Marius Grannæs

Reducing Memory Latency by Improving Resource Utilization

Doctoral thesis
for the degree of philosophiae doctor

Trondheim, June 2010

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Doctoral thesis
for the degree of philosophiae doctor

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-2177-1 (printed version)
ISBN 978-82-471-2178-8 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2010:106

Printed by NTNU-trykk

Typeset with LaTeX 2ε in Computer Modern 10pt

Abstract

Integrated circuits have been in constant progression since the first prototype in 1958, with the semiconductor industry maintaining a constant rate of miniaturisation of transistors and wires. Up until about the year 2002, processor performance increased by about 55% per year. Since then, limitations on power, ILP and memory latency have slowed the increase in uniprocessor performance to about 20% per year. Although the capacity of DRAM increases by about 40% per year, the latency only decreases by about 6 – 7% per year. This performance gap between the processor and DRAM leads to a problem known as the memory wall.

This thesis aims to improve system memory latency by leveraging available resources with excess capacity. This has been achieved through multiple techniques, but mainly by using excess bandwidth and improving scheduling policies.

The first approach presented, destructive read DRAM, changes the underlying assumptions about the contents of a DRAM cell being unchanged after a read. The latency of a read is reduced, but the rest of the memory system requires changes to conserve data.

Prefetching predicts what data is needed in the future and fetches that data into the cache before it is referenced. This dissertation presents a technique for generating highly accurate prefetches with good timeliness called *Delta Correlating Prediction Tables (DCPT)*. DCPT uses a table indexed by the load's address to store the delta history of individual loads. Delta correlation is then used to predict future misses. *Delta Correlating Prediction Tables with Partial Matching (DCPT-P)* extends DCPT by introducing *L1 hoisting* which moves data from the L2 to the L1 to further increase performance. In addition, DCPT-P leverages *partial matching* which reduces the spatial resolution of deltas to expose more patterns.

The interaction between the memory controller and the prefetcher is especially important, because of the complex 3D structure of modern DRAM. Utilizing open pages can increase the performance of the system significantly. Memory controllers can increase bandwidth utilization and reduce latency at the same time by scheduling prefetches such that the number of page hits are maximized. The interaction between the program, prefetcher and the memory controller is explored.

This thesis examines the impact of having a shared memory system in a CMP. When resources are shared, one core might interfere with another core's execution by delaying memory requests or displacing useful data in the cache. This effect is quantified and which components are most prone to interference between cores identified. Finally, we present a framework for measuring interference at runtime.

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree philosophiae doctor (PhD). The work herein was performed at the Department of Computer and Information Science, NTNU, under the supervision of Professor Lasse Natvig.

This thesis consists of two parts. The first part consists of introduction, background, research process, a summary of papers and final conclusions. The second part is the main contribution, presented as a collection of nine research papers.

Acknowledgements

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

First, I would like to thank my advisor Lasse Natvig for his support, guidance and keeping me on track. He has allowed me to explore this wonderful area of computer science in the way I wanted. My co-advisors, Professor Mads Nygård and Dr. Gaute Myklebust deserves credit for guiding this work, especially in the early phases. Their insights provided much valuable knowledge for completing this thesis.

I would especially like to thank my co-authors: Dr. Haakon Dybdahl, Magnus Jahre and Dr. Per Gunnar Kjeldsberg. Thank you for all the marvellous computer architecture discussions that have led to such wonderful research ideas. Asbjørn Djupdal deserves my gratitude as he did a wonderful job proof-reading the final manuscript for this thesis. I would also like to thank all my coworkers at the Complex Systems Section at NTNU: Morten, Dragana, Gunnar T, Gunnar L, Pauline, Kostas, Bjørn-Magnus, Frode, Sigve, Christina, Jan-Christian, Thorvald, Magnus, Nils and Truls.

I would like to thank Professor Per Stenström for letting me visit Chalmers during the spring of 2007. This visit was very inspiring and very valuable. Thanks to

all the people I met there; Martin, Magnus, Wolfgang and Daniel. Thank you for taking so good care of me and making my stay so memorable.

I would like to thank the Faculty of Information Technology, Mathematics and Electrical Engineering and the Department of Computer and Information Science for funding my research. NOTUR provided computational resources which were very valuable for making simulating systems easier and less time-consuming. I would like to thank the HiPEAC European network of excellence. I was fortunate to attended the ACACES summer school on two separate occasions, both of which were very interesting.

I would like to thank my family for inspiring me and supporting me through all these years. Finally, I would like to thank Maria Kristina for her love and support.

Marius Grannæs
June 09, 2010

Contents

List of Figures

List of Tables

Abbreviations

AMPM	Access Map Pattern Matching
API	Application Programming Interface
AWS	Aggregated Weighted Speedup
C/DC	CZone/Delta Correlation
CDP	Content-Directed Prefetching
CMP	Chip Multi-Processor
CPU	Central Processing Unit
DCPT	Delta Correlating Prediction Tables
DCPT-P	Delta Correlating Prediction Tables with Partial Matching
DDR	Double Data Rate
DPC	Data Prefetching Championship
DRAM	Dynamic Random Access Memory
EDP	Energy Delay Product
ED^{2}P	Energy Delay Squared Product
eDRAM	Embedded Dynamic Random Access Memory
FCFS	First-Come First-Served
FIFO	First In, First Out
FR-FCFS	First-Ready First-Come First-Served
GA	Genetic Algorithms
GHB	Global History Buffer
GHB-LDB	Global History Buffer - Local Delta Buffer
HMS	Harmonic Mean of Speedups

ICCD	International Conference on Computer Design
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
IQ	Instruction Queue
ITRS	International Technology Roadmap for Semiconductors
JILP	Journal of Instruction-Level Parallelism
LRU	Least Recently Used
MSHR	Miss Status Holding Register
MLP	Memory Level Parallelism
NFQ	Network Fair Queuing
NUCA	Non-Uniform Cache Architecture
OoO	Out-of-Order
PC	Program Counter
PC/DC	Program Counter/Delta Correlation Prefetching
PDFCM	Prefetching based on a Differential Finite Context Machine
PVT	Page Vector Table
RL	Reinforcement Learning
ROB	Reorder Buffer
RPT	Reference Prediction Tables
SDP	Stride Directed Prefetching
SMS	Spatial Memory Streaming
SMT	Simultaneous Multithreading
SRAM	Static Random Access Memory
TCP	Tag Correlating Prefetching
TLB	Translation Lookaside Buffer
TMS	Temporal Memory Streaming
TPS	Transactions Per Second
WAM	Weighted Arithmetic Mean
WHM	Weighted Harmonic Mean

Chapter 1

Introduction

There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed – you can't bribe God.
– Anonymous

1.1 The Memory Gap

Each year exponentially more transistors can be put into a single integrated circuit [57, 99]. Moore's law is the empirical observation that the number of transistors that can be placed on an integrated circuit, with respect to minimum component cost, will double every 24 months. Increased transistor density, in turn, translates into faster computers for consumers.

Up until about the year 2002, processor performance increased by about 55% per year [47]. Since then, limitations on power, *Instruction Level Parallelism (ILP)* and memory latency have slowed the increase in uniprocessor performance to about 20% per year. Although the capacity of *Dynamic Random Access Memory (DRAM)* increases by about 40% per year, the latency only decreases by about 6-7% per year [111]. This gap between the processor and DRAM leads to a performance problem known as the *"memory wall"* (or *"memory gap"*) [149]. Figure 1.1 shows the relative uniprocessor performance versus memory latency.

The most important technique in overcoming the memory wall was the introduction of caches in the memory hierarchy [47, 128]. Caches were first introduced in literature in 1968 in a description of the memory system in a IBM Model 85 [137]. Caches are smaller and faster memories which exploits spatial and temporal locality. Spatial locality is the tendency for programs to access data that is close in address space, for example instructions. Temporal locality is the tendency for programs to access the same data repeatedly. Examples include: read-modify-write

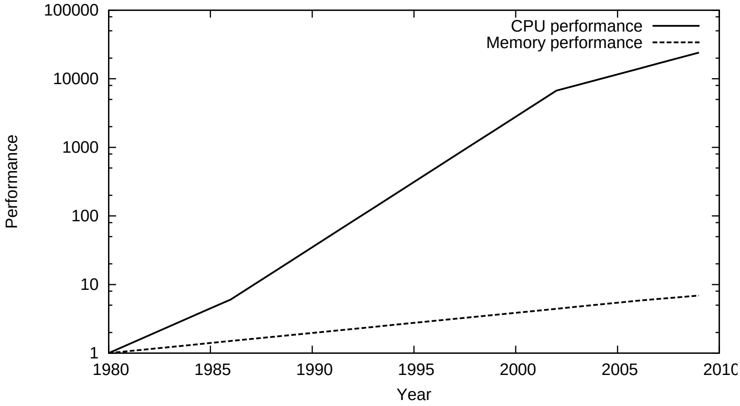


Figure 1.1: Development of CPU performance versus memory latency [47].

cycles and single variables in tight loops. These observations can be exploited by moving recently used data closer to the processor. This makes it faster to access data on average, which in turn speeds up overall computation. The importance of this technique is clearly shown in figure 1.2 where the size of the cache is plotted as a function of the year of introduction for some Intel processors.

1.2 Analyzing the Memory Hierarchy

Equation 1.1 shows a simplified[1] analytical model to calculate the overall system latency given a single level cache memory hierarchy. For a more complete analytical model see Jacob et al. [63].

$$L_{system} = L_{cache} + p_{miss} \cdot (L_{main\ memory} + L_{congestion}) \tag{1.1}$$

In this equation L_{system} is the overall memory system latency as observed by the processor. Decreasing L_{system} can thus increase overall system performance. L_{cache} is the latency of the cache. p_{miss} is the probability that the data is not found in the cache. If the data is not found in the cache, then the data is found in main memory. The latency of main memory consists of two components: $L_{main\ memory}$ is the minimum time to transfer data over the memory bus. Additionally, modern processors (or *Chip Multi-Processors (CMPs)*) can issue multiple memory requests that can be serviced concurrently which can cause congestion, which in turn increases latency ($L_{congestion}$).

[1] This model assumes no virtual memory and infinite cache bandwidth. In addition, in most implementations the latency of a cache miss is different from a cache hit.

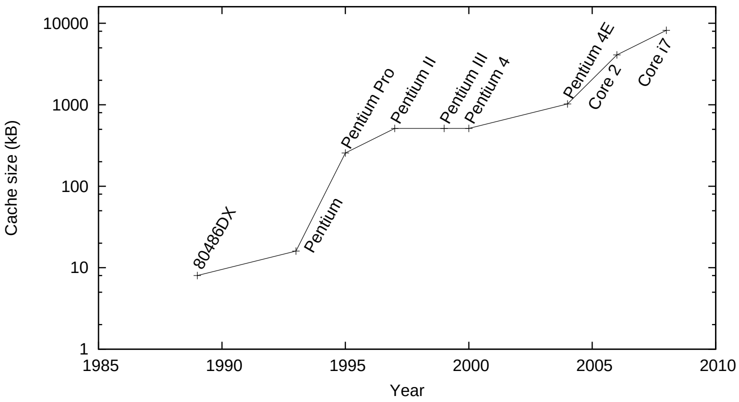


Figure 1.2: Size of the last level on-die cache as a function of year of introduction for some Intel microprocessors. Note that only one cache size is shown per processor. In practice Intel varies the amount of cache on a processor as a way to differentiate products and to enhance yield [147].

1.3 Overcoming the Memory Wall

To increase memory system performance there are three main strategies: Tolerating or hiding the latency, increasing bandwidth utilization and moving to a parallel throughput-oriented architecture (CMPs).

1.3.1 Tolerating or Hiding the Memory Gap

Naturally, because of the importance of the memory system in achieving high performance several techniques have been developed to decrease or tolerate the memory system latency. Using a memory hierarchy of caches is a technique that hides the memory latency from the processors viewpoint. Although access to main memory is slow, in most cases the data that is needed will be in a faster cache. Prefetching or speculative loads, i.e. moving data from main memory to caches speculatively, can hide more of main memory latency [47]. Scratchpad memory makes the programmer explicitly move data from main memory to faster storage [8], which can increase performance. The Cell processor uses such an approach where each processor core has a relatively small local storage area [72, 79, 148].

Because cache misses will occur, it is important to be able to tolerate these events and, if possible, continue execution. *Out-of-Order (OoO)* execution allows the processor to continue execution of instructions which do not depend on the load [47]. By using caches that can handle multiple concurrent misses (lock-up free) the processor can then issue multiple loads that might also miss in the caches while waiting for the original load to complete [116].

In most operating systems, the processor will schedule another thread if it is stalled waiting for a long I/O operation. *Simultaneous Multithreading (SMT)* takes this further by allowing multiple threads to execute on the same core [141]. If one core stalls because of a load, the other threads will be able to continue execution.

The UltraSPARC T1 ("Niagara") processor uses up to eight cores which can process four threads simultaneously [77]. This adds up to a total of 32 concurrent threads. The idea is that by having a large number of concurrently executing threads, stalling is minimized.

1.3.2 Increasing Bandwidth Utilization

Overall, the amount of off-chip communication is limited by the number of pins on the chip package [52]. In the short term (2007-2015) *International Technology Roadmap for Semiconductors (ITRS)* projects that the number of pins will increase by only 7% p.a. [57]. Because the number of transistors per chip increases much faster, this results in that the number of transistors per pin increases exponentially. This in turn increases the bandwidth requirements per pin.

To meet this demand for increased bandwidth, a variety of techniques have been employed. DRAM interfaces have moved from asynchronous to synchronous with fast page buffers. The fast page buffers holds the most recently used data in a faster access buffer to exploit the same spatial locality as caches. *Double Data Rate (DDR)* memory was introduced to further increase effective bandwidth by transferring data on both edges of the DRAM clock signal. These techniques have all increased bandwidth by a significant amount, but at the expense of higher latency [47]. In addition, these advances have increased the complexity of the DRAM interface, thus increasing the interest in DRAM scheduling policies [119]. In particular, scheduling decisions can effect the utilization of the memory bus by prioritizing requests that can utilize the fast page buffer (page hits). Furthermore, by speculatively prefetching data into the cache, a lower latency can be traded for higher bandwidth usage.

1.3.3 Parallel Throughput-Oriented Architectures

Recently, there has been a shift in the industry from uniprocessors to CMPs. A CMP is multiple processor cores in a single package [109]. This shift is partially due to the memory gap, but equally important is the limits on ILP, power dissipation and design complexity. This also marks a shift in programming paradigms. To achieve maximum performance, a parallel implementation of the application is required [112]. Additionally, this shifts the focus away from single-threaded performance to system throughput. However, due to Amdahl's law [47], there are limits to the maximum speedup achievable by having a parallel implementation as some portions of the code are bound to be sequential. In practice, some applications are more difficult to parallelize than others. Webservers or search engines are typical examples of programs that can easily be made parallel as each client can use a separate thread and there are typically more users than cores available. Other applications are harder to parallelize, because of dependencies between computations that forces serialization, such as long pointer-chains. In this context, there are two optimization goals: reducing the overall memory latency for the serial portion and increasing memory throughput and decreasing latency in the parallel portion.

1.4 Research Questions

The main research question for this thesis is:

> *How, and at what cost, can memory system latency be reduced by improving resource utilization?*

This question can be subdivided further according to equation 1.1 into the following subquestions:

1. How can excess memory bandwidth be utilized to achieve a lower **maximum** memory latency ($L_{main\ memory}$) ?
2. How can excess memory bandwidth be utilized to achieve a lower **average** memory latency (p_{miss}) ?
3. How does scheduling decisions in modern highly parallel and complex DRAM interfaces affect the bandwidth/latency trade-off (p_{miss} and $L_{congestion}$) ?
4. How does interference in the shared memory system affect Chip Multiprocessor performance ($L_{congestion}$) ?

1.5 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 contains background information regarding measuring performance, DRAM, caches and prefetching. Chapter 3 describes the research process, introduces each paper, describes the simulators used and the methodology. Each paper is described in chapter 4 with a breakdown of the roles of each author and a retrospective view (where applicable). Chapter 5 concludes the thesis with a summary of contributions, some thoughts on future work and an outlook. The appendix holds each paper I have authored or coauthored in chronological order. These papers are reproduced faithfully with regard to the published text, but has been reformatted to increase readability.

Chapter 2

Background

Those who don't know history are doomed to repeat it.
– Edmund Burke

2.1 Performance and Fairness Metrics

2.1.1 Measuring Performance

Measuring performance is a tricky task even for a single processor system. The ideal measure of performance is the wall clock time needed to complete a computation [47, 129]. "Completing a computation" can have different meanings from a user and a system perspective. A user is often interested in the *response time* of the system. That is the time it takes from the user issues a request to the completion of that request. The system has a different, and perhaps conflicting, view. In the system view the objective is to maximize the overall *throughput*. Throughput is a measure of the amount of computation performed by the system as a whole during a time interval. These two views can be conflicting, because the system might opt to delay one task (thus increasing that task's response time) in order to prioritize some other task, which would in turn increase throughput.

However, measuring wall clock time is not always practical. This is especially true when simulating a computer system where running an entire benchmark suite to completion could take several weeks. Thus, *Instructions Per Cycle (IPC)* is often used as a proxy for overall performance. The IPC of a system can often be measured directly through performance counters, which are present in most modern high-performance processors [14]. In practice architects often simulate a portion of the benchmark and measure the IPC during that portion of the benchmark. Another approach is to reduce the dataset, which in turn reduces simulation time [150].

Such an approach can lead to non-representative performance measurements due to phase changes in program behaviour. This effect can be mitigated through the use of Simpoints [113]. Simpoints uses a statistical model to select several representative points in the program execution and aggregates the results from several such points. A related approach is used by the SMARTS system [150]. SMARTS uses random samples and uses statistics to determine when the measurements converge and thus stop simulating.

IPC can be misleading as an indicator of performance in situations where a program can commit instructions, but fail to make forward progress. This is especially true in multi-threaded applications where threads can be waiting in a spinlock. A spinlock is often implemented as a tight loop which a modern processor can execute very quickly in terms of IPC. In this case, IPC will be high, but the actual work that is performed is none. This has lead to the development of more work-oriented metrics such as *Transactions Per Second (TPS)*, where the number of useful (database-)transactions per second is measured.

In practice, one is often more interested in the *speedup* that is achieved by using a certain technique rather than raw IPC numbers. Speedup is calculated according to equation 2.1 [47]. In this equation *new* refers to the enhanced system, while *old* refers to a system without the enhancement.

$$\text{Speedup} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} \tag{2.1}$$

If the same program with the same dynamic instructions are run and with the same clock frequency, then equation 2.1 can be rewritten to include IPC as shown in equation 2.2.

$$\text{Speedup} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} = \frac{\text{IPC}_{\text{new}}}{\text{IPC}_{\text{old}}} \tag{2.2}$$

In recent years there has been an increased interest in reducing the amount of power required by the processor. This interest has been sparked by the increasing number of mobile devices, which are powered by batteries. Therefore, decreasing the power requirements of the processor increases the life-time of the device considerably. In addition, as processors dissipate more power, the core temperature increases. To keep processors stable, significant cooling is required, which adds to the overall operating cost of the system [9].

It is possible to measure power (P) directly, but this is often not a very useful metric by itself as it does not give any indication of the computational performance. A more useful metric is the *Energy Delay Product (EDP)*. This metric has an equal balance between the energy requirements and the performance of the system. However, the problem with EDP is that because it favors energy and performance equally, the metric favors small and slow processors, because power consumption increases more than linearly with performance. Thus, the delay is often squared (ED^{2}P) or cubed (ED^3P) thus increasing the emphasis on performance [43].

2.1.2 Aggregating Performance Numbers

To properly characterize an architectural technique it must be simulated on a wide range of programs in order to ensure that the proposed technique applies to a broad range of applications, rather than exploiting a feature of a particular program. This is often achieved through using a benchmark suite which is comprised of several benchmarks. It is often useful to aggregate the performance results from the entire benchmark suite into a single number.

The simplest approach is to use *Weighted Arithmetic Mean (WAM)* as shown in equation 2.3. In this equation a measurement of program i is denoted by M_i. Each program is given a weight (ω_i) which can be adjusted according to user preference (program execution time, program importance in day-to-day use, etc.). Typically, when running experiments with a fixed number of cycles per benchmark WAM can be used with equal weights to measure average IPC [68].

$$\text{WAM} = \frac{1}{n}\sum_{i=1}^{n} \omega_i \cdot M_i \tag{2.3}$$

Another possibility is to use the *Weighted Harmonic Mean (WHM)* which is shown in equation 2.4. This metric is typically useful when aggregating rates [68, 129].

$$\text{WHM} = \frac{n}{\sum_{i=1}^{n} \frac{\omega_i}{M_i}} \tag{2.4}$$

The third option is the geometric mean shown in equation 2.5. The use of the geometric mean is discouraged by several researchers [62, 68, 129]. The problem with the use of the geometric mean is that it is less useful as an predictor of actual performance [129]. Furthermore, it is harder to visualize than the harmonic and arithmetic mean, because it uses an n-dimensional space.

$$G = \sqrt[n]{\prod_{i=1}^{n} M_i} \tag{2.5}$$

2.1.3 Multiprogrammed Workload Metrics

In a multiprocessor or *Chip Multi-Processor (CMP)* it is possible to increase the performance of one thread at the expense of another. One thread might use a disproportional amount of shared resources, such that a second thread's performance suffers. To measure this effect it is convenient to use a fairness metric such as the one proposed by Gabor et.al. [37, 41], as shown in equation 2.6.

$$\text{Fairness} = \min_{j,k} \left(\frac{\text{Speedup}_j}{\text{Speedup}_k} \right) = \min_{j,k} \left(\frac{\left(\frac{\text{IPC}_j^{MP}}{\text{IPC}_j^{\text{Alone}}} \right)}{\left(\frac{\text{IPC}_k^{\text{MP}}}{\text{IPC}_k^{\text{Alone}}} \right)} \right) \tag{2.6}$$

In this metric the speedup[1] for every core/processor is computed relative to the performance of that core running alone in the system (i.e. no sharing of resources). A fairness value of 1 indicates that all cores have equal speedup, while 0 indicates that at least one core is not making forward progress.

Furthermore, optimizing for this fairness metric alone is meaningless as it is easy to slow down every thread such that this metric approaches 1. Instead, this metric must be coupled with a performance metric, such as *Aggregated Weighted Speedup (AWS)* or *Harmonic Mean of Speedups (HMS)* [41, 92, 130]. AWS[2] is defined as [130]:

$$\text{AWS} = \sum_{i=1}^{n} \text{Speedup}_i = \sum_{i=1}^{n} \frac{\text{IPC}_i^{\text{MP}}}{\text{IPC}_i^{\text{Alone}}} \tag{2.7}$$

Where n is the number of processors/cores in the system and the speedup is calculated compared to a baseline where the core does not compete for resources (i.e. the other cores are idle).

HMS is defined as [92]:

$$\text{HMS} = \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{Speedup}_i}} \tag{2.8}$$

2.2 Main Memory

2.2.1 Memory Cells

Dynamic Random Access Memory (DRAM) is the most common technology used to implement main memory. To store a single bit of information, a capacitor and transistor is used as shown in figure 2.1. Such a DRAM cell works by storing a charge in the capacitor (C_B) [48]. If the storage capacitor (C_B) is charged to the supply voltage (V_{DD}) then the cell stores a 1. To access the data the cell transistor

[1] In practice, there will be a slowdown, because the performance of running alone in the system is higher.

[2] The *weight* in AWS and HMS is not explicit in these equations. The weight is inversely proportional to IPC. Lower IPC threads will get a higher speedup compared to high IPC threads with an equal increase in the number of committed instructions.

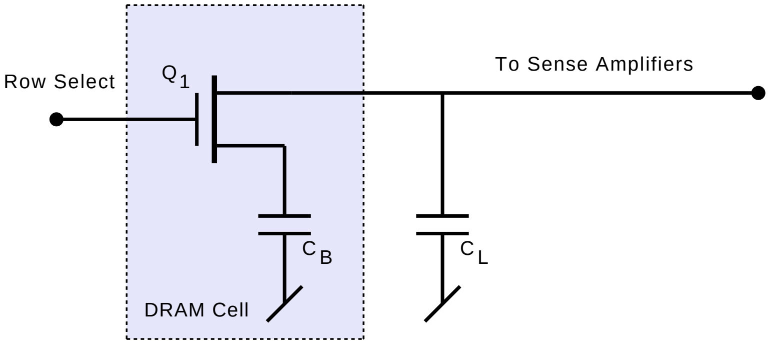


Figure 2.1: Schematic diagram of a single DRAM cell.

(Q_1) must be switched on. Because the sense line is comparatively large it has a capacitance (C_L) significantly larger than C_B. Thus the change in voltage in the sense line is comparatively small according to equation 2.9.

$$\Delta v = V_{DD}\frac{C_B}{C_B + C_L} \tag{2.9}$$

This small voltage change is amplified by sense amplifiers at the end of the sense line. After the data has been read, the sense lines must be returned to their neutral state such that charge transferred to the sense lines does not interfere with later reads. This action is normally known as *precharging*. The capacitor C_B will slowly leak its charge over time due to leakage. In order to preserve data the cell contents must be read and then rewritten periodically by the system (*refresh*). This interval is typically in the order of once per millisecond [48]. The performance impact of refreshing can be neglible by using techniques to mask this periodic operation [42].

Static Random Access Memory (SRAM), on the other hand, is made entirely in transistors as shown in figure 2.2. The cell can be in two stable states: Either Q_1 and Q_4 are active, or Q_2 and Q_3 are active. In this figure the two transistors Q_A control access to the data stored in the cell, in a similar manner as Q_1 for DRAM cells. Because SRAM uses six transistors per bit, while DRAM only requires one transistor and a capacitor per bit, SRAM requires six to eight times as much area as DRAM [96]. SRAM has three advantages compared to DRAM: It has lower latency than DRAM, no need for refresh and can be built in the same logic-process used by high-performance processors. Thus, SRAM is often used for on-chip caches.

Embedded Dynamic Random Access Memory (eDRAM) offers a compromise. It uses a similar cell as regular DRAM. However, eDRAM can be integrated with the processor, because it uses the same logic-based process used in high-performance processors [61]. The use of a non-optimized process for eDRAM results in increased area requirements per bit, but the requirement is still much less than for SRAM [96]. This has led some researchers to investigate the possibility for using eDRAM as

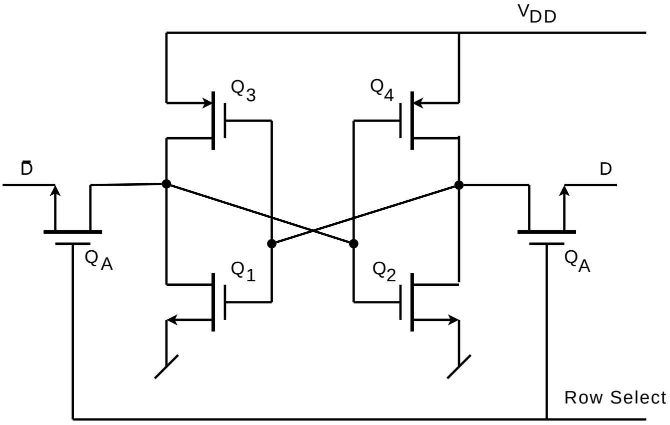


Figure 2.2: Schematic diagram of a single CMOS SRAM cell. Note that it is possible to construct other types of SRAM cells using other technology and number of transistors.

on-chip caches [139], scratchpad memory [8], or main memory [88].

Recently, 3D stacking has become an option for integrating DRAM on a chip. 3D stacking is a technique where multiple dies are stacked on top of each other and inter-die vias connect them [26, 88, 115]. This technique has two major advantages. First, it reduces average wire latency, because the dies are typically very close. Secondly, because the dies can be produced separately, each die can be produced in different production technologies, thus enabling mixing high-density DRAM processes with high performance logic processes [12].

2.2.2 DRAM Organization

In order to store more than a single bit, DRAM cells are organized in a matrix as shown in figure 2.3. The row address is first decoded into activating a single row in the matrix (Row Select). This in turn activates all DRAM cells in that row. Each DRAM cell outputs its content into the corresponding bit lines, which in turn is amplified by the sense amplifiers. Finally, the column decoder selects the relevant bits and the data is transferred to the processor.

The matrix organization at the core of DRAM has changed very little over the past couple of decades. However, there has been a number of significant improvements in the interface to the matrix. The most significant improvements are fast page mode, synchronous transfer and *Double Data Rate (DDR)* transfer [47]. However, as shown in figure 2.4 these improvements have for the most part improved memory bandwidth, rather than reduced memory latency [111].

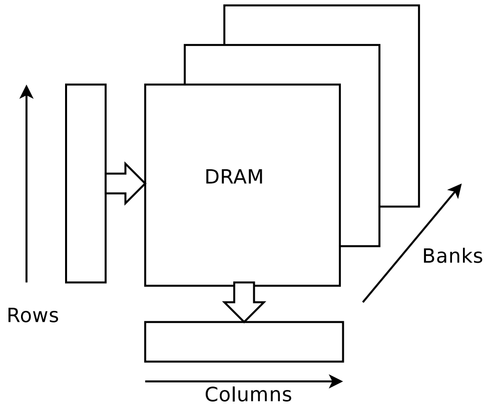


Figure 2.3: Organization of DRAM.

2.2.2.1 Fast Page Mode

Most programs exhibit spatial locality. Spatial locality is the tendency for programs to access data that are close in address space. To improve performance multiple columns are read by the sense amplifiers and stored in a row buffer [22]. Accessing data in this row buffer has a much lower latency as it bypasses the need for the original data from the DRAM cells. An access to data that is located in this buffer is often referred to as a *page hit*. In contemporary DRAM this row buffer is typically 1-8KB large.

The contents of this buffer is controlled by the memory controller. Leaving data in the buffer blocks the precharging of the bit lines, because the buffer is closely tied to the sense amplifiers. Thus, the memory controller has to make a trade-off between leaving the data in the buffer (open page policy) and precharging the bit lines (closed page policy) [2]. The open page policy lowers latency if there is a page hit. Conversely, the closed page policy lowers the latency if there is a page miss. Thus, the best policy depends on the amount of spatial locality in the execution of the program.

2.2.2.2 Synchronous DRAM

Up until early 1997, all DRAM was asynchronous [98]. In an asynchronous design there is no central clocking common to both the DRAM and the *Central Processing Unit (CPU)*. Instead, the bus was designed to use timing constraints and/or timing strobes. In particular, the CPU had to wait for one memory transfer to complete before issuing another memory request. Using a synchronous design (where the processor and DRAM module use a single master clock) made it possible to pipeline requests to different DRAM banks. A bank is essentially another DRAM matrix, which can be independently accessed (though multiple banks may share

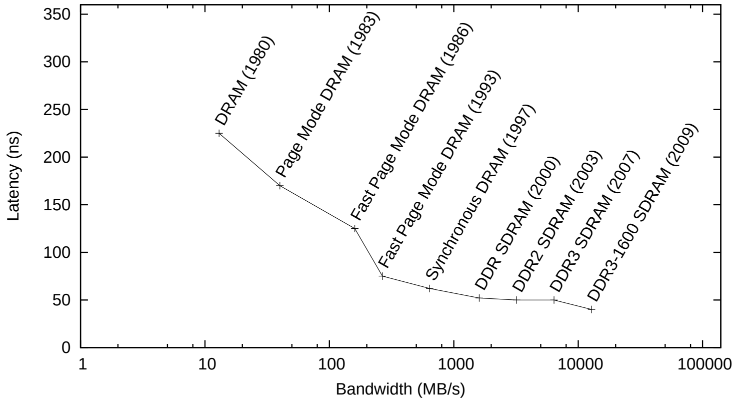


Figure 2.4: DRAM latency as a function of bandwidth for common consumer-grade main memory technologies [111, 146].

the same memory bus). Additionally, a synchronous design removed the need for timing strobes, which reduces latency [22]. Overall, this technique reduced latency dramatically, while increasing throughput at the same time.

2.2.2.3 Double Data Rate

The third major innovation in DRAM technology was the introduction of DDR DRAM. In DDR DRAM data is transferred on both edges of the clock, thus effectively doubling the bandwidth to main memory [25, 98].

2.2.3 DRAM Scheduling

DRAM controllers have typically served memory requests in a *First-Come First-Served (FCFS)* manner. Because of the increased complexity and parallelism in modern DRAM it is possible to increase performance or enforce memory fairness by reordering requests [104, 119]. Memory scheduling is currently a very active research field. The research focuses mostly on five distinct areas: Exploiting open pages, prioritizing critical loads, minimizing bank conflicts, increasing fairness and prefetch scheduling.

2.2.3.1 Increasing Page Hit Rates

Accessing an open page results in a page hit, which has a much lower latency than a regular operation. Rixner et al. [119] introduced *First-Ready First-Come First-Served (FR-FCFS)*. In FR-FCFS requests that use an open page are prioritized over other requests with the following priority rules:

1. Row-hit requests before row-miss requests.
2. Column commands over row commands.
3. Older requests before newer requests.

In *burst scheduling* multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization [123]. In addition, burst scheduling prioritize reads over writes to reduce access latency. Pending writebacks to an open page are serviced after all reads to this page have been serviced.

However, prioritizing reads over writes is not always beneficial. If writebacks are not serviced the write queue will become full and block the memory controller, which cascades through the memory system. To avoid this, it is possible to estimate the ratio of reads to writes, such that writes and reads can be prioritized accordingly [55].

Because of the complexity of DRAM scheduling, some researchers have examined the possibility of using *Reinforcement Learning (RL)* [58]. In this approach, the RL-agent senses the current state of its environment and executes an action. If the action is beneficial, it receives an reward, which reinforces the possibility of using the same action given the same state. Overall, the RL-agent tries to maximize it's reward over the long term. In DRAM scheduling, a high data bus utilization represents a reward, while the possible commands and their attributes are the state.

2.2.3.2 Memory Criticality

Another important aspect when scheduling DRAM accesses is that not all memory requests are equally important to the performance of a program. By predicting which loads are more important than others it is possible to prioritize these loads over other requests and thus increase performance. One possibility for predicting load criticality is to examine the *Reorder Buffer (ROB)* and *Instruction Queue (IQ)* [47] occupancy status [153].

One of the biggest bottlenecks in modern processors is off-chip memory. Because modern memory interfaces can handle multiple simultaneous memory requests, it is critical for performance to exploit this property. *Memory Level Parallelism (MLP)* refers to the system's ability to issue multiple overlapping memory requests simultaneously.

Batch scheduling increases both page hit-rates and MLP by using batches [100]. A batch is formed when the previous batch of requests is completed. All memory requests in a batch are serviced before any other request. This strategy makes starvation impossible and increases fairness. In addition, higher priority processes (either set by the operating system, or by a heuristic) is serviced first. This ensures that MLP is increased by ensuring that all requests from a given process are serviced as simultaneously as possible. The priority rules for batch scheduling are:

1. Requests within the current batch before any other requests.

2. Row-hit requests before row-miss requests.
3. Requests from higher-priority threads before requests from lower priority threads.
4. Oldest request before newer requests.

2.2.3.3 Minimizing Bank Conflicts

Bank conflicts are one of the main reasons for reduced memory bus utilization in modern, high-bandwidth memory interfaces. Therefore, a number of researchers have looked into how these conflicts can be reduced by changing the way memory addresses map on to banks. For instance, bit-reversal mapping results in a high probability of placing two adjacent rows in different DRAM banks [124]. Consequently, high row buffer hit rates are achieved at the same time as the probability of bank conflict is reduced.

2.2.3.4 Fairness

In CMPs, the memory bus is shared between all processing cores. This can cause unfairness as one high locality thread can effectively starve other threads, or get an unfair portion of off-chip bandwidth. A number of researchers have looked into how the off-chip interconnect can be shared in a fair way [60, 101, 108, 117]. In general, these techniques divide bandwidth among threads according to their priorities at the same time as requests are scheduled in a way that improves DRAM throughput.

2.2.3.5 Prefetch Prioritization

Prefetching (section 2.4) consumes bandwidth. Prioritizing prefetches and demand requests equally can thus delay a useful demand request and cause memory bus congestion [85, 107]. However, prioritizing demand requests over prefetches diminishes the usefulness of prefetching, because the prefetches are issued too late or not at all.

One approach to this problem is to use a dedicated prefetch queue which holds prefetches that are ready to be issued [85]. Then, the memory controller can adaptively chose to issue these prefetches depending on the estimated accuracy of the prefetches. Thus, in a scarce bandwidth situation with an estimated low accuracy, the memory controller can simply ignore the prefetch requests. In a high accuracy situation, it can chose to prioritize reads and prefetches equally which in turn can result in higher page-hit ratios.

2.3 Cache

Caches are the most important technique in bridging the processor - memory gap. Conceptually, caches duplicate data from main memory into smaller and faster storage [128]. Because of *spatial* and *temporal locality*, the data needed by the processor is often found in caches [47]. Typically, caches form a part of a larger memory hierarchy as shown in figure 2.5. In this figure there are two levels of cache between main memory and the CPU.

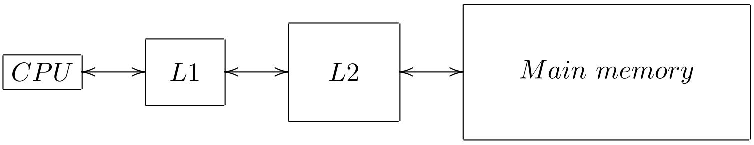


Figure 2.5: Example of a memory hierarchy with 2 levels of cache.

The fastest type of storage is the registers within the CPU itself. Next, the L1 cache is typically 32 – 64Kb large and has a latency of 2-3 clock cycles. The L2 cache is typically 512 KB – 16 MB large and has a latency of about 20 clock cycles. Equation 2.3 shows the overall system latency for this organization[3].

$$L_{system} = L_{L1} + p_{L1\ miss} \cdot (L_{L2} + p_{L2\ miss} \cdot L_{Main\ Memory}) \tag{2.10}$$

Because of spatial and temporal locality, the probability of **not** finding the required data in the first level of cache is quite low ($p_{L1\ miss}$), even though it is quite small compared to main memory. This decreases the second term in the equation leading the average overall memory latency (L_{System}) to be low. Increasing the size of the cache also increases the probability of a cache hit. However, increasing the size also increases it's latency as shown in figure 2.6. Furthermore, increasing the size also increases the energy requirements significantly.

2.3.1 Set-associative caches

There are several ways to build a cache in hardware. Because caches can only hold a small portion of main memory at any point, some way to map main memory to cache is needed. To exploit *spatial locality* a cache usually stores data in chunks called cache blocks (lines), which are typically larger than the wordsize of the machine.

[3] This model, like the model in equation 1.1 assumes no virtual memory and infinite cache bandwidth. In addition, in most implementations the latency of a cache miss is different from a cache hit.

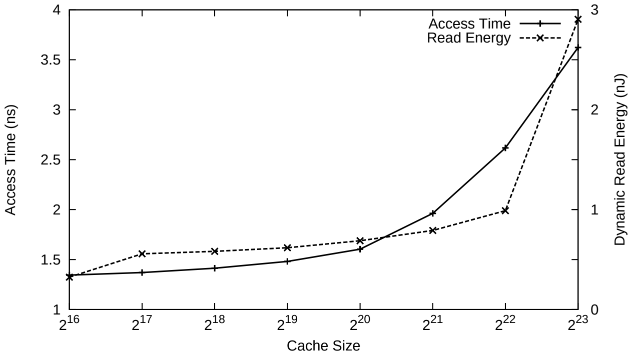


Figure 2.6: Cache access time and energy as a function of size. CACTI [127] was used to model this 4-way associative cache with 64B cache lines.

A cache can be organized in several ways:

- *Direct mapped* - A cache block can only be placed in one position based on its address.
- *Fully associative* - A cache block can be placed anywhere in the cache.
- *Set associative* - A cache line can be placed in exactly one *set*. Each set can hold n cache blocks. If there are n cache blocks per set, the cache is called n-way set associative.

In essence, a direct mapped cache can be viewed as a 1-way set-associative cache. Similarly, a fully associative cache can be viewed as a set associative cache with only one set.

Figure 2.7 shows how cache lookup is performed in a set-associative cache. The address is split into three parts: the tag, the index and the offset. The index is used to index two SRAM arrays, the tag array and the data array. Since this is a two-way set associative cache, each index holds two tags. The tags from this array is compared to the tag portion of the address. If either tag matches, the data is in the cache (cache hit). The corresponding data line is then brought out of the data array by using a multiplexer. Since the cache lines are typically longer than the size of a word, the offset is used to further select what data from the cacheline to forward.

2.3.2 Cache Misses

There are three main reasons why data is not found in the cache. These are:

Definition 1 (Compulsory [47])*:*
The very first access to a block cannot be in the cache, so the block must be brought

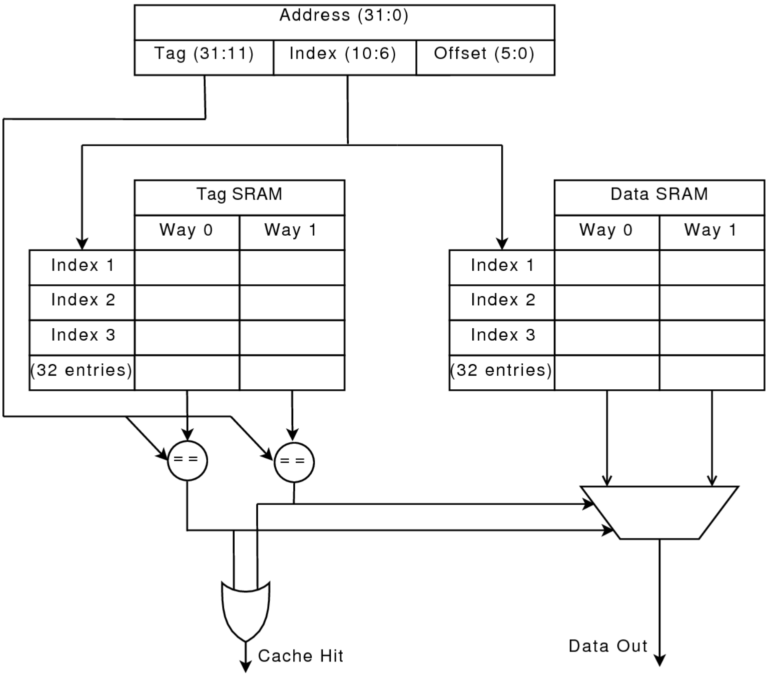


Figure 2.7: Organization of a 4K two-way set associative cache with 64B cache lines.

into the cache. These are also called cold-start misses or first-reference misses.

Definition 2 (Capacity [47])*:*
If the cache cannot contain all the blocks needed during the execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.

Definition 3 (Conflict [47])*:*
If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses or interference misses.

In a multiprocessor where data is shared between processors or cores, there is a fourth type of cache miss called *coherence miss*. A coherence miss occurs due to cache flushes to keep multiple caches coherent in a multiprocessor [31, 47, 140]. As an example, consider a two core CMP with separate private caches: Both cores reads the value of variable X from main memory. The value of X is then stored in both private caches. Core 1 then proceeds to modify X and stores the value. Now the value of X in main memory and core 2's cache differs from the value in core 1's

Block Address	Target Information	Valid Bit

Figure 2.8: Conceptual MSHR entry

cache. These values are said to be *invalid*. An access by core 2 to X would then cause a *coherence miss*.

2.3.3 Replacement Policies

After a cache miss new data is inserted into the cache. However, because of the cache's limited capacity other data must be removed from the cache. There are several possible replacement policies such as: *Least Recently Used (LRU)*, *First In, First Out (FIFO)* and random [128]. The most common replacement policy for general purpose processors is LRU, where the least recently accessed cache block is removed.

With the increased interest in CMPs and *Non-Uniform Cache Architecture (NUCA)* there has been revived interest in cache replacement policies. Most cache misses are not performance-critical. In many cases, execution can continue regardless whether the load is a cache hit or miss. By reducing the number of isolated performance-critical cache misses, it is possible to increase the amount of MLP and performance [116]. When a cache block is evicted in NUCA, it can be moved to another cache. In that case, a policy for selecting a new cache is needed [32, 34].

2.3.4 Miss Status Holding Registers

Processors which can execute instructions *Out-of-Order (OoO)* has the potential to issue multiple independent loads. To support this capability, caches need to be able to service more than a single access at a time. In particular, it must be able to handle multiple misses. To achieve this, it must keep an account of which misses are being serviced further down the memory hierarchy [80].

Miss Status Holding Registers (MSHRs) can be used for this purpose. A conceptual MSHR is shown in figure 2.8. A cache can sustain as many misses as there are MSHRs without blocking. Each MSHR holds the address that is being serviced and the target information for that miss. The target information is mainly what instruction caused the miss, and thus which instruction is waiting for the data and the destination register.

2.4 Prefetching

Prefetching is a technique to reduce the number of misses in a cache through predicting future memory references and fetching the corresponding data before it is referenced by the CPU. It is especially effective for reducing *compulsory* misses, as caches only retain previously referenced data. This can potentially speed up execution significantly as p_{miss} decreases and L_{system} decreases. However, because prefetching is a speculative technique some prefetched data will not be used, which causes cache pollution and increased bandwidth usage. A *good* prefetch is defined as:

Definition 4 (Good prefetch [136])*:*
A prefetch is classified as good if the prefetched block is referenced by the application before it is replaced or bad otherwise.

A useful metric for dealing with prefetching is *accuracy*. Accuracy is a metric for how often the prefetcher's prediction is correct:

Definition 5 (Accuracy [136])*:*
The accuracy of a given prefetch algorithm that yields G good prefetches and B bad prefetches is calculated as:

$$Accuracy = \frac{G}{G+B} \tag{2.11}$$

It is not enough for a prefetcher to be accurate if the prefetches are issued too late. A prefetch must be issued sufficiently in advance so that it can be inserted into the cache before it is referenced. This property is known as *timeliness*.

However, high accuracy and timeliness is not enough to ensure high performance. A significant portion of the program's original cache misses must be eliminated to increase performance. This is covered in the *coverage* metric:

Definition 6 (Coverage [136])*:*
If a conventional cache has M misses without using any prefetch algorithm, the coverage of a given prefetch algorithm that yields G good prefetches and B bad prefetches is calculated as:

$$Coverage = \frac{G}{M} \tag{2.12}$$

2.4.1 Sequential Prefetching

The simplest prefetching scheme is sequential prefetching [128]. Sequential prefetching simply fetches the next cache block when a cache block is accessed. Although this policy is simple, it is very effective because of sequential locality. Because processors are much faster than main memory it is in practice necessary to fetch blocks further away than the next block such that the data is ready when the processor needs it. This is known as the prefetch distance:

Definition 7 (Prefetch distance [143])*:*
If a loop contains small computational bodies, it may be necessary to initiate pre-

fetches δ iterations before the data is referenced where δ is known as the prefetch distance and is expressed in units of loop iterations:

$$\delta = \left\lceil \frac{l}{s} \right\rceil \tag{2.13}$$

l is the average cache miss latency, measured in processor cycles and s is the estimated cycle time of the shortest possible execution path through one loop iteration.

Additionally, it might be beneficial to fetch multiple blocks at the same time. This parameter is the prefetch degree:

Definition 8 (Prefetch degree [143])*:*
It is possible to increase the number of blocks prefetched by any arbitrary number K. This number is known as the prefetching degree. As an example; a prefetching degree of 1 fetches 1 block from memory, while a prefetching degree of 3 fetches 3 blocks from memory.

These two parameters are often collectively referred to as the prefetcher's *aggressiveness*. Increasing coverage usually comes at the expense of accuracy. A good prefetching scheme must thus balance the aggressiveness of the prefetcher to ensure a good trade-off between accuracy and coverage within the system's limited off-chip bandwidth and cache capacity.

Tagged prefetching is a simple improvement over sequential prefetching [142]. In this scheme prefetched data is marked with a single bit in the cache. When this block is accessed the prefetcher knows that the previous prefetch for this data was successful and can initiate a request for the next line. This information can also be used to estimate prefetcher accuracy [135].

In most implementations, the prefetched data is inserted directly into the cache, which can cause useful data to be evicted. Another option is to use dedicated structures to hold the prefetched data such as stream buffers [71, 110]. A stream buffer is a structure that holds prefetched data which can be tailored to the type of prefetcher used. It is typically accessed in parallel with the main cache.

Another possibility is to predict which blocks in the cache is not needed anymore [82]. This information can be used to initiate prefetching for a new block to replace the old block, or it can be used to decide which block to replace when inserting new prefetches.

Additionally, in a CMP or multiprocessor system prefetching might cause invalidation of cache blocks in other cores [66]. For example, core 1 might hold an exclusive copy of a variable X, when core 2 decides to prefetch that block into it's own cache. This forces core 1 to downgrade it's copy of X to a shared state. However, if core 1 modifies X later, core 2's copy must be invalidated which can decrease performance. A possible solution is to use instruction based sharing prediction to guide when to prefetch shared data, or simply avoid prefetching shared data [75].

2.4.2 Instruction-Based Prefetchers

When the processor's referencing pattern strides through nonconsecutive memory blocks, sequential prefetching will cause needless prefetches and will thus become ineffective [142]. An initial approach to this was *Stride Directed Prefetching (SDP)* [40]. SDP has a table indexed by the load address as shown in figure 2.9. When a load instruction is first encountered, its *Program Counter (PC)* and the data address is stored in the table and the valid bit is set. The second time the instruction is encountered the stride (delta) between the current data address and the stored value is computed. Finally, the current address plus the stride is prefetched.

PC Address	Last Address	Valid Bit

Figure 2.9: Format of a Stride Directed Prefetching entry.

An improvement over SDP is *Reference Prediction Tables (RPT)* [18, 23]. RPT extends SDP by adding state information as shown in figure 2.10. Several variants to the basic state machine has been proposed [23]. The basic principle is to use an initial state when a load is first encountered. On the next miss, the stride between the first miss address and the current is computed and stored in the table, and the entry enters the training state. On the third miss, a new delta is computed. If that delta matches the one found in the table, the entry enters the prefetching state and prefetches are issued by using the computed delta.

PC Address	Last Address	Stride	State

Figure 2.10: Format of a Reference Prediction Table entry.

A further enhancement is the use of a *Global History Buffer (GHB)* [106]. A GHB is essentially a FIFO containing the last misses observed by the memory system as shown in figure 2.11. Each entry in the GHB is linked to the previous entry which originated from the same load instruction by a pointer. By traversing the linked list a miss history can be obtained for that load. In *Program Counter/Delta Correlation Prefetching (PC/DC)* [106], the deltas between consecutive misses are computed and stored in a delta table as shown in figure 2.11.

After the history of deltas are computed, delta correlation begins. Delta correlation means searching for the most recent pair of deltas (9 and 1 in figure 2.11) in the delta history. In this example, the pair can also be found at the end of the delta history (top of the delta table). The deltas after the pair are then added to the current miss address, and prefetches are issued for the calculated addresses.

Further refinements have been proposed, such as *Global History Buffer - Local Delta Buffer (GHB-LDB)*, which improves upon the GHB prefetcher by also including global correlation (as opposed to the local correlation directed by the PC of the load) [30]. By doing global analysis, inter-load patterns can be seen, such as constant global stride. In addition, GHB-LDB includes pattern matching for the most common stride.

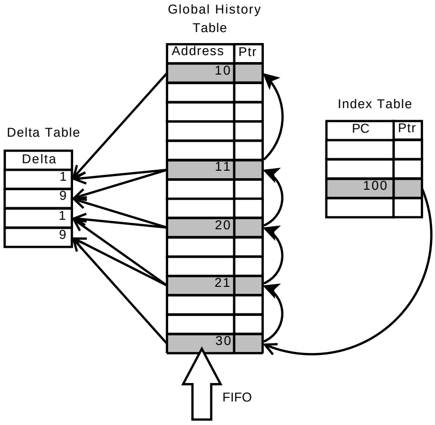


Figure 2.11: Construction of the delta table in PC/DC.

A third variant is *Prefetching based on a Differential Finite Context Machine (PDFCM)*, which uses a hash-based approach with two tables [118]. The History Table is indexed by the PC which contains a hashed representation of the recent history of that entry. This hash points to an entry in the Delta Table which contains the predicted delta. By computing new hashes based on the predicted deltas, an arbitrary prefetch degree and distance can be calculated.

2.4.3 Address-Based Prefetchers

Instruction-based prefetchers have been shown to be very effective [114]. The problem with instruction-based prefetchers is that they require the load address, which either requires that the load-address is transmitted along with the memory request or coupling the prefetcher with the processor core. A third option is to not use this information at all and only use the miss address.

Markov Prefetchers uses a 1-history Markov model in order to predict future references [70]. Such a model uses a graph where each node represents a cache block. Each transition from node X to node Y is assigned a weight representing the fraction of references to X that are followed by a reference to Y. When X is accessed, then the outgoing edges from X is examined. The weighting on the edges can be used to reject or accept prefetching to the node which the edge points to. However, keeping an entire Markov model in memory would require $O(n^2)$ amount of storage as each cache block can theoretically be preceded by every other. Thus, a practical implementation must limit the number of nodes and edges per node [4] [70].

Hu et al. observed that the misses that occurred within a cache set had highly repetitive patterns when looking at the tag portion of the miss addresses [49]. The

[4] The original paper used 4 edges per node.

per-set tag sequence for one set would often repeat on other cache sets. This observation is used in *Tag Correlating Prefetching (TCP)* to reduce the size of tables, because a single tag sequence covers multiple address sequences (one for each set in the cache).

A very common pattern in media applications (video and audio) is sequential data being used for computation and not used again. This type of access pattern is often called *streaming*. The basic stream prefetcher [71] starts prefetching when it detects such streams by detecting sequential miss addresses. The stream prefetcher will then prefetch the next sequential addresses into a *stream buffer*. This basic operation can be extended by using any of the techniques discussed above, such as stride detection and Markov prediction [125].

2.4.4 Spatial Locality Prediction

Another approach is to detect when there is a high level of spatial locality in the program [69]. When the program has high spatial locality it is potentially beneficial to fetch more than a single cache line into the cache. One approach for detecting high spatial locality is to have a separate tag-array that mimics a cache with larger cache blocks. If there are more than a set threshold of hits to the same larger virtual cache block, then there is a high probability of high spatial locality and a larger block of data can be prefetched into the actual cache [69]. Rather than using a separate tag-array, it is possible to use a smaller table of bitvector or offsets to represent the same information [17, 78]. By indexing these patterns with the PC of the load, it is possible to use this bitvector when that load misses and prefetch according to the bitvector.

Taking these ideas further, *Spatial Memory Streaming (SMS)* uses code correlation across loads [131]. In this approach, an initial *trigger access* to a spatial region starts recording subsequent accesses to the same spatial region. The blocks that are touched are stored in a bitvector representing the spatial region. The recording stops when the first cache block from the spatial region is evicted from the primary cache. This pattern can then be used to prefetch large spatial blocks. The memory requirements of SMS can be quite large, but it is possible to compress the size of the tables by using rotated patterns [39].

Streams of memory also exhibit temporal locality (i.e. the exact same sequence of addresses are observed in succession) [132]. This observation is exploited in *Temporal Memory Streaming (TMS)* by storing the observed miss address stream in a circular buffer and using it to detect repeating patterns [132]. This approach is especially useful for programs using shared memory.

Access Map Pattern Matching (AMPM) uses another approach where the patterns are stored in bitvectors [59]. The key observation is that modern compilers can obfuscate memory access orderings, especially when loop unrolling is performed. Consequently, more patterns can be discovered by ignoring temporal information. Each spatial region is tracked by using a 2-bit vector for each cache line in that region. This vector is analyzed to see if there are any constant stride patterns in that vector. If there are any patterns, the predicted pattern is prefetched.

2.4.5 Linked Data Prefetchers

Linked data structures pose special problems for prefetchers as the address of the next cache block to be referenced is often embedded in the data currently being fetched. One approach, called *Content-Directed Prefetching (CDP)*, is to monitor the incoming data for possible pointers [21]. CDP uses a virtual address matching predictor, which examines each word in the cache block separately. Most virtual addresses share the same common high-order bits. If a value in the incoming cache block has the same high-order bits as the address of the block, then the value is predicted to be a pointer. However, this approach can lead to many false positives (values which are identified as pointers, but are not), and it has a potential for creating an exponential amount of prefetches as each prefetched block can contain multiple new pointers (for example in trees) [35]. A possible solution to this problem is to use compiler-generated hints which indicate which positions in the incoming cache blocks are possible pointers [35].

However, because the prefetcher must follow the same linked list as the main program it is difficult to achieve good timeliness. One possibility is to embed jump-pointers into the data structure [73, 121]. These pointers point to an entry further down the linked list, such that this entry might be prefetched. However, maintaining these jump-pointers can be difficult when data is inserted and removed.

A less invasive possibility is to use pointer caches [20, 83]. A pointer cache holds mappings between heap pointers and the address of the heap object they point to. If there is a hit in the pointer cache and a miss in the regular data cache, the entire object can be fetched at the same time. Furthermore, this technique can be used for value prediction, which in combination with runahead execution [102] can be very effective at increasing MLP [20, 103].

2.4.6 Adaptive Prefetchers

Most prefetchers have a static configuration with regard to prefetch distance and degree as well as other prefetcher-specific parameters. To provide the best average performance across a multitude of benchmarks a moderately aggressive prefetching configuration would be used. This leads to performance degradation on some programs, as the prefetching is too aggressive, leading to memory bus congestion and cache pollution. On other programs, the full potential of prefetching can not be achieved because the aggressiveness is too low. Consequently, adapting the prefetching parameters to the running program by analyzing its behaviour can be beneficial.

By using the prefetch tags the accuracy of the prefetcher can be estimated. This is done by measuring how many times a cache block with the prefetch bit set is accessed relatively to the total number of prefetches issued. [24, 135]. If the prefetcher's accuracy is estimated to be high, then the aggressiveness of the prefetcher can be increased. Similarly, the timeliness of the prefetcher can be estimated by tracking which prefetches have been issued, but have not been completed before a demand miss occurs. If the timeliness of the prefetcher is determined to be low, then the prefetch distance can be increased [135].

An alternative approach is to track program phases [107]. If a program phase change occurs, a search for a better prefetcher configuration is initiated. This search continues until a good configuration is found [7, 126]. Multiple techniques for detecting phase changes exists, such as examining instruction working sets, basic block vectors[5] and conditional branch counts [28, 29].

One problem with stream prefetchers is that the length of the stream is unknown. Overestimating the stream length leads to useless prefetches, while underestimating it lowers performance. The length of a stream can be predicted by using a histogram of previous stream lengths [53, 54].

2.4.7 Runahead Execution

OoO execution can only hide a certain amount of latency before the instruction window of the processor becomes full and the processor must stall. Runahead Execution allows the processor to continue execution speculatively while the processor waits for a long latency load [102, 103]. To enable a processor to run speculatively it must have capabilities for checkpointing its state such that when the processor unblocks from the memory stall it can restore its non-speculative state. While the processor runs in this speculative mode it encounters loads further down in the program, which can then be issued speculatively as prefetches. This increases the probability of the loads that were encountered during runahead mode will be in the cache when the processor resumes normal execution.

2.4.8 Software Prefetching

Prefetching can be accomplished in software by extending the instruction set with appropriate opcodes [16]. These instructions fetch data into the cache, but do not block execution of the program on a cache miss. Most modern high-performance processors incorporate such instructions [74]. These instructions can either be inserted manually by the programmer [122] or by the compiler [19]. This type of prefetching can be especially useful in programs where the programmer has essential information regarding the underlying data structure. Using this knowledge she can provide hints to the compiler or insert prefetch instructions directly [90].

[5] Basic block vectors is a method for characterizing the currently executing program using performance counters etc.

Chapter 3

Research Process and Methodology

Science is a way of trying not to fool yourself. The first principle is that you must not fool yourself, and you are the easiest person to fool.
– Richard Feynman

3.1 Research Process

3.1.1 Master Thesis

This thesis is a continuation of my master thesis entitled *"Bandwidth-Aware Prefetching in Chip Multiprocessors"* [44]. Because prefetching can potentially consume large amounts of bandwidth, this work examined how prefetching impacts systems with limited bandwidth. Furthermore, I proposed a strategy for estimating future bandwidth usage and used this information to guide how prefetches were scheduled.

I implemented a general framework for prefetching in SimpleScalar [5] and several prefetching heuristics (sequential [128], RPT [18] and *CZone/Delta Correlation (C/DC)* [106]). SimpleScalar has a very simple DRAM model, which does not model bandwidth contention at all. Instead, it only provides a fixed latency regardless of the number of concurrent DRAM accesses. I extended this simple model with a more accurate and realistic DRAM model. Most importantly, this model supported contention and open pages. Because of sequential locality, the probability of hitting an open page when prefetching is high, which reduces the latency of prefetching.

Furthermore, I extended SimpleScalar such that it was capable of simulating a

CMP. This was accomplished by using several SimpleScalar instances which communicated through shared memory.

3.1.2 Destructive Read DRAM - Paper I & II

Our two first papers were inspired by a paper by Hwang et al. [56]. This paper described a destructive read DRAM macro, which had lower latency than a regular DRAM macro. However, when data was read out of the cell, the contents were destroyed.

In order to preserve data, some other mechanism is needed. Hwang et al. used a large (25% of the size of main memory) writeback buffer. Because there were 4 main memory banks in their design, this size ensured that the system would never need to issue both a writeback and a read to the same main memory bank at the same time.

Haakon Dybdahl was a senior PhD student at the time. He became interested in this technique and we started to discuss ways to improve upon it. Our idea was to use the cache as the writeback buffer. By using an existing structure, we would eliminate much of the area overhead. However, this would also introduce some contention in the main memory interconnect.

Haakon integrated parts of the memory model I developed for my master thesis into his model of destructive read DRAM and we started discussing methodology. Haakon conducted several experiments and wrote the first draft of the paper, which I commented on and improved.

After writing Paper I it became obvious that a power estimate would be valuable. Haakon integrated Wattch [13] and HotLeakage [152] into our simulator. However, we did not know how to estimate the power requirements of our technique and asked Per Gunnar Kjeldsberg for assistance. He helped us develop a power model for accessing the destructive read DRAM and interpret the results. This work resulted in Paper II.

3.1.3 Shadow Tags - Paper III

After completing the destructive read DRAM project, Haakon started to examine cache replacement policies in CMPs. He developed a cache replacement policy which worked better than LRU in some cases, while degrading performance in others [33]. In order to detect at runtime when to use his replacement policy and when to use LRU Haakon used a shadow tag directory[1].

A shadow tag directory is similar to a regular cache tag directory. However, there is no corresponding data for that tag directory. The purpose of this tag directory

[1] Interestingly, Qureshi et al. developed the same approach to increase MLP at the same time [116].

is to simulate a cache with a different replacement policy. The L2 access stream would be inserted both into the shadow tag directory and the regular cache. The number of cache hits and misses are recorded for both the regular tag directory and the shadow tag directory. The regular cache would use one replacement policy and the shadow tag directory would use the other. After a set interval of cache misses, the number of cache misses in the two tag directories are examined. If there are less misses in the shadow tag directory, the policies are swapped, such that the replacement policy that was used on the shadow tags is now used on the regular cache and vice versa.

Shadow tags was interesting, because it offered a more robust method for evaluating prefetcher configurations with regard to accuracy and bandwidth utilization than the methods used in my master thesis. The idea was to use a shadow tag directory to evaluate a particular prefetch configuration and use that configuration if it proved to be better than the current one. In particular, this allowed the prefetcher to be turned off if that was the most effective solution.

There were two main obstacles: The first is that there is a very large configuration space for the prefetchers. In order to find a good prefetch configuration, a good candidate prefetch configuration had to be chosen. The initial approach used hill-climbing, but that approach was slow to converge. The final paper used a random configuration.

The second problem was evaluating which configuration was the better of the two configurations being explored. Naively selecting the configuration with the fewest misses leads to selecting the most aggressive prefetch configuration as there is no penalty for issuing DRAM requests. To solve this problem I was inspired by *Genetic Algorithms (GA)* to use fitness functions to evaluate configurations. This fitness function balanced the reduction in misses to the increase in bandwidth usage. This approach was published in Paper III.

Finally, Sigmund Vinsnesbakk took this idea further by implementing shadow tag prefetching in CMPs by using M5 [10] for his master thesis [145].

3.1.4 Changing Simulators

After working with SimpleScalar for several years, the limitations of that simulator became apparent. In particular, the support for CMPs and the comparatively simple model of the memory hierarchy (no limitations on bandwidth, no MSHRs) became an issue. The biggest issue was that rather than being event-driven the memory hierarchy model was just one function call to compute the latency of a memory request. This meant that it was very difficult to model reordering of memory requests efficiently, which is critical to performance of modern DRAM controllers.

In parallel with my work on Paper III, Arnt Jørgen Lande did an evaluation of several simulators for his master thesis [84]. His initial evaluation included Rsim [51], Asim [36], SimOS [120], Simics [93], TFSim [97], SimFlex [46], GEMS [95] and M5 [10]. Based on his initial evaluation he made a more thorough evaluation on M5 to examine its suitability for our research group's needs. After this evaluation, the group decided to switch to the M5 simulator as it offered CMP support and an event-driven memory hierarchy model.

3.1.5 Prewriting

Although M5 provided many of the features we required for our research, the DRAM model was quite poor. Although bandwidth congestion was modeled, DRAM behaviour such as page-hits, banks and minimum precharge-to-activate latency was not modelled. Additionally, each memory access was scheduled in arrival order, which is a very simple scheduling policy and does not exploit open pages.

Magnus Jahre and I started to implement a DRAM model by examining the DDR2 specification [65]. This implementation used explicit activation, reads, writes and precharge commands. It supported limitations on the number of banks that could be activated, open pages, minimum activate-to-precharge latencies, pipelining of memory request and many other improvements. Secondly, we implemented a flexible memory controller, which could use FR-FCFS [119] and *Network Fair Queuing (NFQ)* [108] scheduling in addition to FCFS scheduling.

While implementing the memory controllers for M5 we noticed that scheduling writebacks was an interesting problem. As noted in section 2.2.3.1 prioritizing demand reads over writebacks pays off as the processor is stalled for a shorter period of time. However, this strategy breaks down if the writeback queue becomes full and the memory controller has to block [123].

We started examining this problem and came up with an idea to speculatively write back dirty cache lines before they were evicted (*"prewriting"*) if there was sufficient bandwidth to do so. However, in practice, this was not a significant problem for the SPEC2000 benchmark suite as the writeback buffer was big enough to keep the controller from blocking. Finally, Lee et al. had published a study on *eager writeback*, which used a similar approach [86]. They showed that *eager writeback* could speed up applications with large numbers of writebacks such as software 3D rendering. This previous study and the lack of good results led to the project's abandonment.

3.1.6 Low-Cost Open-Page Prefetch Scheduling - Paper IV

While working with memory scheduling it became apparent that exploiting page hits was critical for reducing latency and bandwidth congestion. Since prefetchers often prefetch data which are spatially close, this would often lead to page hits. Thus prefetching while a page was open could both decrease latency and increase effective bandwidth. Because a prefetch that is issued to an open page is cheaper (in terms of latency and bandwidth utilization), prefetching accuracy can be low, while still providing a net gain as shown in equation 3.1.

$$\text{Prefetching Accuracy} \cdot \text{Cost of Prefetching} < \text{Cost of Single Read} \tag{3.1}$$

To investigate this further I modified M5 further to include prefetching of several well known prefetching heuristics (sequential [128], RPT [18] and C/DC [107]). I examined several strategies for issuing prefetches, but the key insight came when I plotted prefetch accuracy vs speedup for all the prefetchers in a single graph (See figure IV.3 in paper IV). This graph shows that when the prefetching accuracy was around 40%, equation 3.1 would be balanced. Thus by using different scheduling strategies depending on the prefetcher's accuracy performance could be improved. This work was published at *International Conference on Computer Design (ICCD)* in 2008 (Paper IV).

A couple of months after the presentation of this work at ICCD, Lee et al. published a related study at MICRO [85]. Their work explored using both a dedicated prefetch queue and inserting prefetches directly into the read queue.

3.1.7 Data Prefetching Championship - Paper V & VII

Garzia Perez et al. published a comparative survey of many proposed prefetching heuristics in 2004 [114]. That paper showed that the evaluation of prefetchers could give significantly different results depending on the benchmark and simulator. They also found that evaluating different prefetching schemes was difficult, because the papers describing these techniques often lacked significant details which were important to the implementation.

In order to address these issues the *Journal of Instruction-Level Parallelism (JILP)* organized a *Data Prefetching Championship (DPC)* similar to the earlier branch prediction championships. The idea was that all contestants would use the same simulator to implement their prefetching heuristics. This simulator was provided in binary form, with a simple, well-defined interface that could only be used for prefetching. Each contestant would submit their prefetching code (2 files) to the competition organizers. The organizers would then run the simulator with their own benchmarks. This would ensure a fair comparison of the prefetching heuristics.

During my work with prefetching I had examined several prefetchers and knew their strengths and weaknesses. In particular, Delta correlation proposed by Nesbit et al. [106] is very effective and has a very high accuracy. However, the *Global History Buffer (GHB)* which is often used in combination is not as effective. Because the GHB acts as a FIFO some load-instructions would have much history associated with them, while others would have less. Much of the information contained would also be useless, because there is a limit to the amount of history that is useful in generating new prefetches. Although delay was not modelled in the competition a GHB would require recomputation of the deltas, which would increase cost and delay the prefetches.

Our submission (Paper V), called *Delta Correlating Prediction Tables (DCPT)*, used a table indexed by the PC of the load. This table contained the last n deltas observed by that load which could then be used to issue prefetches by using delta correlation. Because each entry has a fixed size this ensures that the amount of history per entry can not decrease. This property is useful, because this in turn ensures that delta correlation does not produce overlapping prefetches. In addition, because we used a table based approach more state could be associated with each entry. We used this to track the last issued prefetch. This is very useful for eliminating redundant prefetches.

Our submission was awarded 4th place (out of 20 submissions). There were many original and interesting submissions to the competition. Second place was awarded to Dimitrov et al. which used an approach which was very similar to ours [30]. Their approach also used delta correlation, but rather than using a GHB or a table they used a hybrid approach and prefetched into the L1 rather than the L2.

The contestants code was later published on the DPC website [67]. By looking at the contestants code it was clear that our biggest design mistake was the lack of L1 prefetching. In the design of the DCPT we abandoned L1 prefetching too early as our initial experiments showed that the L1 was very sensitive to pollution. We developed a technique called *L1 hoisting* to address this issue. L1 hoisting predicts which data that has been prefetched into the L2 will be used by the processor in the near future and moves it to the L1. Another problem with the design was the lack of handling for pointer chasing or partially irregular patterns. This part was addressed by using *partial matching.* If a pattern is not found using regular delta correlation, then partial matching reduces the spatial resolution by removing the least significant bits from the delta stream. This exposes more patterns, which can then be prefetched. The improved prefetching heuristic is called DCPT-P and is presented in Paper VII.

After the competition I was invited to Ghent by Veerle Desmet to implement DCPT in Unisim [4]. Their ongoing Archexplorer online competition is similar to DPC. Archexplorer tries to find good memory systems by randomly combining known techniques in a memory system hierarchy using random parameters [27]. Each generated memory hierarchy is evaluated in terms of performance, area and power. The port to Unisim was successful, but the processor model (embedded PowerPC core) and area requirements (The baseline configuration has only a very small L1 cache) is not very well suited for prefetching. The team is now moving the competition to a more aggressive processor model.

3.1.8 3D Stacking

I became interested in 3D stacking after reading Gabriel H. Loh's paper on 3D stacked memory architectures [88]. 3D stacking is a technique for stacking dies vertically (see section 2.2.1). The bandwidth between layers can be quite high, because inter-die vias can be densely packed [26]. Furthermore, the dies can be in different technologies. This enables mixing of dies optimized for logic and for DRAM, thus eliminating many of the drawbacks of eDRAM. Loh's paper examined how the very large inter-die bandwidth can be exploited in a 3D architecture. Latency is very low compared to a off-chip solution, because main memory and processor can be integrated on the same chip.

I wanted to explore the possibilities for destructive-read DRAM and prefetching. Using destructive-read DRAM would further reduce the latency of a memory operation. Prefetching, on the other hand, could exploit the large amount of bandwidth available to further decrease latency.

My initial research used a modified version of SimFlex [46] which included a power model. Modelling power and thermal effects is important in 3D architectures, because using multiple layers increase power density, while heat dissipation becomes more difficult [89]. To model thermal effects I started using HS3D [87]. HS3D is based on Hotspot [50] and models thermal effects in 3D stacked architectures.

I made some progress simulating a 3D stacked architecture. Modelling power dissipation requires a good model of the processors, in terms of floorplan, thermal properties of materials, power dissipation of individual components, static power dissipation, etc. Developing a realistic model of power dissipation for Paper II required considerable research, even though this type of single-die chips have been in production for some time. Developing realistic models for a 3D architecture would be even more difficult and the project was abandoned.

3.1.9 Memory System Interference - Paper VI & VIII

During our work with CMPs it became apparent that contention for shared resources would cause problems both in terms of performance and latency. Because access to shared resources is traditionally managed on a FCFS basis, programs which access shared resources more often would get a larger share of the resources. As an example, program A can displace data in the cache that is needed by program B, thus reducing program B's performance. In other words, program A *interferes* with program B.

Magnus Jahre began to investigate this property of CMPs and I became interested in this work because prefetching could potentially increase interference, and thus unfairness. Prefetching is typically triggered by a miss in the cache. A program with many misses in the cache, would thus trigger more prefetches than a program with few misses. If the prefetches are not accurate, then the program will consume an even larger part of the shared resources.

Our initial investigation (Paper VI) into fairness was focused on establishing which components of the memory system contributed the most to unfairness. We did this because there is much previous research that focuses on solving the problem at the component level, rather than understanding the nature of interference at a system level [11, 60, 105]. In this work we found that the memory controllers are the main source of interference.

Paper VI used a static off-line method for determining interference. Our next paper (Paper VIII) focused on determining interference at run-time. Determining interference at run-time would enable us to enforce fairness by dividing resources accordingly. Our approach tries to determine the latency of every memory operation that each core would observe if there was no interference (i.e. no sharing of memory resources). Thus, the difference between the actual latency observed in the shared memory system and the estimated latency becomes our estimate of the interference each core observes. Shadow tags are used to estimate the effect of cache sharing and a novel system for estimating the effect of sharing off-chip bandwidth. This system essentially simulates at runtime a private memory controller by storing a virtual private scheduling of memory requests.

Magnus Jahre continues to research this area and is currently working on using this run-time estimation technique in combination with a mechanism to regulate memory accesses to increase fairness in CMPs.

3.1.10 Opportunistic Prefetch Scheduling - Paper IX

The final paper in this thesis started out as an idea for a prefetcher. In my previous work I had seen how important exploiting open pages is for high performance. Traditional prefetchers take as input the miss address streams and produces prefetch addresses as an output regardless of the state of the DRAM system. The original idea behind opportunistic prefetch scheduling was to turn this around. The idea was to take the state of the memory system (i.e. which pages are open at the time) and produces prefetch addresses. The key component of this system was a *Page Vector Table (PVT).* A DRAM page is typically much larger than a single cache line. The PVT is a table indexed by the page address. Each table entry has a bit vector where each bit corresponds to one cache line in that DRAM page. The idea was to track both the access pattern and issued prefetches using the same structure.

However, while researching this idea, we discovered that the method we developed for issuing prefetches could be used in the general case. By decoupling how prefetches are generated and how prefetches are issued we could explore how different prefetching scheduling policies affected known prefetchers. Inspired by the positive feedback we got from Paper IV we started an exploration of the design space combining many well-known prefetchers with different prefetch scheduling policies and bandwidth constraints.

3.2 Research Methodology

3.2.1 Simulators

The research conducted at the Computer Architecture Research Group at NTNU is primarily done by using simulations of systems. Using simulations, rather than developing hardware, allows us to rapidly test our ideas without large investments. However, because a simulation is not real hardware, special care must be taken to ensure the accuracy and applicability of our simulations. The research presented in this thesis has been conducted by using several simulators, each with strengths and weaknesses. Our group has chosen to use publicly available simulators rather than develop our own to conserve effort and reduce verification work.

3.2.1.1 SimpleScalar

SimpleScalar [5, 15] is a cycle-accurate simulator capable of simulating out-of-order superscalar processors. It was developed by Todd Austin during his PhD at the University of Wisconsin in Madison. Today, the simulator is developed and supported by SimpleScalar LLC. It can accurately model a wide range of processors, and give accurate information about cache performance as well as other aspects within the processor.

SimpleScalar has been used extensively in the computer architecture research community. In June 2009, Google Scholar reports that the main papers regarding SimpleScalar has been cited over two and a half thousand times. SimpleScalar has seen extensive testing and verification. As our group focuses primarily on memory systems, it is interesting to examine how the memory system works in SimpleScalar.

The biggest problem with the memory system in SimpleScalar is how it is designed. When the processor core wants to access the memory hierarchy it calls a function (*cache_access*) with the address of the load. This function returns an int representing the latency of the cache operation. If the data is not found in that cache, the *cache_access* function calls the next level cache's *cache_access* function. The problem with this approach is that the function must return a latency, it cannot defer computing the latency until a later time. This in turn makes memory access reordering impractical to simulate. For instance, in FR-FCFS memory scheduling one access might skip ahead in the memory controllers queue if it hits an open page. This would delay memory accesses that have preceded it. This is difficult to simulate in SimpleScalar, because the latency of previous operations have already been computed and used in the simulation.

The DRAM model is also very simple. It uses a fixed latency model with no bandwidth constraints. In addition, the model lacks support for MSHRs, cache bandwidth and interconnect.

Because of SimpleScalar's popularity there has been numerous extensions to the simulator [94]. We have used Wattch [13] for dynamic energy and power estimation and HotLeakage [152] for static energy and power estimation. Our own extensions include a CMP model and a DRAM model, which models contention and open pages, and shadow tag directories.

3.2.1.2 M5Sim

M5 [10] (version 1.1) is now the primary simulator used by our research group. This simulator (version 1.1.) reuses some of the code from SimpleScalar, but extends it with an eventdriven [151] memory model. This is particularly useful for our research group as it allows for the reordering of memory accesses. In addition, it models cache to cache buses and MSHRs. However, DRAM is modelled by using a constant latency and no memory access reordering is performed.

We have extended M5 to include a detailed model of DDR2 memory, multiple types of memory controllers, prefetching[2], crossbar interconnects and interference measurement.

M5 uses an event queue to hold all the events that needs to be serviced by the system. An event is essentially an object with a process() method. The event queue is sorted primarily on the time the event occurs and secondary on the priority of the event. The simulator removes the head event from the queue and calls the process() method on that event. That event might create new events, which are then scheduled by inserting them into the event queue.

Each memory request is modelled as a single object holding the relevant information regarding that specific memory request such as virtual and physical address. The memory hierarchy uses events and a request/response system to move these objects. When a component in the memory hierarchy receives a memory request, it can either compute the latency for that operation directly and schedule a response event or it can put the request into a queue and defer the scheduling of the response event. We use this property extensively in our implementation of memory controllers.

3.2.1.3 CMP$im

CMP$im [64] was the simulator used in the *Data Prefetching Championship (DPC)*. This simulator uses memory traces obtained from execution of regular binaries using the Pin tool [91]. CMP$im was modified by the contest organizers and distributed as a binary with a small *Application Programming Interface (API)* for prefetching. This API consisted of only 6 functions. The most important function was the *IssuePrefetches* function which was called every cycle by the simulator with the current memory hierarchy activity such as hits and misses in the caches. This function was to be written by the contestants. There were two functions for issuing

[2] M5 1.1 supports some prefetching in the default installation, but this implementation was inadequate for our needs.

prefetches into the L1 and L2 respectively and three functions for getting and setting prefetch bits in the cache.

Because of it's simplicity the simulator had a number of shortcomings. DRAM latency was modeled as a constant regardless of page hits or misses. DRAM bandwidth was modelled as a queue which would service one DRAM request every 10 cycles. This queue could hold 1000 entries and it's content could not be inspected by the prefetchers. Many of the contestant thus made their own mechanism for tracking outstanding loads and prefetches, as this is very useful for reducing redundant prefetches. Furthermore, the API did not allow access to the data that was returned by the DRAM. This effectively made pointer-prefetching impossible as the pointers are embedded in this data.

Despite it's weaknesses, the most import property of this simulator is that every contestant used it. This enabled a fair comparison of the prefetching heuristics.

3.2.1.4 CACTI

CACTI [127] is an advanced cache model capable of modeling the timing, power requirements and area of any given cache. It is very useful for understanding the trade-offs between power, area and timing. The model used is very complex and considers most of the available design-techniques. We have used this tool mainly for estimating the latency of caches, but also for power estimation in Paper II.

3.2.2 Benchmarks

3.2.2.1 SPEC2000 and SPEC2006

In the majority of the work presented in this thesis we have used the SPEC2000 benchmark suite [133]. The SPEC2000 benchmark suite consists of 26 programs which are intended to measure the performance of a computer system in a standardized way. The original intent of SPEC2000 is to enable hardware manufacturers, compiler vendors, and operating system vendors to fairly compare products. Thus, the benchmark suite is intended to be run on actual hardware, rather than simulations of hardware. Because it is a standardized set of programs, it has been used extensively in the computer architecture research community.

SPEC2000 has now been superseded by SPEC2006. SPEC2006 was used in Paper V and VI. This is mainly because the organizers of DPC hinted that they would use a subset of SPEC2006 in their evaluation. Because CMP$im uses Pin, it is relatively easy to obtain traces using regular hardware.

3.2.2.2 Multi-Programmed Workloads

Each SPEC2000 benchmark runs as a single thread, and is thus not useful for evaluating the performance of a CMP. To evaluate the performance of CMPs I have randomly generated several multi-programmed workloads. A multi-programmed workload is simply running multiple benchmarks at the same time, one per core.

However, if there are multiple instances of the same SPEC benchmark in a workload, special care must be taken. A naive approach would start each benchmark concurrently, but this leads to problems as the benchmarks would execute nearly in lockstep. Thus the program phases would change at the same instance which would be a very uncommon event in a real system. The easiest solution to avoiding this problem is to either avoid having workloads with multiple instances of benchmarks or start each instance at a separate point in time.

3.2.2.3 Running Experiments

Running the entire benchmark suite to completion would take a considerable amount of time. To avoid this I have used two different strategies.

The first strategy is to use a reduced dataset called *lgred* [76]. This reduced dataset is crafted so that the execution time of the benchmark is reduced by a significant fraction, but so that the original mix of instructions is roughly the same. However, a reduced dataset usually has a smaller working set. Because the working set becomes smaller, the need for large caches and off-chip bandwidth becomes less and the memory system is stressed less. This is unfortunate because it is the performance of the memory system we are focused on improving. Using a reduced dataset reduces the relative importance of the memory subsystem.

The second strategy is fast-forwarding and limited simulation. This strategy simulates only a portion of the original benchmark. This avoids the problem with a small working set, but because the benchmark is not run to completion it is difficult to determine if the smaller sample is representative for the entire benchmark. In particular, the initialization code at the beginning of the benchmarks execution is not representative. Thus, each benchmark is fast-forwarded past the initialization portion of the benchmark. This has the additional benefit of warming up the caches with useful data, such that a more representative memory access behaviour is seen.

However, the problem remains that one cannot be certain that a single datapoint chosen randomly is representative for the entire workload. There exists methods for statistically making such measurements, such as Simpoints [113] or SMARTS [150]. We have not used these methods, because they are not integrated with M5 (version 1.1.), or any of the simulators we have used in our research. Additionally, using these methods for evaluating multiprogrammed workloads have been shown to require considerable amounts of simulation time to be accurate [144].

Fortunately, these methods focus on making the measurements on a benchmark as accurate as possible such that it is possible to give accurate performance numbers for any given technique. In my work I am usually more interested in comparing two techniques to each other and the relative performance difference between them. Thus, the actual performance for a technique for the entire benchmark is not as important as it's relative performance compared to another technique in a smaller segment of the benchmark.

3.2.2.4 Exploring the Solution Space

In computer architecture research there are so many degrees of freedom that it is impractical to simulate the entire solution space. Parallelizing the exploration of the solution space is fortunately trivial. Each combination of parameters and benchmark/workload can be run independently. This makes it possible to explore a larger portion of the solution space using large clusters of machines.

Our research group have been fortunate to be given a generous amount of computing resources from Notur, which is the Norwegian metacenter for computational science. This has enabled us to explore very large solution spaces in a very short amount of time.

Chapter 4

Research Contributions

Doubt grows with knowledge.
– Johann Wolfgang von Goethe

This thesis is a collection of papers that I have authored or coauthored during my time as a PhD student. Each paper is presented in the appendix. Many of these papers had very small figures to conserve space. Rather than including the double-column PDF files, I have opted to reformat each paper to increase readability of the graphs and text. However, I have not altered the text or figures, only the layout and size.

The order of papers presented here are in rough chronological order. In practice, some of the research in these papers have been conducted concurrently.

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

4.1.1 Abstract

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-of-order 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, write-back 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.

4.1.2 Retrospective View

This work looked at an architecture where there is much bandwidth available and the DRAM matrix itself is the biggest contributor to latency. One problem with eDRAM is that it is less dense than pure DRAM technologies. Recently, there has been an increased focus on 3D stacking techniques. This technique allows for the integration of DRAM processes with logic processes. This allows for the combination of high density DRAM and fast logic. Utilizing destructive read in such a context could be interesting.

On the presentation side of this paper, we have used IPC as our metric. The problem with using IPC directly is that many of the more interesting cases becomes quite small in the graphs. Using speedup rather than IPC would have been preferable.

4.1.3 Roles of the Authors

Dybdahl came up with the idea for this paper. On the implementation side, he implemented destructive read DRAM in SimpleScalar, while I made the DRAM bandwidth model. Dybdahl and I had long discussions regarding methodology and how to present our results. He conducted all of the experiments, and wrote the paper, while I reviewed it and made improvements to the manuscript. Natvig worked as an advisor and gave us many valuable comments which improved the overall quality of the paper.

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

4.2.1 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%.

4.2.2 Retrospective View

As Paper II and Paper I were in many ways researched in parallel, most of the comments for Paper I are applicable for this paper as well. On the presentation side, the lack of normalization makes many of the graphs hard to read. This makes comparison of the breakdown of energy components harder.

I did not pursue power simulations in the rest of my work, because power simulations were at that time outside the focus of our research group.

4.2.3 Roles of the Authors

The distribution of work was similar to Paper I. To simulate power and energy, Dybdahl integrated Wattch and Hotleakage. Kjeldsberg helped us develop the model for destructive DRAM energy consumption and power and helped with his expertise on power modelling in general. Natvig worked as an advisor and made many valuable comments which improved the overall quality of the paper.

4.3 Paper III: Hardware Prefetching Using Shadow Tagging

4.3.1 Abstract

This paper presents a novel technique for dynamic selection of parameters for prefetching heuristics based on the use of shadow tag directories. Previous methods have been either static, made for a specific prefetching heuristic, or based on phase detection and tuning. The most flexible of these methods is phase detection and tuning. However, it has a serious drawback as it degrades performance while exploring the parameter space, as each configuration is tested on the running program. Our approach explores the parameter space using an extra structure called a shadow tag directory. This allows us to explore the parameter space without interfering with the running program, such that a larger parameter space can be explored without impacting performance. This paper examines the performance of this technique on tagged sequential prefetching, czone/delta correlation prefetching and reference prediction tables. In addition, we compare our results with a feedback directed approach. We show an overall 24% improvement over the best static sequential prefetcher and an 18% improvement versus feedback directed sequential prefetching on memory intensive SPEC benchmarks.

4.3.2 Retrospective View

Shadow tags are very versatile structures which can be used for many purposes. In this paper we used it to explore possible prefetcher configurations. The presentation in this paper could have been better. In particular, we use IPC directly, instead of using speedup. This makes figure III.2 harder to read and makes the significant improvements on Ammp hard to see. In addition, we use the harmonic mean throughout the paper which makes the relative difference between the techniques less pronounced to the reader. After finishing this paper, Sigmund Visnesbakk implemented prefetching using shadow tags in CMP by using M5 [10] for his master thesis [145].

4.3.3 Roles of the Authors

I did most of the work on this paper, although I must credit Haakon Dybdahl for giving me the idea of examining the use of shadow tags for prefetcher configuration. Lasse worked as an advisor and provided helpful comments and improvements to the many revisions of this paper.

4.4 Paper IV: Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors

4.4.1 Abstract

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM is exploited to increase throughput.

Traditionally, prefetching reduces latency by fetching data before it is needed. In this paper we explore how prefetching can be used to increase memory throughput. We present our own low-cost open-page prefetch scheduler that exploits the 3D structure of DRAM when issuing prefetches. We show that because of the complex structure of modern DRAM, prefetches can be made cheaper than ordinary reads, thus making prefetching beneficial even when prefetcher accuracy is low. As a result, prefetching with good coverage is more important than high accuracy. By exploiting this observation our low-cost open page scheme increases performance and QoS. Furthermore, we explore how prefetches should be scheduled in a state of the art memory controller by examining sequential, scheduled region, CZone/Delta Correlation and reference prediction table prefetchers.

4.4.2 Retrospective View

This paper explores how prefetches should be scheduled in a modern DRAM controller. At the time, this was a largely unexplored area [38]. After I presented this paper at ICCD, Lee et al. presented another approach to the same problem at the International Symposium on Microarchitecture. They noticed that some benchmarks would perform better when prefetches were inserted directly into the read queue. On other benchmarks, performance would increase if the prefetches were inserted into a dedicated prefetch queue. Their approach uses the estimated prefetch accuracy to choose where to insert new prefetches.

This work was continued and resulted in another paper (Paper IX) which further investigates prefetch scheduling.

4.4.3 Roles of the Authors

The initial idea and preliminary investigations were carried out by me.

I proposed the initial design. This design was then refined through extensive discussions with Jahre. I implemented the refined idea in our common simulator framework which leverages code produced by both Jahre and me. Furthermore, I devised the initial experimental methodology and planned which experiments should be carried out. The experiment plan and methodology was then discussed thoroughly with Jahre.

I wrote the first draft of the paper and had the final word in all matters regarding the paper. Jahre read the draft thoroughly and provided significant improvements to the presentation, organization and language. Natvig helped with proof-reading and guidance.

4.5 Paper V: Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables

4.5.1 Abstract

This paper presents a novel prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed techniques, RPT prefetching by Chen and Baer and PC/DC prefetching by Nesbit et al. It combines the storage-efficient table based design of Reference Prediction Tables (RPT) with the high performance delta correlating design of PC/DC. DCPT substantially reduces the complexity of PC/DC prefetching by avoiding expensive pointer chasing in the GHB and recomputation of the delta buffer.

We show that DCPT prefetching can increase performance by up to 3.7X for single benchmarks, while the geometric mean of speedups across all SPEC2006 benchmarks is 42% compared to no prefetching.

4.5.2 Retrospective View

This paper was written for the DPC contest. We received 4^{th} place out of 20 submissions. Because of the competition rules, we had to use the CMP$im simulator which has a number of shortcomings (described in section 3.2.1.3). Competition rules limited the number of pages to four and dictated the use of the geometric mean for aggregating performance numbers.

After the competition I had the opportunity to examine the other contestant's code. During my examination of this code I discovered that the three top contestant used some form of L1 prefetching. Additionally, some of the submissions also had techniques for predicting pointer-chasing code. This lead to paper VII which addresses these shortcomings of the original technique.

4.5.3 Roles of the Authors

The division of labour is similar to Paper IV (See section 4.4.3).

4.6 Paper VI: A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures

4.6.1 Abstract

The potential for destructive interference between running processes is increased as Chip Multiprocessors (CMPs) share more on-chip resources. We believe that understanding the nature of memory system interference is vital to achieve good fairness/complexity/performance trade-offs in CMPs. Our goal in this work is to quantify the latency penalties due to interference in all hardware-controlled, shared units (i.e. the on-chip interconnect, shared cache and memory bus). To achieve this, we simulate a wide variety of realistic CMP architectures. In particular, we vary the number of cores, interconnect topology, shared cache size and off-chip memory bandwidth. We observe that interference in the off-chip memory bus accounts for between 63% and 87% of the total interference impact while the impact of cache capacity interference can be lower than indicated by previous studies (between 5% and 32% of the total impact). In addition, as much as 11% of the total impact can be due to uncontrolled allocation of shared cache Miss Status Holding Registers (MSHRs).

4.6.2 Retrospective View

This paper is very latency oriented. However, it is important to remember that increased memory system latency does not necessarily decrease overall system performance. It would be interesting to study more closely the correlation between memory system interference and system performance.

4.6.3 Roles of the Authors

The initial idea and preliminary investigations were carried out by Jahre.

Jahre proposed the initial design. This design was then refined through extensive discussions with me. Jahre implemented the refined idea in the common simulator framework which leverages code produced by both me and Jahre. Furthermore, Jahre devised the initial experimental methodology and planned which experiments should be carried out. The experiment plan and methodology was then discussed thoroughly with me.

Jahre wrote the first draft of the paper and had the final word in all matters regarding the paper. I read the draft thoroughly and provided significant improvements to the presentation, organization and language. Natvig helped with proof-reading and guidance.

4.7 Paper VII: Multi-Level Hardware Prefetching using Low Complexity Delta Correlating Prediction Tables with Partial Matching

4.7.1 Abstract

This paper presents a low complexity table-based approach to delta correlation prefetching. Our approach uses a table indexed by the load address which stores the latest deltas observed. By storing deltas rather than full miss addresses, considerable space is saved while making pattern matching easier. The delta-history can predict repeating patterns with long periods by using delta correlation. In addition, we propose L1 hoisting which is a technique for moving data from the L2 to the L1 using the same underlying table structure and partial matching which reduces the spatial resolution in the delta stream to expose more patterns.

We evaluate our prefetching technique using the simulator framework used in the Data Prefetching Championship. This allows us to use the original code submitted to the contest to fairly evaluate several alternate prefetching techniques. Our prefetcher technique increases performance by 87% on average (6.6X max) on SPEC2006.

4.7.2 Roles of the Authors

The division of labour is similar to Paper IV (See section 4.4.3).

4.8 Paper VIII: DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems

4.8.1 Abstract

Chip Multi-Processors (CMPs) commonly share hardware-controlled on-chip units that are unaware that memory requests are issued by independent processors. Consequently, the resources a process receives will vary depending on the behavior of the processes it is co-scheduled with. Resource allocation techniques can avoid this problem if they are provided with an accurate interference estimate. Our Dynamic Interference Estimation Framework (DIEF) provides this service by dynamically estimating the latency a process would experience with exclusive access to all hardware-controlled, shared resources. Since the total interference latency is the sum of the interference latency in each shared unit, the system designer can choose estimation techniques to achieve the desired accuracy/complexity trade-off. In this work, we provide high-accuracy estimation techniques for the on-chip interconnect, shared cache and memory bus. This DIEF implementation has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

4.8.2 Roles of the Authors

The division of labour is similar to Paper VI (See section 4.6.3).

4.9 Paper IX: Exploring the Prefetcher/Memory Controller Design Space: An Opportunistic Prefetch Scheduling Strategy

4.9.1 Abstract

Prefetching is a well-known technique for bridging the memory gap. By predicting future memory references the prefetcher can fetch data from main memory and insert it into the cache such that overall performance is increased. Modern memory controllers reorder memory requests to exploit the 3D structure of modern DRAM interfaces. In particular, prioritizing memory requests that use open pages increases throughput significantly.

In this work, we investigate the prefetcher/memory controller design space along three dimensions: prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that selectively prioritizes prefetches to open DRAM pages such that performance regressions are minimized. Opportunistic prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

4.9.2 Roles of the Authors

The division of labour is similar to Paper IV (See section 4.4.3).

Chapter 5

Concluding Remarks

All models are wrong, but some are useful.
– George E. P. Box

5.1 Conclusion

This dissertation has examined several techniques for reducing system memory latency. This has been achieved through multiple approaches, but mainly by using excess bandwidth and scheduling policies.

Destructive read DRAM changes the underlying assumptions about the content in DRAM cells being unchanged after a read. By doing this, the latency of a read is much smaller, but it requires changes to the rest of the memory system so that data is not lost. Because data must not be lost, writeback of cache content must be performed. This increases the amount of bandwidth used. If the system has enough bandwidth to support this change in DRAM semantics, then overall latency is reduced.

The second approach discussed in this thesis is prefetching. Prefetching is a technique for predicting what data is needed in the future and fetching that data into the cache before it is referenced. Prefetching is speculative, thus some of the data that is prefetched will not be used and some bandwidth will be wasted. This thesis presents a technique for generating highly accurate prefetches with good timeliness called DCPT. DCPT uses a table indexed by the load to store the delta history of individual loads. This delta history is then used in Delta Correlation to predict future misses. The next version, DCPT-P, introduces *L1 hoisting* which moves data from the L2 to the L1 to further increase performance, and *partial matching* which reduces the spatial resolution of deltas to expose more patterns.

The interaction with the memory controller is especially important in prefetching.

Utilizing open pages can increase the performance of the system significantly. By exploiting this, prefetching can increase bandwidth utilization and reduce latency at the same time. This affects prefetch scheduling decisions. Scheduling prefetches will often delay a demand read or a writeback. However, if the prefetch is issued to an open page and the prefetch is accurate, then doing so pays off in terms of reduced average latency. Because of this, it can pay off to issue many low accuracy prefetches, rather than issue few highly accurate prefetches.

Finally, this dissertation has examined the impact of having a shared memory system in CMPs. When resources are shared, one core might interfere with another core's execution by delaying memory requests or displacing useful data in the cache. This thesis quantifies this effect and identifies which components are most prone for generating interference between cores. Finally, a system for determining interference at runtime is presented.

5.2 Contributions

In section 1.4 I formulated the main research question for this thesis as:

> *How, and at what cost, can memory system latency be reduced by improving resource utilization?*

This question was further subdivided into four questions. In this section I will review these questions in light of the papers presented in this thesis.

1. How can excess memory bandwidth be utilized to achieve a lower maximum memory latency ?

Paper I and II approaches this question through the use of destructive read DRAM. By changing the semantics of a read to DRAM, the latency of main memory DRAM operations can be drastically reduced, especially in eDRAM. This approach requires additional bandwidth between main memory and cache as every line in the cache is initially dirty and must be written to main memory to conserve data.

2. How can excess memory bandwidth be utilized to achieve a lower average memory latency ?

Paper III introduces a technique for reconfiguring prefetching heuristics depending on the amount of available bandwidth and estimated accuracy. Paper VI and IX examines how prefetches can be scheduled in order to maximize the amount of page hits, thus increasing DRAM throughput and lowering latency. This effect relies on the relative lower cost of a page hit versus a page miss in DRAM. Paper V and VII lowers average memory latency by introducing a new prefetching heuristic called DCPT.

3. How does scheduling decisions in modern highly parallel and complex DRAM interfaces affect the bandwidth/latency trade-off ?

Paper IV and IX examines prefetch scheduling in detail and finds that because of the relative cost difference between page hits and page misses, scheduling prefetches to open pages is beneficial, even with relatively low prefetch accuracy. Depending on the details of the DRAM interfaces, this might make it more beneficial to issue many low accuracy prefetches, rather than few high accuracy prefetches.

4. How does interference in the shared memory system affect Chip Multiprocessor performance ?

Because of the shared memory subsystem in modern CMPs, there is significant probability that the cores will interfere with each other's execution. This effect is documented in Paper VI. This paper found that the most significant contributor to interference is the shared DRAM controllers. Paper VIII expands on this work and introduces a framework for determining interference at runtime. This is done by estimating the latency of a memory request if the memory system was not shared. The difference between this estimate and the actual time it took for the completion of the request is the estimated interference.

5.3 Future Work

Simulating such complex systems leads to many interesting details being left out. With the exception of Paper II, power is not modelled in this thesis. This is due to the difficulty of modelling the power requirements of an unknown component without implementing it in hardware (or hardware description languages). This also applies to area requirements. Where it has been possible we have tried to estimate area based on the memory requirements of the techniques presented as this often becomes the dominant contributor to area.

Furthermore, the simulator leaves out interesting aspects of the system such as virtual memory. In particular it would be interesting to examine how prefetching should handle page faults and *Translation Lookaside Buffer (TLB)* misses. Can page faults be predicted? And if so, is it beneficial to bring in pages from disk to main memory and warm up the TLB with sufficient timeliness?

This thesis presents many contributions to the state of the art. However, many questions remains unanswered. In particular, the impact of prefetching on fairness is unclear. Can prefetching increase fairness, or is it inherently unfair? Furthermore, how do these techniques apply to new 3D stacking techniques?

The interaction between the program, the prefetcher and the memory controller is interesting. Pointer-chasing codes still present a significant problem for prefetching. While scientific code often uses sequential access across arrays to increase computation speeds, commercial applications are often pointer-intensive. Finding good solutions to these problems will give a large performance benefit to this large class of applications.

5.4 Outlook

The techniques presented in this thesis makes some underlying assumptions about the memory hierarchy, which may or may not hold in the future. There are two main assumptions. The first assumption is that lower memory latency increases performance. The second assumption is that there is enough off-chip bandwidth to support the techniques presented in this thesis.

Both of these assumptions are directly linked to what kind of computing environment will be dominant in the future. The recent shift to CMPs has prompted a more throughput oriented view on performance. In a pure throughput oriented system it is acceptable to delay one thread waiting for memory if there is available work in another thread. Prefetching might be useful to increase page hit-rates and thus increase DRAM throughput.

Many applications cannot be made completely parallel and will have a serial portion. Amdahl's law states that increasing the number of cores will only increase the execution speed of the parallel portion of the program, while the execution speed of the serial portion is not affected [3]. As the number of cores increases, the serial portion becomes more dominant. For such programs it is beneficial to use heterogeneous CMPs with one high performance core optimized for single-thread performance and several small cores optimized for throughput for the parallel portion of the program [6, 81, 138]. In such a system, prefetching can reduce the memory latency in the serial portion of the program and increase page hit rates in the parallel portion of the benchmark.

One of the reasons for turning to CMPs is the increased power consumption of modern processor cores. This increase in power consumption reduces battery time for mobile devices and causes thermal problems. Using prefetching increases energy usage as more data is moved between main memory and cache [45]. However, as static power consumption becomes a larger component of the overall power consumption, it is possible to reduce power by reducing overall execution time. Agarwal et al. demonstrated that it is possible to transfer the performance gains of prefetching to an overall reduction in energy consumption [1].

Furthermore, as the number of cores increases, the severeness and probability of interference increases. This can lead to load unbalances and violations of the quality of service requirements. Developing and implementing techniques for fairly sharing memory system resources with acceptable performance is a difficult task. Providing fairness becomes difficult because each core can interact both positively and negatively with every other core in many parts of the system. Novel techniques to address these issues are needed in the future.

Bibliography

[1] D. Agarwal, S. Pamnani, G. Qu, and D. Yeung. Transferring performance gain from software prefetching to energy reduction. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 241–244, May 2004.

[2] J. Alakarhu and J. Niittylahti. A comparison of precharge policies with modern DRAM architectures. In *Electronics, Circuits and Systems, 9th International Conference on*, volume 2, pages 823–826, 2002.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference*, pages 483–485, 1967.

[4] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters*, 6(2):45–48, July-December 2007.

[5] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, 2005.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, 2000.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES: Proceedings of the tenth international symposium on Hardware/Software codesign*, pages 73–78, 2002.

[9] C. Belady. Cooling and power considerations for semiconductors into the next century. In *Low Power Electronics and Design, International Symposium on.*, pages 100–105, 2001.

[10] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[11] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[12] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3D) microarchitecture. *Micro*, 0:469–479, 2006.

[13] D. Brooks, V. Tiwari, and M. 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.

[14] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[15] D. Burger and T. M. Austin. Simplescalar toolset 3.0b, 2003. `http://www.simplescalar.com`.

[16] D. Callahan, K. Kennedy, and A. 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.

[17] C. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *High Performance Computer Architecture, HPCA-10. Proceedings. 10th International Symposium on*, pages 276–287, Feb. 2004.

[18] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[19] L. Chi-Keung and T. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *Computers, IEEE Transactions on*, 48:134–141, Feb. 1999.

[20] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 62–73, 2002.

[21] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, 2002.

[22] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 222–233, 1999.

[23] F. Dahlgren and P. Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(4):385–398, Apr. 1996.

[24] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 1, pages 56–63, Aug. 1993.

[25] B. Davis, T. Mudge, B. Jacob, and V. Cuppu. DDR2 and low latency variants. In *Proc. Memory Wall Workshop at the 26th Ann Int'l Symp. Computer Architecture*, May 2000.

[26] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon. Demystifying 3D ICs: The pros and cons of going vertical. *IEEE Design and Test of Computers*, 22(6):498–510, 2005.

[27] V. Desmet, S. Girbal, and O. Temam. Archexplorer.org: Joint compiler/hardware exploration for fair comparison of architectures. In *INTERACT-13, Workshop on Interaction Between Compilers and Computer Architecture*, 2009.

[28] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings. 29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.

[29] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, 2002.

[30] M. Dimitrov and H. Zhou. Combining local and global history for high performance data prefetching. In *Data Prefetching Championship-1*, 2009. `http://www.jilp.org/dpc/`.

[31] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. *SIGARCH Comput. Archit. News*, 21(2):88–97, 1993.

[32] H. Dybdahl and P. Stenstrom. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 2–12, 2007.

[33] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. In *MEDEA Workshop, PACT*, 2006.

[34] H. Dybdahl, P. Stenström, and L. Natvig. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *SIGARCH Comput. Archit. News*, 35(4):45–52, 2007.

[35] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. *High Performance Computer Architecture, 2009. IEEE 15th International Symposium on*, pages 7–17, 2009.

[36] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[37] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *Micro, IEEE*, 28(3):42–53, May-June 2008.

[38] W. fen Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[39] M. Ferdman, S. Somogyi, and B. Falsafi. Spatial memory streaming with rotated patterns. In *Data Prefetching Championship-1*, 2009. `http://www.jilp.org/dpc/`.

[40] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110, 1992.

[41] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 149–160, 2006.

[42] M. Ghosh and H.-H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked drams. *Microarchitecture, IEEE/ACM International Symposium on*, 0:134–145, 2007.

[43] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[44] M. Grannaes. Bandwidth-aware prefetching in chip multiprocessors. Master's thesis, Norwegian University of Science and Technology, Norway, June 2006.

[45] Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz. Energy characterization of hardware-based data prefetching. *ICCD*, 00:518–523, 2004.

[46] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. In *SIGMETRICS Perform. Eval. Rev.*, pages 31–34, 2004.

[47] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition.* Morgan Kaufmann Publishers, 2007.

[48] M. N. Horenstein. *Microelectronic circuits and devices.* Prentice Hall, 1996.

[49] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 317, Washington, DC, USA, 2003. IEEE Computer Society.

[50] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotspot: a compact thermal modeling methodology for early-stage VLSI design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5):501–513, May 2006.

[51] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49, 2002.

[52] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future cmps. *Parallel Architectures and Compilation Techniques, International Conference on*, 2001.

[53] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, 2006.

[54] I. Hur and C. Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *HPCA '09: Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pages 443–454, 2009.

[55] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2004.

[56] C.-L. Hwang, T. Kirihata, M. Wordernan, J. Fifield, D.Storaska, D. Pontius, G. F. 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.

[57] International Technology Roadmap for Semiconductors. ITRS roadmap, 2007. `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[58] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 39–50, June 2008.

[59] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching prefetch: Optimization friendly method. In *Data Prefetching Championship-1*, 2009. `http://www.jilp.org/dpc/`.

[60] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS '07: Proc. of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, 2007.

[61] S. Iyer and H. Kalter. Embedded DRAM technology: opportunities and challenges. *Spectrum, IEEE*, 36(4):56–64, Apr 1999.

[62] B. Jacob and T. Mudge. Notes on calculating computer performance. Technical Report 231-95, University of Michigan, March 1995.

[63] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *IEEE Trans. Comput.*, 45(10):1180–1194, 1996.

[64] A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS*, 2008.

[65] *DDR2 SDRAM Specification.* JEDEC Solid State Technology Association, May 2006.

[66] N. Jerger, E. Hill, and M. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 177– 188, March 2006.

[67] JILP. DPC Webpage. `http://www.jilp.org/dpc/online/DPC-1Program.htm`, 2009.

[68] L. K. John. More on finding a single number to indicate overall performance of a benchmark suite. *SIGARCH Comput. Archit. News*, 32(1):3–8, 2004.

[69] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu. Run-time spatial locality detection and optimization. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 57–64, 1997.

[70] D. Joseph. Prefetching using markov predictors. In *The 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[71] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373, May 1990.

[72] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49 (4/5):589–604, 2005.

[73] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *High-Performance Computer Architecture. HPCA-6. Proceedings. Sixth International Symposium on*, pages 206–217, 2000.

[74] K. Kaspersky. *Code Optimization: Effective Memory Usage.* A-List Publishing, 2003.

[75] S. Kaxiras and J. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 161–170, Jan 1999.

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

[77] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, March-April 2005.

[78] D. M. Koppelman. Neighborhood prefetching on multiprocessors using instruction history. *Parallel Architectures and Compilation Techniques, International Conference on*, 2000.

[79] K. Krewell. Cell moves into the limelight. *Microprocessor Report*, Feb. 2005.

[80] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201, 1998.

[81] R. Kumar, D. M. Tullsen, and N. 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, 2006.

[82] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 144–154, 2001.

[83] S.-C. Lai and S.-L. Lu. Hardware-based pointer data prefetcher. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 290 – 298, Oct. 2003.

[84] A. J. Lande. Evaluering av chip multiprosessor simulatorer (in norwegian). Master's thesis, Norwegian University of Science and Technology, Norway, June 2006.

[85] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008.

[86] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 11–21, 2000.

[87] G. M. Link and N. Vijaykrishnan. Thermal trends in emerging technologies. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 625–632, 2006.

[88] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[89] G. L. Loi, B. Agrawal, N. Srivastava, S.-C. Lin, T. Sherwood, and K. Banerjee. A thermally-aware performance analysis of vertically integrated (3-d) processor-memory hierarchy. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 991–996, 2006.

[90] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. *SIGPLAN Not.*, 31(9):222–233, 1996.

[91] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[92] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[93] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[94] N. Manjikian. More enhancements of the simplescalar tool set. *SIGARCH Comput. Archit. News*, 29(4):5–12, 2001.

[95] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[96] R. E. Matick and S. E. Schuster. Logic-based eDRAM: origins and rationale for use. *IBM J. Res. Dev.*, 49(1):145–165, 2005.

[97] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, 2002.

[98] T. Mitra. Dynamic random access memory: A survey. *Research Proficiency Examination Report*, march 1999.

[99] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965.

[100] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. *SIGARCH Comput. Archit. News*, 36(3):63–74, 2008.

[101] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[102] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[103] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *38th Annual International Symposium on Microarchitecture (MICRO-38)*, pages 233–244, 2005.

[104] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 80–87, 2004.

[105] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[106] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Micro, IEEE*, 25:90–97, Jan. 2005.

[107] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[108] K. J. Nesbit, N. Aggarwal, J. L., and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Proc. of the 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[109] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[110] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 24–33, Apr 1994.

[111] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[112] D. A. Patterson. Computer science education in the 21st century. *Commun. ACM*, 49(3):27–30, 2006.

[113] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*, June 2003.

[114] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, 2004.

[115] K. Puttaswamy and G. H. Loh. Implementing caches in a 3D technology for high performance processors. *ICCD*, pages 525–532, 2005.

[116] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, 2006.

[117] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[118] L. M. Ramos, J. L. Briz, P. E. Ibáñez, and V. Viñals. Multi-level adaptive prefetching based on performance gradient tracking. In *Data Prefetching Championship-1*, 2009. `http://www.jilp.org/dpc/`.

[119] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, 2000.

[120] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 3(4):34–43, 1995.

[121] A. Roth and S. Gurindar S. Effective jump-pointer prefetching for linked data structure. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 111–121, 1999.

[122] S. Sandeep. Gcc-inline-assembly-howto, March 2003. `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[123] J. Shao and B. Davis. A burst scheduling access reordering mechanism. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 285–294, 2007.

[124] J. Shao and B. T. Davis. The bit-reversal SDRAM address mapping. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 62–71, 2005.

[125] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, 2000.

[126] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings. 30th Annual International Symposium on Computer Architecture*, pages 336– 347, 2003.

[127] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2, Compaq Western Research Laboratory, August 2001.

[128] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[129] J. E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10), October 1988.

[130] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[131] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 252–263, 2006.

[132] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News*, 37(3):69–80, 2009.

[133] SPEC. Spec 2000 benchmark suites, 2000. `http://www.spec.org`.

[134] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[135] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

[136] V. Srinivasan, E. Davidson, and G. Tyson. A prefetch taxonomy. *Computers, IEEE Transactions on*, 53:126–140, Feb. 2004.

[137] G. C. Stierhoff and A. G. Davis. A history of the IBM systems journal. *IEEE Annals of the History of Computing*, 20(1):29–35, 1998.

[138] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 253–264, 2009.

[139] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 51–62, 2008.

[140] J. Torrellas, H. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[141] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, 1995.

[142] S. Vander Wiel and D. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, Jul 1997.

[143] S. VanderWiel. A survey of data prefetching techniques. Technical Report 5, University of Minnesota, October 1996.

[144] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Fame: Fairly measuring multithreaded architectures. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 305–316, 2007.

[145] S. Vinsnesbakk. Implementation and testing of shadow tags in the m5 simulator. Master's thesis, Norwegian University of Science and Technology, Norway, June 2008.

[146] Wikipedia. CAS Latency, 2009. `http://en.wikipedia.org/wiki/Cas_latency`, Retrieved 18.Jul 2009.

[147] Wikipedia. List of intel microprocessors, 2009. `http://en.wikipedia.org/wiki/List_of_Intel_microprocessors`, Retrieved 18.Jul 2009.

[148] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Computing Frontiers*, 2006.

[149] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture New*, 23(1), march 1995.

[150] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and comparing prevailing simulation techniques. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 266–277, Feb. 2005.

[151] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation, 2nd ed.* 2000.

[152] 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, Mar. 2003.

[153] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 213–224, 2005.

[154] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. *Eighth International Symposium on High-Performance Computer Architecture, 2002*, pages 107–116, 2002.

Papers

I Haakon Dybdahl, Marius Grannæs and Lasse Natvig, "Cache Write-Back Schemes for Embedded Destructive-Read DRAM", *Architecture of Computing Systems (ARCS)*, 2006

II Haakon Dybdahl, Per Gunnar Kjeldsberg, Marius Grannæs and Lasse Natvig, "Destructive-Read in Embedded DRAM, Impact on Power Consumption", *Journal of Embedded Computing*, Vol 2, Issue 2, 2006

III Marius Grannæs and Lasse Natvig, "Hardware Prefetching Using Shadow Tagging", *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008

IV Marius Grannæs, Magnus Jahre and Lasse Natvig, "Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors", *XXVI IEEE International Conference on Computer Design (ICCD)*, 2008

V Marius Grannæs, Magnus Jahre and Lasse Natvig, "Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables", *Data Prefetching Championship - 1*, 2009

VI Magnus Jahre, Marius Grannæs and Lasse Natvig, "A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures", *11th IEEE International Conference on High Performance Computing and Communications*, 2009

VII Marius Grannæs, Magnus Jahre and Lasse Natvig, "Multi-Level Hardware Prefetching using Low Complexity Delta Correlating Prediction Tables with Partial Matching", *Accepted to the 5th HiPEAC Conference*, 2010, **Nominated for best paper award**

VIII Magnus Jahre, Marius Grannaes and Lasse Natvig, "DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems", *Accepted to the 5th HiPEAC Conference*, 2010

IX Marius Grannæs, Magnus Jahre and Lasse Natvig, "Exploring the Prefetcher / Memory Controller Design Space: An Opportunistic Prefetch Scheduling

Strategy", *Preprint submitted to Journal of Instruction Level Parallelism, January 2010*

Paper I

Cache Write-Back Schemes for Embedded Destructive-Read DRAM

Haakon Dybdahl, Marius Grannæs and Lasse Natvig

In *Architecture of Computing Systems (ARCS)*, 2006

Abstract

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-of-order 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, write-back 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.

I.1 Introduction

The pipeline of a processor is now running at a higher frequency than main memory. Caches are used to reduce the number of memory accesses that request data from main memory and hence reduce the effect of the slow main memory. However, caches have several disadvantages, they require substantial chip area, increase power consumption and do not work equally well for all applications. Other mechanisms are out-of-order execution, prefetching and thread switching. These techniques increase the complexity of the processor, power consumption and chip area as well. Increasing the chip area increases the cost of manufacturing the chip as the yield and number of chips per wafer are reduced. Increasing the power consumption increases the cost of packaging the chip due to increased cooling requirements and reduces the operation time when powered by batteries.

By reducing the latency of the main memory itself the processor core and cache system can be simplified without degrading performance. The latency of main memory can be decomposed into different parts: Cache miss latency, latency of bus to main memory and DRAM latency. The latency of the off-chip bus to main memory can be eliminated by integrating the DRAM and processor on the same chip. Several projects have researched into merging processors and memory over a long period of time ([2, 4, 6, 10, 11, 14–16, 18, 19]). There have been many obstacles in producing chips which have both dense main memory and fast logic. However, embedded DRAM has become more common during the last years, 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[7] and Nintendo's GameCube.

Embedding DRAM does not reduce the latency of the DRAM bank itself, and even with the off-chip bus latency eliminated the DRAM is much slower than the pipeline of the processor.

Reading a memory cell with DRAM technology clears the content of the cell. The data is stored as a charge in a capacitor and this charge is used by the sense amplifier in the DRAM bank to enter a logic state depending on the content. Data has to be written back, and this causes part of the long DRAM latency. The sense amplifier writes back the data to the cell before the data is sent out of the chip. A new type of embedded DRAM has been prototyped by Hwang et al. [8] which omits this write-back of the data to the memory cell. All computer programs assume that a read to the memory is not destructive so there should be a way of conserving the content of the memory. In the pioneer work by Hwang et al. [8] they used a write-back buffer which was made of SRAM technology. This guaranteed that write-backs are done without disturbing fast read accesses to the memory. The prototype required a write-back buffer with a storage capacity of 25% of the DRAM banks. This size was needed to ensure adequate buffering and total removal of congestion. In a computer with caches, congestions only causes the processor to stall and causes a degradation of performance. There is no need to remove all congestion, but there should be as few as possible.

We have earlier studied the effect on power consumption by using destructive-read DRAM [3]. The findings were only a slightly increase in system power consumption (0.5% and 3% for 16kbyte and 2kbyte caches respectively). This paper studies writing back data without using large write-backs buffers. The baseline architecture is a small processor with small caches and embedded DRAM. This represents an embedded system. We compare performance by utilizing the modification proposed by Hwang et. al and compare the performance in terms of instructions per clock cycle (IPC). We propose two new schemes for write-backs based on the existing cache in a processor. The cache architecture is modified in different ways so data are conserved.

Our findings are that the cache can be much smaller without degrading instructions per clock cycle (IPC) compared to conventional main memory. The large write-back buffer in the prototype can be omitted by using the cache for this purpose.

The next section describes the concept of destructive-read DRAM. Section I.3 presents two new schemes for conserving data based on cache write-backs. Experimental methodology and results are covered in Section I.4 and I.5. Related work are found in Section I.7, discussion in Section I.6 and we conclude in Section I.8.

I.2 Embedded Destructive-Read DRAM

I.2.1 Embedded Memory

The two dominating technologies for storing data inside a chip are static memory (SRAM) technology and dynamic memory (DRAM) technology. Chips based on DRAM memory are cheaper, denser and consume less power than memory chips based on other technologies with the same storage capacity; therefore it is the main choice for main memory. However, 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 in logic gates, whereas in DRAM the capacitors have to be read and decoded into logic signal voltages and data has to be written back to the cell. SRAM is made of logic gates and integrates well with processor cores. SRAM is therefore more suited as technology for on-chip caches.

DRAM chips are highly optimized for storing data and a large amount of the design is analog. DRAM uses capacitors to store data (see Figure I.1) which fills a large area of the chip. Logic circuits on the other hand are optimized for speed and power distribution.

Merging these technologies results in compromises. In a *processor in memory* solution the processor is slow while in a *memory in processor* the memory is less dense. However, much effort has been put into reducing compromises when merging memory and logic, and this is claimed to be easier to do with future technology

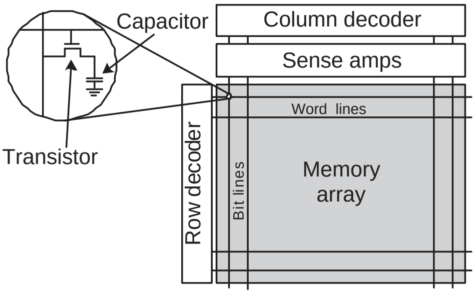


Figure I.1: A logical sketch of a DRAM macro.

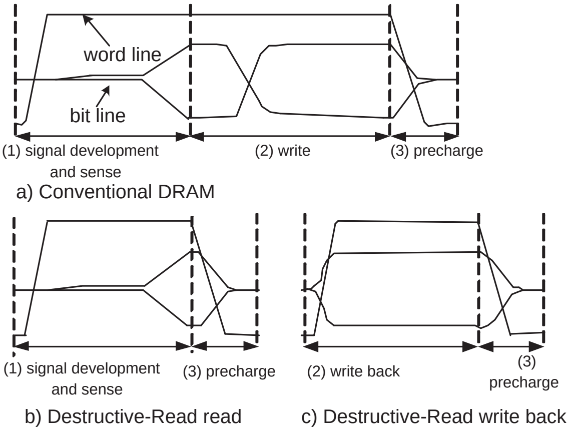


Figure I.2: Conceptual waveform diagrams of conventional DRAM architecture vs. destructive-read [8].

[5]. Embedded DRAM is still not as dense (bits per area) as pure DRAM chips (typically 50% less bits per area).

The latency of a DRAM bank is a function of the size of the bank. Reducing the size of the memory array reduces the length and capacity of the lines and thus latency. Each memory bank has extra circuitry such as sense amplifiers, and reducing the size of the memory array increases this overhead and hence lowers density. Reducing the size of the memory bank reduces dynamic power consumption since a smaller bank is activated, but the static leakage from the added transistors will limit the optimal minimum size.

I.2.2 Destructive-Read DRAM

Destructive-read DRAM [8] is a modified version of conventional DRAM. A memory bank with conventional DRAM is shown in Figure I.1. The row decoder is the first

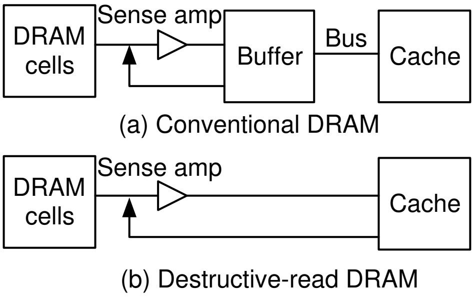


Figure I.3: Conceptual view of DRAM.

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 I.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 I.3a the locking works as a buffer. From this buffer the data is 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 I.2a) is split into two cycles (see Figure I.2b and c) Destructive-read DRAM does not lock the data after reading (as shown in Figure I.3b). Instead the data are sent directly out of the cell, in this case to 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 I.2c. However, write-back can be done later in contrast to conventional DRAM where read and write-back are one single operation.

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. The WBB could write to several banks simultaneously and required significant chip area. Later a new scheme was made where the WBB was replaced with destructive-read DRAM[9]. Both designs guaranteed that write-backs never conflicted with read operations.

I.3 New Write-back Schemes

The design by Hwang et. al included a large write-back buffer and in this work we utilize the cache of the system to do this task so the write-back buffer can be removed. However, it is not obvious when data should be written back from the cache and how this will impact performance since the cache is much smaller than the original write-back buffer. As shown in the evaluation section, simulations show that these schemes work well.

We call the first scheme 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 we call this the *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 not being changed. There is only one access on the memory bus. For destructive-read with immediate write-back scheme, the data is first transferred 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. An important factor is that the DRAM is embedded. Insertion of extra latches for each DRAM bank will require substantial chip area. 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 will therefore 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 executed 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 I.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. 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 I.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

Cache content before instruction is executed

	Immediate	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Line 0	Line 1
1	ADR[0]→R1	L0	L0	L0																	x	y
2	ADR[1]→R2				S0	S0	S0	L1	L1	L1											0	y
3	R1+R2→ADR[0]										S1										0	1
4	ADR[2]→R2											S1	S1	S0	S0	S0	L2	L2	L2		0 *	1

	Delayed	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Line 0	Line 1
5	ADR[0]→R1	Sx	Sx	Sx	L0	L0	L0														x	y
6	ADR[1]→R2							Sy	Sy	Sy	L1	L1	L1								0	y
7	R1+R2→ADR[0]																				0	1
8	ADR[2]→R2														S0	S0	S0	L2	L2	L2	0	1

Figure I.4: Example of execution with the two different write-back schemes. DRAM bus activity is shown with LA for reading and SA for writing address A. 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.

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.

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

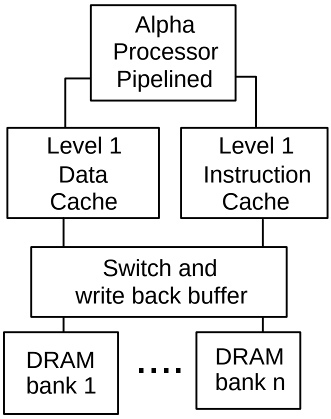


Figure I.5: The simulated computer.

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.

One important question is 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 depends on the program as well.

I.4 Methodology

The purpose of our simulation is to study the performance of different write-back schemes with destructive-read DRAM and compare this to conventional embedded DRAM and to a theoretical write-back scheme with free write-backs for data conversation. The simulator is based on SimpleScalar version 3 [1]. It is extended to simulate a configurable number of DRAM 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 I.5. The target is a computer with embedded memory and one level of 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. Both are one kbyte, two way set associative caches with 64 bytes cache lines. 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)[12] 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*). Sample tests show 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 (i.e. 6 ns), 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. Access time is 3 ns.
- *Delay* is the delayed write-back scheme. The cache behaves like a normal cache, but the lines are always written back on replacement. Access time is 3 ns.
- *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

```
/* 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;
}
```

Figure I.6: Source code for the initial experiment.

performance degradation imposed by the extra write-backs for conserving data. Access time is 3 ns.

- *Hwang* represents the original scheme from Hwang. Access time is 3 ns. The accesses are guaranteed to be congestion free. No fast cache is included.

I.5 Evaluation

I.5.1 Initial experiment

An initial experiment was run to verify the predictions regarding performance (see Figure I.4) 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 I.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 memory latency becomes dominant. This configuration does not reflect a real system, 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 line 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 I.7 and are according to predictions.

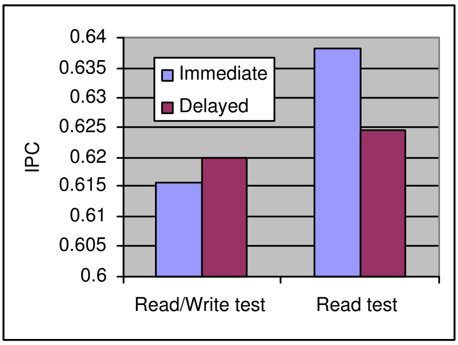


Figure I.7: Results from the initial experiment.

I.5.2 IPC for Different Write-back Schemes

Simulation of the different write-back schemes for the baseline architecture is shown in Figure I.8. Average values are shown to the right. First of all we see that all applications benefit from destructive-read DRAM except for *art* where the Hwang scheme degrades performance. Secondly we see that the schemes with free write-backs for data conversation, *no cost*, does not perform much better than the *delayed* and *immediate* schemes. This indicates that the write-backs are well hidden in both the *delayed* and *immediate* schemes. Even though Hwang scheme perform well for some applications such as *mgrid* and *crafty* due to less congestion, the overall performance is lower than the other scheme. This is because there is no fast cache in this scheme.

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 differences in performance of the applications are related to memory access patterns and locality of the applications. Applications with poor locality that are memory bound such as *ammp*, *mcf* and *crafty* benefit the most from destructive-read DRAM. Applications with good locality that are CPU bound such as *lucas* and *sixtrack* have less but significant advantage of destructive-read DRAM. The buffer size of Hwang's original scheme was 25% of the DRAM size. For the simulated applications the size of the buffer 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 1% of the performance is lost due to extra write-backs for both immediate and delayed write-back schemes.

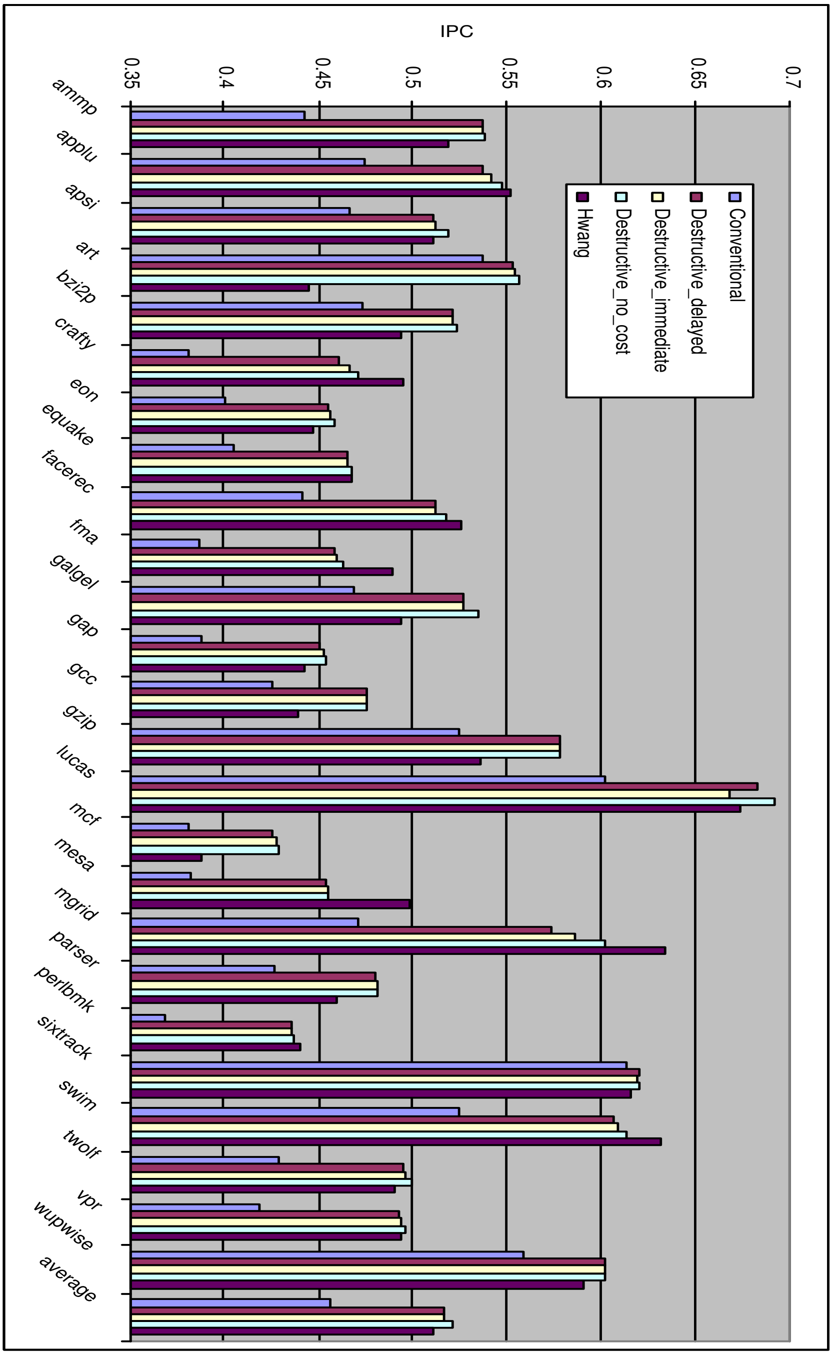


Figure I.8: Performance of baseline configuration for different SPEC2000 benchmarks.

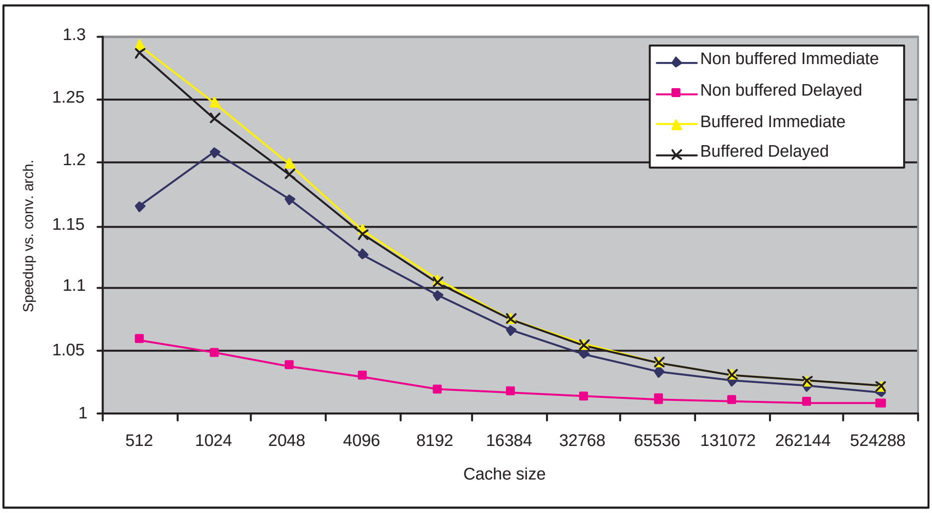


Figure I.9: Speedup in terms of increased IPC.

I.5.3 Cache size

The IPC for different cache sizes are shown in Figure I.9. Two different configurations are simulated, with a small write-back buffer in addition to the cache, and without this write-back buffer. We see that this small write-back buffer (one entry) has a big impact on the performance for the delayed write-back scheme. The buffer changes the access pattern for a memory access; in cases with congestion the delayed write-back is performed after the read access. For the immediate write-back schemes the buffer has less impact, but is still significant.

Smaller caches increase the *miss rate* and the number of accesses to the memory system. Larger caches increase the *hit rate* until a point where compulsory misses start to dominate. The immediate write-back scheme is slightly better than the 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 I.10. In the delayed write-back scheme, data are not written back immediately. For modified data (by the 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 being replaced for data caches. 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, performance is similar except for small caches where the immediate write-back scheme is better.

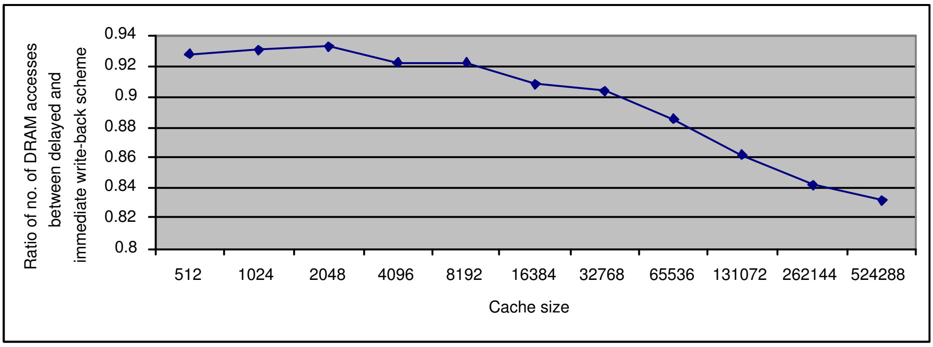


Figure I.10: Comparison of the total number of accesses to DRAM from caches for the two different write-back schemes. For larger caches more data are modified before they are written back to DRAM, and this gives the delayed write-back scheme an advantage.

I.5.4 Latency and number of DRAM banks

Simulation of different DRAM-latencies is shown in Figure I.11. As latency increases, performance degrades as the CPU is stalled. Even though the IPC is degraded by increasing the latency of the memory, the speedup of using destructive-read memory increases with increasing cache size. The delayed and immediate schemes degrade faster than the no cost scheme due to conflicts between reading and writing. Increasing the number of memory banks reduces the probability of congestion. The effect of increasing (or decreasing) the number of DRAM banks is shown in Figure I.12a. In this configuration each bank is independent and can handle one memory access each simultaneously. In addition to increasing the maximum number of parallel accesses, increasing bank count decreases the probability that two accesses are to the same memory bank as data are spread out to more banks.

I.5.5 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 I.12b. 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. A write-back buffer with only one entry for each memory channel (total 256 bytes) is adequate for the simulated configuration.

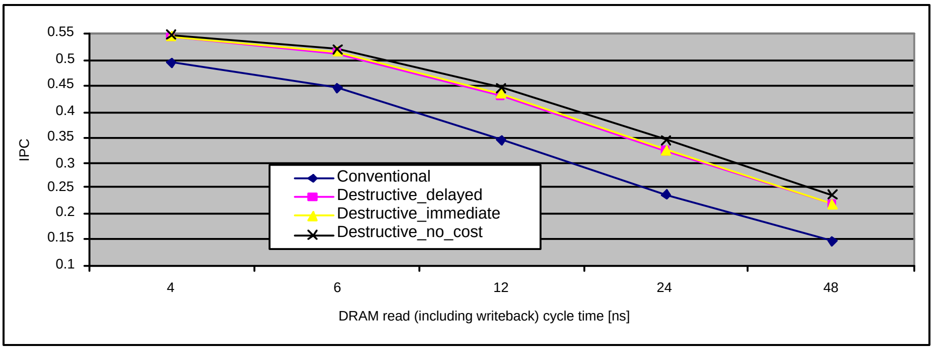


Figure I.11: Average IPC as a function of latency. Write-backs become blocking as latency is increased. The values on the x-axis are non-linear, the first value is increased from 3 to 4 ns to match the cycle time of the CPU.

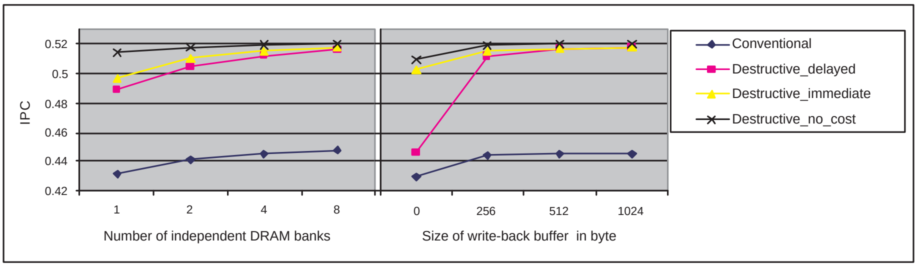


Figure I.12: (a) To the left, average IPC as a function of the number of DRAM banks. For a small number of DRAM banks, write-backs blocks performance. (b) To the right, average IPC as function of number of buffer size. By turning off the write-back buffer, the delayed write-back model has to wait for 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.

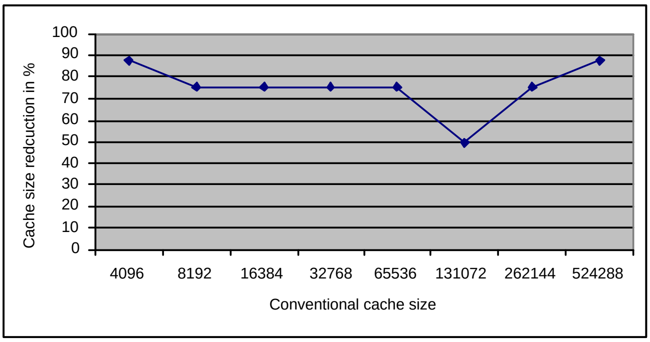


Figure I.13: Cache size saving in % by using destructive-read DRAM compared to conventional DRAM for equal or better performance. This figure is based on Figure I.9.

I.6 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 bus, 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 smaller 1kbyte cache.

The simulations show that the process of writing back data is hidden quite well (about 1% 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 I.13. Even though the baseline for the simulation has small caches, we have found that the cache size can be reduced with median 75% for cache size in the range 4kbytes to 512 kbytes when applying destructive-read DRAM without degrading performance. 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).

We have used a constant latency for the caches in our simulation. In real caches the latency of the cache is a function of the size of the cache. Smaller caches are faster than larger caches. A more accurate model would be to reduce the latency of the cache for smaller caches. This would be an advantage for the destructive-read DRAM schemes, and by using a constant cache access time we introduce an error that is a disadvantage for our schemes.

We have simulated a factor two difference in latency for destructive-read compared to conventional read from DRAM. This was based on the number from a prototype. However, we have shown that the write-backs are hidden very well. Less than 1% of the performance is lost due to congstion for the baseline scheme. Therefore, even a small decrease in latency for destructive-read DRAM will increase IPC.

I.7 Related Work

Most of projects that have researched into merging processors and memory have not looked 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 architectures to utilize the extra bandwidth: the IRAM project [17], Yukon [11], Terasys [6] and Execube [13]. The Mitsubishi M32R/D [15] 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 ([2, 14]).

I.8 Conclusion

In order to reduce cache sizes we have looked into increasing the speed of the DRAM bank in combination with enhancing the tasks of the cache. In our schemes the caches are responsible for conserving data read from DRAM memories. We have shown that this does not infer any bottlenecks and that the cache size can be dramatically reduced by 75% compared to a conventional architecture without degrading performance in terms of IPC. The large write-back buffer used in the prototype by Hwang et. al can be eliminated without significant performance degradation. The possible reduction in cache size reduces both dynamic and static power consumption as well as system size. The 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, Feb. 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. Kjeldsberg, M. Grannæs, and L. Natvig. Destructive-read in embedded DRAM, impact on power consumption. *Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures, Issue 2, 2006*, 2006.

[4] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In CICC, Boston, MA, pages:30.6.1-30.6.4, 1992.

[5] T. Furuyama. Trends and challenges of large scale embedded memories. Custom Integrated Circuits Conference, 2004. Proceedings of the IEEE 2004, Page(s):449 - 456, 2004.

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

[7] L. Gwennap. Embedded DRAM use rises. Nikkei Electronics Asia, June, June 2003.

[8] C.-L. Hwang, T. Kirihata, and M. W. 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.

[9] B. 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.

[10] 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.

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

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

[13] P. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and logic chip for massively parallel systems. IEEE, Adv. Research in VLSI,16th Conf. on, 1995.

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

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

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

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

[18] 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. ISBN 0-89791-786-3. doi: http://doi.acm.org/10.1145/232973.232984.

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

Paper II

Destructive-Read in Embedded DRAM, Impact on Power Consumption

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

In *Journal of Embedded Computing*, Vol 2, Issue 2, 2006

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*

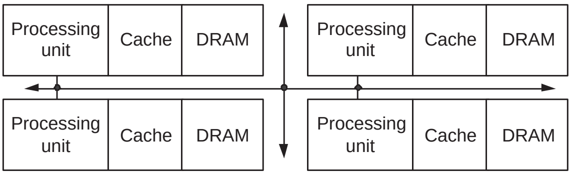


Figure II.1: Multiple cores and memory in a single chip.

II.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.

The target architecture is shown in Figure II.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 II.2. The models used for power estimation is described in Section II.3. Section II.4 describes the simulations that are run and Section II.5 describes the results. Section II.6 is discussion and Section II.7 conclusions followed by references.

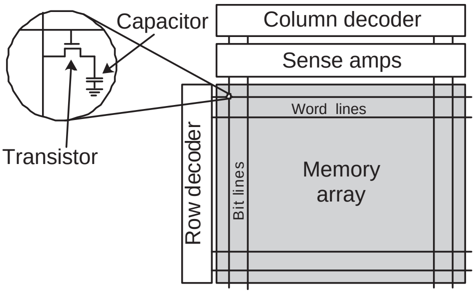


Figure II.2: Inside a DRAM bank.

II.2 Embedded Destructive-Read DRAM

II.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 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 II.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].

II.2.2 Related Work

Several projects have done research into merging processors and memory ([4, 6, 7, 15, 17, 20, 22, 23, 28, 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

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 II.1: Random cycle time for various memories [11].

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 NP-1c[8] network processor and Nintendo's GameCube. Embedded DRAM is becoming commercially available at different speeds. Table II.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].

II.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 II.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 II.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

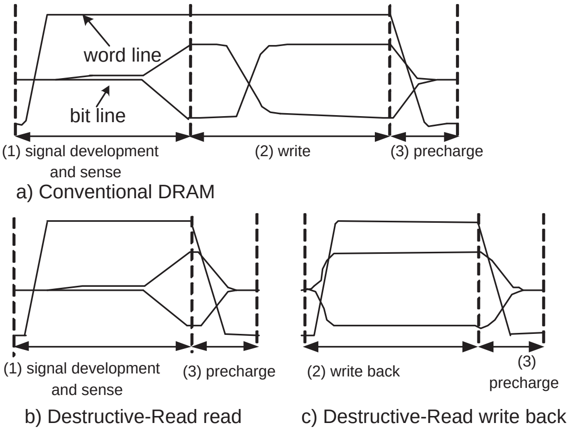


Figure II.3: Conceptual waveform diagrams of conventional DRAM architecture vs. destructive-read [10].

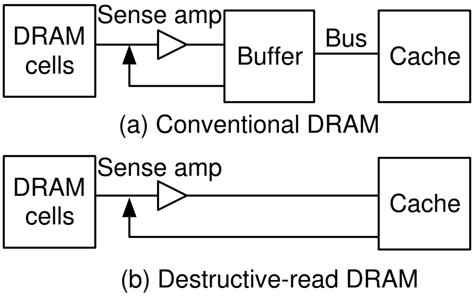


Figure II.4: Conceptual view of DRAM.

leaving the capacitor discharged. In the second phase the logic state is locked. In Figure II.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 II.3a) is split into two cycles (see Figure II.3b and c). The destructive-read DRAM does not lock the data after reading (as shown in Figure II.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 II.3c. However, write-back is not performed immediately after the read, this in contrast to conventional DRAM where read and write-back are a single operation.

II.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 II.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.

II.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 Equation II.1.

$$P_{dynamic} = C_{switched} * V_{dd}^2 * f_{clk} \tag{II.1}$$

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.

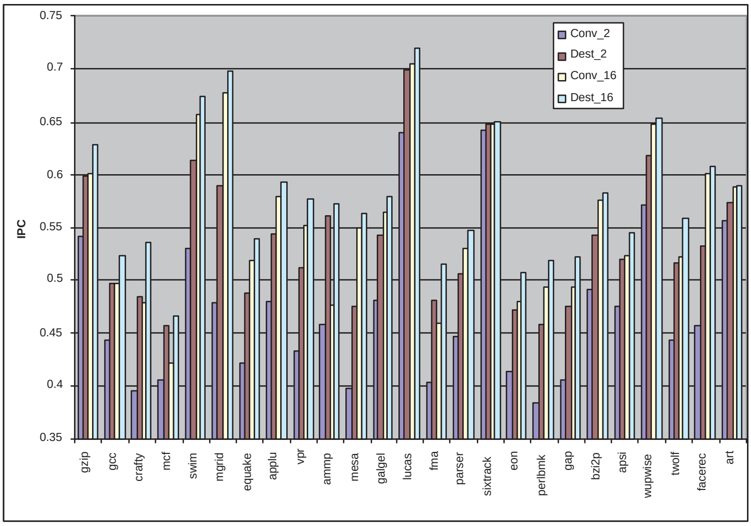


Figure II.5: IPC for the applications in the SPEC2000 benchmark suite for conventional (conv) and destructive-read (dest) DRAM with 2 and 16 kbytes cache configuration. The average IPC is 13.7% higher for *dest_2* compared to *conv_2*, and 4.5% higher for *dest_16* compared to *conv_16*.

II.3.0.1 Voltage and Frequency

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

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

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.

II.3.1 Power model of DRAM with bus

The data flow of accessing DRAM from cache is shown in Figure II.6. The address and control signals are sent to the DRAM bank. Data are read out of the bank

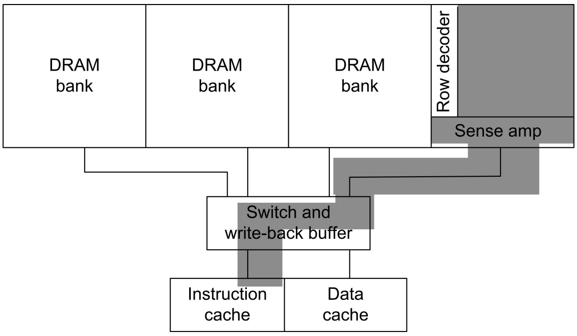


Figure II.6: Data communication when accessing a memory bank.

and written back along the route shown as a thick gray line. Equation II.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 II.5.

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

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

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

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 $E_{SWITCH} = 1nJ$.

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.

II.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 II.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

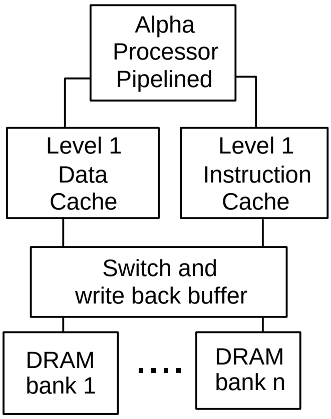

Alpha Processor Pipelined
Level 1 Data Cache
Level 1 Instruction Cache
Switch and write back buffer
DRAM bank 1
....
DRAM bank n

Figure II.7: The simulated single chip computer.

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).

II.5 Results

The energy consumption for the various SPEC2000 benchmarks is shown in Figure II.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 II.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 II.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

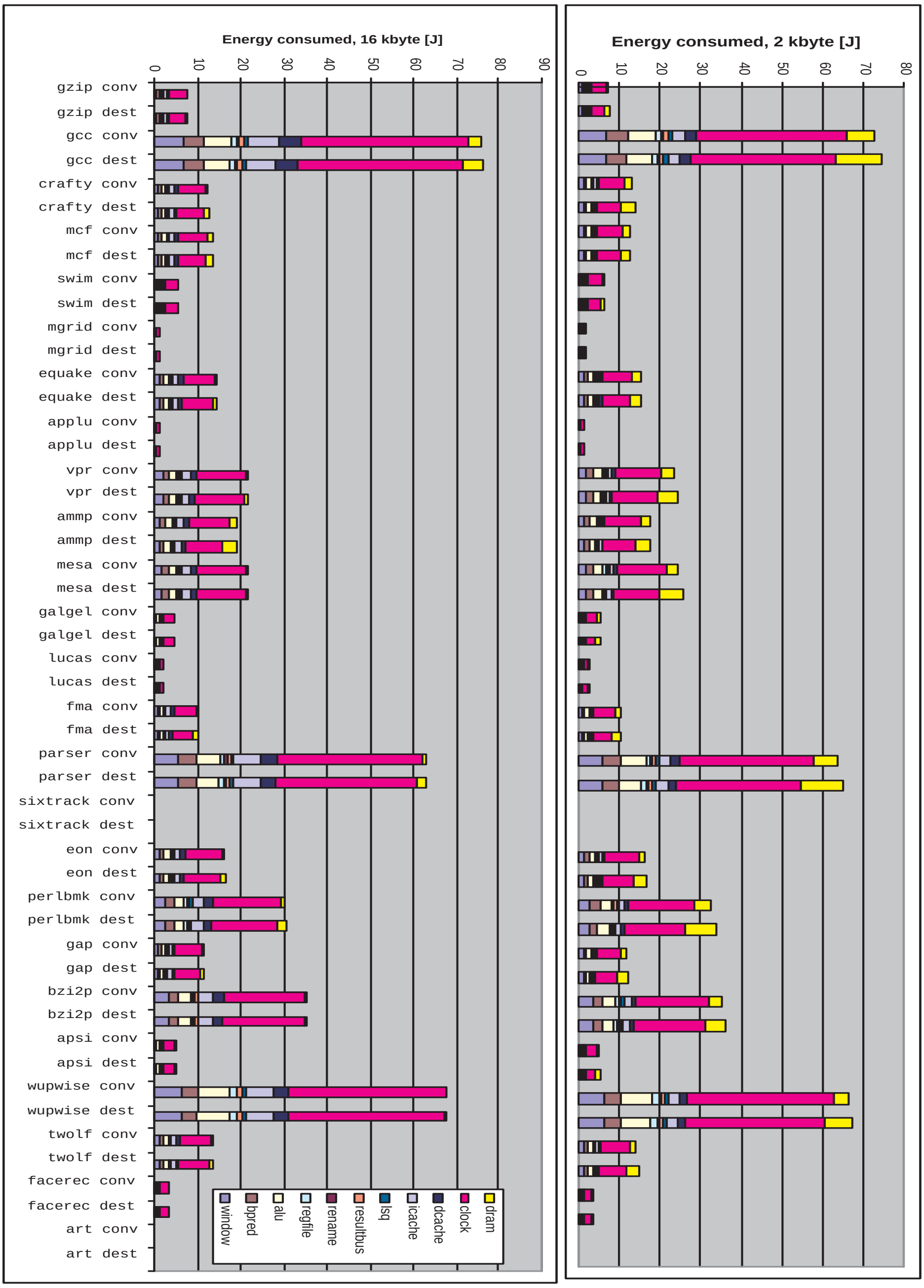


Figure II.8: Energy consumption for various applications in the SPEC2000 suite for conventional DRAM (conv) and destructive-read DRAM (dest). Total cache size is 16 kbytes to the left and 2 kbytes to the right. Average increase in power consumption from conventional to destructive are 0.5% and 3% for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

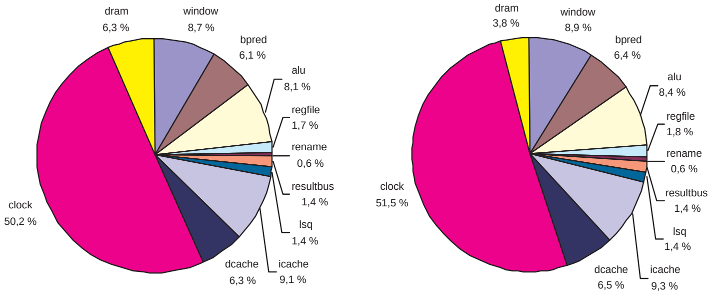


Figure II.9: Energy consumption for *gcc* for destructive-read DRAM to the left and conventional DRAM to the right. Cache size is 16 kbytes.

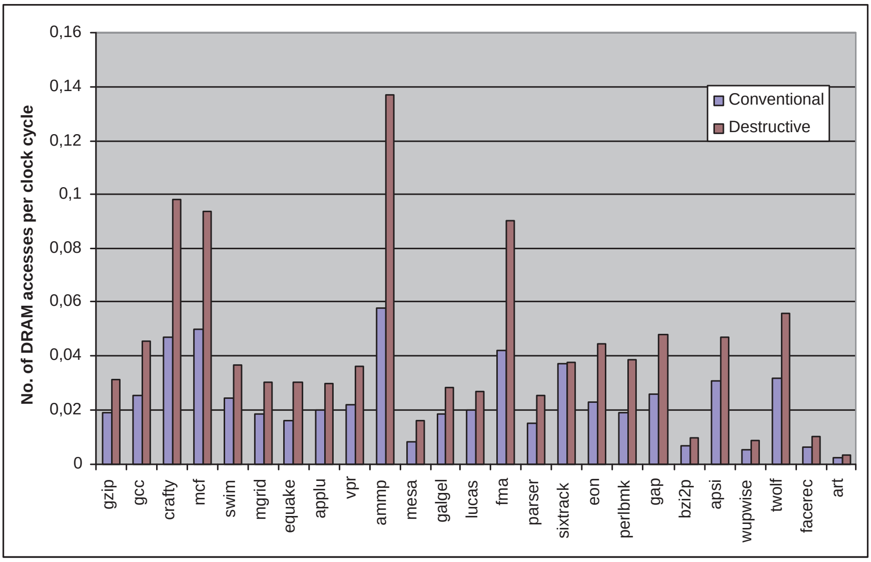


Figure II.10: Number of DRAM accesses per clock cycle for 16 kbytes cache. Due to increased IPC the number of DRAM accesses can be more than doubled for each clock cycle.

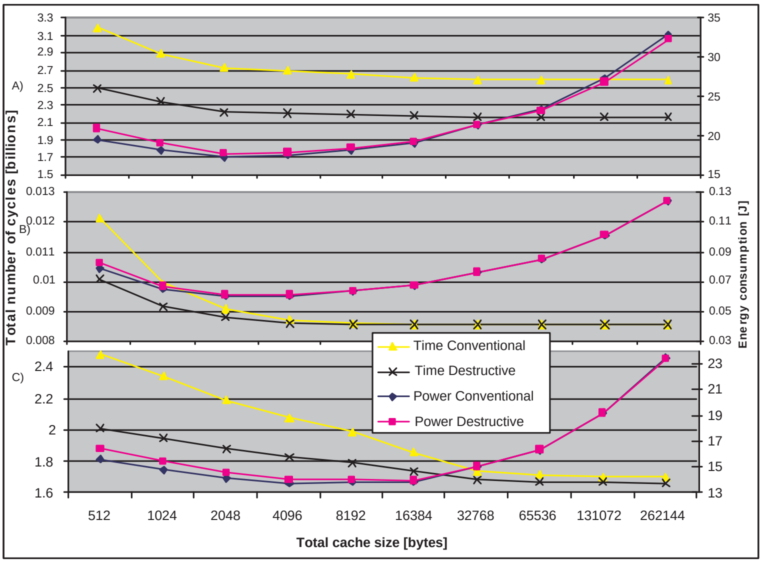


Figure II.11: Number of clock cycles and energy consumption for a) *Ammp.*, b) *Art* and c) *Twolf.*

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 II.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 II.12c). A cache of size 16 kbytes is the optimal for minimizing (linearly) both energy consumption and execution time.

For the *art* application the energy consumption and execution time are shown in Figure II.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 II.12b). Caches of 4 kbytes is the optimal size for minimizing (linearly) both energy consumption and execution time.

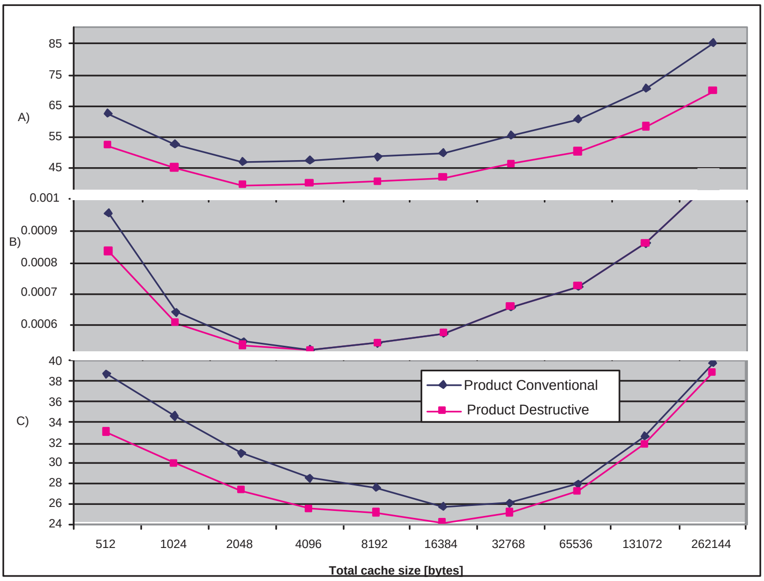


Figure II.12: Product of execution time and energy consumption for a) *Ammp.*, b) *Art* and c) *Twolf*

A study of performance and energy consumption for *ammp* application as a function of cache size is shown in Figure II.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 II.12a). Caches of 2 kbytes is the optimal size for minimizing (linearly) both energy consumption and execution time.

II.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 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 II.5 and II.6. The values are taken from simulations. By looking at the components in Equation II.5 and comparing this to Equation II.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.

$$E_{conv} = 13.2J + 58 * 10^6 * E_{MEM} \tag{II.5}$$

$$E_{dest} = 12.9J + 97 * 10^6 * E_{MEM} \tag{II.6}$$

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.

II.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.

Bibliography

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. IEEE Computer, Volume 35, Issue2, Feb. 2002.

[2] D. Brooks, V. Tiwari, and M. 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. ISBN 1-58113-232-8. doi: http://doi.acm.org/10.1145/339647.339657.

[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] 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, pages:14-25, June 2002.

[5] H. Dybdahl, M. Grannaes, and L. Natvig. Cache write-back schemes for embedded destructive-read DRAM. Submitted to ARCS 2006, 2006.

[6] D. G. Elliott, W. M. Snelgrove, and M. 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, Apr. 1995.

[8] L. 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] C.-L. Hwang, T. Kirihata, M. Wordernan, J. Fifield, D.Storaska, D. Pontius, G. F. 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. IBM microelectronics presentation: Embedded DRAM comparison charts. IBM Microelectronics, 2003.

[12] ITRS. International technology roadmap for semiconductors. http:public.itrs.net, 2003.

[13] S. S. Iyer, J. 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. 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, Oct. 1999.

[16] M. Khellah and M. 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, Apr. 2003.

[18] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[19] P. 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, Mar. 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, Nov. 1997.

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

[24] P. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, A. V. E. Brockmeyer, C. Kulkarni, and P. Kjeldsberg. Data and memory optimization techniques for embedded systems. ACM Trans. Design Automation of Electronic Systems, 6(2):149–206, Apr. 2001.

[25] Y.-H. Park, H.-J. Yoo, and J. 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, Aug. 1996.

[27] V. Raghunathan, M. B. Srivastava, and R. 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. ISBN 1-58113-688-9. doi: http://doi.acm.org/10.1145/775832.776059.

[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. ISBN 0-89791-786-3. doi: http://doi.acm.org/10.1145/232973.232984.

[29] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Western Research Lab, Research Report 2001/2, 2001.

[30] L. Yerosheva, S. Kuntz, J. Brockman, and P. Kogge. A microserver view of HTMT. Parallel and Distributed Processing Symposium, Proceedings 15th International, pages:10, Apr. 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, Mar. 2003.

Paper III

Hardware Prefetching Using Shadow Tagging

Marius Grannæs and Lasse Natvig

In *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008

Abstract

This paper presents a novel technique for dynamic selection of parameters for prefetching heuristics based on the use of shadow tag directories. Previous methods have been either static, made for a specific prefetching heuristic, or based on phase detection and tuning. The most flexible of these methods is phase detection and tuning. However, it has a serious drawback as it degrades performance while exploring the parameter space, as each configuration is tested on the running program. Our approach explores the parameter space using an extra structure called a shadow tag directory. This allows us to explore the parameter space without interfering with the running program, such that a larger parameter space can be explored without impacting performance. This paper examines the performance of this technique on tagged sequential prefetching, czone/delta correlation prefetching and reference prediction tables. In addition, we compare our results with a feedback directed approach. We show an overall 24% improvement over the best static sequential prefetcher and an 18% improvement versus feedback directed sequential prefetching on memory intensive SPEC benchmarks.

III.1 Introduction

The performance of general purpose microprocessors continue to increase at a rapid pace, but main memory has not been able to keep up. In essence, the processor is able to process several orders of magnitude more data than main memory is able to deliver on time. Numerous techniques have been developed to tolerate or compensate for this gap, including out-of-order execution, caches, prefetching and bypassing [9].

By utilizing prefetching, data that has not been referenced before can be inserted into the cache by analyzing the behaviour of the program and anticipating what data is needed in the future. Previous prefetching engines, such as sequential [17], reference prediction tables [4] (RPT) and czone/delta correlation [11] (C/DC), have static configurations. To provide the best average performance across a multitude of benchmarks a moderately aggressive prefetching configuration would be used. This leads to performance degradation on some programs, as the prefetching is too aggressive, leading to memory bus congestion and cache pollution, while on other programs the full potential of prefetching can not be achieved because the aggressiveness is too low.

Consequently, there has been much interest in dynamic parametrization of prefetching heuristics. The idea is to adapt the prefetching heuristic to the running program by analyzing its behaviour. After analyzing the behaviour of the program, the parameters of the prefetcher(prefetching degree, prefetch distance, czone size etc) is adjusted accordingly.

III.1.1 Contributions

This paper investigates a general method for dynamically selecting prefetching parameters based on a shadow tag directory. Conceptually, a shadow tag directory is similar to a regular cache tag directory, which allows us to simulate the effects of altering the prefetcher configuration without interfering with the running program [7, 8, 14]. This method can be adapted to new prefetching heuristics with little effort. In addition, we show the benefit of dynamic parameterization with a sample heuristic that optimizes for performance by estimating prefetcher usefulness and bandwidth usage. The method is then evaluated on the memory intensive benchmarks in the SPEC2000 suite. We evaluate the technique combined with three different prefetchers, sequential, C/DC and RPT prefetching. We compare our scheme to no prefetching, their static counterpart, feedback directed prefetching and a perfect L2 (a cache that always hits). We show an overall 24% improvement over the best static sequential prefetcher and an 18% improvement versus feedback directed sequential prefetching on memory intensive SPEC benchmarks.

III.2 Previous Work

III.2.1 Feedback Directed Prefetching

A few researchers have investigated dynamic parameterization of prefetching heuristics in the past. A direct method is the feedback-based approach, which measures accuracy, timeliness and cache pollution caused by the prefetcher at runtime [19]. The values obtained are then fed into a state machine which in turn increases or decreases aggressiveness accordingly. Feedback directed prefetching estimates prefetcher accuracy by tagging each cache line with a single bit to signify that the line has been prefetched, but not yet accessed. Two counters are used; one that is incremented every time a prefetch is issued and another that is increased when a cache line that has the bit set is accessed. The ratio can then be used to estimate prefetcher accuracy. If the accuracy is high then the prefetcher aggressiveness is increased. Prefetcher timeliness and cache pollution is estimated in a similar way.

III.2.2 Tuning

A more general approach to dynamic reconfiguration is tuning. It has been successfully used in other areas as well, such as adapting the size of the issue queue to make it more power-efficient [3]. AC/DC prefetching is an extension of the static C/DC prefetcher that uses tuning to adapt to program phases [12].

Tuning works by detecting program phase changes by using instruction working sets, basic block vectors (BBV) and conditional branch counts [6]. When the program changes phase, it is likely that it will benefit from a reconfiguration of its resources [5]. The prefetching hardware will then search through the parameter space and select the best configuration for that phase. Each configuration is tried for a set amount of time (measured in clock cycles, number of L2 misses etc). When all the configurations have been explored, the best configuration is then used until the next phase change. However, as the parameter space grows, the time consumed by the search can become large. Thus, performance is degraded while the algorithm searches [15].

Tuning is usually implemented as a state machine with three states: stable, unstable and tuning [1, 5]. The stable state indicates that the best configuration is selected for that particular portion of the program. When a program undergoes a phase change it enters the unstable state. In this state the tuning algorithm simply waits while the program stabilizes. Once the program has stabilized, the tuning algorithm enters the tuning state, where it will search for the best selection of parameters.

Efficient tuning relies on good phase detection mechanisms to reliably detect phase changes and identify similar phases. Phase change detection must be accurate, but allow for minor changes in program behaviour, so that it doesn't trigger tuning unnecessarily.

III.2.3 Shadow Tag Directories

A shadow tag directory is functionally similar to a regular cache directory. However, the shadow tag directory does not have a corresponding data array. Its purpose is to simulate, at runtime, the result of having a different prefetching configuration. Both directories receive the same memory access stream, and is manipulated accordingly.

Dybdahl et al. used this technique to augment a cache replacement policy for chip multiprocessors [7]. They used shadow tags for dynamically switching between their cache replacement policy and the traditional LRU policy, based on their relative performance. A similar work by Dybdahl et al. used shadow tags to improve cache partitioning in chip multiprocessors [8]. Simultaneously, Qureshi et al. used shadow tags[1] to switch between a cache replacement policy that optimizes the amount of memory level parallelism and the traditional LRU policy [14].

Both papers illustrate that the shadow tag directory only needs to contain about 20 sets to be effective. Qureshi establishes this by use of an interesting statistical proof. Although the two papers differ in the selection of the 20 sets, the results are similar.

III.3 Methodology

III.3.1 Shadow Tag Controlled Prefetching

We propose a new approach to dynamic tuning of prefetching heuristics by using a shadow tag directory. Our architecture has two levels of cache and main memory as shown in Figure III.1. The main prefetching engine is connected to the second level cache. All second level cache accesses are also propagated to the shadow tag directory. The shadow tag directory is similar in configuration to the L2 cache, but does not hold any actual data. It is connected to another prefetch engine running with an alternative configuration.

The controller is a small unit responsible for reconfiguring the prefetch engines based on data gathered from the two tag directories. After 2000[2] cache misses have occurred in the real L2 cache, the performance of the two configurations are compared. If the results from the shadow tags are better than the results from the real L2 cache, then the configuration of the shadow-prefetcher is applied to the main prefetcher.

In either case, the shadow-prefetcher is then reconfigured, so that another configuration is explored.

[1] Qureshi et al. uses the term auxiliary tag directory.

[2] Earlier research has by Dybdahl[8] has shown that comparing every 1000 misses is preferable. We found that an interval of 2000 L2 misses provided more robustness.

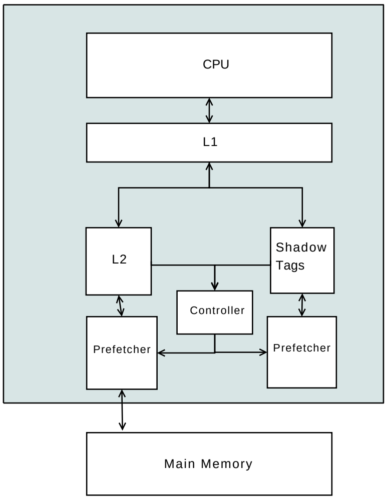


Figure III.1: The proposed architecture. The L1 cache is connected to both the L2 cache and the shadow tag directory. The controller evaluates the performance of the two configurations and reconfigures the prefetchers accordingly.

The shadow tag directory is similar to the L2 tag directory, every access by the processor is inserted into the tag directory. In addition, the miss address stream fed into the shadow-prefetcher which uses this information to generate (simulated) shadow-prefetches. These shadow prefetches are then inserted into the shadow tag directory, so that the shadow tag directory reflects the state it would have been in if the prefetches would have been issued.

To estimate the hardware overhead of using a shadow tag directory we have used the cache modelling tool Cacti [16]. In 65nm technology the tag directory of the L2 cache used in this paper uses 0.12 mm^2, while the L2 itself uses 7.48 mm^2. A shadow tag array would thus increase the size of the L2 by 1.6%.

However, both Dybdahl and Qureshi [7, 8, 14] have shown that it is only necessary to replicate 20 of the 512 sets for accurate predictions. In other words, the area requirements for the shadow tags can potentially be reduced to around 0.005 mm^2, or about an increase in L2 size of 0.06%.

Additionally, it is possible to use the shadow tag directory for other performance enhancing techniques such as increasing the amount of memory level parallelism [14] and temporal locality of the cache [7, 8]. Thus this hardware cost can be amortized across several techniques.

In this paper we have used a full-sized shadow tag directory for our simulations, replicating all the sets in the original L2 cache.

III.3.2 Prefetch Configuration Selection Heuristic

The overall goal is to increase performance in terms of IPC (Instructions Per Clock Cycle). It is not possible to directly measure the effect on IPC by using shadow tags as they contain no data. By using the methods described by Srinath [19] it is possible to estimate prefetcher accuracy, timeliness and cache pollution in addition to the number of cache misses and cache hits.

When a prefetch is issued, a counter is increased (indicating the number of prefetches issued) and a prefetched-bit is set in the corresponding cache line. This bit is already present when using sequential prefetching, and thus causes no additional overhead. The first time a cache line with this bit set is referenced by the program, the bit is cleared and another counter (indicating the number of successful prefetches) is increased. By dividing the two numbers, we get an estimate of the prefetchers accuracy.

A simple method for estimating memory traffic is used. We record the number of L2 misses that would have occurred if the shadow tag directory configuration was used. By dividing the number of clock cycles elapsed by the amount of L2 misses, we get an estimate of the amount of memory traffic that would occur if the shadow configuration is used.

Initially, the real prefetcher is set to a prefetching degree of zero (or off). The shadow prefetcher is set to a random prefetching degree (0-16) and a random prefetch distance (0-16) (and a random czone size (4KB-4MB) if applicable). After 2000 L2 misses has occurred, the two configurations are evaluated.

We use a fitness function to evaluate both configurations:

$$F = Hits - \frac{Late}{4} - \lfloor 2^{BW-T} \rfloor \tag{III.1}$$

Our heuristic has three components. First, we use the number of cache hits. Since we evaluate the two configurations after 2000 L2 misses, this gives us indirectly a measurement of the hit-ratio. The second component is the number of late prefetches. Late prefetches are prefetches that have been issued, but have been issued too late to fully cover the complete memory latency. This component is estimated by using a prefetch bit in the MSHRs in a similar manner as the method used by Srinath [19]. To decrease the number of late prefetches a prefetcher needs to issue prefetches with a larger *prefetch distance*. The last component is a simple function depending on the bandwidth usage. We estimate bandwidth usage by using the number of cache misses that have occurred and the time elapsed, this number is denoted BW (Bandwidth Usage). To ensure that this component does not become dominant when there is ample bandwidth available, we subtract a fixed threshold value from this number (T).

If the fitness of the shadow tag directory exceeds that of the real cache, the configuration of the shadow tag prefetcher is adopted to the real prefetcher. Then a new

Clock Frequency	4 GHz
Processor Width	4 instructions/cycle
Register Update Unit	64 instructions
Load/Store Queue	32 instructions
Fetch Queue	16 instructions
Functional Units	4 ALUs, 1 Integer Multiply/Divide 4 FPUs, 1 Floating Point Multiply/Divide
Branch Predictor	Combined, Bimodal 4K entry table, 2-level 1K table, 10 bit history table, 4K Chooser, 4-way 512 entry BTB, 15 cycles miss predict penalty
TLB (D & I)	128 entry full associative, 30 cycle miss penalty
Level 1 D-Cache	8K 4-way, 64B blocks, LRU 2 cycle latency
Level 1 I-Cache	8K 4-way, 64B blocks, LRU 2 cycle latency
Level 2 Cache	512K 8-way, 128B blocks, LRU, 7 cycle latency
Main Memory	160 cycle latency, max bandwidth 9GB/s

Table III.1: The simulation parameters used with SimpleScalar.

random configuration for the shadow tags is chosen and the process is repeated.

III.3.3 Experimental Setup

For the evaluation of this proposed architecture we have extended the Simplescalar [2] simulator with shadow tag directories, prefetching and a new model for main memory that simulates contention. Our main memory model models DDR2 memory and accounts for split transactions, open/closed pages, burst mode, multiple channels and pipelining. The setup can be found in table III.1. We used the reduced data set [10] SPEC2000 benchmarks [18]. This datasets allows us to run each benchmark to completion. To compensate for the small datasets, the L2 cache is relatively small compared to the aggressive core. This was done to force more misses in the L2 cache and further stress the memory subsystem.

Out of the 26 benchmarks in the suite, we have selected the 10 most memory intensive applications measured in terms of the number of memory accesses per instruction. On the remaining 16 benchmarks in the SPEC2000 benchmark suite we observe no significant performance improvement or degradation. This is due to the entire data set of the benchmark fitting inside the L2 cache, thus giving little opportunity for prefetching. Thus these 16 benchmarks are omitted in the remainder of this paper.

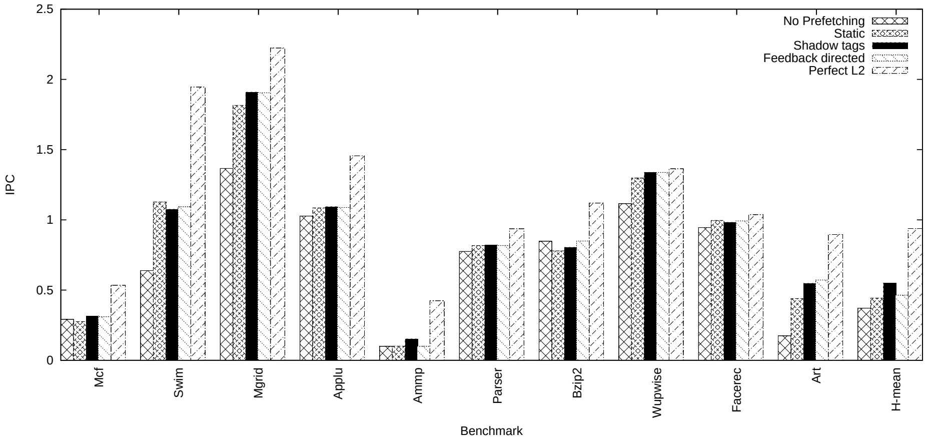


Figure III.2: Performance of dynamic parameter selection on static prefetching.

III.4 Results

We have examined our technique on three different prefetching heuristics, tagged sequential, czone/delta correlation (C/DC) and reference prediction tables (RPT). We compare our dynamic scheme to no prefetching, the best static parameter selection, feedback directed prefetching and a perfect L2 (a L2 that never misses). The original implementation of feedback directed prefetching included a method for controlling the insertion policy of prefetched cachelines. This part of feedback directed prefetching has been omitted to make it easier to compare the two methods.

III.4.0.1 Sequential Prefetching

In figure III.2 we show the performance of the 10 most memory-constrained benchmarks in the SPEC2000 benchmark using sequential prefetching. First, we observe that the static configuration leads to degraded performance on both Mcf and Bzip2 compared to the case of no prefetching, while our shadow tag scheme only has a minor regression on Bzip2. Furthermore, we see significant improvements on the Ammp benchmark. Shadow tag prefetching gets an IPC of 0.55 while feedback directed prefetching gets an IPC of 0.46. This is due to the very low accuracy (4%) of sequential prefetching, which leads feedback directed prefetching to not increase aggressiveness to the required level. In total we observe a 18% increase in harmonic mean compared to feedback directed prefetching, 24% improvement over the static configuration and 48% over no prefetching. In addition, we observe that the largest regression compared to feedback directed prefetching is 5% (Bzip2).

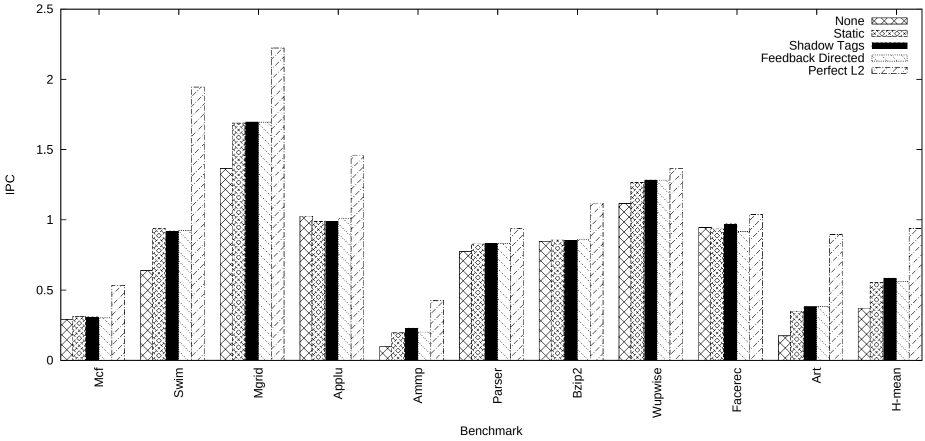


Figure III.3: Performance of dynamic parameter selection on C/DC prefetching.

III.4.0.2 C/DC prefetching

In figure III.3 we show the performance of the techniques on czone/delta correlation prefetching. Feedback directed prefetching has no method for controlling czone size directly, so we have used the best static value for the czone size. We observe that the static configured C/DC prefetcher shows regressions on Applu and Facerec. However, shadow tags performs better than both feedback directed and static prefetching. In addition, the performance on Ammp is increased even further. Overall, we observe an increase in harmonic mean by 58% compared to no prefetching, 5.6% improvement versus static prefetching and a 5% improvement versus feedback directed prefetching. It is worth noting that we observe a significantly higher prefetching accuracy on C/DC prefetching than sequential prefetching.

III.4.0.3 RPT prefetching

Reference Prediction Tables is a very robust technique. Our results for this prefetching technique is shown in figure III.4. It has a very high prefetch accuracy (more than 90%), but lower coverage than the other prefetchers. Even with a static configuration we observe no regressions versus the case of no prefetching. The dynamic schemes perform as well or better than the static configuration in all but two cases: Ammp and Facerec. We observe an increase of 58.2% in harmonic mean by using shadow tags versus no prefetching. However, the gains versus static prefetching is smaller, only 4.8%. The difference versus feedback directed is minimal, 0.48%. It should be noted that RPT prefetching requires that the address of the load-instruction is included with every memory request, thus making it expensive to implement in hardware.

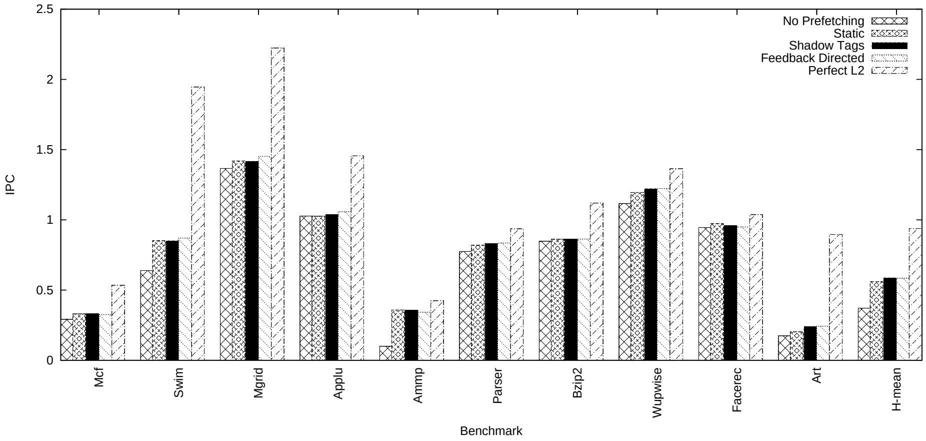


Figure III.4: Performance of dynamic parameter selection on RPT prefetching.

III.4.1 Bandwidth Usage

Prefetching will necessarily increase bandwidth requirements (unless the prefetcher is 100% accurate). Figure III.5 shows the bandwidth usage for each benchmark. The numbers have been normalized to the case of no prefetching. Overall, sequential prefetching requires a substantially more bandwidth than C/DC or RPT prefetching. Furthermore, the bandwidth requirements of Ammp and Bzip2 stands out as excessive. This can be justified, especially on Ammp where performance is increased by a considerable amount. On average shadow tag prefetching requires 11% more bandwidth for sequential prefetching compared to a static configuration, while it requires 0.7% more on C/DC prefetching and 3.4% more on RPT prefetching. However, it should be added that our heuristic is designed to use whatever bandwidth is available.

III.4.2 Sensitivity Analysis

In this section, we look at some of the parameters that we have used in our heuristic. In figure III.6 we look at how often the reconfiguration occurs (*Misses between reconfiguration*) and the number of consecutive checks where the shadow tags must outperform the real cache for the configuration of the shadow tags to be adopted (*Count*). These two parameters do not have a significant impact on the harmonic mean of IPC. The difference in IPC is only 2.5%. It should be noted that configurations such as 500 L2 misses and a count of 1 is more unstable. Some benchmarks will benefit greatly, while others gets reduced performance, however, the speedups and slowdowns evens up over multiple benchmarks.

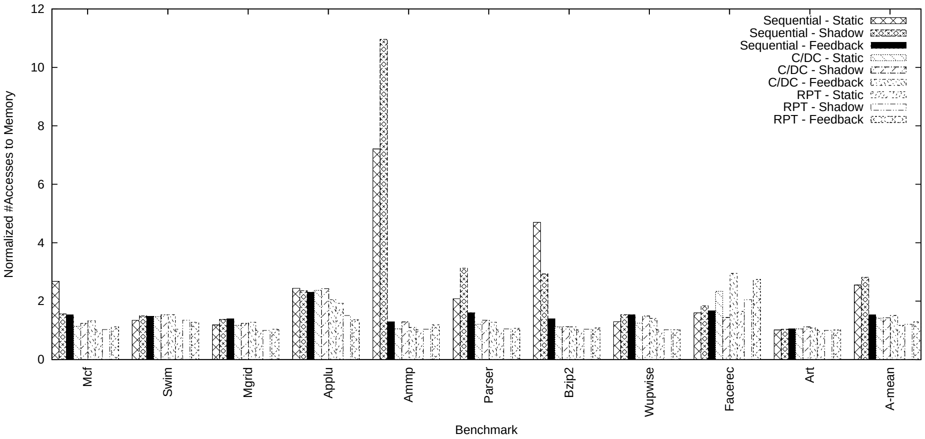


Figure III.5: Number of main memory accesses for different combinations of prefetching heuristics and parameter selection methods. Values are normalized to no prefetching.

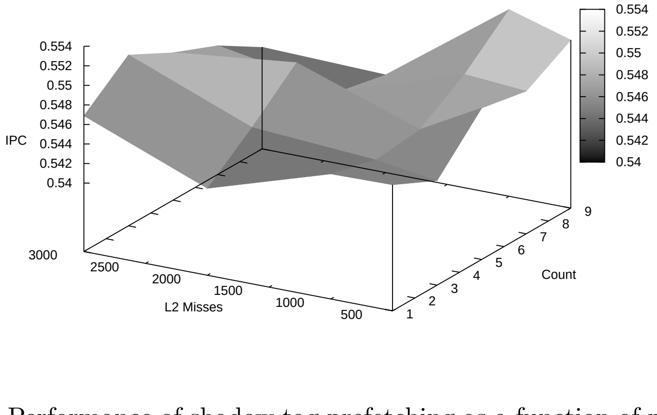


Figure III.6: Performance of shadow tag prefetching as a function of parameters to the heuristic.

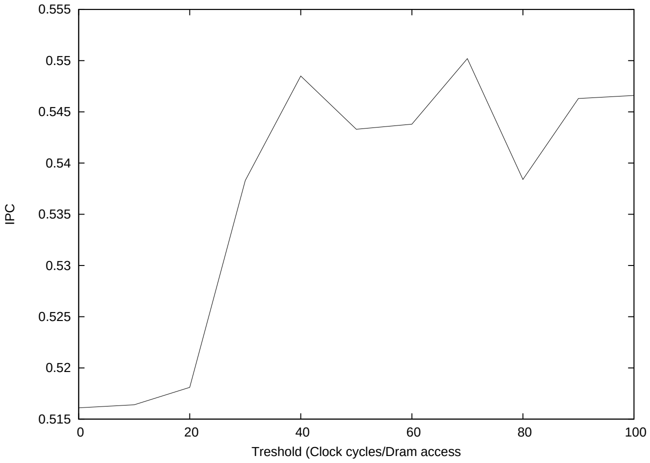


Figure III.7: Performance of shadow tag prefetching as a function of the bandwidth threshold parameter.

Figure III.7 shows the harmonic mean of IPC for the benchmarks as a function of the bandwidth threshold. The number on the X-axis represents the average number of clock cycles between memory accesses. A low number denotes that a configuration with a high bandwidth usage will be penalized less. This graph shows that if the heuristic is not penalized for using too much bandwidth then performance drops. It is worth noting the sharp increase in IPC in the interval 20 to 40. This interval represents a bus utilization around 80%.

III.5 Discussion

III.5.1 Parameter Space Exploration

In this paper the configuration to be tested on the shadow tags were chosen randomly across the whole parameter space. We have also experimented with using hill climbing to explore the parameter space. Hill climbing only looks at small changes in the configuration and selects the best configuration out of those tested. A known problem with hill climbing is that it can get stuck in local optima and thus not find the global optimum. We believe that such local optima are not likely to be stable as the program runs. However, because hill climbing “moves” slower across the parameter space, we observed a 18% decrease in performance on some benchmarks that were too short for the hill climbing technique to converge on a good solution.

There are other approaches that could be used, such as genetic algorithms and simulated annealing [13].

III.5.2 Clearing the Shadow Tags

Initially, we were concerned that not clearing the shadow tag directory between two configurations would pollute subsequent measurements. We examined the effect of copying the contents of the real tag directory to the shadow tag directory after each reconfiguration. We found that copying had little impact on performance (we observed only a 0.1% improvement by using a copying function). In addition, the cost of such a mechanism would be prohibitive both in terms of area and power.

III.6 Conclusion

In this paper we present a novel technique for dynamic parameterization of prefetching heuristics. By using this technique we observe an overall improvement in IPC by 24% over the static configuration and a 18% improvement over feedback-directed prefetching. However, the relatively large improvements seen on single benchmarks are equally important. In addition, we observed no regressions against the case of no prefetching, which makes the method robust and only one regression against the case of a static prefetcher.

In terms of cost, the L2 is increased in size by 1.6% for an overall gain of 24%. It might be possible to reduced this to about 0.06% if the methods by Dybdahl et al. and Qureshi et al. can be used with prefetching. The shadow tag directory can be used for other purposes as well, such as optimizing memory level parallelism and enhancing the cache replacement policy, thus amortizing this cost over several performance enhancing techniques.

Bibliography

[1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, 2000. ISBN 1-58113-196-8. doi: http://doi.acm.org/10.1145/360128.360153.

[2] D. Burger and T. M. Austin. Simplescalar toolset 3.0b, 2003. `http://www.simplescalar.com`.

[3] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Power-Aware Computer Systems: First International Workshop, PACS 2000*, 2000.

[4] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[5] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings. 29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.

[6] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, 2002.

[7] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. In *MEDEA Workshop, PACT*, 2006.

[8] H. Dybdahl, P. Stenstrom, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of IEEE International Conference on High Performance Computing*, 2006.

[9] J. L. Hennesey and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan Kaufmann Publishers, 2003. ISBN 1-55860-724-2.

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

[11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Micro, IEEE*, 25:90–97, Jan. 2005.

[12] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[14] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2608-X. doi: http://dx.doi.org/10.1109/ISCA.2006.5.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings. 30th Annual International Symposium on Computer Architecture*, pages 336– 347, 2003.

[16] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2, Compaq Western Research Laboratory, August 2001.

[17] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356887.356892.

[18] SPEC. Spec 2000 benchmark suites, 2000. `http://www.spec.org`.

[19] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

Paper IV

Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors

Marius Grannæs, Magnus Jahre and Lasse Natvig

In *XXVI IEEE International Conference on Computer Design (ICCD)*, 2008

Abstract

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM is exploited to increase throughput.

Traditionally, prefetching reduces latency by fetching data before it is needed. In this paper we explore how prefetching can be used to increase memory throughput. We present our own low-cost open-page prefetch scheduler that exploits the 3D structure of DRAM when issuing prefetches. We show that because of the complex structure of modern DRAM, prefetches can be made cheaper than ordinary reads, thus making prefetching beneficial even when prefetcher accuracy is low. As a result, prefetching with good coverage is more important than high accuracy. By exploiting this observation our low-cost open page scheme increases performance and QoS. Furthermore, we explore how prefetches should be scheduled in a state of the art memory controller by examining sequential, scheduled region, CZone/Delta Correlation and reference prediction table prefetchers.

IV.1 Introduction

Chip Multiprocessors have been introduced by virtually all makers of high performance processors. CMPs shifts the focus away from the traditional uniprocessor paradigm, where low latency and instruction-level parallelism (ILP) is important to a paradigm where throughput and thread-level parallelism (TLP) dominates.

This shift is reflected in the memory subsystem as well, where the memory controllers have traditionally been used to reduce system latency. However, as more cores are added to a chip, off-chip bandwidth are shared across cores, thus increasing the pressure on this resource and lowering locality in the memory access stream. Thus, memory controllers have been designed to optimize for maximum throughput, at the expense of increasing worst-case latency.

This increase in throughput has been made possible by exploiting the 3D structure of modern DRAM [4]. DRAM is organized in several banks. Each bank is organized as a matrix of rows and columns of DRAM cells as shown in figure IV.1. In a normal read operation, a bank and a row is first selected for activation. The charges from this row of capacitors are then amplified by sense-amplifiers in the DRAM module and stored in a large latch. Each such row is commonly referred to as a page. A page is normally about 1KB to 4KB large, whereas a cacheline is typically 64-256B large. Thus, a page will typically hold several consecutive cachelines. The portion of the page that was requested is then transferred over the data-bus. When the page is no longer needed, the memory controller instructs the DRAM module to write the latch contents back into the DRAM cells, preserving the contents of the page. This is referred to as closing the page.

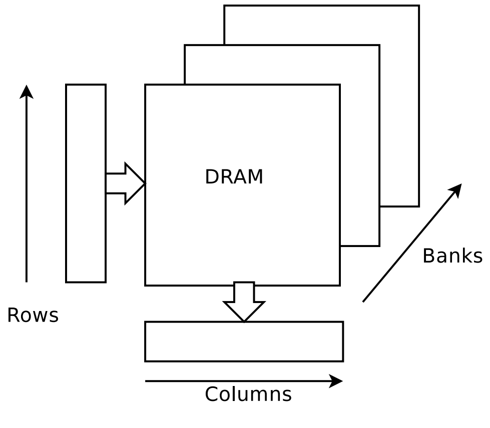


Figure IV.1: The 3D structure of modern DRAM.

In terms of latency, opening and closing a page is expensive, while getting data out of the latches and over the data-bus is comparatively cheap. In addition, there is a minimum allowed time between opening and closing a page (the minimum activate-to-precharge latency). Thus a single read is slow, but reading the next cache block is relatively cheap as the page is already open and the data is in the latch. This property is exploited by the First Ready, First Come, First Served (FR-FCFS) memory controller proposed by Rixner et al. [16]. This type of memory controller allows accesses that uses an already open page to be scheduled even if the request is not the oldest.

Traditionally, prefetching has been used to decrease latency for a single operation by speculatively bringing data into the cache before it is needed. In this paper we exploit the 3D structure of modern DRAM to demonstrate how prefetching can be used to increase off-chip bandwidth utilization. Because there is a lower cost associated with fetching data that resides in an open page, we prefetch this data, provided that our confidence that the data will be useful is high enough. In addition, we show that prefetching can be effective at relatively low accuracy, due to the low cost of piggybacking prefetches compared to single reads. Finally, we present our low cost open page prefetching scheduling heuristic which exploits this observation.

IV.2 Previous Work

IV.2.1 Prefetching

Previously, Wei-Fen et al. [8] have examined how prefetches can be scheduled in a uniprocessor context with Rambus DRAM. They used a dedicated prefetch queue with a LIFO insertion policy with a scheduled region prefetcher. In addition, Cantin et al. [2] exploited open pages to increase the performance of their stealth prefetcher.

There exists a multitude of different prefetching schemes. The simplest is the *sequential prefetcher* [18], which simply fetches the next block whenever a block is referenced. However, more complex types exists as well, such as the *CZone/Delta Correlation (C/DC)* prefetcher proposed by Nesbit et al. [12, 13]. C/DC divides memory into CZones and analyses patterns contained in the reference stream by using a Global History Buffer (GHB) to store recent misses to the cache. Lin et al. [9] introduced *scheduled region prefetching* (SRP) which issues prefetches to blocks spatially near the addresses of recent demand missed when the memory channel is idle. Other types, such as the *Reference Prediction Table Prefetcher* (RPT) proposed by Chen and Baer [3] examines the pattern generated by a load instruction with a state machine. Somogyi et al. proposed *Spatial Memory Streaming* (SMS) [19]. SMS uses code-correlation to predict spatial access patterns.

IV.2.2 Memory Controllers

Memory access scheduling is the process of reordering memory requests to improve memory bus utilization. Rixner et al. [16] showed that significant speed-ups are possible when memory request reordering is applied to stream processors. In addition, Shao et al. [17] proposed *burst scheduling* in which multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization. Finally, Zhu et al. [24] showed that it is beneficial to divide the memory requests into smaller parts, and give priority to the words responsible for a processor stall in a multi-channel DRAM system.

CMPs, processors with SMT support and conventional shared-memory multiprocessors also benefit from memory access scheduling. Zhu et al. [23] showed that DRAM throughput could be increased in an SMT processor by using ROB and IQ occupancy status to prioritize requests. Furthermore, Hur et al. [5] use a history-based arbiter to adapt the DRAM port and rank schedule to the application's mix of reads and writes for the dual-core Power5 processor. In addition, Natarajan et al. [11] showed that a significant performance improvement is available by exploiting memory controller features in a conventional, shared-memory multiprocessor.

In CMPs, the memory bus is shared between all processing cores and a number of researchers have looked into how this can be accomplished in a fair way [6, 10, 14, 15]. In general, bandwidth is divided among threads according to their priorities at the same time as requests are scheduled in a way that improves DRAM throughput.

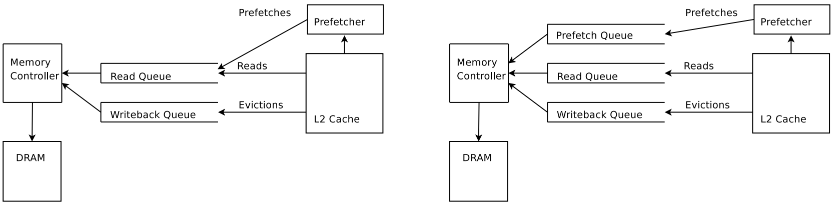
IV.3 Prefetch Scheduling

A prefetching heuristic can be characterized by using two distinct metrics: *Accuracy* is a measure of how many of the issued prefetches have actually been useful to the processor [22], while coverage measures how many of the potential prefetches have been issued.

Because prefetching is a speculative technique, there are two potential sources for performance degradation. Firstly, prefetching consumes additional bandwidth as some data transferred over the memory bus is not used. Secondly, it can pollute the cache, by displacing data that is still needed.

The FR-FCFS memory scheduler [16] is a high throughput memory scheduler. It exploits the 3D structure of modern DRAM by allowing requests that would access an already open page to bypass the normal FCFS queue. FR-FCFS prioritizes memory requests in the following manner: 1) Ready operations (operations that access open pages), 2) CAS (column selection) over RAS (row selection) commands, and 3) Oldest request first. In addition, reads have a higher priority than writes.

There are two basic ways to introduce prefetching into the FR-FCFS memory controller. The simplest approach is to insert prefetch requests into the read queue, as shown in figure IV.2(a). A more sophisticated approach introduced by Lin Wei-Fen et al. [8] is to use a dedicated queue for prefetches as shown in figure IV.2(b). In this approach, prefetches are prioritized after writebacks, so the priority rule becomes: Prioritize read operations over writeback operations over prefetch operations.



(a) Conventional prefetch scheduling (b) Dedicated prefetch queue

Figure IV.2: Prefetch scheduling policies

IV.4 Low cost open page prefetching

After a demand read to DRAM is serviced, the page that the demand read resided in is still open, and in most cases cannot be closed due to the minimum activate to precharge latency. Other DRAM banks can still be utilized. If a prefetch or read is issued to this open page, there is little latency as the data requested is already in the latch. In this paper we refer to this as *piggybacking*.

By allowing prefetches to piggyback on regular read requests, the cost of prefetching is effectively reduced. In the dedicated prefetch queue approach, prefetches are only issued if they can piggyback on another request, or if the bus is idle. Suppose a processor requires data at locations X_1 and X_2 that are located on the same page at times T_1 and T_2. There are two separate outcomes: If T_1 and T_2 are sufficiently close, both requests will be in the memory controller at the same time, and request 2 can piggyback on request 1. Thus the page only needs to be opened once. If the two requests are sufficiently separated in time, the two requests cannot be piggybacked on each other, thus forcing the page to be opened twice. This reduces overall throughput. In the second case, prefetching X_2 can increase performance by both reducing latency and increase memory throughput. However, because prefetching is a speculative technique, its prediction for what data is needed in the future might be wrong. Thus, there is a break-even point where the benefit of prefetching is balanced against the cost of prefetching.

To test this assumption we have conducted experiments on 4 different prefetching heuristics (Sequential, SRP, C/DC and RPT) with 10 different prefetching configurations (each) on 40 different workloads. We measured the accuracy of the prefetcher and the IPC improvement (versus a configuration with no prefetching). Our results are shown in figure IV.3.

In this graph it is clear that most of the points fall into 2 quadrants. One where accuracy is below 38% and performance is decreased, while another where accuracy

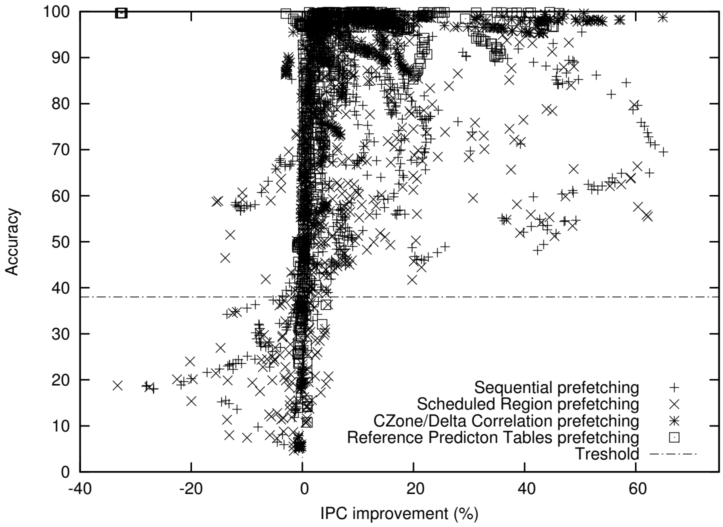


Figure IV.3: IPC improvement as a function of accuracy

is above 38% and performance is increased.

Our prefetch scheduler exploits this observation by measuring prefetch accuracy at runtime. If the accuracy falls below a treshold (in our experiments 38%) then prefetches are no longer piggybacked on open pages and only issued if the bus is idle.

We use an accuracy estimator similar to the one used by Sriniath et al. [21]. When a prefetch is issued, a counter is increased (indicating the number of prefetches issued) and a prefetched-bit is set in the corresponding cache line. This bit is already present when using sequential prefetching, and thus causes no additional overhead. The first time a cache line with this bit set is referenced by the program, the bit is cleared and another counter (indicating the number of successful prefetches) is increased. By sampling the successful prefetch counter every time the 10 bit issued counter wraps, we get an estimate of the prefetchers accuracy.

IV.5 Methodology

We used the system call emulation mode of the cycle-accurate M5 simulator [1] to evaluate our scheme. The processor architecture parameters for the simulated 4-core CMP are shown in table IV.1, and table IV.2 contains the baseline memory system parameters. We have extended M5 with a crossbar interconnect, a detailed DDR2 memory bus and DRAM model, a FR-FCFS memory controller and prefetching.

Table IV.1: Processor Core Parameters

Parameter	Value
Processor Cores	4
Clock frequency	3.2 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table IV.2: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB, 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB, 8-way set associative, 64B blocks, 1 cycle latency
Level 2 Unified Shared Cache	4 MB, 16-way set associative, 64B blocks, 14 cycles latency, 8 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 9 cycles latency, 64B wide transmission channel
DDR2 memory	400 Mhz Clock, 8 banks, 1KB pagesize, 4-4-4-12 timing, dual channel in lock-step

Our DDR2-implementation [7] models separate RAS, CAS and precharge commands. In addition, we model pipelining of requests, independent banks, burst mode transfers and bus contention. The FR-FCFS memory controller has a 128 entry read-queue, 64 entry writeback queue and a 128 entry prefetch queue. As the conventional method of issuing prefetches has no separate prefetch queue, the read queue has been increased to 256 entries to make comparison more fair in terms of area. Unless otherwise noted, we use 4KB regions in scheduled region prefetching, 256KB CZones, a 1024-entry global history buffer and a 16-entry reference prediction table.

The SPEC CPU2000 benchmark suite [20] is used to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in table IV.3. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a random number of clock cycles between 1 and 1.1 billion. Then, detailed simulation is carried out for 100 million clock cycles measured from the clock cycle the last core finished fast forwarding. As our metric of throughput we have used the average IPC of all 4 cores. In most cases, performance is measured as the relative increase in speed compared to the no prefetching case.

IV.6 Results

IV.6.1 Scheduled Region Prefetching

In figure IV.4 we show the relative performance of each of the prefetch scheduling policies. In this experiment we use a scheduled region prefetcher (SRP) with 4KB regions. The conventional and dedicated prefetch queue options give an average of 14.4% increase in performance versus the no prefetching case, while the average increase for our scheme is 17.1%. In addition, prefetching causes performance degradation in 9 out of the 40 cases. Our prefech scheduling policy reduces the performance penalty on 6 of these workloads. However, a lot of information is lost in averages. For instance, the performance increased on workload 1 is only 1% in other schemes, while our method increases performance by 15%. Similar results can be seen in workload 6, 7, 23, 25, 27, 28, 32 and 38.

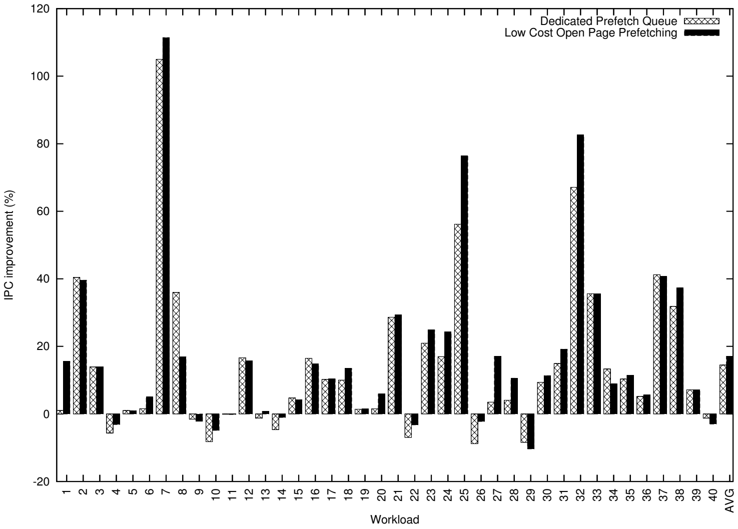


Figure IV.4: Speedup in IPC relative to no prefetching using a FR-FCFS memory controller.

IV.6.2 Importance of Coverage

In figure IV.5 we show the average relative performance increase by using other types of prefetchers, including Scheduled Region Prefetching, CZone/Delta Correlation and Reference Prediction Tables. Both C/DC and RPT prefetching have high accuracy. Because yhe prefetching accuracy is higher than the treshold in almost all workloads, our method degrades into the dedicated prefetch queue. In turn, the performance of our prefetch scheduling scheme is almost equal to the

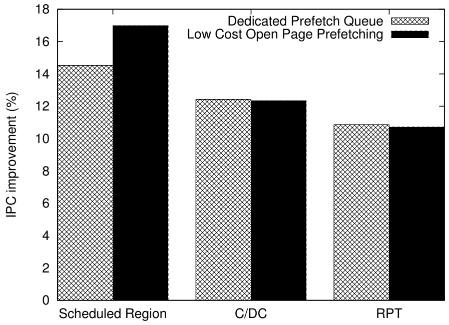


Figure IV.5: Average speedup in IPC relative to no prefetching.

dedicated prefetch queue scheme. However, this graph shows another interesting property. Scheduled Region Prefetching, which has a comparatively low accuracy, outperforms both of the more complex prefetcher heuristics. This is due to it having a much higher prefetch coverage. It provides more prefetches with *acceptable* accuracy, thus increasing performance.

IV.6.3 Insertion policy

In our scheme and the dedicated prefetch queue scheme there is a separate queue for handling prefetches. There are multiple possibilities on how to insert new prefetches into the queue. If the prefetch queue is full, then there are two possibilities, either discard the prefetch or insert the prefetch and evict the oldest prefetch. In figure IV.6 we show the performance of FIFO and LIFO policies with and without evictions. From this graph it is clear that evicting old data is beneficial, as well as using a LIFO policy. Evicting old prefetches is useful, because newer prefetches are based on newer demand reads, thus increasing both the accuracy and the probability that it can be piggybacked. The LIFO policy ensures that the newest prefetches are given priority over old ones. As shown in the graph, for both techniques, evicing old data is preferable, while a LIFO policy gives marginally better results over FIFO.

IV.6.4 Treshold parameter

In figure IV.7 we show the average speedup as a function of the required accuracy (treshold). In effect, setting the treshold to 0% makes the low cost open page prefetcher a dedicated queue prefetcher. Both RPT and C/DC prefetching have a very high accuracy, so the treshold doesn't affect performance until it becomes too high, effectively disabling prefetching, and in turn degrades performance. In addition, the peak for both sequential and scheduled region prefetching is relatively

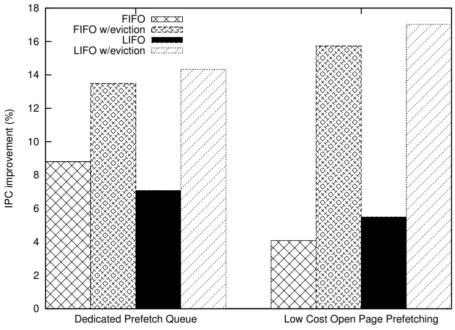


Figure IV.6: Effects of insertion policy on average IPC speedup.

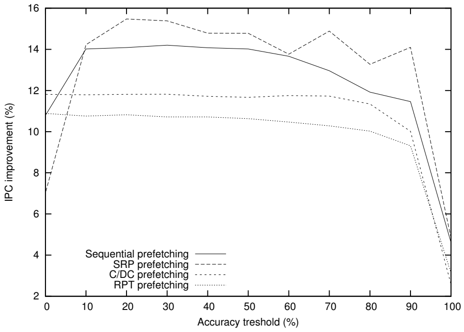


Figure IV.7: IPC improvement as a function of treshold

low (around 20-30 %). This further supports the observation that coverage is more important as long as accuracy is acceptable.

IV.6.5 Quality of Service

We have measured the maximum slowdown for any thread compared to the case where no prefetching is performed on each workload to get an indicator of the quality of service. Figure IV.8 shows the maximum performance degradation as a function of the number of workloads included. This graph shows three important properties. Firstly, 25% of the workloads experience no performance degradation on any thread when doing prefetching. Secondly, our scheme gives consistently higher quality of service. Using the other scheme 33% of the workloads show a thread getting a performance degradation of above 10%. In our scheme only 20% of the workloads show a thread getting more than 10% performance degradation. Finally, the maximum degradation for any thread for our scheme is only 36%, while the maximum for the dedicated prefetch queue approach is 49%.

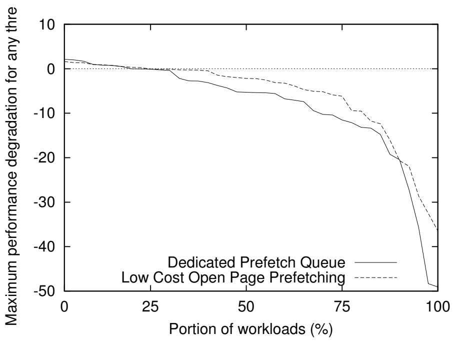


Figure IV.8: Maximum IPC degradation for any thread as a function of workloads.

IV.7 Discussion

Our results show that it is more important to have good prefetching coverage, while having *acceptable* accuracy. This is due to the relatively lower cost of piggybacked prefetches compared to isolated demand reads. Normally prefetch heuristics have been optimized for maximizing accuracy, so that the impact on bandwidth is as low as possible. This is due to the assumption that the cost of a single prefetch is about the same as a demand read. By carefully scheduling prefetches so that they are piggybacked on normal demand reads, this assumption no longer holds. We have demonstrated that a simpler, high coverage prefetcher outperforms more sophisticated high accuracy prefetchers in a bandwidth-constrained, 4-core chip multiprocessor system.

In our prefetch scheduling heuristic, we have used an accuracy estimator to control when prefetches should be issued. Other researchers have used such an estimator to control the aggressiveness of the prefetcher [21]. Such a technique can be used in conjunction with our scheduler. By using a feedback directed prefetcher, coverage can be increased while keeping accuracy at an acceptable level, thus providing higher performance.

Our simulator does not include a power model. However, our scheme piggybacks prefetches on demand reads. If a prefetch is successful then a later read is not needed, thus reducing the number of pages opened and closed, which in turn reduces power consumption in the DRAM module. Prefetching invariably increases bus traffic as some data transferred is not needed. Our scheme reduces the amount of useless traffic compared to other schemes by filtering out prefetches with low accuracy, thereby saving power.

Table IV.3: Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mgrid, perlbmk, parser	11	vpr, twolf, applu, eon	21	perlbmk, apsi, lucas, equake	31	mgrid, equake, vpr, eon
2	lucas, gcc, mcf, twolf	12	galgel, crafty, mgrid, swim	22	vpr, crafty, vpr, mcf	32	wupwise, gap, twolf, facerec
3	eon, eon, mesa, facerec	13	twolf, fma3d, galgel, vpr	23	gzip, equake, mgrid, mesa	33	galgel, equake, lucas, gzip
4	vortex1, ammp, equake, galgel	14	bzip, vpr, bzip, equake	24	facerec, applu, fma3d, lucas	34	facerec, gcc, facerec, apsi
5	gcc, galgel, apsi, crafty	15	galgel, crafty, vpr, swim	25	gap, applu, parser, facerec	35	mesa, mcf, swim, sixtrack
6	applu, equake, art, facerec	16	mcf, wupwise, mesa, mesa	26	mcf, apsi, twolf, ammp	36	mesa, sixtrack, equake, bzip
7	applu, gap, gcc, parser	17	applu, parser, apsi, perlbmk	27	swim, sixtrack, ammp, applu	37	mcf, gap, gcc, vortex1
8	gap, swim, twolf, mesa	18	mgrid, perlbmk, gzip, mgrid	28	art, fma3d, swim, parser	38	facerec, lucas, mcf, parser
9	sixtrack, fma3d, apsi, vortex1	19	mcf, sixtrack, gcc, apsi	29	apsi, gcc, vortex1, twolf	39	twolf, eon, mesa, eon
10	ammp, bzip, equake, parser	20	ammp, gcc, art, mesa	30	mgrid, gzip, apsi, equake	40	apsi, apsi, mcf, equake

IV.8 Conclusion

In this paper we have shown that by carefully scheduling prefetches so that they piggyback on ordinary demand reads, performance can be increased. This is done by exploiting the 3D structure of modern DRAM, where opening and closing pages is an expensive operation. As it becomes more important to issue prefetches that can be piggybacked on ordinary demand reads, emphasis shifts from high accuracy to high coverage with *acceptable* accuracy.

We have demonstrated our own prefetch scheme on a state of the art memory controller that exploits these findings. Our prefetch policy outperforms traditional scheduling policies in terms of performance, quality of service and power consumption.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. *SIGPLAN*

Not., 41(11):274–282, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1168918.1168892.

[3] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[4] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 222–233, 1999.

[5] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2004.

[6] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS '07: Proc. of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, 2007.

[7] *DDR2 SDRAM Specification.* JEDEC Solid State Technology Association, May 2006.

[8] W.-F. Lin, S. K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50 (11):1202–1218, 2001. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/12.966495.

[9] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[10] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[11] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI '04: Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 80–87, New York, NY, USA, 2004. ACM. ISBN 1-59593-040-X. doi: http://doi.acm.org/10.1145/1054943.1054954.

[12] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Micro, IEEE*, 25:90–97, Jan. 2005.

[13] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[14] K. J. Nesbit, N. Aggarwal, J. L., and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Proc. of the 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[15] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[16] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[17] J. Shao and B. Davis. A burst scheduling access reordering mechanism. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 285–294, 2007.

[18] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356887.356892.

[19] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. *SIGARCH Comput. Archit. News*, 34(2):252–263, 2006. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1150019.1136508.

[20] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[21] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

[22] V. Srinivasan, E. Davidson, and G. Tyson. A prefetch taxonomy. *Computers, IEEE Transactions on*, 53:126–140, Feb. 2004.

[23] Z. Zhu and Z. Zhang. A performance comparison of dram memory system optimizations for smt processors. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 213–224, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.2.

[24] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. *Eighth International Symposium on High-Performance Computer Architecture, 2002*, pages 107–116, 2002.

Paper V

Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables

Marius Grannæs, Magnus Jahre and Lasse Natvig

In *Data Prefetching Chamionship - 1*, 2009

Abstract

This paper presents a novel prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed techniques, RPT prefetching by Chen and Baer and PC/DC prefetching by Nesbit et al. It combines the storage-efficient table based design of Reference Prediction Tables (RPT) with the high performance delta correlating design of PC/DC. DCPT substantially reduces the complexity of PC/DC prefetching by avoiding expensive pointer chasing in the GHB and recomputation of the delta buffer.

We show that DCPT prefetching can increase performance by up to 3.7X for single benchmarks, while the geometric mean of speedups across all SPEC2006 benchmarks is 42% compared to no prefetching.

V.1 Introduction

The performance of general purpose microprocessors continue to increase at a rapid pace, but main memory has not been able to keep up [6]. In essence, the processor is able to process several orders of magnitude more data than main memory is able to deliver on time. Numerous techniques have been developed to tolerate or compensate for this gap, including out-of-order execution, caches and prefetching.

Prefetching predicts what data the processor will need in the future, and fetch that data from main memory before it is referenced. Most prefetching heuristics work by finding patterns in the memory access stream and use this knowledge to predict future accesses.

In this paper we present a new prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed prefetcher techniques, combining them and refining the ideas to achieve better performance. This heuristic provides a significant speedup (42% on average), while only needing 4KB of storage.

V.2 Previous Work

Many prefetching heuristics have been proposed in the past. The simplest is the *sequential prefetcher* [8], which simply fetches the next block when there is a miss in the cache. An improvement over this simple heuristic is the *tagged sequential prefetcher* which adds an extra bit per cache line (the tag). This bit is set when a block is prefetched into the cache. If there is a cache hit on a block where this bit is set, then the next cacheline is fetched.

Perez et al. [7] did a comparative survey in 2004 of many proposed prefetching heuristics and found that tagged sequential prefetching, reference prediction tables (RPT) and Program Counter/Delta Correlation Prefetching (PC/DC) were the top performers.

V.2.1 Reference Prediction Tables

Reference Prediction Tables is a strided prefetching heuristic originally proposed by Chen and Baer in 1995 [1]. Although improvements to the original design have been proposed [2], the basic design is the same.

As the name implies, RPT prefetching is a large table indexed by the address of the load which caused the miss. Each table entry has the format shown in figure V.1.

The first time a load instruction causes a miss, a table entry is reserved, possibly evicting the table entry for an older load instruction. The miss address is then

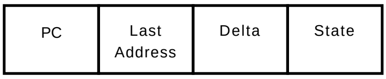


Figure V.1: Format of a Reference Prediction Table entry.

recorded in the *last address* field and the *state* is set to initial. The next time this instruction causes a miss, *last address* is subtracted from the current miss address and the result is stored in the *delta* (stride) field. *Last address* is then updated with the new miss address. The entry is now in the training state. The third time the load instruction misses a new delta is computed. If this delta matches the one stored in the entry, then there is a strided access pattern. The prefetcher then uses the delta to calculate which cache block(s) to prefetch.

V.2.2 PC/DC Prefetching

In 2004, Nesbit et al. [5] proposed a different approach using a Global History Buffer (GHB). The structure of the GHB is shown in figure V.2.

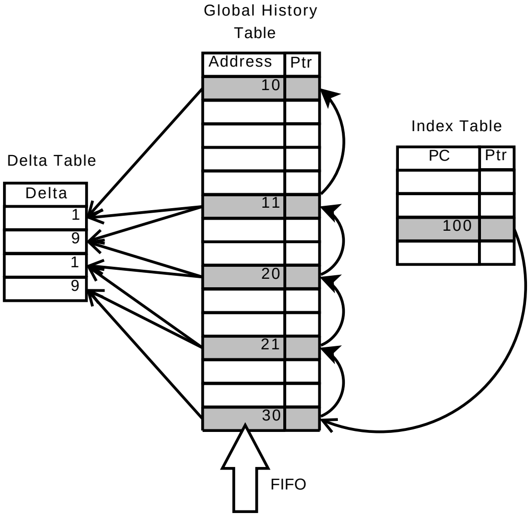


Figure V.2: Example of a Global History Buffer.

Each cache miss or cache hit to a tagged (prefetched) cache block is inserted into the GHB in FIFO order. The index table stores the address of the load instruction and a pointer into the GHB to the last miss issued by that instruction. Each entry in the GHB has a similar pointer, which points to the next miss issued by the same instruction. By traversing the pointers, the history of the latest misses issued by a certain instruction can be obtained.

PC/DC prefetching calculates the deltas between successive cache misses and stores them in a delta-buffer. The history in figure V.2 yields the following address stream and corresponding deltas:

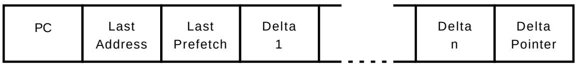
Address:	10		11		20		21		30
Deltas:		1		9		1		9	

Table V.1: Example delta stream.

The last pair of deltas is (1,9). By searching the delta-stream (correlating), we find this same pair in the beginning. A pattern is found, and prefetching can begin. The deltas after the pair are then added to the current miss address, and prefetches are issued for the calculated addresses.

V.3 Delta Correlating Prediction Tables

Our prefetch heuristic combines the approaches of both RPT and PC/DC prefetching by using a table based approach to delta correlation. In DCPT we use a large table indexed by the address (PC) of the load. Each entry has the format shown in figure V.3.



PC	Last Address	Last Prefetch	Delta 1		....	Delta n	Delta Pointer

Figure V.3: Format of a Delta Correlating Prediction Table Entry.

The *last address* field works in a similar manner as in RPT prefetching. Each delta is initially set to 0 and the delta pointer points to the first delta. The n delta fields acts as a circular buffer, holding the last n deltas observed by this load instruction and the *delta pointer* points to the head of this circular buffer. The buffer is only updated if the delta is non-zero. Each delta is stored as a n bit value. If the value cannot be represented with only n bits, a 0 is stored in the delta buffer as an indicator of an overflow error.

After updating the circular buffer, the deltas are traversed in reverse order, looking for a match to the two most recently inserted deltas. If a match is found the next stage begins. The first prefetch candidate is generated by adding the delta after the match to the value found in *last address*. The next prefetch candidate is generated by adding the next delta to the previous prefetch candidate. These candidates are stored in a temporary buffer. This process is repeated for each of the deltas after the matched pair including the newly inserted deltas. If a prefetch candidate matches the value stored in *last prefetch*, the content of the prefetch candidate buffer up to this point is discarded.

In the example in table V.1 the last pair of deltas is (1,9). Searching from the left, we find this pattern at the beginning. The first delta after the pattern is 1.

This delta is then added to the last address (30), producing a prefetch request for address 31. The next delta is 9, adding 9 to 31 yields 40, producing a prefetch request for address 40.

After computing the prefetch candidate buffer, every prefetch candidate is looked up in the cache to see if it is already present. If it is not present, then it is checked against the miss status holding registers to see if a demand request for the same line has already been issued. Third, the candidate is checked against a buffer that holds other prefetch request that have not been completed. This buffer can only hold 32 prefetches, if it is full, then the prefetch is discarded. Finally, the *last prefetch* field is updated with the address of the issued prefetch.

V.4 Methodology

To evaluate the performance of our prefetcher, we have used the SPEC2006 [9] benchmarks with the CMP$im simulator [4]. Each benchmark was fast forwarded 40 billion instructions and then a memory trace of the next 100 million instructions was recorded.

The simulated processor is a 15 stage, 4-wide OoO processor with a 128 entry instruction window with perfect branch prediction in accordance with competion rules [3]. A maximum of two loads and one store can be issued per clock cycle. The L1 is a 32KB 8-way set associative cache with a latency of 1 cycle. In this paper we use either a 512KB or a 2MB L2 cache, both 16 way set associative with a 20 cycle latency. Main memory has a 200 cycle latency.

The tagged sequential prefetcher was configured with a prefetching degree of 5, and a distance of 4. The RPT prefetcher has a 256 entry table, a prefetching degree of 16 and a distance of 4. To keep within the 32 Kbit limit set by the competition [3], the PC/DC prefetcher has a 702 entry GHB and a 32 entry delta buffer. Our prefetcher was set up with a 98 entry table with 19 12-bit deltas. These parameters were found experimentally to maximize performance on each prefetcher.

V.5 Results

In figure V.4, we compare the performance of the 4 prefetchers relative to no prefetching in a system with unlimited bandwidth. Prefetching has very little impact on performance ($< 2\%$) for the benchmarks *perlbench*, *gcc*, *gobmk*, *sjeng*, *gamess*, *namd*, *dealII*, *povray* and *tonto* and are not shown to conserve space. Furthermore, the results have been split into two graphs so that the benchmarks showing large speedups does not dwarf the others and the geometric mean of speedups.

Although DCPT and PC/DC prefetching share the same underlying pattern recognition engine, DCPT is able to capture more of the potential due to a more space-

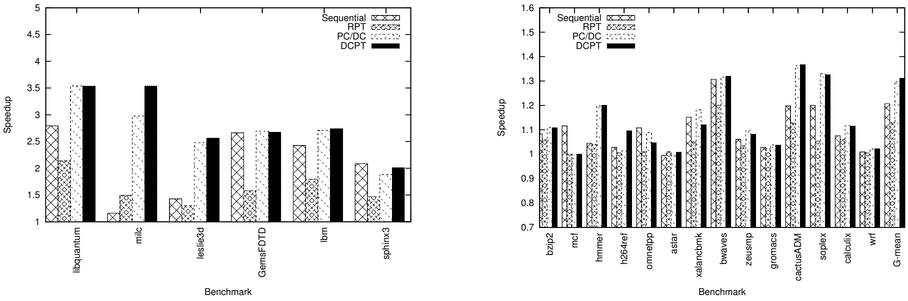


Figure V.4: Speedup compared to no prefetching. 2 MB L2 cache with unlimited bandwidth.

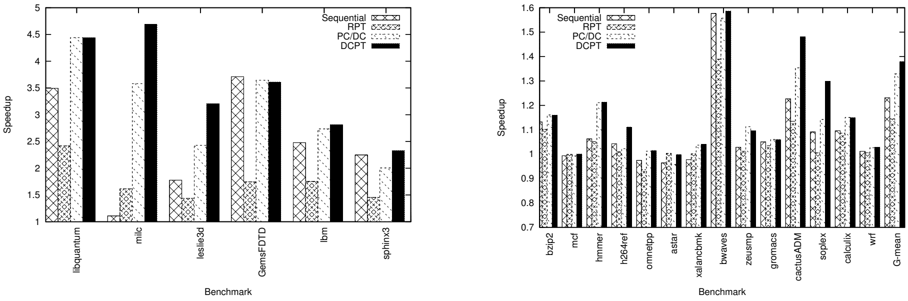


Figure V.5: Speedup compared to no prefetching. 2 MB L2 cache with limited bandwidth.

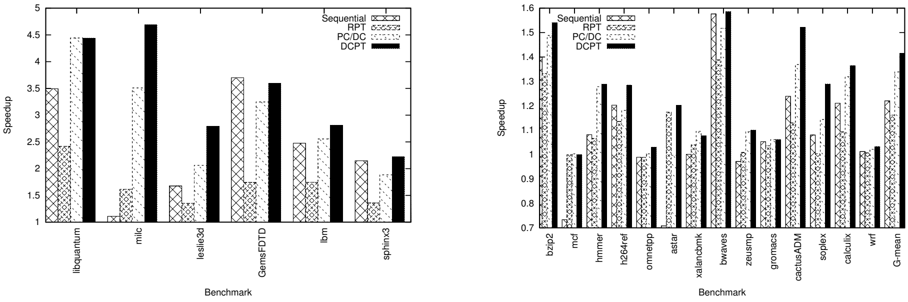


Figure V.6: Speedup compared to no prefetching. 512KB L2 cache with limited bandwidth.

efficient implementation. Because there is no penalty for issuing several prefetches, sequential prefetching performs quite well on several benchmarks but is unable to capture the patterns observed in *milc* and *leslie3d*. Overall, DCPT prefetching acheives a geometric mean of speedups of 1.31, while PC/DC achieves 1.29.

Our second experiment, shown in figure V.5, limits the bandwidth to one request per 10 clock cycles. In this configuration there is a more significant performance difference between DCPT (1.38 geometric mean speedup) and PC/DC (1.32 geometric mean speedup), even though there are several benchmarks where prefetching has no effect. Again there is a marked difference between DCPT and PC/DC in both *milc* and *leslie3d*.

In figure V.6 we reduce the size of the L2 cache to 512KB. In this case, sequential prefetching causes a severe slowdown on *mcf* and *astar*. However, it is also the top performer on *GemsFDTD*. Again, DCPT outperforms the other prefetchers.

Surprisingly, RPT prefetching does not perform very well. This is mainly due to it being to conservative with respect to bandwidth and at the same time not being able to detect the same access patterns as PC/DC and DCPT. In this configuration, DCPT achieves a geometric mean speedup of 1.42 vs 1.33 for PC/DC.

V.5.1 DCPT Parameters

One of the main differences between DCPT and PC/DC is that DCPT stores deltas, while PC/DC stores entire addresses in its GHB. Because the deltas are usually quite small, fewer bits are needed to represent a delta than a full address. In figure V.7 we show the average portion of deltas that can be represented with a given amount of bits across all SPEC2006 benchmarks. Additionally, the geometric mean of speedups is plotted as a function of the number of bits used per delta. In this experiment we have used a 256 entry DCPT prefetcher with 16 deltas per entry.

Although the coverage steadily increases with the amount of bits used, speedup has a distinct knee at around 7 bits. Thus, high deltas are not useful for prefetching.

In figure V.8 we show the geometric mean of speedups as a function of the number of deltas per table entry. In this experiment we used 16 bits deltas and 256 table entries. In effect, increasing the number of deltas increases the prefetch distance of DCPT, thus the optimal choice will be both processor and program dependant. However, there is a clear trend that performance flattens after about 14 deltas per table entry.

Finally, in figure V.9 we show the geometric mean of speedups as a function of the number of table entries. There is a steady performance improvement up to about 100 table entries. After this point, there is virtually no gain in adding extra entries.

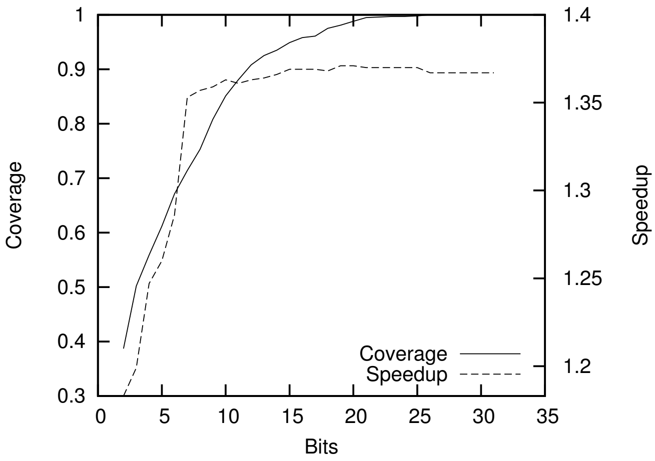


Figure V.7: Coverage and speedup as a function of the number of bits used to represent a delta.

V.6 Discussion

Our proposed prefetching technique is quite memory efficient. However, the complexity of calculating each prefetch is high. Each calculation involves searching the entire length of the deltas for possible matches and then adding the remaining deltas. Thus, each calculation has a fixed latency as each delta is either used in a comparison or an addition which lends itself well to pipelining.

Because of the relative infrequency of L2 misses several design points are available depending on the needed performance and available power and area. At one end of the spectrum, a single comparator and a single adder is sufficient to implement DCPT in addition to the memory storage. At the other end, the pattern matching step can be performed in a single cycle, provided enough comparators, while the additions can be performed by several pipelined adders.

In all our experiments, we unrealistically assumed that the calculation would not take any time to perform. We experimented with increasing the delay of the calculation from 1 to 100 clock cycles, and found there was only a minor ($< 1\%$) performance impact in the same configuration. However, this penalty can be offset by increasing the number of deltas per entry - indirectly increasing the prefetch distance and thus timeliness.

In most cases, the patterns observed are quite simple, as they often repeat themselves after only a few deltas. We did some initial experimentation with storing fewer deltas per entry and extrapolate the pattern from those deltas. However, there was little to gain from this technique, and we chose to eliminate it from the final design to keep it simpler.

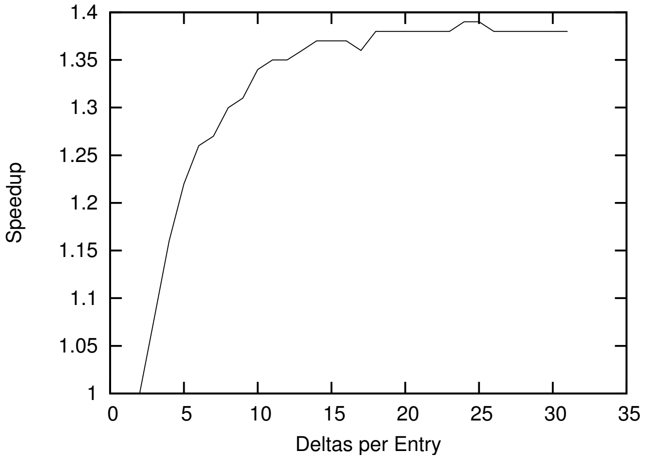


Figure V.8: Speedup vs. the number of deltas per entry.

Furthermore, because most memory access patterns are relatively stable, the last prefetch candidate is often the only one that is not filtered out by the *last prefetch* entry. This observation can be exploited by only calculating the last possible prefetch candidate.

In our experiments we used n bits to represents the delta range, representing the values between 2^{n-1} and -2^{n-1}. We observed more positive deltas than negative deltas, leading us to believe that adding a bias to the delta might be beneficial. We did not explore this further as the maximum potential of this technique would be equal to adding a single extra bit to each delta.

V.7 Conclusion

In this paper we have presented a new prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed techniques, Reference Prediction Tables by Chen and Baer [1] and PC/DC prefetching by Nesbit et al. [5]. It combines the table based design of RPT and the delta correlating design of PC/DC, as well as improving upon the ideas.

We show that DCPT prefetching can increase performance by up to 3.7X, while the average speedup across all benchmarks is 42%. This is an improvement over PC/DC prefetching by 27.2%.

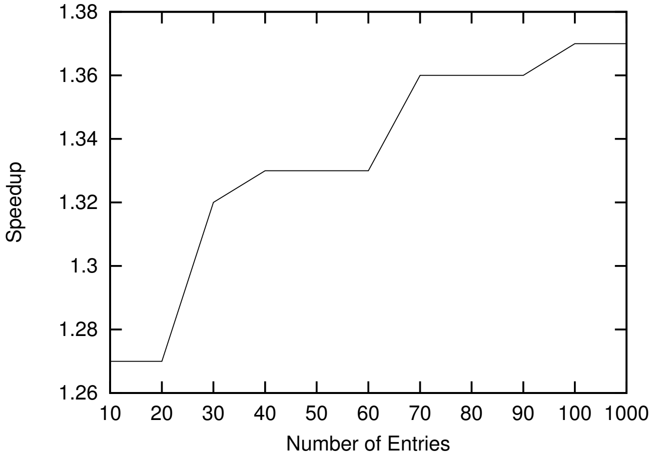


Figure V.9: Speedup vs table size.

Bibliography

[1] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[2] F. Dahlgren and P. Stenstrom. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(4):385–398, Apr. 1996.

[3] DPC-1. Data prefetching championship rules. URL `http://www.jilp.org/dpc/framework.html`.

[4] A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS*, 2008.

[5] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *High-Performance Computer Architecture, International Symposium on*, 0:96, 2004. ISSN 1530-0897. doi: http://doi.ieeecomputersociety.org/10.1109/HPCA.2004.10030.

[6] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1022594.1022596.

[7] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.25.

[8] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356887.356892.

[9] SPEC. Spec 2006 benchmark suites, 2006. `http://www.spec.org`.

Paper VI

A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures

Magnus Jahre, Marius Grannæs and Lasse Natvig
in *11th IEEE International Conference on High Performance Computing and Communications*, 2009

Abstract

The potential for destructive interference between running processes is increased as Chip Multiprocessors (CMPs) share more on-chip resources. We believe that understanding the nature of memory system interference is vital to achieve good fairness/complexity/performance trade-offs in CMPs. Our goal in this work is to quantify the latency penalties due to interference in all hardware-controlled, shared units (i.e. the on-chip interconnect, shared cache and memory bus). To achieve this, we simulate a wide variety of realistic CMP architectures. In particular, we vary the number of cores, interconnect topology, shared cache size and off-chip memory bandwidth. We observe that interference in the off-chip memory bus accounts for between 63% and 87% of the total interference impact while the impact of cache capacity interference can be lower than indicated by previous studies (between 5% and 32% of the total impact). In addition, as much as 11% of the total impact can be due to uncontrolled allocation of shared cache Miss Status Holding Registers (MSHRs).

VI.1 Introduction

Chip Multiprocessors (CMPs) or multi-core architectures are the prevalent architecture for modern general-purpose, high-performance processors. In these architectures, it is common to share some part of the hardware-controlled memory system between cores. When multiple processes are run concurrently, the presence of shared resources makes destructive interference possible. In addition, the on-chip shared resources are managed by simple hardware policies that are unaware that the requests belong to different processes. The performance effects caused by destructive interference are hard to predict since they are a consequence of the runtime interaction between the memory request streams from co-scheduled processes. Consequently, destructive interference is an undesirable property and a considerable research effort has been aimed at developing techniques that reduce its performance impact [3, 16].

Figure VI.1 illustrates that the current CMP memory systems are unable to provide predictable performance. To evaluate interference, we use a baseline configuration called the *private mode* where the benchmark is run in one of the processing cores while the remaining cores are idle. Consequently, it has exclusive access to all shared resources. Conversely, all benchmarks in a workload are run concurrently and compete for access to the shared resources in the *shared mode*. Figure VI.1 shows the private- and shared mode IPCs of all benchmarks in two of our 40 randomly generated workloads. These measurements are taken from the 4-core, crossbar-based architecture with 4 memory channels which is the architectural configuration with the lowest amount of interference of the configurations used in this work. In workload 17, *facerec* and *mgrid* are heavily impacted by interference with a performance reduction of 46% and 21%, respectively. However, the performance of *mcf* is only reduced by 1%. This illustrates that the performance impact of interference can be substantial and that it does not affect all running processes equally. Furthermore, the performance impact of interference is unpredictable since *facerec* is only slowed down by 7% in workload 13. Since these effects are clearly undesirable, there is a need for architectural techniques that provide predictable performance and improve fairness.

Previously, cache capacity interference has received a great deal of attention [3, 6, 9, 12, 15, 21] while only a few researchers have proposed techniques that reduce memory bus interference [16, 17, 19]. Furthermore, there has been little interest in the details of designing a complete, thread-aware memory system [2, 10, 18]. A first step towards a unified approach to reducing interference in the hardware-managed memory system is to develop an understanding of the problem. For instance, we found that memory bus interference accounts for 64% of the total amount of interference while cache capacity interference only accounts for 25% with a powerful 4-channel memory bus in our 4-core crossbar-based CMP. When the complexity of current fair cache sharing techniques is taken into account, the fairness requirements on the system must be strict for thread-aware cache techniques to be worth the cost.

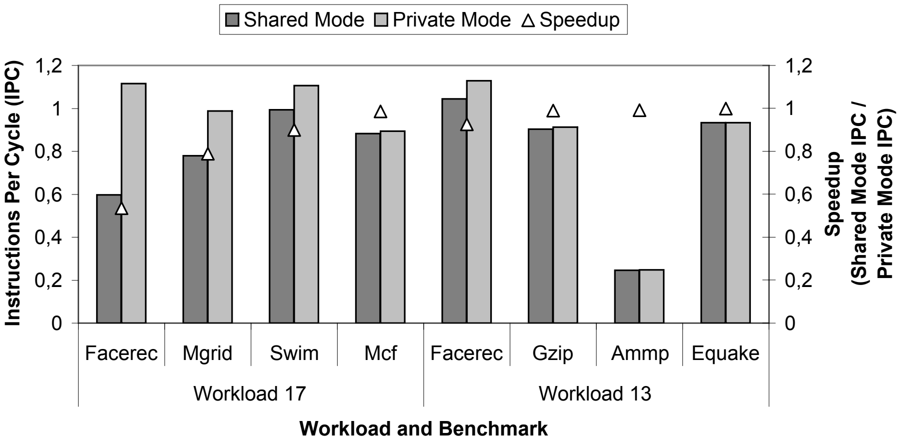


Figure VI.1: Performance Impact of Interference in the 4-core, Crossbar-Based CMP with 4 Memory Channels

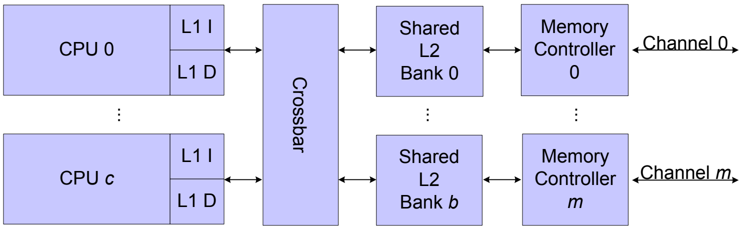


Figure VI.2: Crossbar-based CMP

In this work, we aim to increase the understanding of the interference problem and thus help architects achieve good complexity/fairness trade-offs. This understanding is developed through detailed analysis of interference at the memory request level. Consequently, we are able to analyze both the relative interference impact of the different shared units as well as the distribution of interference penalties. Handling memory bus interference yields the largest gain, and we believe that employing a fairness-aware technique here will be sufficient for many architectures and usage scenarios. However, we have also observed interference due to shared cache Miss Status Holding Register (MSHR) allocation which must be handled if the fairness requirements are sufficiently strict. Finally, we show that the main driver of memory system interference is insufficient memory bus bandwidth. Since this parameter is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board, it is likely that thread-aware memory

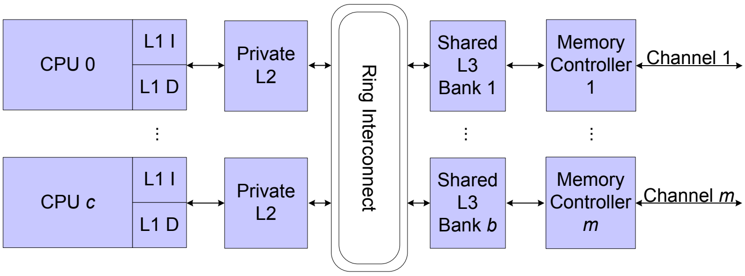


Figure VI.3: Ring-based CMP

bus schedulers will become a necessity in the near future.

VI.2 Related Work

It is common to aim an interference reduction technique at providing fairness and/or Quality of Service (QoS). A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [12]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when it is co-scheduled with any other process [3]. Furthermore, the allowed slow-down can depend on the priority of the process. In other words, the objective of fairness techniques is not to remove interference completely but to equalize its impact on all running processes.

There has been a considerable amount of research on how the performance impact from interference can be reduced in the hardware-controlled, shared memory system. However, most of these studies have focused on a single component of the entire system. For example, techniques have been proposed to reduce cache capacity interference [3, 6, 9, 12, 15, 21], cache bandwidth interference [20] and memory bus transfer interference [16, 17, 19]. Unfortunately, a technique that reduces interference in one component is not adequate to provide interference control for the complete memory system. Consequently, a few researchers have investigated how a chip-wide resource management technique can be designed. Iyer et al. [10] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [18] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. In addition, Bitirgen et al. [2] showed how machine learning can be applied to the resource allocation problem. The focus of these works has been to partition all shared resources amongst processes according to some allocation policy. In this work, we investigate the impact of interference and provide guidance on how trade-offs can be handled in resource allocation implementations.

VI.3 Methodology

VI.3.1 Chip Multiprocessor Architectures

There is still considerable debate regarding the high-level organization of CMPs [7, 13, 23]. Therefore, we use two different CMP architectures that are similar to current general-purpose, high-performance CMP implementations for our interference investigations. Furthermore, we scale these architectures according to the expected improvements in process technology [8]. The first CMP type uses a crossbar interconnect to connect the private L1 caches to a large, shared L2 cache as shown in Figure VI.2. Unfortunately, the crossbar does not scale in terms of area [14]. Consequently, we also use a different CMP model where a bi-directional ring is used as the interconnect. Since the ring has lower bandwidth than the crossbar, we add a private L2 cache to each processor to reduce the number of accesses to the interconnect. This is reasonable since the ring uses considerably less area than the crossbar. Furthermore, the number of processing cores and memory bus channels can be configured in both processor models which makes it possible to investigate the impact of memory system interference across a wide range of realistic CMP architectures. For convenience, we will refer to these architectures by the tuple c-i-m where c is the number of cores, i is the interconnect and m is the number of memory bus channels.

VI.3.2 Measuring and Reporting Interference

To gather accurate interference measurements, it is convenient to compare to a baseline where interference does not occur [4]. In this work, we create such a baseline by letting the process run in one processing core and leaving the remaining cores idle. Consequently, the process has exclusive access to all shared resources and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for the shared resources in the *shared mode*. Mutlu and Moscibroda observed that memory system interference is related to the memory latencies in the shared and private modes with the formula: $interference\ penalty = shared\ mode\ latency - private\ mode\ latency$ [17].

In our CMP models, there are three shared units: the interconnect, the memory bus and the shared cache. To assess the interference impact of each of these units, we partition the memory request latency through the shared memory system as shown in Table VI.1. For the interconnect, we divide the latency into three types: entry, transfer and delivery. The interconnect has a finite entry queue. If this queue becomes full, the interconnect can not accept any more requests and the request is delayed in the private cache MSHR. We refer to this as *Interconnect Entry Interference* if it causes a different delay in the shared mode than in the private mode. Furthermore, the shared cache can block. In this case, all requests waiting behind a request for a blocked bank are delayed since reordering requests can cause starvation. We refer to interference arising from this situation as *Interconnect Delivery*

Table VI.1: Shared Memory System Latency Breakdown

Type	**Description**
Interconnect Entry	The number of cycles a request was kept in the private cache MSHR before it is accepted into a interconnect queue
Interconnect Transfer	The number of cycles spent in the interconnect queue plus the interconnect transfer latency
Interconnect Delivery	The number of cycles a request was delayed because a shared cache bank could not accept requests due to insufficient buffer space
Memory Bus Entry	The number of cycles a request was delayed in a shared cache MSHR before it was accepted into a memory controller queue
Memory Bus Transfer	The number of cycles a request spent in the memory controller queue plus the number of cycles used to retrieve the requested data from DRAM
Cache Capacity	The number of cycles used to service misses that would not occur if the process had exclusive access to the shared cache

Interference. Finally, *Interconnect Transfer Interference* is the difference between the shared mode and private mode latencies when there is no cache blocking.

In the memory bus, we divide the latency into two types: entry and transfer. Again, the entry delay is the number of cycles the request is kept in an MSHR before it is accepted into the memory bus queue. If this latency is different for the shared and private modes, we refer to it as *Memory Bus Entry Interference*. In addition, *Memory Bus Transfer Interference* is the difference between the memory bus queue latency plus service latency in the two modes. Since there is no buffer allocation in the shared cache on a response, the memory bus does not have a delivery latency.

Finally, competition for space in the shared cache can lead to *Cache Capacity Interference*. Unlike the interference types discussed above, cache capacity interference does not have a latency value directly associated with it. The key observation is that if a request experiences a bus transfer latency in the shared mode and no bus transfer latency in the private mode, we have a miss in the shared cache that would have been a hit if the process had the entire cache to itself. The extra latency caused by this event in our CMP models is the number of cycles used to service the request in the memory bus. Consequently, the latency penalty of cache capacity interference is the sum of the bus entry latency and the bus transfer latency of the request.

Figure VI.4 illustrates the two stage process of gathering interference measurements and aggregating them for a single architecture. In the first stage, we create a compact representation of the measured interference for each benchmark in all workloads and architectures. First, we record the latency of all shared mode memory requests and all private mode memory requests. For all shared mode requests, we find the corresponding private mode request and compute the interference penalties for all interference types. If there are more than one request for the same address, we assume that the requests occur in the same order in both the private and shared modes. Then, we create a histogram representation of the data by counting the number of requests that experience a certain interference penalty

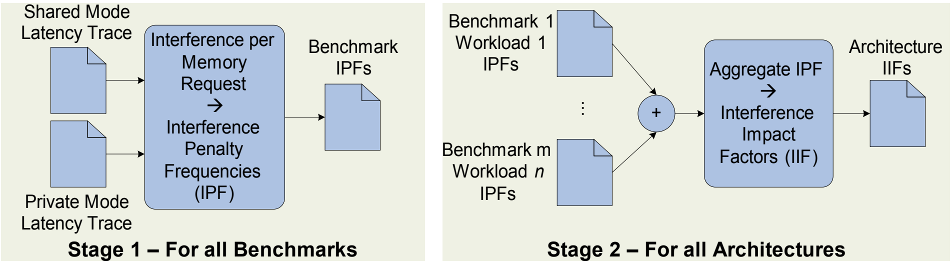

Shared Mode Latency Trace
Private Mode Latency Trace
Interference per Memory Request → Interference Penalty Frequencies (IPF)
Benchmark IPFs
Stage 1 – For all Benchmarks
Benchmark 1 Workload 1 IPFs
⋮
Benchmark m Workload n IPFs
+
Aggregate IPF → Interference Impact Factors (IIF)
Architecture IIFs
Stage 2 – For all Architectures

Figure VI.4: Interference Measurement Workflow

for each interference type. For example, if a request for memory address 15 experiences 12 cycles of interconnect transfer interference, we add 1 to the interconnect transfer interference entry at position 12. We refer to this data as the *Interference Penalty Frequency (IPF)*, and stage 1 of the analysis is complete when we have created IPF files for all workloads and architectures.

Stage 2 is the process of aggregating the per benchmark IPF files into one file for each architecture. First, we sum the request counts for each interference penalty from all files belonging to the architecture of interest. For some of the interference types, it is very common to not experience interference. These entries are of little interest and will dominate the results if we use plot the number of requests per interference penalty directly. Consequently, we devise a new metric called the *Interference Impact Factor (IIF)* that balances the latency penalty of interference against the probability of it arising (i.e. $IIF(i) = i \cdot P(i)$). For example, an experiment that results in 15 requests with 3 cycles interconnect transfer interference and 100 requests in total gives $IIF(3) = 3 \cdot \frac{15}{100}$. When we have computed the IIFs for all interference penalties, stage 2 is finished. In most cases, there is a large range of possible interference values and there is a need to summarize the IIFs for a range of interference penalties into a single number. To do this, we use the *Aggregate Interference Impact Factor (AIIF)* which is simply the sum of the IIFs for all or a subset of the observed interference penalties (i.e. $AIIF(a,b) = \sum_{i=a}^{b} IIF(i)$).

VI.3.3 Processor Model Scaling

To investigate the impact of interference in multi-core architectures, it is important that reasonable parameters are used to scale the latency, bandwidth and capacity of the various units in the memory system. To this end, we have used the International Technology Roadmap for Semiconductors [8] to estimate scaling trends and CACTI 5.3 [25] to find reasonable caches for the multi-core architectures used in this work. Table VI.2 summarizes the main multi-core model parameters. With each improvement in feature size, we double the number of processing cores but use the same core implementation. Furthermore, we follow the ITRS expectation that

Table VI.2: Architecture Parameter Scaling

	Crossbar Based Architecture			**Ring Based Architecture**		
	4-core	**8-core**	**16-core**	**4-core**	**8-core**	**16-core**
ITRS Year of Production	2007	2010	2013	2007	2010	2013
Feature Size (nm)	65	45	32	65	45	32
Shared Cache Size (MB)	8	16	32	8	16	32
Memory Bus Channels	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4
Interconnect Latency (End-to-End/Per Hop)	8/-	16/-	30/-	-/4	-/4	-/8

Table VI.3: Cache Parameters (4-core/8-core/16-core)

	L1 Private Cache	**L2 Private Cache**	**L2/L3 Shared Cache**
Size	64KB	1 MB	8/16/32 MB
Associativity	2	4	16
Access Latency (cycles)	3/2/2	9/6/5	16/12/12
Cycle Time (cycles)	2	4/3/2	4
MSHRs / WB (per bank)	16MSHRs/4WB	16	16/32/64
Banks	1	1	4
Area (mm^2)	2.3/1.1/0.5	14.6/7.0/3.6	94.0/91.9/84.7

the interconnect transfer latency will roughly double with each technology generation. The only exception is the per hop latency of the 4-core ring architecture which we assume is limited by the cache cycle time. To account for this latency increase, we double the ring bandwidth across generations. Since the ITRS projections for off-chip bandwidth results in a large range of possible pin counts, we simulate all architectures with 1, 2 and 4 independent memory channels.

Table VI.3 contains the parameters of our scaled on-chip caches. Here, we choose to keep the percentage of the total chip area occupied by L2 and L3 caches in the ring-based CMP constant. We use the same shared cache for the crossbar based CMP, but here we only use two levels of caches. Consequently, we assume that the area made available by using a two level cache hierarchy is sufficient to implement a crossbar interconnect. To reduce the shared cache access time and increase the opportunity for cache access parallelism, we divide the shared cache into 4 banks.

VI.3.4 Simulation Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments. The processor architecture parameters for the simulated CMPs are shown in Table VI.4. Table VI.5 contains the interconnect and memory bus parameters, and the cache parameters are outlined in Table VI.3. We have extended M5 with crossbar and ring interconnects and a detailed DDR2-800 memory bus and DRAM model [11]. For the shared mode, we generated 40 different 4-core workloads (Table VI.6), 20 8-core workloads (Table VI.7) and 10 16-core workloads (Table VI.8) by picking benchmarks at random from the full SPEC CPU2000 benchmark suite [24]. The only requirement given to the random selection process

Table VI.4: Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	4 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table VI.5: Interconnect and DRAM Interface

Parameter	Value
Crossbar Interconnect	8/16/30 cycles end-to-end transfer latency, 32 entry request queue, Pipelined (2/4/6 pipe stages)
Ring Interconnect	4/4/8 cycles per hop transfer latency, 1/1/2 pipe stages per hop, 32 entry request queue, 1/2/2 request rings, 1 response ring
Point to Point Link	4/3/2 transfer latency, 32 entry request queue
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [22], Closed page policy

is that a benchmark can only appear once in each workload. These workloads are fast-forwarded for 1 billion clock cycles before we gather traces for 100 million clock cycles. For our interference measurement methodology to be accurate, it is critical to minimize the difference between the memory requests in the shared and private modes. To ensure this, we use static cache partitioning and an infinite bandwidth interconnect and memory bus during fast forwarding such that the simulation sample starts on a similar instruction in both modes. Furthermore, we run the shared mode experiments first and then retrieve the number of instructions the benchmark committed. Then, we run the private mode simulation for the exact same number of instructions.

Since our processor cores are out-of-order, we can get cache misses from wrong path instructions that only occur in either the private or shared mode. Secondly, the start and termination of the simulation sample is not perfectly synchronized between the two modes. Thirdly, our memory controller reorders requests to achieve high page hit rates which can affect the private cache access patterns and miss rates. For these reasons, there can be small differences between the private and shared mode memory request traces. We remove these differences by applying two preprocessing steps before analyzing the traces. Firstly, we remove the requests for addresses that only occur in the private or shared modes. Secondly, we remove the superfluous requests of the mode that has the most requests in the cases where there are a different number of requests for the same address in the shared and private modes. These steps result in the removal of 0.1% of the observed requests.

Table VI.6: Randomly Generated 4-core Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	mesa, twolf, art, vpr	9	crafty, twolf, bzip, perlbmk	17	mgrid, facerec, mcf, swim	25	twolf, crafty, bzip, art	33	swim, gap, vortex1, perlbmk
2	art, vortex1, applu, crafty	10	eon, twolf, galgel, crafty	18	equake, applu, eon, gzip	26	applu, gap, perlbmk, crafty	34	equake, twolf, bzip, galgel
3	gap, eon, art, wupwise	11	vortex1, eon, art, equake	19	galgel, mesa, gzip, gcc	27	galgel, facerec, eon, mesa	35	applu, eon, fma3d, vortex1
4	fma3d, applu, parser, swim	12	gzip, lucas, twolf, apsi	20	art, galgel, parser, eon	28	vpr, crafty, applu, vortex1	36	lucas, ammp, twolf, fma3d
5	mcf, swim, gzip, vortex1	13	facerec, ammp, gzip, equake	21	bzip, gzip, perlbmk, eon	29	twolf, vpr, swim, wupwise	37	eon, parser, bzip, mcf
6	swim, galgel, apsi, applu	14	swim, sixtrack, mgrid, vortex1	22	vpr, swim, apsi, gcc	30	parser, mesa, vortex1, gcc	38	vpr, vortex1, wupwise, applu
7	gzip, wupwise, eon, equake	15	sixtrack, fma3d, parser, mcf	23	art, applu, perlbmk, mesa	31	lucas, mgrid, sixtrack, gap	39	lucas, mgrid, swim, gzip
8	sixtrack, gcc, facerec, perlbmk	16	twolf, galgel, crafty, applu	24	facerec, eon, bzip, mesa	32	facerec, galgel, vpr, sixtrack	40	gzip, swim, eon, fma3d

VI.4 Results

Modern out-of-order processors and memory systems contain a substantial amount of logic dedicated to hiding memory latency. Since our interference measurement methodology is latency focused, it is necessary to verify that the observed interference result in an asymmetric performance reduction. To this end, we use the fairness metric of Gabor et al. [5]. This metric expresses the difference between the largest and smallest shared mode slowdowns for one workload and provides values in the range from 0 to 1 where 1 indicates that the slowdown is the same for all benchmarks. A value of 0 indicates that at least one benchmark is not making forward progress.

Figure VI.5 shows the distribution of fairness metric values for all 4-core CMPs used in this work. Here, we plot the lowest fairness value observed when a certain number of workloads are taken into account for the different CMP architectures. The main observation from Figure VI.5 is that many workloads have reasonably good fairness values. However, there are also workloads where interference leads to large

Table VI.7: Randomly Generated 8-core Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mcf, vpr, fma3d, equake, sixtrack, galgel, bzip	6	applu, mcf, perlbmk, parser, crafty, eon, galgel, fma3d	11	ammp, lucas, wupwise, eon, twolf, fma3d, gcc, equake	16	apsi, ammp, vortex1, vpr, gap, perlbmk, art, bzip
2	crafty, vortex1, facerec, ammp, bzip, parser, mcf, perlbmk	7	fma3d, gzip, lucas, perlbmk, bzip, apsi, crafty, gap	12	mcf, galgel, gap, gzip, swim, sixtrack, vpr, fma3d	17	gzip, art, equake, facerec, eon, apsi, gcc, wupwise
3	lucas, vpr, mesa, apsi, swim, art, gzip, twolf	8	swim, gzip, ammp, facerec, perlbmk, equake, gcc, apsi	13	mesa, fma3d, gap, lucas, wupwise, galgel, sixtrack, parser	18	perlbmk, gap, parser, swim, sixtrack, fma3d, lucas, vortex1
4	art, mcf, perlbmk, wupwise, ammp, applu, mesa, swim	9	gap, mcf, vpr, apsi, vortex1, lucas, parser, applu	14	bzip, mgrid, facerec, art, eon, swim, equake, apsi	19	lucas, mesa, apsi, fma3d, mcf, parser, crafty, gcc
5	eon, apsi, equake, vpr, fma3d, facerec, gcc, vortex1	10	mcf, sixtrack, vpr, swim, gzip, mgrid, ammp, lucas	15	swim, vpr, gap, facerec, twolf, sixtrack, mcf, crafty	20	gcc, perlbmk, sixtrack, parser, vortex1, eon, facerec, galgel

performance differences between the benchmarks (i.e. low fairness). This supports the claim that interference-aware techniques are necessary to reduce performance variability.

Figure VI.6 shows the interference results for all architectures examined in this work. The main observation is that memory bus transfer interference is the major interference contributor across all architectures. This trend is also visible in Figure VI.5. Cache capacity interference is the second most important source of interference, but its impact is considerably smaller than the impact of bus interference. In addition, there are architectures (e.g. 16-CB-4) where the impact of cache capacity interference is small. Finally, there is more interconnect transfer interference in the crossbar interconnect than in the ring for the 16-core CMP. This seemingly counter intuitive result is due to two factors. Firstly, the ring-based architecture has a private L2 cache that reduces the pressure on the interconnect. Secondly, we do not increase the number of banks in the shared cache which reduces the parallelism available in the crossbar.

Figure VI.7 shows the interference distribution for the 4-core CMP for both interconnects and all memory bus configurations used in this work. Here, the interference impact factors are aggregated into bins of size 300, and we remove all bins that have a AIIF value of less than 0.35 to improve readability. While Figure VI.6 showed that interference is reduced when more memory bus bandwidth is made available, Figure VI.7 illustrates that the interference distribution also changes significantly. For the bandwidth constrained architectures (e.g. Figure VI.7(a) and

Table VI.8: Randomly Generated 16-core Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks
1	lucas, art, ammp, bzip, sixtrack, vpr, gzip, fma3d, equake, gcc, vortex1, facerec, galgel, crafty, apsi, twolf	6	parser, mesa, bzip, vortex1, vpr, fma3d, gap, gcc, perlbmk, gzip, mcf, crafty, eon, equake, facerec, galgel
2	lucas, ammp, mgrid, bzip, swim, crafty, galgel, equake, vortex1, parser, vpr, eon, wupwise, gzip, twolf, mcf	7	gzip, sixtrack, gap, fma3d, eon, galgel, perlbmk, art, bzip, ammp, equake, lucas, parser, facerec, apsi, crafty
3	lucas, ammp, art, bzip, twolf, applu, facerec, apsi, mesa, eon, swim, galgel, gzip, crafty, gap, perlbmk	8	perlbmk, gzip, apsi, twolf, wupwise, gap, vpr, mgrid, galgel, facerec, gcc, eon, mcf, lucas, fma3d, ammp
4	crafty, twolf, mgrid, applu, wupwise, swim, parser, fma3d, mesa, perlbmk, facerec, gcc, lucas, vortex1, galgel, bzip	9	mgrid, art, facerec, gcc, vpr, gzip, parser, ammp, fma3d, galgel, crafty, applu, twolf, bzip, mcf, apsi
5	bzip, facerec, vortex1, ammp, gzip, swim, fma3d, equake, lucas, apsi, applu, vpr, perlbmk, sixtrack, mcf, mesa	10	apsi, swim, crafty, art, sixtrack, ammp, galgel, lucas, vortex1, gzip, perlbmk, vpr, gcc, mesa, gap, equake

VI.7(d)), the interference impact increases to a maximum before it decreases. In the 4-channel architectures (Figure VI.7(c) and VI.7(f)), the largest interference impact is in the 0 to 300 bin and the impact decreases rapidly. The interference impact of the low penalty bins is significantly higher for the 4-channel architectures but the total impact is lower because of the distribution's short tail.

Figure VI.7 illustrates that the cache capacity interference impact is heavily dependent on the amount of memory bus interference. The reason is that the cost of cache capacity interference is the memory bus service time of the additional requests. Furthermore, the impact from interconnect transfer interference is small across all architectures. Although this interference type occurs very frequently, the interference penalty is small which results in a low interference impact. In addition, there is some interconnect delivery interference in all architectures which is due to shared cache blocking. The impact from this type of interference is large enough that it most likely must be dealt with in architectures with strict QoS requirements.

There is also a considerable amount of constructive interference. With the 4-Ring-1 architecture (Figure VI.7(a)), constructive memory bus interference leads to a noticeable impact in the -1500 to -1200 cycles bin. This can be explained by taking into account that our memory controller allows some requests to skip past the queue to achieve higher page hit rates and better memory bus utilization [22]. For the interconnect transfer interference, the impact from constructive interference is much lower. In this case, the constructive interference is due to some benchmarks having significant interconnect delays when they have the memory bus to themselves. In the shared mode, memory bus interference reduces execution speed enough that the interconnect congestion disappears which results in lower transfer delays in the shared mode.

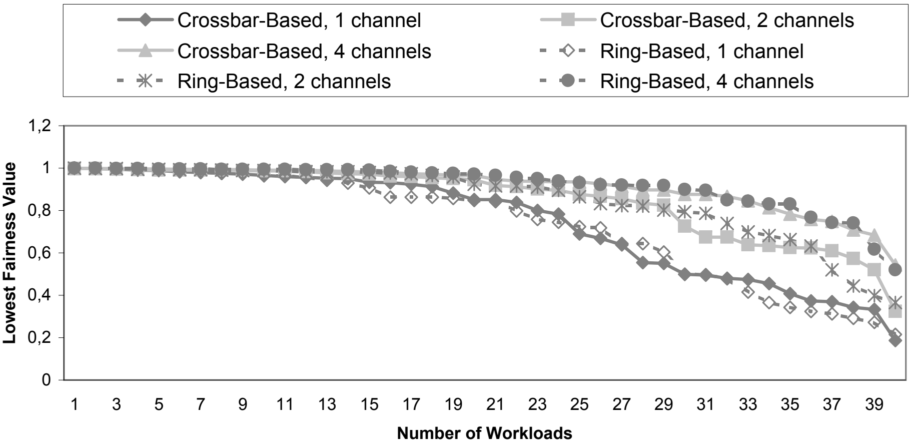


Figure VI.5: 4-core Fairness Metric Values

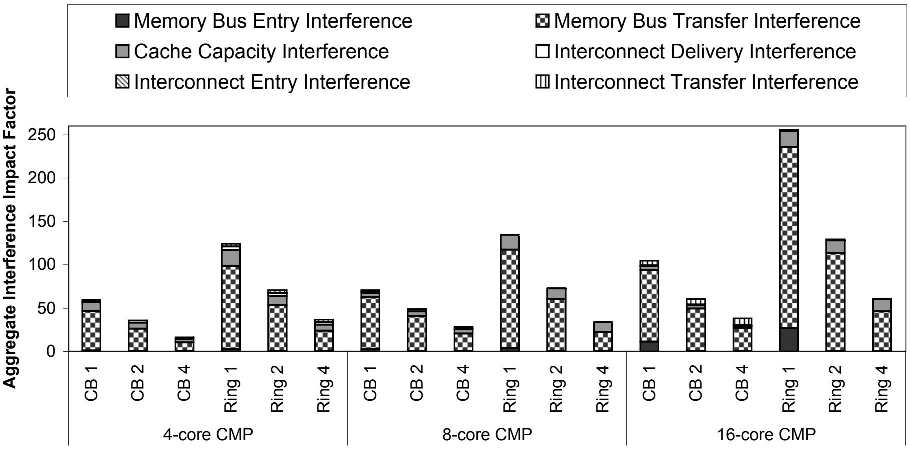
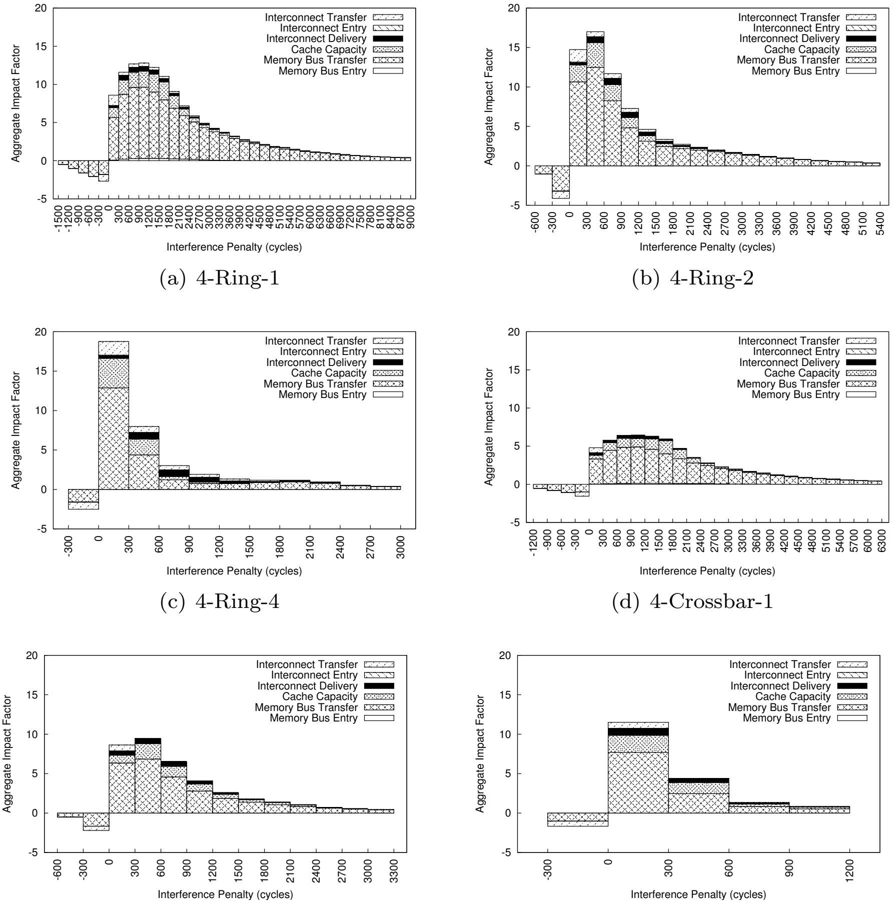


Figure VI.6: Interference Impact Breakdown



(a) 4-Ring-1

(b) 4-Ring-2

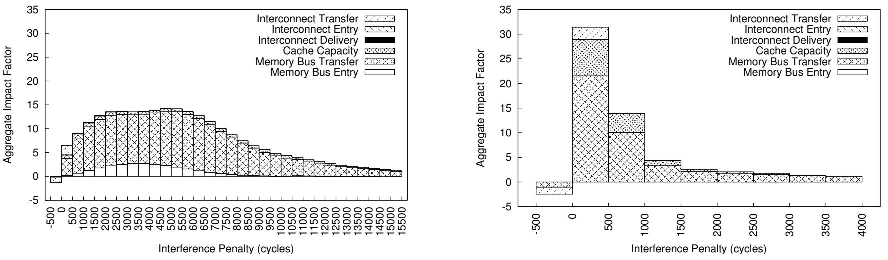
(c) 4-Ring-4

(d) 4-Crossbar-1

(e) 4-Crossbar-2

(f) 4-Crossbar-4

Figure VI.7: 4-core CMP Interference Impact (*cores-interconnect-channels*)



(a) 16-Ring-1

(b) 16-Ring-4

Figure VI.8: 16-core Ring Interference Impact

To illustrate the impact on interference by increasing the number of processing cores, we show the results of two 16-core ring-based architectures in Figure VI.8. Here, we use a bin size of 500 and only show bins that have an AIIF value of 1.0 or more. As expected, Figure VI.8(a) shows that there is a large amount of interference if the memory bus bandwidth is not scaled with the number of cores. Furthermore, memory bus entry interference has a considerable impact for this architecture. Consequently, a significant part of the interference is due to shared cache misses not being accepted into the memory bus queue because it is full. This further illustrates the need for fair buffer management observed in all 4-core architectures. Figure VI.8(b) shows the effect of increasing the number of memory bus channels to 4. Here, the distribution has a considerably shorter tail. However, the impact of the 0 to 500 cycle bin is large which indicates that low-penalty interference is frequent. In other words, providing more resources reduces the impact of interference but does not remove it. This indicates that fairness techniques are useful even when there are no severe performance bottlenecks.

VI.5 Conclusion and Further Work

In this work, we have shown that the impact of interference will increase as more cores are added to the chip by investigating a variety of realistic CMP architectures with 4, 8 and 16 cores. Consequently, techniques that reduce this interference are needed in future CMPs. We found that memory bus interference is the major source of interference and it is responsible for between 63% and 87% of the total interference impact depending on the architectures. Furthermore, it is unlikely that this situation will improve in the future as memory bus bandwidth is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board. We also observed that cache capacity interference can be a relatively small part of the total interference impact (between 5% and 32%). Consequently, adding a fair memory controller might be sufficient to achieve acceptable fairness and QoS for many near-term architectures. However, we have also observed architectures where 11% of the total interference impact is due to the shared cache MSHR allocation policy for which no solutions are currently known.

In this work, we have developed an understanding of memory system interference that can be useful for future research. However, we have only investigated CMPs where no fairness techniques have been implemented. A possible avenue of further work is to investigate how implementing fairness techniques in one shared unit will influence the interference impact of the other shared units. For instance, a cache capacity sharing technique might reduce the overall number of cache misses enough to reduce the impact of memory bus interference. On the other hand, it can potentially increase the number misses by limiting the cache space available to a process which might result in more memory bus interference. In addition, we observed that shared cache blocking and memory controller blocking can be important contributors to interference in certain architectures. One possible solution to this problem is to allocate MSHR entries and memory bus queue space per thread. However, this must be done carefully to ensure that the provided resources are utilized efficiently.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[5] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, pages 149–160, 2006.

[6] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[7] H. Hofstee. Power Efficient Processor Architecture and the Cell Processor. *HPCA 11: 11th Int. Symp. on High-Performance Comp. Arch.*, pages 258–262, 2005.

[8] ITRS. International Technology Roadmap for Semiconductors - 2007 Edition. `http://www.itrs.net/`.

[9] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07: Proc. of the 2007 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Sys.*, pages 25–36, 2007.

[11] *DDR2 SDRAM Specification.* JEDEC Solid State Technology Association, May 2006.

[12] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[14] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *ISCA '05: Proc. of the 32nd Int. Symp. on Comp. Arch.*, pages 408–419, 2005.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *HPCA '08: Proc. of the 13th Int. Symp. on High-Perf. Comp. Arch.*, 2008.

[16] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[17] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[18] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[19] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 208–222, 2006.

[20] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[21] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Proc. of the 27th An. Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008*, pages 1–15, 2008.

[24] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[25] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACI 5.1. Technical report, HP Laboratories Palo Alto, 2008.

Paper VII

Multi-Level Hardware Prefetching using Low Complexity Delta Correlating Prediction Tables with Partial Matching

Marius Grannæs, Magnus Jahre and Lasse Natvig

Accepted to the 5th HiPEAC Conference, 2010

Abstract

This paper presents a low complexity table-based approach to delta correlation prefetching. Our approach uses a table indexed by the load address which stores the latest deltas observed. By storing deltas rather than full miss addresses, considerable space is saved while making pattern matching easier. The delta-history can predict repeating patterns with long periods by using delta correlation. In addition, we propose L1 hoisting which is a technique for moving data from the L2 to the L1 using the same underlying table structure and partial matching which reduces the spatial resolution in the delta stream to expose more patterns.

We evaluate our prefetching technique using the simulator framework used in the Data Prefetching Championship. This allows us to use the original code submitted to the contest to fairly evaluate several alternate prefetching techniques. Our prefetcher technique increases performance by 87% on average (6.6X max) on SPEC2006.

VII.1 Introduction

In 2004, Gracia Perez et al. [13] published a paper that evaluated several prefetching techniques in a common framework. They found that several techniques were not as good as the original authors claimed. This discrepancy was due to researchers using different simulator infrastructure and benchmarks as well as the difficulty in implementing other techniques due to a lack of documentation. In this work, we avoid these problems by using the simulation infrastructure and original code from the first Data Prefetching Championship (DPC-1). This competition was similar to the earlier JILP Championship Branch Prediction Competition (CBP). In order to ensure a fair comparison of prefetcher performance, the organizers published a common simulator framework. Each prefetcher could use a maximum of 4KB of storage, but there was no limit on prefetcher complexity. Each contestant submitted their code to the competition for evaluation. This code was later published. This allows us to do a fair comparison with the top three DPC entries using their submitted code.

Our submission, Delta Correlating Prediction Tables (DCPT), used a table indexed by the PC of the load [7]. Each table entry stores a large amount of history per load instruction in the form of deltas. By storing deltas rather than full miss addresses, we save a significant amount of memory and make pattern matching easier. Pattern matching is done by using Delta Correlation, originally proposed by Nesbit et al. [11]. This technique is very effective at detecting patterns with periods shorter than the amount of history stored.

In this paper, we improve DCPT by proposing DCPT-P which incorporates many of the lessons learned during DPC-1. We introduce the concept of L1 hoisting, which is a highly accurate and timely method for moving data into the L1 cache. L1 hoisting does not require complex additions to the L1 cache which could interfere with the critical path of the processor. The key idea in L1 hoisting is to first issue prefetches to the L2 cache with a high prefetch distance, thus ensuring timeliness in the L2 cache. To further increase performance, we predict when the prefetched data will soon be used and hoist it to the L1 cache.

Second, we introduce partial matching which is a technique to enhance delta correlation in hard to predict cases such as pointer chasing. Partial matching reduces the spatial resolution in the delta stream to reveal more possibilities for prefetching. Thus, this technique increases coverage at the price of reduced accuracy, for an overall increase in performance.

VII.2 Previous Work

Because of the large gap between the latency of the processor and main memory, prefetching has a large potential for increasing processor performance. Therefore, it has been an active research topic for several decades. The simplest prefetcher is sequential (next line) prefetching, which simply fetches the next line whenever a cache line is accessed, thus exploiting spatial locality [15]. Its improvement, tagged sequential prefetching, uses an extra bit per cache line to indicate that this cache line was prefetched. When the processor subsequently hits in the cache on a cache block with this bit set, it fetches the next block.

Reference prediction tables use a table to store the recent history of a single load [1]. Each table entry is indexed by the address of the load and contains the last miss address as well as the delta (the difference between the address of the latest consecutive misses) as well as a state [2]. Then, on the next miss, the delta between the first miss address and the current is computed and stored in the table and the entry enters the training state. Finally, on the third miss, a new delta is computed. If that delta matches the one found in the table, the entry enters the prefetching state and prefetches are issued by using the computed delta.

The use of a Global History Buffer (GHB) was proposed by Nesbit et al. [11]. A GHB is essentially a FIFO containing the last misses observed by the memory system. Each entry in the GHB is linked to the previous entry of its class by a pointer. Because of the versatility of the GHB, a class can be defined in multiple ways such as belonging to the same memory region (C/DC) or originating from the same load (PC/DC) [12]. In PC/DC the entries in the GHB belong to the same class if they originate from the same load instruction.

By traversing the linked list, a miss history can be obtained for that load. This operation can be expensive in terms of energy and latency as the GHB structure is read multiple times to generate the miss history. In PC/DC, the deltas between consecutive misses are computed and stored in a delta table. This operation is repeated every time a L2 miss occurs. After the history of deltas are computed, delta correlation begins. Delta correlation means searching for the most recent pair of deltas in the delta history. If a corresponding pair is found in the delta history, the deltas after the match is used to predict future deltas.

During the first Data Prefetching Championship (DPC-1) several novel prefetcher designs were presented. Second place was awarded to GHB-LDB (Global History Buffer - Local Delta Buffer) which was proposed by Dimitrov et al. [3]. GHB-LDB improves upon the PC/DC prefetcher by also including global correlation (as opposed to the local correlation directed by the PC of the load) and most common stride prefetching. Furthermore, their prefetcher issues prefetches directly into the L1 cache.

Third place was awarded to Ramos et al. [14] for their multi-level prefetcher based on the PC/DC concept. Their PDFCM (Prefetching based on a Differential Finite Context Machine) prefetcher uses a hash-based approach with two tables. The History Table is indexed by the PC which contains a hashed representation of the recent history of that entry. This hash points to an entry in the Delta Table which contains the predicted delta. By computing new hashes based on the predicted deltas, an arbitrary prefetch degree and distance can be used.

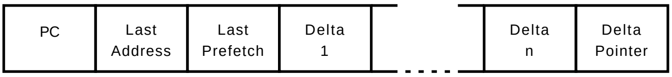


Figure VII.1: Format of a single DCPT-P entry.

Finally, the winner was the AMPM (Access Map Pattern Matching) prefetcher proposed by Ishii et al. [9]. Their prefetcher divides memory into hot zones similar to Czones [12]. Each hot zone is tracked by using a 2-bit vector for each cache line in that zone. This vector is then analyzed to see if there are any constant stride patterns in that zone. If there are any patterns, the predicted pattern is prefetched.

VII.3 Delta Correlating Prediction Tables

VII.3.1 Overview

The core of our prefetching heuristic is a table indexed by the PC of the load. Each entry has the format shown in Figure VII.1. In addition to the PC tag, each entry holds the last miss address, the address of the last prefetch that was issued in addition to a circular buffer containing the last n deltas. The circular buffer is managed by the *delta pointer*. This field points to the most recently added delta.

This organization has a number of advantages. Each entry holds a comparatively large history which can be used to predict any repeating pattern as long as the period is shorter than $n - 2$. In addition, entries do not compete for space, thus ensuring that the amount of history per entry is monotonically increasing, which reduces the risk that prefetches are issued for the same line. Finally, by storing deltas, rather than full miss addresses it is possible to save considerably memory space.

In Figure VII.2, we show the portion of deltas we observed that can be represented as a function of the number of bits used to represent each delta in the table. By far, the most common delta is one which is to be expected as this represents the common sequential pattern. As the number of bits per delta increases, the portion of the deltas we can represent increases monotonically.

Figure VII.2 also plots the performance impact of increasing the number of bits used to represent a single delta. Interestingly, the speedup has a much steeper slope than the coverage. Performance rises sharply as one increases the number of bits up to 12, and then trails off. Although more bits increases the information content, performance degrades because of false matches (high delta values are often generated by pointer chasing codes). Thus, performance can be improved and the memory footprint reduced by limiting the number of bits used.

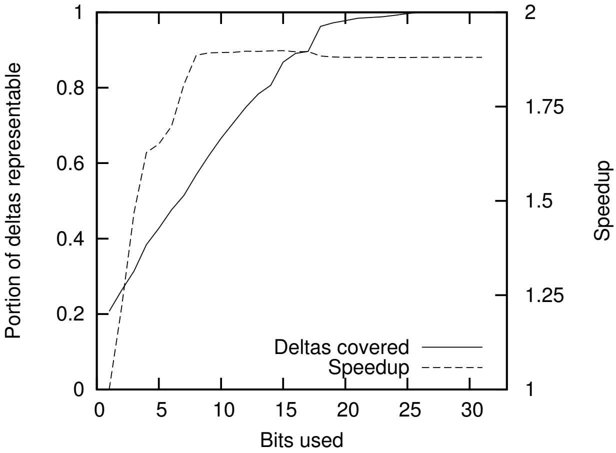


Figure VII.2: Impact of increasing the numbers of bits used to represent a delta.

VII.3.2 DCPT-P Implementation

A basic implementation of the DCPT-P pipeline is shown in Figure VII.4. When there is an access to the L2 cache the same request enters the pipeline. The first step is to look up the PC of the load in the table. If a corresponding entry is not found, an old entry is replaced using a LRU replacement policy. This new entry is initialized with the miss address and the rest of the entry is initialized to zero.

If a corresponding entry is found, we first compute the delta between the current access and the value stored in *last address*. If the delta is not zero, then the delta is stored in the circular buffer and the *delta pointer* and *last address* is updated. In our experiments, the L2 cache uses 128 byte cache blocks. To conserve space we mask out the lower six bits (64). Thus, a delta of two represents an increment of a single cache block. As shown by Hur et al. [8], many streams are short (2-4 cache lines). By using deltas that are smaller than a cache block we enable DCPT-P to start prefetching without waiting for too many misses to the L2. If we cannot represent a delta with the available bits, we store a zero instead (not valid). Finally, the entry is passed on to the pattern matching step.

The pattern matching logic is similar to the logic used in PC/DC [11]. In essence, we search for the first occurrence of the last pair of deltas in the circular stream. In Figure VII.3, we show the distribution of match locations in a 20 entry circular delta buffer. There are two peaks. The first peak is at the first possible position (the last two deltas in the circular buffer matches the first two deltas). This position represents constant strides or repeating patterns (for example 1-2-1-2-1-2). However, a match in the first possible position does not necessarily mean that the other stored deltas are redundant. Consider a blocking implementation of a matrix multiply. In this situation, the access pattern would be a series of sequential

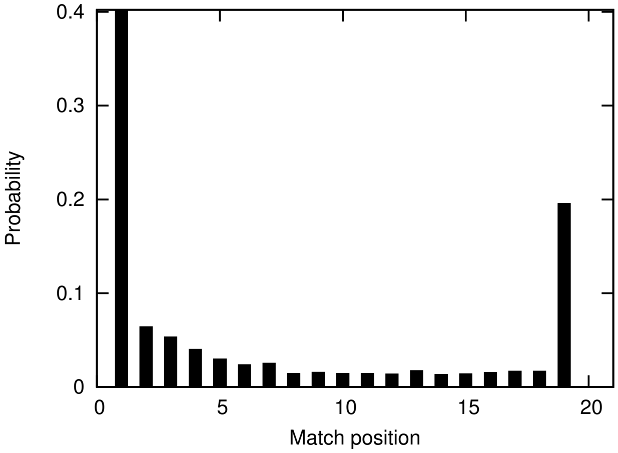


Figure VII.3: Position in the circular buffer where a match is found.

accesses followed by a large stride when the blocking algorithm moves to the next row, which in turn would be followed by a series of sequential accesses. By storing multiple deltas in this manner, this behaviour can be effectively captured by DCPT-P. The last peak (at 19) represent situations where the pattern is not found. This data point is included to illustrate the amount of times no pattern is found. Our implementation of the pattern matching step uses several comparators working in parallel in combination with a priority encoder as shown in Figure VII.5.

The next step is to generate prefetch candidates. The first prefetch candidate is generated by adding the first delta after the match to the current miss address. The second prefetch candidate is generated by adding the second delta after the match to the previous miss address. This is done for all deltas after the match. Thus, by increasing the number of deltas per table entry the prefetch distance is also increased.

Table VII.1: Example delta stream.

Address:	10		11		20		21		30
Deltas:		*1*		*9*		**1**		**9**	

As an example, consider the stream shown in Table VII.1. In this example, time increases to the right (i.e. the most recent address observed is 30). The last pair of deltas is thus (1,9) (Marked with **boldface**). We search for this pair of deltas and find the same pair of deltas in the beginning of the stream (Marked with *italics*). The next delta after this match is 1. We then add 1 to the last *last address* (30) and obtain 31. This is our first prefetch candidate. The next delta is 9. In a similar manner we add 9 to the previous prefetch candidate and obtain 40. We repeat this procedure for all the deltas in the circular buffer.

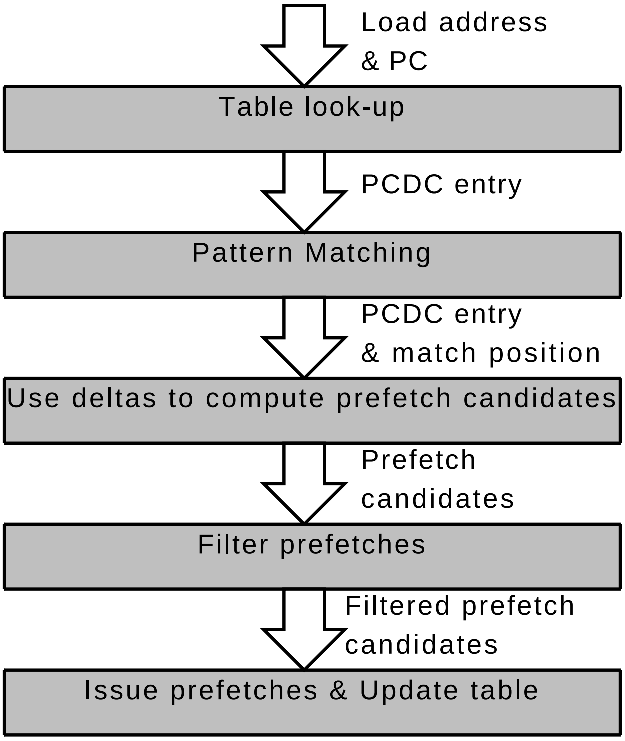


Figure VII.4: DCPT-P Pipeline

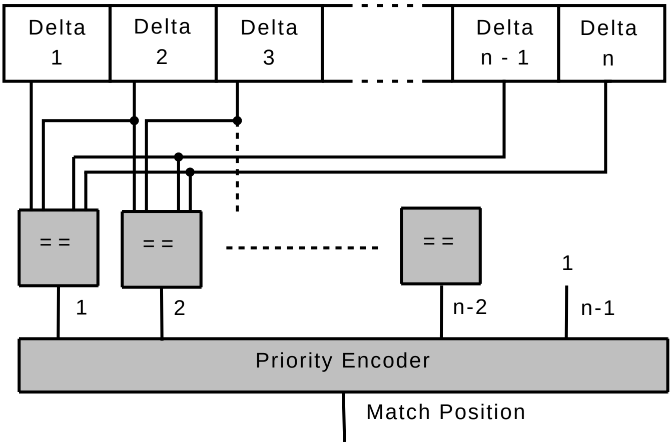


Figure VII.5: Pattern matching implementation.

This approach generates several redundant prefetches so prefetch filtering is needed. The most important mechanism is the *last prefetch* field in each entry. This entry keeps the address of the last prefetch issued by that entry. If a candidate is made that matches the *last prefetch* field during prefetch candidate generation, all previous prefetch candidates are dropped. In the steady state, this ensures that only a single prefetch is issued.

We use a 32 entry pending prefetch buffer to store the prefetches that have been issued. This table serves a dual purpose; first it is checked prior to issuing a prefetch request, thus eliminating redundant prefetches. Second, by only allowing 32 outstanding prefetch requests we limit the amount of bandwidth used by the prefetcher and the probability of severe bandwidth contention.

VII.3.3 L1 Hoisting

Although the greatest latency is from the last level cache to the main memory, there is a significant performance potential to prefetching into the L1 cache. However, due to its limited capacity, cache pollution becomes a significant problem. To avoid this, highly accurate and timely prefetches are needed. In addition, because the L1 cache is on the critical path it becomes much more difficult to construct large and complex prefetch heuristics that interact with the L1 access stream without degrading overall performance.

To overcome this problem we propose L1 hoisting. L1 hoisting is a natural addition to DCPT-P. DCPT-P is highly accurate, but issuing prefetches directly into the L1 cache brings the data in too early and displaces data that is currently needed, which in turn reduces overall performance. Our solution is prefetch hoisting. The first prefetch candidate that is generated is treated as a candidate for prefetch hoisting as well. This candidate is predicted to be the next required by the processor. In the steady state, this candidate has already been prefetched into the L2 by an earlier miss by the same load. Thus, we check if this block is present in the L2. If it is present, then the block is moved (hoisted) into the L1. Even though prefetch distance is low (only one block) it is enough to be timely, because the latency from the L1 cache to L2 cache is much lower than the latency from L1 to main memory,

VII.3.4 Partial Matching

DCPT captures most regular repeating patterns. However, many programs exhibit more complex and irregular patterns. Consider the code from soplex shown in Listing VII.1. Although the load in line 7 might seem hard to predict there is some structure to the addresses issued. One pattern of deltas we observed was *-2, -1, 4, -2, -3, -3, -1, 3.* In this case, there are no repeating pair of deltas, but most deltas are small. Because the observed deltas are so small, using previous deltas to issue new prefetches might be beneficial. Another pattern we observed was *9, 9, 9, 9,*

```
for (i = x.size(); i-- > 0; ++xi) {
   svec = const_cast<SVector*>(& A[*xi]);
   elem = &(svec->element(0));
   last = elem + svec->size();
   y = vl[*xi];
   for (; elem < last; ++elem)
      v[elem->idx] += y * elem->val;
}
```

Listing VII.1: Loop from 450.Soplex

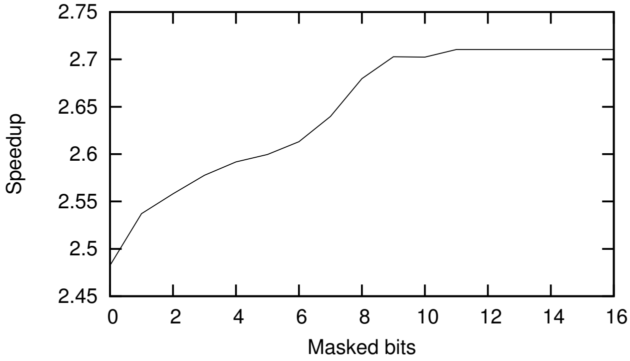


Figure VII.6: Speedup of Sphinx as a function of LSB masked in partial matching.

-54, 73, 9, 9, 9. In this case, a regular pattern is interrupted by an abrupt jump. Simply prefetching using the most common delta (9) would be preferable.

In this work, we propose a general approach to exposing such patterns called partial matching. If a pattern is not found using the exact match, we try partial matching. In essence, we reduce the spatial resolution by masking out the least significant bits and try to find a match using only the MSB's of the delta. This allows us to issue prefetches in both of the cases above.

In Figure VII.6, we show the speedup of the benchmark sphinx as a function of the number of LSB masked. Increasing the number of masked bits increases the number of prefetches issued. In the case of Sphinx, many of these prefetches are hits, but in other benchmarks increasing the number of masked bits increases the probability of cache pollution and wasted bandwidth.

VII.4 Methodology

Gracia Perez et al. [13] showed that the choice of simulator and benchmarks as well as the implementation of other data cache mechanisms can severely bias the results when evaluating prefetcher performance. Therefore, to evaluate our prefetcher proposal we have used the Data Prefetching Championship (DPC-1) simulator framework [4] as well as the code submitted by the contestants to the competition.

The simulator framework is based on the CMP$im simulator [10]. This framework models a simple 15 stage, 4 wide out-of-order core with a 128-entry instruction window. The core can issue a maximum of two loads and a single store each cycle. The framework models a two level cache hierarchy, consisting of a 32KB, 8-way L1 cache with 64B cache lines. The L2 is a 2MB 16-way set-associative cache with 128 Byte cache lines and a LRU replacement policy. The second level cache has a 20 cycle latency, while main memory has a 200-cycle latency. Each cache is coupled with a queue for storing outstanding requests to the next level in the hierarchy. These queues issues requests in FIFO order and does not prioritize demand requests over prefetch requests [4]. The queue to main memory issues one request per 10 clock cycles, while the queue to the L2 issues 1 per clock cycle. This simulator setup was referred to as configuration 2 in DPC-1.

For our experiments we have generated traces for the SPEC2006 [16] benchmark suite. Each benchmark was fast forwarded by 40 billion instructions and then executed for 100 million instructions. The benchmarks were compiled with the Intel C Compiler version 10.0.

To evaluate the performance of our prefetching heuristic we have selected 5 state-of-the-art prefetchers. In the study by Gracia Perez et al. [13] mentioned earlier, Reference Prediction Tables [1] and PC/DC using a GHB [11] were found to give the highest performance. Therefore, we have implemented these two approaches with the same 4KB limitation. In addition, we have selected the top three performers from DPC-1. The contestants' prefetching code was made public after the competition so we have used their code without modification. The top performers were AMPM [9], PDFCM (Maxperf) [14] and GHB-LDB [3].

To keep within the same 4KB limit imposed on the other prefetcher implementations we have used a 95 entry table with 20 12-bit deltas. On the pattern matching pass with partial matching we mask the low 8 bits of the delta. The pending prefetch buffer can hold a maximum of 32 requests.

VII.5 Results

We begin our evaluation by comparing the performance of our prefetcher to the top three DPC-1 prefetchers, Reference Prediction Tables and PC/DC with the SPEC2006 benchmark suite. The results are shown in Figure VII.7 and VII.8. In all of the results presented in this paper, speedup refers to a speedup compared

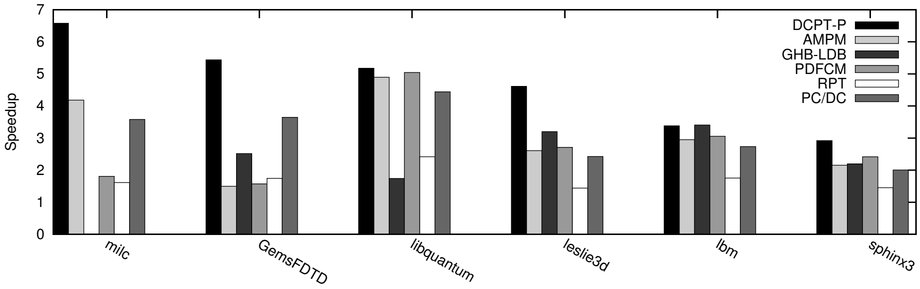


Figure VII.7: 2 MB L2 cache. Benchmarks with large speedups.

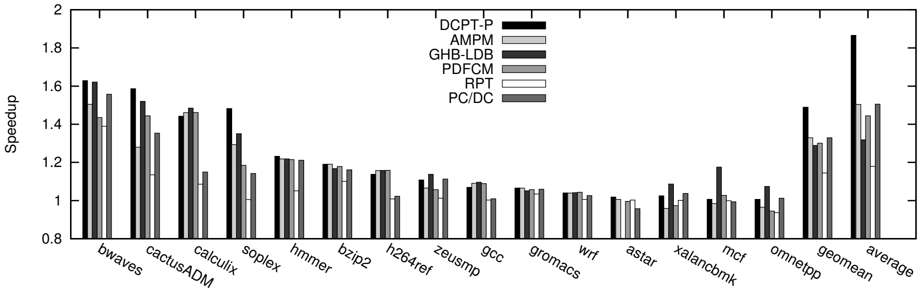


Figure VII.8: 2 MB L2 cache. Benchmarks with small speedups.

to a baseline where no prefetching is performed. Because there is a wide range of speedups (up to 6.6X) we have opted to use two graphs to increase readability. In addition, we do not show the benchmarks dealII, gobmk, tonto, perlbench, sjeng, gamess, namd, povray. In all of these benchmarks, the performance impact of prefetching was less than 5% for all the prefetchers. In cases where the simulation did not terminate within 48 hours we show an speedup of 0, rather than tampering with the original code.

Overall, DCPT-P shows good performance across all benchmarks. DCPT-P is the best performing prefetcher on 11 of the 21 benchmarks shown. The good performance of both soplex and sphinx3 is due to partial matching. Leslie3d and milc benefits greatly from the L1 hoisting technique. Also, it is worth noting that GHB-LDB performs very well on xalncbmk, mcf and omentpp. This is due to the global (intra-PC) analysis done by this type of prefetcher. However, GHB-LDB performs worse than it's predecessor, PC/DC, on GemsFDTD and libquantum. Although both GHB-LDB and PDFCM both extends PC/DC, their performance is on average almost equal. Although AMPM prefetching is not the best prefetcher for any single benchmark, it nevertheless achieves signifcant speedups across the entire benchmark suite. On average, DCPT-P provides an arithmetic mean speedup

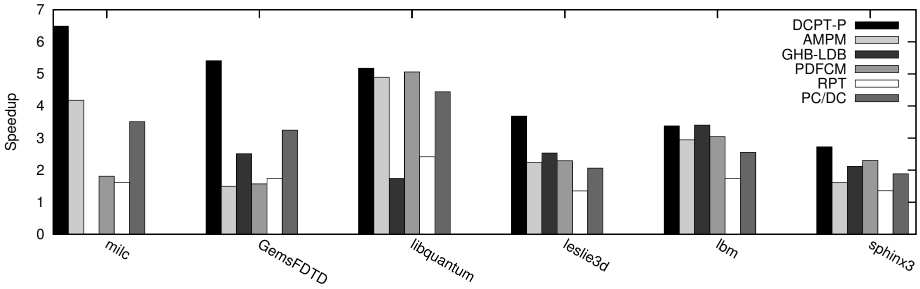


Figure VII.9: 512KB L2 cache. Benchmarks with large speedups.

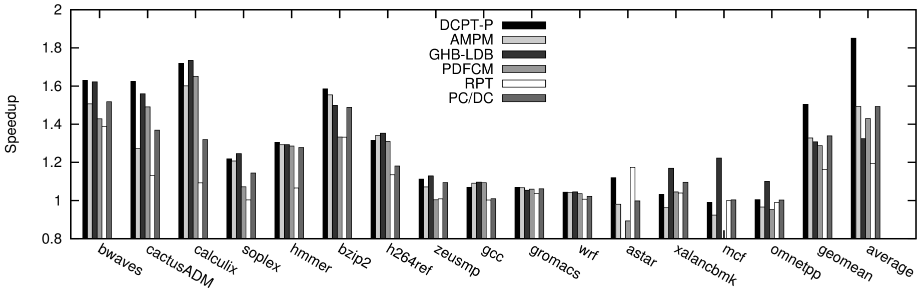


Figure VII.10: 512KB L2 cache. Benchmarks with small speedups.

of 87%. AMPM, GHB-LDB and PDFCM has speedups of 50%, 32% and 44% respectively.

In Figure VII.9 and VII.10 we reduce the L2 capacity to 512KB. This is the same configuration as config 3 in DPC-1. Overall, we observe the same general trends. The most significant changes from reducing the size of the L2 can be observed on leslie3d, calculix, bzip2 and h264ref. In this configuration PDFCM causes performance degradation on astar and omnetpp. Surprisingly, RPT prefetching is the best prefetcher on astar. On this benchmark, most of the other prefetchers has very high miss rates, especially when prefetching into the L1 cache. Thus, the more conservative prefetcher performs well. Additionally, the benefits of GHB-LDB on mcf, omnetpp and xalncbmk increases.

Figure VII.11 and VII.12 provides insight into the relative performance benefits of the three techniques proposed in this work. Undoubtedly, the basic DCPT design is responsible for most of the performance gain. This is because it is responsible for bridging the last level cache to main memory gap and thus has the most potential. Both Partial matching and L1 hoisting contribute to the overall performance. Interestingly, the effects of the two does not seem to be cumulative, but rather

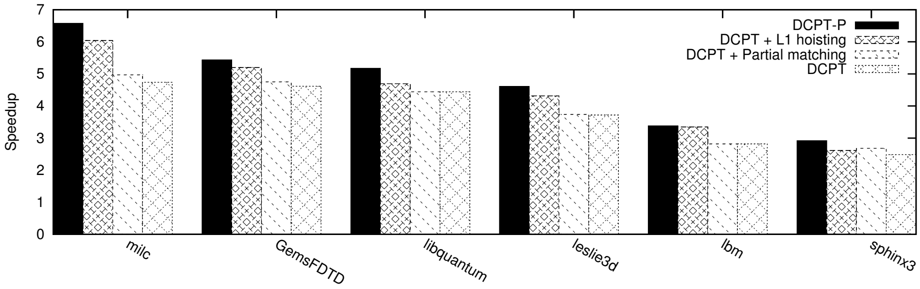


Figure VII.11: Breakdown of performance contribution of DCPT-P. Benchmarks with large speedups.

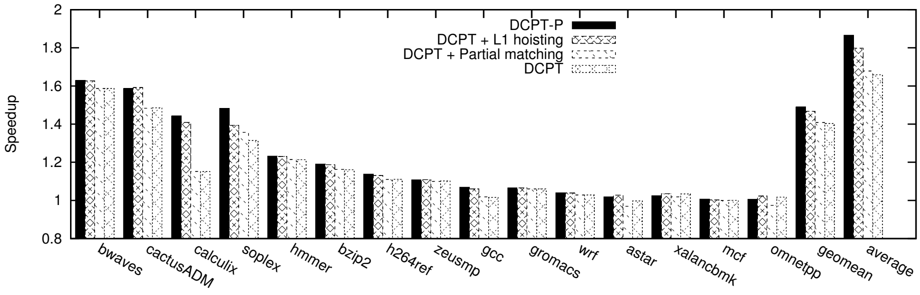


Figure VII.12: Breakdown of performance contribution of DCPT-P. Benchmarks with small speedups.

synergistic. For instance, on libquantum, switching off partial matching reduces performance somewhat. Switching off L1 hoisting reduces performance even more, but there is no difference between this configuration and switching both L1 hoisting and partial matching off. On both omnetpp and astar we see that partial matching actually causes a performance degradation. This effect is due to the much lower accuracy of partial matching, which in turn causes bandwidth saturation.

VII.5.1 Area and performance trade-offs

So far, we have focused our attention on performance. However, it is possible to optimize for area as well. The largest structure in DCPT-P is the table holding the entries. In this section, we explore the area and performance trade-off of changing some of the key table parameters. In Figure VII.13, we show the performance impact of increasing the number of deltas in each entry. The speedups are reported relative to the same case with no prefetching. Although the unlimited bandwidth

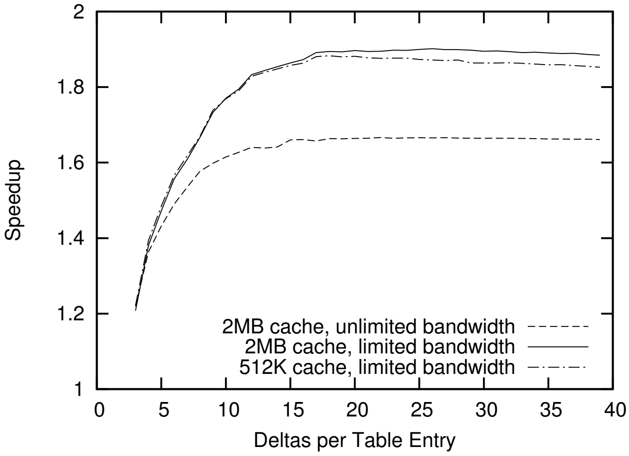


Figure VII.13: Average speedup as a function of the number of deltas in each entry

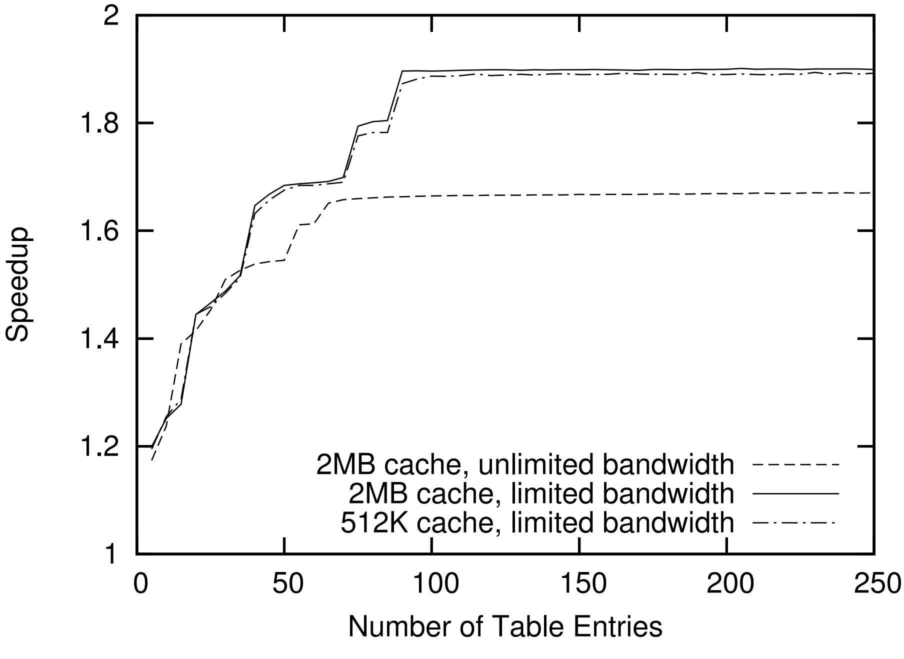


Figure VII.14: Average speedup as a function of the number of table entries

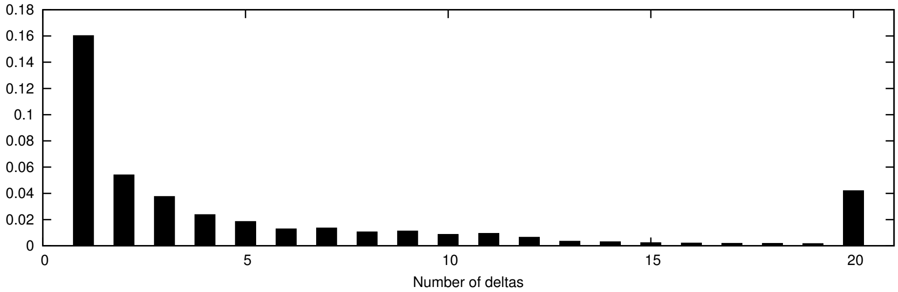


Figure VII.15: Distribution of the number of deltas registered in a table entry upon replacement.

case has higher absolute performance, the relative speedup of prefetching is lower. Increasing the number of deltas has three distinct effects. Firstly, it increases the probability of a match, thus the number of prefetches increases. Secondly, it increases the effective prefetch distance. Finally, it increases power and area as the number of comparators has to be increased. Although DCPT-P is highly accurate, a large prefetch distance can cause problems, because blocks are fetched too soon. This poses a problem because the blocks may be either evicted before they are used and/or displace other data that is currently needed. This effect can be seen by examining the difference between the 2MB and 512K cases in Figure VII.13. In the 512K case, performance starts to drop after about 18 deltas, and declines faster than in the 2MB case. Additionally, the knee in the graph in the bandwidth unlimited case is shifted to the left compared to the bandwidth limited cases. This suggests that a higher prefetching distance can mask some transient bandwidth contention as well.

In Figure VII.14, we show the average speedup as a function of the number of entries in the DCPT-P table. Performance increases as the number of entries is increased. After roughly 100 entries there is no performance gain in increasing the size of the table.

VII.6 Discussion

In the design of DCPT-P we have omitted several interesting design ideas, either because they provide little performance benefit or that they will increase the overall complexity of the design and obscure the more central mechanisms in DCPT-P. In this section, we will discuss some of these design options.

In Figure VII.15, we show the distribution of the number of deltas that has been registered in a entry when it is replaced. DCPT-P requires at least three deltas before it can begin prefetching. As such, the wast majority of table entries are never used for actual prefetching. Thus, much of the table space is wasted on inactive table entries. A possible solution is to use two tables. The first table is a smaller version of the DCPT table, that can hold up to two deltas. If the entry produces more deltas, then that table entry is promoted into the larger table. A second approach is to modify the simple LRU replacement policy in the table to give increased weight to entries with more deltas.

We observed that several of the patterns are simple repeating patterns with a short period. It is possible to capture much of the benefit of DCPT-P by using fewer deltas and analyze the delta pattern to see if it repeats. If it does, then the pattern can be extrapolated. In addition to decreasing the storage requirements by requiring fewer deltas, this approach also gives the possibility of varying the prefetch distance dynamically [5, 17].

The pattern matching step is at the core of the DCPT-P heuristic. It is possible to implement this step in a variety of ways depending on the performance and area requirements. Our implementation uses several comparators to examine every possible match location in parallel. To reduce the number of comparators, it is possible to split this step into multiple stages. Consequently, pattern matching can be performed in an iterative fashion by reusing the comparators. As previously shown in Figure VII.3, the probability of finding a match in the beginning of the delta stream is high. This is because of the prevalence of repeating patterns with short periods. Thus, the probability of finding a match during the first few iterations is high, reducing the average latency.

Another possibility is to limit the search to a subset of the deltas, thus reducing the number of comparators or iterations needed. We investigated limiting the number of deltas searched for a match. As expected, reducing the probability of finding a match decreases overall performance because patterns with long periods are not detected.

Partial matching increases coverage at the cost of decreased prefetcher accuracy. In our implementation we treat prefetches generated by full and partial matching equally. In a more bandwidth-constrained environment it might be beneficial to not treat them equally and only issue prefetches generated by partial matching if there is ample off-chip bandwidth available [6].

Finally, we looked at allowing partial matching to issue multiple prefetches per delta. Because partial matching reduces spatial resolution, the deltas after the match also have reduced resolution. It is possible to compensate for this reduced resolution by issuing multiple prefetches covering the range of possible LSBs. However, because partial matching reduces overall accuracy, we found that issuing multiple prefetches quickly saturated off-chip bandwidth which resulted in reduced performance.

The simulation framework we have opted to use has some limitations. For instance, the look-up time of the predictor is not accounted for. Furthermore, a very simple DRAM model is used, the 4KB storage limit is somewhat arbitrary and techniques which can deal with large off-chip meta-data has been developed [18]. Overall, we chose to use this framework so that a fair comparison with previously proposed prefetchers could be conducted.

VII.7 Conclusion

In this paper, we have presented a novel low-complexity prefetching heuristic called DCPT-P. DCPT-P uses a table indexed by the PC of the load. Each table entry stores a large amount of history per load instruction in the form of deltas. By storing deltas rather than full miss addresses, we save a significant amount of memory and make pattern matching easier. Pattern matching is done by using Delta Correlation, originally proposed by Nesbit et al. [11]. This technique is very effective at detecting patterns with periods shorter than the amount of history stored.

We also introduce the concept of L1 hoisting. L1 hoisting is a technique that combines with DCPT-P to issue highly accurate and timely prefetches into the L1 cache. To deal with several real-world problems with prefetching, we have introduced a mechanism called partial matching which reveals previously hidden patterns by reducing spatial resolution.

Our technique builds upon and expands several ideas presented during the first data prefetching championship (DPC-1). We have examined the top performers extensively and extracted key properties of these prefetchers and improved upon their ideas and synthesised them into a low complexity, storage efficent and high performance prefetcher. By using the code submitted to the DPC-1 contest we can be confident that the comparison with other prefetching techniques is accurate. On average, DCPT-P provides an arithmetic mean speedup of 87% on the SPEC2006 benchmark suite.

Bibliography

[1] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[2] F. Dahlgren and P. Stenstrom. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(4):385–398, Apr. 1996.

[3] M. Dimitrov and H. Zhou. Combining local and global history for high performance data prefetching. In *Data Prefetching Championship-1*, 2009.

[4] DPC-1. Data prefetching championship rules. `http://www.jilp.org/dpc/framework.html`.

[5] M. Grannaes and L. Natvig. Dynamic parameter tuning for hardware prefetching using shadow tagging. In *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[6] M. Grannaes, M. Jahre, and L. Natvig. Low-cost open-page prefetch scheduling in chip multiprocessors. In *IEEE International Conference on Computer Design (ICCD) 2008*, 2008.

[7] M. Grannaes, M. Jahre, and L. Natvig. Storage efficient hardware prefetching using delta correlating prediction tables. In *Data Prefetching Championships*, 2009.

[8] I. Hur and C. Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *HPCA '09: Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pages 443–454, 2009.

[9] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching prefetch: Optimization friendly method. In *Data Prefetching Championship-1*, 2009.

[10] A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS*, 2008.

[11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *High-Performance Computer Architecture, International Symposium on*, 0:96, 2004. ISSN 1530-0897. doi: http://doi.ieeecomputersociety.org/10.1109/HPCA.2004.10030.

[12] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[13] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.25.

[14] L. M. Ramos, J. L. Briz, P. E. Ibáñez, and V. Viñals. Multi-level adaptive prefetching based on performance gradient tracking. In *Data Prefetching Championship-1*, 2009.

[15] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356887.356892.

[16] SPEC. Spec 2006 benchmark suites, 2006. `http://www.spec.org`.

[17] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

[18] T. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *High Performance Computer Architecture (HPCA)*, 2009.

Paper VIII

DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems

Magnus Jahre, Marius Grannaes and Lasse Natvig
Accepted to the 5th HiPEAC Conference, 2010

Abstract

Chip Multi-Processors (CMPs) commonly share hardware-controlled on-chip units that are unaware that memory requests are issued by independent processors. Consequently, the resources a process receives will vary depending on the behavior of the processes it is co-scheduled with. Resource allocation techniques can avoid this problem if they are provided with an accurate interference estimate. Our Dynamic Interference Estimation Framework (DIEF) provides this service by dynamically estimating the latency a process would experience with exclusive access to all hardware-controlled, shared resources. Since the total interference latency is the sum of the interference latency in each shared unit, the system designer can choose estimation techniques to achieve the desired accuracy/complexity trade-off. In this work, we provide high-accuracy estimation techniques for the on-chip interconnect, shared cache and memory bus. This DIEF implementation has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

VIII.1 Introduction

Chip Multi-Processors (CMPs) commonly share parts of the memory system. While some CMPs have private caches and only share off-chip bandwidth, other CMPs share an on-chip interconnect and cache space between cores. This resource sharing is often beneficial since it can improve resource utilization compared to a private design and facilitates efficient inter-core communication. However, sharing may also adversely affect performance when the system resources are insufficient for co-scheduled processes. This is due to the use of rudimentary hardware policies like First Come First Served (FCFS) and Least Recently Used (LRU) which were primarily designed for use in single-core processors. These policies do not provide predictable resource allocations because processes with higher access frequencies receive a larger part of the shared resource [11, 15]. Since CMPs often run multiprogrammed workloads, the performance of a single process can be heavily influenced by the processes it is co-scheduled with.

Resource allocation techniques that attempt to alleviate interference problems, commonly aim their effort at improving *fairness* and/or *Quality of Service (QoS)*. A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [9]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when co-scheduled with any other process [3]. Nesbit et al. [12] propose a high-level architecture for resource allocation systems which divide the system into three independent, cooperating modules. Here, the *feedback mechanisms* provide measurements of the current resource utilization and/or the performance of the running programs. Then, the *allocation policy* decides on a new and improved resource allocation and implements this with the *allocation mechanisms*. Since resource allocations do not change very often, allocation policies should be implemented in software to achieve flexibility. On the other hand, allocation and feedback mechanisms that interact closely with the hardware resources, must be implemented in hardware for efficiency.

In this work, we provide the first detailed implementation of a unified feedback mechanism for the hardware-managed, shared memory system called the Dynamic Interference Estimation Framework (DIEF). DIEF dynamically estimates the average memory latency a process would experience if it had exclusive access to all shared resources. In addition, DIEF measures the actual shared memory latency to establish the relative latency impact from sharing effects. Choosing average memory latency as the interference metric has the advantage that the total interference latency is the sum of the interference latency of each shared unit. Consequently, the system designer can choose interference estimation techniques that achieve the appropriate accuracy/complexity trade-off. Since processing cores can hide latency, an allocation policy needs a performance-oriented feedback mechanism to complement DIEF which can be provided by well-known techniques like performance counters [19].

In this work, we aim our efforts at providing an accurate DIEF implementation.

To accomplish this, we develop interference measurement mechanisms for ring and crossbar interconnects, shared caches and a multi-channel DDR2 memory bus. These mechanisms are tested on a variety of CMP architectures with 4, 8 or 16 cores, 2 or 3 cache levels and 1, 2 or 4 memory bus channels. DIEF is very accurate for these architectures and has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

VIII.2 Background

VIII.2.1 Interference Definition and Metrics

When evaluating CMP memory system fairness, it is convenient to compare to a baseline where interference does not occur. One way of creating such a baseline is to let the process run in one processing core of the CMP and leave the remaining cores idle [5, 10]. Consequently, the process has exclusive access to all shared resources, and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for shared resources in the *shared mode*. We refer to a baseline created in this way as a *Single Program Baseline (SPB)*.

It is also possible to create a fairness baseline by statically partitioning all shared resources equally among the processors [3]. We refer to this baseline type as a *Multiprogrammed Baseline (MPB)*. The main advantage of MPB is that it exists in the shared mode. Consequently, it is easy to ensure that a fairness technique does not perform worse than the baseline. However, MPB also has three major disadvantages. Firstly, it only accounts for interference in the resources that have been statically and equally partitioned. This can lead to erroneous results if important interference sources are missed. Secondly, static and equal division of DRAM bandwidth does not lead to a static and equal division of latency [11]. The reason is that the latency of a request depends heavily on which requests was issued before it. Consequently, it may be difficult to implement a good static and equal sharing baseline for the memory interface. Finally, the relationship between performance and resource allocation is rarely linear [15]. Consequently, a process may experience severe performance degradation in the statically shared baseline. If a fairness technique then removes this degradation, one might be lead to believe that the technique also improves throughput when the degradation in fact was due to the baseline's suboptimal resource allocation.

These problems can be avoided by using the Single Program Baseline (SPB). Unfortunately, SPB does not exist in the shared mode. By definition, it requires that the performance in the shared mode is compared to the interference-free private mode. In this work, we provide a feedback mechanism that estimates SPB latency at runtime. We define the interference I_i experienced by a request i as the difference between the shared mode latency L_i and private mode latency $\mathcal{L}_i$ (i.e.

$I_i = L_i - \mathcal{L}_i$). This definition is an extension of the interference definition by Mutlu and Moscibroda [11].

The shared mode estimate of the private mode latency $\hat{\mathcal{L}}_i$ may be different from the actual private mode latency $\mathcal{L}_i$. Consequently, it is important that a feedback mechanism minimizes the difference between these values. We define the measurement error for request i to be $E_i = \hat{\mathcal{L}}_i - \mathcal{L}_i$. Since the interference estimate $\hat{I}$ is related to the private mode latency estimate $\hat{\mathcal{L}}$ by the formula $\hat{\mathcal{L}}_i = L_i - \hat{I}_i$, the feedback mechanism can choose to estimate either $\hat{\mathcal{L}}_i$ or $\hat{I}_i$ and compute the other. A dynamic resource allocation technique will use $\hat{\mathcal{L}}$ to establish the relative impact of interference on the different running processes. Consequently, the impact of the error depends on the shared mode latency L. To account for this we define the relative error $\mathcal{E}_i = E_i/L_i$. We aggregate multiple errors by using the arithmetic mean, standard deviation and root mean squared error of E and $\mathcal{E}$.

VIII.2.2 Modern Memory Bus Interfaces

Memory bus scheduling is a challenging problem due to the 3D structure of DRAM consisting of rows, columns and banks. Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict* and is very expensive in terms of latency. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

Rixner et al. [17] proposed the First Ready - First Come First Served (FR-FCFS) algorithm for scheduling DRAM requests. Here, memory requests are reordered to achieve high page hit rates which result in increased memory bus utilization. This algorithm prioritizes requests according to three rules: prioritize ready commands over commands that are not ready, prioritize column commands over other commands and prioritize the oldest request over younger requests.

VIII.3 Shared Memory System Latency Taxonomy

The main advantage of measuring interference in terms of average round trip latency through the shared memory system is that the total interference of a single

Table VIII.1: Memory System Latency Taxonomy

Module	Type	Description	SM	PM	Int.
Interconnect	Entry (*ie*)	The number of cycles a request is kept in the private cache MSHR before it is accepted into an interconnect queue	L_i^{ie}	$\mathcal{L}_i^{\mathrm{ie}}$	I_i^{ie}
	Queue (*iq*)	The number of cycles spent in the interconnect queue	L_i^{iq}	$\mathcal{L}_i^{\mathrm{iq}}$	I_i^{iq}
	Transfer (*it*)	The number of cycles spent on transferring the request from source to destination	L_i^{it}	$\mathcal{L}_i^{\mathrm{it}}$	I_i^{it}
	Delivery (*id*)	The number of cycles a request was delayed because a shared cache bank could not accept requests due to insufficient buffer space	L_i^{id}	$\mathcal{L}_i^{\mathrm{id}}$	I_i^{id}
Shared Cache	Capacity (*cc*)	The number of cycles used to service a miss that would not occur if the process had exclusive access to the shared cache	-	-	I_i^{cc}
Memory Controller	Entry (*me*)	The number of cycles a request was delayed in a shared cache MSHR before it was accepted into a memory controller queue	L_i^{me}	$\mathcal{L}_i^{\mathrm{me}}$	I_i^{me}
	Queue (*mq*)	The number of cycles a request spent in the memory controller queue	L_i^{mq}	$\mathcal{L}_i^{\mathrm{mq}}$	I_i^{mq}
	Transfer (*mt*)	The number of cycles the request occupied the memory data bus	L_i^{mt}	$\mathcal{L}_i^{\mathrm{mt}}$	I_i^{mt}
Shared Memory System	Total	The total number of cycles a request uses through the entire hardware-controlled, shared memory system	L_i	$\mathcal{L}_i$	I_i

request is the sum of the interference it experiences in each of the shared units. Consequently, it is possible to independently implement and validate the feedback mechanism for each source of interference. In this work, we develop a comprehensive view of memory system interference which is shown in Table VIII.1.

The hardware-controlled, shared memory system commonly consists of three types of units. Firstly, an interconnect is needed to connect the private caches to one or more shared caches. Secondly, there can be one or more levels of shared caches with varying sharing degrees. Finally, off-chip bandwidth can be shared between cores. Although the organization of these shared units will vary from CMP to CMP, we believe that this model captures the essential types of interference in the hardware-controlled, shared memory system.

Within these units, the shared resources are either *bandwidth* or *capacity*. In the memory bus and interconnects, bandwidth is the main shared resource. However, memory requests are kept in finite buffers while waiting for access to the shared transmission channels. Consequently, there are also different forms of capacity in-

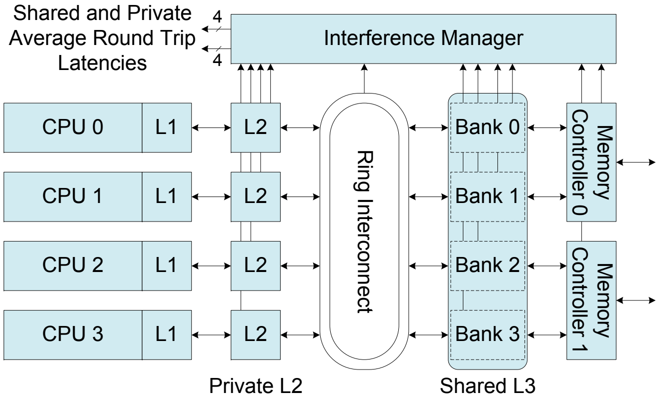


Figure VIII.1: Dynamic Interference Estimation Framework (DIEF) Architecture

terference in these units. We divide the latency through the units where bandwidth is the main shared resource into four parts. The *entry* latency is the latency the request experiences while waiting to be accepted into the input queue. Then, the *queue* latency is the number of cycles it spends in the queue before it is granted access to the resource. The next latency type is the *transfer* latency which is the number of cycles it takes to transfer the request from source to destination. Finally, it might not be possible to deliver the request if the destination lacks sufficient buffer space. In this case, the request experiences an additional *delivery* latency. There is no delivery latency in the memory bus since the last level cache must be able to receive responses to avoid deadlocks.

To provide system-wide, latency-based interference measurements, the latency cost of shared cache interference misses must be established. This problem can be solved by observing that interference misses are associated with the latency penalty of retrieving the data from the next cache level or memory. If we assume one level of shared caches, the cache capacity interference experienced by request i is the sum of request i's memory bus entry, queue and transfer latency ($I_i^{\mathrm{cc}} = L_i^{\mathrm{me}} + L_i^{\mathrm{mq}} + L_i^{\mathrm{mt}}$). For convenience, we use the first letter of the shared unit (i.e. i, c or m) and the first letter of the latency type (i.e. e, q, t, d or c) to produce a two-letter identifier (e.g. **i**nterconnect **e**ntry is **ie**).

VIII.4 The Dynamic Interference Estimation Framework

The purpose of a dynamic interference estimation technique is to provide a reliable measure of how memory system interference affects the running processes. In

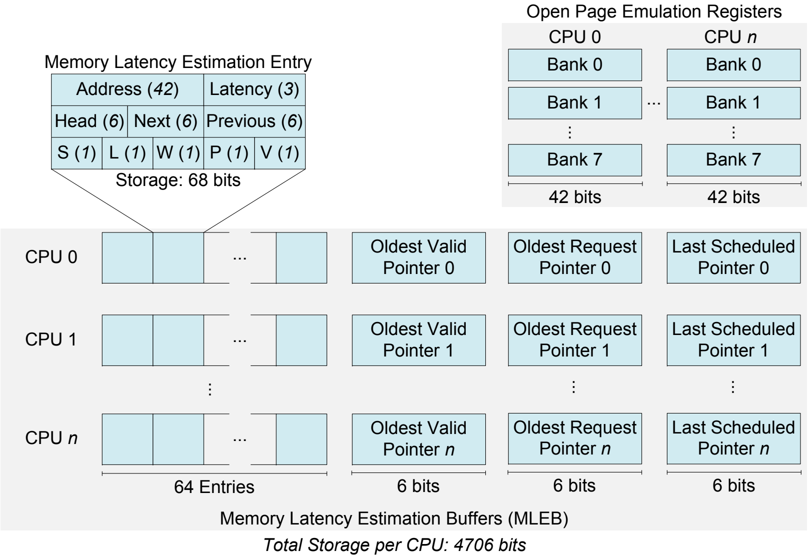


Figure VIII.2: Private Memory Bus Emulation

this work, we propose the Dynamic Interference Estimation Framework (DIEF) that continuously monitors all shared units to provide accurate interference estimates. Figure VIII.1 shows DIEF's high-level architecture where each shared unit is augmented with extra functionality (not on the unit's critical path) that measures interference and/or latencies at runtime. These measurements are continuously communicated to the Interference Manager which uses it to measure the shared mode average round trip latency L and create an estimate $\hat{\mathcal{L}}$ of the private mode latency $\mathcal{L}$. Since memory bus interference is the interference type with the largest impact, most of our efforts are directed at estimating this latency type [7]. The operating system must inform DIEF of context switches to ensure that the measurements are not polluted by the actions of other processes. In the case of multi-threaded applications, the operating system also needs to instruct DIEF to treat the application's set of processing cores as one entity. Without loss of generality, we consider the situation where each core runs one single-threaded application in the remainder of this work.

VIII.4.1 Estimating Private Memory Bus Latency ($\hat{\mathcal{L}}^{\text{mt}}$, $\hat{\mathcal{L}}^{\text{mq}}$ and $\hat{\mathcal{L}}^{\text{me}}$)

VIII.4.1.1 Estimating Transfer and Queue Latencies ($\hat{\mathcal{L}}^{\text{mt}}$ and $\hat{\mathcal{L}}^{\text{mq}}$)

Modern memory bus scheduling algorithms reorder requests to improve memory bus throughput [17]. Therefore, the *execution order* of memory requests depend on

Table VIII.2: Status Bits

S	Transfer latency estimation $\hat{\mathcal{L}}^{\mathrm{mt}}$ is valid
L	$\hat{\mathcal{L}}^{\mathrm{mq}}$ and $\hat{\mathcal{L}}^{\mathrm{mt}}$ has been computed
W	The request is a write
P	Entry is private mode only
V	Entry is valid

Table VIII.3: $\hat{\mathcal{L}}^{\mathrm{mt}}$ Estimates

	Next State	
Prev. State	Read Bank i	Write Bank i
Hit (any bank)	40	40
Miss (any bank)	120	110
Conflict Read Bank i	200	190
Conflict Write Bank i	260	250
Conflict Read Bank j	170	160
Conflict Write Bank j	260	250

the memory bus queue contents and can be very different in the shared and private modes. However, the *arrival order* of requests is very similar. Consequently, it is possible to estimate the private mode *execution order* by emulating the private scheduling algorithm on the shared mode requests. Then, the private *execution order* and bank state determine the transfer latency estimate $\hat{\mathcal{L}}^{\mathrm{mt}}$. The queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}$ can be estimated by following the private *execution order* and accumulating transfer latencies.

Figure VIII.2 shows the hardware support needed to emulate a private memory bus. This hardware is not on the critical path and consists of *n Memory Latency Estimation Buffers (MLEB)* (one for each processor). Each time the memory controller receives a request from a certain CPU, it is added to the corresponding MLEB. When the request is serviced by the memory controller, the state stored in this buffer is used to estimate its private mode queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}$ and transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}$. This calculation can be allowed to take on the order of tens of processor cycles since the memory bus is commonly clocked at a much lower frequency than the processing core.

Each estimation entry has a head pointer, a next pointer and a previous pointer. The previous/next pointers store the private *execution order* by pointing to the element that was scheduled before/after the request in the private mode. The head pointer points to the estimation entry that was the next to be serviced when the request was added, and it is used to estimate queue latency. Furthermore, each entry contains five status bits: S, L, W, P and V. These are explained in Table VIII.2. Finally, the *Oldest Valid Pointer* points to the oldest valid MLEB entry, the *Oldest Request Pointer* points to the oldest non-serviced entry and the *Last Scheduled Pointer* points to the most recently scheduled entry.

To improve estimation accuracy, we add the Open Page Emulation Registers. These where originally proposed by Mutlu and Moscibroda [11] and are used to estimate whether a request is a page hit, miss or conflict. Here, each register holds the address of the last accessed memory page. These registers are also used to schedule requests according to the FR-FCFS scheduling algorithm [17].

Generally, there are more queued requests in the MLEB than in the private mode

Algorithm 1 Private Memory Bus Queue and Transfer Latency Estimation

```
procedure ESTIMATEPRIVATELATENCIES(Memory request $r$)
    while $r$ not serviced do
        Emulate FR-FCFS scheduling of elements within horizon given by the Page Locality Factor
    Initialize request pointer $c$ to point to head($r$) and queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}_{\mathrm{r}}$ to 0
    while $c$ is not equal to $r$ and $c$ is scheduled before $r$ do
        Increment queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}_{\mathrm{r}}$ with the transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{c}}$ of request $c$
        Update $c$ by following the next pointer of $c$
    Invalidate any entries that are no longer needed to compute queue and transfer latencies
    return transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{r}}$ and queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}_{\mathrm{r}}$ of request $r$
```

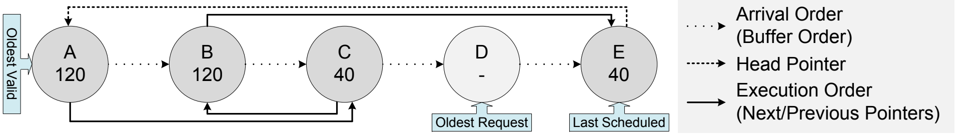


Figure VIII.3: Memory Bus Queue and Transfer Latency Estimation Example

memory bus queue since competition for the bus is more severe in the shared mode. This can result in overestimating the number of page hits if the process has sufficient page locality. To account for this, we add a parameter called the *Page Locality Factor*. This factor determines the number of estimation entries that should be examined while looking for a page hit. Setting the page locality factor to 1 assumes no reordering in the private memory system.

If we ignore the effects of shared cache interference, the requests that reach the memory bus are the ones that are not filtered out by the on-chip caches. Since we use the same cache hierarchy in the shared configuration and the baseline, the order of the memory request are nearly identical but their timing will be different. However, there may be differences resulting from the interleaving of writebacks and reads since the memory controller may reorder requests differently in the two configurations. When cache interference is taken into account, the request stream can be very different. Consequently, the shared cache interference technique should identify both private- and shared-only requests and communicate this information to the memory bus interference technique.

Finally, we need to produce estimates of the shared mode queue latency. This can be accomplished by adding a register for all queue entries and incrementing it with the memory bus transfer latency every time a request is finished. Alternatively, a request can be assigned a timestamp on arrival and this timestamp can then be compared to the value of a counter when the request is issued.

The Latency Estimation Algorithm Algorithm 1 summarizes the estimation algorithm for the private memory bus transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}$ and queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}$. We illustrate the estimation procedure with the example in Figure VIII.3. There are five queued requests, and request E has just been serviced by the shared mode memory controller. To determine the transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{E}}$ of E, the estimation algorithm emulates scheduling requests within the limit given by the Page Locality Factor. In this example, request A is serviced first and its transfer latency is estimated. Then, request C is serviced before B since it is a private mode page hit. Finally, request E is serviced before D since it accesses the same page as B which gives $\hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{E}} = 40$. Then, we can estimate $\hat{\mathcal{L}}^{\mathrm{mq}}_{\mathrm{E}}$ by following E's head pointer to A and accumulating the transfer latencies of all elements between A and E in the private execution order. Consequently, the queue latency estimate for E is $\hat{\mathcal{L}}^{\mathrm{mq}}_{\mathrm{E}} = \hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{A}} + \hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{C}} + \hat{\mathcal{L}}^{\mathrm{mt}}_{\mathrm{B}} = 120 + 40 + 120$.

There are a number of possible transfer latencies due to different active pages, overlapping of commands with data transfers from other banks and timing constraints regarding when a bank can be precharged. However, we observed that only a small number of these possible latencies occur frequently in the private mode. Consequently, it is possible to store the most common transfer latencies in a lookup table. Then, the latency is determined by whether the previous and next requests are to the same bank and whether they are reads or writes. This lookup table is created at design time by analyzing the private mode access behavior for the chosen memory bus type. Table VIII.3 shows the lookup table of the DDR2 memory bus used in this work.

A private-only entry (P bit set) can be invalidated when its latency is not needed to compute the queue latency of any other element. For shared mode entries, the latency of the entry must also be computed before it can be deleted. In addition, we require that the most recently scheduled element is not invalidated. The deletion algorithm is based on the observation that the head pointer of the oldest undeletable element e in the *arrival order* will point to the oldest head element h in the *arrival order*. Consequently, we know that all elements after h in the *execution order* are needed to compute the queue latency for e. If an entry has been removed due to insufficient buffer space, we use the last computed transfer and queue latency.

VIII.4.1.2 Estimating Memory Bus Entry Interference $\hat{I}^{\mathrm{me}}$

When the memory bus queue becomes full, the memory controller blocks and the requests remain in the shared cache MSHRs. We account for this interference by observing that the maximum number of requests a processor core can issue simultaneously is the sum of MSHRs and writeback buffers in the last-level private cache. Furthermore, the shared buffers will be dimensioned to handle roughly c times this number of requests (c = number of cores) since too few buffers will lead to frequent performance bottlenecks. The effect of this observation is that a single core will not be able to fill the buffers in the shared part of the memory system. Consequently, any shared mode latency due to memory bus blocking is interference.

VIII.4.2 Estimating Cache Capacity Interference $\hat{I}^{\mathrm{cc}}$

To identify shared cache interference misses, we use an Auxiliary Tag Directory (ATD) [4, 16] per core. Each time a request is received in the shared cache, the request is inserted into the ATD belonging to the processor that sent the request. Consequently, the ATD contains the tags the processor would have had in the shared cache if it was running alone. On each access, we compare the output from the ATD with the output from the actual cache. If the request is a hit in the ATD and a miss in the real cache, we store a timestamp and tag the request as a shared mode only cache miss. This bit is used to keep the request out of the memory bus private mode latency estimation. When the request has been serviced in the memory bus and returned to the cache, we retrieve its latency and communicate it to the Interference Manager as cache capacity interference. We also record if an ATD entry would have been written to in the private mode. In this case, a replacement would have triggered a writeback in the private mode. When this happens, we insert a private mode only writeback request into the memory bus private mode latency estimation.

In this work, our aim is to accurately measure interference. Consequently, we are willing to invest a fair bit of area into making the estimates accurate. We use CACTI version 5.3 [20] to establish that the size of each ATD is roughly 4% of the shared cache area. Qureshi et al. [16] showed that sampling as few as 16 to 32 sets can be sufficient to represent cache behavior. With 32 sets, the area of each ATD is reduced to around 0.01 % of the shared cache area. In DIEF, using set sampling is not straight forward since the memory bus interference estimation mechanism needs to know which misses are shared-only interference misses. This problem can likely be avoided at the cost of reduced accuracy by using an estimated interference miss probability to select requests for the memory bus interference estimation. The area overhead can be further reduced at the cost of accuracy and measurement latency by time multiplexing the ATDs. Work in this direction is underway.

VIII.4.3 Estimating Interconnect Interference ($\hat{I}^{\mathrm{ie}}$, $\hat{I}^{\mathrm{iq}}$, $\hat{I}^{\mathrm{it}}$ and $\hat{I}^{\mathrm{id}}$)

The main component of interconnect interference is due to requests having to wait for access to the shared transmission medium ($\hat{I}^{\mathrm{iq}}$). It is easy to measure interference in the ring and crossbar interconnects used in this work since latency is independent of access order. If a processor i is not able to issue a request because a request r from processor j is being transferred, we add the number of cycles request r occupied the transmission medium for each delayed request from processor i to the interference estimate. Since the interconnects may be pipelined, the number of cycles a processor delays another processor may be less than the transfer latency. In the ring interconnect, the transfer latency depends on which core the process is scheduled on and this needs to be taken into account when estimating interference. Again, we assume that all blocking due to full buffers is interference.

Table VIII.4: CMP Models

Interconnect	#CPUs	Process	Private Cache	Shared Cache	Memory Bus
Crossbar, 8/16/30 cycles end-to-end transfer latency, 32 entry queue	4	65 nm	2-way 64KB L1 Data, 2-way 64KB L1 Inst.	16-way 8MB L2	DDR2-800, 4-4-4-12 timing, 8 banks, 1KB pages, 64 entry read queue, 64 entry write queue, FR-FCFS, Open Page Policy
	8	45 nm		16-way 16MB L2	
	16	32 nm		16-way 32MB L2	
Ring, 4/4/8 cycles per hop transfer latency, 32 entry queue	4	65 nm	2-way 64KB L1 Data, 2-way 64KB L1 Inst., 4-way 1MB Unified L2	16-way 8MB L3	
	8	45 nm		16-way 16MB L3	
	16	32 nm		16-way 32MB L3	

VIII.5 Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments and have extended M5 with crossbar and ring interconnects as well as a detailed DDR2-800 memory bus and DRAM model [8]. We model two CMP architectures that are similar to current general-purpose, high-performance CMP implementations and identify these models by the name of the on-chip interconnect (i.e. crossbar or ring). Table VIII.4 summarizes the CMP models used in this work, and a further discussion of the models is provided by Jahre et al. [7]. The only difference between Jahre et al.'s configuration and ours is that we use an open page policy in the memory controller. We also use Jahre et al.'s 40 4-core workloads, 20 8-core workloads and 10 16-core workloads that were generated by picking benchmarks at random from the full SPEC CPU2000 benchmark suite [18]. The only requirement given to the random selection process is that a benchmark can only appear once in each workload. These workloads are fast-forwarded for 1 billion clock cycles before we run detailed simulation for 100 million clock cycles. To achieve synchronized measurements of L and $\mathcal{L}$, it is critical to minimize the difference between the memory requests in the shared and private modes. To ensure this, we use static cache partitioning and an infinite bandwidth interconnect and memory bus during fast forwarding such that the simulation sample starts on a similar instruction in both modes. Furthermore, we run the shared mode experiments first and then retrieve the number of instructions the benchmark committed. Then, we run the private mode simulation for the exact same number of instructions.

VIII.6 Results

In this section, we present the results from our experiments with DIEF. When not otherwise stated, we use our best performing configuration with 8192 requests per sample, a page locality factor of 3 and a 64 entry bus estimation buffer. These values were found empirically by extensive simulation.

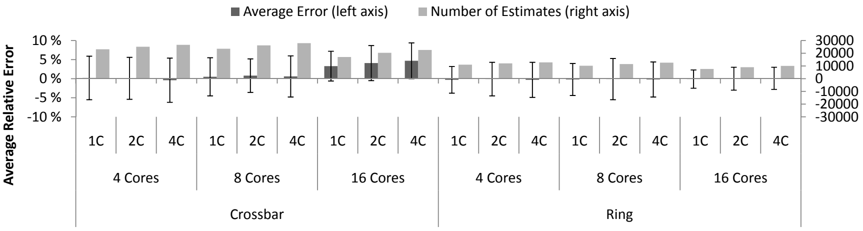


Figure VIII.4: Relative Estimation Errors and Number of Estimates

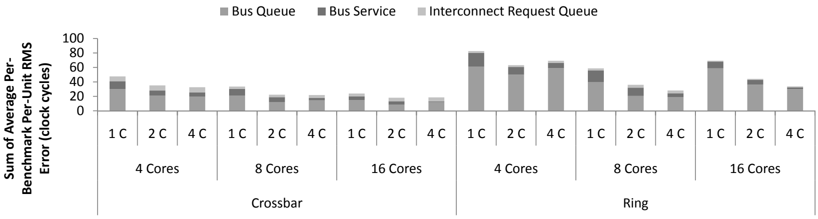


Figure VIII.5: Interference Estimation Error Breakdown

VIII.6.1 Estimation Accuracy

Figure VIII.4 shows the average relative error and one standard deviation of all estimates produced by DIEF. In addition, Figure VIII.4 contains the number of estimates used to compute these statistics. We use the abbreviations 1C, 2C and 4C to represent 1 memory bus channel, 2 memory bus channels and 4 memory bus channels, respectively. The main observation is that the average error is close to zero in all cases. Furthermore, the standard deviation is at most 5.8%.

Figure VIII.5 breaks down the average root mean squared (RMS) error for all architectures used in this work. We have removed all interference types where the average RMS error is less than 2 clock cycles to improve readability. Furthermore, cache capacity interference is not included since it has no corresponding private mode latency. Figure VIII.5 shows that most of the measurement error is due to the memory bus queue estimate $\hat{\mathcal{L}}^{\text{mq}}$. This is not surprising as our queue latency estimation model does not cover the case where a request is delayed by page hits that arrive after it. Furthermore, our model does not accurately predict the difference between the number of simultaneously queued requests in the two models. However, given the good average accuracy shown in Figure VIII.4, the measurements are likely accurate enough to be used by a dynamic fairness technique. Another observation is that the absolute measurement error is larger in the ring architectures. This is due to poor utilization of the L3 cache because the private L2 caches reduce the access frequency. Consequently, a cache thrashing process is able to evict a larger amount of the data needed by a less cache intensive thread which in

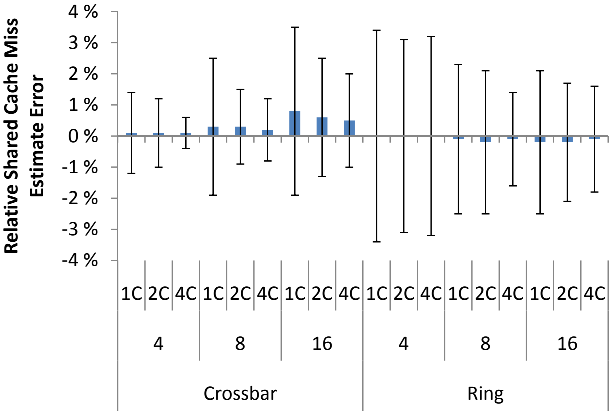


Figure VIII.6: ATD Estimation Error

turn puts a larger strain on the memory bus.

To quantify the accuracy of the ATD interference miss estimates, we count the number of actual misses and the number of interference misses. Here, the shared mode miss count estimate is computed by subtracting the number of additional shared cache misses identified by the ATD from the shared mode miss count. Then, we adapt the relative error metric to cache misses by using the estimated number of misses $\hat{\mathcal{M}}$, the actual private mode number of misses $\mathcal{M}$ and the shared mode number of misses M ($\mathcal{E} = (\hat{\mathcal{M}} - \mathcal{M})/M$). Figure VIII.6 shows that our ATD-based interference miss estimation has an average relative estimation error of at most 0.8% and maximum standard deviation of 3.4%.

Figure VIII.7 shows the distribution of the memory bus queue RMS errors for the 4-core CMP models. Here, we represent the measurement error for each instance of a benchmark by the average RMS error of the estimates for this benchmark. Then, we sort the average RMS errors such that each point in the figure represents the maximum average RMS error observed for a certain percentage of benchmarks. Figure VIII.7 shows that the memory bus queue estimates are very accurate. When 60% of the benchmarks are taken into account, the worst average RMS error observed for any architecture is 20 clock cycles. However, there is a short tail where the measurement error is significant. Since the average round trip memory latency is high in these cases, the values are most likely good enough to be used by dynamic resource allocation techniques. Finally, the lines stop at 82% for the ring architecture and 97% for the crossbar architecture because some benchmarks have too few memory requests to produce any estimates.

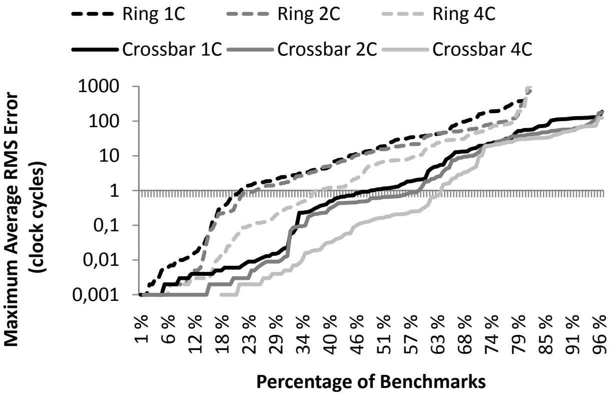


Figure VIII.7: 4-core Bus Queue Error

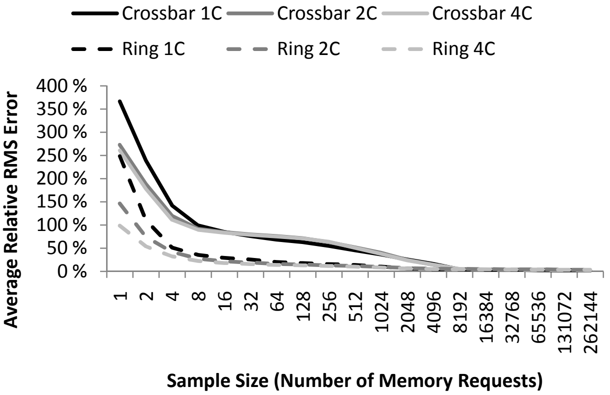


Figure VIII.8: Root Mean Squared Error. 8-core CMP Sample Size Accuracy Impact

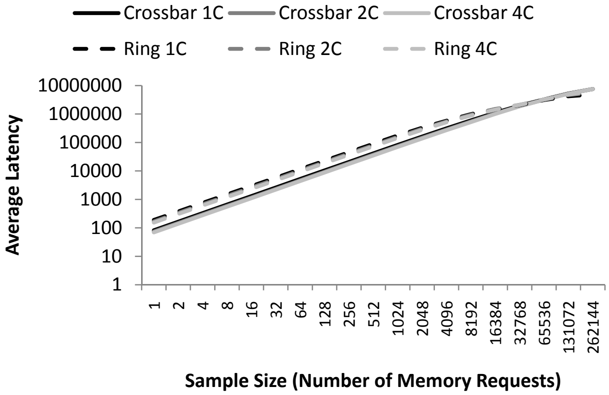


Figure VIII.9: Average Latency Between Estimates. 8-core CMP Sample Size Accuracy Impact

VIII.6.2 DIEF Parameters

In this section, we provide an empirical analysis of DIEF's main parameters: sample size, page locality factor and memory bus estimation buffer size. The choice of sample size is a trade-off between achieving low variability and receiving new estimates often enough to make high quality resource allocation decisions. Figure VIII.8 and VIII.9 shows the average relative RMS error and average latency between estimates for the 8-core architectures. Our choice of 8192 requests per sample is on the flat part of the error plot and has an acceptable average latency.

Figure VIII.10 shows the average RMS error for different page locality factors. The general trend is that the page locality factor should be low because there is usually more locality in the shared mode estimation buffer than in the private memory bus queues. This is because a larger number of requests are available to the scheduler in the shared mode due to more competition. A page locality factor of 3 is the best overall. Finally, Figure VIII.11 shows the error resulting from varying the memory bus estimation buffer size. Here, 64 entries are necessary to achieve low error for the ring architecture.

VIII.7 Related Work

A few researchers have addressed the issue of dynamic interference measurement in CMPs. Cache Scouts [21] is a shared cache interference measurement technique that estimates interference by counting the number of cache blocks that are evicted by different processors. Consequently, they assume that all blocks evicted by a different processor would have been reused which may lead to measurement errors.

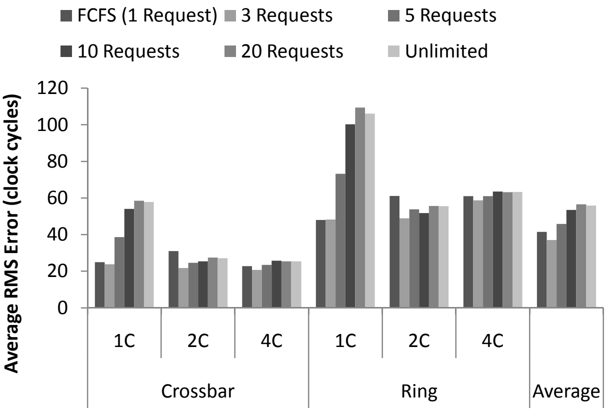


Figure VIII.10: 4-core Page Locality Factor

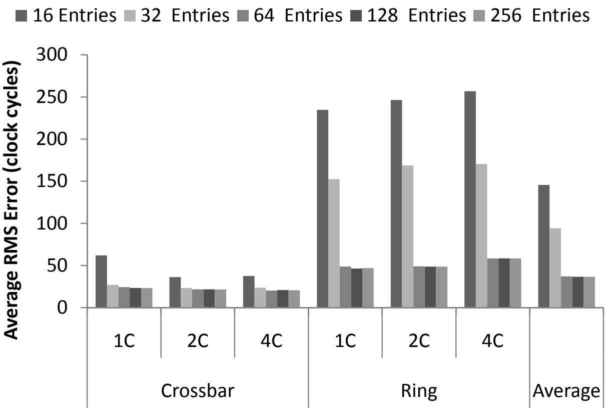


Figure VIII.11: 4-core Bus Buffer Size

Mutlu and Moscibroda [11] propose a run-time interference measurement technique that they use to guide a memory bus scheduling algorithm in a system with private caches.

Most previous studies that aim to improve resource sharing in CMP memory systems, have focused on a single component of the entire system. For example, techniques have been proposed to reduce cache capacity interference (e.g. [3, 9]), cache bandwidth interference [14] and memory bus interference [10, 11, 13]. In addition, a few researchers have investigated how a chip-wide resource management technique can be designed. Iyer et al. [6] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [12] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. In addition, Bitirgen et al. [2] showed how machine learning can be applied to the resource allocation problem.

VIII.8 Conclusion

Accurate feedback mechanisms are needed to implement robust resource allocation systems in future CMPs. In this work, we propose the Dynamic Interference Estimation Framework (DIEF) which is the first detailed implementation of a unified feedback mechanism for CMP memory systems. DIEF is a collection of techniques that cooperate to estimate the average memory latency a process would experience if it had exclusive access to all shared resources. Choosing the average memory latency as the unit of interference has the advantage that the total memory latency is the sum of the latency in each shared unit. Consequently, CMP designers can choose estimation techniques that achieve the desired accuracy/complexity trade-off for each shared unit. In this work, we describe a high accuracy DIEF implementation which has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

Acknowledgments

This project was supported in part by the Norwegian Metacenter for Computational Science (NOTUR). Lasse Natvig is a member of HiPEAC2 NoE.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches. In *MEDEA '06: Proc. of the 2006 workshop on MEmory performance*, pages 45–52, 2006.

[5] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[6] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[7] M. Jahre, M. Grannaes, and L. Natvig. A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures. In *HPCC '09: 11th IEEE Int. Conf. on High Performance Computing and Communications*, pages 622–629, 2009.

[8] *DDR2 SDRAM Specification.* JEDEC Solid State Tech. Association, May 2006.

[9] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[10] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[11] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[12] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[13] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[14] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[15] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 423–432, 2006.

[16] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA '06: Int. Symp. on Comp. Arch.*, pages 167–178, 2006.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[18] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[19] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22 (4):64–71, 2002.

[20] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACI 5.1. Technical report, HP Laboratories Palo Alto, 2008.

[21] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.

Paper IX

Exploring the Prefetcher/Memory Controller Design Space: An Opportunistic Prefetch Scheduling Strategy

Marius Grannæs, Magnus Jahre and Lasse Natvig
Preprint submitted to the Journal of Systems Architecture,
November 2010

Abstract

Prefetching is a well-known technique for bridging the memory gap. By predicting future memory references the prefetcher can fetch data from main memory and insert it into the cache such that overall performance is increased. Modern memory controllers reorder memory requests to exploit the 3D structure of modern DRAM interfaces. In particular, prioritizing memory requests that use open pages increases throughput significantly.

In this work, we investigate the prefetcher/memory controller design space along three dimensions: prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that selectively prioritizes prefetches to open DRAM pages such that performance regressions are minimized. Opportunistic prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

Keywords: Memory Systems, Prefetch Scheduling, Prefetching, Opportunistic Prefetch Scheduling, DRAM, Page Vector Table

IX.1 Introduction

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM (Figure IX.1) is exploited to increase throughput. Because of this 3D structure, the latency of a memory operation varies depending on bank conflicts and open pages. In particular, fetching data from open pages is beneficial as it has low latency and increases DRAM throughput [14].

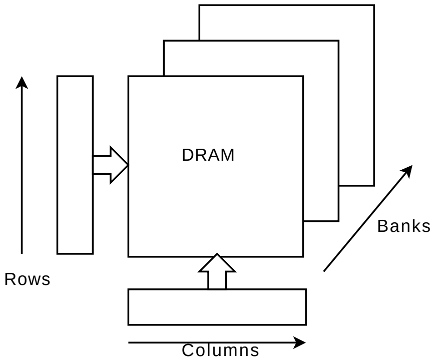


Figure IX.1: 3D structure of DRAM.

Prefetching reduces latency by fetching data before it is needed. Research has shown that prefetching can also increase bandwidth utilization by tight integration of the prefetcher with the memory controller [4, 8]. This is achieved through prioritizing prefetches based on prefetcher accuracy, available memory bandwidth and interaction with open pages. Fetching data from an open page has a much lower cost than a complete cycle of opening a page, fetching the data and closing the page. Because of this difference in cost, speculatively prefetching data from an open page is beneficial even at relatively low prefetcher accuracy [4].

Because of the complex interaction between the prefetcher and the memory controller, prefetch scheduling can be as important as determining which addresses to prefetch. In this work, we investigate the prefetcher/memory controller design space along three dimensions: prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs, which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that selectively prioritizes prefetches to open DRAM pages such that performance regressions is minimized. This scheduling strategy is able to bridge the gap between the simpler prefetchers (e.g. sequential) and the more complex prefetchers in future multicore architectures where there will be more contention for off-chip bandwidth. Using simple prefetchers is an advantage because they are easier to implement and verify.

IX.2 Related Work

IX.2.1 Prefetching

There exists a multitude of different prefetching schemes. The simplest is the *sequential prefetcher* [16], which simply fetches the next block whenever a block is referenced. However, more complex types exists as well, such as the *CZone/Delta Correlation (C/DC)* prefetcher proposed by Nesbit et al. [12, 13]. C/DC divides memory into CZones and analyses patterns contained in the reference stream by using a Global History Buffer (GHB) to store recent misses to the cache. Lin et al. [10] introduced *scheduled region prefetching* (SRP) which issues prefetches to blocks spatially near the addresses of recent demand missed when the memory channel is idle. Other types, such as the *Reference Prediction Table Prefetcher* (RPT) proposed by Chen and Baer [3], examines the pattern generated by a load instruction with a state machine. *Delta Correlating Prediction Table* (DCPT) is a table based approach which stores the history of each load instruction in the form of address deltas [5]. DCPT prefetches new data by using delta correlation to find patterns in the history of deltas. Somogyi et al. proposed *Spatial Memory Streaming* (SMS) [17]. SMS uses code-correlation to predict spatial access patterns.

IX.2.2 Memory Controllers

Memory access scheduling is the process of reordering memory requests to improve memory bus utilization. Rixner et al. [14] showed that significant speed-ups are possible when memory request reordering is applied to stream processors. In addition, Shao et al. [15] proposed *burst scheduling* in which multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization. Finally, Zhu et al. [20] showed that it is beneficial to divide the memory requests into smaller parts, and give priority to the words responsible for a processor stall in a multi-channel DRAM system.

CMPs, processors with SMT (Simultanous Multi-Threading) support and conventional shared-memory multiprocessors also benefit from memory access scheduling. Zhu et al. [19] showed that DRAM throughput could be increased in an SMT processor by using ROB (ReOrder Buffer) and IQ (Instruction Queue) occupancy status to prioritize requests. Furthermore, Hur et al. [6] use a history-based arbiter to adapt the DRAM port and rank schedule to the application's mix of reads and writes for the dual-core Power5 processor. In addition, Natarajan et al. [11] showed that a significant performance improvement is available by exploiting memory controller features in a conventional, shared-memory multiprocessor.

IX.3 Prefetch Scheduling Strategies

Earlier work have focused on the interaction between a specific prefetcher and the memory controller. Wei-Fen et al. [9] have examined how prefetches can be scheduled in a uniprocessor context with Rambus DRAM. They used a dedicated prefetch queue with a LIFO insertion policy with a scheduled region prefetcher. Cantin et al. [2] exploited open pages to increase the performance of their stealth prefetcher. In this work, we decouple the prefetching scheduling strategy from the prefetcher and examine new combinations of prefetcher and prefetch scheduling strategy. This allows us to do a design space exploration where we examine many combinations of prefetchers and prefetch scheduling strategies.

The simplest way to schedule prefetches in a memory controller is to treat them as demand reads. This method requires no additional infrastructure and most controllers can easily accommodate this technique. Because the prefetches are treated as reads, they cannot be discarded by the memory controller which in turn can lead to memory congestion and memory controller blocking due to full queues. In this paper, we refer to this strategy as the *Direct* prefetch scheduling strategy.

This situation can be improved by adding an additional queue called the *prefetch queue* [2, 8, 9]. A dedicated prefetch queue can hold prefetches separate from the reads which enables the memory controller to selectively issue or discard prefetches. Because prefetches can be discarded, the memory controller can choose to discard prefetches in the prefetch queue rather than block. However, because there are now two (in addition to the write queue) queues, a method for choosing which queue to issue reads or prefetches from is needed.

The most restrictive method is to only issue prefetches from the dedicated prefetch queue when there are no other operations pending. We refer to this as the *Idle* prefetch scheduling strategy. A more aggressive approach is to schedule prefetches when any of the prefetches currently in the queue would read data from a currently open page. This is often beneficial because a prefetch into an open page costs less than a demand read. In this paper, we refer to this strategy as the *Ready* prefetch scheduling strategy.

The accuracy of the prefetcher can be measured at runtime by tagging each prefetched cache block with a prefetch bit. This bit is set when a cache block is inserted by a prefetch. The first time the cache block is accessed, the bit is cleared and a counter is updated. Similarly, when a prefetch is issued another counter is updated. The ratio between these two counters is the estimated accuracy.

Grannaes et al. used this estimated accuracy to switch between the *Idle* and the *Ready* prefetch scheduling strategy [4]. If the estimated accuracy was below 40%,

Page Address	Bit Vector															
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
100	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table IX.1: Example Page Vector Table showing a strided prefetch pattern for page address 100.

the *Idle* strategy was used. If it was over 40% the *Ready* strategy was used. In this work, we refer to this strategy as the *Low Cost* strategy.

Lee et al. used the estimated accuracy to switch between the *Ready* and the *Direct* scheduling strategies [8]. When the accuracy was estimated to be high, the *Direct* strategy was used. Conversely, the *Ready* strategy was used when the accuracy was estimated to be low. In this paper, we refer to this strategy as the *Hybrid* strategy.

IX.3.1 Opportunistic Prefetch Scheduling

In this paper, we propose a prefetch scheduling strategy called *Opportunistic Prefetch Scheduling* which is strongly tied to the observation that prefetching from open pages is beneficial. Instead of using a queue of prefetches, we use a *Page Vector Table* (PVT) indexed by the page-address. An example of a PVT is shown in Table IX.1. The page-address is the portion of the memory address which maps to physical DRAM pages. In our setup, each DRAM page is 1KB large and each cache block is 64B large. Each PVT entry consists of a 16-bit vector where each bit represents one cache block in the page. When the prefetcher generates a prefetch for a cache block, it looks up the DRAM page in the table and sets the corresponding bit in the vector for that page.

The table is then consulted before the memory controller closes any page. If there are bits set in the corresponding bit-vector and no demand reads have been issued for those cache blocks, prefetches are issued for those addresses. This approach ensures that prefetches are only issued if they access an open page. The bitvector representation is very compact as only one bit is stored per prefetch (excluding the page tag) compared to the traditional approach using queues which holds the full address.

To reduce the potential for performance regressions, *Opportunistic* estimates the accuracy of the issued prefetches to choose one of two substrategies. If accuracy is high, prefetches are issued when a page is closed. If accuracy is low, prefetches are only issued when a page is closed and there are no demand reads in the queue.

IX.4 Methodology

To examine the impact of different prefetch scheduling strategies we have used the M5 simulator [1]. The processor architecture parameters for the simulated 4-core

Table IX.2: Processor Core Parameters

Parameter	Value
Processor Cores	4
Clock frequency	3.2 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table IX.3: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB, 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB, 8-way set associative, 64B blocks, 1 cycle latency
Level 2 Unified Shared Cache	4 MB, 16-way set associative, 64B blocks, 14 cycles latency, 8 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 9 cycles latency, 64B wide transmission channel
DDR2 memory	400 Mhz Clock, 8 banks, 1KB pagesize, 4-4-4-12 timing, dual channel in lock-step
Memory Controller	128 entry queue, Ready First - First Come, First Served policy for reads

CMP are shown in Table IX.2, and Table IX.3 contains the baseline memory system parameters. We have extended M5 with a crossbar interconnect, a detailed DDR2 memory bus and DRAM model and a detailed FR-FCFS (First Ready, First Come, First Served) [14] memory controller with integrated prefetching capabilities.

Our DDR2-implementation [7] models separate RAS, CAS and precharge commands. In addition, we model pipelining of requests, independent banks, burst mode transfers and bus contention. All prefetchers use a prefetching degree of 10. We use 1KB regions in scheduled region prefetching, 256KB CZones, a 1024-entry global history buffer and a 16-entry reference prediction table. DCPT uses a 128 entry table with each entry holding 20 18-bit entries. These values were found by extensive simulation aimed at identifying values that are suitable across a wide range of scheduling strategies and memory systems.

The SPEC CPU2000 benchmark suite [18] is used to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in Table IX.4. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. Each workload is first fast forwarded 1 billion clock cycles and then detailed simulation is carried out for 100 million clock cycles.

Table IX.4: Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mgrid, perlbmk, parser	11	vpr, twolf, applu, eon	21	perlbmk, apsi, lucas, equake	31	mgrid, equake, vpr, eon
2	lucas, gcc, mcf, twolf	12	galgel, crafty, mgrid, swim	22	vpr, crafty, vpr, mcf	32	wupwise, gap, twolf, facerec
3	eon, eon, mesa, facerec	13	twolf, fma3d, galgel, vpr	23	gzip, equake, mgrid, mesa	33	galgel, equake, lucas, gzip
4	vortex1, ammp, equake, galgel	14	bzip, vpr, bzip, equake	24	facerec, applu, fma3d, lucas	34	facerec, gcc, galgel, apsi
5	gcc, galgel, apsi, crafty	15	galgel, crafty, vpr, swim	25	gap, applu, parser, facerec	35	mesa, mcf, swim, sixtrack
6	applu, equake, art, facerec	16	mcf, wupwise, applu, mesa	26	mcf, apsi, twolf, ammp	36	mesa, sixtrack, equake, bzip
7	applu, gap, gcc, parser	17	applu, parser, apsi, perlbmk	27	swim, sixtrack, ammp, applu	37	mcf, gap, gcc, vortex1
8	gap, swim, twolf, mesa	18	mgrid, perlbmk, gzip, mgrid	28	art, fma3d, swim, parser	38	facerec, lucas, mcf, parser
9	sixtrack, fma3d, apsi, vortex1	19	mcf, sixtrack, gcc, apsi	29	apsi, gcc, vortex1, twolf	39	twolf, eon, mesa, lucas
10	ammp, bzip, equake, parser	20	ammp, gcc, art, mesa	30	mgrid, gzip, apsi, equake	40	apsi, gzip, mcf, equake

IX.5 Results

IX.5.1 Performance

Figure IX.2 shows the average speedup of each prefetch scheduling strategy with five different prefetchers (Sequential, RPT, C/DC, SRP and DCPT) in a system with one DRAM channel. *Opportunistic* performs well in combination with both the sequential and the SRP prefetcher. For RPT, C/DC and DCPT, the *Ready* and *Low cost* strategies performs slightly better. However, *Opportunistic* prefetch scheduling is able to bridge the performance gap between the simple sequential prefetcher and the more complex RPT, CDC and DCPT prefetchers. This is because the overall accuracy of these prefetchers is typically higher than for sequential and SRP prefetching. This same effect can be observed for the *Direct* strategy where performance is low for the sequential and SRP prefetcher while it is higher for the other prefetchers. This is because the strategy does not make any distinction between prefetches and demand reads. Thus, inaccurate prefetches can disrupt demand reads. The performance for the *Idle* strategy is comparably low for most prefetchers because it issues less prefetch requests due to it's strict policy. The difference between the *Low cost* and *Ready* strategies is also apparent. In combination with high accuracy prefetchers such as RPT, C/DC and DCPT, the *Ready* strategy performs better than *Low Cost.* In combination with low accuracy prefetchers, the situation is reversed. Finally, the performance of the *Hybrid* strategy is between

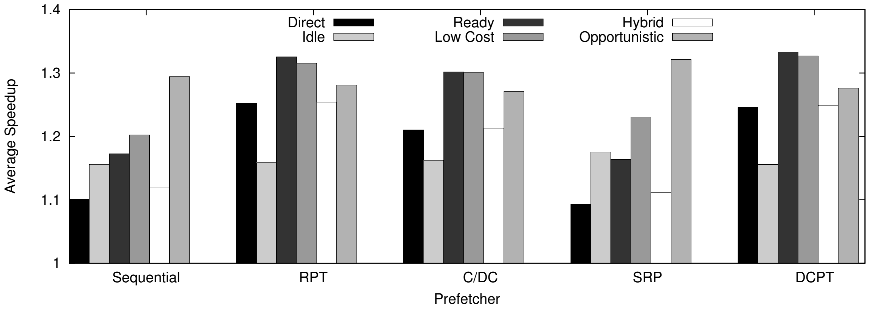


Figure IX.2: Average speedup for all cores over all workloads for different scheduling strategies and prefetchers.

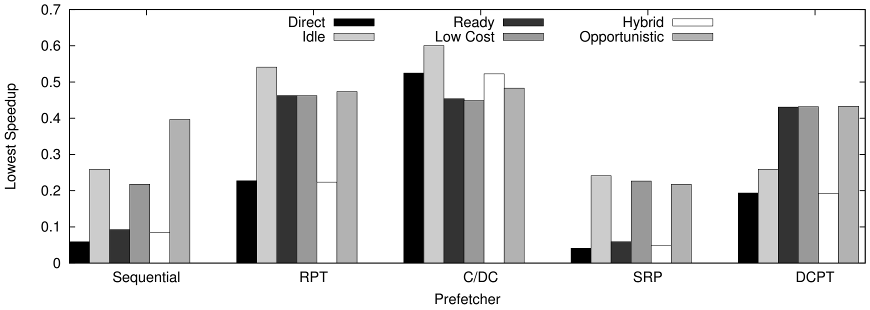


Figure IX.3: Lowest speedup for any core in any workload for different scheduling strategies and prefetchers.

the *Ready* and *Direct* strategies.

IX.5.2 Maximum Performance Regression

Prefetching can drastically increase average system performance. Because prefetching is a speculative technique it might also lead to performance regressions on some workloads. In Figure IX.3, we show the lowest speedup for any core on any workload for all the prefetch scheduling strategies. Overall, we observe that the prefetchers with low accuracy shows the largest performance regressions. The strategies utilizing prefetch accuracy measurements (*Low cost*, *Opportunistic*) perform quite well as they are mostly able to adapt to this situation, thus avoiding large performance regressions. The *Direct* strategy shows quite large performance regressions because it does not differentiate between prefetches and demand reads. Thus, a prefetcher which issues many useless prefetches can saturate off-chip bandwidth and delay demand reads. The *Idle* strategy has comparatively low performance regressions, because the strategy is inherently defensive and only issues prefetches when there

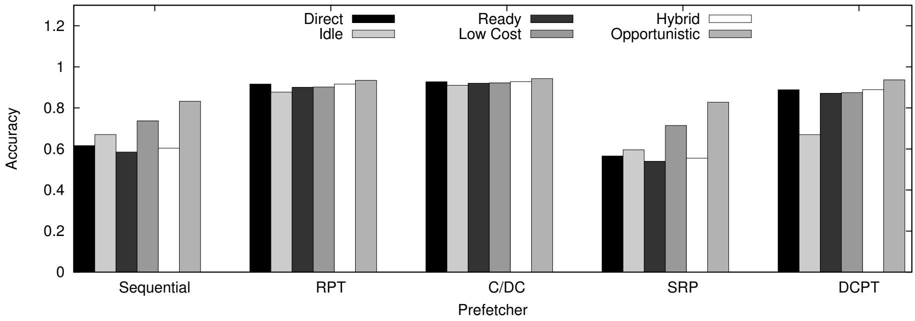


Figure IX.4: Average accuracy for all workloads.

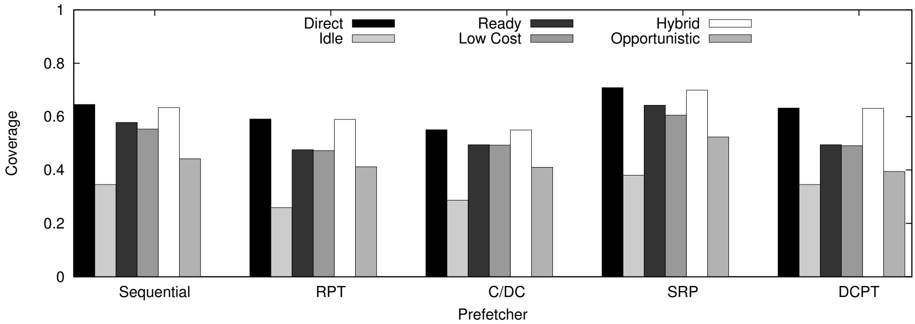


Figure IX.5: Average coverage for all workloads.

are no demand reads in the queue. *Hybrid* uses accuracy estimates to select between two strategies (*Direct* and *Ready*). By increasing the threshold, the behaviour of this strategy can be made more similar to the *Ready* strategy.

IX.5.3 Accuracy and Coverage

Figure IX.4 shows the average accuracy of every workload in each combination of prefetcher and prefetch scheduling strategy. Note that only prefetches that have been issued by the memory controller are shown in this figure. This is not necessarily the same as the accuracy of the prefetcher because the prefetch scheduling strategy may drop prefetches. However, the *Direct* strategy does not drop prefetches and issues all prefetches generated by the prefetcher. By examining the results for sequential and SRP, we observe that *Idle*, *Low Cost* and *Opportunistic* are able to achieve higher degrees of accuracy than the *Direct* approach. In the high accuracy prefetchers, there is little difference in the scheduling strategies.

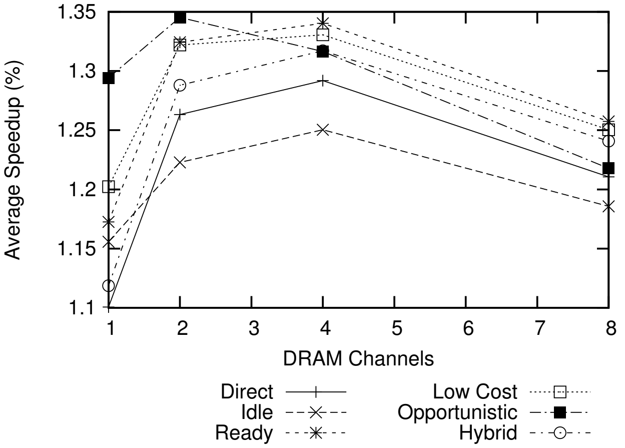


Figure IX.6: Effect of increasing the amount of bandwidth available on sequential prefetching.

Figure IX.5 shows the average coverage of every workload in each combination of prefetcher and prefetch scheduling strategy. Because the *Direct* strategy issues every prefetch generated by the prefetcher, it has a very high coverage. Conversely, *Idle* has very low coverage because it issues few prefetches. *Opportunistic* has a comparatively low coverage compared to the other prefetch scheduling strategies because it issues fewer prefetches than the other strategies. The *Hybrid* strategy is very near the performance of the *Direct* strategy in terms of coverage. This is due to the value of the accuracy threshold used in our experiments. *Low Cost* has a slightly lower coverage than *Ready* for the low accuracy prefetchers (sequential, SRP), because it drops prefetches when the accuracy becomes too low.

IX.5.4 Increasing DRAM Bandwidth

Figure IX.6 and IX.7 shows the effect of increasing the amount of bandwidth in the system for sequential and RPT prefetching respectively. Note that the speedup is computed versus a system with the same amount of bandwidth but with no prefetching. Thus, the speedup for 8 DRAM channels is lower than for 1 DRAM channel, although the performance is higher. For sequential prefetching, we observe that the relative performance of *opportunistic* versus the other prefetch scheduling strategies is highest in low bandwidth situations. Furthermore, we observe that *Idle* performs well compared to the other scheduling algorithms with one channel but worse when more bandwidth is available. The reason is that *Idle* issues prefetches cautiously which makes it less likely to create congestion.

For RPT prefetching we observe a similar pattern as the amount of bandwidth is increased. RPT is a high accuracy prefetcher and the aggressive strategies such as

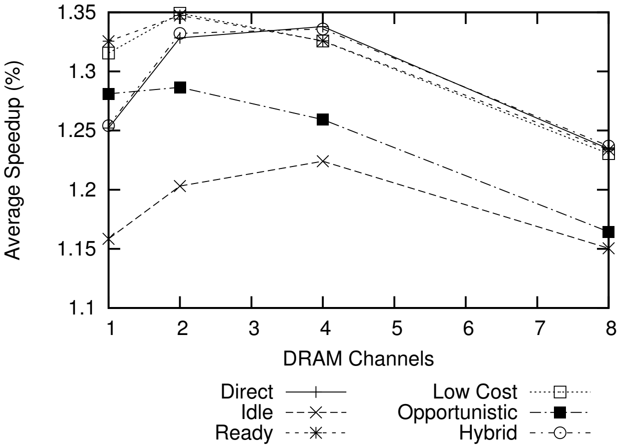


Figure IX.7: Effect of increasing the amount of bandwidth available on RPT prefetching.

Direct, *Low Cost*, *Hybrid* and *Ready* performs better when bandwidth is increased compared to *Idle* and *Opportunistic.*

IX.6 Discussion

Three of the strategies (*Low Cost*, *Hybrid* and *Opportunistic*) examined in this paper utilizes prefetch accuracy measurements. All of these use a threshold value to switch between two strategies. Typically, if accuracy is high, then an aggressive strategy is used. Conversely, if accuracy is low, then a more restrictive strategy is used. Thus, the behaviour of these strategies can be adapted by changing the threshold value. In this paper, we have used the same threshold value for all three strategies for easier comparison. The value we have chosen matches the value used in previous work [4, 8].

In this work, we use the same method for estimating prefetch accuracy for all strategies. This method uses special bits in the cache to mark prefetched cache blocks (cache tagging). However, *Opportunistic* prefetch scheduling offers another method. All prefetches generated by the prefetcher are stored in the PVT, while the cache tagging method only stores prefetches that have been issued and completed. Thus, the cache tagging method measures the combined accuracy of the prefetcher and the scheduling strategy, while the PVT isolates the accuracy of the prefetcher. We have examined this method and found that it offers slightly better performance. However, the threshold value has to be changed, because this method measures something different (issued prefetches accuracy vs. generated prefetches accuracy). Therefore, we have opted to use the same estimation technique for all prefetchers to achieve a fair comparison.

IX.7 Conclusion

It is clear from our results that no single prefetch scheduling strategy is suitable for every scenario. The best strategy depends on a variety of factors such as: the prefetcher, the memory controller, the amount of memory bandwidth, the application, design complexity and the amount of acceptable performance regressions. For instance, *Idle* is a good option for minimizing performance regressions. On the other end, *Low Cost* and *Ready* provides the highest average performance. *Opportunistic* provides a trade-off between these two. Because it actively reduces performance regressions, it also provides the highest average performance for sequential and SRP prefetchers.

In this paper, we have presented a novel prefetch scheduling strategy called *Opportunistic*. This strategy emphasises the use of open pages to provide good average performance without large performance regressions. It is particularly suited for systems with relatively low amounts of bandwidth combined with highly aggressive prefetchers. As more cores compete for the same shared off-chip bandwidth, utilizing this limited resource becomes more important. *Opportunistic* prefetch scheduling addresses this problem by utilizing open pages to increase effective bandwidth, while using accuracy estimates to avoid bandwidth saturation. We show that *Opportunistic* prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. *SIGPLAN Not.*, 41(11):274–282, 2006.

[3] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.

[4] M. Grannaes, M. Jahre, and L. Natvig. Low-cost open-page prefetch scheduling in chip multiprocessors. In *XXVI IEEE International Conference on Computer Design (ICCD)*, 2008.

[5] M. Grannaes, M. Jahre, and L. Natvig. Storage efficient hardware prefetching using delta correlating prediction tables. In *Data Prefetching Championships*, 2009.

[6] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2004.

[7] *DDR2 SDRAM Specification.* JEDEC Solid State Technology Association, May 2006.

[8] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008.

[9] W.-F. Lin, S. K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50 (11):1202–1218, 2001.

[10] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[11] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI '04: Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 80–87, 2004.

[12] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Micro, IEEE*, 25:90–97, Jan. 2005.

[13] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[14] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[15] J. Shao and B. Davis. A burst scheduling access reordering mechanism. *High Performance Computer Architecture, IEEE 13th International Symposium on*, pages 285–294, 2007.

[16] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. *SIGARCH Comput. Archit. News*, 34(2):252–263, 2006.

[18] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[19] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 213–224, 2005.

[20] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. *Eighth International Symposium on High-Performance Computer Architecture, 2002*, pages 107–116, 2002.