Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In Custom Integrated Circuits Conference, Boston, MA, pages:30.6.1-30.6.4, May 1992.

[5] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory; the Terasys massively parallel PIM array. IEEE Computer, pages:23-31, April 1995.

[6] Linley Gwennap. Embedded dram use rises. Nikkei Electronics Asia, June, June 2003.

[7] C-L. Hwang, T. Kirihata, and M Wordernan et.al. A 2.9ns random access cycle embedded DRAM with a destructive-read architecture. VLSI Circuits, Digest of Technical Papers, IEEE Symposium on, p:174-175, 2002.

[8] Embedded DRAM comparison charts, 2003. http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Embedded_DRAM.

[9] International technology roadmap for semiconductors, 2003. http://public.itrs.net/.

[10] B.L. Ji, S. Munetoh, C-L. Hwang, M. Wordeman, and T. Kirihata. Destructive-read random access memory system buffered with destructive-read memory cache for SoC applications. VLSI Circuits, Digest of Technical Papers. Symposium on, pages:85 - 88, June 2003.

[11] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Patnaik, and J. Torellas. FlexRAM: Towards an advanced intelligent memory system. Int. Conference on Computer Design, 1999.

[12] G. Kirsch. Active memory: Micron's Yukon. Parallel and Distributed Processing Symposium, Proceedings. International, number of pages:11, April 2003.

[13] AJ KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[14] P.M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and logic chip for massively parallel systems. IEEE, Advanced Research in VLSI, Proceedings. Sixteenth Conference on, pages:4-16, March 1995.

[15] K. Mai, T. Paaske, N. Jayasena, W R. Ho, Dally, and M. Horowitz. Smart Memories: A modular reconfigurable architecture. ISCA, June 2000.

[16] Y. Nunomura, T. Shimizu, and O. Tomisawa. M32R/D-integrating DRAM and microprocessor. Micro, IEEE, Volume: 17, Issue: 6, pages:40-48, November 1997.

[17] M. Oskin, F.T. Chong, and T. Sherwood. Active Pages: A model of computation for intelligent memory. International Symposium on Computer Architecture, Barcelona, Spain, 1998.

[18] D. Patterson, T. Anderson, and K. Yelick. A case for intelligent DRAM: IRAM. Presented at Hot Chips VIII, Palo Alto CA, pages:18-20, August 1996.

[19] L.V. Yerosheva, S.K. Kuntz, J.B. Brockman, and P.M. Kogge. A microserver view of HTMT. Parallel and Distributed Processing Symposium, Proceedings 15th International, number of pages:10, April 2001.

Destructive-Read in Embedded DRAM, Impact on Power Consumption

Haakon Dybdahl and Per Gunnar Kjeldsberg and Marius Grannæs and Lasse Natvig

Journal of Embedded Computing, Vol 2 Issue 2, 2006.
IOS Press.
Pages 1–12.

Destructive-Read in Embedded DRAM, Impact on Power Consumption

Haakon Dybdahl[1] and Per Gunnar Kjeldsberg[2] and Marius Grannæs[1] and Lasse Natvig[1]

[1] Dept. of Computer and Information Science,Norwegian University of Science and Technology,N-7491 Trondheim, Norway, [dybdahl, Marius.Grannas, lasse]@idi.ntnu.no

[2] Dept. of Electronics and Telecommunications , Per.Gunnar.Kjeldsberg@iet.ntnu.no

Abstract. This paper explores power consumption for destructive-read embedded DRAM. Destructive-read DRAM is based on conventional DRAM design, but with sense amplifiers optimized for lower latency. This speed increase is achieved by not conserving the content of the DRAM cell after a read operation. Random access time to DRAM was reduced from 6 ns to 3 ns in a prototype made by Hwang et. al. A write-back buffer was used to conserve data. We have proposed a new scheme for write-back using the usually smaller cache instead of a large additional write-back buffer. Write-back is performed whenever a cache line is replaced. This increases bus and DRAM bank activity compared to a conventional architecture which again increases power consumption. On the other hand computational performance is improved through faster DRAM accesses. Simulation of a CPU, DRAM and a 2 kbytes cache show that the power consumption increased by 3% while the performance increased by 14% for the applications in the SPEC2000 benchmark. With a 16 kbytes cache the power consumption increased by 0.5% while performance increased by 4.5%.
Keywords: *Embedded DRAM, power estimation, Simplescalar, destructive-read memory, processing in memory*

1 Introduction

Main memory and *central processing units* (CPUs) have both become increasingly powerful during the last 30 years, but their progress have taken different directions. CPUs have got faster clock cycles and more computations per clock cycle while main memory can store more data. Today there is a magnitude of difference in cycle time between main memory and CPUs, often referred to as the *processor memory performance gap*. Reducing the effect of this gap has been a main research objective for decades. This has resulted in various mechanisms such as caches, out-of-order scheduling, prefetchers and simultaneous multithreading. These mechanisms consume a substantial amount of power and are thus a challenge when designing battery driven equipment, but also for design of high performance CPUs. For systems with multiple cores on a single chip

the overall power usage limits the performance and/or the number of cores that can be integrated.

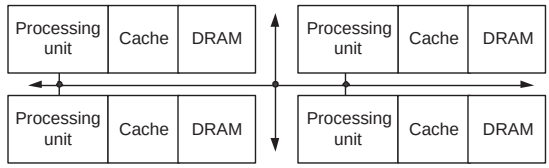


Fig. 1. Multiple cores and memory in a single chip.

The target architecture is shown in Figure 1. Each processor is relatively simple and has its own cache, DRAM banks and a communication channel to the other processors. Hwang et al. [10] made a prototype which enables lower latencies in DRAM by modifying the sense amplifiers to omit write-backs. Reading a memory cell thus destroys its content and the only copy now exists in a write-back buffer. To ensure that later write-backs of data to DRAM do not inflict a performance penalty, this buffer must at least be as large as the DRAM bank size. We have proposed a new scheme for write-back utilizing the usually smaller cache instead of this large additional write-back buffer [5]. In our scheme the size of the cache is not dependent on DRAM bank size. Simulations of a system with 2 kbytes cache show that speed is improved by 14% with our approach compared to conventional DRAM. Due to the increased speed of the DRAM, the cache size can be reduced by a factor four in a system with destructive-read DRAM compared to conventional DRAM without degrading performance. Our previous work has not considered energy consumption, and this is the topic for this work.

The concept of destructive-read DRAM is explained in Section 2. The models used for power estimation is described in Section 3. Section 4 describes the simulations that are run and Section 5 describes the results. Section 6 is discussion and Section 7 conclusions followed by references.

2 Embedded Destructive-Read DRAM

2.1 Embedded Memory

DRAM is cheap, dense and consumes little power compared to other technologies; therefore it is the main choice for main memory. DRAM is traditionally found on separate chips and connected to the CPU through off-chip buses. These off-chip buses introduce capacitance which again increases power consumption and latency. The number of pins available on the CPU packages introduces a practical limit for the bus width. By merging main memory and CPUs, this external bus is eliminated. Main memory has a much higher internal bandwidth

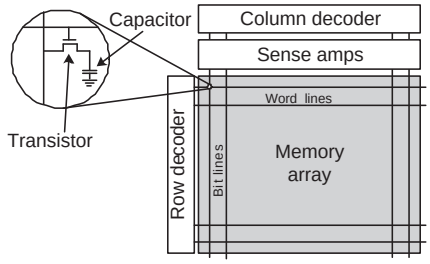


Fig. 2. Inside a DRAM bank.

than what is available through the off-chip buses. By utilizing embedded DRAM the internal bandwidth of main memory is available to the CPU. However, the process of merging main memory and central processing unit is not trivial as DRAM uses capacitors to store data (see Figure 2). DRAM chips are optimized for these analog circuits. Logic circuits on the other hand are optimized for speed and power distribution. As a consequence, embedded DRAM is not as dense (bits per area) as conventional DRAM chips. The actual density depends on the technology used [13]. The most sophisticated technology combines processes from DRAM manufacturing and CMOS logic chip manufacturing while the simplest generates the cells in pure CMOS. An additional advantage of embedded DRAM, compared to the normally faster embedded SRAM, is that the standby leakage power is much smaller. This factor becomes increasingly important as the technology continues to shrink below 180nm [12].

2.2 Related Work

Several projects have done research into merging processors and memory ([28], [4,6,7,15,17,20,22,23,30]). Most of these projects assume a conventional DRAM design. The C*RAM project [6] is an exception where small processing elements are integrated into the sense amplifiers, utilizing the parallelism available at that level. A scaled down prototype was made. It was a SIMD computer with single bit processors. This architecture is only suitable for problems with high data locality because of limited communication between the single bit processing elements.

Many other projects use SIMD architectures to utilize the extra bandwidth: e.g. the IRAM project [26], Yukon [17], Terasys [7] and Execube [19]. The Mitsubishi M32R/D chip [22] and Saulsbury et. al [28] use the bandwidth to increase the number of bits in the data bus between main memory and cache. FPGA and independent processors have also been proposed to utilize the bandwidth ([4, 20]). During the last few years, embedded DRAM has become more common, and chips are in mass production with this type of memory integrated with graphics or network processors such as Sony's Playstation 2, Xbox 360, EZchip's

Type	Access Time
PC133 SDRAM	71.4nS
DDR266 DRAM	58.8nS
RL DRAM	25.0nS
130nm embedded DRAM	12.0nS
90 nm embedded DRAM Dense	8.0nS
90 nm embedded DRAM Fast	4.5nS
DDR SRAM	3.3nS
Embedded SRAM	2.0nS

Table 1. Random cycle time for various memories [11].

NP-1c [8] network processor and Nintendo's GameCube. Embedded DRAM is becoming commercially available at different speeds. Table 1 contains random access latency time for various memory technologies. By reducing the size of each memory bank, a smaller unit is activated during an access. As will be explained in Section III, the bank size has a large influence on the power consumption.

A comprehensive study of different aspects of memory and data intensive design can be found in [3] and [24].

2.3 Destructive-Read DRAM

Destructive-read DRAM [10] is a modified version of conventional DRAM. A memory bank with conventional DRAM is shown in Figure 2. A charged capacitor (normally) represents the logic high value, while an uncharged capacitor represents the logic low value. The row decoder is the first component activated in a read access. It enables one *word line* and causes all transistors in that row to be activated. These transistors connect the capacitors in the memory array to the *sense amplifiers* through *bit lines*. The bus out of a DRAM macro often has fewer lines than the number of bit lines inside the macro. The column decoder controls which subset of bit lines that are read or written. The sense amplifiers work in three phases as shown in Figure 3a. In the first phase the charge (or lack of charge) from the capacitor drives the sense amplifier into a logic high (or low), in both cases leaving the capacitor discharged. In the second phase the logic state is locked. In Figure 4a the locking works as a buffer. From this buffer the data is both sent to the processor and written back to memory. In the final phase the bit lines are precharged so they are ready for the next access. Destructive-read DRAM memory works differently. The read operation of conventional DRAM (see Figure 3a) is split into two cycles (see Figure 3b and c). The destructive-read DRAM does not lock the data after reading (as shown in Figure 4b). Instead the data are sent directly out of the chip, in this case to cache memory. Since data is not sent back to memory, the capacitor is left discharged and data is destroyed. Data is conserved by writing it back to DRAM later as shown in 3c. However, write-back is not performed immediately after

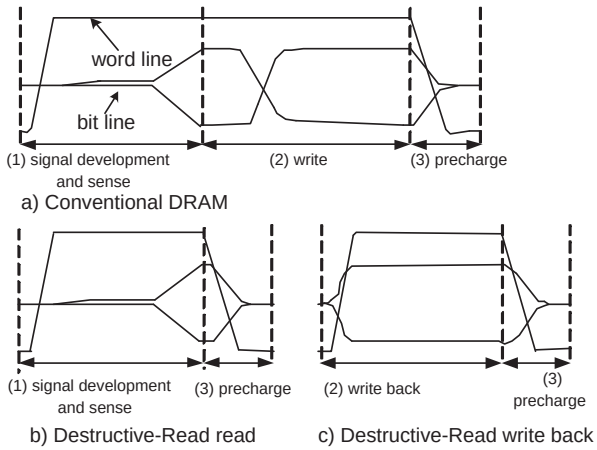


Fig. 3. Conceptual waveform diagrams of conventional DRAM architecture vs. destructive-read [10].

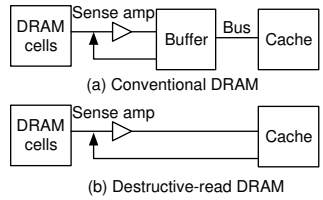


Fig. 4. Conceptual view of DRAM.

the read, this in contrast to conventional DRAM where read and write-back are a single operation.

2.4 Write-backs

Hwang et al. made a prototype where random access time to DRAM was reduced from 6 ns (conventional read) to 3 ns (destructive-read). The prototype had four independent memory banks and a large write back buffer (WBB) that was the same size as one memory bank. The WBB was made out of SRAM. The purpose of the WBB was to hide write-backs, not to reduce latency. The WBB could write-back to several banks simultaneously and required significant chip area. Later a new scheme was made where the WBB was replaced with destructive-read DRAM [14]. The designs guaranteed that write-backs never conflicted with read operations. We have proposed new schemes that remove the large write-back

buffer and increase performance by utilizing the cache [5]. The data is first read from DRAM and into the cache. At this time data are not stored in DRAM, only in the cache. Data are written back to DRAM when the cache line is replaced. In a conventional scheme data is only written back if data is modified, while in this scheme data is always written back. Therefore this scheme can be compared to a cache that always has dirty cache lines. Write-backs are partially hidden by using several memory banks so data can be read from one bank while writing to a different bank. However, in some cases the read operation and the write back access the same memory bank, causing a delay. Figure 5 shows the number of instructions per clock cycle (IPC) for a system with conventional DRAM and our destructive-read DRAM scheme, both with a 2 and 16 kbytes cache for the applications in SPEC2000 benchmark suite.

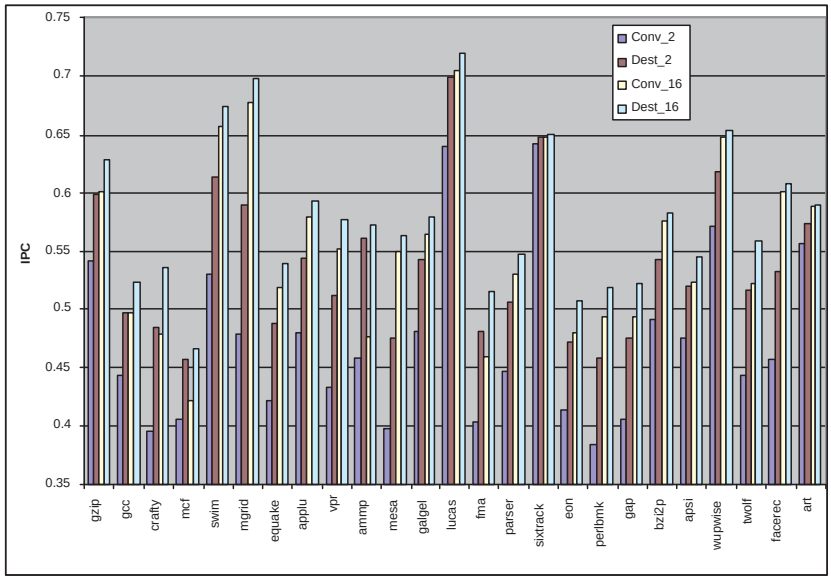


Fig. 5. IPC for the applications in the SPEC2000 benchmark suite for conventional (conv) and destructive-read (dest) DRAM with 2 and 16 kbytes cache configuration. The average IPC is 13.7% higher for *dest_2* compared to *conv_2*, and 4.5% higher for *dest_16* compared to *conv_16*.

3 Model for Power Consumption

Power consumption in computers can be divided into *static* and *dynamic* power consumption. Dynamic power consumption for a CMOS chip is shown in Equa-

tion 1.

$$P_{dynamic} = C_{switched} * V_{dd}^2 * f_{clk} \quad (1)$$

V_{dd} is supply voltage, f_{clk} is frequency and $C_{switched}$ is the total effective switched capacitance, i.e. is the average capacitance of the transistors and communication lines that are switch in each clock cycle. In synchronous designs such as a microprocessor the clock distribution adds a significant value to $C_{switched}$.

As chips get denser, leakage power increases, and transistors consume more power without switching [12]. This is called static power consumption. Caches have higher leakage currents per stored bit compared to DRAM because of higher transistor count, and will therefore consume more power in denser technologies compared to DRAM.

3.1 Voltage and Frequency

The following relationship between frequency and power has been described by Khellah and Elmasry [16]:

$$T_d \approx \frac{C_L * V_{dd}}{\kappa * (V_{dd} - V_{th})^{\bar{\alpha}}} \quad (2)$$

T_d is the delay of a CMOS gate, where C_L is the load capacitance, κ is a factor that depends on the process and gate size, $\bar{\alpha}$ takes a value between one and two, V_{th} is the threshold voltage and V_{dd} is the supply voltage. Even though the formula is not complicated, a lot of complexity is hidden in the $\bar{\alpha}$, V_{th}, C_L and κ factors. However, what can be read from this formula is that for a given technology and circuit, the maximum operating frequency is a function of supply voltage. This fact is used for power saving in commercial computer systems in a technique called dynamic voltage-frequency scaling. The technique reduces both frequency and voltage for the system when maximum computational performance is not needed.

3.2 Power model of DRAM with bus

The data flow of accessing DRAM from cache is shown in Figure 6. The address and control signals are sent to the DRAM bank. Data are read out of the bank and written back along the route shown as a thick gray line. Equation 3 shows a model for the energy consumption (E_{MEM}) for these components for the execution of a complete application, e.g. one of the benchmarks from Figure 5.

$$E_{MEM} = N_{ACC} * (E_{DRAM} + E_{SWITCH} + E_{BUS}) \quad (3)$$

N_{ACC} is the number of DRAM memory accesses, and for a single access: E_{DRAM} is energy consumed in the DRAM banks, E_{BUS} is energy consumed by the bus and E_{SWITCH} is energy consumed by the switch and the write-back buffer.

Yong-Ha Park et. al. [25] have made a model for the dynamic power consumption of embedded DRAM at an abstraction level that matches the simulation

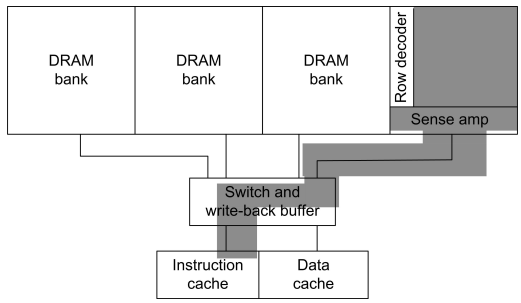


Fig. 6. Data communication when accessing a memory bank.

model that we are using for the processor. We have based Equation 4 on this model.

$$E_{DRAM} = N_{ROW} * E_{ROW} + N_{COL} * E_{COL} \tag{4}$$

N_{ROW} is the number of row activations, N_{COL} is the number of column activations, E_{ROW} is energy consumed by activation of a row in the embedded DRAM and E_{COL} is energy consumed by activation of a column in the embedded DRAM. All these variables are for a single memory access. Burst transfer is not used in this work because communication is on-chip and the bus width is increased instead. In this way the column decoder is not needed as one word line of the memory array is read or written simultaneously. Refresh of DRAM is not different for the two DRAM models and is therefore omitted for simplicity. The result is that N_{COL} is one and N_{ROW} is the number of data bits which is 512 in our model. The consequence is that E_{DRAM} is equal for all memory accesses because each access activates one row and 512 columns. In order to quantify E_{DRAM} we use a DRAM macro made by Morishita et.al [21]. The power consumption is 0.260 W at 250 MHz (in 130 nm technology) for accessing a 16 Mb bank. This is approx. 1 nJ for one access with 128 bits, and we conservatively multiply this with 4 for 512 bits width resulting in $E_{DRAM} = 4nJ$ for our DRAM bank. Power consumption depends on DRAM macro size as this impacts wire lengths, the number of cells activated in parallel, etc. Smaller macros consume less power while larger macros consume more power per access. Research on interconnection is a large topic and several techniques exists (see for example [27]). Ron Ho et.al. [9] have made efforts to quantify energy consumption for buses, and found that a wire of length 10 mm in 180 nm with 1.8 volt require ¡ 1 pJ per bit for techniques with voltage swing reduction and ¡ 10 pJ for a simple bus wire. We conservatively use the simple wire and 10 mm bus (10 pJ). This result in $E_{BUS} = 5.4pJ$ for 544 bus lines (512 data + 32 address).

The power consumption for the write-back buffer and switch was modeled as a 2 kB cache with four banks and 64 bytes block size in CACTI [29] (in 180 nm technology). It consumes 0.31 nJ per bank per access. We conservatively use $E_{SWITCH} = 1nJ$.

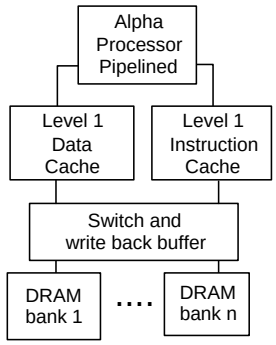


Fig. 7. The simulated single chip computer.

This results in $E_{MEM} = 10.5nJ * N_{ACC}$. The same energy model is applied to both destructive-read DRAM and conventional DRAM, and used for both read and write operation. This is acceptable as our main goal is to compare the two techniques, not necessarily to have exact estimates. There are fewer operations performed in the destructive-read DRAM since no write-back is performed during each read. Static power consumption is not included in this power model as it is not different for conventional and destructive-read DRAM. Using destructive-read DRAM results in shorter execution time and the static power consumption will be slightly lower. If in error, the power consumption penalty of using the destructive-read DRAM will therefore be overestimated.

4 Simulations

The purpose of our simulations is to study the energy consumption and performance of the destructive-read DRAM and compare this to conventional DRAM. For simplicity, only one node of the parallel architecture is simulated. The simulator is based on SimpleScalar version 3 [1]. It is extended with Wattch [2] and HotLeakage [31]. We have further extended it to simulate a configurable number of DRAM banks with congestion and a configurable stand alone write-back buffer. A logical sketch of the simulated computer is shown in Figure 7.

Wattch with *HotLeakage* contain parameters for power consumption for different technologies and the simulation model is fully configurable. However, they do not contain values for DRAM and memory buses to DRAM. For this, the estimate of 10.5 nJ per access from the previous section is used. The configurations for the baseline of the simulations are cycle-true simulation with a five stage Alpha processor at 180 nm technology. Processor properties are single issue, static branch prediction, no translation look-aside buffer, in-order execution, single decode, single commit and single ALU. There are two independent caches, one instruction cache and one data cache each with 64 bytes cache lines, two ways

set associative, and cache size of one kbytes each. Latency is one clock cycle. Eight independent memory banks are used for simulation of congestion. Memory bus width is the same as cache line width (64 bytes). Latency of conventional DRAM is six clock cycles for a read operation. For destructive-read, this is three clock cycles for reading and three clock cycles for writing. DRAM refresh is not simulated as it is presumed to have little and similar effects on the result for both configurations. Total access time for a cache miss includes one clock cycle for cache plus access time for DRAM. A write-back buffer is implemented for each memory bank capable of storing one cache line.

SPEC2000 applications were used as benchmark with *lgred* (large reduced input dataset) [18] as the data set. One of the 26 applications found in the *SPEC2000* did not work with the simulator (*vortex* application).

5 Results

The energy consumption for the various SPEC2000 benchmarks is shown in Figure 8. The total energy consumption level shows little difference between the two models. However, for the destructive-read DRAM model, a larger part of the energy is consumed in the DRAM. On average the energy consumption is increased with 0.46% for destructive-read DRAM compared to conventional DRAM. As will be shown later, this is compensated by a much larger increase in performance.

Details of the energy consumption components for *gcc* is shown in Figure 9. Since the destructive-read DRAM model results in less computational time, less energy is spent on clock distribution and other active waiting components. On the other hand more energy is spent on DRAM components.

The average number of DRAM accesses per clock cycle is shown in Figure 10. The number of accesses is more than doubled for some applications. This is because more instructions are executed per clock cycle due to lower memory latency. Other applications show little increase, as more data is modified by the processor and has to be written back in both schemes.

Three of the applications from the SPEC2000 benchmark were selected for further study: *art, ammp* and *twolf*. *art* was chosen because of the small energy consumption in DRAM, *ammp* was chosen due to high amount of DRAM energy usage and *twolf* was chosen as an average application.

Performance and energy consumption for *twolf* as function of cache size is shown in Figure 11c). Optimizing for low execution time gives larger cache sizes and optimizing for low energy consumption results in cache size of 4 kbytes. The destructive-read architecture consumes more energy than the conventional model for this application. The difference is largest for small caches. This is due to reduced DRAM traffic for larger caches caused by lower miss-ratio in the cache. Performance is better for destructive-read DRAM, especially for small caches. The product of execution time and energy consumption is shown in Figure 12c). A cache of size 16 kbytes is the optimal for minimizing (linearly) both energy consumption and execution time.

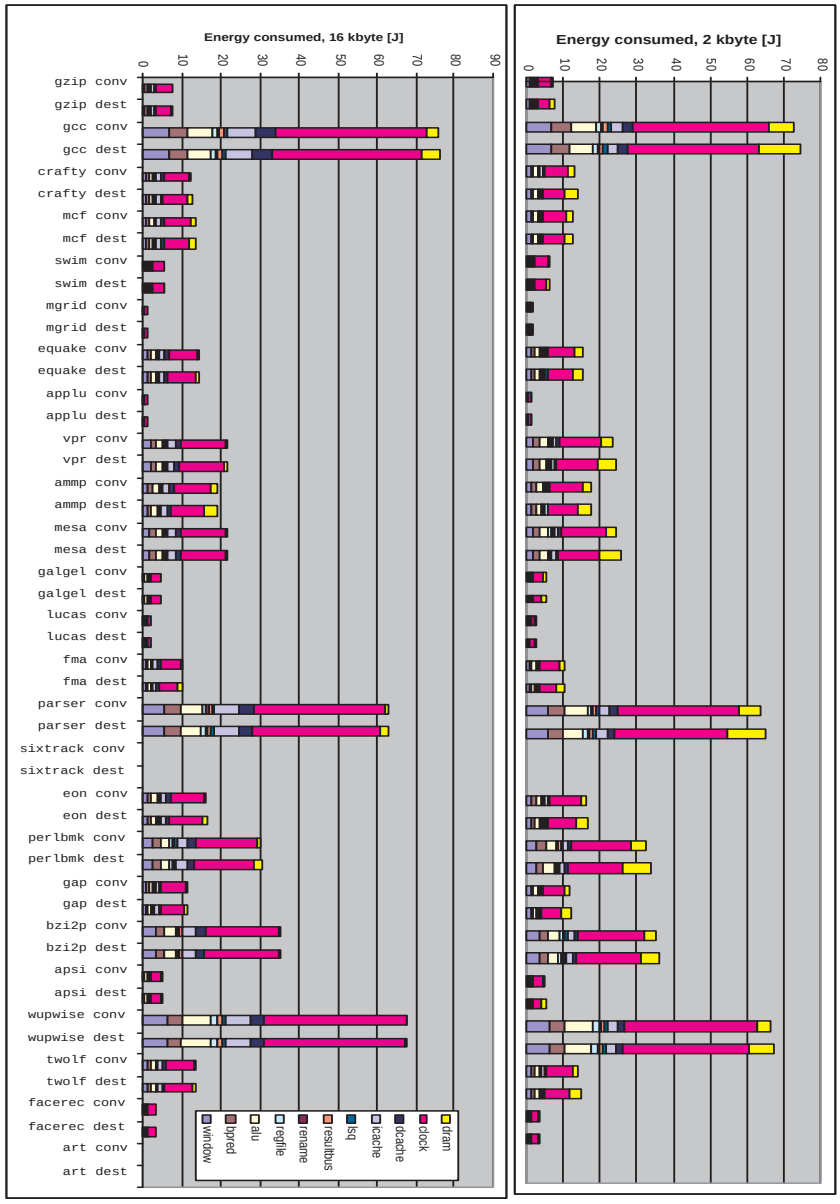


Fig. 8. Energy consumption for various applications in the SPEC2000 suite for conventional DRAM (conv) and destructive-read DRAM (dest). Total cache size is 16 kbytes to the left and 2 kbytes to the right. Average increase in power consumption from conventional to destructive are 0.5% and 3% for 16 kbytes and 2 kbytes caches respectively. The graphs of *sixtrack* and *art* are invisible due to short execution time.

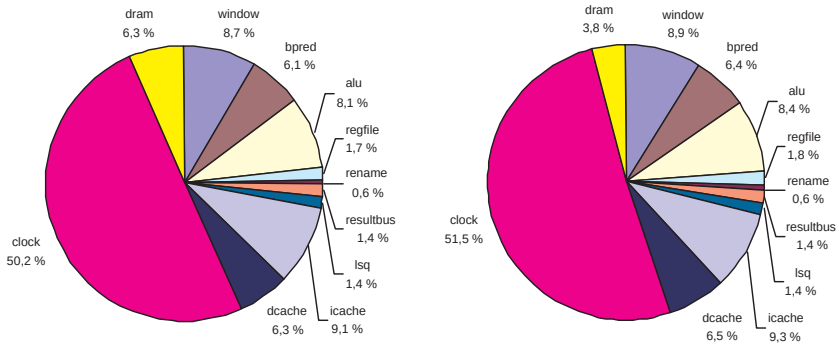


Fig. 9. Energy consumption for *gcc* for destructive-read DRAM to the left and conventional DRAM to the right. Cache size is 16 kbytes.

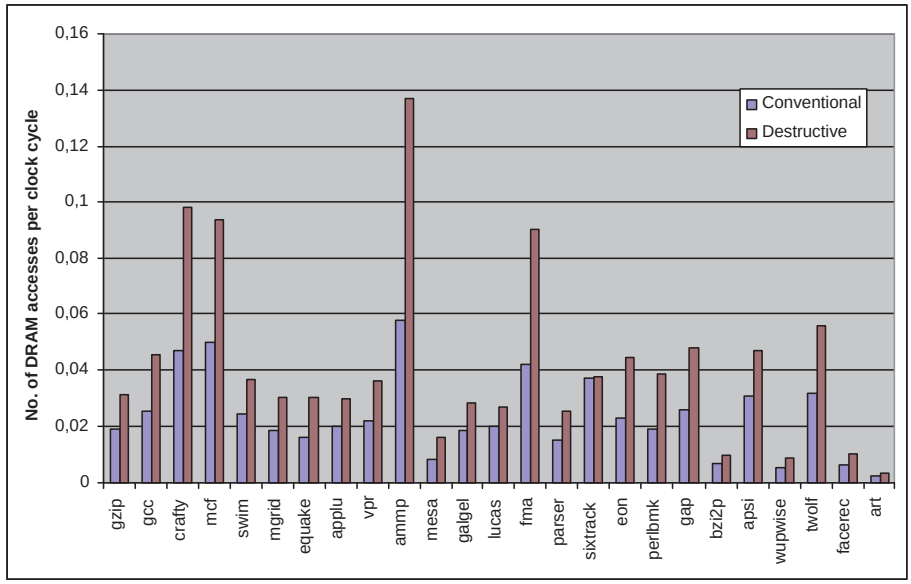


Fig. 10. Number of DRAM accesses per clock cycle for 16 kbytes cache. Due to increased IPC the number of DRAM accesses can be more than doubled for each clock cycle.

For the *art* application the energy consumption and execution time are shown in Figure 11b). For small caches there is a difference while for caches larger than 4 kbytes there is little difference. The product of execution time and energy consumption is shown in Figure 12b). Caches of 4 kbytes is the optimal size for minimizing (linearly) both energy consumption and execution time.

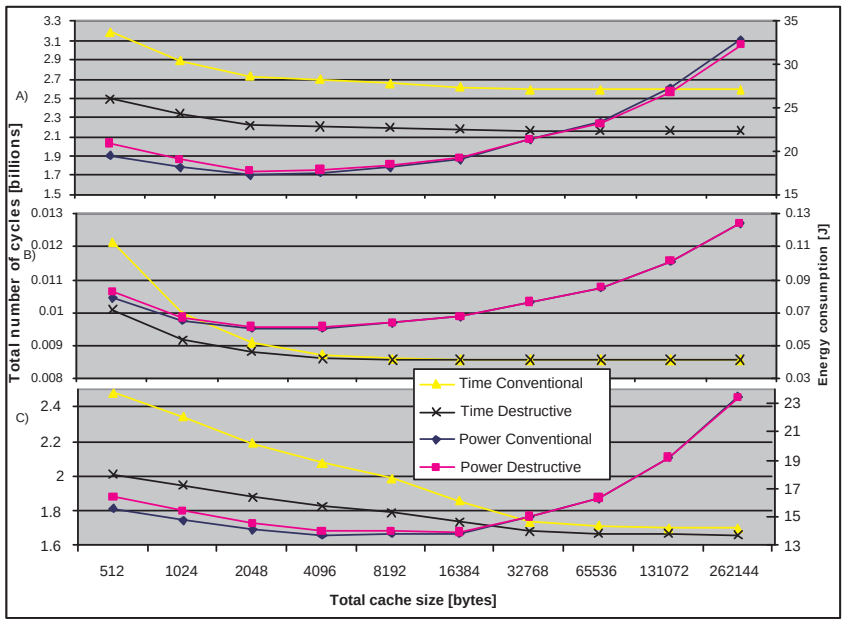


Fig. 11. Number of clock cycles and energy consumption for for a) *Ammp.*, b) *Art* and c) *Twolf.*

A study of performance and energy consumption for *ammp* application as a function of cache size is shown in Figure 11a). For larger caches the destructive-read model requires less energy than the conventional model. This is due to shorter execution time. The product of execution time and energy consumption is shown in Figure 12a). Caches of 2 kbytes is the optimal size for minimizing (linearly) both energy consumption and execution time.

6 Discussion

It is assumed that $E_{MEM} = 10.5nJ$ is consumed for each access to the DRAM subsystem. This assumption influences the power consumption and not the computational speed (IPC). The impact of this value depends on factors such as cache miss ratio and the power consumption of the processor. We have assumed

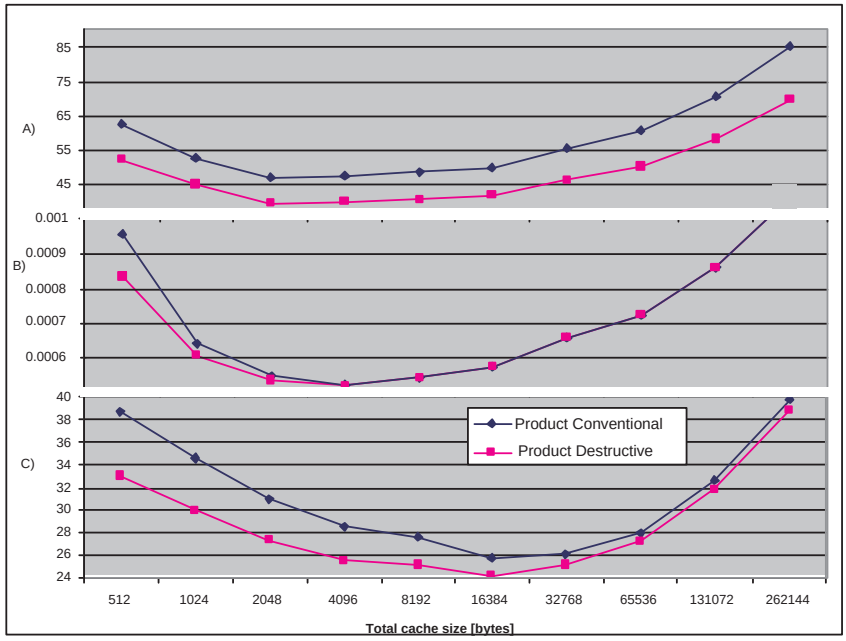


Fig. 12. Product of execution time and energy consumption for a) *Ammp.*, b) *Art* and c) *Twolf*

that energy consumption is proportional with the number of DRAM accesses plus the energy consumption in processor and cache. For the *twolf* application with 16 kbytes cache the power consumption is shown in Equations 5 and 6. The values are taken from simulations. By looking at the components in Equation 5 and comparing this to Equation 6 it can be seen that a system with conventional DRAM uses more energy in the processor and less in DRAM memory compared to destructive-read DRAM, and vice versa. The number of DRAM accesses is increased from 58 million to 97 million with the destructive-read DRAM configuration, an increase of 70%. The processor itself consumes 2% less energy since it executes the task in less time. With higher leakage currents in denser technologies this difference will be even more significant.

$$E_{conv} = 13.2J + 58 * 10^6 * E_{MEM} \quad (5)$$

$$E_{dest} = 12.9J + 97 * 10^6 * E_{MEM} \quad (6)$$

For the simulated architecture, technology, and applications, destructive-read DRAM has slightly higher power consumption. Smaller DRAM memories will result in a smaller E_{MEM}, and can result in *less* energy consumption for DRAM due to shorter computation time. The result is also dependent on processor architecture: More complex processor will require more static power. Also, denser technology will have more leakage current and computation time will be more important. These estimates should be conservative since static power is still not significant at 180 nm technology.

There are several benefits by using destructive-read DRAM. (1) Cache size can be reduced by a factor four, decreasing chip area correspondingly [5]. This has a positive impact on power because each access to the smaller cache consumes less energy. In dense technologies with high leakage currents this will be even more the case, since caches have many transistors. (2) Power consumption in CPU is also reduced since computation time is reduced. However, since the number of accesses to the DRAM is increased the total power consumption is also increased. In our simulation models power is increased by 0.5% and 3% for 16 and 2 kbytes cache respectively. For the same execution time, the frequency and voltage can be scaled down when utilizing destructive-read DRAM since the instructions per clock cycle (IPC) is higher. This reduces power consumption, but is not analyzed in this paper. (3) The performance is increased, in our simulation this is 5% with 16 kbytes caches and 14% with 2 kbytes caches.

In systems with multiple processors and shared memory with cache coherence protocol, the cache line has to be marked dirty when read so that data is conserved. For multiple processors with message passing the data has to be read through the corresponding cache, i.e. the local processor should process the messages in order to conserve the data.

7 Conclusions

Destructive-read DRAM looks promising both from an energy and a speed perspective. More energy is spent on the DRAM memory and bus, but the execution

time is reduced and energy is saved in the processor. Denser technologies with higher leakage currents will benefit even more as more energy is wasted when the processor is stalled waiting for memory access to finish. For the simulated architecture the average power consumption were increased with 0.5% and 3% while the performance were increased with 4.9% and 14% for the applications in SPEC2000 benchmark for 16 kbytes and 2 kbytes caches respectively. Benefits from using destructive-read DRAM are increased performance and a reduction of chip area (by reducing cache size). Lower energy consumption might be possible through voltage-frequency scaling, but this also depends on the configuration and technology used. With higher leakage currents in denser technologies the destructive-read DRAM will be even more beneficial as the leakage currents (static power consumption) is higher.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. IEEE Computer, Volume 35, Issue2, February 2002.
2. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.
3. F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G.Kjeldsberg, T. Van Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors.* Kluwer Acad. Publ., Boston, USA, 2002. ISBN 0-7923-7689-7.
4. Jeff Draper, Chang Woo Kang, Ihn Kim, Gokhan Daglikoca, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, and Chun Chen. The architecture of the DIVA processing-in-memory chip. Proc. 16th ACM Int'l Conf. Supercomputing, pages:14-25, June 2002.
5. H. Dybdahl, M. Grannæs, and L. Natvig. Cache write-back schemes for embedded destructive-read DRAM. Submitted to ARCS 2006, 2006.
6. D. G. Elliott, W. Martin Snelgrove, and Michael Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In Custom Integrated Circuits Conference, Boston, MA, pages:30.6.1-30.6.4, May 1992.
7. M. Gokhale, B. Holmes, and K. Iobst. Processing in memory; the Terasys massively parallel PIM array. IEEE Computer, pages:23-31, April 1995.
8. Linley Gwennap. Embedded DRAM use rises. Nikkei Electronics Asia, June, June 2003.
9. R. Ho, K. Mai, and M. Horowitz. Efficient on-chip global interconnects. IEEE Symposium on VLSI Circuits, 2003.
10. Chomg-Lii Hwang, T. Kirihata, M. Wordernan, J. Fifield, D.Storaska, D. Pontius, G. Fredernanand B. Ji, S. Tomashot, and S. Dhong. A 2.9ns random access cycle embedded DRAM with a destructive-read. VLSI Circuits Digest of Technical Papers, Symposium on, pages:174-175, June 2002.
11. IBM microelectronics presentation: Embedded DRAM comparison charts. IBM Microelectronics, 2003.
12. International technology roadmap for semiconductors. http:public.itrs.net, 2003.

13. S. S. Iyer, Jr. J. E. Barth, P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak. Embedded DRAM: Technology platform for the Blue Gene/L chip. IBM J. Res & Dev. vol 49 no. 2/3 March/may, 2005.
14. B.L. Ji, S. Munetoh, C-L. Hwang, M. Wordeman, and T. Kirihata. Destructive-read random access memory system buffered with destructive-read memory cache for SoC applications. VLSI Circuits, Digest of Technical Papers. Symposium on, pages:85 - 88, June 2003.
15. Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Patnaik, and J. Torellas. FlexRAM: Towards an advanced intelligent memory system. International Conference on Computer Design, October 1999.
16. M.M. Khellah and M.I. Elmasry. Power minimization of high-performance submicron CMOS circuits using a dual-vdd dual-vth (DVDV) approach. ACM Int'l Symp. Low-Power Electronics and Design, pages:106-108, 1998.
17. G. Kirsch. Active memory: Micron's Yukon. Parallel and Distributed Processing Symposium, Proceedings. International, pages:11, April 2003.
18. AJ KleinOsowski and David J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.
19. P.M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and logic chip for massively parallel systems. IEEE, Advanced Research in VLSI, Proceedings. Sixteenth Conference on, pages:4-16, March 1995.
20. K. Mai, T. Paaske, N. Jayasena, W R. Ho, Dally, and M. Horowitz. Smart Memories: A modular reconfigurable architecture. ISCA, June 2000.
21. F. Morishita, I. Hayashi, H. Matsuoka, K. Takahashi, K. Shigeta, T. Gyohten, M. Niiro, H. Noda, M. Okamoto, A. Hachisuka, A. Amo, H. Shinkawata, T. Kasaoka, K. Dosaka, K. Arimoto, K. Fujishima, K. Anami, and T. Yoshihara. A 312-MHz 16-Mb random-cycle embedded DRAM macro with a power-down data retention mode for mobile applications. Solid-State Circuits, IEEE Journal of, Vol.40, Iss.1, Pages: 204- 212, 2005.
22. Y. Nunomura, T. Shimizu, and O. Tomisawa. M32R/D-integrating DRAM and microprocessor. Micro, IEEE, Volume: 17, Issue: 6, pages:40-48, November 1997.
23. M. Oskin, F.T. Chong, and T. Sherwood. Active Pages: A model of computation for intelligent memory. International Symposium on Computer Architecture, Barcelona, Spain, 1998.
24. P.R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, A. Vandercappelle E. Brockmeyer, C. Kulkarni, and P.G. Kjeldsberg. Data and memory optimization techniques for embedded systems. ACM Trans. Design Automation of Electronic Systems, 6(2):149–206, April 2001.
25. Yong-Ha Park, Hoi-Jun Yoo, and Jeonghoon Kook. Embedded DRAM (eDRAM) power-energy estimation for system-on-a-chip (SoC) applications. Proceedings of the 15th International Conference on VLSI Design (VLSID), p. 625, ASP-DAC/VLSI, 2002.
26. D. Patterson, T. Anderson, and K. Yelick. A case for intelligent DRAM: IRAM. Presented at Hot Chips VIII, Palo Alto CA, pages:18-20, August 1996.
27. Vijay Raghunathan, Mani B. Srivastava, and Rajesh K. Gupta. A survey of techniques for energy efficient on-chip communication. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 900–905, New York, NY, USA, 2003. ACM Press.
28. A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: the case for processor/memory integration. In *ISCA '96: Proc. of the 23rd annual int. symp. on Computer architecture*, pages 90–101, New York, NY, USA, 1996. ACM Press.

29. Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Western Research Lab, Research Report 2001/2, 2001.
30. L.V. Yerosheva, S.K. Kuntz, J.B. Brockman, and P.M. Kogge. A microserver view of HTMT. Parallel and Distributed Processing Symposium, Proceedings 15th International, pages:10, April 2001.
31. Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. R. Stan. HotLeakage: An architectural, temperature-aware model of subthreshold and gate leakage. University of Virginia Dept. of Computer Science, Tech. Report CS-2003-05, March 2003.

Enhancing Lower Level Cache Performance by Early Miss Determination and Block Bypassing

Haakon Dybdahl and Per Stenström

Presented at Asia-Pacific Computer Systems Architecture Conference (ACSAC), Shanghai, China, 2006.

Lecture Notes in Computer Science (LNCS) 3894
Springer.
Pages 52–66

Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination*

Haakon Dybdahl[1] Per Stenström[2]

[1] Dept. of Computer and Information Science,Norwegian University of Science and Technology,N-7491 Trondheim, Norway, dybdahl@idi.ntnu.no
[2] Dept. of Computer Engineering, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Goteborg, Sweden, per@ce.chalmers.se

Abstract. While bypassing algorithms have been applied to the first-level cache, we study for the first time their effectiveness for the last-level caches for which miss penalties are significantly higher and where algorithm complexity is not constrained by the speed of the pipeline. Our algorithm monitors the reuse behavior of blocks that are touched by delinquent loads and re-classify them on-the-fly. Blocks classified as bypassed are only installed in the level-1 cache. We leverage the algorithm to early send out a miss request for loads expected to request blocks classified to be bypassed. Such requests are sent to memory directly without tag checks at intermediary levels in the cache hierarchy. Overall, we find that we can robustly reduce the miss rate by 23% and improve IPC with 14% on average for memory bound SPEC2000 applications without degrading performance of the other SPEC2000 applications.

1 Introduction

As the speedgap between processor and memory has increased, the cache memory hierarchies have become deeper and the last-level caches (typically L2 or L3) have become bigger. Unfortunately, continuing along this route yields diminishing returns on investments because cache hit rate improves quite modestly with cache size and adding more levels increases the penalty taken when a request misses at all levels. Thus, it is important to study techniques that increase the utilization of deep cache memory hierarchies and that reduce the miss penalty.

One source of poor resource utilization is blocks with streaming behavior. Typically, there are multiple accesses to such blocks just after the miss. After this initial burst of accesses, the reuse distance is typically very long. When such blocks are installed in the cache, they may trigger replacement of blocks with a shorter reuse distance, thereby increasing the miss rate. One approach to reduce the detrimental effect of such blocks is to *bypass* them, rather than installing them. Several techniques to predict blocks subject to bypassing have been studied in the past. They typically fall into two broad categories – static [4, 15] and

* This work is partly sponsored by the HiPEAC Network of Excellence funded by EU under FP6.

dynamic [5–10,13,15,16]. In the static approach, either blocks touched by specific memory instructions in the program are bypassed, or the compiler partitions memory blocks that should be bypassed into special address space regions that are bypassed. In the dynamic approach, bypassing is based on the past behavior of an instruction or a block which guides future decisions whether to bypass or not. Statistics are stored in special data structures which guide bypassing decisions. Blocks predicted to have a reuse distance longer than their lifetime in the cache will be bypassed. To exploit the spatial locality of the initial burst of accesses, most approaches assume that they are installed in a special buffer that is significantly smaller than the cache. A misprediction can increase the miss rate: If the block is reused after it has been replaced from the special buffer but before it would have been replaced in the cache, the bypass operation results in a miss that would have been avoided without bypassing. While published prediction techniques have achieved high prediction coverage, they sometimes increase the miss rate due to low accuracy which leads to inconsistent performance gains.

While previous attempts using bypassing were applied to first-level caches, we study in this paper block bypassing algorithms for last-level caches. They have a much higher potential than for the first-level cache for several reasons. First, the first-level cache is heavily constrained by the speed of the pipeline. Hence, cache management algorithms must be simple. Second, the latency of first-level cache misses that hit in subsequent levels does not incur much penalty because it can be often successfully hidden by the latency tolerance capability of multiple-issue out-of-order cores of moderate issue rate. Third, bypassing has much higher potential at the lower levels as the miss penalty is significantly higher. On the contrary, mispredictions are also much more costly making algorithm robustness a key issue. However, assuming a two-level cache hierarchy, which forms the base for our experiments, our overall strategy is to bypass blocks at the second level but always install them at the first level. As a result, as long as the reuse distance for incorrectly predicted bypassed blocks is smaller than the size of the level-1 cache, there will be no penalty for mispredictions.

Previous studies [1,12] have shown that a few load instructions are responsible for most of the cache misses, called *delinquent loads*. Our approach is to base the prediction of which blocks to be bypassed by detecting such loads and record them at run-time. However, we validate the correctness of the prediction and change it by also monitoring whether we erroneously bypass a block with a reuse distance shorter than the L2 cache but longer than the L1 cache by monitoring the reuse at the L2 cache. While this basic approach uses some of the components from the dynamic scheme proposed by Tyson et al. [15], we found that the Tyson scheme increased the miss rate for many of the applications. We identified several useful extensions that eventually offered more consistent performance improvements. For example, by storing tags for bypasses we can identify when bypasses are done incorrectly and stop bypassing for the involved instruction.

Another disadvantage of using deeper memory hierarchies addressed in the paper, is the increased miss penalty for the requests that miss at all levels. If one could determine that a miss will not be satisfied at any level without doing tag

checks at all levels, the miss penalty can be reduced quite significantly. We also present results for a simple early miss determination approach that leverages our bypassing algorithm. If the instruction that causes the L1 miss is predicted to bring a bypassed block into the cache, it is likely that it will not be found at any level. We then send this request speculatively to the memory, potentially reducing the miss penalty.

We evaluated our algorithm using 23 applications from SPEC2000 on a simulation model based on SimpleScalar V3 modeling a 4-issue out-of-order core. On average we improved the L2 miss rate by 23% as compared to the upper-bound achieved by an oracle algorithm which is 34%. The infrastructure needed to monitor access behavior is quite small – the storage area of the cache is increased with less than 7%. We found that our early miss determination scheme could correctly predict that the block is not available for 27% of the cases with only 1% of the accesses incorrectly predicted as misses on average. Overall, the bypassing algorithm together with the early miss determination scheme improved the IPC by 14% for the memory bound applications. The algorithm is robust and for the non-memory-bound applications the average IPC is slightly improved in contrast to Tyson's scheme [15] where most of the applications suffer.

Our baseline architecture and the Tyson algorithm are presented in the next section. Sections 3 to 5 present the new schemes including our bypassing algorithm, an oracle algorithm and our early miss determination algorithm. Experimental methodology and our evaluation are found in Sections 6 and 7. We relate our findings to prior work in Section 8 and conclude in Section 9.

2 The Tyson Scheme for Dynamic Bypassing

Two schemes for dynamic bypassing of memory accesses for the L1 cache were described and evaluated by Tyson et al. in [15]. Only one of them, called *improved dynamic bypassing scheme*, improved the performance and therefore we do not consider the other scheme. We call this scheme the *Tyson scheme*. The principle of this scheme is that a few instructions load data that pollute the cache. These instructions shall not store data in cache – data is bypassed to the processor. This increases the hit rate for the other instructions and for the total system. Another advantage is that when bypassing data only a single word is read from main memory and hence the memory bandwidth usage is reduced.

2.1 Structures

A table associates a counter with each instruction that is a candidate for bypassing as shown in Fig. 1(a). Instructions are identified by their static memory address (*inst*). The cache blocks are extended with the *fetched by* field which refers to an entry in the instruction table, see Fig. 1(b).

2.2 Algorithm

The Tyson scheme uses the following events:

Instruction	Counter
inst a	2
inst c	3
...	...

(a) The instruction table.

Index	Tag	Cache line	Fetched by inst
0	*tag a*	..	*inst a*
1	*tag c*	..	*inst c*
....	...	..	..

(b) The extended cache structure.

Fig. 1. The structures for Tyson Scheme

1. *Cache miss.* The instruction (*inst*) that triggers a cache miss is inserted into the instruction table with a zero counter if it is not already present. For instructions that are already present, the counter is incremented. In Fig. 1(a), if instruction $inst_a$ is requesting a non-existing cache block, $inst_a$'s counter is increased from two to three.
2. *Cache hit.* The instruction (*inst*) that caused the cache hit is looked up in the instruction table and its counter is decremented. The instruction referred to by the *fetched by* field is also looked up in the instruction table, and if present its counter is decremented. For example if instruction $inst_x$ requests the cache block with index 1, which was fetched by $inst_c$, the counter of $inst_c$ is decremented from value three to value two, see Fig. 1.
3. *Bypassing.* A bypass is performed when an instruction causes a cache miss and the instruction is found in the instruction table with its counter equal or greater than a preset threshold value. Instead of loading a cache line, only a single word is fetched from main memory and it is not stored in L1 cache.

The threshold value for bypassing and the maximum value for the counter in our evaluation of the Tyson scheme later in the paper is set to three.

2.3 Discussion

The processor is able to load a single word of eight bytes in the Tyson scheme, and this is said to be four times more efficient than loading a cache line that consists of four words. This is not true for today's systems with wide data buses, interleaved main memory banks, pipelined memory accesses and burst transfer options. Incorrect or very aggressive bypassing of cache lines in the Tyson scheme have a limited impact since the latency of bypassing a cache line, i.e., fetching a single word, is less than loading a cache line (in their model). In [15] it was found that the memory bandwidth requirements was reduced by more than 20% for integer applications which is not surprising as most of the integer application do not benefit from prefetching of longer cache lines. It was found that the cache hit ratio was increased by up to 26% for some floating point benchmark applications, and for some cache configuration the cache hit ratio was increased by 2-3% on average. Unfortunately, the performance was found to be unstable across the benchmarks and the median performance is a degradation of the cache hit rate.

Using Tyson's scheme in an L2 setting increases the aggressiveness by which blocks are bypassed. Instructions with three consecutive cache misses trigger bypassing if the fetched cache lines are not re-accessed in L2 between the execution

of each of these instructions. Data is more likely to be re-accessed in the L1 cache than in L2 cache since the L1 cache filters out some of the hits in the L2 cache. This makes Tyson's algorithm more aggressive in bypassing the L2 cache than in its original setting in the L1 cache and creates a serious disadvantage.

In general, out-of-order execution processors tolerate memory latency to some extent, which makes L1 misses that hit in L2 less serious. However, misses caused by incorrect bypassing in the last level cache (L2 in our experiments) is expected to stall the processor for a significantly longer time. Consequently, a more sophisticated heuristic is needed to control the bypassing of the L2 cache. The accuracy of the bypassing heuristics then becomes crucial.

3 New Scheme for Bypassing

Our proposed scheme increases the precision of the bypassing using a feedback loop in which the correctness of its decisions is used as inputs to the heuristics controlling the algorithm. In addition to keeping the tag for the present cache line, each cache block contains the tag for the other cache line that would have been present if the previous fetch was bypassed/not bypassed. Detection of eviction of cache blocks that are not accessed is used to determine if instructions should be bypassed. All this information is used for the feedback loop to enable or disable bypassing for each instruction which results in robust performance.

3.1 Structures

Like in the Tyson scheme, instructions that cause cache misses are inserted in a table as shown in Fig. 1(a). Instructions are identified by their static memory address (*inst*). The cache block is extended with new fields as shown in Fig. 2. The *fetched by* field contains the instruction (*inst*) that fetched the cache line.

Index	Tag	Cache line	Fetched by inst.	Used	Shadow inst.	Shadow tag	Shadow status
0	*tag a*	..	*inst a*	*FALSE*	*instb*	*tag b*	*bypassed*
1	*tag c*	..	*inst c*	*TRUE*	*instd*	*tag d*	*replaced*
....	...	..	....	...	..	..	..

Fig. 2. The cache structure is extended to include data used for the heuristics

The *used* field is used to detect cache blocks that are replaced without being accessed, an indication that the cache line should have been bypassed. It is reset when the cache line is replaced, and set on a cache hit. The *shadow instruction* is set to the instruction (*inst*) that caused the replacement or that was bypassed. For a replaced cache line the *shadow tag* is updated with the tag of the replaced cache block, and for bypassed cache lines the *shadow tag* is updated with the value of the tag of the block that is bypassed. *shadow status* indicates if the last request was bypassed or caused a replacement.

3.2 Algorithm

The events that trigger actions in the new scheme are the following:

1. *Cache miss.* An instruction that triggers a cache miss is inserted into the instruction table with a zero counter if it is not already present. The counter is incremented for the instruction. For example, if instruction $inst_a$ is requesting a non-existing cache block, its counter is increased (Fig. 1(a)). The *shadow tag* is updated with the tag of the data that was in the cache block, the *shadow instruction* with the instruction that caused the miss and the *shadow status* is set to "replace".
2. *Cache hit.* The counter for an instruction that triggers a cache hit is decreased if present in the instruction table. If the instruction referred to by the *fetched by* field in the cache block is in the instruction table, its counter is also decreased. Finally, the field *fetched by* is cleared. This means that fetching data for other instructions is only used once by the heuristic. For example if instruction $inst_x$ requests a cache block with index 1 which was fetched by $inst_c$, its counter is decreased from value three to two, see Fig. 2.
3. *Cache hit caused by bypassing.* A cache hit is said to be caused by bypassing when the cache block contains information about bypasses for that cache block. The counter for the instruction referred to by the *shadow instruction* is increased if the instruction is present in the instruction table. If not, it is inserted. Finally the shadow information is cleared which means that this event is only used once. For example if data with tag_a in index 0 is requested, the $inst_b$ will be inserted into the instruction table.
4. *Cache miss caused by bypassing.* A cache miss is said to be caused by bypassing when the tag of the requested data is found in the *shadow tag* and the *shadow status* indicates bypassing. The counter for the instruction that caused the bypassing is decreased in the instruction table, if present. For example if the data with tag_b is requested in index 0, the counter for $inst_b$ is decreased if the instruction was in the instruction table.
5. *Cache miss caused by not bypassing.* A cache miss is said to be caused by *not* bypassing when the requested tag matches the *shadow tag* and the *status bit* indicates that the shadow data reflects a replacement. The counter for the instruction that replaced the cache block is increased. If the instruction is not present in the table, the instruction is inserted. For example if data with tag_d is requested with index 1, the counter for $inst_c$ is increased.
6. *Data replaced without being used.* This happens when an instruction loads a cache block that is not accessed before it is overwritten. The counter of the instruction that fetched the data is increased, if present. For example if the first cache block is replaced, the counter for $inst_a$ is increased.
7. *Bypassing.* A bypass is triggered by a cache miss and requires that the counter for the instructions that caused the cache miss is above a threshold limit. The cache block is loaded from main memory and into the L1 cache without being stored in the L2 cache. The *shadow tag* is updated with the tag of the data that is bypassed, the *shadow instruction* with the instruction that caused the miss and the *shadow status* is set to "bypass".

Each event can increase or decrease the value of the counter with different values, see Fig. 3, and different levels of threshold and maximum value of the counter can be used.

		Tyson	New Scheme	Shadow	Replace	Equal	Tyson II
1	Cache miss	1	1	0	0	1	2
2	Cache hit	-1	-2	0	0	-1	-3
3	Cache hit caused by bypassing	0	3	1	0	1	0
4	Cache miss caused by bypassing	0	1	1	0	1	0
5	Cache miss caused by not bypassing	0	-1	-1	0	-1	0
6	Data replaced without being used	0	1	0	1	1	0
7	Bypassing threshold/max counter value	3	9	3	3	3	6

Fig. 3. Different configurations for the heuristics for bypassing

3.3 Discussion

The shadow data is used to monitor the consequences of bypasses and replacements. Different applications benefit from different parameter settings. However, as we will see, the described algorithm with the new scheme (see Fig. 3) works well across different benchmarks and for different cache sizes.

Structures for storing the data shown in Figures 1(a) and 2 are assumed to be implemented in hardware. The instruction table is limited in size and all simulations are done with a size of 32 elements. The resources needed are therefore small in comparison to the ones needed for an L2 cache. The extension of the cache block to include information for the heuristics will however require more hardware. The number of bits needed to store a cache block for a conventional architecture and the new scheme are shown in Fig. 4. The *shadow tag* is of the same size as the tag for the cache block. Given a system with 8-GByte physical memory (33 bits) and a 1-MByte 4-way set associative cache 6 bits are used to address the byte within the cache line and 12 bits are used to index the cache block which leaves $33 - 6 - 12 = 15$ bits for the tag. There are two references to two instructions in the instruction table, i.e. the fields *fetched by instruction* and *shadow instruction*. These contain the address for the instruction in memory. The address of the instruction is used as a key and is 33 bits. By changing the algorithm slightly so that instructions are only placed in the instruction table on cache misses, these two fields only need to point to an instruction in the instruction table and it is not necessary to store the whole address of the instruction. This modification does not incur any measurable performance degradation. However, when these address fields only point to the instruction table, there is no way of detecting when the instructions in the table are replaced. Therefore the instruction address should be hashed and stored in the cache block to discover when instructions are replaced. This results in 5 bits for pointing into the instruction table, and let us assume 4 bits for hashing the instruction address. The total increase in number of bits of storage is less than 7% for the cache block with pointers as shown to the right in the table.

Field	Tag	Cache Line Data	LRU data	Fetched by inst.	Used	Shadow inst.	Shadow tag	Shadow status	Total
Conv. architecture	15	512	2	0	0	0	0	0	529
New Scheme	15	512	2	33	1	33	15	1	612
Instruction pointers	15	512	2	9	1	9	15	1	564

Fig. 4. The number of bits used for storing a cache block for a computer with 8 GBytes memory (33 bits) and a 1 MByte 4-way set associative cache

Bypassing breaks the *inclusion*; blocks in L1 cache are not guaranteed to exist in the L2 cache. The L1 cache must be accessed to get the latest values by memory coherence schemes. However, this is no different from conventional caches with delayed write back schemes. Directory coherence protocols can be used to know what data that are loaded in L1.

4 Oracle Bypassing Algorithm

In this section, we derive an algorithm to assess how optimally our scheme can avoid misses by bypassing blocks in the cache. Optimal algorithms depend on the problem space, e.g. deciding which cache block that should be replaced. In this case the trace of memory accesses can be analyzed and blocks are installed in the cache if the reuse distance is shorter than the reuse distance for the block that is replaced [3, 14]. However, we are not considering the replacement policy for the cache. We are interested in the optimal algorithm for deciding whether to bypass cache blocks combined with the standard least recently used (LRU) policy. This makes the optimal algorithm more complex, it can not just look at the reuse distance to decide upon bypassing or not.

We have made a new algorithm that requires only a single run of the benchmark and is optimal for direct-mapped caches and set-associative caches with random replacement policies. The algorithm is based on the idea to postpone the decision regarding bypassing until it is known whether it reduces the cache miss rate or not. For an instruction that causes a miss, the algorithm maintains the cache state for the two possible outcomes: bypassing versus not bypassing. That is, each cache block is extended dynamically to keep track of data for both outcomes. On a hit to the cache block it is known which requests that should have been bypassed and which should be replaced, and the simulation will only consider one of the alternatives for the rest of the simulation. Consider the following example. Several requests that map to the same cache block appear in the following order: a, b, c, and a. Without bypassing, b will replace a, c will replace b and then a will replace c, i.e. only cache misses. With optimal bypassing b and c are bypassed, and the last a will become a hit instead of a miss. Our oracle algorithm will extend the cache structure dynamically to contain first a, then a and b, and then a, b and c. In the end when a is requested again, it sees that by bypassing b and c, a hit is generated for a, and the decision about bypassing is done. The extra data stored can be deleted at this point. An application without any hits will build up a large data structure, but only linear to the number of instructions. Set-associative caches with LRU algorithms are more complex.

Storing information about both decisions doubles the storage requirement for each missed block. One way to reduce the storage requirements is to change the replacement policy. For each bypass we change the LRU cache block to become the recently used in the same way as a replacement of the cache blocks does, i.e. bypass and replacement change the LRU stack in the same way. A decision to bypass or not only regards one element in the cache and only one additional element need to be stored for each miss. Again, the data structure needed is linear in the number of instructions (in case of only misses).

The result is an oracle algorithm with a cache with an LRU replacement policy that is not LRU when accesses are bypassed. This tends to underestimate the performance gains of bypassing and hence reduces the upper bound. With this caveat, the oracle algorithm is nevertheless used to assess if there is a potential for using bypassing, and whether there is room for improvement.

5 Early Miss Determination

A problem with deep hierarchies is that the miss penalty is increased for each level added. By predicting early on whether a request will miss at all levels, and accessing main memory in parallel with tag checks at the intermediary level, the miss penalty is expected to be reduced substantially [11].

If the data is found in the cache, the data from cache is used. However, we assume that there is no way of removing the ongoing memory access to main memory system. Miss-prediction will thus increase the memory bandwidth usage and can stall other memory accesses. We look at using the heuristic for bypassing data in the new scheme to early determine cache misses and launch a memory access. Instructions that are in the instruction table and with maximum counter value are considered to miss in the L2 cache. An oracle scheme for early miss determination would manage to predict all cache misses.

6 Methodology

Simulation is used to study the efficiency of the two schemes for bypassing memory accesses for the L2 cache, the oracle algorithm, and early miss determination scheme. The *sim-cache* model is used for studying cache miss rates and the execution model is single-issue in-order. No timing information is included in these simulations. The *sim-outorder* model used for studying IPC improvements is a clock-cycle level out-of-order execution model with non-blocking caches. The models are part of SimpleScalar version 3 [2]. These models are extended to simulate memory congestion, the different bypass schemes, the oracle algorithm, and the early miss determination scheme. A logical sketch of the simulated system is shown in Fig. 5.

The baseline for the simulator is shown in Fig. 6. Data and instruction look-aside buffers (DTLB and ITLB) are not simulated in the *sim-cache* model. We assume that the hit and miss latency for the L2 cache are equal.

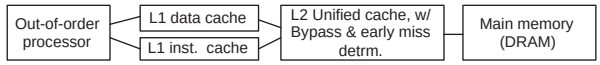


Fig. 5. The simulated single processor core with the memory hierarchy

Parameter	Value
RUU size	128 instructions
LSQ size	64 instructions
Fetch queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Functional Units	4 INT ALUs, 4 FP ALUs, 1 INT Multiply/Divide, 1 FP Multiply/Divide
Branch Predictor	Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history table, 4K Chooser
BTB	512-entry, 4 way
Mispredict Penalty	7 cycles
L1 Instruction/Data Cache	32K, 4-way (LRU), 64 B Blocks, 1 cycle latency
L2 Cache	Unified, 1 M, 4-way (LRU), 64 B Blocks, 24 cycles latency
Main Memory	200 cycles first chunk, 10 cycles inter chunk, 4 independent subbanks
I-TLB/D-TLB	128-entry, fully associative, 30 cycles miss penalty

Fig. 6. Micro-architectural parameters

The size of the instruction table is 32 for all experiments. The original Tyson scheme used a branch predictor like table for the L1 cache. This means that our implementation of Tyson is slightly different, and the reason is that we are bypassing the L2 cache.

SPEC2000 applications were used as benchmarks with the reference data sets. Each simulation is forwarded one billion instructions and then simulated for two billion instructions.

7 Evaluation

7.1 L2 Cache Misses Reduction

There are two groups of applications that are of interest: (a) memory bound applications that should obtain increased speedup and reduced cache miss rate for L2 cache and (b) non-memory-bound applications. The last group will not receive much benefit, but should not be slowed down or suffer from an increased cache miss rate. The reduction of the L2 cache miss rate is shown for the Tyson scheme, the new scheme, and the Oracle algorithm for SPEC2000 applications in Fig. 7. For the 1-MByte configurations and the memory bound applications (*art*, *mcf* and *ammp*) all schemes reduce the average miss rate. However, for the rest of the applications Tyson's scheme increases the miss rate by average 43%. The scheme is aggressive and therefore is only suitable for applications that benefit from bypassing. The new scheme is more robust and only increases the miss rate with average 2%. Compared to the Oracle scheme 68% of the possible misses are removed for the memory bound application and the new scheme, which means that the scheme is working well but there is still room for improvement.

For the 256-KByte configuration Tyson scheme does not work even for the memory bound application. The new scheme reduces the miss rate by 4% while the Oracle scheme obtains an 8% reduction.

		256k L2				1 MByte L2			
Spec benchmark	# of L2 acesses	Convent. Miss rate	Reductions of miss rate in % Tyson	New Sch.	Oracle	Convent. Miss rate	Reductions of miss rate in % Tyson	New Sch.	Oracle
art	308461787	0.826	3	9	15	0.603	58	47	67
mcf	265670845	0.616	-44	-5	1	0.599	-13	-3	5
ammp	132534118	0.920	5	7	8	0.845	27	25	29
average	235555583	0.787	-12	4	8	0.682	24	23	34
swim	77712825	0.591	-17	-2	0	0.590	-20	-2	0
applu	63107223	0.667	0	0	1	0.666	0	0	2
gcc	59690438	0.153	13	16	40	0.037	-19	2	23
lucas	51951704	0.862	-7	3	15	0.843	-16	3	14
facerec	36104376	0.657	-34	5	15	0.360	-69	-7	1
apsi	34560805	0.435	-9	4	18	0.206	-60	0	0
mgrid	31808765	0.760	-14	2	15	0.478	-53	-1	0
parser	24836151	0.349	-76	-5	11	0.165	-111	-18	7
galgel	20719227	0.880	4	13	24	0.352	-61	50	75
bzi2p	17548828	0.384	-33	-6	9	0.175	-27	3	14
crafty	16956861	0.065	-24	-2	21	0.007	-2	0	15
gzip	15660311	0.072	-3	4	19	0.039	-13	-6	1
gap	9244337	0.505	-78	0	0	0.499	-78	0	0
wupwise	8914610	0.679	-33	0	6	0.628	-44	-7	2
fma	8064048	0.000	0	-36	0	0.000	2	-37	0
equake	2758253	0.028	-84	-1	0	0.028	-84	-1	0
eon	2293674	0.000	0	-13	0	0.000	0	-13	0
perlbmk	799811	0.285	-146	-2	16	0.239	-132	-5	5
average	26818458	0.410	-30	-1	12	0.295	-43	-2	9

Fig. 7. Reduction of cache miss rate for different bypass schemes for two sizes of L2

7.2 Early Miss Determination

The results for using the proposed scheme for early miss determination is shown in Fig. 8. Each bar in the graph consists of four parts. The top part, which is black, is the amount of the L2 cache accesses that are predicted as misses incorrectly. These increase the memory bandwidth usage by a modest 1%. 27% of the memory accesses with cache misses are correctly predicted as misses. The advantage of the early miss determination depends on the latency of cache misses in the L2 cache. This miss latency increases with the size of the cache.

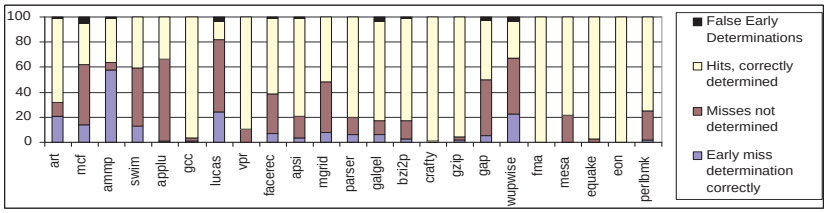


Fig. 8. Memory accesses to the L2 cache divided into four groups

7.3 Memory Bandwidth Reduction

The reduction of the memory bandwidth usage is shown in Fig. 9. The total number of accesses is decreased for both the new scheme and also when the new

scheme is combined with early miss determination. Tyson's scheme reduces the number of accesses well for the *MCF* application even though it increases the miss rate for the same application. This is because Tyson's scheme reduces the number of write-backs by 48% for the L2 cache. By comparison the new scheme only reduces the number of write-backs by 8%.

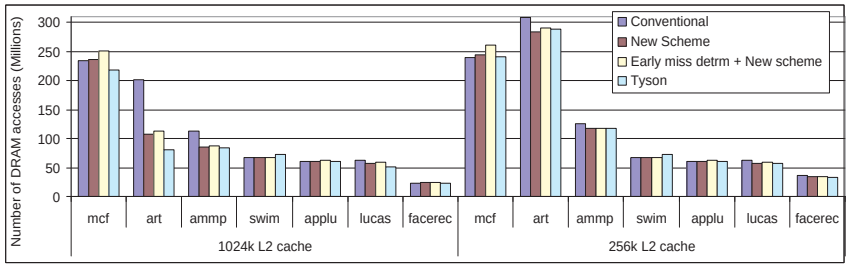


Fig. 9. The number of DRAM accesses (sum of L2 misses and L2 write-backs) executed for the different applications in the SPEC2000 benchmark

7.4 Instructions Per Clock Cycle (IPC)

We have studied the IPC for two different configurations, with and without main memory congestion. Four memory banks are used for simulation with congestions. These are interleaved so that one cache line fits into one bank. Each bank can handle only one read or write access at any time, which means that the entire system is capable of handling up to four read/write accesses simultaneously. There is no writeback buffer for the main memory, but L1 and L2 have writeback buffers. The model without congestion uses unlimited size writeback buffers to main memory and an unlimited number of parallel read operations can be executed simultaneously. Simulation with congestion is shown in Fig. 10 and no congestion in Fig. 11. These figures represent the speedup compared to the IPC of the conventional architecture which is shown in Fig. 12. We see two important observations: The first is that the performance for the memory bound applications is improved significantly. However, for the Tyson scheme the performance is decreased for non-memory bound application. For *galgel*, the simulation with congestion results in an IPC degradation of 22% and for the non-congestion simulation the IPC is degraded by 10%. There are three applications that are memory-bound: *ammp*, *art* and mcf. With memory congestion modeled, Tyson's scheme is able to increase the IPC quite well. This is not true when not using congestion because the bandwidth consumed by write-backs of dirty cache blocks then does not have any impact. Tyson's scheme bypasses write-backs from dirty L1 cache blocks well. Overall, the combination of bypassing and early miss determination can improve IPC with up of up to 34%. For

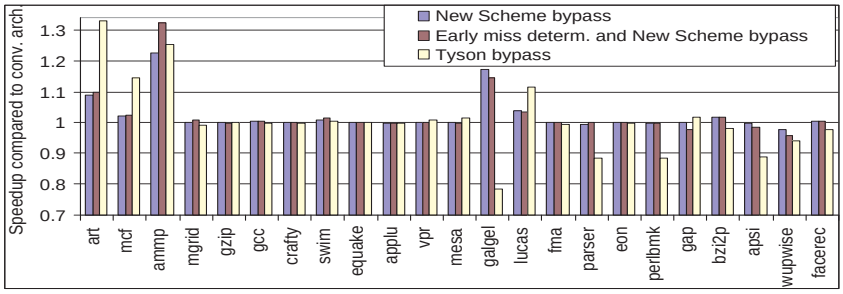


Fig. 10. Speedup compared to conventional architecture, with simulation of congestion in main memory

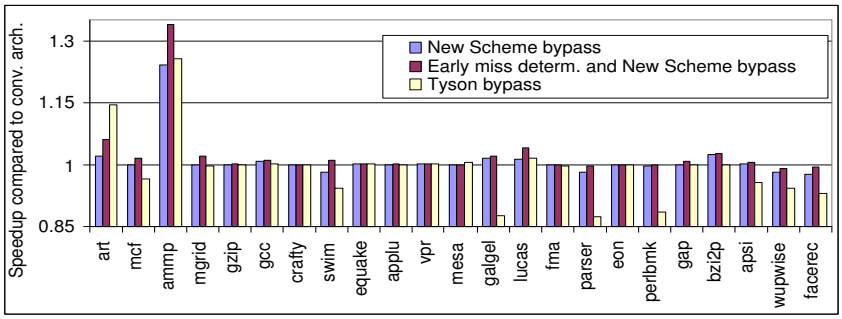


Fig. 11. Speedup compared to conventional architecture, with no simulation of congestion in main memory

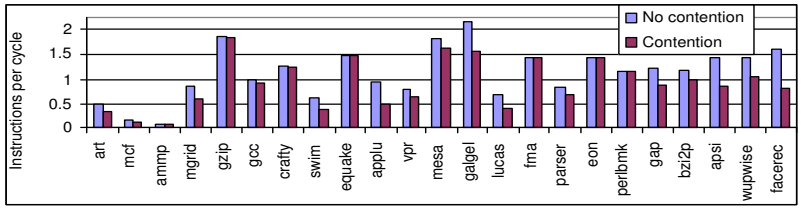


Fig. 12. IPC with and without simulation of congestion to main memory

applications with many incorrect early determinations (see Fig. 8) the IPC can be degraded.

7.5 The Heuristics

In the description of the algorithm used by the new scheme, the counter was added/subtracted with different values for different events. Different configurations for the heuristic is shown in Fig. 3. In the *shadow* configuration only the shadow events are enabled. This is learning by considering what could have been in the cache line if the bypass decision was inversed. In some sense this is learning by history and mistakes. In the *Replace* configuration bypassing is done when an instruction fetches a cache block which is replaced without having any cache hits. The configuration is aggressive since there are no negative numbers in the configuration. The *Equal* configuration increases/decreases the counter with the value one for all events. The *Tyson II* is the Tyson algorithm tweaked a little bit to become less aggressive. The different configurations from Fig. 3 are evaluated in Fig. 13. Even though the new scheme has overall good performance, tuning the heuristic differently for each application has a potential for improving the performance further.

	Tyson	New Scheme	Shadow	Replace	Equal	Tyson II	Conventional
ammp	1	3	10	7	8	0	37
apsi	82	0	0	22	42	18	0
art	0	26	46	4	7	33	140
bzi2p	54	0	23	226	26	7	3
crafty	17	1	29	562	13	3	0
facerec	75	7	9	74	70	36	0
galgel	231	24	0	203	22	7	147
gcc	24	1	0	2	11	4	3
gzip	17	7	1	0	14	10	1
lucas	11	3	0	6	17	9	5
mcf	14	3	12	34	9	8	0
mesa	13	1	0	0	2	3	0
mgrid	52	1	2	12	45	21	0
parser	124	18	20	100	67	35	0
swim	17	1	0	8	14	28	0

Fig. 13. The numbers show the increase in cache miss ratio compared to the best configuration. The best configurations are grayed and have value zero

8 Related work

Bypassing can reduce conflict misses by using a bypass buffer in parallel with a direct mapped cache [6, 7, 10]. However, direct mapped caches are not used in state-of-the-art high performance microprocessors and reduce the potential for these techniques.

9 Conclusion

This work is the first to explore the gains of bypassing for last-level caches. The potential for improvement is higher compared to bypassing L1 cache because the

latency of an L2 miss is much higher than an L1 miss seen from the processor. A new scheme for bypassing is presented based on a feedback loop. This improves the performance in terms of cache miss ratio and IPC to the same level as simpler schemes for memory bound applications, but it does not degrade the performance for non-memory bound applications which is the case for earlier schemes. We include early miss determination which further improves performance by predicting misses in the cache simply by using the heuristics for the bypassing scheme. This reduces the latency time for memory accesses at the cost of a small increase in bandwidth usage. The final contribution is our establishment of an upper-bound of miss-rate reduction in last-level caches by devising an oracle algorithm. Even though this algorithm is not fully optimal, it shows that there is room for improvements and more research is needed.

References

1. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *MICRO 26*, 1993.
2. T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. IEEE Computer, Vol. 35, Issue2, 2002.
3. L. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2):78-101, 1966.
4. Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. *Annual Hawaii International Conference on System Sciences*, 1989.
5. J. Jalminger and P. Stenstrom. A cache block reuse prediction scheme. *Microprocessors and Microsystems, V28*, pages 373–385, 2004.
6. L.K John and A. Subramanian. Design and performance evaluation of a cache assist to implement selective caching. *Proc. of Intl. Conf. on Comp. Design*, 1997.
7. T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W Hwu. Run-time cache bypassing. *IEEE Transactions on Computers, V48 I12*, pages 1338–1354, 1999.
8. M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *CF '04: Proc. of the 1st conf. on Computing frontiers*, 2004.
9. M. Karlsson and E. Hagersten. Timestamp-based selective cache allocation. *High Performance Memory Systems, Springer-Verlag*, 2003.
10. Scott McFarling. Cache replacement with dynamic exclusion. In *ISCA '92*, pages 191–200. ACM Press, 1992.
11. G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *HPCA*, 2003.
12. V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *int. symp. on Code generation and optimization*, 2004.
13. J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS*, 1998.
14. R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Joint International Conference on Measurement and modeling of computer systems*, 1993.
15. G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to cache management. *MICRO*, 1995.
16. Wayne A. Wong and Jean-Loup Baer. Modified LRU policies for improving second-level cache behavior. *HPCA*, 2000.

An LRU based Replacement Algorithm Augmented with Frequency of Access in Shared Chip Multiprocessor Caches

Haakon Dybdahl, Per Stenström and Lasse Natvig

Presented at the MEDEA Workshop (held in conjunction with PACT Conference), Seattle, USA, 2006.

To be published in ACM SIGARCH Computer Architecture News.

An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches

Haakon Dybdahl[1] Per Stenström[2] Lasse Natvig[1]

[1] Dept. of Computer and Information Science,Norwegian University of Science and Technology,N-7491 Trondheim, Norway, [dybdahl,lasse]@idi.ntnu.no
[2] Dept. of Computer Engineering, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Goteborg, Sweden, per@ce.chalmers.se

Abstract. This paper proposes a new replacement algorithm to protect cache lines with potential future reuse from being evicted. In contrast to the recency based approaches used in the past (LRU for example), our algorithm also uses the notion of *frequency of access*. Instead of evicting the least recently used block, our algorithm identifies among a set of LRU blocks the one that is also least-frequently-used (according to a heuristic) and chooses that as a victim. We have implemented this replacement algorithm in a detailed simulation model of a chip multiprocessor system driven by SPEC2000 benchmarks. We have found that the new scheme improves performance for memory intensive applications. Moreover, as compared to other attempts, our replacement algorithm provides robust improvements across all benchmarks. We have also extended an earlier scheme proposed by Wong and Baer so it is switched off when performance is not improved. Our results show that this makes the scheme much more suitable for CMP configurations.

1 Introduction

Typically, the first one or two levels of cache in a chip multiprocessor are private to each processor whereas the last level cache is a shared cache. A shared cache can shield the long (and increasing) memory latency more effectively as it can leverage the sharing of data among cores. Unfortunately, shared last level caches also introduce the following two problems. First, while a cache miss will stall the processor that caused the cache miss, the other processors may still be served. However, the off-chip bandwidth can be a bottleneck. According to ITRS [12], the number of I/O pins is not expected to scale with Moore's law. As a result, the off-chip bandwidth may become a bottleneck in a near future. Such a scenario means that a cache miss in the last level cache will not only slow down the processor that has to wait for main memory; it may also slow down the other processors due to off-chip communication contention. Second, when one processor causes cache misses, cache lines will be read from main memory and inserted into the last level cache. This may cause capacity and conflict misses for the other processors. Therefore cache misses for one processor may increase the number of caches

misses for the other processors as well. The bottom line is that a cache miss in the last level can decrease performance in three ways (1) it stalls the requesting processor, (2) it consumes potentially precious off-chip bandwidth, and (3) it may increase the number of cache misses for other processors.

It is well documented that the widely implemented *least recently used* (LRU) replacement policy for caches is far from optimal for many applications in uniprocessor configurations [7]. In a CMP configuration, the suboptimal behavior of LRU is amplified further.

This paper proposes a new algorithm for caches that augments the basic LRU algorithm with the notion of frequency. Each cache line is extended with a counter that is used in combination with the LRU algorithm to pick cache lines for eviction. The LRU algorithm selects a set of least-recently-used *potential* victims. Our extension selects among these potential victims, the one with the lowest access frequency. In addition, it may also victimize the block requested at a miss by not caching it at all in the last-level cache but rather installing it in the higher level cache. We refer to this as selective caching [3, 5, 6, 9, 14]. While previously proposed work has also considered counters for monitoring access frequency [8, 11, 13], they did not consider shared caches in CMPs and used a different and more complex heuristic [7]. By contrast, our approach is more tractable. Wong and Baer also proposed a scheme for improving the LRU replacement policy for the last level cache [16]. While our scheme maintains reuse information on each cache block (address based), they maintain reuse information based on the instruction that touches the blocks (instruction based). As a result, they can amortize the information of reuse on multiple blocks touched by the same instruction. However, as we shall see, their scheme provides unstable performance in a shared CMP cache setting.

Another contribution in this paper is a mechanism for switching between two replacement policies. This mechanism monitors the performance of two alternative replacement policies and switches between the two when the alternative policy performs better.

As for the rest of the paper, Section 2 describes our new schemes in detail. Section 3 then describes our evaluation methodology. We move on to present the experimental results in Section 4 and relates our findings to work by others in Section 5. We conclude in Section 6.

2 The New Schemes

In a conventional cache, the most recently used cache lines are protected by selecting as a victim the LRU cache line. The new scheme protects the most recently used cache lines, but picks one of the LRU cache lines (not necessarily *the* least recently used line). Among the LRU cache lines, the access pattern, where frequency and recency of use are important factors, determines which cache line that is a candidate for eviction.

We also describe a method for *switching* dynamically between basic LRU and the new scheme based on which of them that performs best.

2.1 Structures

The cache blocks in the new scheme are extended with a counter for each block, see Table 1. For a cache line of 64 bytes the overhead of the counter is less than 2%.

The basic structures needed for *switching* are the shadow tags (duplicated tags) for some sets in the last-level cache. The alternative scheme is run on these shadow tags as the schemes do not require the cache line data itself. These few tags, less than 1% of the number of tags in the main cache, are used to represent the behavior for all of the sets in the last level cache. The number of bits required for these extra tags, LRU bits and counters are insignificant compared to the total number of bits for the cache.

2.2 The New Replacement Policy

The counters for the cache blocks in the matching set are modified depending on the type of access:

- Hit: The counter for the cache line is increased by the constant value hit_{read} or hit_{write} up to a maximum value $maximum$, for read or write hits, respectively.
- Any access (i.e. read hit, write hit and miss): The *counters* for all cache lines in the set are decreased by one (if they are larger than one), except for the cache line which has a hit, if any. This decrementation in combination with the maximum value for the counter implements a maximum time for protection, even if the cache line was accessed an unlimited number of times in the past.

The replacement policy is a heuristic based on the positions in the LRU stack and the values of the *counters*. The most recently used cache lines are not candidates for replacement on a cache miss. The counters for the $victim\_candidates$ number of LRU cache lines are compared. The cache line with the lowest value becomes the candidate for eviction. The counter of this line is compared against a value called $bypass\_threshold$. If the value is above the threshold value, the incoming cache line is not stored in the last level cache, but is bypassed to the higher level cache (i.e. L1 and L2). In this case all cache blocks in the set are predicted to have higher temporal locality than the incoming cache line. Otherwise when a cache line is replaced, the cache line is marked as most recently used and the counter is initiated with a constant value $init\_counter$.

The scheme implements protection based on the following properties:

- The most recently used cache lines are never replaced. This implements a streaming buffer; before a cache line can be evicted it has to pass the buffer whose length is the associativity of the cache minus $victim\_candidates$.
- Cache lines that are accessed frequently are not replaced because their counters are increased by the hits. However since the *counters* are decremented for each access to the set, cache lines that are accessed frequently and used

recently will be kept compared to cache lines that are accessed frequently but not so recently.
– The algorithm has two different parameters for hits to cache lines, one for write and one for read (hit_{write} and hit_{read}). A higher value for write hits improves protection for dirty cache lines. This can reduce off-chip bandwidth usage as dirty cache lines introduce an extra memory access when evicted.

Tag	Cache line data	LRU bits	Counter
123	.....	1	42
456	.....	0	34
789	.....	2	44
120	.....	3	45

Table 1. Cache structure for new scheme.

To illustrate the algorithm, consider $bypass\_threshold = 100$ and $victim\_candidates = 2$. The first line that will be evicted in Table 1 is the cache line with tag 789. With $victim\_candidates = 3$, the cache line with tag 123 is evicted. With $bypass\_threshold = 43$ and $victim\_candidates = 2$ the incoming data will not be stored in the cache, but is bypassed to a higher level cache/processor.

The following parameters were found using a small training set and later used in all simulations of CMP configurations assuming a 16-way last-level cache: $init_{counter} = 11$, $hit_{read} = 60$, $hit_{write} = 61$, $bypass\_threshold = 140$, $victim\_candidates = 5$ and $maximum = 177$. They are not believed to be optimal, and further tuning might improve performance.

2.3 Implementation Cost

A straightforward implementation of the scheme would be to use one arithmetic unit for each cache line counter so they can all be decremented in parallel. However, there are ways to reduce the amount of added logic for the scheme. A reference value representing the zero level can be incremented instead of decrementing all the counters for each access. The true value for each counter is the difference between the stored counter value and the reference value. The counters should be extended with a bit or two in order to know when the counter has a negative value (otherwise a large negative number can be interpreted as a large positive value).

The scheme can be implemented even simpler by approximation. Instead of decrementing counters for all cache lines within the set, only one counter is decremented for each access to the set. The cache line can be selected in a round robin fashion, so for example for a four-way set-associative cache all counters are decremented after four accesses. This will reduce the number of bits that is needed for the counter and the size of the arithmetic logic.

2.4 Policy Switching

Since not all sets are represented in the shadow tags, a selection of the sets that should represent the cache must be done. We experimented with a randomly picked selection, selection based on prime numbers for the indexes and by simply choosing the sets with lowest indexes. The 16 sets with the lowest indexes were chosen (out of 2048) to represent the cache as it showed the highest performance in our simulations and is believed to have the simplest hardware realization. Hits and misses are counted for both the shadow tags and the sets with the matching index in the real cache. After a number of accesses (2000 accesses in our experiments) to these tags the performance is compared for the two different schemes. If the alternative scheme has fewest cache misses, the two schemes are exchanged. The counters are then reset and reevaluation is done after a number of accesses to these tags again.

3 Methodology

We use a cache simulator as well as a detailed pipeline-level simulator to get statistics on miss reduction and improvements in instructions-per-cycle (IPC), respectively. The models are part of SimpleScalar version 3 [1]. These models are extended to simulate the new scheme, Wong and Baer's scheme and an oracle scheme, and are extended to simulate CMP configurations.

The baseline parameters for the simulator are shown in Table 2. All of the *SPEC2000* benchmark applications were used with the reference data sets except for two: The simulator had compatibility problems with *vortex* and *sixtrack*, and they are not included in the experiments. We create multiprogrammed workloads for our CMP architecture as follows. In each experiment, four randomly picked applications are run in parallel. Since there can be several instances of the same application in an experiment, each application is randomly forwarded between 0.5 and 1.5 billion instructions before simulation of two hundred millions cycles.

3.1 Wong and Baer's Scheme

Wong and Baer proposed a scheme for improving last-level (L2) cache performance by extending the LRU policy with information about temporal locality in the references [16]. Cache blocks loaded from an instruction classified as non-temporal is replaced first. They investigated two strategies, a profile-based and run-time based, where the run-time showed best performance. The run-time approach uses a small *locality table* to keep track of instructions that load data with temporal locality. For the CMP configuration, we gave each processor its own locality table of 32k entries. When *switching* is applied and the scheme is deactivated, the locality table and locality bits of the cache are still updated, but the LRU cache line is always selected as victim.

Parameter	Value
Register Update Unit Size	128 instructions
Load Store Queue	64 instructions
Fetch queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Functional Units	4 INT ALUs, 4 FP ALUs, 1 INT Multiply/Divide, 1 FP Multiply/Divide
Branch Predictor	Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history table, 4K Chooser
Branch Target Buffer	512-entry, 4 way
Mispredict Penalty	7 cycles
L1 Instruction/Data Cache	64K, 2-way (LRU), 64 B Blocks, 2/3 cycle latency
L2 Instruction/Data Cache	128/256K, 4-way (LRU), 64 B Blocks, 7/7 cycle latency
L3 Cache	2 MByte unified, 16-way (LRU), 64 B Blocks, 19 cycle latency
Main Memory	250 cycles first chunk, 4 cycles inter chunk. Chunk size 8 bytes. 9 GBytes/s theoretical limit for 4.5 GHz processor
I-TLB/D-TLB (Translation Lookaside Buffers)	128-entry, fully associative, 30 cycles miss penalty
Chip multiprocessor	4 processors sharing the L3 cache

Table 2. Parameters for the simulated processors.

3.2 Oracle Scheme

An optimal replacement algorithm replaces the cache lines with the longest future reuse distance [2]. Consider a direct mapped cache which contains a block A. If Block B is requested, the question is whether A should be replaced or B bypassed. Our key to future knowledge is to postpone replacement decisions. B is put on a stack together with A. Each set has an independent FIFO stack for all recent accesses. If the next request is A, B was bypassed, or if the next request is B, B replaced A. By adjusting the size of the FIFO stacks, each contains hits to x different blocks (not counting multiple hits to the same block) and the intermediate misses, where x is the associativity of the cache. Requests found in the stacks are cache hits.

4 Evaluation

Several of the SPEC2000 applications have a small working set which more or less fits into the L1 and L2 cache. These applications are not sensitive to enhancements of the last-level (L3) cache and hence not relevant for the evaluation of the proposed scheme. Nevertheless, it is important that our proposed scheme is robust also for non memory-insensitive applications.

We first classify the applications with respect to their sensitivity to last-level cache performance. Then we consider the two schemes, include *switching*, consider *selective caching*, examine robustness and compare performance with an oracle scheme.

4.1 Classification of Workloads

The number of last level cache accesses is shown in Figure 1 for different SPEC2000 applications. We classify the applications as (a) either being last-level cache in-

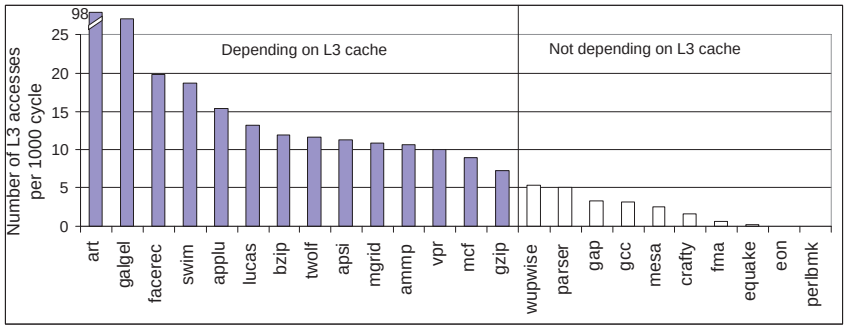


Fig. 1. Classification of applications based on number of access in L3 cache.

tensive or (b) not depending on the last-level cache. The applications with more than seven last-level cache accesses per thousand clock cycles are classified as last-level cache intensive. The rationale behind this is that there could potentially be a last-level cache miss every 150 clock cycles which add another few hundred cycles to the execution time. The number of cache misses per clock cycle could have been used to classify the applications as well, but then the applications that work well for conventional LRU would not have been included in the evaluation of the proposed scheme. If the new scheme degraded performance for these applications it might not have been detected.

4.2 Performance of the New Scheme

The speedup for the different schemes relative to a conventional shared cache is shown in Figure 2 based on 14 experiments with 4 random applications. Even

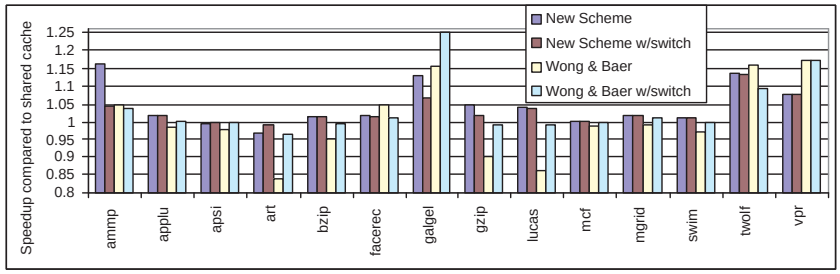


Fig. 2. The speedup of different schemes compared to shared cache.

though one application (*art*) has a slowdown of 3%, the rest of the applications have either equal or increased performance by the new scheme. *ammp*, *galgel* and *twolf* have improvements of 16%, 13% and 14% respectiviely. The average performance improvement is 5%.

4.3 Performance of the Wong and Baer's Scheme

The improvements for some of the applications with Wong and Baer's scheme are slightly better than for the new scheme with 17%, 16% and 16% for *vpr*, *twolf* and *galgel* respectively (see Figure 2). The problem is the instability of the scheme with degradation of 16%, 14% and 10% for *art*, *lucas* and *gzip* respectively. The average performance is therefore close to zero (0.3% speedup).

4.4 Impact of Policy Switching

The intention with switching is to stabilize the performance of the schemes, i.e. turn it off when it does not perform well. The single application with degraded performance (*art*) for the new scheme is stabilized with switching enabled (see Figure 2). However, the advantage of the new scheme is reduced. Due to the stable performance of the new scheme it does not improve performance to turn it off when performance has been low for some time because it is still likely that it will improve performance for the upcoming cycles.

The more unstable Wong and Baer's scheme benefits more from switching as turning it off avoids performance degradation in many cases. The average improvement with switching is 4% and very nice speedup for *galgel* of 25%. By enabling switching the performance degradation is reduced from 16% to 3% for *art*.

Switching enables the scheme with lowest sum of cache misses for all cores. This improves performance for applications which have many cache misses. These applications are the slowest and therefore switching improves harmonic mean quite well. Without switching the harmonic mean is reduced by 8% and 13% and with switching enabled the harmonic mean is improved by 4% and 13% for the new scheme and Wong and Baer's scheme respectively.

4.5 Selective Caching

In the cache miss model, the total number of cache misses that is reduced by selective caching on top of the improved replacement algorithm is 0.5% for the memory bound applications, and the total number of accesses to main memory is reduced by 1%. The parameter *bypass_threshold* for the algorithm is relatively high; most data will be stored in the last level cache. In other words, only in special cases all the 16 cache lines in the cache are predicted to have higher temporal locality than the data that are on the way from main memory. Therefore selective caching only slightly increases performance.

4.6 Robustness

The result of experiments that contain both last level intensive and non-last level cache intensive applications is shown in Figure 3 based on 40 experiments each with four randomly picked applications. Many of the involved applications hardly access the last level cache which increases the cache size for the other applications. The result is that there is no need to restrict cache usage since there is enough cache, and hence the new schemes do not improve performance much. The new scheme improves the average performance by 0.6% and the Baer and Wong scheme by 0.2% with switching and degrade the performance by 3% without switching. Even though *art* suffers from 7% degraded performance it can be said that both the new scheme and Wong and Baer's scheme with switching are robust without serious degradation of performance across all applications.

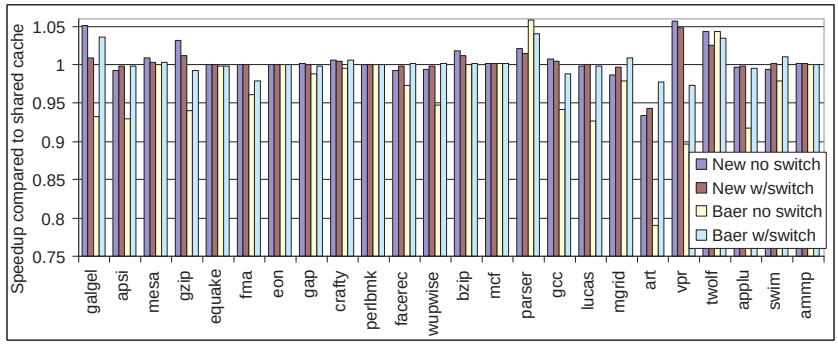


Fig. 3. Speedup compared to a conventional shared cache.

4.7 Potential for improvement

The effect on the number of cache misses for two experiments are shown in Figure 4. This figure is based on cache only simulation. A combination containing

applications with low cache miss rates are shown to the left and with high miss rates to the right. The Oracle scheme shows there is still potential for improvements. Approximately one third of the cache misses could have been avoided by a perfect replacement policy. Even though such a policy is not implementable, there is room for improvements.

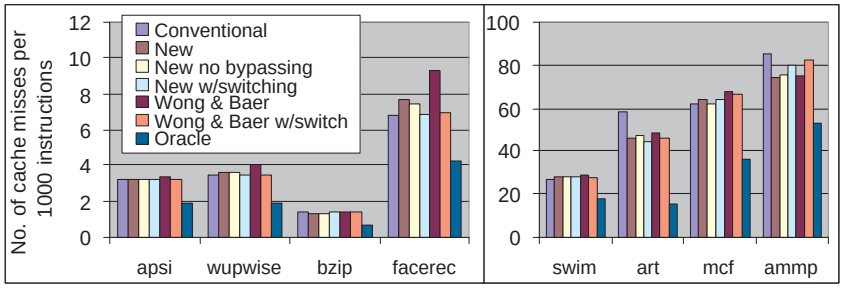


Fig. 4. Number of cache misses for four applications in the CMP configuration.

5 Related Work

Robinson and Devarakonda made an algorithm intended for file systems and databases [13] with a similar structure as the new scheme, but for a fully associative cache. They extend the LRU algorithm so not necessarily the LRU cache line is evicted, but the line with the lowest counter among the LRU cache lines. Access time for a disk is typically 200k times longer than for DRAM, and therefore their algorithm was implemented in software. The access pattern for the last level cache in a CMP configuration is different from the access pattern for a file cache, and therefore we apply a different heuristic to pick the cache line that is predicted not to have temporal locality.

Kharbutli and Solihin [8] use a counter to predict when cache lines have no more temporal locality and should be evicted. Two algorithms were presented. One is based on access interval: For each cache line, it records the number of accesses to the set where the line resides, between consecutive accesses to that line. The other is based on live time: For each cache line, it records the number of accesses to itself during the interval in which the line resides continuously in the cache. A table is used to store information about earlier access patterns. Their scheme is different than our new scheme since cache lines are categorized as either having temporal locality or not (when counter reaches maximum value), while in our scheme the cache lines are compared. Another difference is that our scheme consider frequency and recency of use, while they predict when cache lines loose their temporal locality based on repeating behavior patterns.

Bypassing has been studied earlier and the heuristics for bypassing have been based on the program counter of the instruction that accesses the cache [4, 9, 14] or by compiler analysis [3, 14]. None of these studies look at the replacement policy, and one reason is that most of them study direct mapped caches. With higher associativity and improved replacement policies, bypassing becomes less important. In our study bypassing counts for only a small percentage of the total performance increase.

Wang, McKinley, Rosenberg and Weems looked at compiler techniques for improving the replacement policy, and included a limited study with prefetching [15]. There are problems with a compiler approach. (1) The program has to be compiled for a specific hardware platform which makes the code less portable. (2) In CMP configurations the compiler will not have the information about the other applications that are sharing the same cache in contrast to a run-time implementation.

The number of conflict misses can be reduced by utilizing multi-lateral caches, i.e. several independent cache banks at the same level in the hierarchy. By splitting the cache into two independent caches, one larger cache with the data that are predicted to have temporal locality and one smaller cache for data with unknown or predicted low temporal locality, lines with long reuse distance do not displace lines with shorter reuse distance. Rivers and Davidson show that a smaller streaming (i.e. a FIFO) cache can be used to decide when data has short enough reuse distance to be inserted into a larger cache [10]. John and Subramanian use an annex cache [5], blocks are moved from the annex cache into the main cache after several hits. Other approaches for deciding whether data should be stored in the temporal or non-temporal cache is to use run time analysis based on the program counter of the instructions that access the cache [7] or by the addresses of the data that are accessed [6, 11]. We do not categorize data as either temporal or non-temporal, our scheme compares the predicted temporality between different cache lines and is therefore a different approach. Instead of protecting cache lines by storing them in separate caches, our scheme protects cache lines within the set by enhancing the replacement policy. Most of these papers focus on a low associativity first level cache for uniprocessor system, while we study a last-level 16-way set-associative cache for a CMP configuration.

6 Conclusion

Improving last level cache performance reduces off-chip bandwidth requirements and stalling of processor cores. We have improved the performance by augmenting the replacement algorithm to protect cache blocks that are likely to be reaccessed. This improves the performance of the system when there is competition for cache space and cores evict cache blocks for other cores. In cases where the cache space is large enough, the scheme does not degrade performance and can be said to be robust. The complexity of the scheme is low, and only eight bits are required for the heuristic per cache line. We have also studied the performance of Wong and Baer's scheme and found it to be unstable in CMP context; Sev-

eral applications were suffering. We extended the scheme so that it is switched off when the performance is not increased. This extension makes the scheme much more suitable. The oracle algorithm shows that there is still a significant potential for improvement in utilization of caches.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. IEEE Comp., V35I2, 2002.
2. L. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2), 1966.
3. Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. *Proceeding of the 22th Annual Hawaii Inter. Conf. on System Sciences*, 1989.
4. Haakon Dybdahl and Per Stenström. Enhancing lower level cache performance by early miss determination and block bypassing. *ACSAC*, 2006.
5. L.K John and A. Subramanian. Design and performance evaluation of a cache assist to implement selective caching. *Proc. of Intl. Conf. on Comp. Design, pp. 510-518*, 1997.
6. T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W Hwu. Run-time cache bypassing. *IEEE Trans. on Comput. V48I12*, 1999.
7. M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *Proc. of Computing Frontiers*, 2004.
8. M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. ICCD, 2005.
9. Scott McFarling. Cache replacement with dynamic exclusion. ISCA, 1992.
10. J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP, Vol. 1*, pages 154–163, 1996.
11. J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS '98*.
12. International technology roadmap for semiconductors. http://public.itrs.net/, 2005.
13. J.T. Robinson and M.V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.
14. G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to cache management. *Proc. of the 28th Annual Symp. on Microarchitecure*, 1995.
15. Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the compiler to improve cache replacement decisions. PACT, 2002.
16. W. A. Wong and J-L. Baer. Modified LRU policies for improving second-level cache behavior. *HPCA*, 2000.

A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors

Haakon Dybdahl, Per Stenström and Lasse Natvig

To be presented at International Conference on High Performance Computing (HiPC), Bangalore, India, 2006.

To be published by Springer
2006

A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors*

Haakon Dybdahl[1] Per Stenström[2] Lasse Natvig[1]

[1] Dept. of Computer and Information Science,Norwegian University of Science and Technology,N-7491 Trondheim, Norway, [dybdahl,lasse]@idi.ntnu.no
[2] Dept. of Computer Engineering, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Goteborg, Sweden, per@ce.chalmers.se

Abstract. Chip multiprocessors (CMPs) usually employ shared, last-level caches to use on-chip memory resources effectively. Unfortunately, conventional replacement policies applied to shared caches fail to partition memory resources among cores to achieve an optimal execution throughput. This paper presents a novel replacement policy that dynamically estimates how many misses would be eliminated if one more block per set would be allocated to a certain processor taking into account the extra misses for some other processor. Our implementation makes novel use of *shadow tags* for the estimation. We show that it can yield 50% higher execution throughput on a 4-way CMP and in contrast to previously proposed schemes, we did not observe any noticeable degradation of performance for any application in the SPEC2000 we used.

1 Introduction

The first levels of cache in a chip multiprocessor (CMP) are often private to each processor whereas the last-level cache is usually shared. A shared cache can shield the long (and increasing) memory latency more effectively as it leverages the sharing of data among cores. However, conventional replacement policies, such as LRU, blindly victimize blocks regardless of which processor it belongs to. As the following experiment reveals, this can lead to lower than optimal global performance.

Figure 1 shows cache behavior for a limited sample period for some benchmark applications from SPEC2000. For the *mcf* application, the accesses either result in a cache hit in the most recently used (MRU) cache block in the set, or the accesses cause a cache miss. Even though the simulated cache is 16-way, all hits are to the MRU block, and hence the other 15 blocks per set are not needed. However, in a sharing situation, all the cache misses will evict cache blocks for other processors. Putting a constraint on the number of cache blocks per processor for each set may prevent this from happening. *crafty* behaves similarly although it needs two cache blocks in each set. *gzip* and *gcc* do not need any

* This work is partly sponsored by the HiPEAC Network of Excellence funded by EU under FP6.

constraints as there are few misses. Since they are reusing the cache blocks, they do not evict cache blocks for other processors.

An LRU algorithm augmented with balancing of cache partitions between processors can clearly improve performance as this can restrict one processor from using too much cache space. Partitioning cache resources across processors for shared caches have been studied before [1, 2] with algorithms based on monitoring the number of cache blocks allocated to each processor in the whole cache. Taking *mcf* and *crafty* as examples, this approach can erroneously victimize the most recently used blocks in a set. In fact, we have noticed that it provides mixed performance results.

In this paper, we take a radically different approach by monitoring and restricting the number of cache blocks allocated to a certain processor for each *set*. Our cache partitioning aware replacement policy associates with each block the ID of the processor that fetched the block into the cache. Additionally, a shadow tag per processor is associated with each set. When a block fetched by a processor is evicted, it will be kept track of by the shadow tag for that processor and set. With this basic infrastructure, the replacement policy dynamically monitors how many misses could be avoided if a processor had one more block by simply counting the hits to the LRU block and to the shadow tag for each processor. The new replacement algorithm victimizes the block associated with a processor that needs fewer blocks in favor of a processor that needs more blocks. Suh et al. [3, 4] also use counters for choosing a victim but do not use shadow tags to estimate what would be the potential gain of allocating more blocks to a processor.

Our detailed evaluation, based on most of the applications in SPEC2000, reveals that the new policy can increase the execution throughput of independent applications run on each processor core by up to 50%. Moreover, we did not see any noticeable degradation in performance with respect to any application. We also found that making decisions on a per-set granularity usually works

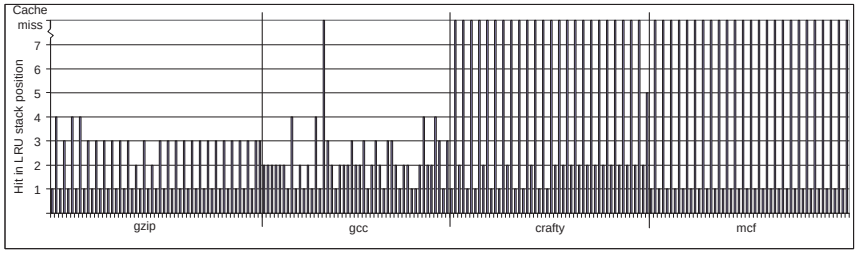


Fig. 1. The cache misses and hits in different LRU stack positions in the last-level cache for selected applications. Each access is either a hit in a position in the LRU stack or a cache miss (top line in the graph). Position 1 represents the most-recently-used (MRU) cache block in that set, position 2 is a hit in the second MRU position and so on.

better than making it on a per-cache granularity as previous work suggested. Additionally, our scheme can be implemented with modest storage overhead.

The new scheme is described in Section 2. Sections 3 and 4 present the evaluation methodology and the results, respectively. Related work is discussed in Section 5 and we conclude in Section 6.

2 The New Scheme

Earlier work on partitioning cache resources in shared caches have used the total number of cache blocks per processor as a parameter for sharing [2] or other parameters for each processor such as miss rate and IPC rate [1]. Even though we also consider the total number of blocks allocated per processor, we noticed, and show in this paper, that this provides mixed results. Therefore, we present in this section a novel technique that uses the number of blocks per processor for each set as a basis for replacement decisions.

The new replacement policy is based on two guiding principles. First, we use the notion of overbooking of resources to increase utilization of all blocks in a set. Therefore, the number of blocks allocated to processors per set is larger than the number of blocks in the set. Second, the cache partitioning aware replacement policy aims at maximizing the total throughput, i.e. the number of instructions committed per time unit, by minimizing the total number of cache misses. If the total throughput is expected to increase by reducing the size of the partition for one processor and increasing it for a different processor, the change is done. In the next four sections, we present the infrastructure needed (Section 2.1), the replacement policy (Sections 2.2 and 2.3), and an analysis of the implementation costs (Section 2.4).

2.1 Structure

The hardware structures needed for the new scheme for a four core CMP are shown in Figure 2. Each cache block is extended with *processor identification* as shown in Figure 2(a). This field is updated with the value from the requesting processor every time a cache block is installed in the cache. When a cache block is evicted, the tag of the block is stored in the shadow tag table for the processor that fetched the block, see Figure 2(b). The last block that was evicted for processor 2 in set 1 is the block with tag *f*. Accesses that miss in the cache, but have a tag match in the shadow tag table, would hit in the cache if the partition had one more block in this set. This event is counted by the shadow tag's hit counter, see Figure 2(c). For example if processor 1 requests the block with tag *a* in set 0, the counter for the shadow tag's hits will increase from 10 to 11. The other counter, *hits in the LRU blocks*, is increased when a request hits in the LRU block for the involved processor in the cache. This number represents the increase in number of misses as a result of reducing the cache size by one block per set.

Index	Tag	LRU data	Cache line data	Processor ID
..	..	..	..	..

(a) Each cache block is extended to include processor ID.

Set number	Processor 1	Processor 2	Processor 3	Processor 4
0	*a*	*b*	*c*	*d*
1	*e*	*f*	*g*	*h*
..	..	..	..	..

(b) This is the structure with the shadow tags. Each cache set has one shadow tag per processor.

Counter	Processor 1	Processor 2	Processor 3	Processor 4
Hits in the LRU blocks	2	3	2	3
Hits in the shadow tags	10	11	9	2

(c) There are two global counters per processor.

Description	Processor 1	Processor 2	Processor 3	Processor 4
Max. no. of blocks in set	2	3	2	3

(d) The partitioning parameters (one per processor) used by the replacement policy.

Fig. 2. The extra storage requirements for the new scheme.

A constraint is associated with each processor that limits the maximum number of blocks that can be in each set, see Figure 2(d). These values are used by the replacement algorithm to select a cache line for eviction.

2.2 The Partitioning Aware Replacement Policy

Algorithm 1 describes the new replacement policy for sharing cache space with the constraints from Figure 2(d). The search for a victim block starts at the bottom of the LRU stack (step 2) and steps the LRU stack towards the MRU block. If the processor that owns the cache block has too many cache blocks within the set (step 4), this block is chosen for eviction (step 5). If no block is found, the LRU cache block is evicted (step 8).

Algorithm 1 Pseudo code for finding a block for eviction. The function returns the position of the block to evict.

```
1: function FIND BLOCK TO EVICT
2:     for LRU_stack_pos ←number of blocks per set, 1 do
3:         proc_id ← get processor that owns block(LRU_stack_pos)
4:         if max no of blocks in set[proc_id] < no of blocks in set(proc_id) then
5:             return LRU_stack_pos
6:         end if
7:     end for
8:     return position of LRU block
9: end function
```

2.3 Balancing the Cache Partition Sizes

The algorithm reevaluates the partition sizes per processor (see Figure 2(d)) on a regular basis. In our experiments we use 2000 cache misses in the last-level cache to trigger a reevaluation. The processor with the highest gain for increasing the cache size, i.e. the processor with most hits to its shadow tags (see Figure 2(c)), is compared to the processor with the lowest loss of decreasing cache size, i.e. the processor with the fewest hits to its LRU block. If the gain is higher than the loss, one cache block (per set) is given to the processor with the highest gain. The counters are reset after each reevaluation period.

In the initial partitioning, the total cache capacity is shared equally among all processors. Additionally, each processor receives two extra blocks per set. This *overbooking* of resources provides slack in case processors do not distribute their data uniformly.

A different approach used by Kim et al. [1] is changing the partitioning and measuring the performance difference, and then roll back if the performance was not increased. This might work for a two processor CMP, but complexity and uncertainty grow fast with increasing processor count since all processors influence the performance of each other. We have therefore not perused this technique.

2.4 Implementation Cost

The storage requirement for the new scheme used with the architecture presented in the evaluation section is 152 kbit. This is an increase of 0.5% in the storage requirement for the last-level cache. The storage is used for shadow tags (16%) and processor IDs in the blocks (84%). The evaluation section presents a CMP architecture with 4 processors, 4 MByte 16 way last-level cache with 4096 sets and 24 bits tag (we assume a 32 bit architecture). The shadow tags require $s*p*t$ bits where s is number of sets, p is number of processors, t is bits per tag, and for our architecture this is 384 kbit. However, shadow tags are not needed for all sets as shown in later in Section 4.4. We find that monitoring only 6% of the sets is sufficient for estimating cache sizes with almost no degradation of performance, and the number of bits is reduced downto 24 kbit. The field for the ID of the processor that fetched the block requires $\log_2 p$ bits per block, and this ID is required for every block. Our architecture with $4096 * 16$ blocks requires totally 128 kbit for this field. Finally, the storage for the two counters and one register per processor, sums up to 96 bits ($p * 3 * w$) if each register/counter (w) is 8 bit.

Most of the required logic is simple and can be allowed to be rather slow since accesses to main memory take hundreds of clock cycles. The fastest logic required is for indicating hits in the LRU block which requires access to the processor ID of all blocks in the set. The cycle time for this logic has to be as fast as a cache hit, but the logic can be pipelined (there is no need to immediately update the counter).

The physical placement of the shadow tags could be close to the cache tags. However, the latency requirement for these tags is more relaxed than for the cache tags. The shadow tags can therefore be moved further away from the processor and may operate at a lower voltage and hence consume less power.

3 Methodology

Simulation is used to compare the efficiency of the new scheme with a conventional LRU scheme and to earlier cache partitioning methods. The simulated architecture is shown in Figure 3. The baseline chip multiprocessor (CMP) architecture has four processors that share the last-level cache. We use a detailed

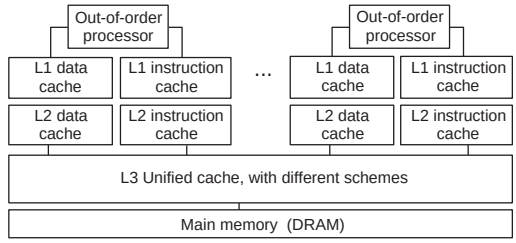


Fig. 3. Simulated architecture.

pipeline-level clock cycle-accurate out-of-order execution model simulator with non-blocking caches to get statistics on improvements in instructions-per-cycle (IPC). The model is based on SimpleScalar version 3 [5], but is extended to simulate the new schemes and CMP configurations including congestion to main memory.

The baseline parameters for the simulator are shown in Table 1. All of the *SPEC2000* benchmark applications were used as workload with the reference data sets except for two. The simulator had compatibility problems with *vortex* and *sixtrack*, and they are not included in the experiments.

We create multiprogrammed workloads for our CMP architecture as follows. In each experiment, four randomly picked applications are run in parallel. Each application is randomly forwarded with cache system enabled between 0.5 and 1.5 billion instructions and then two hundred millions cycles are simulated.

4 Evaluation

Several of the SPEC2000 applications have a small working set which more or less fits into the L1 and L2 cache. These applications are not sensitive to enhancements of the last-level (L3) cache. The goal of our scheme is to improve performance for the applications which are sensitive to the performance of the

Table 1. Parameters for the simulated processors.

Parameter	Value
Register Update Unit Size	128 instructions
Load Store Queue	64 instructions
Fetch queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Functional Units	4 INT ALUs, 4 FP ALUs, 1 INT Multiply/Divide, 1 FP Multiply/Divide
Branch Predictor	Comb., Bimodal 4K table, 2-Level 1K table, 10-bit history table, 4K Chooser
Branch Target Buffer	512-entry, 4 way
Mispredict Penalty	7 cycles
L1 Instruction/Data Cache	64K, 2-way (LRU), 64 B Blocks, 2/3 cycle latency
L2 Instruction/Data Cache	128/256K, 4-way (LRU), 64 B Blocks, 7/7 cycle latency
L3 Cache	4 MByte unified, 16-way (LRU), 64 B Blocks, 19 cycle latency
Main Memory	250 cycles first chunk, 4 cycles inter chunk. Chunk size 8 bytes. 9 GBytes/s teoretical limit for 4.5 GHz processor
I-TLB/D-TLB	128-entry, fully associative, 30 cycles miss penalty
Chip multiprocessor	4 processors sharing the L3 cache

last-level cache. Additionally, the scheme should be robust and should not degrade the performance for any application. We first classify the applications with respect to their sensitivity to last-level cache performance in Section 4.1. We then consider the performance improvements and robustness of the new scheme in Sections 4.2 and 4.3, respectively. The last part of the evaluation is concerned with sensitivity analysis and comparisons with related schemes.

4.1 Classification of Workloads

The number of cache accesses to the last-level cache is generated by forty experiments with random configurations of four applications. The numbers of accesses per application for all experiments are averaged and shown as a logarithmic plot in Figure 4. We classify the applications as (a) either being last-level cache intensive or (b) not depending on the last-level cache. The applications with more than 2 million last-level cache accesses are classified as last-level cache intensive. The number of cache misses could have been used to classify the applications as well, but then the applications that work very well for conventional LRU would not have been included in the evaluation of the new scheme. If the new scheme degraded performance for these applications it might not have been detected.

4.2 Throughput Improvements

This subsection evaluates the throughput improvements (i.e. total number of instructions committed) for the last-level cache intensive applications found in the previous section. The configuration is a four processor CMP. Each processor has 200 million clock cycles simulated (the clock is synchronized with the other processors) after warm up. Sixty different experiments were run with the last-level cache intensive applications. Each application is represented 18 times on average in these experiments. The total number of instructions committed for each application for the new scheme divided by the total number of instructions

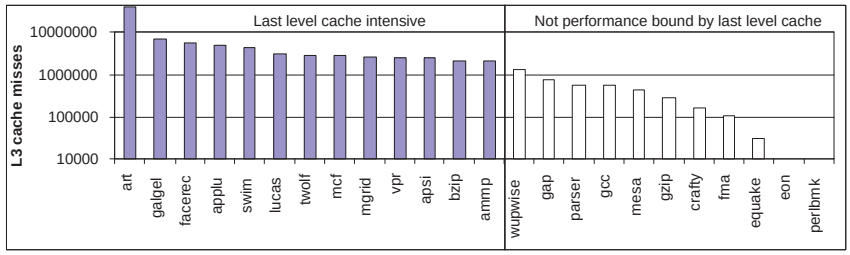


Fig. 4. Average number of last-level cache accesses for each application in a logarithmic plot.

committed by the conventional scheme is shown in Figure 5. Positive speedups (i.e. > 1) are shown for all of the applications and the best numbers are for *vpr*, *twolf* and *art* with speedups of 1.51, 1.30 and 1.29 respectively. Not all applications benefit from slightly larger caches which is one reason why not all applications run much faster with the new scheme.

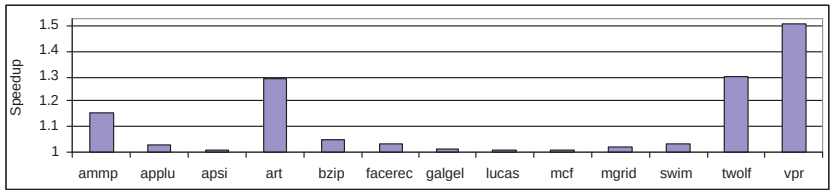


Fig. 5. Speedups for last-level cache intensive applications.

Instead of considering improvements per application, improvement per experiment (that is four randomly picked applications run together) is shown in Figure 6 as *average speedup*. The total number of committed instructions for all experiments is increased by 7% for the new scheme compared to the conventional scheme. All experiments have a performance gain, i.e. the sum of committed instruction for the four benchmarks run in parallel is increased. This shows that the new scheme provides a robust and improved performance across different experiments.

A computer system is often bound by the slowest running application. In these cases, the harmonic mean is more important than the average mean [6]. The *harmonic speedup* in Figure 6 is the harmonic mean of the four applications with the new scheme divided by the harmonic mean of the same four applications with the conventional scheme. The highest speedup on the right hand side of the graph is 2.0. This shows that the new scheme is not only increasing performance for the fastest running applications, but also improves performance for the slowest

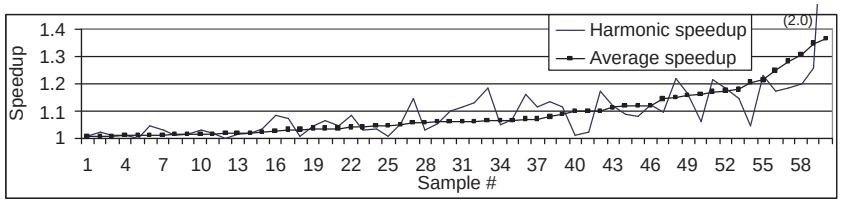


Fig. 6. The speedup for each experiment summing up all instructions committed per core in the CMP. The applications used are the last-level cache intensive category. The results are sorted on average speedup.

running applications. This is not surprising since the goal of the scheme is to reduce the total number of cache misses and the slowest running application often has most cache misses.

4.3 Robustness

Even though the new scheme works well for the last-level cache intensive applications, the new scheme should not degrade performance when other applications are included. Including applications that are not that memory intensive results in less stress on the last-level cache because these applications do not access the L3 frequently. In some of the experiments this means inserting sharing constraints in a system that should not have any constraints because none of the processors are using too much last-level cache space. However, as shown in Figure 7, the new scheme works impressively well in this setting. The figure shows speedups when combining both categories of the SPEC2000 benchmarks. Even though there are some degradation of performance of about 1% for *facerec* and 0.5% for *wupwise*, speedups of 13%, 10%, 5% and 4% are provided for *twolf*, *vpr*, *parser* and *art* respectively.

The increase in number of committed instructions per experiment is shown in Figure 8. Even though there are some experiments with fewer committed instructions, in most cases the performance is improved. The total number of committed instructions is 0.8% higher with the new scheme compared to a conventional architecture. By comparing this graph to the graph for the applications which are intensive to the last-level cache (Figure 6), we see that the performance gain is now reduced. This is due to the lower competition of the cache since many applications do not require much cache space in L3 and hence there is less room for improvement. The bandwidth required in and out of the chip is also reduced and the effect of this bottleneck becomes less significant.

4.4 Reducing Number of Shadow Tags

The experiments in the previous sections were done with four shadow tags for every set to predict marginal gains of increasing cache size for the processors.

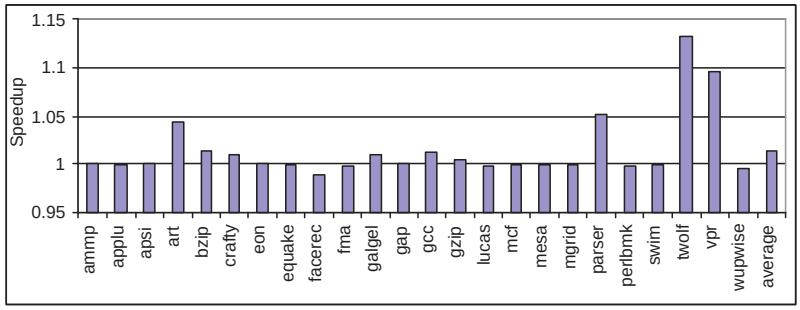


Fig. 7. Total number of instructions committed per application with the new scheme divided by the number of instructions committed per application with a conventional scheme.

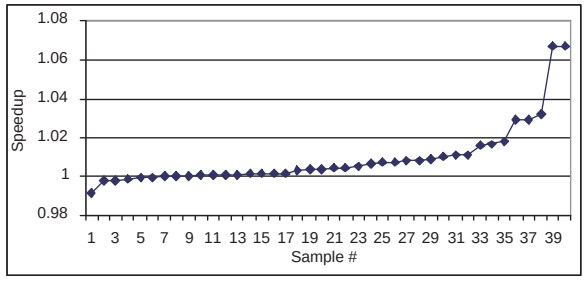


Fig. 8. The speedup for each experiment summing up all instructions committed per core in the CMP. Combination of both categories SPEC2000 applications are used.

However, it is not necessary to implement shadow tags in all sets. Previous work has revealed that monitoring the sets with the lowest index works well and better than randomly generated subsets or subsets based on prime numbers [7]. The result of monitoring 1/16 of the sets with lowest index is shown in Figure 9. As shown, the median performance is slightly increased by only having shadow tags

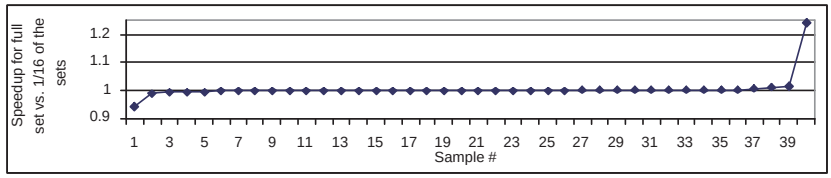


Fig. 9. The speedup of monitoring all sets vs. monitoring only 1/16 of the sets.

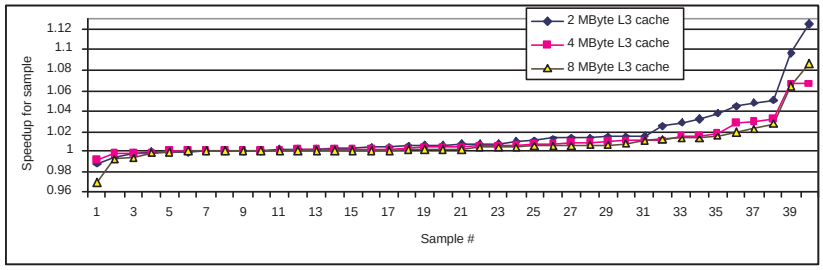


Fig. 10. The speedup for the new scheme with different L3 cache sizes. Combinations of both categories SPEC2000 applications are used as workload. Each line in the graph is sorted independently.

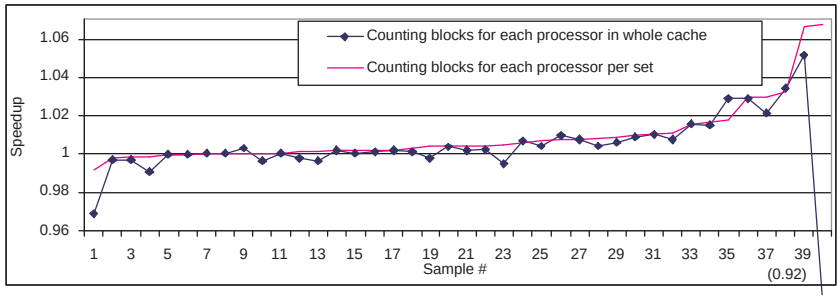


Fig. 11. Two different replacement policies: Counting blocks in the whole cache per processor vs. per set.

in a subset of all cache sets. We conjecture that it is due to the higher contention and congestion in the sets with the lowest number. This makes monitoring these sets more important, and hence improves performance. There is one experiment however where reducing the number of shadow tags causes a penalty of 24%. The total number of committed instructions are reduced by 0.2% for all experiments when reducing the number of shadow tags. LRU hits are counted for in all sets, but the numbers are normalized when compared to shadow tag hits. The cost of monitoring only $1/16 \approx 6\%$ of the sets is not very high and is discussed in Section 2.4.

4.5 Cache Size Sensitivity

The speedup of the new scheme when using both categories SPEC2000 applications for 2, 4 and 8 MByte L3 caches is shown in Figure 10. We see a higher gain for a 2 MByte than for a 4 MByte cache. This is due to more conflicts in a smaller cache and more room for improvements. For the 8 MByte cache the gains are slightly lower, as expected, and some experiments show degradation of

performance of up to 3%. An 8 MByte cache is quite large for the SPEC2000 applications making constraints less effective.

4.6 Comparison with Earlier Work

Different Granularity of the Cache Constraints Earlier attempts to partition the cache dynamically have been partitioning the cache globally and counted the total number of blocks used per processor [1, 2]. In the new scheme we count the number of blocks per processor *per set* when finding the block for eviction. In Figure 11 the new scheme is compared to a modified new scheme where counting is done globally. Lower performance is seen when counting globally for a configuration with a 16 way cache. Experiments with a 4 way cache result in lower performance for the new scheme. This is because there is no room for constraints with so few blocks in each set.

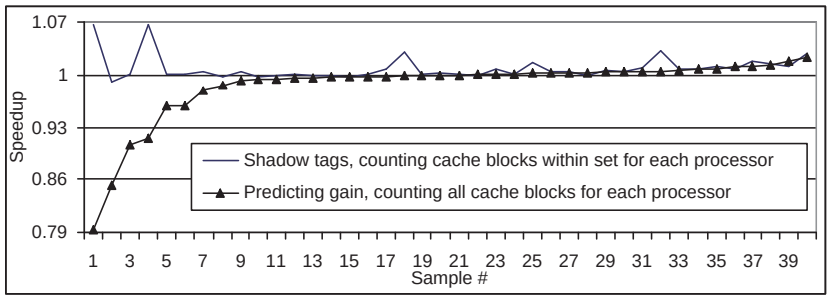


Fig. 12. Two different schemes: (a) Predicting the marginal gains with counting blocks in the whole processor and (b) the new scheme (shadow tags and counting blocks within set).

Not Using Shadow Tags Earlier work did not use the extra shadow tags for calculating gains. Suh and Rudolph have looked at dynamic cache partitioning [2], and instead of shadow tags they used counters for hits to the LRU and second LRU cache blocks, and a formula for estimating the marginal gains: $M(p) = 2 * H(p) - Q(p)$, where $M(p)$ is the marginal gain for increasing the cache size for processor p with one block per set, $H(p)$ is the number of hits in the LRU cache blocks for all sets and $Q(p)$ is the number of hits in the second LRU cache blocks in all sets. The results from this approximation combined with counting globally are shown in Figure 12. Even though this approximation shows speedup for some experiments, the overall result is a degradation of performance. The figure includes a graph for the new scheme, and as shown the new scheme results in a much more stable performance for the same experiments.

5 Related Work

Chishti et al. proposed a scheme where cache blocks evicted for one processor can be stored in the cache storage for other processors if free space is available [8]. Zhang and Asanovic [9], and Chang and Sohi [10] evaluate a combination of shared and private caches. The goal is to have the size of the shared cache available to all processors with the speed of the local cache. None of these studies have provided constraints on capacity usage as in our scheme nor have they considered the marginal gains/loss for different cache partitioning.

6 Conclusion

Conventional chip multiprocessors do not consider cache usage per processor when deciding which cache block to evict. The scheme proposed in this work establishes constraints on the number of cache blocks in each set that each processor can allocate. We have shown that our new scheme outperforms the LRU replacement policy in almost every case and hence increases performance compared to conventional architectures. Compared to previous work it has much better robustness. This is due to two contributions: (a) our use of shadow tags which is an accurate measure of increased performance for increased cache sizes and (b) putting constraints within each single set instead of across all sets.

Methods for increasing the overall performance can lead to starvation for the slowest processors. i.e. the processors with highest miss rate. However, since the goal of the scheme is to minimize the total number of cache misses, this does not happen with our scheme.

References

1. Kim, S., Chandra, D., Solihin, Y.: Fair cache sharing and partitioning on a chip multiprocessor architecture. PACT (2004)
2. Suh, G., Devadas, S., Rudolph, L.: Dynamic cache partitioning for simultaneous multithreading systems. IASTED Parallel and Dist. Computing Systems (2001)
3. Suh, G.E., Devadas, S., Rudolph, L.: A new memory monitoring scheme for memory-aware scheduling and partitioning. HPCA (2002)
4. Suh, G.E., Devadas, S., Rudolph, L.: Dynamic partitioning of shared cache memory. The Journal of Supercomputing, Vol 28, No 1 (2004)
5. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. IEEE Computer, Volume 35, Issue2 (2002)
6. Smith, J.E.: Characterizing computer performance with a single number. Communications of the ACM **31**(10) (1988) 1202–1206
7. Dybdahl, H., Stenström, P.: Enhancing lower level cache performance by early miss determination and block bypassing. submitted to ICCD (2006)
8. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Optimizing replication, communication, and capacity allocation in CMPs. SIGARCH Comput. Arc. News **33**(2) (2005)
9. Zhang, M., Asanovic, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: ISCA. (2005)
10. Chang, J., Sohi, G.S.: Cooperative caching for chip multiprocessors. ISCA (2006)

An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors

Haakon Dybdahl and Per Stenström

To be presented at International Symposium on High-Performance Computer Architecture-13 (HPCA-13), Phoenix, USA, 2007.

To be published by IEEE
2007

An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors*

Haakon Dybdahl[1] Per Stenström[2]

[1] Dept. of Computer and Information Science,Norwegian University of Science and Technology,NO-7491 Trondheim, Norway, dybdahl@idi.ntnu.no
[2] Dept. of Computer Engineering, Dept. of Computer Engineering, Chalmers University of Technology, SE-412 96 Goteborg, Sweden, per@ce.chalmers.se

Abstract. The significant speed-gap between processor and memory and the limited chip memory bandwidth make last-level cache performance crucial for future chip multiprocessors. To use the capacity of shared last-level caches efficiently and to allow for a short access time, proposed non-uniform cache architectures (NUCAs) are organized into per-core partitions. If a core runs out of cache space, blocks are typically relocated to nearby partitions, thus managing the cache as a *shared cache*. This uncontrolled sharing of all resources may unfortunately result in pollution that degrades performance.
We propose a novel non-uniform cache architecture in which the amount of cache space that can be shared among the cores is controlled dynamically. The adaptive scheme estimates, continuously, the effect of increasing/decreasing the shared partition size on the overall performance. We show that our scheme outperforms a private and shared cache organization as well as a hybrid NUCA organization in which blocks in a local partition can spill over to neighbor core partitions.

1 Introduction

Two important challenges for next generation microprocessors are the slow main memory and the limited off-chip bandwidth. Efficient management of the last-level on-chip cache is therefore important in order to accommodate a larger number of cores in future multi-core architectures.

A last-level multi-core cache can be organized as private partitions for each core or having all cores sharing the entire cache. The shared cache organization can be utilized more flexibly by sharing data between cores. However, it is slower than a private cache organization. In addition, private caches do not suffer from being polluted by accesses from other cores by which we mean that other cores displace blocks without contributing to a higher hit rate.

* Per Stenstrom is a member of the HiPEAC Network of Excellence funded by EU under FP6.

Non-uniform cache architectures (NUCA) are a proposed hybrid private/shared cache organization that aims at combining the best of the two extreme organizations [2,3,5,8,9,16] by combining the low latency of small (private) caches with the capacity of a larger (shared) cache. Typically, frequently used data is moved to the shared cache portion that is closest to the requesting core (processor); hence it can be accessed faster. Recently, NUCA organizations have been studied in the context of multi-core systems as a replacement for a private last-level cache organization [3, 6]. The cache is statically organized into private partitions but a partition attached to one core can also keep blocks requested by other cores. When a block is installed in a certain partition, a replaced block from that partition will be installed in a neighbor's partition, picked by random. As a result, on a miss in one partition, all other partitions are first checked before accessing main memory. While this hybrid scheme provides fast access to most blocks, it can suffer from *pollution* because of the uncontrolled way by which partitions are shared among cores.

Our contribution in this paper is a novel NUCA design for multi-cores based on private partitioning in which the sizes of the core-local partitions that are shared are chosen adaptively to maximize the overall performance. We show in the paper that our adaptive scheme outperforms the uncontrolled sharing of private partitions in [3] which is prone to pollution effects.

Our work is inspired by earlier work on dynamic partitioning of the resources in *shared caches* among cores [7, 10, 13]. In the NUCA setting, the new issue becomes how to select the *size of the shared partition* which is not addressed in the earlier work. We combine and extend several existing mechanisms intended for solving problems in other contexts. The contribution of this paper is the unique combination and usage of these mechanisms and the novel architecture for the last-level cache.

In our organization, hits to the private partitions are fast, while hits to neighboring partitions are slower. The size of the private partition is dynamically controlled and balanced against the other cores. Cores that can best utilize the cache get more private cache space, but the private cache space is never larger than the local last-level cache. The cache usage in the shared partition is controlled as well. The size of the private partitions for each core is dynamically adjusted to minimize the total number of cache misses by determining the change in the total miss rate.

We compare the performance of the new scheme with the performance of private and shared cache organizations. Additionally, we also compare it with the NUCA scheme proposed by Chang and Sohi [3] in which essentially all last-level cache resources can be shared. The new scheme outperforms all these schemes. Our simulations show that we can improve the performance by more than 20% for the memory-intensive applications in the SPEC2000 suite compared to private caches.

We describe the new scheme in Section 2. Sections 3 and 4 then present the evaluation methodology and the results, respectively. Related work is discussed in Section 5 and we conclude in Section 6.

2 The Adaptive Partitioning Scheme

An architectural framework for a four-core system with the proposed scheme is shown in Figure 1. Note that this is a conceptual view rather than implying a physical layout. Each core has three levels of cache, where the L3 cache is the last-level cache. The *sharing engine* implements sharing of the last-level cache among the cores. Hits in the core-local L3 cache partition are faster than the hits in the neighboring last-level cache partitions.

The L3 cache usage is controlled in two ways: (a) a part of the cache is private and inaccessible by the other cores, (b) the size of the private cache partition and the number of cache blocks in the shared partition of the cache are controlled on a per-core basis in order to minimize the *total* number of cache misses. The division of private and shared partitions is conceptually shown in Figure 2. Each core has its own local cache with two partitions: A private partition for its own cache blocks and a shared cache partition intended for all the cores. Each set is divided among the four cores. The most recently used cache blocks for each core are stored in the private (and fast) partition of the set. Looking for a tag match in the set is a two phase process. First the tags in the private cache partition are checked then, if there is no match, the rest of the set is checked.

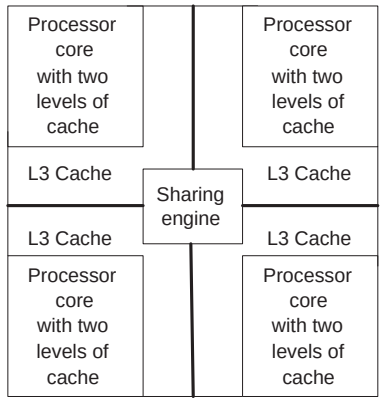


Fig. 1. Conceptual view of the on-chip architectural framework (not floorplan).

The sharing engine depicted in Figure 1 consists of several components: (a) a method for estimation of the best private/shared partitioning of caches, (b) a method for sharing the cache and (c) a replacement policy for the shared cache space. These components and the necessary structures are described in the following subsections.

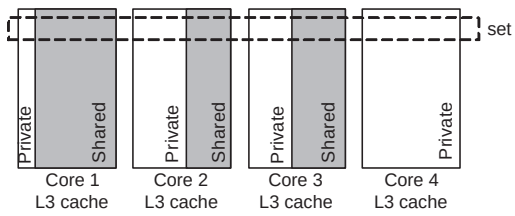


Fig. 2. Sharing of the L3 cache. Each local cache has a private and potentially a shared partition.

2.1 Estimation of Partition Sizes

Our scheme adapts the size of the private partitions by essentially increasing or decreasing the number of blocks per set for each private partition but keeping the number of sets fixed. Obviously, the potential benefit of balancing the cache partitions depends on the cache-size sensitivity of the applications that are running. For illustrative purposes, we show the cache-size sensitivity for five applications in Figure 3 for a fixed snapshot (in processor cycles) of the entire execution. The graph shows the number of cache misses per application as a function of the number of cache blocks per set but with a fixed number of sets for the cache.

For *mcf*, the innermost graph, only a single block per set is required. There is no benefit from increasing the number of blocks per set as the remaining misses are likely cold misses. On the other hand, *gzip* requires four blocks per set to avoid most misses.

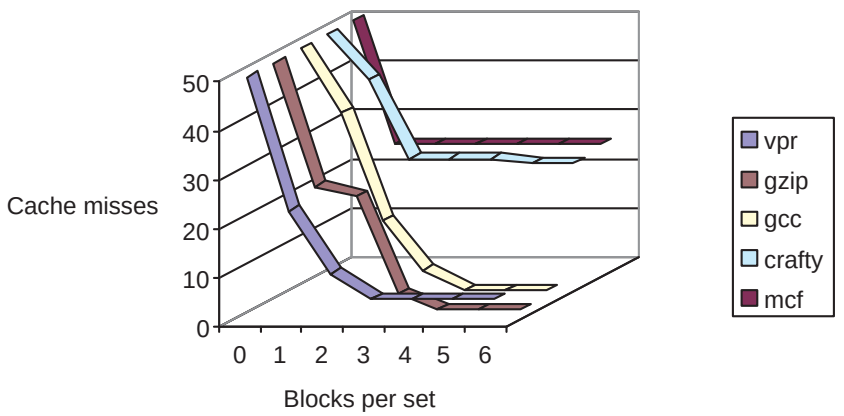


Fig. 3. Number of misses as a function of number of blocks per set.

Our scheme adjusts the shared partition size if the total number of misses is expected to go down if one core gives up a block to the benefit of another. For example if *mcf* and *gzip* both had three blocks per set it would make sense to change the partition so that *mcf* had two blocks and *gzip* had four blocks per set since the total number of cache misses is then reduced (see Figure 3). Adjusting partition sizes dynamically requires two methods: one for estimating the gain of increasing the cache size and one for estimating the loss of decreasing the cache size per application/core. We present the methods chosen next.

The method for estimating the number of cache misses that would have been avoided if the cache size was increased with one block per set is implemented as follows. Each set has a register for each core that is referred to as *shadow tag* according to Figure 4(b). When a cache block is evicted from the L3 cache, the tag of the block is stored in the *shadow tag* associated with the core that fetched the block into cache. On a cache miss, the tag of the miss is compared to the *shadow tag* for that set. If there is a match, the counter *hits in shadow tags* for the requesting core is increased. This counter is shown in Figure 4(c).

The method for estimating the number of increased cache misses as a result of decreasing the cache size with one block per set is derived from Suh et al. [13] and works as follows. If there is a hit in the LRU block for the requesting core, a counter is increased for that core. This counter (see Figure 4(c)) represents the number of cache misses that would have occurred if the cache size is reduced by one block per set.

The algorithm re-evaluates the private partition sizes per core on a regular basis. In our experiments, we use 2000 cache misses in the last-level cache to trigger a re-evaluation. This period is long enough to measure cache sensitivity and short enough to make the scheme dynamic. The core with the highest gain for increasing the cache size, i.e. the core with most hits to its shadow tags, is compared to the core with the lowest loss of decreasing the cache size, i.e. the core with the fewest hits to its LRU blocks. If the gain is higher than the loss, one cache block (per set) is provided to the core with the highest gain. The counters are reset after each re-evaluation period.

In the initial partitioning, 75% of the local cache partition acts as a private cache whereas 25% is a contribution to the shared partition.

2.2 The Structures

The hardware structures needed for the new scheme for a four core multi-core chip are shown in Figure 4. Even though the structures are shown as tables, the physical layout can be divided and distributed among the cores. Each cache block is extended with a *core identification* as shown in Figure 4(a). This field is updated with the value from the requesting core every time a cache block is installed in the cache. When a cache block is evicted from the last-level cache, the tag of the block is stored in the shadow tag table for the core that fetched the block, see Figure 4(b). The last block that was evicted for core 2 in set 1 is the block with tag *f*. Accesses that miss in the cache, but have a tag match in the shadow tag table, would hit in the cache if the private partition had one

Index	Tag	LRU data	Cache line data	Core ID
..	..	..	..	..

(a) This is an example of a cache block. Each cache block is extended to include the core ID.

Set number	Core 1	Core 2	Core 3	Core 4
0	*a*	*b*	*c*	*d*
1	*e*	*f*	*g*	*h*
..	..	..	..	..

(b) This is the structure with the shadow tags. Each cache set has one shadow tag per core.

Counter	Core 1	Core 2	Core 3	Core 4
Hits in the LRU blocks	2	3	2	3
Hits in the shadow tags	10	11	9	2

(c) There are two global counters per core.

Description	Core 1	Core 2	Core 3	Core 4
Max. no. of blocks in set	2	3	2	3

(d) The partitioning parameters (one per core) used by the replacement policy.

Fig. 4. The extra storage requirements for the new scheme.

more block in this set. This event is counted by the shadow tag's hit counter according to Figure 4(c). For example if core 1 requests the block with tag *a* in set 0, the counter for the shadow tag's hits will increase from 10 to 11. The other counter, *hits in the LRU blocks*, is increased when a request hits in the LRU block. This number represents the increase in number of misses as a result of reducing the cache size by one block per set.

A constraint (*max. no. of blocks per set*) is associated with each core that limits the maximum number of blocks that can be in each set as shown in Figure 4(d). This value reflects the total maximum number of blocks in each set for both the private partition and the shared partition. If the value is larger than the maximum size of the private partition (i.e. the associativity of the local cache) the core can use the shared space in addition to the private partition. If the value is smaller than the maximum size of the private partition, some cache blocks in the private partition are shared. In this case the core is still allowed to allocate *one* block in the shared partition. This increases the cache space flexibility and utilization. In such cases, the allocated block will be a candidate for eviction by the replacement policy for the shared partition.

2.3 Management of the Partitions

Each L3 cache is divided into a private and a shared partition as shown in Figure 2. The private partition is not shared and is managed by a least recently

used (LRU) replacement policy. To locate a block in the cache, the partitioning does not matter. However, to find a victim using LRU, the private partition is only considered and blocks belonging to the shared partition are not involved. Compared to a conventional LRU algorithm this requires that only a part of the blocks in a set is affected by an eviction. In our evaluation we use a 4-way private cache so there are only four different cases to consider and the amount of extra logic should hence be small.

In order to describe the behavior of the entire cache scheme, let us consider the key events:

- **Cache hit in private portion of L3 cache.** The block that is hit is moved to the top of the LRU stack, and the others are moved down. No access to the shared partition of the cache set is required.
- **Cache hit in neighboring L3 cache.** Before this happens, a miss occurred in the private partition of the last-level cache. Then all the neighboring caches are checked in parallel since the cache block can be in any of these caches. The cache block with the hit is moved to the local cache. The LRU cache block in the private cache replaces the block with a hit in the shared cache, and the block is set as MRU in the shared cache.
- **Cache miss.** The block is requested from main memory and inserted into the private cache. The LRU block in the private partition of the cache is inserted into the shared partition of the cache. The block that is evicted is found according to Algorithm 1. We describe the algorithm in the next subsection.

2.4 The Replacement Policy

On cache misses, data loaded from main memory is always allocated in the private partition of the cache as most recently used. This normally requires that one block is evicted from the private partition of the cache. The evicted block is allocated in the shared cache partition. Each core has a minimum of 1 cache block per set in the shared block partition, so space is guaranteed for this block. The algorithm for finding which block to evict in the shared cache partition in order to allocate the evicted cache block from the private cache is shown in Algorithm 1. The search for a victim block starts at the bottom of the LRU stack (step 2) and steps the LRU stack towards the MRU block. If the core that owns the cache block has too many cache blocks within the set (step 4), this block is chosen for eviction (step 5). If no block is found, the LRU cache block is evicted (step 8).

2.5 Repartitioning of the Cache

Repartitioning the cache might sound expensive. However, the only changes that are needed are in the partition parameters for the private and shared cache. This influences the replacement policy of the cache, and the eviction process on cache misses that actually repartition the cache later on. When the partition size for

one core is decreased, the "extra" cache blocks are not invalidated, but they are valid until they are evicted. When a private set is controlled for a hit, the tags for all blocks in the set are compared to the tag of the requested block including the blocks that are not in the partition of the private set. This lazy repartitioning, i.e. repartitioning only involves changing the parameters for the replacement policy, requires virtually no effort and can therefore be done at any pace.

Algorithm 1 Pseudo code for finding a block for eviction. The function returns the position of the block to evict.

```
1: function FIND BLOCK TO EVICT
2:     for LRU_stack_pos ←no blocks per set, 1 do
3:         proc_id ← core owns block(LRU_pos)
4:         if max no of blocks in set[proc_id] < count no of blocks in set(proc_id)
   then
5:             return LRU_stack_pos
6:         end if
7:     end for
8:     return number of blocks per set
9: end function
```

2.6 Discussion

One implication of using the new scheme is that some applications do not get the cache space they require because some other applications can utilize the cache more efficiently in the sense that the total miss rate will be lower. This means that an application that frequently accesses the last-level cache and which benefits from a large cache space is likely to get more cache capacity. An application that infrequently accesses the last-level cache, but that would also experience a lower miss rate from a larger cache space will less likely get a larger cache space since the number of misses removed by increasing the cache space for that application is lower. Consequently, applications that access the cache rarely or that do not benefit from a larger cache space will receive modest cache capacity if other more demanding cores (on the same chip) benefit from a large cache space.

The speed of the application (instructions per clock cycle) will depend on the number of accesses to main memory since these take several hundred clock cycles. The new scheme will prioritize these slow running applications if increasing the cache size helps. This is different from maximizing the average speed of the cores in a multi-core chip. In fact, the objective is to maximize the harmonic mean performance of the cores. As Smith points out [12]: This is more important than optimizing the average performance since most systems are often bound by the slowest running application. The result is that performance might be sacrificed for fast running applications to speed up slower running applications compared to a conventional shared cache.

2.7 Implementation Cost

The implementation cost can be divided into the extra storage required by the new scheme and the extra logic required.

The shadow tags require $s * p * t$ bits where s is the number of sets, p is the number of cores, t is the number of bits per tag. However, shadow tags are not needed for all sets as shown later in Section 4.6. We have found that monitoring only 6% of the sets is sufficient for estimating cache sizes with no degradation of performance. The field for the ID of the core that fetched the block requires $\log_2 p * b$ bits where b is number of cache blocks. Finally, the storage for the two counters and one register per core is $p*3*w$ bits for w-bits registers and counters. The total storage cost is then $0.06 * s * p * t + \log_2 p * b + p * 3 * w$ bits. For the baseline architecture used in the evaluation section the storage requirements are increased with 152 Kbits. 16% of this is used for shadow tags and 84% is used for core IDs in the blocks. This is an increase of 0.5% of the storage requirement for a 4-MByte last-level cache.

The logic and communication in the sharing engine require some extra cache logic. However, a conventional multi-core chip with private caches also requires logic for connecting the caches as they usually share the off-chip bus. Most of the logic in the sharing engine can be rather slow since its latency is overlapped by the slow memory access latency. We therefore believe that the area and power budget for the sharing engine are modest, however more work is required to quantify this.

3 Methodology

Simulation is used to compare the efficiency of the new scheme with a conventional LRU-based shared cache, with private caches and with recently proposed NUCA schemes. The simulated architecture is shown in Figure 1. The baseline chip multiprocessor (CMP) architecture has four cores per chip with private L3 cache partitions that can also be shared. We use a detailed pipeline-level, out-of-order execution model simulator with non-blocking caches to get statistics on improvements using the instructions-per-cycle (IPC) metric. The model is based on *SimpleScalar* version 3 [1], but is extended to simulate the new schemes and CMP configurations including congestion to main memory.

The baseline parameters for the simulator are shown in Table 1. The latency for the L3 cache is 19 cycles for a shared cache and 14 cycles for a smaller private cache. A hit in a neighboring cache is assumed to take 19 cycles. The increase from 14 to 19 cycles is caused by extra communication latency and the time to handle the cache miss in the local cache. We assume serial last-level caches where tag and data lookup are separated. The latency introduced by the cache itself is lower for a miss than for a hit since data lookup is not performed for misses. The numbers are based on recent processors from AMD and Intel, other papers [3, 16] and CACTI 4.0 [15]. These numbers do not only depend on the architecture but also on the implementation. Therefore we only show relative performance in the evaluation section.

Table 1. The baseline configuration used for the experiments.

Parameter	Value
Register Update Unit Size	128 instructions
Load Store Queue	64 instructions
Fetch queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Functional Units	4 INT ALUs, 4 FP ALUs, 1 INT Multiply/Divide, 1 FP Multiply/Divide
Branch Predictor	Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history table, 4K Chooser
Branch Target Buffer	512-entry, 4 way
Mispredict Penalty	7 cycles
L1 Instruction/Data Cache	64K, 2-way (LRU), 64 B Blocks, 2/3 cycle latency
L2 Instruction/Data Cache	128/256K, 4-way (LRU), 64 B Blocks, 9/9 cycle latency
Shared L3 Cache	4 MByte unified, 16-way (LRU), 64 B Blocks, 19 cycle latency
Private L3 Cache	1 MByte per processor, 4-way (LRU), 64 B Blocks, 14 cycle latency for private cache, 19 cycle latency for neighboring cache
Main Memory	260 cycles first chunk (258 for private cache), 4 cycles inter chunk. Chunk size 8 bytes. 9 GBytes/s theoretical limit for 4.5 GHz processor
I-TLB/D-TLB (Translation Lookaside Buffers)	128-entry, fully associative, 30 cycles miss penalty
Processor cores	4 independent cores

All of the *SPEC2000* benchmark applications were used with the reference data sets except for two: The simulator had compatibility problems with *vortex* and *sixtrack*, and they are not included in the experiments. We create multiprogrammed workloads for our CMP architecture as follows. In each experiment, four randomly picked applications are run in parallel. Each application is randomly forwarded between 0.5 and 1.5 billion instructions and then we simulate two hundred million cycles.

We do not consider sharing of cache blocks in this paper as would be the case had we used parallel workloads. However we hypothesize that the new scheme will be effective also for such workloads and will address it in our future work.

4 Results

Several of the SPEC2000 applications have a small working set which more or less fits into the L1 and L2 cache. These applications are not sensitive to enhancements of the last-level (L3) cache and hence not relevant for the evaluation of the proposed scheme. Nevertheless, it is important that our proposed scheme is robust also for non-memory-insensitive applications. We compare the performance of the new scheme with a pure private cache organization because the performance of such an organization is quite predictable and well understood. We first classify the applications with respect to their sensitivity to last-level cache performance. We then consider the speedup of our scheme, the effects of larger caches and technology scaling and show the results of reducing the number of shadow tags. Finally we compare the performance of our scheme to an earlier proposed NUCA scheme.

4.1 Classification of Workloads

The number of last-level data cache accesses is shown in Figure 5 for different SPEC2000 applications. We classify the applications as (a) either being last-level cache intensive or (b) not depending on the last-level cache. The applications with more than nine last-level data cache accesses per thousand clock cycles are classified as last-level cache intensive. The rationale behind this is that there could potentially be a last-level cache miss every hundred clock cycles which add another few hundred cycles to the execution time. The number of last-level cache misses per clock cycle could have been used to classify the applications as well, but then the applications that work well for a conventional LRU scheme would not have been included in the evaluation of the proposed scheme. If the new scheme degraded performance for these applications it might not have been detected.

4.2 Speedup of the Proposed Scheme

The harmonic mean of the instructions per clock cycle (IPC) for all four cores per experiment is shown in Figure 6 for the last-level cache-intensive applications.

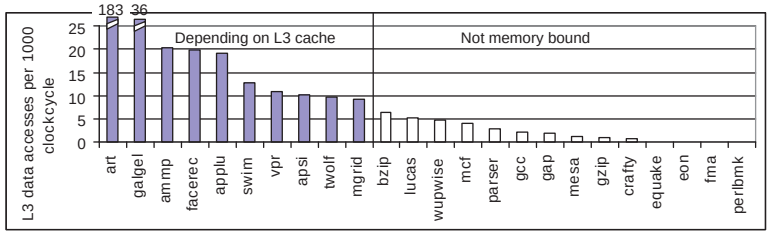


Fig. 5. Classification of applications based on number of misses in L2 data cache.

Each experiment consists of four randomly picked applications run in parallel as described in the methodology section. The experiments are sorted by the performance of the new scheme relative to a private cache organization with the experiments with the highest speedup to the right. Except from a single experiment, the new scheme has equal or higher performance than both the private cache and shared cache schemes. The shared cache does a good job of speeding up the performance for the last-level cache intensive applications which is why the new scheme only has 2% higher harmonic speedup while the average speedup is 5%. Compared to private caches the harmonic mean of the new scheme is 21% higher while the average speedup is 13% higher.

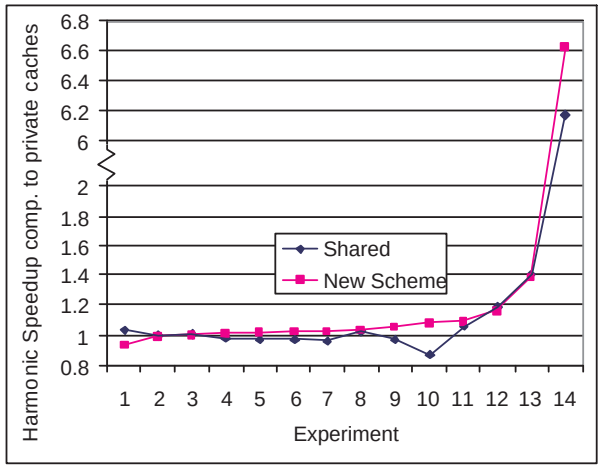


Fig. 6. Harmonic mean of IPC for the last-level cache intensive benchmarks per experiment.

The performance of the proposed scheme is shown for each last-level cache intensive application in Figure 7. The performance is compared against that of

private caches, shared caches and private caches where each cache is the size of the shared cache (4 x size private). By considering the speedup with the 4 x larger private cache we see which applications that could benefit from larger caches. These are *ammp*, *art*, *twolf* and *vpr*. The proposed scheme works well for these four applications while a shared cache degrades the performance for two of them compared to private caches.

The new scheme degrades performance for some of the applications. This is expected since these applications are not given the same amount of cache space as with the private and shared schemes because some other application are believed to benefit more from using the cache space.

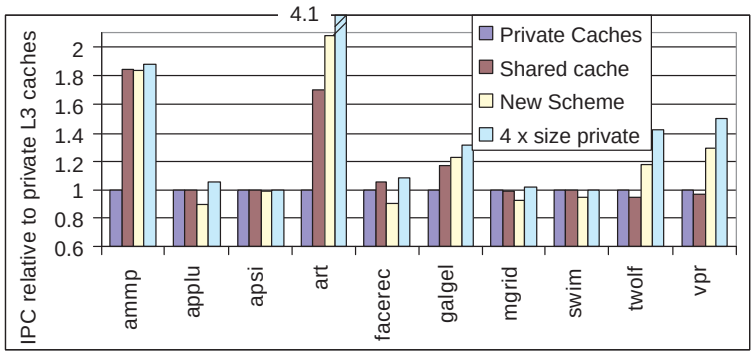


Fig. 7. Speedup for the last-level cache intensive benchmarks.

4.3 Speedup for All SPEC2000 Benchmarks

The speedup per application compared to private caches for all SPEC2000 applications is shown in Figure 8 with both application categories; last-level cache intensive and last-level cache non-intensive applications. The poor performance of *wupwise* is caused by an experiment that contains three instances of *ammp* and a single instance of *wupwise*. *ammp* can utilize the cache space better than *wupwise* and therefore the performance is degraded to speedup the slow *ammp*. The actual numbers for the IPC for this case is for the proposed scheme: *wupwise*: 1.326 and *ammp*: 0.0323, 0.0322 and 0.0319, and for the other schemes *wupwise*: 1.7974 and for *ammp*: 0.0319 x 3. Looking at the harmonic mean, the new scheme improves the performance (although very little) while the other scheme degrades the performance (also very little). Even though the performance of *wupwise* is degraded, the new scheme makes the correct decision since the goal is to increase the harmonic mean.

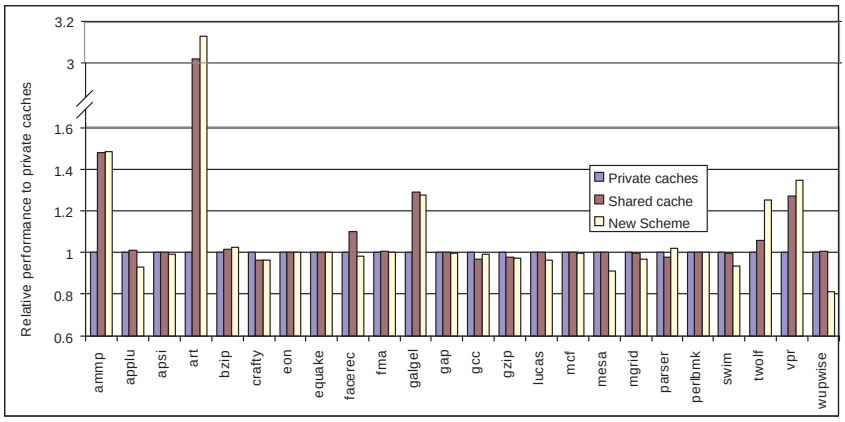


Fig. 8. Speedup for all benchmarks.

4.4 Effects of Larger Caches

The SPEC2000 benchmark applications do not require very large caches. To illustrate this, Figure 9 shows the performance for an 8-MByte L3 cache. Most of the applications do not run faster with the larger cache as seen by the performance of the four times larger (4 x 8 MByte) private cache. For a simple comparison, the timing model is the same as used with the 4-MByte cache. The proposed scheme actually degrades the performance for many applications. This is because the proposed scheme infers constraints in a system that really does not need any restrictions because the cache size is so large compared to the requirement.

4.5 Impact of Technology Scaling

As technology gets denser in the future, the clock cycle will become slightly shorter while the communication latency is expected to stay the same. We have run experiments to find the impact of future technology scaling. The cycle time is assumed to be reduced by 30% for the core, which causes the number of cycles to be increased for the different caches since much of the cache cycle time is due to communication latency: The latency of the L2 cache is increased from 9 to 11 clock cycles, the latency of the L3 private/shared cache is increased from 14/19 to 16/24 and the main memory access time is increased from 258/260 to 330/338 for pure private/shared. The increase in clock cycles is based on that the cache partitions that are close to the cores have less increase than the last-level cache since a shorter communication distance is involved. As the technology scales, the access to main memory becomes slower and the shared cache becomes slower. The results are shown in Figure 10. On average (shown to the right in the graph) the new scheme has the highest gain. This is because the new scheme removes most memory accesses to main memory which becomes increasingly important.

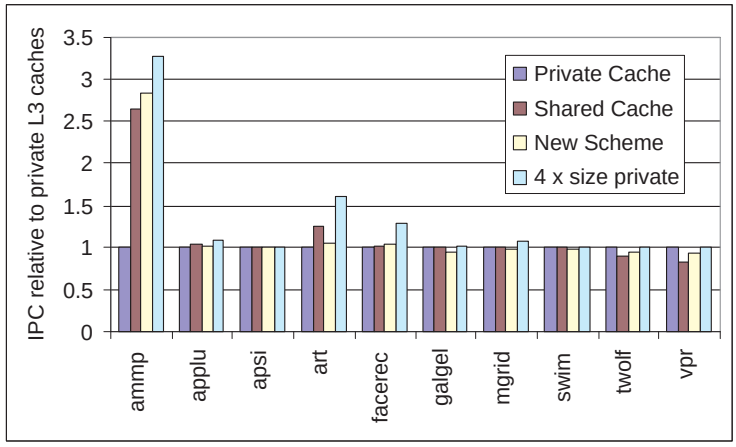


Fig. 9. 8 MByte l3 cache.

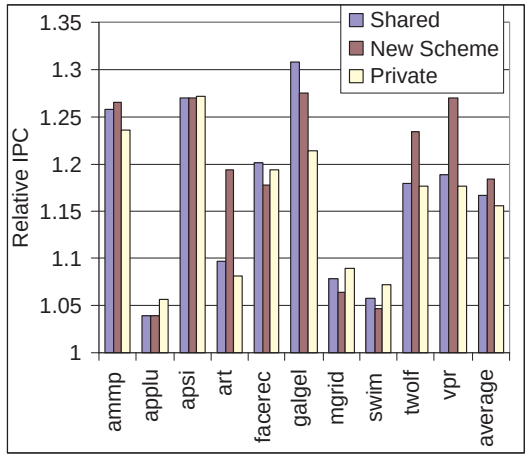


Fig. 10. The impact of scaling the technology.

4.6 Reducing the Number of Shadow Tags

The experiments in the previous sections were done with four shadow tags for every set to predict marginal gains of increasing cache size for the cores. However, it is not necessary to implement shadow tags in all sets. Our previous work has revealed that monitoring the sets with the lowest index works well and better than randomly generated subsets or subsets based on prime numbers [7]. We have run simulations with shadow tags in only 1/16 of the sets with the lowest index and found that the average IPC was increased with 0.1% while the harmonic IPC was decreased by 0.1%. This means that the tags with the lowest index represent the whole cache very well for the new scheme. LRU hits are counted for in all sets, but the numbers are normalized when compared to shadow tag hits. The cost of monitoring only $1/16 \approx 6\%$ of the sets is not very high and is discussed in Section 2.7.

4.7 Comparison with another NUCA Scheme

We implemented a hybrid scheme based on Chang and Sohi's work [3]. Their scheme is based on a chip multiprocessor with private caches. However, when a cache block is evicted from a private cache it might get installed in the neighboring cache. To illustrate the scheme through an example, consider a core a that has cache a and another core b that has cache b. When core a loads new data into its private cache, one block is evicted to make space for the new data. If this block was loaded by the core that owns the private cache (a), and it is evicted due to an access by the same core (a), this block is installed into a random neighboring cache as the most recently used (MRU). In this case this is cache b since there are only two caches. If the block that was evicted from cache a belonged to cache b, it must earlier have been evicted from cache b, and therefore it is not allocated again. When spilling a cache block from cache a into cache b, a block from cache b has to be evicted as well. This block is not allocated in some other caches to avoid ripple effects.

We call this scheme for "random replacement" and compare it against the proposed scheme for memory-intensive applications in Figure 11. The proposed scheme in general works better than the random-replacement scheme. This is not surprising as the random-replacement scheme works best when not all cores are competing for the cache resources. Figure 12 shows an experiment with both benchmark categories. In this case the proposed scheme is not that superior compared to the random replacement scheme. This is due to the many applications that do not use the last-level cache space.

5 Related Work

Recently there has been a significant interest on NUCA schemes for chip multiprocessors with papers addressing the sharing of data, migration of cache blocks and reducing the hit time [2, 3, 6, 9, 16]. Important aspects of these works are

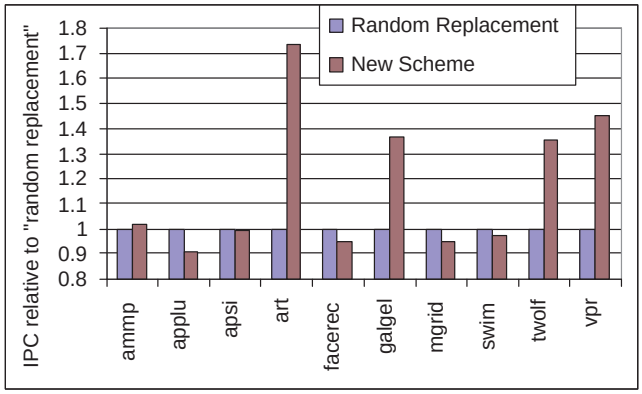


Fig. 11. Performance of the new scheme relative to "random replacement".

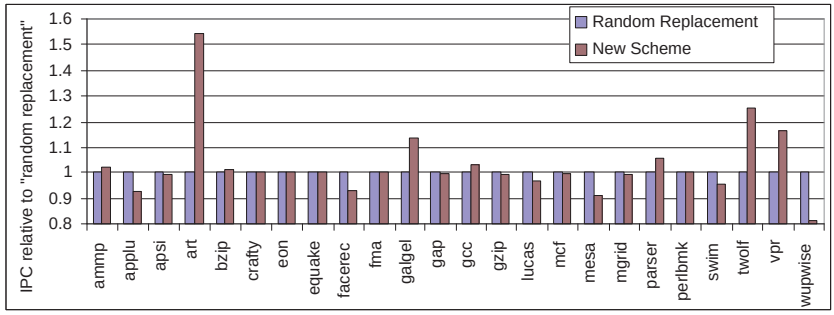


Fig. 12. Performance of the new scheme relative to "random replacement".

cache coherence protocols and duplication of shared cache blocks to reduce access latencies for several cores. Our work is concerned with optimizing the cache usage per core and is complementary to these works.

Even though this is the first paper to consider adaptive cache partitioning for NUCAs, several of the components used are a combination or extension of earlier described methods. We acknowledge this prior art next. Suh et al. described a way of partitioning a cache for multithreaded systems by estimating the best partition sizes [13, 14]. They counted the hits in the LRU position of the cache to predict the number of extra misses that would occur if the cache size was decreased. A heuristic used this number combined with the number of hits in the second LRU position to estimate the number of cache misses that are avoided if the cache size is increased. We extended this scheme with shadow tags and counters for estimating the number of hits that would be avoided by increasing the cache size [7]. The method presented in [7] increases the precision of the predictions.

In this paper we use the same mechanism, but apply it to a NUCA organization to control the size of the private partitions. The previous work considered adjusting the size of the cache partitions within a shared cache while we in this paper adjust the shared partition size of the local last-level caches as well as controlling the number of blocks per core in the shared partition. The previous works did not consider a shared partition with variable size, nor did they look at combining private and shared caches. Evaluation of cache partitioning in shared chip multiprocessor caches has also been studied earlier by Kim et al. [10] for a two-core CMP where a trial and fail algorithm was applied. Trial and fail as a partitioning method does not scale well with increasing number of cores since the solution space grows fast.

Spilling evicted cache blocks to a neighboring cache was described by [3, 6]. They did not consider putting constraints on the sharing or methods for protection from pollution. No mechanism was described for optimizing partition sizes. We extend their work by insertion of the constraints on cache usage, divide the cache space in private and shared partitions and mechanisms for finding the best partition sizes. As our results show, the extensions improve performance significantly.

A mechanism for protecting cache blocks within a set was described by Chiou et al. [4]. Their proposal was to control which blocks that can be replaced in a set by software in order to reduce conflicts and pollution. The scheme was intended for a multi-threaded core with a single cache. We use the same mechanism to divide the blocks within a set into two partitions, a private partition and a shared partition. We then combine the shared partitions from several caches into a single shared cache space.

Qureshi et al. independently developed a mechanism very similar to shadow tags [11]. The only significant difference is in the selection criteria for the tags and they have included a formal proof of the concept based on the assumption that all sets affect performance equally.

6 Conclusion

The performance of multi-core systems relies on an effective cache system. We have considered improving the cache system by adapting the cache usage per core to its needs by protecting its most recently used data in the last-level cache.

Simulations show that the new scheme has higher performance in terms of instructions per clock cycle than private cache organizations because of a higher utilization and potentially larger caches for applications that benefit from an increased cache size. Compared to shared caches performance is in general increased due to (a) lower access time and (b) improved sharing. Hits to the local cache are faster than in a shared cache due to its smaller size and lower associativity. Each core is protected from pollution by the other cores and is given a cache size which minimizes the total number of cache misses for all cores. Earlier NUCA schemes do not consider partitioning and protection of cache blocks in the last-level cache. As technology scales and latency becomes more dominant, the proposed scheme is predicted to be even more advantageous because it reduces the number of cache misses which will be an increasing bottleneck. Even though we have only simulated a four-core processor, we believe the scheme will scale to systems with a higher processor count. Additionally, the proposed scheme only requires modest hardware for implementation.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. IEEE Comp., V35I2, 2002.
2. Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. MICRO 37, 2004.
3. Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. ISCA, 2006.
4. D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. Proceedings of Design Automation Conference, Los Angeles, 2000.
5. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. MICRO 36, 2003.
6. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. ISCA, 2005.
7. Haakon Dybdahl, Per Stenström, and Lasse Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. HiPC, 2006.
8. Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. ICS, 2005.
9. Changkyu Kim, Doug Burger, and Stephen W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. IEEE Micro, vol. 23, no. 6, pp. 99-107, 2003.
10. Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2004.

11. Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. ISCA, 2006.
12. J. E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10), 1988.
13. G. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. IASTED Int. Conf. on Parallel and Distributed Computing Systems, 2001.
14. G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. HPCA, 2002.
15. David Tarjan, Shyamkumar Thoziyoor, and Norman P. Jouppi. Cacti 4.0. Technical report HP Laboratories Palo Alto, HPL-2006-86, 2006.
16. Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. ISCA, 2005.