NTNU
Innovation and Creativity

Bandwidth-Aware Prefetching in Chip Multiprocessors

Marius Grannæs

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Lasse Natvig, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

A promising area of computer architecture research is chip multiprocessors (CMP) — also called multicore architectures. A CMP is an architecture where multiple cores are embedded into a single chip. These new processors typically have one or two levels of cache memory that are private to each processor, and one common shared L2 or L3 cache. The processors on the chip also share the communication channel to the off-chip memory, thus they compete for several shared resources. On the other hand, fetching of instructions or data from external memory or storing in the shared cache are operations that potentially may help other processors. This potential balance between negative competition and positive cooperation gives new challenges for prefetching. Performance counters that monitor the usage of the various architectural resources will probably be important for implementing efficient CMP prefetching.

Marius Grannæs completed in the course TDT4720 Computer Design and Architecture, Specialization the project entitled " Simulation of Hardware Based Prefetching in SimpleScalar". The project was limited to uniprocessor architectures, it presented a simulator framework and some initial experiments on prefetching. The goals for the diploma work are:
* Experiments evaluating more types of prefetching methods
* An extension of the simulator framework for studying of prefetching in relevant CMP architectures
* Experiments evaluating some of the most common prefetching methods in CMPs
* Studies of the topic "performance counters" in this context
* If time allows, development and evaluation of a new CMP prefecthing algorithm based on performance counters

Assignment given: 19. January 2005
Supervisor: Lasse Natvig, IDI

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Each year exponentially more transistors are put into integrated circuits [1, 2]. Moore's law is the empirical observation that at our rate of technological development, the complexity of an integrated circuit, with respect to minimum component cost, will double every 18 months [3]. Increased transistor density, in turn, translates into faster computers for consumers. In addition, architectural advances in microprocessor design has contributed to increased performance.

CPU performance increased by 35% per year until 1986 and by 55% per year after 1986 [4]. DRAM density has also increased at approximately the same rate. In terms of latency, DRAM has not seen such huge improvements (only 7% per year). This is shown in figure 1.1. Because of this speed difference, main memory cannot keep up with the processor anymore. Thus the processor frequently stalls while waiting for data from memory. This problem is known as the "memory wall" or "memory gap" [5].

In a recent (march 2006) article by the ACM president, David Patterson argues that performance of microprocessors have only increased by 20% per year since 2002. This is due to three separate causes:

- The lack of additional power for a chip to dissipate.
- The lack of additional instruction-level parallelism to exploit.
- The lack of improvement in memory latency.

Decreasing main memory latency is difficult, because there is a lower bound on the minimum latency possible. Ultimately, main memory latency has a lower bound determined by the speed of light and the distance from the memory modules to the CPU. At 4GHz, a signal can only travel 7.5cm per clock tick. This causes a natural bound for the lowest possible main memory latency, as traditional main memory is off-chip. Given such a limit, approaching it becomes ever more difficult. However, given current electronic designs, the RC-delays become the dominant contributing factor to latency. Finally, there is a practical limitation caused by economics [7], because increased capacity sells chips, while decreased latency does not.

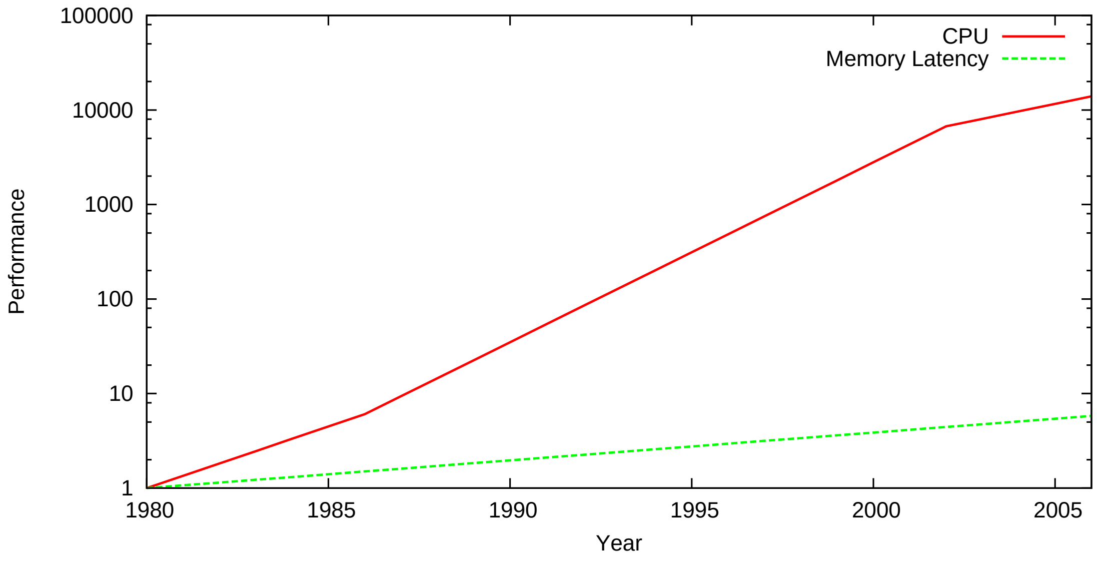

Figure 1.1: Development of CPU performance versus memory latency [4, 6].

1.2 Reducing Latency

It is likely that the memory gap will not be closed in the foreseeable future and it is therefore important to develop techniques that can either tolerate or decrease latency. Numerous architectural techniques have been developed to compensate for this gap. These include caches, out-of-order execution, chip-multiprocessing, simultaneous multithreading, run-ahead execution, bypassing and prefetching. In addition, some have looked at integrating main memory into the processor. The IRAM project [8] is one such initiative. This technique reduces the gap by decreasing the physical distance the signals will have to travel. In addition, by integrating main memory in such a way increased bandwidth also becomes available.

Cache is so far the most successful technique used to bridge the gap. Conceptually, caches are smaller, but faster memories that store the most often used data [4]. Thus most of the time, the data required by the processor is in the cache, and there is no need for a stall. Everything that is fetched from main memory is stored in the cache, possibly displacing other data. Whenever data referenced by the processor is not present in the cache, these data must be fetched from main memory. Such cache misses reduce the effectiveness of caches.

Out-of-order execution is a technique commonly used in modern processors to increase throughput. Basic out-of-order execution allows instructions to execute in another order than that specified by the program. This is done through a dynamic analysis of the instruction stream (scoreboarding [4]). Allowing other instructions to execute while another instruction waits for memory increases performance considerably. However, because the instruction window is bounded in size, there are limits for how much latency out-of-order execution can hide. Effectively, out-of-order execution hides L1 misses, but cannot hide L2 misses. Thus it is the most effective if most of the code and data reside in the cache.

Run-ahead execution is a hot research topic [9, 10]. Basic run-ahead execution allows programs to continue speculatively while a memory stall is in effect. By predicting the value of the returned data, the thread can continue running. When the actual value is received from memory, it is compared to the predicted value. If the two values match, the run-ahead was a success and can continue. However, if they do not match, a micro-architectural roll-back have to be done. Such a roll-back will have to restore all registers and flush the pipeline.

Simultaneous Multithreading (SMT) allows other threads to execute while one thread waits for memory, which increases system throughput. In effect, multiple instruction streams utilize the same core at the same time. This technology is among others used in Intel's "Hyperthreading" technology. Normally, two threads executing on a core execute at the same time, where the scheduler picks instructions from each thread in a round-robin fashion. If for some reason, one thread becomes stalled, then the scheduler only submits instructions from the non-blocked thread.

Prefetching is a technique used to increase the effectiveness of caches by trying to predict the memory reference stream. By fetching needed data to the caches before it is actually referenced by the processor, it is possible to achieve a significant performance increase. This technique will be explored in more detail in the remainder of this thesis.

Bypassing is a similar technique to prefetching, however, it's purpose is different. The goal of bypassing is to identify data that will not be reused (it has no *temporal* locality) [11]. Thus, the data do not need to be stored in the cache, effectively *bypassing* the cache. Heuristics used for bypassing share many characteristics with prefetching.

Chip Multiprocessors pack multiple processors into a single chip. Thus cores are able to share caches, which in turn makes it easier for two separate threads to cooperate through shared memory. Since programs often share code in the form of shared libraries (such as libc), sharing such libraries in the cache saves a significant space. By having multiple cores, the focus is shifted away from single thread performance (where latency is important) to throughput (where latency is less important). In essence; by having multiple cores (which can possibly be SMT), the effect of stalling one single thread is much less severe. Chip Multiprocessors is an interesting field, because it opens up a variety of new possibilities in terms of architecture (this will be explored further later in this thesis). In addition, most commercial vendors of high-end microprocessors are currently offering CMPs. Lawrence Spracklen and Santosh G. Abraham outline the challenges in CMP in their paper "Opportunities and challenges" [12]. One of those challenges is to adapt prefetching to CMP.

It would be too ambitious to try to solve this problem within the time frame of this thesis. This project will approach this challenge by using performance counters to direct prefetching. Nonetheless, as this thesis is a part of a larger PhD thesis, it will also be used as a foundation for further research.

1.3 Subgoals

The main purpose of this thesis is to investigate prefetching in a chip multiprocessor setting. The second purpose is to explore the use of performance counters with regard to prefetching. To achieve these goals, several objectives have been formulated:

- Investigate performance counters in modern processors.
- Investigate modern prefetching methods.
- Expand the simulator (SimpleScalar) from the fifth year project to include more types of prefetching.
- Expand the simulator to include a more realistic DRAM model.
- Expand the simulator to simulate CMPs.
- Develop a methodology to benchmark CMPs.
- Develop an understanding of current prefetching heuristics by conducting experiments.
- Develop an understanding of known prefetching heuristics in a CMP setting by performing simulations.
- Development of a *new* prefetching algorithm based on performance counters.
- Document it's performance through experimentation.

1.4 Fifth Year Project

This thesis is a continuation of my fifth year project. In that project I studied three prefetching algorithms in a uniprocessor context. I developed a framework for handling prefetching in SimpleScalar as well. SimpleScalar is a widely used cycle-accurate microarchitectural simulator. The prefetching heuristics that were already implemented at the beginning of this project are:

- Sequential prefetching
- Delta Correlation prefetching
- CZone/Delta Correlation prefetching

These heuristics are described in section 2.2.2, and will also be used in this thesis.

In addition, much of the theoretical studies regarding prefetching was done in the project. In this thesis I will reuse the framework that I made in the project as well as the three reference algorithms.

1.5 Scope of this Thesis

Prefetching as well as CMP architecture are very large fields. This section limits the scope of the thesis.

Prefetching can be done both in hardware and in software. In software, one can either use special "prefetch" instructions, or generate hints to hardware about possible prefetching opportunities. This thesis will only concern pure hardware controlled prefetching, although some theory around software prefetching will be given to provide context.

CMP offer a whole range of different types of architectures (see section 2.4). In this thesis I will only look at one fixed architecture, due to it's popularity in commercial settings. The cores are separated with a private L1 cache, but share a L2 cache as well as a memory controller.

1.6 Structure of this Document

In chapter 2 I present background material needed to understand this thesis. This chapter contains the necessary information about DRAM, caches, chip multiprocessors, SimpleScalar and prefetching. In chapter 3 the methods used to conduct the experiments are presented. My own extensions to SimpleScalar as well as the benchmarks used are presented and analyzed. Chapter 4 contains the results from my experiments. It is organized in two distinct sections; First, the prefetching heuristics are tested in a uniprocessor environment, in order to establish an understanding of the different prefetching heuristics. Then the heuristics are run in a CMP context to see how it affects prefetching. Chapter 5 discuss the results and I conclude and present some future work chapter 6.

Chapter 2

Background

In this chapter I will describe the theory behind caches, prefetching, DRAM, Chip Multiprocessors and performance counters. Furthermore, I will describe some of the tools that I will use, such as SimpleScalar and CACTI.

2.1 Caches

Before talking about prefetching and CMP, a basic understanding of caches is needed. Caches are used to bridge the memory gap through duplicating data in smaller and faster storage [13]. A cache can be made in many types of technology, but is usually SRAM (while main memory is DRAM). This is done as part of a memory hierarchy (see figure 2.1).

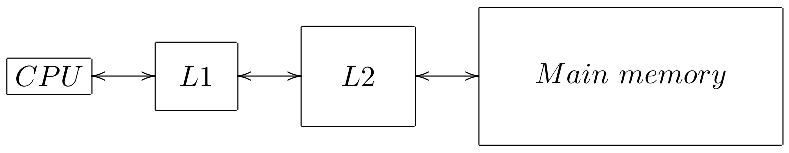

Figure 2.1: Example of a memory hierarchy with 2 levels of cache.

There might be several levels of caches in the memory hierarchy. In my simulations I have opted for a 2-level cache system. Registers are the fastest type of memory, and there is no programmer-visible latency associated with using them. There are usually only a few registers available. The L1 cache is somewhat larger, usually a few kilobytes large. Although its latency varies from processor to processor, it is usually between 1 and 4 clock cycles. The L2 cache is much larger, although smaller than main memory. The amount of L2 cache varies, but is usually measured in megabytes. Its latency is around 20 clock cycles. In comparison, main memory usually has a latency of several hundred clock cycles.

Caches work by exploiting the *spatial* and *temporal locality* seen in memory references [4]. *Spatial* locality refers to the property that data close together in address space tend to be referenced around the same time. *Temporal* locality refers to the property that data that has been referenced is very likely to be referenced in the near future. Caches work by storing every referenced data. Thus, temporal locality is exploited. A whole cache line (usually around 128 bytes large) is brought into the cache each time any part of the line is accessed, thus spatial locality is exploited.

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 cache memory to main memory is needed. Cache can be organized in three major ways:

- *Direct mapped* - A cache line can only be placed in one position based on its address.
- *Fully associative* - A cache line 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 lines.

The most common type is the set associative cache as it provides a compromise between the two extremes. A direct mapped cache would be extremely expensive to implement (in terms of area) when the cache size becomes large as it requires comparison logic for every entry in the cache. A direct mapped cache provides very little flexibility and the possibility of collisions in the working set becomes large. The set associative solution is preferable, because it does not require a lot of comparison logic and because a cache line can be put anywhere in the set, reducing conflicts.

In a set associative cache, the cache lines are organized into sets. Each set can hold n cache lines. If there are n cache lines (or *blocks*) in a set, the cache is called n-way set associative. In addition, each cache has several sets. Data can only map onto one set in the cache, but it can map onto several cache lines in the set, depending on the replacement policy in use.

When a new cache line is put into the cache, an old one is evicted. There are several possible replacement policies available, least-recently-used is the most common one, although, FIFO and other variants are possible.

There are several reasons why some data might not be in the cache. It is useful to categorize these reasons into groups, so that one can more easily reason about them. According to Hennessy and Patterson [4] there are three major categories of misses:

Definition 1 (Compulsory)**.** The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. These are also called *cold-start misses* or *first-reference* misses.

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

Definition 3 (Conflict)**.** 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.* The idea is that hits in a fully associative cache that become misses in a an n-way set-associative cache are due to more than n requests on some popular sets.

These definitions will be used when reasoning about prefetching.

2.1.1 Cacti

CACTI [14] is an advanced cache model capable of modeling the timing, power requirements and area of any given cache. CACTI helps computer architects to understand the trade-offs between power, area and timing. The model used is very complex and considers most of the available design-techniques. I will use this tool to evaluate different cache designs, especially to see the trade-offs between timing and size.

As an example I have run the program to demonstrate the fidelity of the model. The cache being simulated is a 4-way 8KB cache with 64byte cache lines. It has 1 read and 1 write port, and the technology being used is 65nm. The complete output spans several pages, and can be found in appendix A. An explanation of every field can be found in the technical report[14].

The most important numbers from this model for our purposes are the following:

```
Access Time (ns): 0.57591
Cycle Time (wave pipelined) (ns):  0.273554
Total Power all Banks (nJ): 0.139284
Total area    1.074315 (mm^2)
```

If we double the number of read and write ports (which would be required if it was a shared cache), we get the following output:

```
Access Time (ns): 0.729833
Cycle Time (wave pipelined) (ns):  0.361345
Total Power all Banks (nJ): 0.285296
Total area    2.400975 (mm^2)
```

If we use combined read/write ports instead of separate ones we get the following data: For one R/W-port:

```
Access Time (ns): 0.57591
Cycle Time (wave pipelined) (ns):  0.273554
Total Power all Banks (nJ): 0.139284
Total area    1.074315 (mm^2)
```

With two R/W ports:

```
Access Time (ns): 0.658765
Cycle Time (wave pipelined) (ns):  0.328369
Total Power all Banks (nJ): 0.233175
Total area    1.889209 (mm^2)
```

In general, increasing the number of ports of a cache, increases its latency by a small amount and its area is increased substantially. This becomes quite significant when using a shared cache in CMP designs. It is also important to note that the energy requirements becomes much larger when using a multiported cache. It is clear that giving each processor its own R/W port does not scale. Fortunately, bigger caches are divided into banks that are independent of each other, so that parallelism can be achieved without increasing the number of ports. However, it is worth remembering these numbers when discussing shared L2 caches in the upcoming chapters.

2.2 Prefetching

Prefetching is a speculative technique that is used to fetch data from main memory to the cache before being referenced by the processor. Its main purpose is thus to reduce the amount of *compulsory* misses. In order to do so, one must accurately predict future references to memory. Numerous heuristics have been developed, and I will present only a subset of these heuristics in this section.

By using various heuristics a proper prefetching algorithm is able to predict exactly what data is needed by the processor in advance. Because main memory latency is comparatively large (200 clock cycles), this prediction must be made at least the same amount of time in advance. If the prediction is correct the needed data will be in the cache and the processor will avoid a costly stall.

Prefetching does come with a cost. Prefetching a cache line can potentially have two separate ill effects. First, additional bandwidth is used to fetch the prefetched data. This can potentially delay other memory requests that is on the programs critical path of execution. In addition, accessing external memory is expensive in terms of energy. This can become quite significant in hand-held devices. Finally, the third effect is that by fetching a cache line another cache line has to be replaced. The replaced cache line might contain data that is needed in the near future. If the data has been modified (the dirty bit is set), the cache line will need to be written back to memory, thus increasing bandwidth usage.

To reason about prefetching a set of definitions are needed. The following definitions are taken from Srinivasan “A Prefetch Taxonomy” [15].

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

Definition 5 (Accuracy)**.** If a conventional[1] cache has M misses without using any prefetch algorithm, the accuracy of a given prefetch algorithm that yields G good prefetches and B bad prefetches is calculated as:

$$Accuracy = \frac{G}{G+B} \tag{2.1}$$

Definition 6 (Coverage)**.** 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:

$$Coverage = \frac{G}{M} \tag{2.2}$$

The following definitions are taken from VanderWiel and Lilja’s “A survey of Data Prefetching techniques” [16].

Definition 7 (Prefetch distance)**.** If a loop contains small computational bodies, it may be necessary to initiate prefetches δ iterations before the data is referenced where δ is know as the prefetch distance and is expressed in units of loop iterations:

$$\delta = \left\lceil \frac{l}{s} \right\rceil \tag{2.3}$$

[1]A conventional cache in this context is a cache without prefetching

Where 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.

Definition 8 (Prefetch degree)**.** 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.

A prefetching heuristic with a high *accuracy* is a heuristic that generates few needless memory accesses. A prefetching heuristic with a high *coverage* is a heuristic that generates few misses in the cache (and thus high performance). A good prefetching algorithm needs high accuracy and high coverage. A heuristic that has a high accuracy is trivial to create. Most of the algorithms described later can be made extremely conservative and thus increase their accuracy. Coverage can be increased in a similar manner by using more aggressive prefetching. Aggressive prefetching increases the number of prefetches issued. Such an approach is impractical as bandwidth limits it's usefulness.

As a primitive benchmark, prefetching is often compared to a "perfect" cache. A "perfect" cache is a cache that always havs the requested data. In other words, there is no difference between main memory and cache (except for a much lower latency). This value represents a theoretical upper limit for the effectiveness of prefetching. I will use this metric as well.

It is useful to distinguish between data and instructions when discussing prefetching as they exhibit different patterns. Instruction prefetching can be done very accurately by using the information already present in the branch predictor [17, 18]. The information in the branch predictor can be used to predict what instructions will be executed (the program path), and thus prefetch the instructions. Spracklen has shown that there are large performance gains available by using instruction prefetching on CMPs [19].

Data prefetching is somewhat more complex as data can be accessed in various ways and can be structured. Consider a single value, a constant, an array, a data structure, a pointer. Each of these types would likely behave in different manners. Different programming constructs produce different memory access patterns as illustrated in listing 2.1.

On line 2, a simple assignment is performed. This is a scalar pattern. x is read once and then program execution moves forward. A simple loop (as in lines 5-7) produces a sequential pattern. In this code snippet the array is traversed in a sequential manner by incrementing the array index. Lines 10-12 is similar, however, only every third data element is accessed. This pattern is named "strided". The fourth pattern (lines 15-17) a simple pointer chasing snippet. In this example, a linked list is traversed to the end by using the "next"-field of the individual data items. This construct can often be found when a program needs to traverse a list. The last example is somewhat constructed. A random function points into an array. Predicting this case would be hard. However, such code does exist in the form of jump tables and look-up tables. Of course, mixes of the above patterns also exists. The different types are summarized in table 2.1.

Caches exploit spatial locality(see section 2.1) by fetching entire cache lines (128 bytes) even though only a single byte is actually needed. This is a limited form of prefetching and has been shown to be very successful. In the next two sections I will examine prefetching methods in more detail.

```
/* Scalar pattern */
foo = x;

/* Sequential pattern */
for (index = 0; index < 100; index++) {
  foo = foo + array[index];
}

/* Strided pattern - Stride = 3 */
for (index = 0; index < 100; index = index + 3) {
  foo = foo + array[index];
}

/* Pointer chasing in a linked list*/
while (p !=NULL) {
  p = p->next;
}

/* Irregular */
for (i = 0; i < 100; i ++) {
  foo = foo + array[random()]
}
```

Listing 2.1: Example memory patterns

2.2.1 Software Prefetching

Although software prefetching is outside the scope of this thesis, the following background material is included for completeness. Software prefetching is a large group of prefetching techniques. There are two subcategories of software prefetching; explicit prefetching and software hinting. Explicit software prefetching [20] consists of prefetching instructions in the program. As an example a program might issue a prefetch for a part of an array while it is processing another part. Such methods have proven very useful in pointer-chasing programs [21]. Most modern processors have support for explicit software prefetching through special instructions. In addition, most modern compilers have built-in support for prefetching through intrinsics. An example of using intrinsics in GCC can be seen in listing 2.2. Note that this code will compile even on platforms that do not support software prefetching (the directive will simply be ignored).

Modern compilers also have built-in support for automatically inserting software prefetches. GCC has support for prefetching loop arrays, although more advanced prefetching features are planned. In addition several research compilers have support for even more advanced techniques [21], such as pointer prefetching.

The downside of software prefetching is that it introduces additional instructions into the program. This increases the program size, thus decreasing instruction cache performance. In addition, the prefetch instructions will have to run through the entire pipeline as any other instruction. Thus a prefetching instruction occupies space in the pipeline

Type	Description	Example Pattern Memory addresses
Scalar	A simple reference to a single scalar value	42
Sequential	A memory pattern where the address is incremented	42, 43, 44, 45, ...
Strided	A memory pattern where the address is increased by a value larger than one	42, 45, 48, 51, ...
Pointer	No pattern, but effect is due to pointer chasing	42, 3, 18, 7, ...
Irregular	None of the above	97, 2, 1034, 7, ...

Table 2.1: Types of memory access patterns.

```
for (index = 0; index < 100; index = index + 3) {
  /* Prefetch the next element in the array */
  /* In real code the offset will be larger than 3 */
  __builtin_prefetch(array[index+3]);
  foo = foo + array[index];
}
```

Listing 2.2: Prefetching in GCC using intrinsics

that could possibly be used for actual computation.

Hinting is a cooperative method were special instructions in the processor guide the prefetching hardware. There exist numerous such schemes. Guided region prefetching [22] is one of them. The programmer (or compiler) issues hints to the prefetching hardware about regions in memory. As an example, one such message might be “The data in this location is a large array”. The prefetching hardware can then use this information to predict the behavior of the program. This method is often used in conjunction with traditional prefetching methods.

2.2.2 Hardware Prefetching

Pure hardware based prefetching is transparent to the programmer and the compiler. The programmer cannot directly modify the behavior of the hardware prefetching unit. Therefore the prefetching unit will have to detect memory reference patterns at run-time. Compared to software prefetching, this can have both advantages and disadvantages. Hardware prefetching does not have time to analyze the program at a larger scale. It does have dynamic information that is not available to the compiler. Thus a prefetching algorithm can cooperate with other parts of the processor to examine likely paths of execution. In addition, a hardware prefetching unit can be made completely independent of the processor core, avoiding slowdown on the critical path.

Hardware prefetching schemes range from the very simple to the very complex. As with any architectural technique, choosing the right one is a trade-off between power consumption, design complexity, area and performance.

The following sections discuss different prefetching algorithms and are not meant to be a complete reference. Moreover, the following schemes are implemented in the simulator.

Sequential prefetching

Sequential prefetching is a very simple scheme. Whenever a program access cache line X, the system prefetches line X+1 [13]. This is an effective approach exploiting spatial locality, the same principle used by caches. Studies have shown that it can be very effective in a range of programs [23]. There exist two variants of this algorithm. The first variant prefetches whenever there is an access to the cache memory, the second variant prefetches only if it was a miss. The performance of this type of prefetching is documented in my fifth year project.

DC prefetching

Delta correlation prefetching is an algorithm that tries to detect patterns in the miss stream. It stores the miss stream in a Global History Buffer (GHB) [24, 25]. This buffer stores the address of every miss to the L2 cache in a FIFO buffer of fixed size. This has a distinct advantage to other storage techniques in that stale or old data will be discarded first.

On a miss, the delta (the difference between the current address and the previous) is calculated and stored. The GHB is then traversed to find a match of the two deltas.

Miss address:	40		44		46		48		52		54
Delta:		4		2		2		4		2	

Table 2.2: Delta correlation in a GHB, newest miss to the right.

An example can be seen in table 2.2. In this table the miss stream is represented by the addresses that causes misses in the L2 cache. Time increases to the right. The two last misses are to address 54 and 52, thus delta equals 2. The previous delta was 4. The GHB is then scanned backwards for this pair of deltas (4,2). In this example, this pair is detected in the end of the GHB. Then, a prefetch is issued for the current address + the delta *after* the pair. In this case a prefetch would be issued for $54 + (48 - 46) = 56$. If the prefetch degree was larger than 1, the next address would be : $56 + (52 - 48) = 60$.

CD/C prefetching

CZone / Delta Correlation is a newer variant of the delta correlation prefetcher. It was invented by Kyle Nesbit [25].

The idea is that processors have different access patterns to separate regions in memory. For example, the stack might be stored in one area of memory and will show a distinct memory reference pattern. An array might be stored in another area, and may exhibit sequential memory reference behavior. Such areas are called CZones and are usually of a fixed size.

To exploit this, C/DC first stores every miss in a GHB (such as the DC scheme). When a new miss occurs, C/DC determines which CZone it corresponds to. Then every other access in the GHB that corresponds to the same CZone is put into a separate buffer. Then the Delta correlation algorithm is used on this buffer.

Figure 2.2 shows a conceptual model of this type of prefetcher. A table indexes the different CZones with pointers to the global history buffer. This buffer is then traversed while the correlation units looks for matching patterns. A correlation unit is logic that performs the matching described under DC prefetching.

If the size of each CZone is equal to the size of the program, this type of prefetching is reduced to the special case of Delta Correlation prefetching.

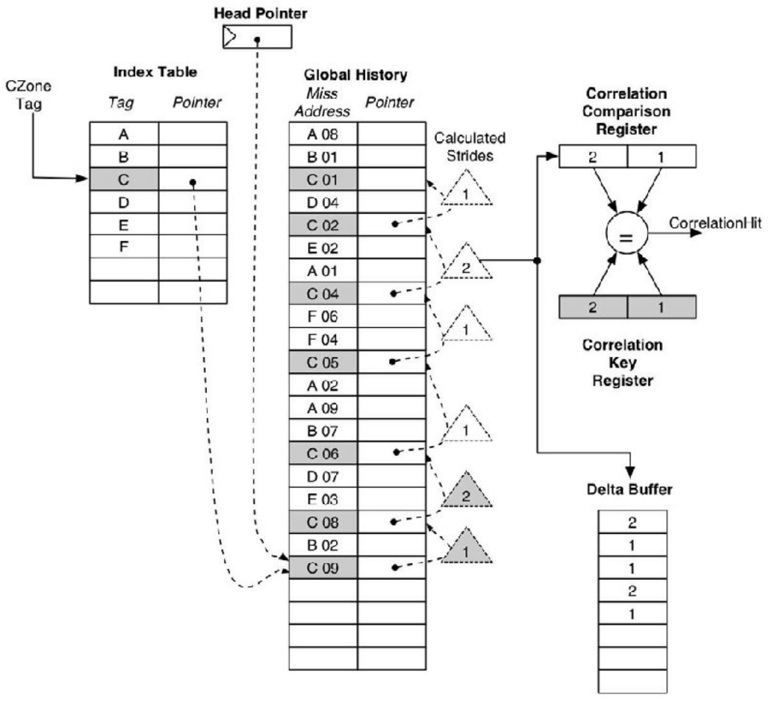

Figure 2.2: CZone / Delta Correlation operation. This diagram is taken from [25].

AVD prefetching

Prefetching dynamic data structures such as linked lists is a difficult task. The first problem lies in identifying what data is pointers to other data. The other problem is timeliness, identifying pointers in such a manner that the prefetch is issued a significant amount of time before the data that is pointed to is actually used. Thus chains of pointers will have to be referenced.

Prefetching pointers is an active research field, and while some progress has been made, no simple, efficient method has yet been devised. The problem is that there is nothing regular about dynamic datastructures. Because they are based on pointers, data that are logically sequential do not have to be sequential in memory. To make matters worse, the memory allocator or garbage collector might move or compact data.

Tag	Prev_addr	Stride	State
43	130	0	Initial
54	126	2	Transient
67	512	10	Steady

Table 2.3: Example Reference Prediction Table.

To identify pointer loads, numerous schemes exists, the simplest one is to match the upper N bits of the address to the value that has been loaded. If the value matches the address, it is considered a pointer load. The idea is to catch near pointers; It is quite improbable that the upper N bits match if the loaded data is simply actual data. If it is a near pointer, the upper bits will probably match. This method is often referred to as Address-Value Delta [26, 9].

The downside of this kind of prefetching is that one has to wait until the data returned from the previous load is available, thus reducing timeliness. Another problem is that the algorithm can only detect “near” pointers. At last, prefetching pointer chains becomes very hard, thus reducing the maximum amount of aggressiveness available.

Other methods track pointer loads in special structures by indexing the load instructions themselves [27, 28].

PC-based methods

As can be seen from listing 2.1 different load-instructions behave differently, some loads will only be executed once, some are part of a tight inner loop, while other load pointers. By tracking how different load instructions behave in a separate table, more information becomes available for prefetching [29].

Tien-Fu Chen and Jean Loup Baer propose a scheme where this information is stored in a reference prediction table (RPT). The RPT is a cache-like structure that stores information about loads based on the address of the load.

An example table is shown in table 2.3. When a load instruction is first encountered it is entered in the table with its tag and the loaded address, stride is set to 0 and the state is “Initial”. If a load instruction is already in the table, the difference between prev_addr and the loaded address is calculated. If the difference is equal to the value in the stride field, prev_addr is updated and a state transition along the “Correct” path according to figure 2.3 occurs. It they do not match, the stride field is updated, prev_addr is updated and a state transition occurs (labeled “incorrect”.)

Dahlgren and Stenström experimented with a version of this scheme with 3 states [23].

Stream prefetching

Stream prefetching is not a method for detecting prefetching patterns, but is a method for actually issuing prefetches. It is intended to be used in conjunction with any of the previous prefetching heuristics. The basic principle is simple; A stream is detected by one of the previous algorithms and a series of prefetching addresses is generated. The first address is prefetched into the L1 cache, the next addresses are prefetched into the L2

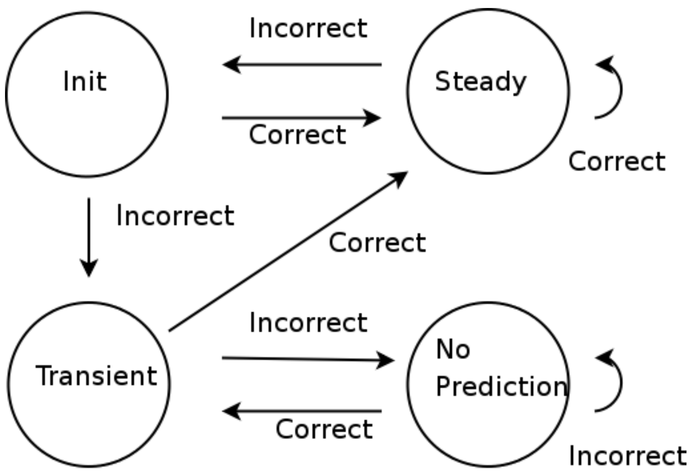

Figure 2.3: State diagram for Chen and Baer's reference predictor.

cache. If a L3 cache is present, some of the prefetches are issued to the L3, cache. Thus a stream is formed [30]. In the beginning, this will not be very efficient as the most of the data will lie in main memory and will not be spread out into the memory hierarchy. Once the steady state is achieved, the first line to prefetch will be held in the L2 cache, and the next lines will be in the L3 cache, thus a large performance benefit can be achieved.

An illustration of the stream-principle can be found in figure 2.4. This type of prefetching is used in the Power4 and 5 by IBM [31, 32] among others. In this work I will use the streaming technique in conjunction with the RPT heuristic.

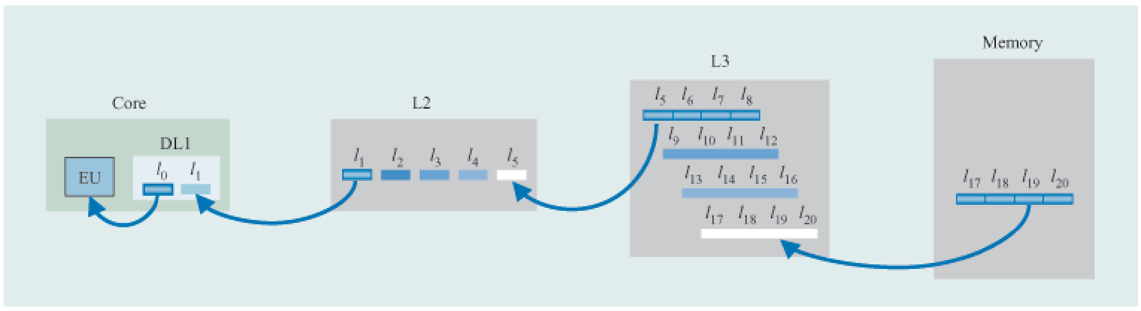

Figure 2.4: Stream prefetching in the Power4 architecture [31].

2.3 DRAM

DRAM (Dynamic Random Access Memory) is the most common form of main memory in computer systems today. This technology has seen tremendous growth in capacity as feature size decrease. This development has also enabled designs with higher bandwidth, whereas latency has not seen such big improvements[7], leading to problems that is known as “The memory wall”[5].

This work is not primarily concerned with DRAM implementations, but requires an accurate model of the memory subsystem to make accurate simulations. Thus it is required to develop an understanding of this system, in order to implement a good model. In order to do so, I will look at some of the different technologies in use today, and with this knowledge develop a model that can be used with the simulator.

DRAM has three major components: The memory core, the bus interface and the memory controller. See figure 2.5. In the following sections I will describe each component.

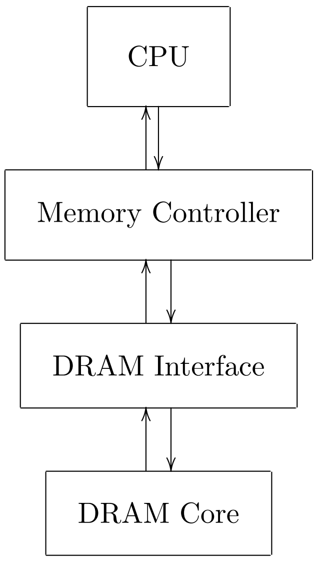

Figure 2.5: Diagram of the connections between the memory controller, the DRAM interface and the memory core.

2.3.1 The Memory Core

The memory core (the part that actually stores information) has been relatively unchanged over the years. Each bit is stored as a small charge in a capacitor as shown in figure 2.6. Its operation is simple; First, the corresponding word line is activated, thus opening the transistor. Then current flows from the capacitor to the bit line. The bitline is connected to a *sense amplifier* (not shown). This component amplifies the signal caused by the charge in the capacitor. That signal is then stored in a latch as the original charge in the capacitor is lost when it is read. Thus logic is needed to write the data from the latch back into the capacitor.

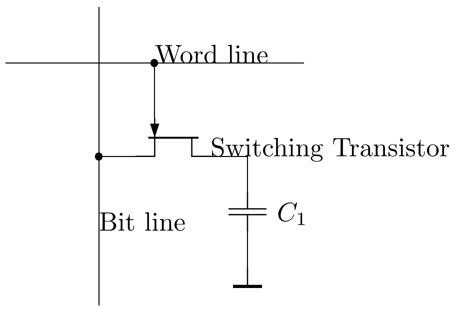

Figure 2.6: A single DRAM cell.

To store many bits, many such elements are connected in a matrix. In such a configuration, a whole row is read at the same time. Because of the small feature size, the bitlines are physically very close together and form a relatively large capacitor. This effected is countered by *precharging* the bitlines before an access to the capacitor, thus decreasing the bias as a result from the previous read. In addition, because capacitors looses charge over time, there is a need to refresh this charge. In DRAM this is commonly done by periodically reading an entire row and writing it back into the capacitors.

There are some interesting parameters commonly used by manufacturers regarding the core of the DRAM chip. The following definitions are taken from John. L. Hennessy and David A. Pattersons textbook “Comnputer Architecture: A Quantitative Approach”[4].

Definition 9 (Access time)**.** Access time is the time between when a read is requested and when the desired word arrives.

Definition 10 (Cycle time)**.** Cycle time is the minimum time between requests to memory.

In addition, a chip might adopt a *closed* or *open page* policy. An open page policy is keeping the last page read from the core in the latches, thus speeding up accesses that hit the same page (row). The downside is that the latch must be flushed if the access does not hit the same page, thus increasing latency in those cases.

2.3.2 The DRAM Interface

Most of the architectural development of DRAM has happened in the interface to the memory core. This development has been driven by three factors:

- The need for higher bandwidth.
- The need for lower latency.
- Cost.

The demand for higher bandwidth was driven trough the need to keep processors occupied with data as they get faster. The need for lower latency was driven trough the expense of stalling the pipeline in a modern processor in case of a cache miss. Cost was a driving factor for bringing high performance memory subsystems to the market. As can be seen with the case of Rambus, the limiting factor of the system was licensing issues and cost[2]. Rambus (`rambus.com`) is a relatively new company specialising in designing memory subsystem. Its first large commercial product used very narrow data paths (16 bits), but with multiple channels. It was a complete redesign of the memory subsystem. It was a good design, but did not get wide acceptance in the marketplace, even with Intels backing.

The first memory systems were asynchronous (no central clock). Instead the memory controller would wait for a specified amount of time before reading data after issuing a request. This system did not scale well, so instead a synchronous interface was made, called SDRAM [33]. This type shared a system clock with the memory controller, enabling two key issues. First, it could be run at a higher speed, and second; commands could be pipelined. The commands for one memory access could be issued while another memory access was read. In addition, multiple banks of SDRAM could be used, thus several accesses could be done in parallel. SDRAM also introduced *burst mode* where a sequential number of bytes in the same memory array row can be pipelined onto the data output bus by only updating the column address on each cycle. Many variants were made of SDRAM including: Enhanced SDRAM, Virtual Channel SDRAM, Fast Cycle DRAM [33].

As the SDRAM architecture reached it maximum potential, a more efficient architecture was needed. DDR (Double Data Rate) SDRAM increases the bandwidth and lowers the latency by transferring data on both the rising and falling edge of the clock. In addition, DDR was license-free (as opposed to Rambus) thus leading it to market acceptance. DDR2 is the new standard, and currently lives alongside of DDR in modern systems. This design uses packets of data transferred over a network, rather than a bus-based system.

XDR[34] is the second generation memory subsystem made by Rambus. It reuses many of the basic ideas, but uses differential signaling to achieve higher speeds. It also uses a more network-like structure, using packets and transactions. Many high-end systems use XDR today, such as Sony Playstation 3.

2.3.3 Memory Controller

The memory controller is the final component in the subsystem. Its main task is to decode the memory request from the bus master (the CPU or a bridge[3]) and convey it over the DRAM interface to the DRAM chips. In addition, it is responsible for refreshing the charge in the memory cells.

Modern memory controllers perform other tasks as well. To be efficient they must be able to handle concurrency, such that the CPU can operate efficiently by pipelining requests as well as sending accesses to multiple banks to enable parallelism. In an SMP[4] environment it also enforces cache coherency, by either snooping the bus (MISO protocol) or by using a directory. It can also prioritize traffic, refresh operations will have a relatively low priority when the charge in the cells are high. Fault tolerance, such as parity checking or ECC is also assigned to the memory controller.

Many modern CPU's (such as AMD64) include the memory controller on-chip. This leads to less flexiblity, but higher performance.

[2]However, Rambus is currently undertaking legal action against several DRAM manufacturers under allegations of price fixing.

[3]A device that connects two similar or dissimilar buses together.

[4]Symmetric MultiProcessing.

2.4 Chip Multiprocessors

Chip Multiprocessors[5] (CMPs) are multiple processors embedded on a single chip. At a minimum the two processor cores must share a package and physical I/O pins. However, because the cores are tightly integrated on the same piece of silicon, there exists other opportunities for sharing as well. The cores can share caches, memory controllers and functional units among others. The tight integration allows for new architectural possibilities, especially when cores cooperate closely (for example in a producer/consumer relationship).

In the literature, it is common to see the term Multiprocessor System-On-Chip (MPSoC). This term refers to the same concept with multiple embedded processors on the same piece of silicon. However, the MPSoC term is most commonly used in system-on-chip designs with heterogeneous processors, such as the Trimedia TR-1300. The TR-1300 is made by Philips as a multipurpose multimedia chip [35]. It has multiple application specific processors to decode video and sound as well as a general purpose VLIW[6] CPU.

Most commercial vendors of high performance microprocessors are now turning to CMPs. Both AMD and Intel manufacture chips with two cores ("Intel Core Duo" and "AMD Athlon X2"). These two vendors are more conservative than others: Sun Microsystems has developed the T-1 Niagara with 8 independent cores, where each one is capable of running 4 threads simultaneously. IBM, Sony and Toshiba have developed the Cell microprocessor in a joint effort. The Cell consists of a heterogeneous architecture, with one main processor and eight smaller ones connected in a ring.

Why do the semiconductor industry embrace CMPs? The reasons for moving to a CMP architecture are numerous, but the main issues are [6]:

1. Increasing power consumption causes thermal problems;
2. Design complexity;
3. Limitations of Instruction Level Parallelism (ILP);
4. Increasing memory-processor gap.

As the clock frequency increases the power requirement increases with it. This poses a problem, as the temperature of the chip increases too. If the temperature goes beyond a certain point, damage to the chip might occur. The idea is that two slower and smaller cores can give the same performance as a larger single core, but at a lower frequency and thus lower power requirements.

Design complexity has become an issue as designs include more functional units and deeper pipelines. Thus high-end microprocessors have become ever increasingly complex to design. To put it another way; As more and more transistors become available to designers, it has become ever increasingly difficult to make good use of them. Using a CMP architecture is promising, as it offers a designer the possibility of reuse. A single CPU design can be reused multiple times on a single chip.

Magnus Ekman and Per Stenström have looked at the trade-off between multiple cores and issue width, and found the optimum width to be around 4 [36]. Although it is

[5]The term multicore is also used in the literature.
[6]Very Long Instruction Word.

theoretically possible to increase ILP up to the data flow limit[7] [4], it is not very practical as it requires very large window sizes, near-perfect branch prediction, very large caches, large register blocks and perfect alias analysis. It is much easier to exploit Thread Level Parallelism (TLP), where parallelism is expressed as threads. This approach is common in server applications such as webservers and database servers, where each execution is only loosely coupled. Such programs maps very elegantly onto a CMP as each processor can handle one thread, instead of using one large processor to interleave the threads execution.

Because the cores in a CMP operates at a lower frequency, the memory-processor gap is reduced. However, the opportunities for prefetching actually increases [12]. This is due to the bandwidth limitations that occur in a CMP due to the sharing of external pins as well as caches.

A CMP opens up a vast array of architectural choices. The cores themselves can be homogeneous or heterogeneous. However, the most active research topic is how to interconnect the cores. The most basic approach is to simply not connect them, or only connect them at the memory controller. A more interesting approach is to share one or two levels of the cache through a crossbar or similar structure[8]. In the next section I will briefly look at what commercial vendors have done. The purpose is not to give a complete overview of all commercial vendors or processors, but rather highlight some differences.

The move to CMP has also prompted a paradigm shift in terms of how programs must be implemented to achieve the highest performance. In a CMP, it is important to segment the problem into multiple pieces in such a way that all the cores can be utilized. It is also important to balance the load across the cores. IBM has released a research compiler called "Octopiler" for the cell architecture that automatically compiles code for uniprocessors to CMP-aware code [38]. However, the performance of the resulting binaries are not very impressive, and for high-performance applications the optimum choice is still doing it by hand.

2.4.1 Commercial Vendors

AMD X2

AMD has chosen a very straight-forward design. The basic X2 is simply two Athlon64 cores on a single die, with separate L2 caches [39]. The cores are connected by a simple crossbar switch to the shared memory controller. This design is very simple and offers pin compatibility with the single core Athlon64 chip. The architecture of the Athlon X2 is shown in figure 2.7.

SUN Niagara

Sun takes a more radical approach to CMP and has designed the Niagara processor. The chip consist of 8 cores, each capable of running 4 threads simultaneously [41]. Each core is about as powerful as the UltraSPARC III. It has a shared 3MB 12-way set associative

[7]The data flow limit is the maximum possible parallelism, that is only limited by actual true data dependencies.

[8]An interesting approach is to build a network on chip, where data is moved as packets on a network [37]. Magnus Jahre at the NCAR group is currently working on extending the simulator framework with such networks.

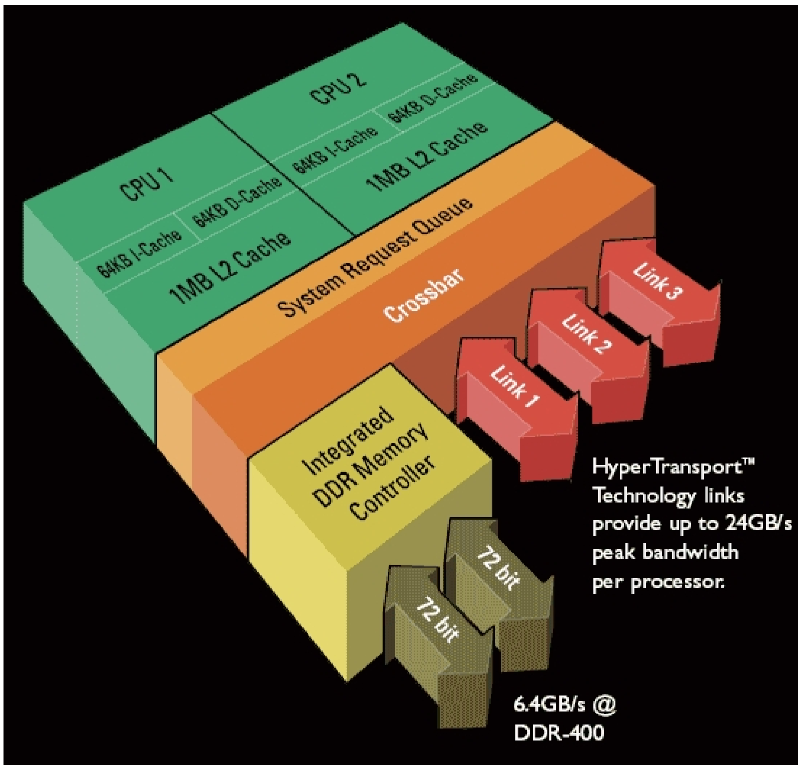

Figure 2.7: The Athlon X2 [40].

level 2 cache. However, the 8 cores share a single floating point unit, making it unsuitable for most scientific applications. The architecture of this chip is shown in figure 2.8. It is worthwhile to notice how the cache is divided into four banks, each one is connected its own memory controller (Marked as "System Interface Buffer Switch Core" in the figure).

The Niagara is targeted at server applications, where most instructions are integer operations. By making each core multithreaded, SUN hopes that the relatively low amount of cache available is sidestepped by the amount of thread level parallelism available.

In 2006, Sun open-sourced the Niagara processor. The full source code can be found on the website `www.opensparc.org`. In addition to the full verilog code, a simulator and the operating system are available under open-source licenses.

IBM Cell

The Cell is a result of a cooperation between Sony, Toshiba and IBM that started in 2000. The Cell has a heterogeneous architecture; It has one large core with a traditional cache [43, 44], and there are eight smaller cores that are optimized for SIMD[9] called SPEs (Synergistic Processing Elements). The smaller cores does not have a cache, but a local store. The SPE can only operate on the local store, but can request DMA transfers to and from main memory. The idea is to let the SPE work on one part of the local store, while DMA transfers data to other parts of memory. In addition, there is a circular ring (EIB) that

[9]Single Instruction Multiple Data.

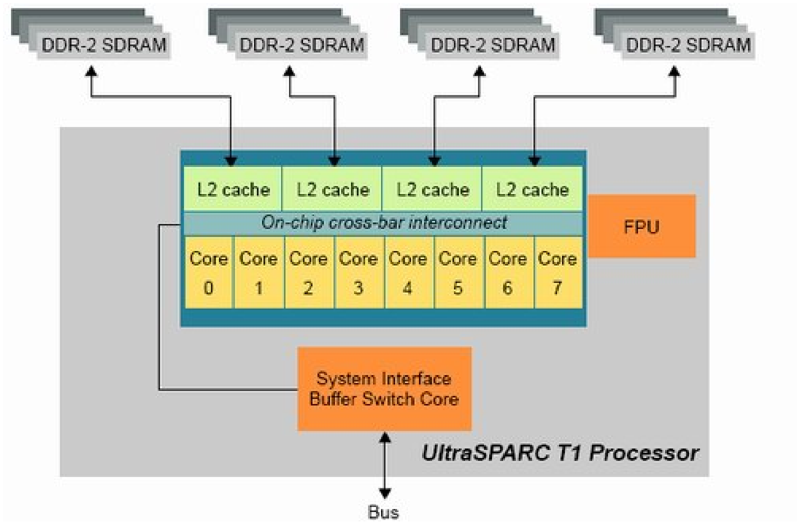

Figure 2.8: The architecture of the Niagara (T1) [42].

handles communication across cores and with the PPE and memory controller. By using a programmer controlled memory hierarchy, significant speedups can be achieved [45]. On scientific kernels a speedup of over 20 times compared to traditional architectures have been shown on single precision arithmetics. The architecture can be seen in figure 2.9.

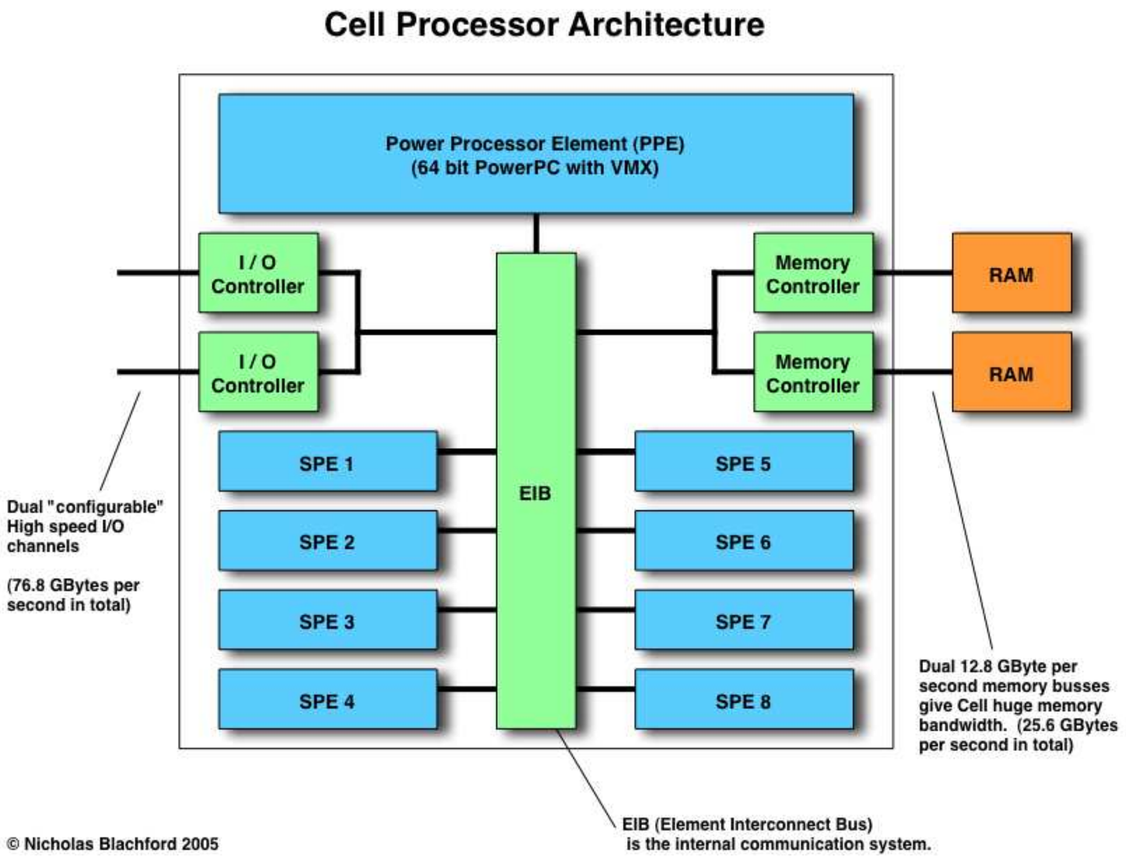
Cell Processor Architecture
Power Processor Element (PPE)
(64 bit PowerPC with VMX)
I / O Controller
I / O Controller
Memory Controller
Memory Controller
RAM
RAM
EIB
SPE 1
SPE 2
SPE 3
SPE 4
SPE 5
SPE 6
SPE 7
SPE 8
Dual "configurable" High speed I/O channels
(76.8 GBytes per second in total)
Dual 12.8 GByte per second memory busses give Cell huge memory bandwidth. (25.6 GBytes per second in total)
EIB (Element Interconnect Bus) is the internal communication system.
© Nicholas Blachford 2005

Figure 2.9: The architecture of the Cell [46].

2.5 SimpleScalar

SimpleScalar [47, 48] 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.

The main purpose of SimpleScalar is to give researchers as well as designers the ability to experiment with different configurations and to give accurate feedback about the performance of a processor. It can produce a pipe-trace that displays what instructions are in what parts of the pipeline. Graphical interfaces exist to make visualization easier. Furthermore, SimpleScalar logs many aspects of the simulation into performance counters, for easy identification of bottlenecks.

SimpleScalar is a very advanced simulator. In addition to supporting cycle-accurate out-of-order superscalar processors, it has support for advanced branch prediction, cache hierarchies, virtual memory, debugging and I/O. It supports a variety of different instruction set (Alpha[10] and PISA[11] in the default distribution).

SimpleScalar has been used by many researchers, and its open nature (the software is open source as well as free for academic uses) has made it a very robust tool. It is intended to be extended, the code is very modular and well written.

Numerous add-ons exist to SimpleScalar [50], such as Wattch [51], that can predict power consumption for a given configuration. Other additions are: Value prediction, hotspot modeling, multi-threading, multiprocessing and different ISAs.

The next release of SimpleScalar (version 4) has a new redesigned core, called MASE [52]. This new core decouples the functional and the timing simulation. Thus, even if the timing simulation contains small, insignificant errors, it will not affect correct program execution.

Other alternative simulators were also evaluated for this thesis, such as Simics [53] and Simflex [54]. Simics does not provide enough detail when dealing with microarchitecture at this level and cannot be used. Simflex was also considered, but changing the simulator core from SimpleScalar would discard a large amount of work done in the fifth year project.

2.5.1 SimpleScalar Model

Discrete Time Specification Systems [55] (DTSS) have a long tradition in the modeling of digital electronics. This is due to the fact that most digital circuitry is governed by a central clock. This clock ticks at a given frequency and the state of the machine is only changed at those discrete time intervals. However, analog circuit elements such as transistors, capacitors and resistors that make up the digital building blocks (such as gates, memory elements and wires) can only be modeled by using detailed Differential Equations Specification Systems (DESS). However, DESS modeling of a Pentium 4 class processor would need several cpu-days to model even a single clock cycle. This is not practical, and thus using a DTSS model is more convenient and produces very accurate results. The drawbacks of using such a simplified model is that information is lost. Specifically the

[10]A RISC processor developed by DEC.

[11]The Portable Instruction Set Architecture - a MIPS-like ISA for use by researchers and students [49].

information about capacitance between wires. This information is needed, for example to compute the power requirements as can be seen in equation 2.4.

$$P \propto CV^2 f \tag{2.4}$$

Where P (the power requirement) is proportional to C (the capacitance of the circuit), V (the supply voltage) and f (the frequency).

At the core of SimpleScalar is a cycle accurate simulation, using a DTSS model of the pipeline. This can be seen in figure 2.10, where the main processor pipeline is at the top. Each step in the pipeline is modeled as a separate component. This is useful because it allows researchers to concentrate on one area, without worrying too much about the effects of their changes in other stages of the pipeline. Moreover, such a coupled model enables researchers to analyze the instructions passing between different stages. In my research this has allowed me to analyze how pipeline stalls forms based on when cache misses occur. In addition, the information provided at each stage lets me analyze what kind of loads typically miss (and are thus good candidates for prefetching).

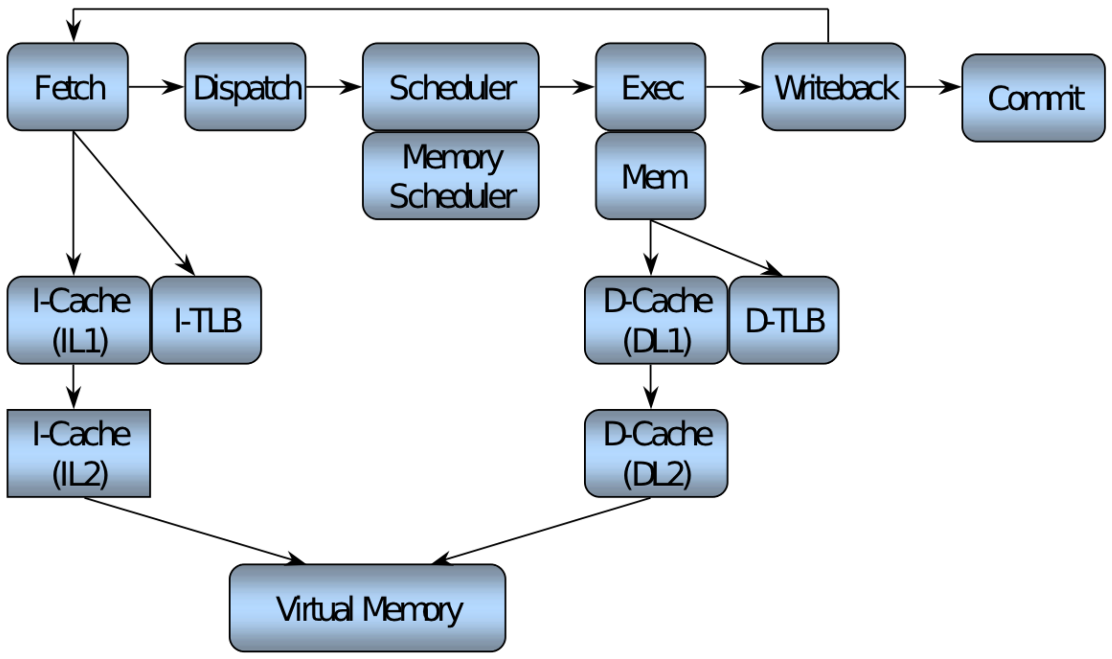

Figure 2.10: The SimpleScalar model.

The simulator processes the pipeline in reverse order. This ensures that instructions that are in the commit stage are processed before instructions in the write-back stage are passed to the commit stage. In that sense instructions move from left to right, while the simulation moves from right to left.

If the processor needs to access memory (either an instruction, or data) it moves vertically in the figure. Because memory accesses are typically very slow, these accesses are not modeled in discrete time. Instead, a function calculates the latency of the operation (in clock cycles) and an event is put in a queue (the load/store queue). This is done to gain simulation speed. As the latency is typically high (200+ clock cycles) there is no point in checking every cycle. This is interesting, because SimpleScalar has two separate simulation mechanisms working simultaneously. Most of the time, SimpleScalar executes a simple DTSS model and updates the pipeline in a cycle-by-cycle basis. However, in the case of long latency operations, such as loads, it uses an event-driven mechanism. This is purely done for performance reasons. An instruction usually finishes a pipeline stage in a single cycle, but a load can consume several hundred clock cycles. Thus, by using a mixed model, huge performance gains can be achieved.

2.6 Performance Counters

Modern processors are complex beasts. Optimizing code for the best performance can be challenging as the programmer must understand every nuance of the architecture. This can be a daunting task as one needs to understand what will cause a performance degradation, simply by looking at the code.

A better approach is to instrument the code. To find out where the performance bottlenecks are, it is possible to time the execution of the program to see where there is potential for optimization. But timing can only tell you where your program is stalling, it cannot tell you why.

Performance counters can give you such insights. A performance counter is simply a counter that counts discrete events. An event can be a cache miss, a cache hit, a branch misprediction, etc. The programmer can thus profile his code by using these counters.

Almost every modern high performance processor today have performance counters [56]. Accessing these counters are usually processor specific. In this section I will look at how performance counters work in a specific CPU (The Athlon XP[12]) and present some high level libraries that do the same work without invoking kernel-level assembly magic.

2.6.1 Performance Counters on x86

As an example, I will look at how performance counters work on the x86 architecture, more specifically, the AMD Athlon XP line of processors. The x86 family of processors is not homogeneous, each generation and each producer has its own set of performance counters and different ways to access them.

The x86 family of processors has one special kind of register; The TSC. The TSC stores the number of elapsed clock cycles since the boot-up of the system. This register can be accessed by issuing the "rdtsc" instruction. The number of clock cycles elapsed is then stored in the edx:eax register pair. This function is particularly useful for timing small snippets of code where precision is important. However, it is important to note that the timing code itself consumes clock cycles, and must be accounted for when using it as a timing device.

Listing 2.3 depicts a code snippet showing how gcc inline assembly can be used to access this register. The "cpuid" instruction is a serializing instruction [57], which makes sure that every previous instruction have committed. The "cpuid" instruction is mainly used for identifying the processor, the serializing effect is only a beneficial side-effect. A cpuid instruction will flush the pipeline before the next instruction is processed.

An explanation of the gcc inline assembly syntax might be in order. The parts of the directive is separated by colons (":"). The first string is the assembler instructions; Multiple instructions can be separated by semicolons (";"). The next part is outputs (or constraints). In the case of rdtsc, this part instructs gcc to put the contents of register eax into the variable eax and the contents of register edx into the variable edx. The next fields are optional, and ommitted from the "rtdsc" instruction. The third field corresponds to inputs and is handled in the same manner. The last field is the "clobbered" registers, which tells gcc which registers will be altered by the code snippet. For a more complete understanding of GCC inline assembly see [58].

[12] I chose the Athlon XP because I needed a computer that I had root access to as well as physical access.

```
static inline unsigned long long rdtsctime() {
    unsigned int eax, edx;
    unsigned long long val;
    __asm__ __volatile__("cpuid": : : "ax", "bx", "cx", "dx");
    __asm__ __volatile__("rdtsc":"=a"(eax), "=d"(edx));
    val = edx;
    val = val << 32;
    val += eax;
    return val;
}
```

Listing 2.3: Elapsed clock cycles performance counter.

In general, the performance counters are accessed through Model Specific Registers (MSR). In turn, the MSRs are accessed through the instructions "rdmsr" and "wrmsr" for reading and writing respectively. The register to be written is set in register ecx, and the parameter or return value is set in eax.

However, issuing a "rdmsr" or "wrmsr" instruction in user-space triggers a segmentation fault[13]. This applies to the "rdpmc" (read performance monitor counter) instruction as well. All of these instructions are privileged instructions and can only be executed by the kernel. This is due to security concerns. An attacker might use the performance counter to gain information about what another user on the system is doing by looking at the performance counters. This type of attack is called a side-channel attack [59].

To enable user-space applications to read performance counters, a bit has to be set in the control registers (cr4). However, the code that sets this bit has to run in kernel mode, thus enabling this bit requires patching the kernel. Fortunately, Linux (and most other operating systems) allows modules that can be dynamically added or removed to a running kernel.

I have created such a module and the code can be found in appendix C. This kernel module is not general purpose, its behavior on other than Athlon XP systems is undefined and might cause damage. It is only intended to show how performance counters work in an actual processor.

On the Athlon XP CPU, MSR 0xC0010000 controls counter number 0 [60]. In the kernel module, I have set this register to count the number of cache misses. The example program reads this value by using the "rdpmc" instruction.

At the core of the example program is the code snippet in listing 2.4. This program must be compiled without optimizations, or the compiler will simply optimize away the loop. By running the program we obtain the following output:

```
Clock is 16667368882865
Number of misses: 265
Clock is 16667369028778
```

Hence the code caused 265 cache misses and 145913 clock cycles elapsed.

[13]Or a general protection fault depending on the OS.

The complete source code can be found in appendix C, which includes functions to read the performance registers, setting the performance registers to the wanted function and enabling user-space to access the performance counters.

```
printf("Clock␣is␣%lld\n", rdtsctime());
start = readpc();

for (i = 0; i < 1000; i++) {
  d[i] = a / c;
}

stop = readpc();
printf("Number␣of␣misses:␣%lld\n", stop-start);
printf("Clock␣is␣%lld\n", rdtsctime());
```

Listing 2.4: Example program using performance counters.

2.6.2 Performance Counter Libraries

Using performance counters directly is a tedious and error-prone task. The resulting code is also very processor specific. Thus there exists numerous performance counter libraries as well as tools. Some of the most common (for Linux) are listed below:

- Oprofile[61];
- VTune[62];
- Performance Application Programming Interface (PAPI)[56].

Oprofile is a tool for profiling Linux programs (as the name implies). It uses the built in performance counters, but also gives applications direct access to the same performance counters. In addition, Oprofile ships with most modern Linux distributions.

VTune is a proprietary tool made by Intel to enable programmers to profile their code through the use of performance counters and sampling.

PAPI is an abstraction layer that enables the application programmer to access the performance counters without worrying about the underlying processor. After installing PAPI on the system, it is very easy to use. An example program can be seen in listing 2.5. Running this program on an AMD Opteron generates the following output: Note the low number of level 2 cache misses, this is the prefetching engine in effect. If there was no hardware prefetching present, one would expect a much larger amount of L2 misses (closer to 100000 divided by the cache line length in *ints*).

```
Number of stall cycles : 1033362
Number of DL1 misses : 5309
Number of DL2 misses : 4
Number of DTLB misses : 190
```

```
/* Example program using PAPI
 * Compile with:
 * gcc papi.c /opt/xd-tools/papi/3.1.0/lib/libpapi.a -o papi
 */

#include "/opt/xd-tools/papi/3.1.0/include/papi.h"
#include <stdio.h>

//Number of events to monitor
#define NUM_EVENTS 4

int main(int argc, char *argv[]) {
  int Events[NUM_EVENTS] = {PAPI_RES_STL, PAPI_L1_DCM,
                  PAPI_L2_DCM,PAPI_TLB_DM};
  long long startvalues[NUM_EVENTS];
  long long stopvalues[NUM_EVENTS];
  int d[100000];
  int i;

  if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK) {
    printf("Could not start counters\n");
    exit(1);
  }
  if (PAPI_read_counters(startvalues, NUM_EVENTS) != PAPI_OK) {
    printf("ERROR: %d \n",
                PAPI_read_counters(startvalues, NUM_EVENTS));
  }
  /* The code to be monitored */
  for (i = 0; i < 100000; i++) {
      d[i] = 6;
  }

  if (PAPI_read_counters(stopvalues, NUM_EVENTS) != PAPI_OK) {
    printf("ERROR: %d \n",
                PAPI_read_counters(stopvalues, NUM_EVENTS));
  }
  printf("Number of stall cycles : %lld\n",
                stopvalues[0] - startvalues[0]);
  printf("Number of DL1 misses : %lld\n",
                stopvalues[1] - startvalues[1]);
  printf("Number of DL2 misses : %lld\n",
                stopvalues[2] - startvalues[2]);
  printf("Number of DTLB misses : %lld\n",
                stopvalues[3] - startvalues[3]);
}
```

Listing 2.5: Example program using PAPI

Chapter 3

Methodology

3.1 Memory Model

To accurately model prefetching, an accurate model of the underlying memory subsystem is needed. In the case of prefetching there are two distinct properties that are especially interesting:

- Latency;
- Bandwidth.

Latency is a measure of the response time (the time between the issue of a memory request and completion). Bandwidth is a measure of how much data the communication channel can transmit at a time. This value can be measured in bytes per second or the number of simultaneous requests. The correlation between these two measures is a function of both latency and the size of each request.

A real-life memory subsystem is complex and implementation specific. However, we need a general model, that is easy to understand and reason about, such that the effects of prefetching becomes clearer. Our model needs support for:

- Parallelism;
- Multiple banks;
- Pipelining;
- Open/Closed pages.

Vinodh Cuppu[63] observed a significant amount of locality in the address stream that reaches the primary memory system (40% on average). A computer architect would use this knowledge to design a modern memory subsystem that exploits this fact. An open page policy would allow memory adresses that are close to the previous adress to receive a speed boost. Spreading the data across multiple banks would also enable more concurrency and thus more bandwidth, which in turn will increase performance.

3.1.1 Implementation Details

To account for parallelism, the notion of *channel* is introduced. A *channel* maps either to a direct Rambus channel or a memory bank, depending on the underlying technology. By using a mapping function, a memory access is mapped to a single channel. This mapping function maps cache lines such that a memory bank holds one cache line, and the next channel holds the next line in a cyclic manner. See figure 3.1. By increasing the number of channels, more bandwidth becomes available.

Pipelining is implemented by checking if some operations can overlap. If a transfer is currently in progress, the memory subsystem can issue the commands to the memory chips while the other completes. The total latency would then be:

$$TotalLatency = TimetoWait + MemoryLatency - CommandTransferTime \qquad (3.1)$$

To exploit locality, our model assumes an *open page* policy (see section 2.3.1). To accomodate this, the last paged accessed is stored and a check to see if the access hits the same page is performed. If it does, the time to complete the transfer is reduced.

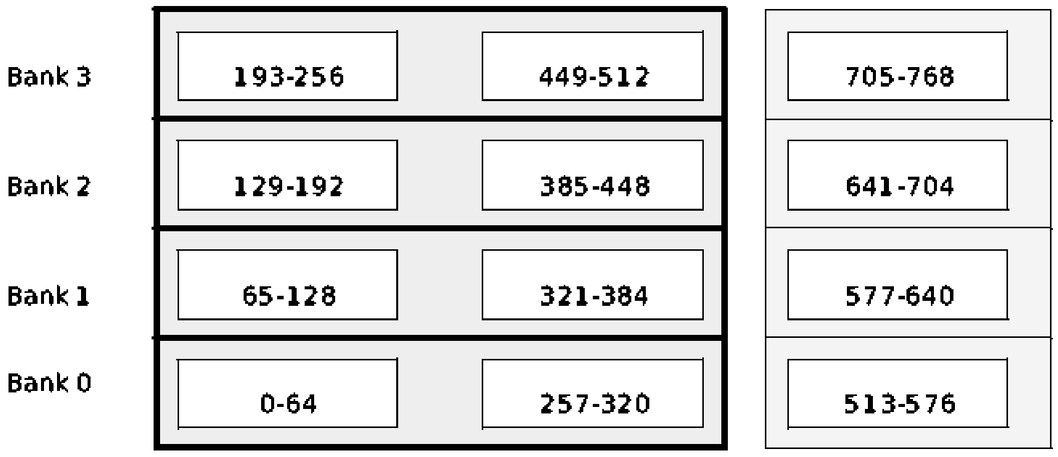

Figure 3.1: Memory organization in the simulator. Each box represents a block (with byte numbers). Each bank is separated vertically and each page is composed of two blocks (shown in grey).

Refresh (Regularly recharging of the capacitors) is not simulated, as it is assumed to affect all benchmarks and all the configurations equally. It can also be modelled as a stochastic process, and thus stall with a given probability on every access. Or the effect of each refresh can be baked into every access, by distributing the effect of refresh on every access. Modern memory controllers can schedule refresh in such a manner that it does not interfere with normal operation.

3.1.2 Model Parameters

As with any model, realistic parameters are key to a realistic simulation. I have chosen to model DDR2, based on its widespread commercial use and its documentation.

```
typedef struct {
  int num_channels;          /* Number of channels */
  int block_size;            /* Size of each block, usaully equal
                              * to l2 block size (in bytes) */
  int page_size;             /* Number of blocks in a page */
  int control_time;          /* Time to transfer data */
  int core_time;             /* Time to transfer from core to
                              * latches */
  int data_time;             /* Time to transfer from latches to
                              * memory controller */
  tick_t *ready_channels;    /* When are the channels ready? */
  md_addr_t *last_address;   /* Last page accessed */
} dram_system_t;
```

Listing 3.1: DRAM-model datastructure.

DDR2 comes in many different configurations, differing mainly in bus speed and latency. These two numbers are used to calculate the peak bandwidth available from a module. As an example - a PC2-4300 DRAM module is a DDR2 module with a peak bandwidth of 4300 GB/s [64]. As of Febuary 2005, this type of chip commonly runs at a clock speed of 400 MHz (Double Datarate) and a CAS (Column Address Select) latency of 4 clock cycles. Because the data bus is only 64 bits wide and a cache line is 128 bytes, 16 data transfers are required, or 8 clock cycles (double data rate).

The CM2X512A-6400 [65], a 512MB DDR2 DRAM module manufactured by Corsair Memory Inc will be the simulator target. It is rated as a 4-4-4-12 module (CAS latency - RAS to CAS latency - RAS precharge time - Cycle Time). From these numbers we see that RAS to CAS is 4 clock cycles, CAS is 4 clock cycles. This gives a total of 16 clock cycles. However, the DDR2 data bus runs at only 400MHz compared to the 4GHz achieved by the main CPU. Thus the 16 clock cycles required by the memory transaction, translates into 160 clock cycles for the processor.

For simplicity, I will assume that the full extent of the RAS to CAS latency can be pipelined. I will also assume that a hit to an open page gives a 4 cycle bonus. Each page is 8 cache lines large (or 1kB). These are the default values for the simulations I will be conducting.

The number of channels available will also be a parameter. The model number (4300) signifies that it can deliver a peak of 4.3GB/s. As each access consumes 16 clock cycles at 400MHz, this translates to about 40ns per access of 128 bytes. Each "channel" will then be able to transfer 3.2GB/s. A full 4300 DRAM chip will then require about 1.5 channels to implement. This number will not be fixed, as the effects of bandwidth on prefetching is one of the key topics of this report.

3.1.3 DRAM Statistics

The DRAM subsystem collects the following information:

- The number of accesses to DRAM.

- The total latency imposed by the DRAM subsystem.
- The average latency of the DRAM subsystem.
- The number and percentage of accesses that hit on open pages.
- The number and percentage of accesses that are stalled due to contention.

Each of these values are implemented as simple performance counters, or calculated as ratios between two counters. These values are available to the end user after a simulator run.

3.1.4 Model Verification

To ensure model validity 25 of the 26 benchmarks from the SPEC benchmark suite were run (The last benchmark would not run with the simulator, due to issues with the data set). The new model was compared to the old model with a fixed latency, set at 160 clock cycles. The results can be seen in figure 3.3.

For CPU-bound applications (such as VPR and Twolf) there were little changes between the models. This was expected, as there is little communication with main memory. In memory-bound applications (such as Apsi and Swim) we see a performance degradation when there is not sufficient bandwidth available. In some benchmarks we see a performance increase when using the new model. This is due to the linear way that these programs access memory, thus increasing the chance of hitting open pages and reducing latency below the fixed latency offered by the standard model.

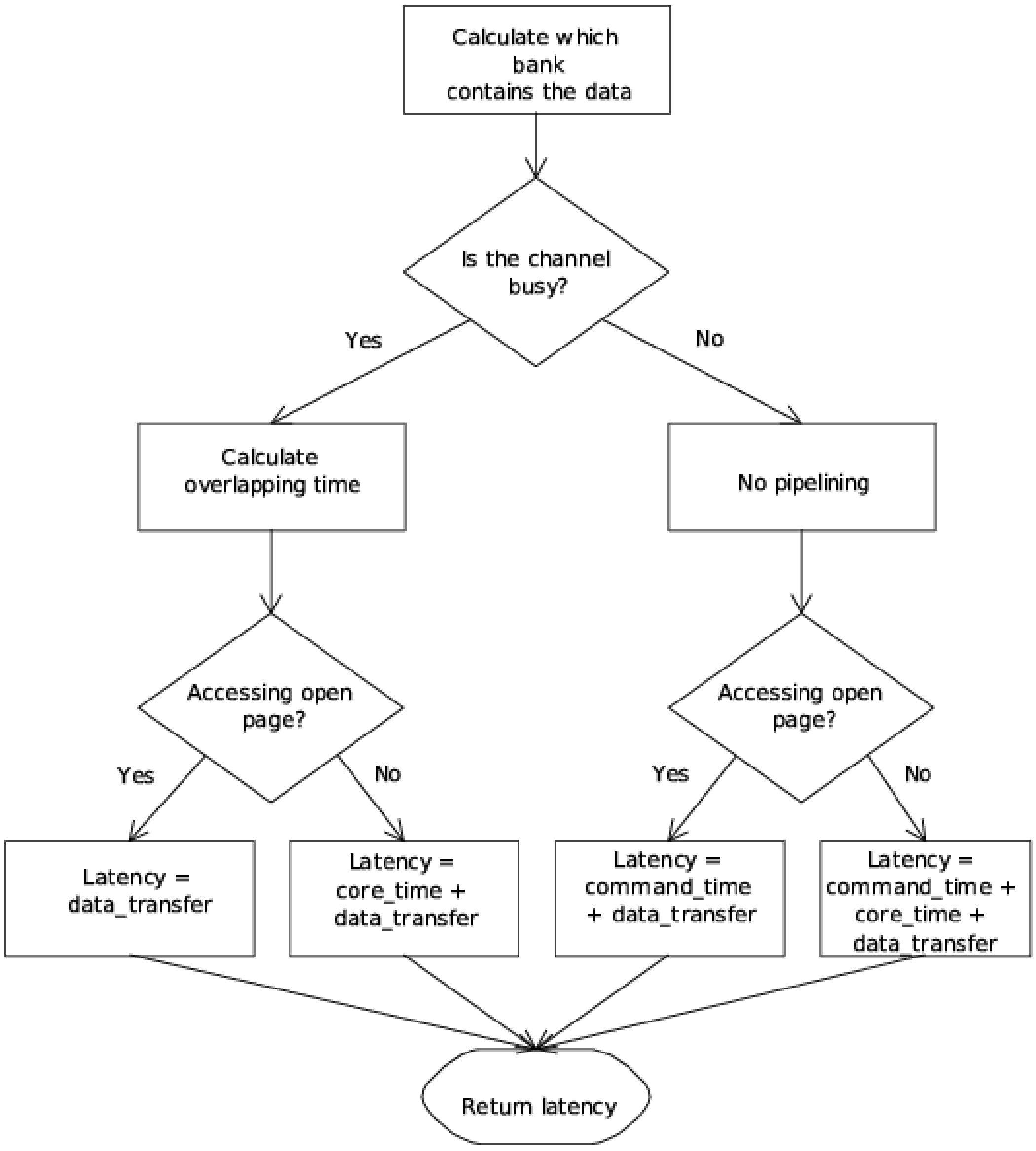
Calculate which bank contains the data
Is the channel busy?
Yes
No
Calculate overlapping time
No pipelining
Accessing open page?
Yes
No
Accessing open page?
Yes
No
Latency = data_transfer
Latency = core_time + data_transfer
Latency = command_time + data_transfer
Latency = command_time + core_time + data_transfer
Return latency

Figure 3.2: Flowchart indicating how the latency of a memory access is calculated.

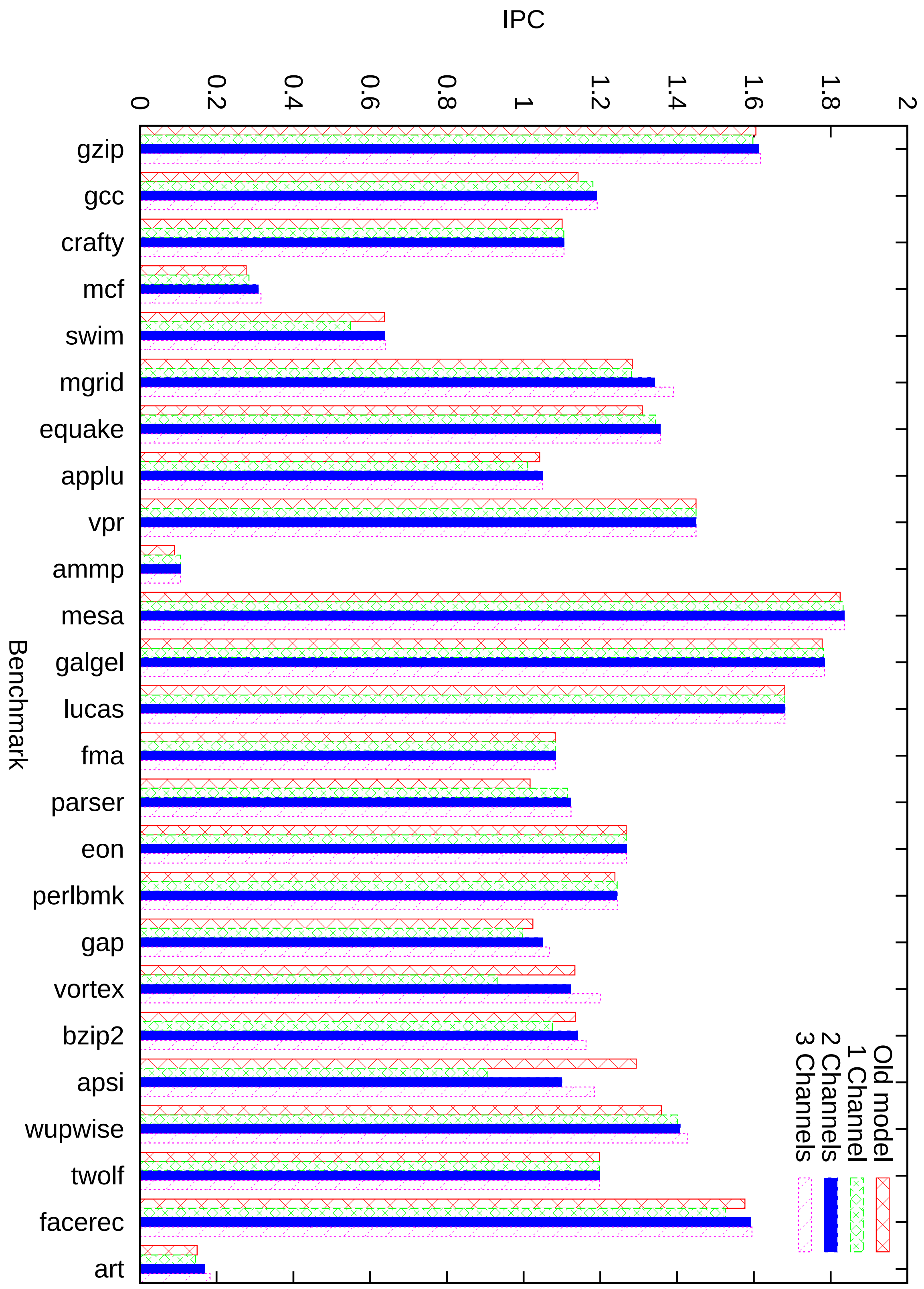

Figure 3.3: IPC of SPEC benchmarks using the old model and 1,2 or 3 channels in the new model.

3.2 Prefetching

Most of the work in implementing prefetching in Simplescalar was done in my fifth year project. The following section includes the documentation produced during the project to make this thesis self contained. In section 3.2.2 the new additions and improvements are presented.

The documentation has been somewhat modified and references to deprecated functions and other modules have been omitted. For a complete description of the module, please refer to my fifth year project report.

3.2.1 Implementation

The modifications needs to be as non intrusive as possible. The design has four major components as depicted in figure 3.4.

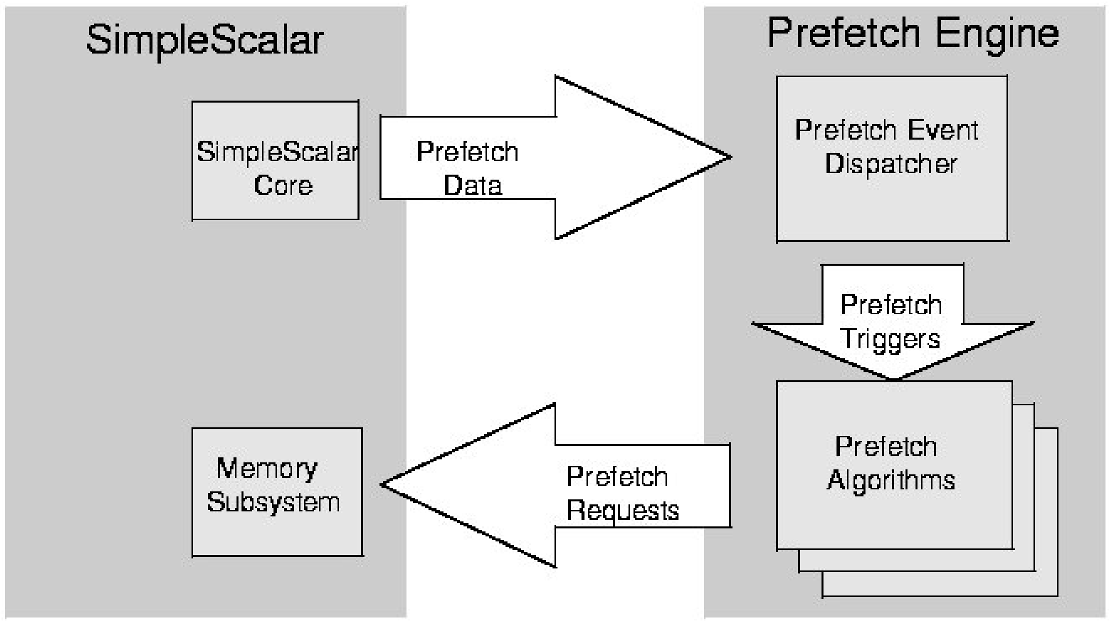

Figure 3.4: Prefetching in SimpleScalar.

The design is centered around prefetch triggers. A trigger is generated in the SimpleScalar core when certain events occur. This trigger is then processed by the prefetching module, which generates prefetches when needed. In brief, the program flow consists of five steps:

1. An event (memory access, cache miss, etc) occurs in the SimpleScalar core.
2. The event data is sent to the dispatcher where it is packed into a trigger.
3. The dispatcher sends the trigger to the selected algorithm.
4. The algorithm decides how it will react to the trigger.

```
typedef struct {
  trigger_type_t type;      /* What happened */
  location_t location;      /* Where something did happen */
  md_addr_t address;        /* Adress of memory access */
  tick_t time;              /* Time of access */
} prefetch_trigger_t;
```

Listing 3.2: The prefetch data type.

5. If the algorithm decides to prefetch, a prefetch request is sent to the memory subsystem.

In general, each prefetch trigger can cause either zero, one or many prefetches, depending on the algorithm being used. To make things modular, both the prefetch dispatcher and the algorithms themselves are put into a separate file (*prefetch.c* and *prefetch.h*).

The changes needed in each module are documented in the following sections. The documentation follows a module based approach as this fits well into the code structure. The modules will be presented in the same order as a prefetch would be handled.

Triggers and Data Structures

At the core of the design is the notion of *prefetch triggers*. A prefetch trigger is generated when special events occur in the simulator. This flow is depicted in figure 3.4. Such an event can be a miss in the L2 cache. The prefetch triggers are implemented as a data structure. This decision has two big advantages: An algorithm only needs to know about one data structure. Second, it makes the framework extensible, as designers can simply add fields to the data-structure without breaking the implementation of other algorithms. The prefetch triggers are defined in *prefetch.h* (see listing 3.2).

This format is well suited because it answers three important questions:

- What kind of event occurred?
- Where did it occur?
- When did it occur?

The *trigger_type_t* is another data format and is implemented as an enum (see listing 3.3). This approach makes the framework extensible, as a designer can add events when needed. In addition the data structure *location_t* (listing 3.4) contains information about where the events occur. Both the location of an event as well as the type of event are used by the prefetch heuristics to determine when prefetching is needed. For example, an heuristic that prefetches to the L2 cache might ignore all events that occur in the L1 cache, or in DRAM.

The address field is used for calculating the prefetch address. It can also be used for other purposes, like looking up data in DRAM for pointer based prefetching.

```
typedef enum {
    Cache_Miss,         /* A miss in the cache */
    Cache_Hit,          /* A hit in the cache */
    Memory_Access,      /* An access to memory by an instruction */
    PC_Update,          /* Program counter is updated */
    No_event            /* Dummy trigger */
} trigger_type_t;
```

Listing 3.3: The trigger type.

```
typedef enum {
    Cache_IL1, /* Event happened in the D-I1 cache */
    Cache_DL1, /* Event happened in the D-L1 cache */
    Cache_IL2, /* Event happened in the I-L2 cache */
    Cache_DL2, /* Event happened in the D-L2 cache */
    DRAM,      /* Event happened in DRAM */
    None       /* When location doesn't matter - eg PC_Update */
} location_t;
```

Listing 3.4: The location data type.

Changes to SimpleScalar Core

The prefetch triggers are generated in the SimpleScalar core. Every interaction with the memory subsystem generates a prefetch trigger. The cache miss handlers are modified to generate prefetch triggers, as well as the simulation core. Each time such an event occurs *process_prefetch_trigger* is called. The *sim-outorder.c* file contains command line parsing and initialization of the prefetch engine.

The code size in this module is kept at a minimum for two reasons: To keep things as modular as possible, and to ease the insertion of this module into SimpleScalar version 4. Command line parsing and initialization code could not be put into the prefetch module as this would depart from the code style of SimpleScalar.

Prefetch Event Dispatcher

The triggers generated in the simulator core are sent to the prefetch event dispatcher. The dispatcher has three distinct functions: First, it packs the data into prefetch triggers. Second, it sends the prefetch trigger to the chosen algorithm. By using function pointers it is easy to swap algorithms while the simulation is running, and to add new algorithms, as only a pointer needs to be updated for the new algorithm to take effect. The pointer name is *prefetch_algorithm* and it can point to any algorithm. The function pointer approach is a very clean solution to this problem as it avoids long *switch* statements or nested *if*s, reducing code bloat and complexity. In addition, by using function pointers, a set structure is imposed on a prefetching algorithm. This is a design decision; By not allowing more than prefetch triggers to be passed as a parameter to the algorithms, the functionality of the reference implementations can be guaranteed.

```
/*
 * This is the sequential prefetching on Miss algorithm.
 * When a miss in the cache occurs on block X, block X+1 is
 * prefetched.
 */
void sequential_miss_prefetch(prefetch_trigger_t trigger) {
  int i;
  if ((trigger.type==Cache_Miss) && (trigger.location ==
     prefetch_location)) {
    for (i=1; i<=prefetch_degree; i++) {
      prefetch(trigger.address + i*target_cache->bsize, trigger.
         time);
    }
  }
}
```

Listing 3.5: Implementation of sequential prefetching.

The third function serves as a guard against rippling effects. Issuing a prefetch will cause a miss to be generated in the target cache. The system will then generate a miss-type prefetch trigger. This in turn can lead to another prefetch being issued, which is not the intended effect. Therefore a flag (*prefetch_attempt*) is set when a prefetch algorithm is executing.

Prefetch Algorithms

The algorithms are intended to be modular. A new algorithm can be plugged into the framework with little effort. It has to be implemented as a function with a single parameter of type *prefetch_trigger_t* and a return type of void. In theory, algorithms only need to call one function, *prefetch()*. This function handles the actual prefetching in SimpleScalar. The designer can thus concentrate on the algorithm and implementation.

As an example consider sequential prefetching. It can be implemented in 5 lines as shown in listing 3.5. The only input is a prefetch trigger. The algorithm uses that trigger to generate a prefetch trough the *prefetch()* function. As can be seen from this example, because the algorithm only use specific parts in the data structure, adding additional data to the prefetch triggers won't break this implementation. And by using function pointers, we gain an extra level of abstraction.

Issuing Prefetches

When an algorithm decides to prefetch data, it can call the function *prefetch()*. This function serves as an abstraction and a simplification of the memory subsystem. In addition to sending the prefetch request to the memory subsystem, it probes the cache to see if the cache line has already been prefetched, if it has, it aborts the prefetch. This is very useful, as it is common for all the prefetch algorithms. If the cache line is not in the cache, the function issues a special kind of memory access, using the prefetch type (see *memory.h*).

Memory Subsystem

A prefetch is handled like any other memory request, with one exception, a prefetched block is marked as prefetched in order to gather data on the prefetching algorithm. Every prefetch issued is counted as well as every successful prefetch. By using these two numbers, the *accuracy* of the algorithm can be inferred.

The memory subsystem has to be able to handle partial prefetches. A partial prefetch is a prefetch that is issued too late, thus hiding only part of the latency.

Statistics

There are many things to consider when evaluating a prefetch scheme. The purpose of prefetching is to speed up execution, therefore the IPC or running time of a program are interesting parameters. In addition, the number of hits and misses to the cache is interesting, because it is a strong indicator of the possible speedup gained by using prefetching. Comparing the number of misses with and without prefetching gives a clear indication of the performance of the prefetching algorithm. SimpleScalar has built in support for these statistics. There are other measurements as well, such as the number of issued prefetches, because it is closely correlated to how aggressive the algorithm is.

3.2.2 New Additions

This section describes the changes made to the original prefetching code developed during my fifth year project. Its purpose is to differentiate between the work done as part of this thesis and the work done as part of the project.

New Infrastructure

The infrastructure that handles prefetching has been improved, especially in the *process_prefetch_trigger* code. Some common decision logic has been moved to this function to speed up simulation, this avoids both code duplication and a costly function call. In addition, the command-line parsing has been made clearer, as the algorithms and the trigger types have been separated into two distinct parameters. The whole cache structure is now available to the prefetchers, as opposed to the previous code, where only the target cache was available for probing and prefetching.

In the previous version of the code, problems could occur with the simulator if the prefetcher would issue memory requests that were out of bounds. This issue has now been resolved by checking if the prefetched address is within the programs address space.

Prefetching Algorithms

Three new algorithms have been implemented:

- Reference Prediction Tables;
- Address-Value Delta;
- Stream prefetching.

These heuristics are described in section 2.2.2.

Finally, due to a bug in the original delta correlation the performance of the delta correlation and the CZone/Delta Correlation algorithm was lower than it should have been. This issue has been fixed in the new code.

3.3 CMP

SimpleScalar is a simulator that is targeted at uniprocessor simulations where there is only one program executing. A significant amount of work has been put into building a CMP extension of SimpleScalar. This work has been done in cooperation with Haakon Dybdahl, a PhD student, at IDI. This section describes how the CMP extension to SimpleScalar works.

3.3.1 Target Architecture

A conservative approach to CMP is to have separate homogeneous cores with private L1 caches. However, the cores still share the L2 cache as well as the memory controller. The L1 caches are connected to the L2 by a crossbar. In addition the L2 cache must be able to handle multiple simultaneous requests.

This architecture is comparable to the upcoming AMD X2 processors, as well as the Niagara T1 processor. Such an architecture is shown in figure 3.3.1, where a CMP with 2 cores are shown, albeit the simulator should be able to handle more cores.

In addition, the simulator should have the following properties:

- Be able to boot current operating systems.
- Be able to execute true parallel programs.
- Support synchronization per clock cycle. This is to ensure the operation of prefetchers that work on a per clock cycle basis.
- Support an ISA that has a cross compiler available.
- Provide per-core statistics.

3.3.2 Implementation

We have based the CMP simulator on SimpleScalar, as we are familiar with its operation and source code, as it has been used in our group for nearly two years. SimpleScalar is a uniprocessor simulator, and the easiest way to convert it to a CMP simulator is to simply run multiple instances at the same time. However, because we want the cores to share resources, a few issues must be resolved:

1. Sharing of the L2 cache.
2. Sharing of the DRAM interface.
3. Synchronization of the cores to ensure correct execution.

These issues will be presented in the next two sections. I will use a 2-way CMP as an example throughout these sections, but the simulator can simulate an arbitrary number of cores (limited by the host machine specifications).

To help with synchronization and provide a way to communicate between cores a separate process has been created, called the controller. The controller is invoked every

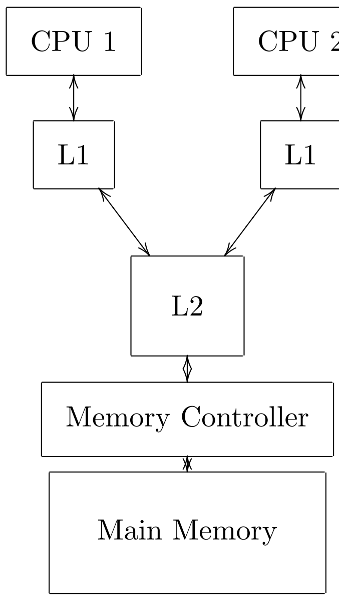

Figure 3.5: Target architecture of the simulated CMP.

10000 clock cycles (user defined), and can perform any operation on both cores. This functionality has been used previously to repartition or reconfigure caches. It can in effect mimic an operating system.

Shared Structures

As each core executes as its own process, the cores do not share address spaces. This leads to a problem, because the data structures representing the cache must be shared between the process. The solution is to put the shared data structures into system shared memory. The files *shared.h* and *shared.c* contains the routines necessary to create and use shared memory on the Linux platform.

Using shared memory for a process in Linux is a five steps procedure:

1. Create the shared memory segment using a unique number to identify it.
2. Attach to the shared memory using the identifier in step 1.
3. Use the shared memory.
4. Detach from the shared memory.
5. When all processes have detached from the segment, it can safely be destroyed.

In practice, core #0 creates the data structures, while the other cores simply uses the shared memory already created. This is the case with both the DRAM and L2 data structures.

However, there is a catch. Pointers cannot be stored in a shared data structure, because different processes might map the physical shared memory addresses to different logical addresses. In essence, if one process creates a pointer in shared memory, it might point to something else for another process. This causes a problem, because the original cache code was very pointer-intensive and had to be reworked.

Because the cores are simulated in separate address spaces, they map their own simulated memory in the same way. In other words, they use the same simulated addresses for different simulated data. To differentiate between which core actually holds valid data in the cache, each cacheline is tagged with the core it belongs to. This has the added benefit of increasing the amount of information regarding cache performance.

As the data structures reside in shared memory, it is important to control access to it, as data corruption might occur if two different processes change the shared structure simultaneously. In our simulator, this is solved by using semaphores with an initialization value of one. This provides a locking mechanism for the critical regions of the program.

When the simulation is completed, the controller destroys the shared memory segments. If the program is aborted (killed or terminated), the shared memory segments are destroyed through a special signal handler in *controller.c*.

Synchronization

To ensure that simulation is cycle accurate, the simulated cores need to be synchronized. This ensures that accesses to DRAM and L2 happen in the correct order.

In the simulator, this is achieved by using semaphores in a ripple chain. The chain works as follows; Core #0 starts executing its first clock cycle, while core #1 waits for its own lock. When core #0 is finished with its first clock cycle, it releases the lock on core #1. Then core #0 increments its cycle counter and waits for its own lock. In essence, the two simulations alternate between executing and waiting.

If a core finishes its simulation before the others, it enters a special state where it simply waits for its own lock, and unlock the next core without doing any actual simulation. When all cores are finished, the controller detects this and terminates the simulation.

This is however not a very efficient way to handle synchronization. It is obvious that the two programs only need to be synchronized when they access the L2 cache. In essence it is only necessary to enforce that L2 accesses occur in the correct order for correct simulation. We have developed such an alternative. It works as a barrier that only allows the core with the lowest clock cycle number to pass. This is implemented as a lock in the same way. We have observed speedups of over 3 by using this method over the ripple method.

Unfortunately, this method cannot be used in this thesis. The controller and this locking method cannot be used simultaneously, as both the controller and the barrier uses semaphores, deadlock can occur. The following example illustrates this: The controller waits for both cores to invoke it. Core #1 waits for the controller to finish, while core #2 waits for L2 access. This issue is currently being worked on.

3.3.3 Implementation Shortcomings

The implemented simulator falls short of the ideal simulator in many ways. Most of these are due to the simulator being based on SimpleScalar.

SimpleScalar on its own cannot boot operating systems. This is both due to its limited ISA emulation (it cannot simulate ring-0 instructions), but also due to the fact that it cannot emulate the full system (disks, network, graphics, bios etc). This limitation is carried on to the CMP version of the simulator. However, the controller can provide limited O/S functionality, but it will have to be written on a per-project basis.

The simulator cannot execute true parallel applications. This is due to the original SimpleScalar program loader. It has no notion of shared memory, and without this information, the extended CMP simulator cannot map shared memory. However, it is possible to simulate shared memory by using system calls. This has been done in a previous project, but requires rewriting the target application to use these special system calls, which can be a significant amount of work.

These two limitations severely restricts the number of benchmarks that can be run on the simulator. This issue will be handled in section 3.5.1.

It is only possible to measure destructive interference with this simulator, as the memory of the benchmarks are assumed to not overlap. There is simply no way that one core can do anything that will benefit the other. It can only create bandwidth contention or displace cache lines. This is a very simplistic assumption, as shared libraries or code is likely to overlap in a realistic system. In a real system, shared libraries (such as libc) might be used by both programs, and therefore, one core can “prefetch” code for the other.

3.4 Bandwidth-Aware Prefetching

This section describes one of the most important contributions of this thesis, namely a new heuristic for prefetching, named bandwidth-aware prefetching. This heuristic is a result from lessons learned both from this thesis work, but also from previous work in the fifth year project.

3.4.1 Motivation

In many areas of computer performance, off-chip memory bandwidth is the limiting factor [45]. As research and industry is moving towards CMP architectures this will become an even more severe problem. In a CMP, the cores share the same physical package and thus the same physical I/O pins. Increasing the number of actual pins on a package is costly, as it represents an increase in cost for both packaging and materials. Thus, the same I/O pins have to be used to serve more cores than previously. In effect, this will reduce the bandwidth per core.

Prefetching in a CMP is still needed [12], as prefetching can significantly increase performance. However, current prefetching methods are not 100% accurate and will thus create unnecessary prefetches that will consume valuable bandwidth.

As the two cores share resources such as the L2 and DRAM, one core might prefetch data to increase its performance. By doing so it might cause bandwidth contention, leading to decreased performance for the other core. In addition, the prefetched data might displace data needed by the other core, thus further decreasing performance.

3.4.2 Idea

The basic idea of bandwidth-aware prefetching is to use existing performance counters to estimate future bandwidth usage to direct prefetching. The reasoning is as follows: If there is little bandwidth contention, then issuing a prefetch will probably not cause future bandwidth contention either. However, if there is bandwidth contention, then the prefetch must be rejected. If the prefetch was accurate, it will probably be delayed for so long that the actual load will be issued before the prefetch is actually dispatched to the memory subsystem. By using such a heuristic, one core cannot overrun the other with prefetches, because the additional prefetches will simply be rejected.

However, predicting future bandwidth usage is hard. The Network Weather Service (NWS) project is an initiative that does research into predicting network performance for computational grids. In a paper by Wolski [66] numerous prediction heuristics are described. However, many of the techniques described in that paper requires considerable computing effort and is thus unsuitable for microarchitectural purposes. A naive, but inaccurate, approach is to use the previous value as a prediction for future values. A running mean value is a much better alternative. Performance counters hold data about previous bandwidth usage. I will use this data to predict future bandwidth usage by averaging the last 3 values. By averaging three values, spikes in bandwidth usage is evened out.

3.4.3 Implementation

Bandwidth-aware prefetching can be seen as a supplement to regular prefetching. The regular heuristics predicts *what* to prefetch, while bandwidth-aware prefetching predicts *when* a prefetch should be issued.

For bandwidth-aware prefetching to work successfully, it is necessary to accurately predict future bandwidth usage. By using performance counters, it is easy to obtain the latency of previous DRAM operations. By averaging the latency of the previous three DRAM operations, we get an indicator of the current bandwidth usage. If the latency per operation is low, it is an indicator that there is little bandwidth contention. If the latency per operation is high, it is a sign that there is memory contention.

If this latency is larger than a set threshold, then the prefetch is simply discarded. If it is less, then the prefetch is processed in the normal way.

3.5 Benchmarks

In the fifth year project I used a subset of the SPEC2000 [67] benchmark. SPEC2000 is a benchmark suite that uses kernels of many common scientific and engineering programs. Many programs are taken directly from common GNU programs such as Gzip, Bzip2 and Gcc. It is commonly used in computer architecture research and is the standard for measuring performance in many settings.

In the fifth year project seven benchmarks were chosen somewhat arbitrarily. The ones used were; Gzip, Gcc, Crafty, Mcf, Swim, Mgrid and Equake. Many benchmarks gained little benefit from prefetching, as they were mainly compute-bound. The benchmarks in the SPEC2000 suite are summarized in tables 3.1 and 3.2.

To decrease simulation time I have used the reduced datasets [68]. The reduced datasets are smaller than the originals, but behave in the same manner. The purpose of the reduced datasets is to decrease simulation time, while keeping the instruction mix of the original workload. It is important to note that when using the reduced datasets[1] the problems are smaller and might more easily fit into the cache, thus cause different behavior from the reference sets. However, in this thesis we are mainly interested in comparing prefetching schemes to each other, thus this problem will not become significant.

Benchmark	Language	Category
164.gzip	C	Compression
175.vpr	C	FPGA Circuit Placement and Routing
176.gcc	C	C Programming Language Compiler
181.mcf	C	Combinatorial Optimization HTML
186.crafty	C	Game Playing: Chess
197.parser	C	Word Processing
252.eon	C++	Computer Visualization
253.perlbmk	C	PERL Programming Language
254.gap	C	Group Theory, Interpreter
255.vortex	C	Object-oriented Database
256.bzip2	C	Compression
300.twolf	C	Place and Route Simulator

Table 3.1: SPEC 2000 Integer benchmarks [67].

Running the reference sets is not an option, as simulation time would amount to days [69], and would leave us with less time to explore the designspace. This would be very costly in terms of computer-time, but it would also limit the number of experiments that can be conducted. The main purpose of the experiments in this thesis is not to achieve high accuracy in specific synthetic benchmarks, but to highlight the effects of prefetching under different circumstances.

A common way to use the reference set in simulations is to fast forward around 1 billion instructions and then do the accurate simulation for around 1 billion instructions. This method also have problems as the large scale structures (branch predictor, caches)

[1]Also called lgred.

Benchmark	Language	Category
168.wupwise	Fortran 77	Physics / Quantum Chromodynamics
171.swim	Fortran 77	Shallow Water Modeling
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field
173.applu	Fortran 77	Parabolic / Elliptic Partial Differential Equations
177.mesa	C	3-D Graphics Library
178.galgel	Fortran 90	Computational Fluid Dynamics
179.art	C	Image Recognition / Neural Networks
183.equake	C	Seismic Wave Propagation Simulation
187.facerec	Fortran 90	Image Processing: Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	Fortran 90	Number Theory / Primality Testing
191.fma3d	Fortran 90	Finite-element Crash Simulation
200.sixtrack	Fortran 77	High Energy Nuclear Physics Accelerator Design
301.apsi	Fortran 77	Meteorology: Pollutant Distribution

Table 3.2: SPEC 2000 Floating-Point benchmarks [67].

would be in a cold state. Simpoint [70] solves this problem by storing the state of branch predictors and caches on specific points. However, this approach assumes that the memory reference stream does not change between executions. This assumption does not hold when using prefetching as different prefetching algorithms will affect the state of the cache, thus Simpoint could not be used.

To understand the basic behaviour of the SPEC2000 benchmarks suite a simple simulation run was done with SimpleScalar. The configuration was chosen to match an Intel Pentium 4 processor as closely as possible in terms of the cache hierarchy. The simulated processor had two 8kb level 1 caches (one instruction cache and one data cache). These caches were 4-way set associative with 32 sets and 64 bytes cache lines. The unified level 2 cache was 512KB and organized as a 8-way set-associative cache with 512 sets and 128 bytes cache lines. The latency of the level 1 cache was 2 clock cycles and the level 2 latency was 7 clock cycles. Memory latency was fixed to 160 clock cycles with no bandwidth limitations. The processor was a 4-way superscalar out-of-order processor. This large issue width will ensure that most of the available ILP will be realized, and the effects of the memory subsystem can be studied. The results from running the benchmarks can be seen in table 3.5.

Table 3.3: Charachteristics of the SPEC2000 benchmarks suite.

Benchmark	IPC	# Insts	# Loads	% Loads	# Acc. L2	# Miss L2	% Miss L2	MPI
Gzip	1.6052	$5.93E+08$	$1.50E+08$	25.23%	$1.86E+07$	$1.58E+05$	0.85%	267.01
Gcc	1.1417	$5.12E+09$	$1.77E+09$	34.67%	$2.47E+08$	$5.26E+06$	2.13%	1027.48
Crafty	1.1002	$8.35E+08$	$2.61E+08$	31.27%	$7.38E+07$	$1.33E+05$	0.18%	159.51
Mcf	0.2771	$7.94E+08$	$2.53E+08$	31.87%	$7.57E+07$	$1.54E+07$	20.31%	19379.82
Swim	0.6374	$4.31E+08$	$1.01E+08$	23.51%	$1.60E+07$	$4.69E+06$	29.28%	10864.85
Mgrid	1.2834	$1.15E+08$	$3.49E+07$	30.34%	$2.32E+06$	$3.75E+05$	16.19%	3266.35
Equake	1.3092	$1.02E+09$	$2.84E+08$	27.77%	$2.27E+07$	$9.40E+05$	4.14%	919.72
Applu	1.0422	$8.82E+07$	$2.31E+07$	26.24%	$2.90E+06$	$3.08E+05$	10.63%	3496.24
Vpr	1.4494	$1.57E+09$	$4.42E+08$	28.21%	$5.21E+07$	$2.73E+03$	0.01%	1.74
Ammp	0.0901	$1.25E+09$	$3.26E+08$	26.12%	$1.47E+08$	$6.97E+07$	47.46%	55851.49
Mesa	1.825	$1.61E+09$	$3.46E+08$	21.54%	$2.16E+07$	$1.15E+05$	0.53%	71.26
Galgel	1.7781	$3.48E+08$	$1.18E+08$	33.85%	$9.37E+06$	$4.08E+04$	0.44%	117.30
Lucas	1.6807	$1.87E+08$	$3.40E+07$	18.17%	$5.32E+06$	$1.84E+03$	0.03%	9.83
Fma	1.0827	$6.68E+08$	$1.62E+08$	24.16%	$5.63E+07$	$3.44E+03$	0.01%	5.14
Parser	1.0171	$4.53E+09$	$1.27E+09$	28.00%	$9.04E+07$	$9.53E+06$	10.54%	2104.86
Eon	1.2676	$1.07E+09$	$3.22E+08$	30.08%	$4.72E+07$	$4.03E+03$	0.01%	3.76
Perlbmk	1.2433	$2.06E+09$	$6.20E+08$	30.10%	$5.10E+07$	$2.42E+05$	0.48%	117.64
Gap	1.0243	$7.61E+08$	$2.40E+08$	31.49%	$3.36E+07$	$1.90E+06$	5.64%	2491.11
Vortex	1.1337	$4.51E+05$	$1.35E+05$	29.92%	$1.70E+04$	$2.75E+03$	16.18%	6110.81
Bzip2	1.1348	$1.82E+09$	$5.09E+08$	27.96%	$2.98E+07$	$5.28E+06$	17.72%	2904.17
Apsi	1.2933	$3.40E+08$	$6.96E+07$	20.46%	$1.86E+07$	$1.59E+06$	8.57%	4685.92
Wupwise	1.3589	$5.22E+09$	$9.53E+08$	18.27%	$3.31E+07$	$8.52E+06$	25.74%	1632.39
Twolf	1.1976	$9.73E+08$	$2.56E+08$	26.32%	$5.26E+07$	$5.72E+03$	0.01%	5.88
Facerec	1.5768	$2.52E+08$	$5.65E+07$	22.38%	$1.80E+06$	$2.80E+05$	15.61%	1112.04
Art	0.1491	$1.66E+09$	$5.20E+08$	31.33%	$2.78E+08$	$9.19E+07$	33.07%	55330.91

Unfortunately, the vortex application did not work correctly due to errors in the data files. The size of each benchmark can be seen as the number of instructions executed. Although the percentage of loads is about constant in every program (20% - 35%), the number of misses in the L2 cache varies by a large amount. Suleyman Sair [71] has used misses per instruction (MPI) as a metric to characterize benchmarks. This metric provides an indication of how memory intensive an application is.

Using a limit of 3000 MPI the following benchmarks are characterized as memory intensive:

- Mcf
- Swim
- Mgrid
- Applu
- Ammp
- Apsi
- Art

These benchmarks will be used for most of the experiments due to their memory-bound nature. A few selected experiments will be run with the full suite to ensure that prefetching does not introduce regressions in compute-bound applications.

3.5.1 CMP Benchmarking

Benchmarking a CMP is much more difficult than benchmarking uniprocessors. Several new questions arise or become more difficult:

1. What benchmarks can be used?
2. If one core experience performance degradation, while the other experience a speedup, how do we measure the net result?
3. Can the performance of the *entire* system be characterized by a single number?

I will start with the first question. On uniprocessors it is very easy to use the SPEC2000 benchmark. They require little or no operating system support and precompiled binaries are available to almost every platform. In addition, it is very commonly used in the literature. The benchmarks themselves have been studied thoroughly. There are 26 benchmarks in the suite, making it possible to run them all and get a good picture of the performance of a system.

There exist numerous benchmarks that can be used for CMP systems. Some of the most commonly seen in the literature are:

- Multiple instances of SPEC2000.
- TPC-C [72]

- SPECWeb [73]
- SPECjAppServer [74]
- Linpack [75]
- NAS Parallel Benchmarks (NPB) [76]

TPC-C is a transaction processing benchmark. It is mainly used for benchmarking database performance. The main idea is to issue a large amount of queries and measure the throughput of the system in terms of completed transactions. It is important to note that the performance measure in this benchmark is TPM (Transactions per minute). Simply measuring IPC would be meaningless, as spinlocks generate a lot of committed instructions, but generates no forward progress in terms of completing transactions. In addition, TPC requires two things that SimpleScalar cannot provide:

- Full system simulation (including OS);
- Shared memory[2].

In addition, the benchmark itself introduces a lot of tuneable parameters, both at the OS level and at the database level. Another problem is that it provides only a single benchmark that produces a single number. Thus it is harder to analyze exactly what causes a speedup or a slowdown. Therefore TPC-C cannot be used as a benchmark for my purposes.

Linpack [75] is a benchmark that is used to benchmark the most powerful supercomputers in the world. The rankings in the top500 list is based on each computers performance on the Linpack benchmark. Linpack does not require a large OS (most supercomputers use small kernels as the operating system on compute nodes). However, it does require an MPI (Message Passing Interface) and a BLAS (Basic Linear Algebra Subprograms) implementation. Both BLAS and MPI are highly tuned to the underlying architecture, and would require a rewrite if the architecture changes. Thus Linpack cannot be used.

Both SPECWeb and SPECjAppServer are transaction oriented benchmarks. They are both aimed at Web-applications. Like TPC, the primary purpose ot these benchmarks is to measure througput in terms of clients served per unit of time. However, the main focus is network performance, both in terms of latency and bandwidth. Thus, these benchmarks cannot be used, for my purposes, as the main bottleneck being studied is beyond the microarchitectural level.

NPB is a benchmark suite developed by NASA designed to help evaluate the performance of parallel supercomputers. The benchmarks themselves are based on computational fluid dynamics and consists of five kernels and three applications. The benchmarks come in different flavours;

- NPB 1: Vendors can choose how to implement the programs using their own programming models.
- NPB 2: MPI based source code, that should be able to run efficiently without modification.

[2]Magnus Jahre at the NCAR group is working on an implementation of shared memory in SimpleScalar.

- NPB 3: Implementation based on OpenMP.

In addition, Grid and multi-zone versions exists. The problem with using NBP is that there exists no MPI or OpenMP implementation. In addition, writing a NBP 1 implementation from scratch would be a very large undertaking.

On the other hand, running multiple instances of the SPEC2000 benchmark is possible. The main advantages of such a solution are:

- I am familiar with how they work;
- Low setup time;
- Designed to evaluate microarchitectural performance;
- Does not require any special OS support or libraries.

However, the SPEC2000 benchmarks are designed to run on uniprocessors. Simply running multiple instances would become unrealistic, as there is no communication between cores. Thus, this choice in benchmarks limits the research to systems with no shared memory or cache coherence problems.

Using multiple SPEC2000 benchmarks is a technique that is very common in SMP / SMT and CMP research, mainly for the above reasons. There is a dramatic shortage of proper mircoarchitectural benchmarks for CMP systems.

Running every combination of the SPEC2000 benchmarks would yield $26^4 = 456976$ experiments for each configuration. This would require an enourmous amount of time, and cannot be considered. A better solution would be to simply use a subset of all the possible permutations and use statistics to evaluate the performance.

However, the statistical distribution of the benchmarks is not known, and one cannot simply assume a gaussian distribution. Thus it is very hard to measure the confidence interval of any given result. A possibility is to simply assume that a “large enough” sample would result in a “small enough” confidence interval.

A weak statistical test that can be used is to simply use a binary result; simply record how *many* cases experience a performance increase compared to how many cases experience a performance degradation. Then the binomial distribution can be used to determine if one configuration is truely better than another in a statistically significant manner.

Characterizing performance with a single number

Measuring performance is a tricky task even for a single processor system. In a CMP system the problem gets more complicated. A change in the architecture might improve performance for some benchmarks while degrade performance for others. When this happens to a CMP, how can we measure the effect on the entire system?

But first, what is performance? James Smith uses the following guideline [77]:

> The time required to perform a specified amount of computation is the ultimate measure of computer performance.

In other words, how much faster can the work be done, if I switch systems?

In my setup I will run several SPEC2000 benchmarks on the simulated CMP system. Some of the benchmarks will experience a performance degradation due to prefetching, while others will experience a speedup. How do you measure the overall effect on system performance?

A possible solution is to simply average the results. The mathematical formula for averaging numbers is well known, and is shown in equation 3.2. In this equation n is the number of programs being averaged and M_i is the running time of the program.

$$A = \frac{1}{n}\sum_{i=1}^{n} M_i \tag{3.2}$$

This method is known as the arithmetic mean. However there are other ways to produce a single number. One can also use the geometric mean (equation 3.3) or the harmonic mean (equation 3.4) [77, 78].

$$G = \sqrt[n]{\prod_{i=1}^{n} M_i} \tag{3.3}$$

$$H = \frac{n}{\sum_{i=1}^{n} \frac{1}{M_i}} \tag{3.4}$$

The use of the geometric mean is questionable. The single number produced by such an operation should be indicative of end user application performance. However, it can be shown that geometric mean does not predict actual performance in a meaningful way. The paper by Smith has numerous examples where this is proven [77].

The harmonic mean is useful for averaging rates [79]. As an example; consider a car driving at 50 km/h. It's destination is 100km away, and will therefore spend 2 hours getting there. On the return trip, the car drives at 100 km/h, and thus spends 1 hour on the return back home. If one takes the arithmetic means of the two speeds one obtains:

$$A = \frac{50 + 100}{2} = 75 \tag{3.5}$$

Which implies that the average speed is 75km/h. However, the total distance travelled is 200km and the driver spends 3 hours in the car, thus the equivalent speed is:

$$A = \frac{200}{3} = 66.67 \tag{3.6}$$

Which is obviously a more informative value. The harmonic mean calculates this value directly:

$$H = \frac{2}{\frac{1}{50} + \frac{1}{100}} = 66.67 \tag{3.7}$$

To summarize, the arithmetic mean is useful for evaluating performance measured in time (Execution time), while harmonic mean is useful for evaluating performance measured in rates (Mflops/IPC).

At last, it is worth noting that normalization should be done *after* averaging performance. If normalization is performed before averaging, the results will become erroneous [78].

Chapter 4

Results

4.1 Overall Plan for the Experiments

This chapter describes the results obtained with the simulator. In computer architecture research there is a large degree of freedom, therefore it is natural to limit the number of parameters to be studied. The fixed parameters are presented in section 4.2.

The experiments presented in this chapter use an iterative approach, the first experiments are performed on an uniprocessor with no bandwidth limitations (section 4.3.1). As mentioned in section 2.2, prefetching data can cause two ill effects:

1. Useful data is evicted from the cache.

2. Prefetching consumes bandwidth.

By eliminating effect two, we can examine effect one directly. The purpose of the first batch of experiments is to simply evaluate effect number one. Some sensitivity analysis is performed to better understand the prefetching heuristics, and for selecting good values for future experimentation.

In section 4.3.2 limited bandwidth is introduced. This allows us to evaluate effect two, since we have data about effect one. This work will mainly focus on benchmarks that are memory-bound.

In section 4.4 we look at prefetching in a CMP environment. This work will be based on the previously obtained uniprocessor results.

4.2 Experimental Setup

This section describes the parameters that I have used in my experiments. The parameters were chosen to match an aggressive 4-way superscalar core. The number of functional units as well as the size of the load/store queue were chosen accordingly. In addition, a more accurate branch predictor was chosen (rather than the default bimodal predictor). The cache hierarchy used is the same as for the Pentium 4. The size of the L2 is small compared to contemporary high-end processors. This choice was deliberate as it would force more capacity misses as well as conflict misses. This is important as the relatively small footprints of the lgred benchmarks would fit entirely in a larger cache.

The default DRAM parameters are taken from section 3.1.2. The setup is summarized in table 4.1.

Parameter	Value
Clock speed	4 GHz
Register Update Unit size	16 instructions
Load/Store Queue size	8 instructions
Fetch Queue size	4 instructions
Fetch, Issue, Decode and Commit width	4 instructions/cycle
Functional units	4 Integer ALU, 1 Integer Multiply/Divide 4 Floating Point ALU 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
Branch misprediction penalty	15 clock cycles
Translation Lookaside Buffer	128 entry full associative (both data and instructions) 30 cycle miss penalty
Level 1 Data cache	8KB 4-way set associative, 64B blocks, LRU replacement policy, 2 cycles latency
Level 1 Instruction cache	8KB 4-way set associative, 64B blocks, LRU replacement policy, 2 cycles latency
Level 2 Unified cache	512KB 8-way set associative, 128B blocks, LRU replacement policy 7 cycles latency
DRAM	2 channels, 40 clock cycle command transfer, 40 clock cycles access time, 80 clock cycles transfer time, Double Data Rate @ 400Mhz,

Table 4.1: Simulation parameters.

4.3 Uniprocessor

Before experimenting with CMP configurations this section will examine each prefetching heuristic in detail. The purpose is to establish a thorough understanding of how each heuristic work.

In the next section we examine how prefetching works on a uniprocessor where there is no bandwidth limitations, in order to examine how increasing prefetching degree interacts with system performance. In section 4.3.2, I will examine how limiting bandwidth affects prefetching. The main focus of that section is to examine how limited bandwidth interacts with the prefetching degree for different heuristics on selected benchmarks. In the last section (4.3.3) I will explore the performance of my own contribution in a uniprocessor context. I will explore how bandwidth aware prefetching interacts with performance, both in terms of IPC and bandwidth usage.

4.3.1 Unlimited Bandwidth

Prefetching can potentially cause performance degradation. This effect can be caused by two issues as stated in section 2.2.

1. Displacing useful data already in the cache.
2. Displacing other useful memory requests on the system bus.

To quantify the effects, a simple experiment was run with no bandwidth limitations. The prefetching degree was set to 1 for every algorithm. In addition, the tables were set to 1024 entries for every type (RPT, Stream, DC and C/DC). The results can be seen in figure 4.1.

In this experiment we see a lot of variation in terms of the effect of adding prefetching. By comparing no prefetching to the 'perfect L2' case we see that some benchmarks can potentially have large speedups due to prefetching. Other benchmarks get almost no benefit of a perfect L2. This corroborates the analysis made in section 3.5. In that section, we observed that some benchmark have relatively few L2 misses per instruction. These benchmarks will therefore gain little from both prefetching and a perfect L2. By removing the benchmarks that receive little or no effect from a perfect L2 we obtain the graph in figure 4.2.

We see that stream prefetching is by far the most efficient prefetcher. In one case (art) it outperforms the perfect L2. This might sound surprising, but stream prefetching prefetches to the L1 cache, thus achieving a small speedup.

Although this experiment is run with a prefetching degree of 1, we see significant improvements with all prefetching algorithms. The performance in terms of IPC is almost always at least equal to the case were no prefetching is performed. This leads to the interpretation that the effect of prefetching displacing other useful data in the cache is relatively small compared to the benefit.

We see that AVD prefetching is not very effective, only a few benchmarks show any speedup at all. This is due to the way AVD prefetching works. SPEC2000 does not contain pointer intensive code, thus the effect would be minimal (the original paper describing AVD used SPEC95 benchmarks [9]). In addition, AVD prefetching cannot prefetch with enough distance to be truly useful.

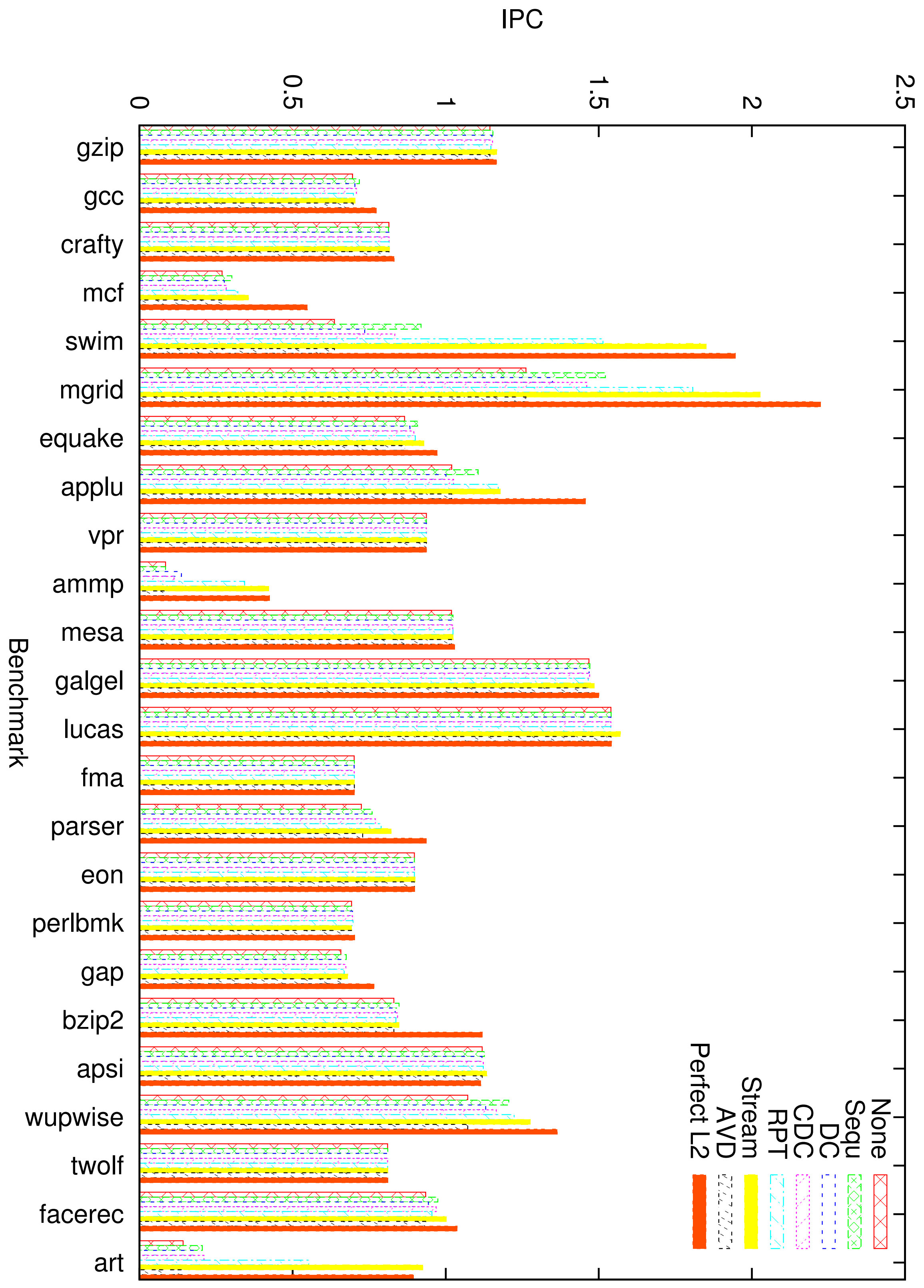

Figure 4.1: Performance of prefetching on processors with unlimited bandwidth.

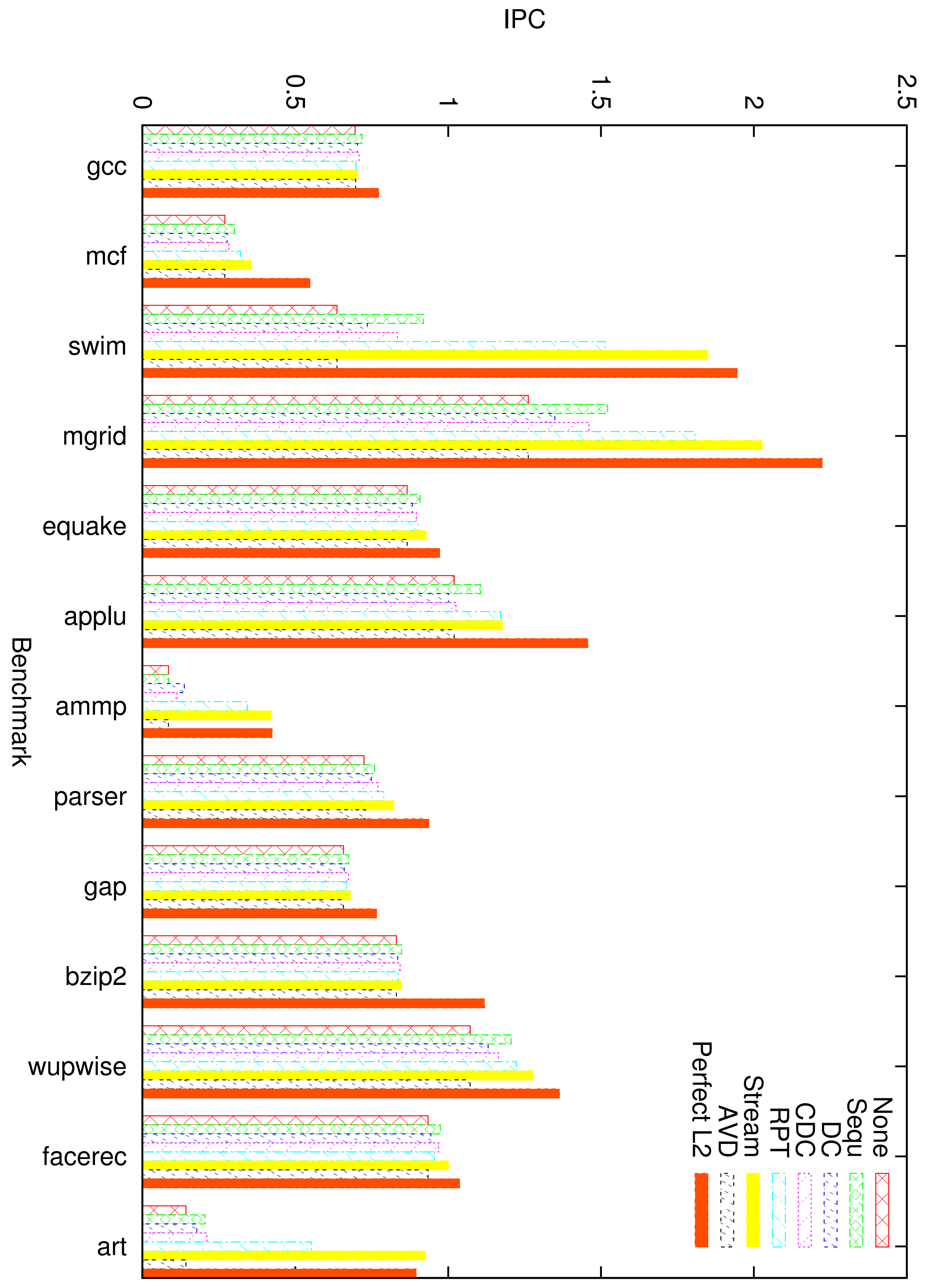

Figure 4.2: Similar to figure 4.1, but with uninteresting benchmarks removed.

We also see that there is a lot of potential for prefetching, as there is a large performance gain when using a perfect L2. We see that the stream- and RPT- prefetchers perform very well in most cases. This is not surprising, given the large amount of information used by these prefetchers.

In figure 4.3 a similar experiment is run, but with a prefetching degree of 8, again, the uninteresting benchmarks are removed. In this experiment we observe that increasing the prefetching degree generally increases the performance benefit of prefetching. This is especially true for the simpler algorithms such as sequential prefetching. We observe that the performance of Mcf is decreased by 8% by using sequential prefetching with a prefetching degree of 8, which was the observed maximum. However, only 6% of the experiments showed a performance degradation. In figure 4.4 we compare the performance of the heuristics with a prefetching degree of one. For sequential prefetching on Swim we observe a performance increase by a factor of 1.7 in terms of IPC. On Art, the speed increase is 138% for sequential prefetching. A similar effect can be seen on Art as well. As in the previous experiment we see little performance degradation due to the effects of prefetching, although more blocks are prefetched. Mcf has the largest speed degradation due to increasing the prefetching degree. This is a relatively minor change, but it is important to remember when moving to CMP, because of the increased pressure on the cache.

The accuracy and coverage of the heuristics are interesting, because they indicate how successful the heuristics are at predicting memory accesses. In table 4.2 the accuracy of each prefetching algorithm is listed. We see that the RPT and Stream prefetchers are very accurate, close to 100 % for many benchmarks. This is not surprising considering the complexity of these heuristics. In addition, we observe that DC prefetching is generally less accurate than sequential prefetching, which is somewhat surprising, given the relatively simple design of the sequential heuristic. However, C/DC prefetching is much more accurate than both of them, and is much more robust across benchmarks. AVD prefetching has a very low accuracy, as most of these benchmarks are not pointer-intensive. Also note that the accuracy for sequential prefetching on Ammp is wrong. The performance counters that were used to count prefetches rolled over and thus produced the wrong results.

In table 4.3 the coverage of each heuristic is listed. Again, AVD performs very badly, it detects less than 7% of the prefetching opportunities in every benchmark. However, it is clear that all heuristics can be improved given the relatively low coverage across all benchmarks. It is interesting to note that on the Art, Perlbmk and Ammp benchmark for the RPT and Stream prefetchers, both coverage and accuracy is above 98%. Such an impressive result will be hard to outperform.

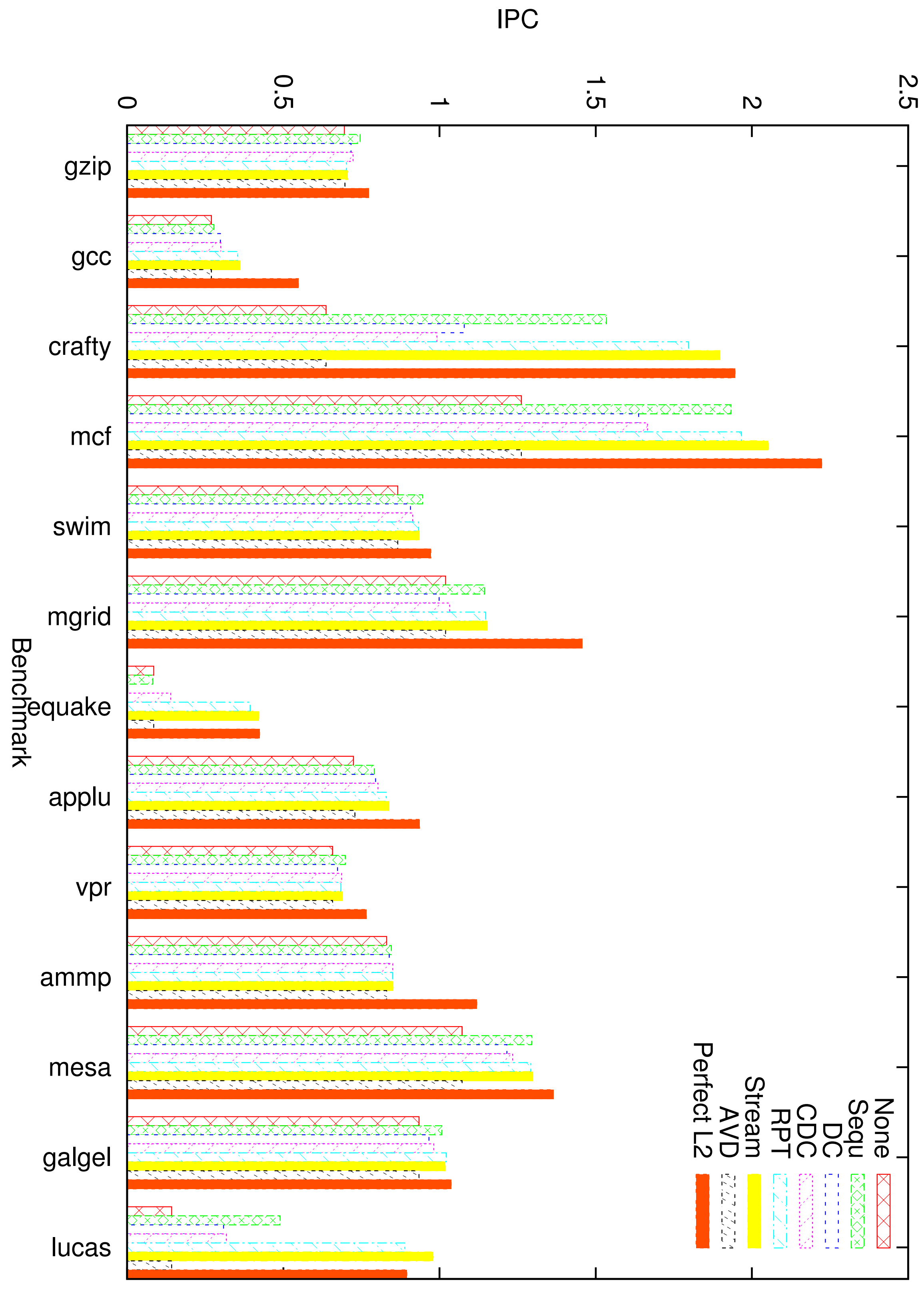

Figure 4.3: Performance of prefetching on processors with unlimited bandwidth. Prefetching degree is set at 8.

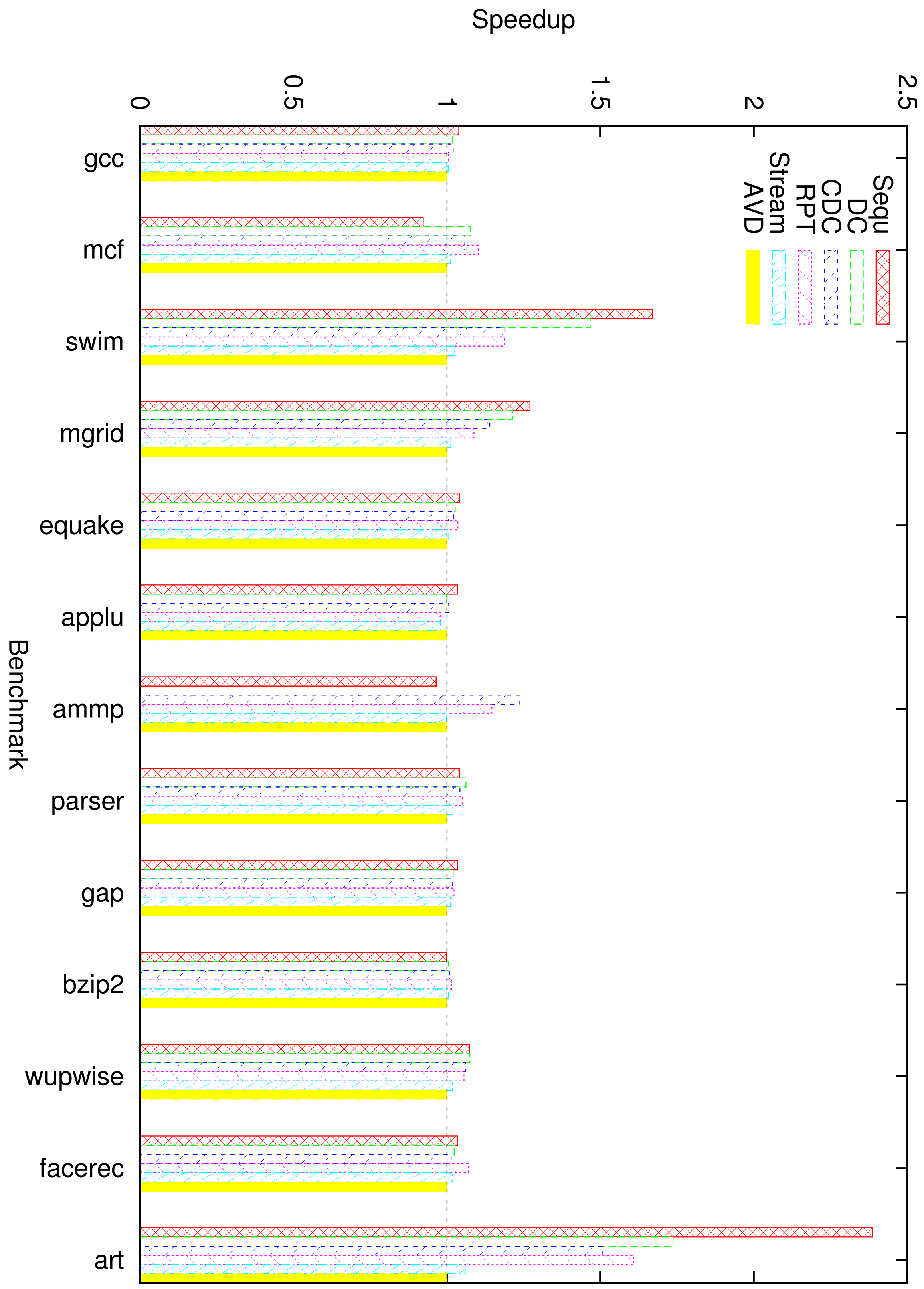

Figure 4.4: Speedup in IPC by increasing the prefetching degree from 1 to 8.

Benchmark	Sequential	DC	C/DC	RPT	Stream	AVD
gzip	72.0 %	40.7 %	90.0 %	99.9 %	99.4 %	38.8 %
gcc	69.4 %	54.7 %	78.5 %	95.6 %	89.4 %	35.5 %
crafty	25.5 %	27.0 %	86.4 %	96.7 %	89.3 %	22.1 %
mcf	72.6 %	49.9 %	91.9 %	99.5 %	98.9 %	64.6 %
swim	80.6 %	49.9 %	85.7 %	98.3 %	94.1 %	30.4 %
mgrid	90.3 %	52.9 %	95.3 %	98.8 %	99.4 %	15.8 %
equake	76.7 %	54.5 %	93.0 %	99.6 %	99.5 %	78.2 %
applu	74.9 %	42.0 %	64.4 %	76.9 %	51.2 %	53.8 %
vpr	90.7 %	87.6 %	88.6 %	97.9 %	97.8 %	22.2 %
ammp	0.3 %	99.2 %	99.6 %	99.5 %	98.4 %	10.8 %
mesa	78.6 %	41.2 %	88.8 %	99.7 %	99.7 %	41.9 %
galgel	57.5 %	60.3 %	85.9 %	81.0 %	70.5 %	39.1 %
lucas	85.5 %	84.0 %	81.3 %	63.6 %	64.9 %	40.0 %
fma	76.1 %	58.8 %	66.0 %	98.1 %	88.5 %	62.5 %
parser	60.8 %	47.8 %	90.9 %	99.5 %	98.5 %	36.0 %
eon	82.0 %	57.6 %	75.7 %	89.2 %	85.2 %	11.1 %
perlbmk	9.0 %	96.2 %	94.1 %	99.5 %	99.5 %	11.6 %
gap	60.2 %	25.6 %	85.0 %	98.4 %	95.6 %	9.7 %
bzip2	32.0 %	48.6 %	85.9 %	97.5 %	95.0 %	39.5 %
apsi	93.3 %	24.0 %	88.9 %	99.5 %	94.9 %	34.8 %
wupwise	88.6 %	46.3 %	90.8 %	96.3 %	85.5 %	31.6 %
twolf	86.8 %	78.0 %	87.9 %	90.8 %	91.2 %	17.1 %
facerec	86.5 %	52.5 %	94.6 %	97.0 %	95.3 %	41.9 %
art	99.1 %	50.5 %	97.5 %	99.9 %	99.6 %	99.9 %

Table 4.2: Accuracy of prefetching heuristics. Values larger than 90% are marked with green, while values smaller than 50 % are marked with red.

Benchmark	Sequential	DC	C/DC	RPT	Stream	AVD
gzip	47.7 %	28.1 %	44.1 %	29.4 %	29.2 %	0.0 %
gcc	39.3 %	15.4 %	22.0 %	5.1 %	5.7 %	4.3 %
crafty	23.5 %	5.0 %	8.0 %	11.2 %	11.4 %	0.0 %
mcf	39.9 %	16.0 %	17.8 %	49.7 %	50.6 %	0.6 %
swim	47.1 %	28.4 %	38.8 %	67.4 %	67.5 %	0.0 %
mgrid	47.5 %	28.3 %	39.0 %	54.2 %	57.6 %	0.0 %
equake	45.0 %	20.0 %	33.4 %	43.8 %	54.7 %	0.0 %
applu	46.1 %	12.1 %	17.0 %	46.8 %	52.0 %	0.0 %
vpr	41.0 %	19.0 %	23.5 %	16.9 %	22.3 %	0.1 %
ammp	0.3 %	47.2 %	30.7 %	97.9 %	97.2 %	0.0 %
mesa	49.6 %	30.1 %	44.8 %	70.8 %	70.8 %	0.0 %
galgel	41.9 %	11.2 %	11.9 %	54.2 %	70.7 %	0.0 %
lucas	43.4 %	22.0 %	24.4 %	1.9 %	2.6 %	0.2 %
fma	40.4 %	13.7 %	18.3 %	3.0 %	3.4 %	0.1 %
parser	32.9 %	20.5 %	29.4 %	47.7 %	51.3 %	6.3 %
eon	40.6 %	8.1 %	18.2 %	6.0 %	8.2 %	0.0 %
perlbmk	8.4 %	30.7 %	33.0 %	98.6 %	98.6 %	0.0 %
gap	35.4 %	11.5 %	30.9 %	12.2 %	15.6 %	0.5 %
bzip2	23.3 %	2.7 %	5.9 %	7.4 %	7.5 %	0.0 %
apsi	49.9 %	22.2 %	46.8 %	0.8 %	0.9 %	0.0 %
wupwise	49.1 %	24.6 %	40.7 %	48.8 %	49.5 %	0.0 %
twolf	37.0 %	19.0 %	23.0 %	3.4 %	16.2 %	0.1 %
facerec	45.4 %	28.2 %	41.2 %	72.9 %	105.4 %	0.0 %
art	49.7 %	29.3 %	47.3 %	99.6 %	99.4 %	0.0 %

Table 4.3: Coverage of prefetching heuristics. Values larger than 70% are marked with green, while values smaller than 5 % are marked with red.

Prefetching Degree

In this section I will examine how increasing the prefetching degree affects the performance of prefetching. I have selected three benchmarks for this experiment. First, I have only chosen benchmarks with a significant potential for prefetching (the difference between no prefetching and a perfect L2 is significant). Then I have randomly selected Mgrid, Swim and Art. More experimentation was not required as all experiments point in the same direction. In addition, this facet of prefetching will be explored in other sections of this thesis.

The results of these experiments can be seen in figures 4.5, 4.6 and 4.7. The most interesting results from these graphs is that performance increases monotonously with increasing prefetching degree. In other words, an increase in prefetching degree does not decrease performance for these benchmarks. This enhances the finding in the previous section regarding data displacement in the cache, where only one benchmark had performance degradation when increasing the prefetching degree.

Further, we see that stream and RPT prefetching reaches their maximum potential quite early. This effect can be most clearly observed in figure 4.6. We also observe that DC and C/DC prefetching does not gain a large benefit from high prefetching degrees. This is due to the matching function in the delta correlation, which limits the maximum prefetch degree available. We also observe that sequential prefetching gains the most benefit from increasing the prefetching degree.

We observe again that stream prefetching outperforms the perfect L2. As before, this is due to the fact that stream prefetching fetches blocks to the L1 cache rather than the L2 cache. Although understandable, this is an impressive result.

In figure 4.7 we observe an interesting thing as the curves for CZone/DC and DC crosses each other. This is simply due to DC issuing more prefetch requests.

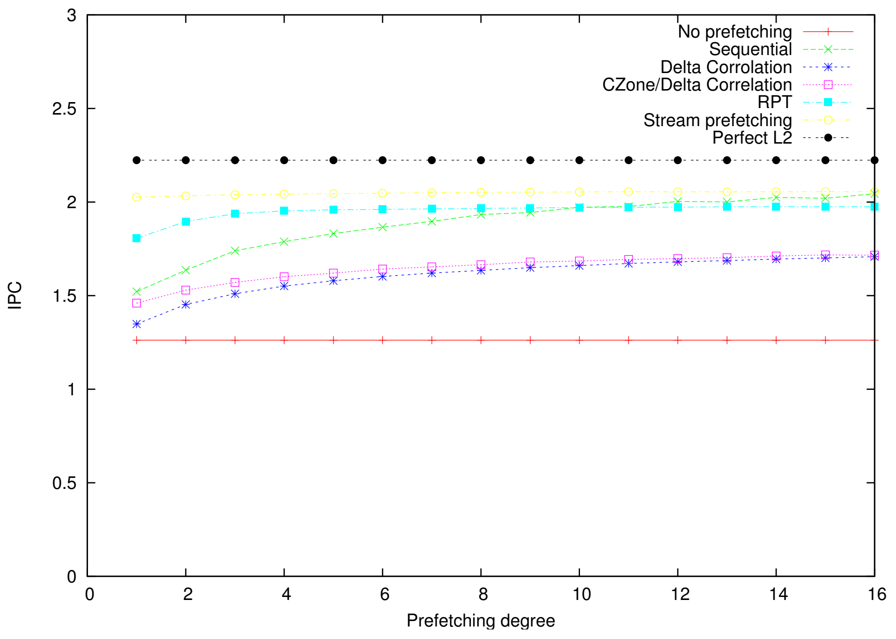

Figure 4.5: Increasing prefetching degree on Mgrid.

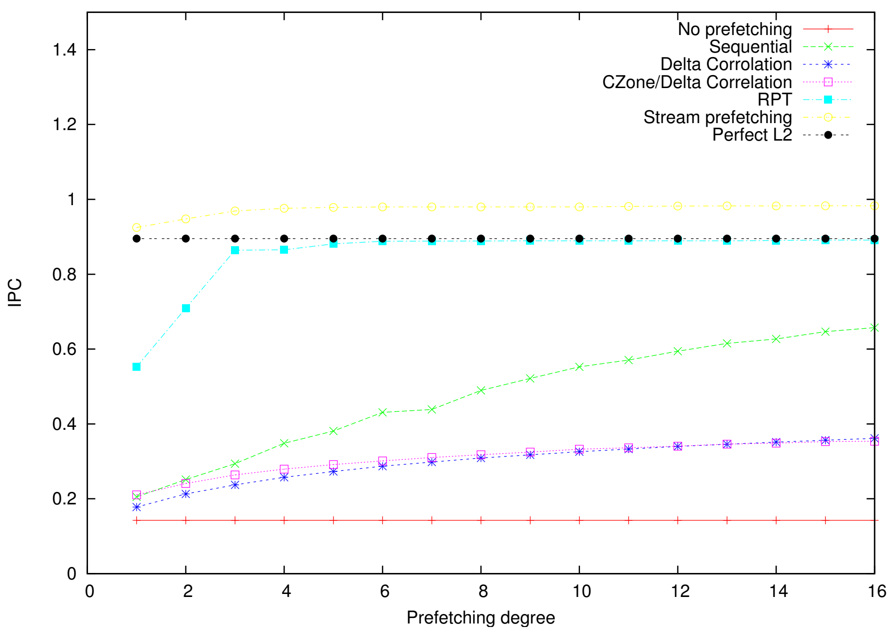

Figure 4.6: Increasing prefetching degree on Art.

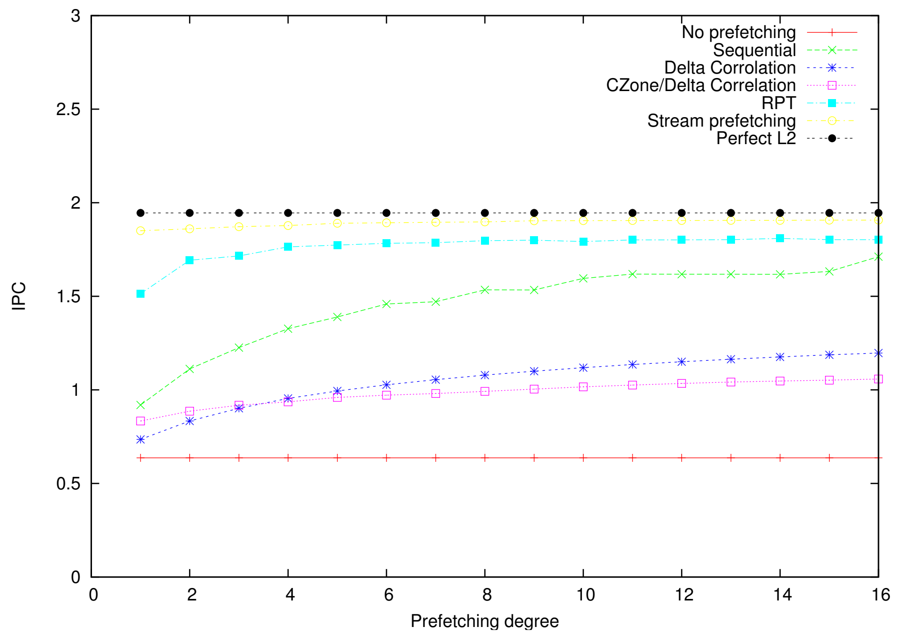

Figure 4.7: Increasing prefetching degree on Swim.

CZone Size

CZone/Delta correlation prefetching have an additional parameter that needs to be examined; namely the size of each CZone (or memory division). I have examined this in 4 different benchmarks; Art, Mcf, Mgrid and Swim. Mcf was added because it produced some interesting results in the original paper [25]. In that paper, increasing the prefetching degree generally decreases performance on Mcf. In addition, increasing the CZone size decreases performance as well, until the CZone size is set to 16MB. If the CZone size is larger or equal to the size of the datasets, C/DC will perform as the DC algorithm.

The experiment was run with the baseline setup and with C/DC prefetching. By varying the prefetching degree and CZone size the results in figures 4.8, 4.9, 4.10 and 4.11 were obtained.

From these experiments we see that too small CZone sizes causes a performance degradation, because 4K is to small for the pattern detection to function properly. This is also the case for too large CZones. However, it is generally better to have too large CZones than too small CZones. In Mgrid, we observe that any size beyond 64K gives acceptable results. This is also true for Swim, which peaks at about 256K CZone size. However, for Mcf, this does not hold.

Based on these experiments an CZone size of 256K was chosen for the remainder of the experiments. This result matches the results found in the original paper by Nesbit.

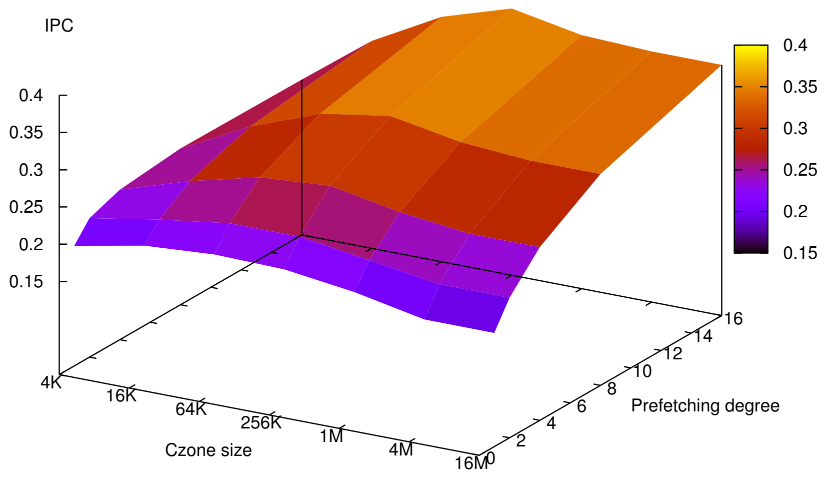

Figure 4.8: Varying CZone size with prefetching degree on C/DC for Art.

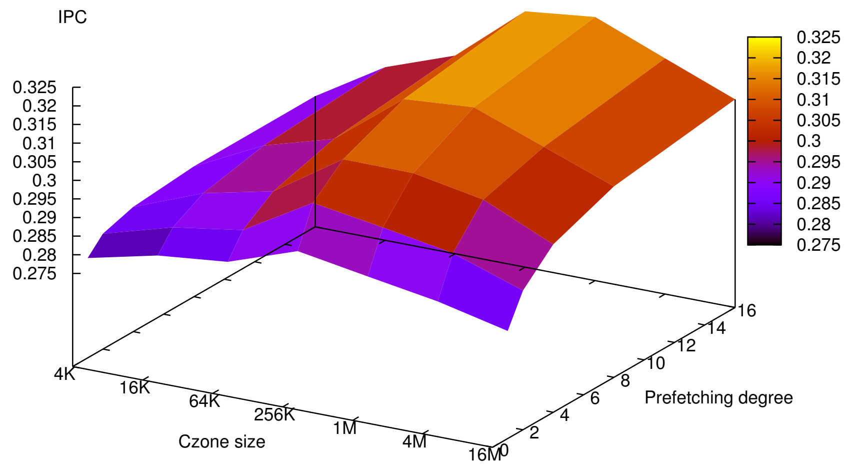

Figure 4.9: Varying CZone size with prefetching degree on C/DC for Mcf.

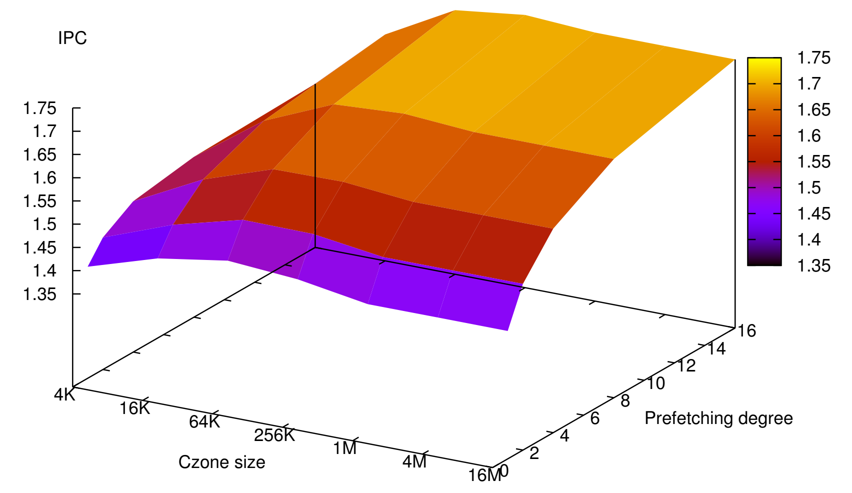

Figure 4.10: Varying CZone size with prefetching degree on C/DC for Mgrid.

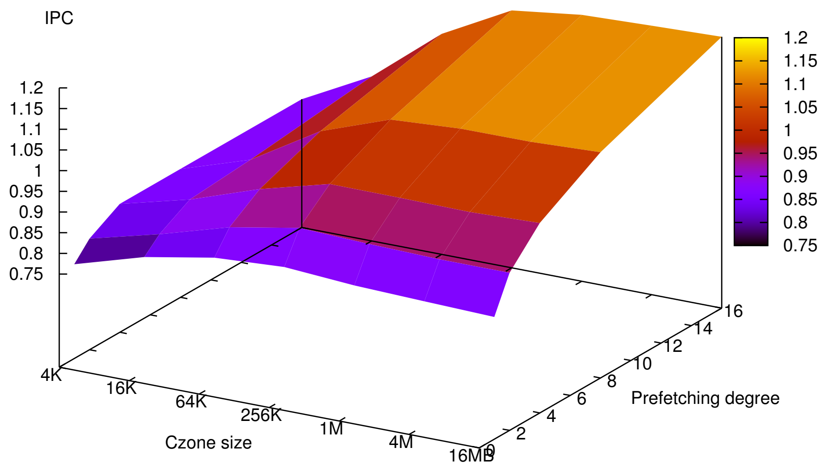

Figure 4.11: Varying CZone size with prefetching degree on C/DC for Swim.

Table Size

Many of the prefetching heuristics studied in this thesis require storage. This section examines the optimal size of the tables. However, it is important to note that the tables are different in complexity. A table entry might vary in size between the different implementations, thus the comparison is not fair. A fair comparison would require a comparison between area requirements. However, it is too simplistic to simply assume that area increases linearly with table size (in bits). Some heuristics require cache-like structures, which increases in area according to the findings in section 2.1.1, while others require FIFO structures. To estimate the area requirements would require an actual VHDL or Verilog model of the prefetches, which is beyond the scope of this thesis.

However, the main purpose of this experiment is to find a table size that is large enough to cover most cases, such that a prefetching algorithm is not cut off due to resource issues.

The experiment was run with a prefetching degree of 1. By varying the size of the table we get the results in figures 4.12, 4.13, 4.14, 4.15 and 4.16.

In figures 4.12 and 4.14 we see that there is little significant change in performance by increasing the table size. A small increase is gained when using DC prefetching and going from a 16 entry to a 32 entry table.

In figure 4.13 a more dramatic increase is observed. We see a clear increase in performance until the table can contain 128 load instructions. This graph also indicates some of the structure of the program, suggesting that there is somewhere between 32 and 128 loads between each delinquent load.

Such a sharp increase can also be observed in figures 4.15 and 4.16. A table size of 1024 entries was chosen because it covers most of the cases, although more performance can be gained by using a larger table.

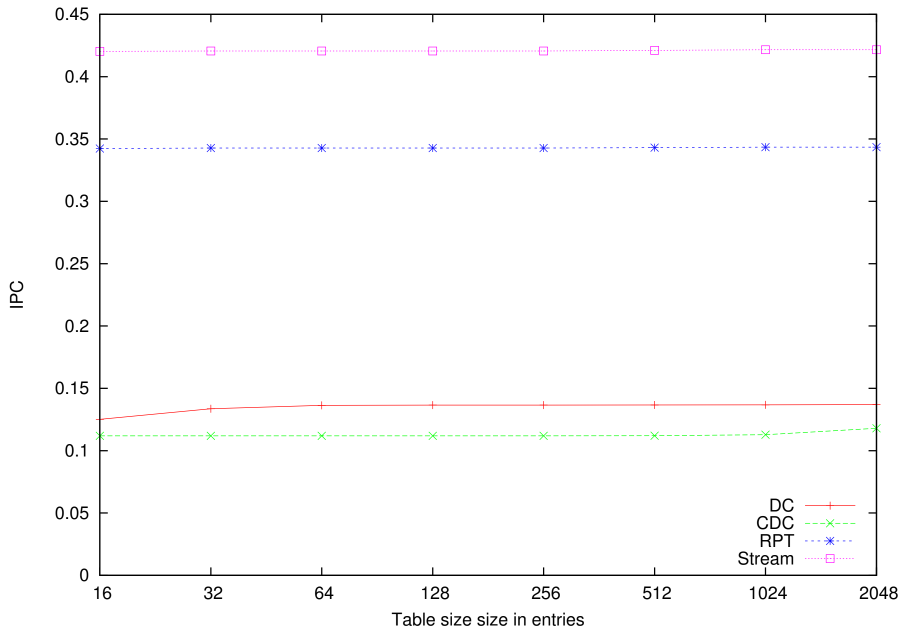

Figure 4.12: Varying Table size on Ammp.

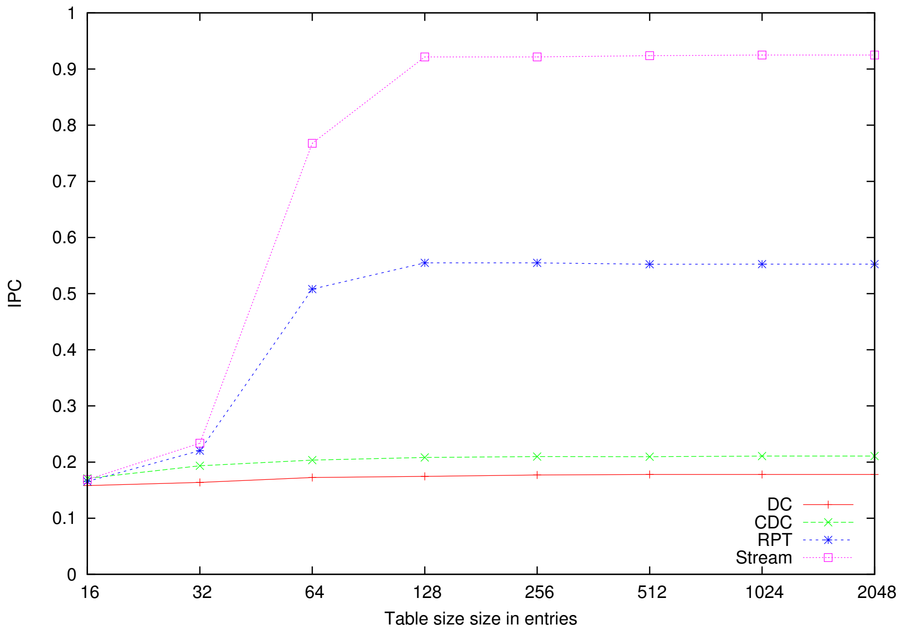

Figure 4.13: Varying Table size on Art.

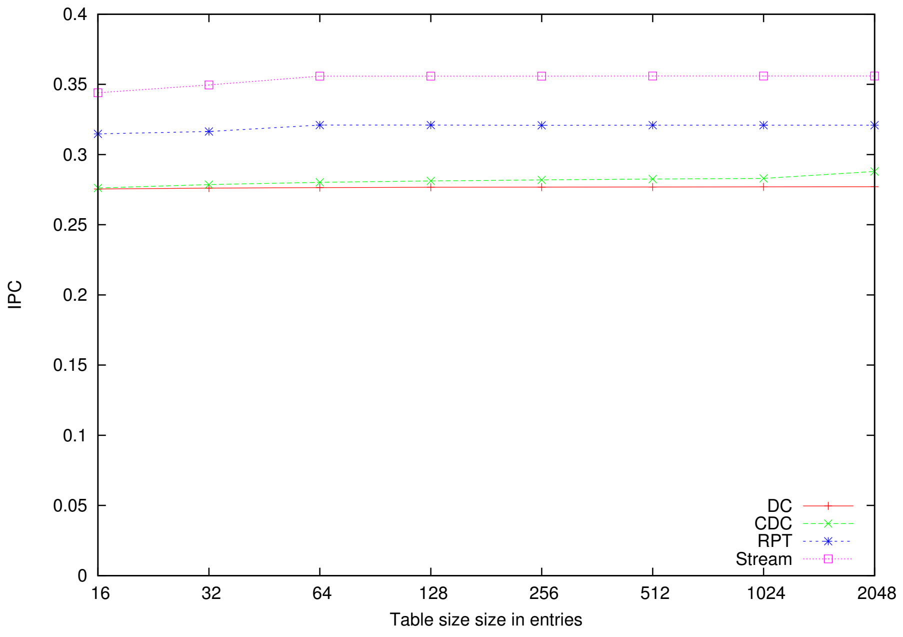

Figure 4.14: Varying Table size on Mcf.

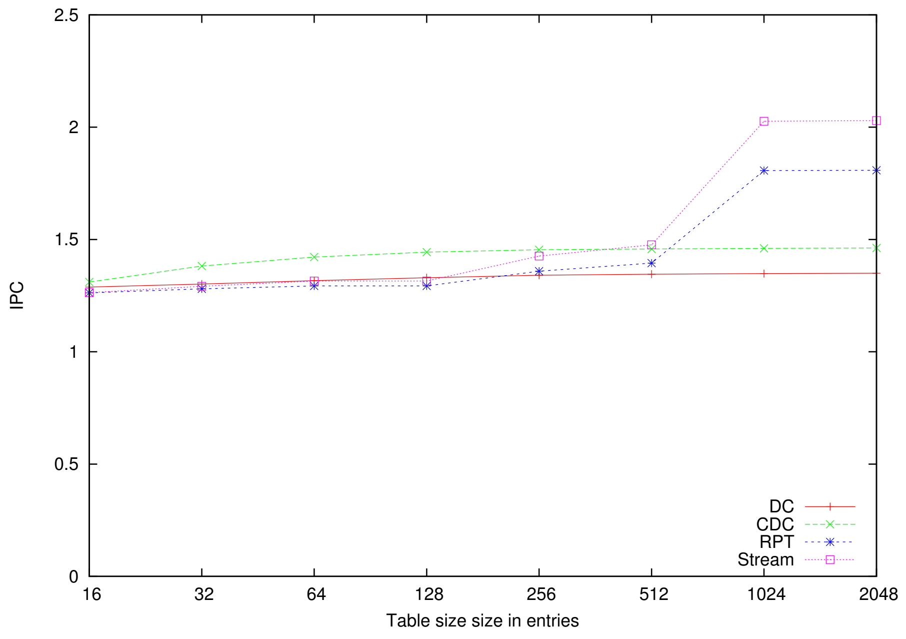

Figure 4.15: Varying Table size on Mgrid.

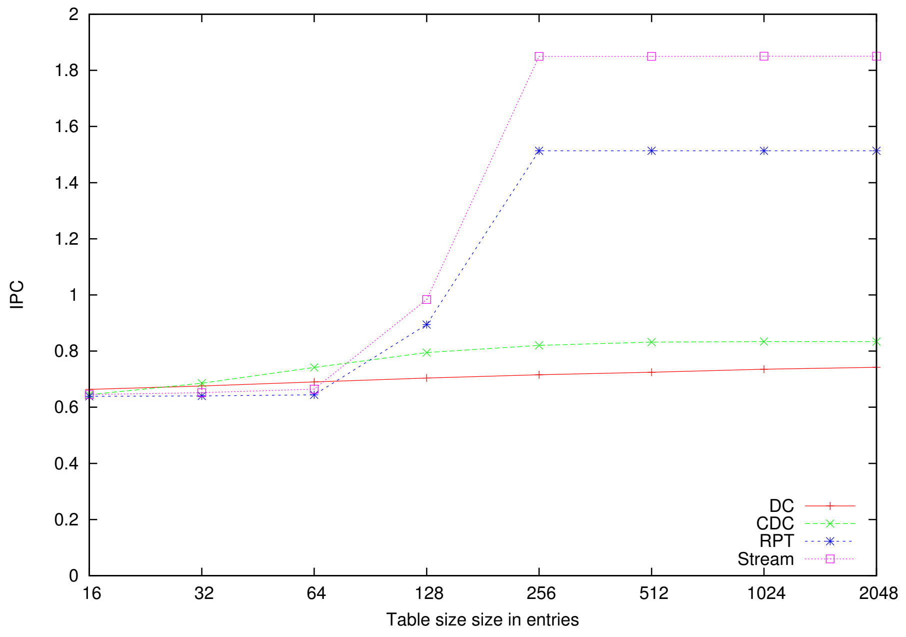

Figure 4.16: Varying Table size on Swim.

4.3.2 Limited Bandwidth

By adding the new DRAM model described in section 3.1 we can study the effect of limited bandwidth on performance. In the previous section we looked how prefetching without any bandwidth limitation affected performance. We now have an idea of how prefetching displaces data in the cache. This information can then be applied in the analysis of prefetching in an environment where bandwidth is limited.

First, we run a simple experiment similar to the one conducted in the previous section. The experimental setup is unchanged, with the exception of the bandwidth limitation.

In figure 4.17 the results from the experiment are documented. In this experiment we look at how performance is affected by limiting bandwidth to a single DRAM channel. It is interesting to compare this to the unlimited version of the same experiment in figure 4.1. To make comparison easier the graph in figure 4.18 was made. This figure shows the speedup of using the new DRAM model using one channel as opposed to the unlimited previous model. Benchmarks that show little (< 1.5 %) performance increase or decrease were omitted.

As noted in section 3.1.4, the effect of using the new DRAM model is varied, some benchmarks experience a speedup due to the effects of open pages (the Ammp benchmark), while others show degradation. Swim and Apsi show significant performance degradation due to the reduced bandwidth available. This is especially true for more bandwidth intensive methods. This indicates that bandwidth contention is a more dominant factor than cache line displacement.

In figure 4.19 we see the results from a similar experiment, but with 2 DRAM channels. Again, the speedup of using the new DRAM model over the old one is shown in figure 4.20. Naturally, we see a performance benefit as more bandwidth is available for prefetching. This is especially true for benchmarks such as Apsi and Swim, which were previously bandwidth limited. More bandwidth increases performance compared to the previous experiment. In addition we observe that the effect of open pages has an especially good effect on sequential prefetching as well as programs that exhibit sequential access patterns.

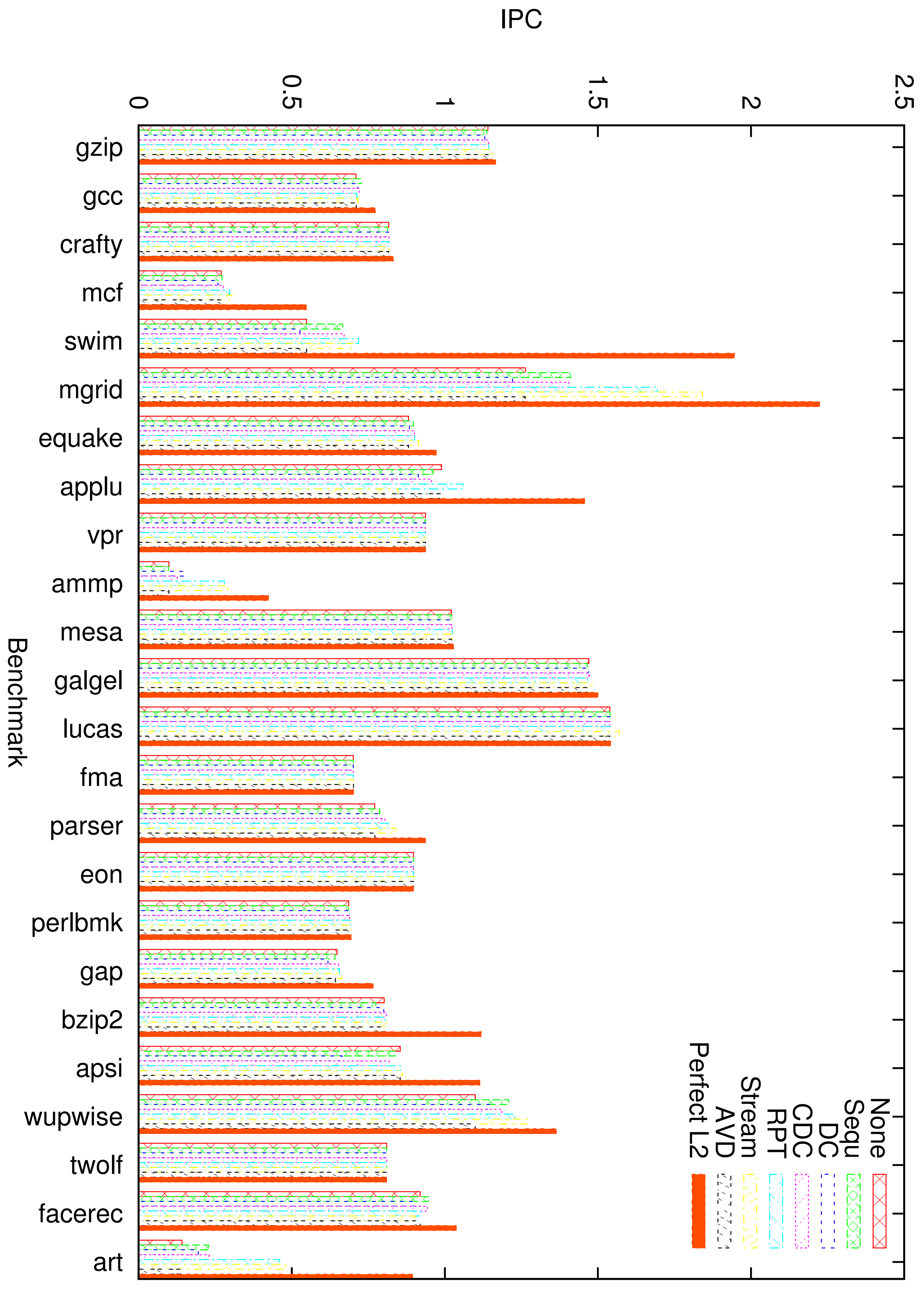

Figure 4.17: Baseline with 1 dram channel.

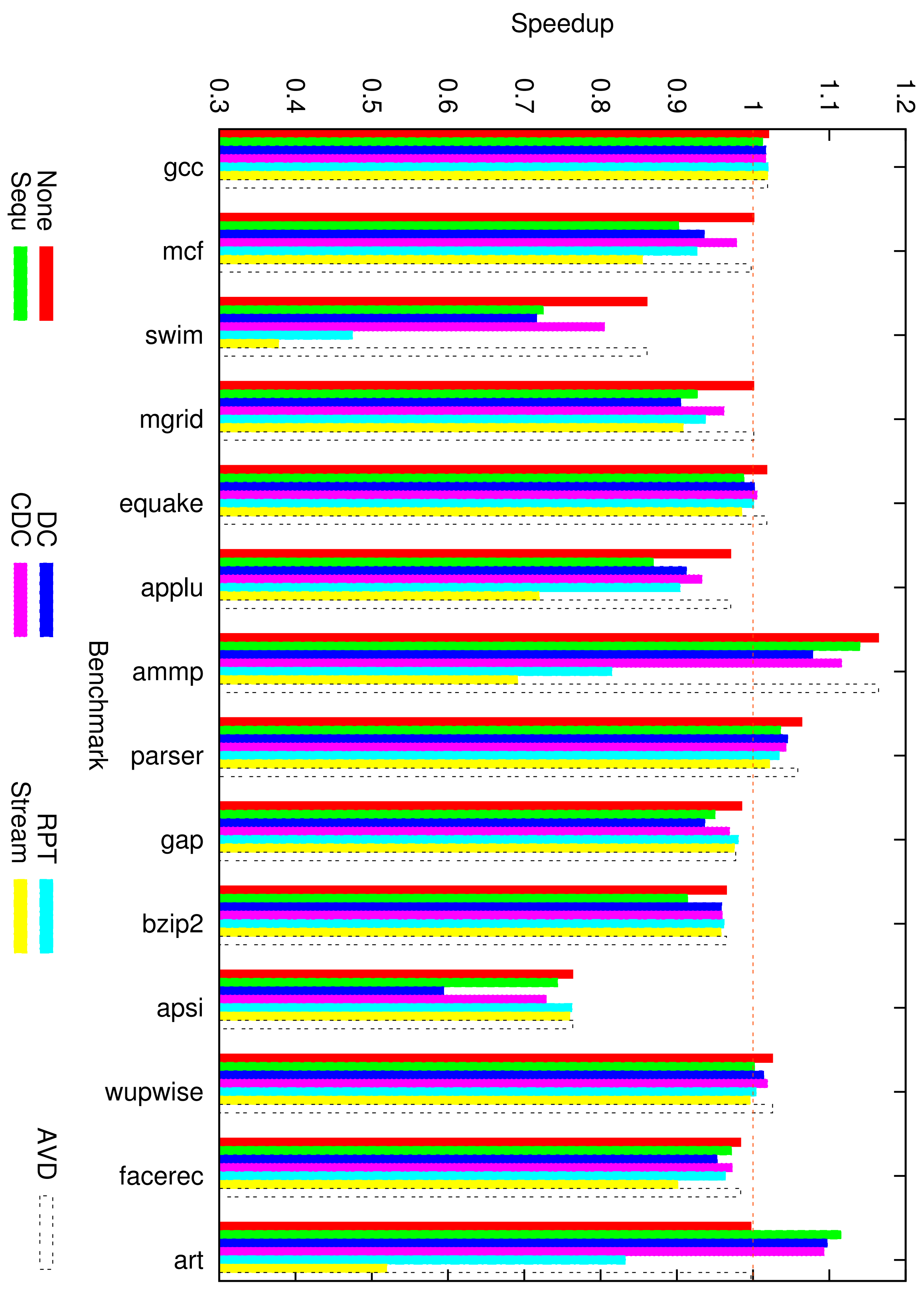

Figure 4.18: Speedup using 1 channel over the unlimited bandwidth model. Benchmarks with less than 1% difference has been removed.

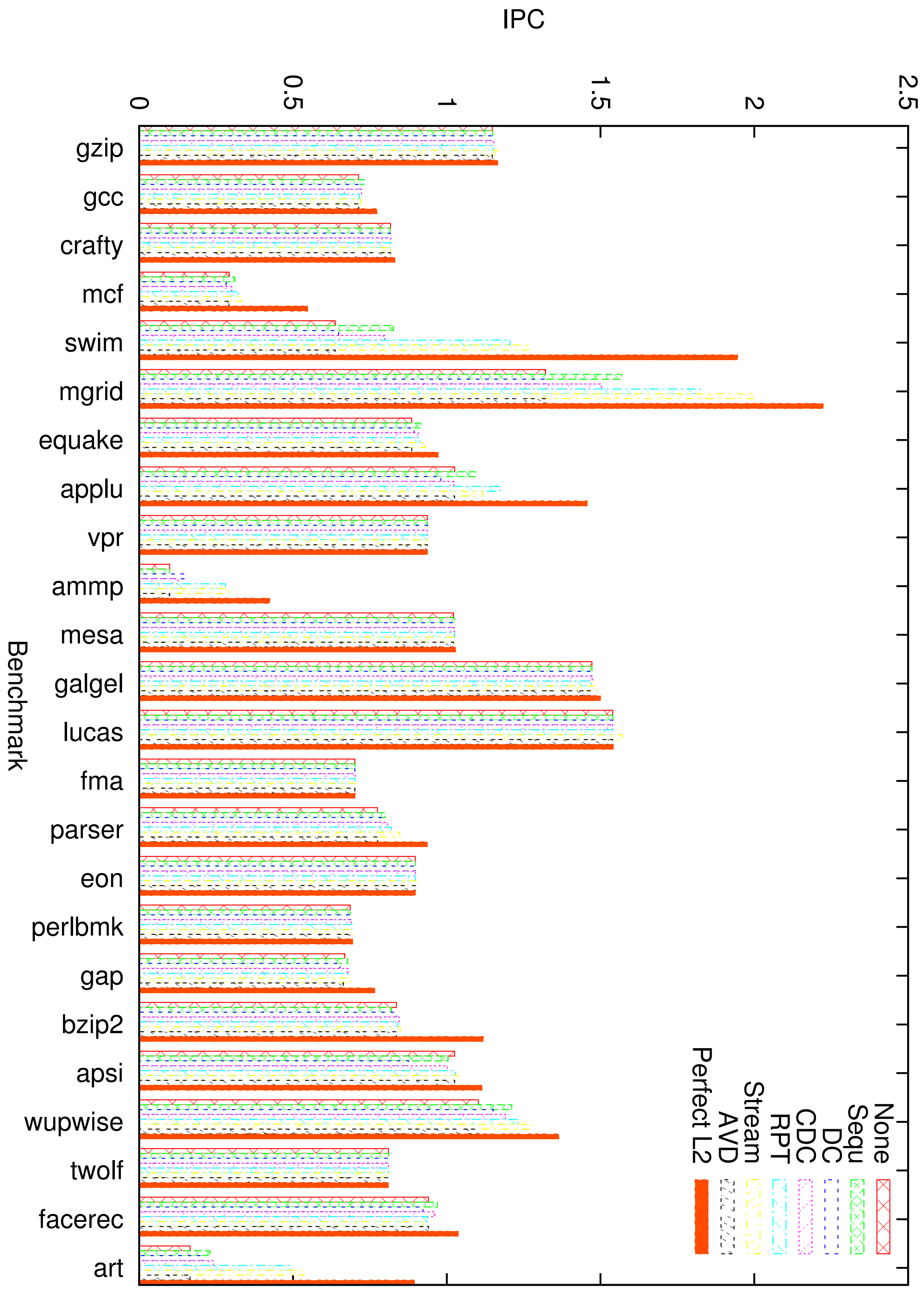

Figure 4.19: Baseline with 2 DRAM channels.

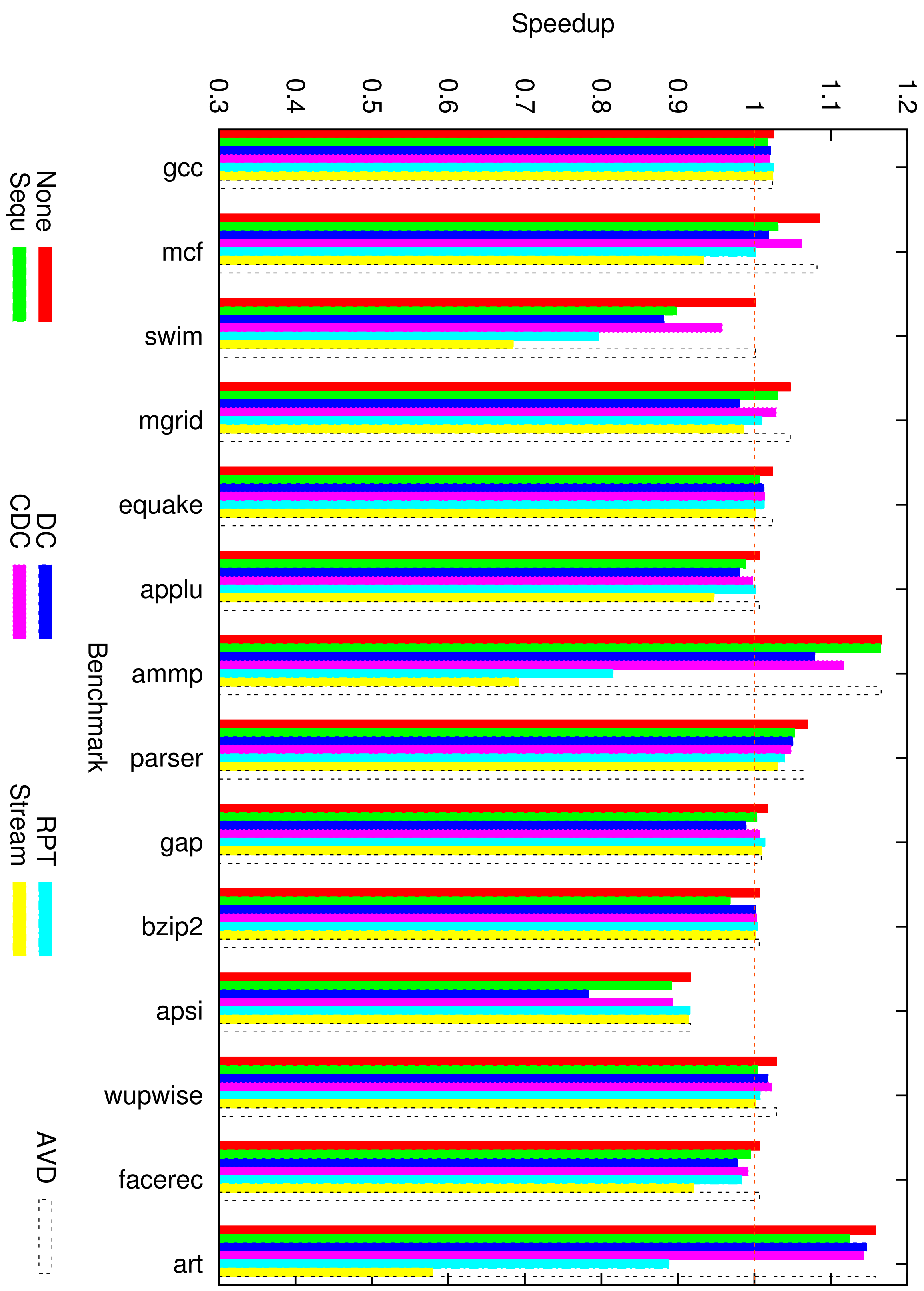

Figure 4.20: Speedup using 2 channels over the unlimited bandwidth model.

Sequential Prefetching

In this section, I will examine how increasing prefetching degree interacts with increasing the amount of bandwidth on sequential prefetching. In the previous section, we observed that sequential prefetching have a monotonous IPC increase as the prefetching degree is increased.

I have conducted five experiments where I have plotted performance (in IPC) as a function of both the number of DRAM channels and the prefetching degree. The benchmarks were selected because they are the most memory intensive according to the preliminary investigations (section 3.5) and in subsequent experiments. The results can be seen in figures 4.21 to 4.25.

It is interesting that the experiments have very different shapes. In figure 4.21 we see how sequential prefetching interacts with Ammp: Increasing the number of channels has the greatest impact on performance, whereas increasing the prefetching degree has little effect.

In figure 4.22 we see a typical scenario: Increasing the prefetching degree increases performance, but the bandwidth required to support prefetching must be present, thus we see a peak with a prefetching degree of 14 and 6 channels.

Mcf behaves quite differently: Increasing the prefetching degree deteriorates performance as it causes contention. This experiment is documented in figure 4.23.

On the Mgrid benchmark (figure 4.24) we observe similar results to the results obtained from Art. We observe that by increasing the prefetching degree we can gain additional performance, but only if there is available bandwidth.

In the final experiment (Swim - figure 4.25), performance degrades as the prefetching degree increases, however, increasing the available bandwidth offsets this trend to some extent.

Generally speaking, no benchmark suffered performance degradation due to increasing the bandwidth available. There are some small exceptions to this, in the form of irregularities. This is most probably due to memory channel conflicts, where the load becomes unevenly distributed across the channels.

When increasing the prefetching degree, three clear patterns emerge. The first group is the group where increasing the prefetching degree also increases performance; This is true for the Art and Mgrid benchmark. The Ammp benchmark gains little from increasing the prefetching degree, while increasing it degrades performance on the Mcf and Swim benchmarks. This result is interesting, as it tells us that one cannot simply increase the prefetching degree for sequential prefetching and assume that it will give increased performance. In addition, we observe that sequential prefetching is heavily dependant on high bandwidth in order to give the maximum benefit.

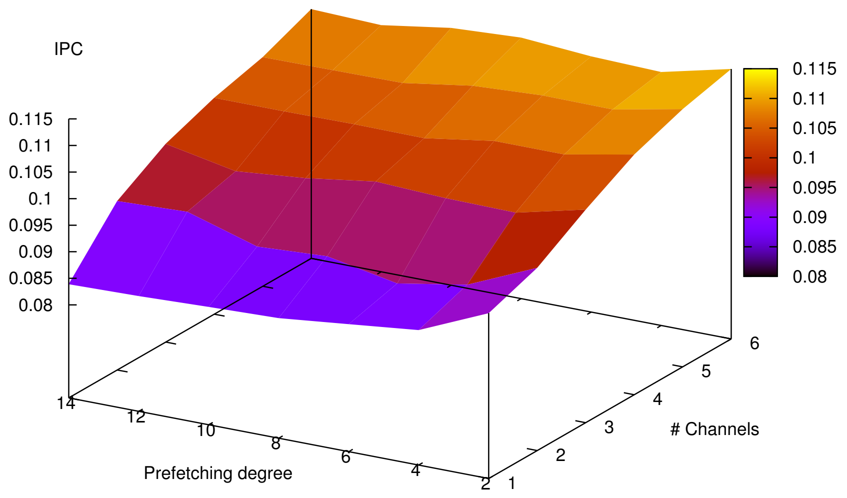

Figure 4.21: Plot of increasing prefetching degree versus available bandwidth for Ammp with sequential prefetching.

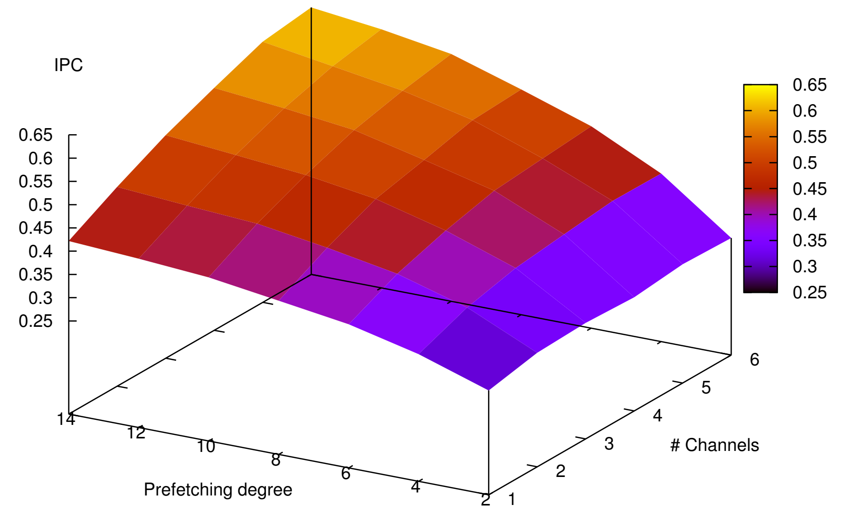

Figure 4.22: Plot of increasing prefetching degree versus available bandwidth for Art with sequential prefetching.

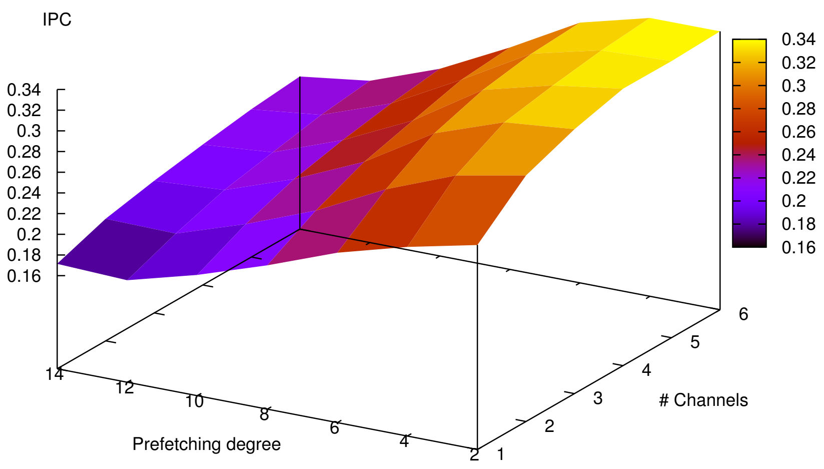

Figure 4.23: Plot of increasing prefetching degree versus available bandwidth for Mcf with sequential prefetching.

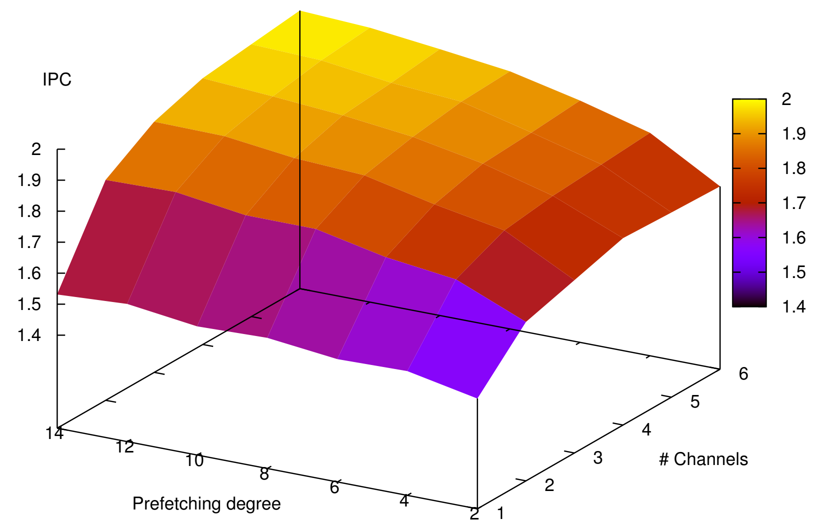

Figure 4.24: Plot of increasing prefetching degree versus available bandwidth for Mgrid with sequential prefetching.

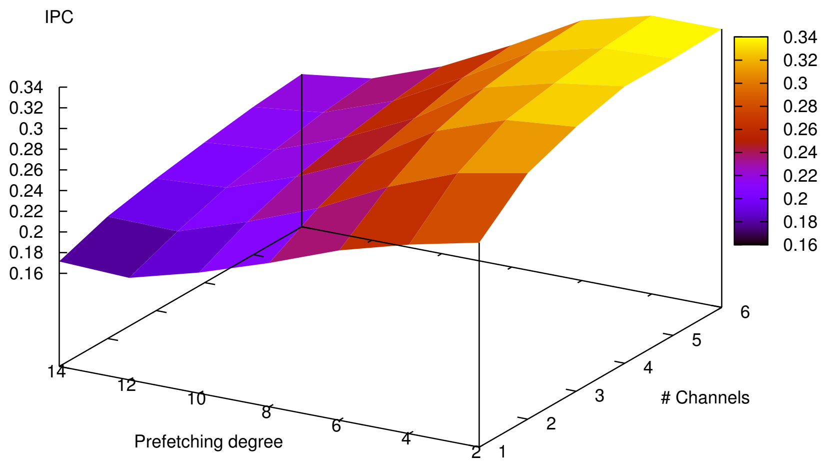

Figure 4.25: Plot of increasing prefetching degree versus available bandwidth for Swim with sequential prefetching.

C/DC Prefetching

In this section I will look at how C/DC prefetching behaves with limited bandwidth in the same manner that I looked at sequential prefetching. I will not look at DC prefetching explicitly, as it behaves in much the same manner as C/DC prefetching, due to the underlying pattern detection scheme discussed in section 2.2.2.

In the Ammp benchmark (figure 4.26) we observe a peculiar shape. This is due to the way that the program interacts with the memory subsystem. In essence, the load across the memory channels become uneven on certain configurations. It is clear that a system with a multiple of three channels outperforms the others. Again, we see that, it is not enough to simply increase the prefetching degree, but there must be available bandwidth to support it.

In the other experiments (figures 4.27 to 4.30) we get almost identical shapes, which is interesting. We do not get the performance degradation observed with sequential prefetching on Swim and Mcf. This result can be interpreted as the C/DC heuristic is much more robust than sequential prefetching. In essence, one can simply assume that increasing the prefetching degree will not deteriorate performance.

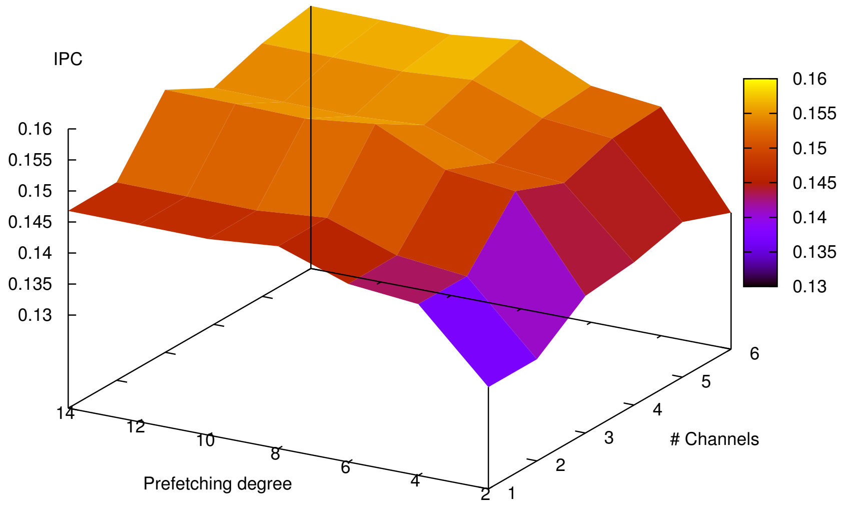

Figure 4.26: Plot of increasing prefetching degree versus available bandwidth for Ammp with C/DC prefetching.

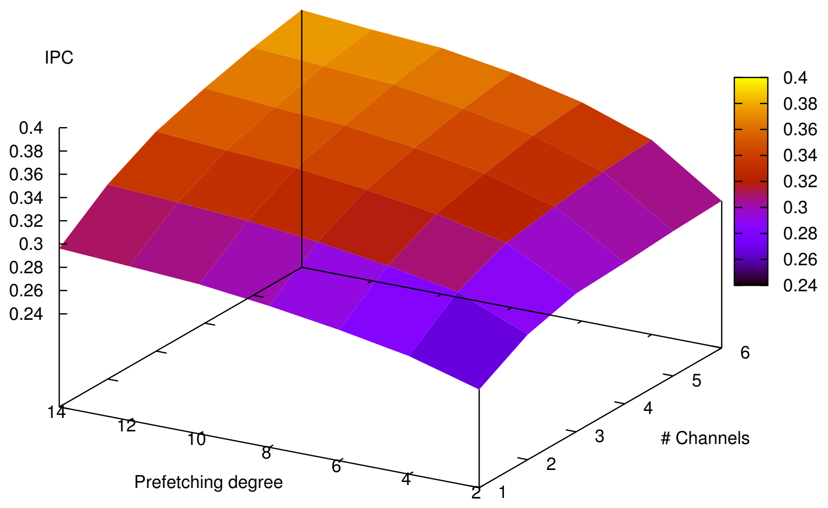

Figure 4.27: Plot of increasing prefetching degree versus available bandwidth for Art with C/DC prefetching.

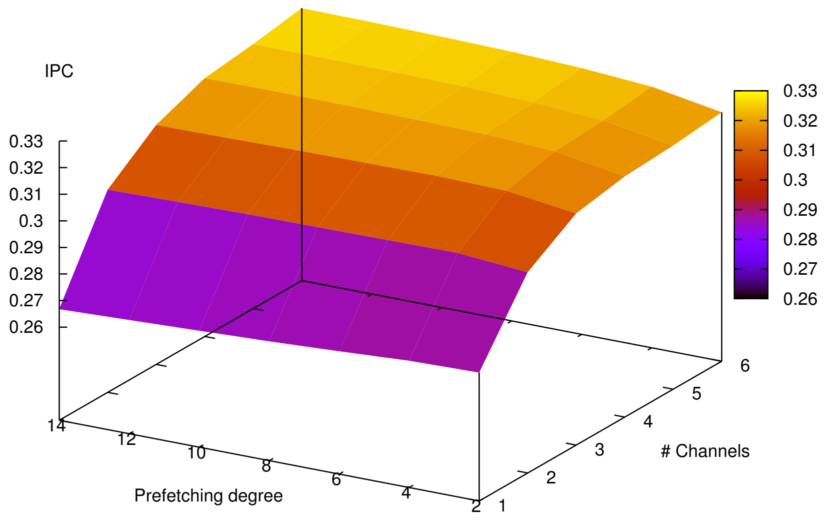

Figure 4.28: Plot of increasing prefetching degree versus available bandwidth for Mcf with C/DC prefetching.

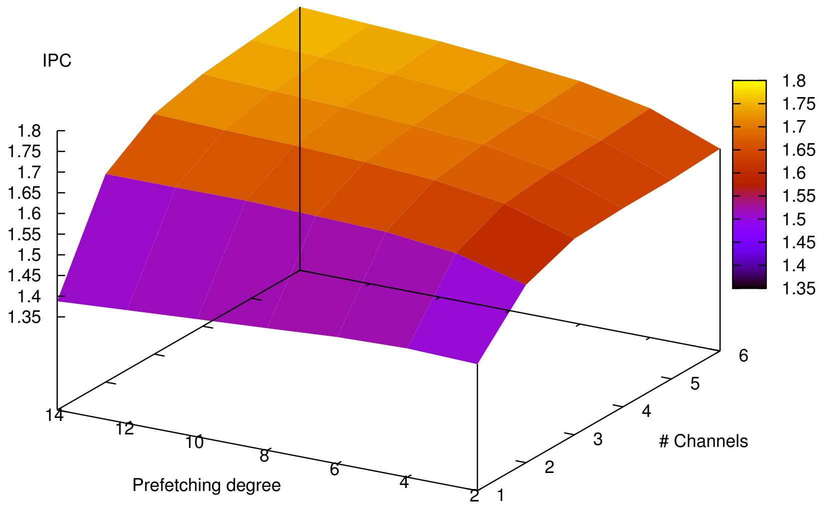

Figure 4.29: Plot of increasing prefetching degree versus available bandwidth for Mgrid with C/DC prefetching.

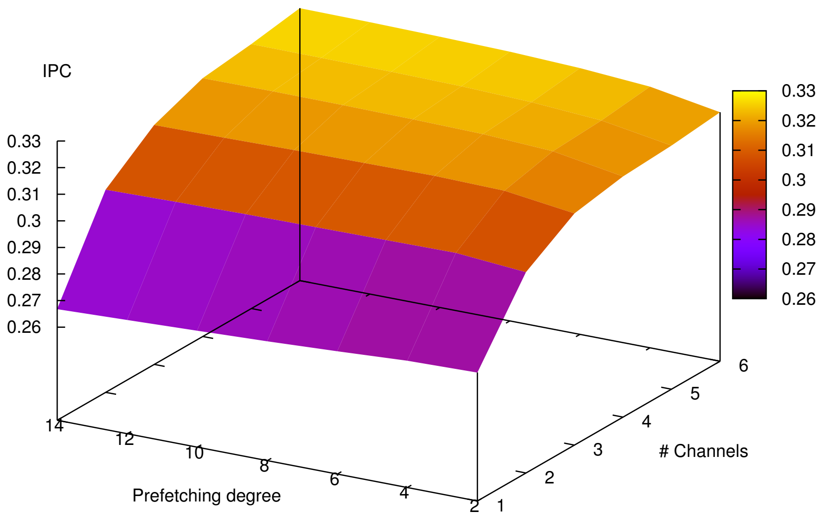

Figure 4.30: Plot of increasing prefetching degree versus available bandwidth for Swim with C/DC prefetching.

RPT Prefetching

In this section I will look at Reference Prediction Table prefetching in order to explore how RPT prefetching interacts with prefetching degree and varying amounts of bandwidth. Because stream prefetching uses the same pattern detection algorithm, I will not do a separate experiment, but rather rely on the results obtained in this section. As a general rule stream prefetching amplifies the results of RPT prefetching, because it also prefetches to the L1 cache. In essence, if the predictor is correct, this will improve performance as L1 misses are turned into L1 hits. However, if the predictor is wrong, the extra pressure on the L1 cache might cause data displacement. As such, the performance of the RPT prefetcher is a very good indicator of the stream prefetchers performance.

I have done the same experiment as in the previous sections, but with a small change in presentation. Because the RPT prefetcher gets effective at very low prefetching degrees, I have selected to show the results with a prefetching degree from 1 to 6, rather than 2 to 14. This was done to improve clarity, and give more detail. The results can be seen in figures 4.31 to 4.35.

In most benchmarks, but especially on Ammp (figure 4.31, we observe the same pattern when increasing the number of memory channels that we observed on C/DC prefetching. Again, this is due to imbalances in the pressure on each channel. We also observe that the number of channels should be a multiple of three. It is interesting to note that this pattern only occurs on RPT and C/DC prefetching, but not in sequential prefetching. This is most probably due to Ammp having a strided access pattern, which RPT and C/DC is able to detect, while sequential prefetching is unable to detect it. This results in a skewed distribution of prefetches across DRAM channels.

On the Art benchmark (figure 4.32) we observe a sharp increase in performance by increasing the number of channels from one to three, but it has less impact beyond three. This is also the case of increasing the prefetching degree. Although there are gains to increasing the prefetching degree beyond 3, these are small compared to the previous ones.

In the next three experiments (Mcf, Mgrid and Swim), we observe that increasing the amount of bandwidth available has the greatest effect on performance, while increasing the prefetching degree has only a minor effect compared to the amount of bandwidth available.

RPT prefetching works in much the same manner as C/DC prefetching. The extra pattern recognition logic is especially significant on the Ammp benchmark. RPT prefetching is also robust, it does not deteriorate performance, whereas sequential prefetching does. In addition, as already observed, it is not very dependent on prefetching degree to function properly, even a low prefetching degree gives a significant performance increase.

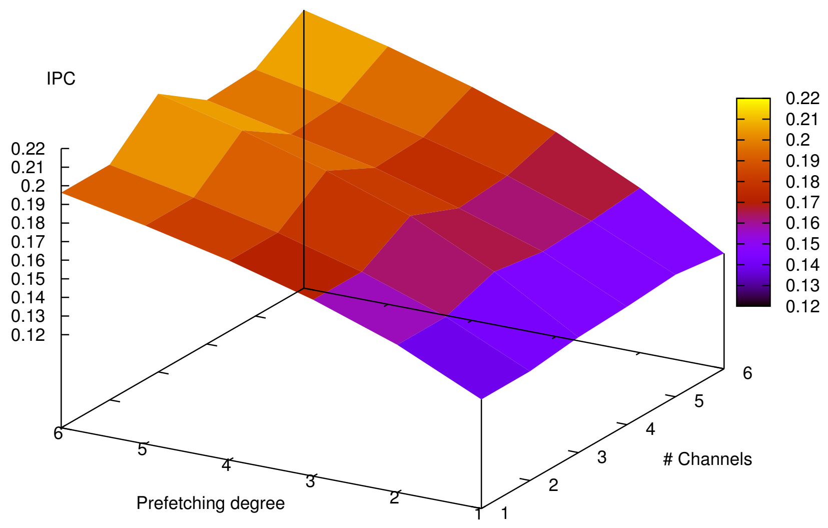

Figure 4.31: Plot of increasing prefetching degree versus available bandwidth for Ammp with RPT prefetching.

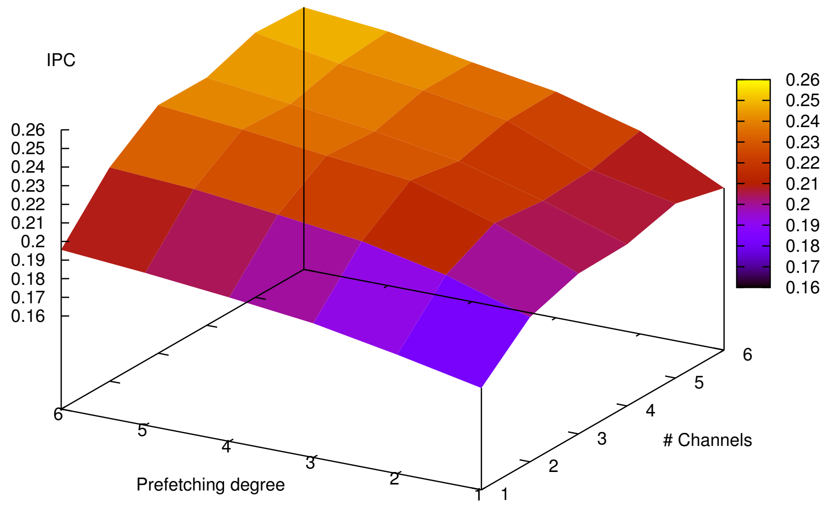

Figure 4.32: Plot of increasing prefetching degree versus available bandwidth for Art with RPT prefetching.

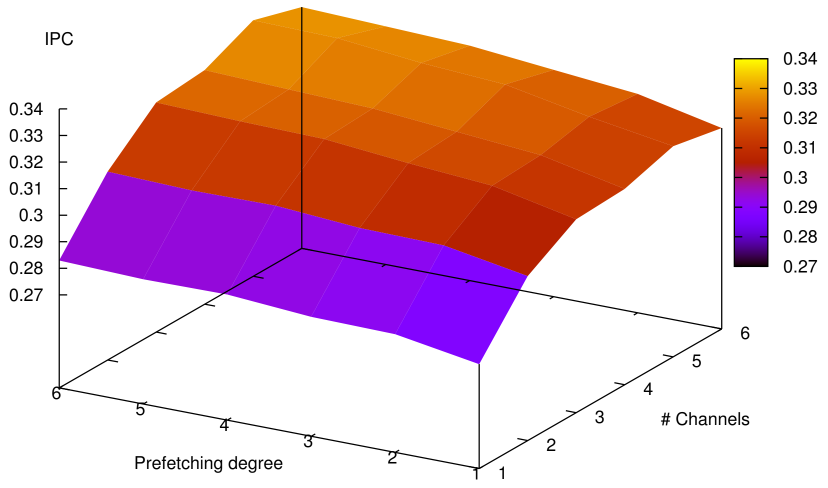

Figure 4.33: Plot of increasing prefetching degree versus available bandwidth for Mcf with RPT prefetching.

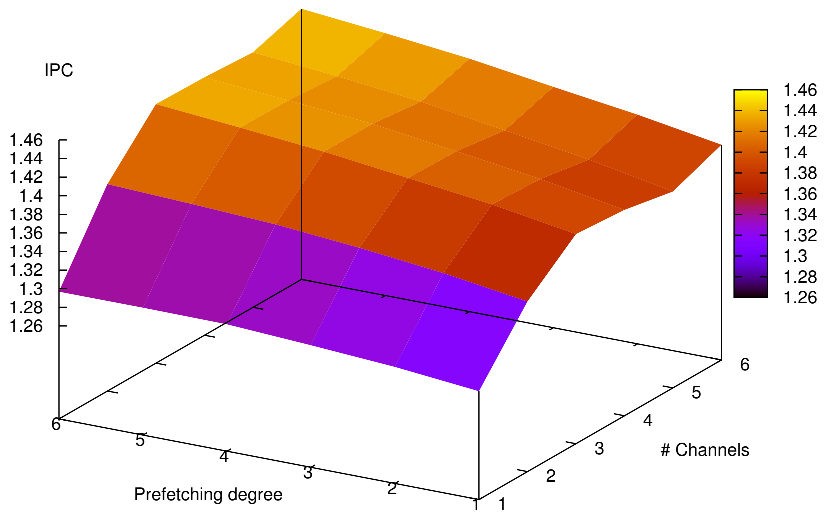

Figure 4.34: Plot of increasing prefetching degree versus available bandwidth for Mgrid with RPT prefetching.

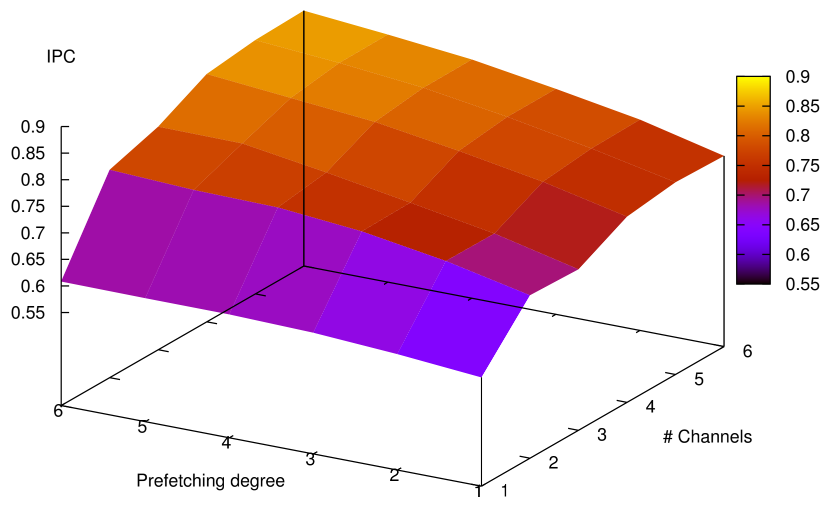

Figure 4.35: Plot of increasing prefetching degree versus available bandwidth for Swim with RPT prefetching.

4.3.3 Bandwidth-Aware Prefetching

The goal of bandwidth aware prefetching is to increase performance through a decrease in memory bandwidth usage. However, because most SPEC2000 benchmarks are not memory intensive, this section will both deal with performance and robustness (how it handles varying loads).

All experiments in this section has been conducted with only one DRAM channel. This was done to increase pressure on the memory subsystem. However, this is not a significant limitation compared to contemporary systems.

In figure 4.36, I have compared the IPC of bandwidth-aware prefetching to the bandwidth-oblivious versions of the same prefetching heuristics. In this experiment the cutoff value is set to 40. This is a very restrictive value, that in practice drops prefetches if any memory access is in progress. In most cases, we observe a performance decrease, as fewer prefetches are issued. This is natural, considering that for most cases, bandwidth is not an issue in a uniprocessor. It is also interesting to note that for sequential prefetching, significant decreases can be observed when the original scheme has a very high accuracy (Mgrid, Wupwise, Art). However, the difference is relatively small in most cases.

There are five noteworthy exceptions to this observation. These are the Ammp and Art benchmarks for the RPT and Stream prefetchers. We see that performance decreases by about 45% in these cases. This is natural, as these benchmarks are very latency-sensitive, while the prefetchers are highly accurate (see table 4.2).

It is interesting to note a performance increase in Applu by using the Stream prefetcher. Applu is one of the few cases where Stream prefetching has a low accuracy, and will therefore get a higher performance if the correct prefetches are dropped.

Figure 4.37 shows the positive aspect of bandwidth-aware prefetching. In this figure the memory bandwidth usage of bandwidth-aware prefetching is compared to the case of no bandwidth-aware prefetching.

In general, sequential and DC prefetching gains the most from this technique, as these heuristics are generally less accurate than the others. We observe that the greatest reductions in bandwidth usage when using sequential prefetching occurs on benchmarks where it has low accuracy. This is also the case for RPT and Stream prefetching in cases where they have a low accuracy (Galgel and Applu).

From these initial experiments, it is clear that bandwidth-aware prefetching works best when accuracy is low. If accuracy is relatively low, then it can provide about the same performance, while reducing the bandwidth requirements significantly.

In figures 4.3.3 and 4.39 I have conducted the exact same experiment, but with the threshold increased to 400. In the first figure we observe that for most, benchmarks and prefetchers, there is still performance reductions, however these are not as large as in the previous case. In addition, we keep most of the reductions in bandwidth usage by increasing the threshold.

The increased threshold also reduces the benefits by a small amount, but for Crafty and Perlbmk, the reduction is not significant compared to the results previously obtained.

In the last experiment, the threshold has been increased to 800. The results can be viewed in figures 4.40 and 4.41. Again, we see the effect of bandwidth-aware prefetching diminishes as the threshold is increased.

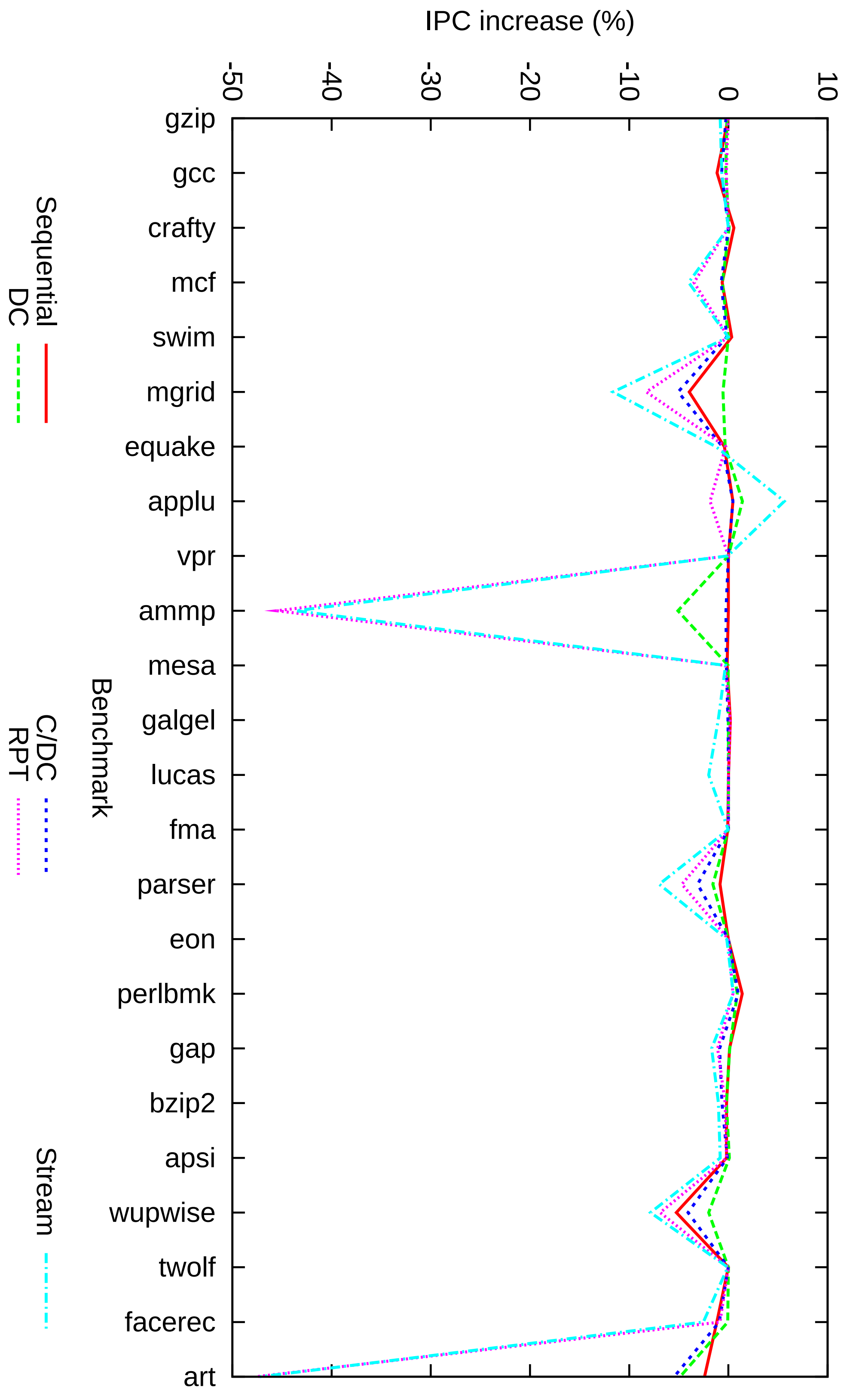

Figure 4.36: Speedup using bandwidth-aware prefetching. Cutoff value is set at 40.

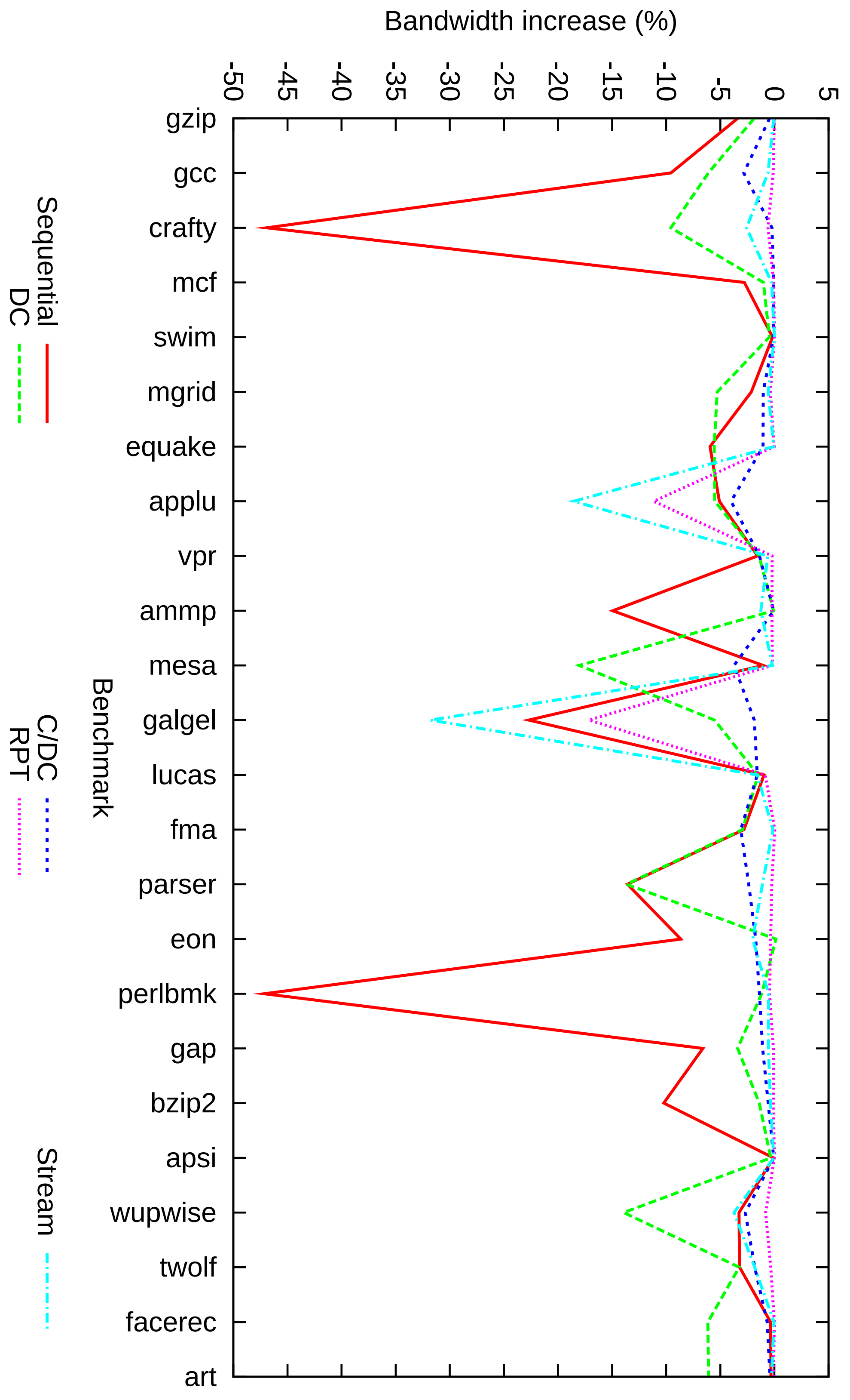

Figure 4.37: Reductions in bandwidth usage by using bandwidth-aware prefetching. Threshold value is set at 40.

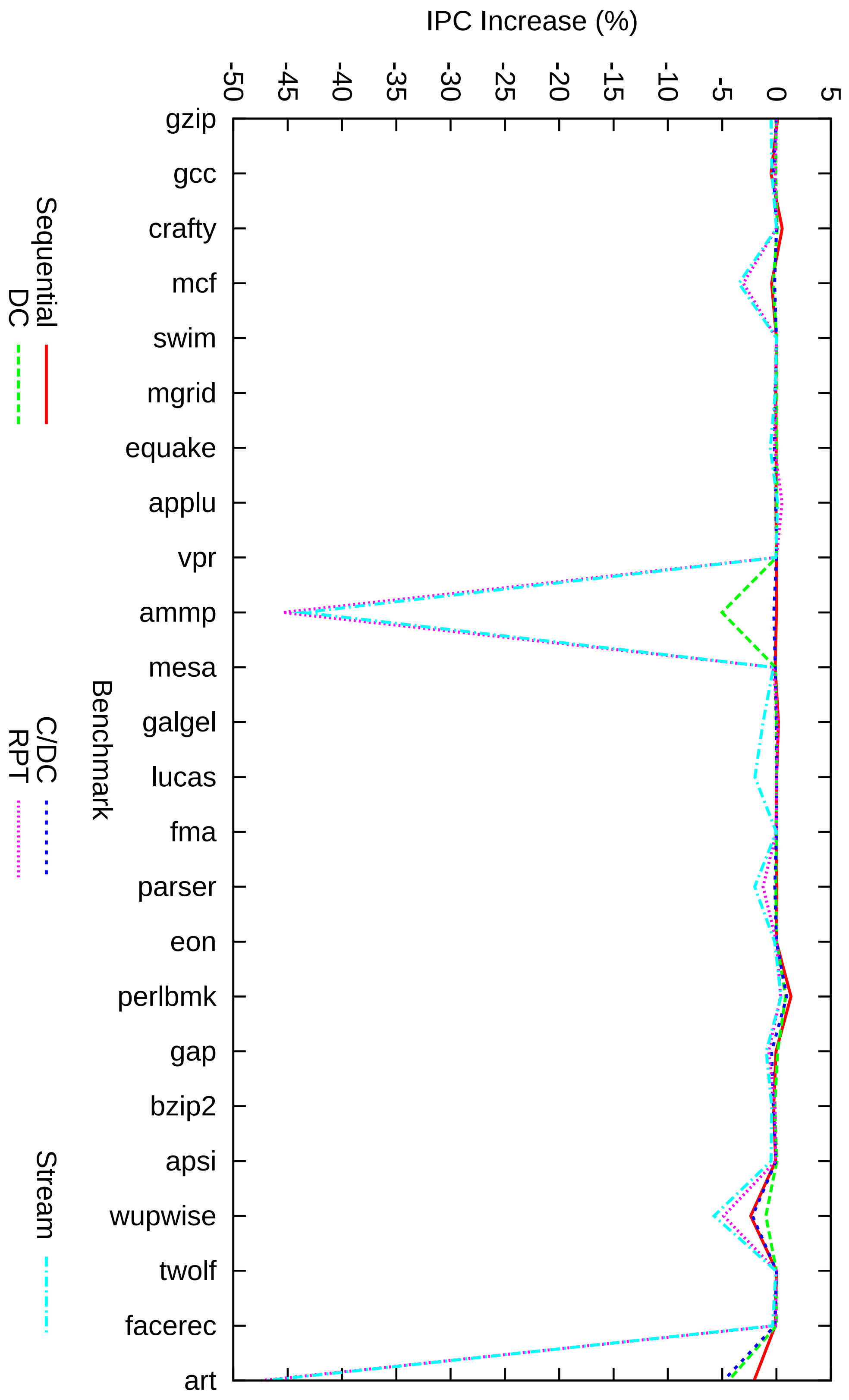

Figure 4.38: Speedup using bandwidth-aware prefetching. Threshold is set at 400.

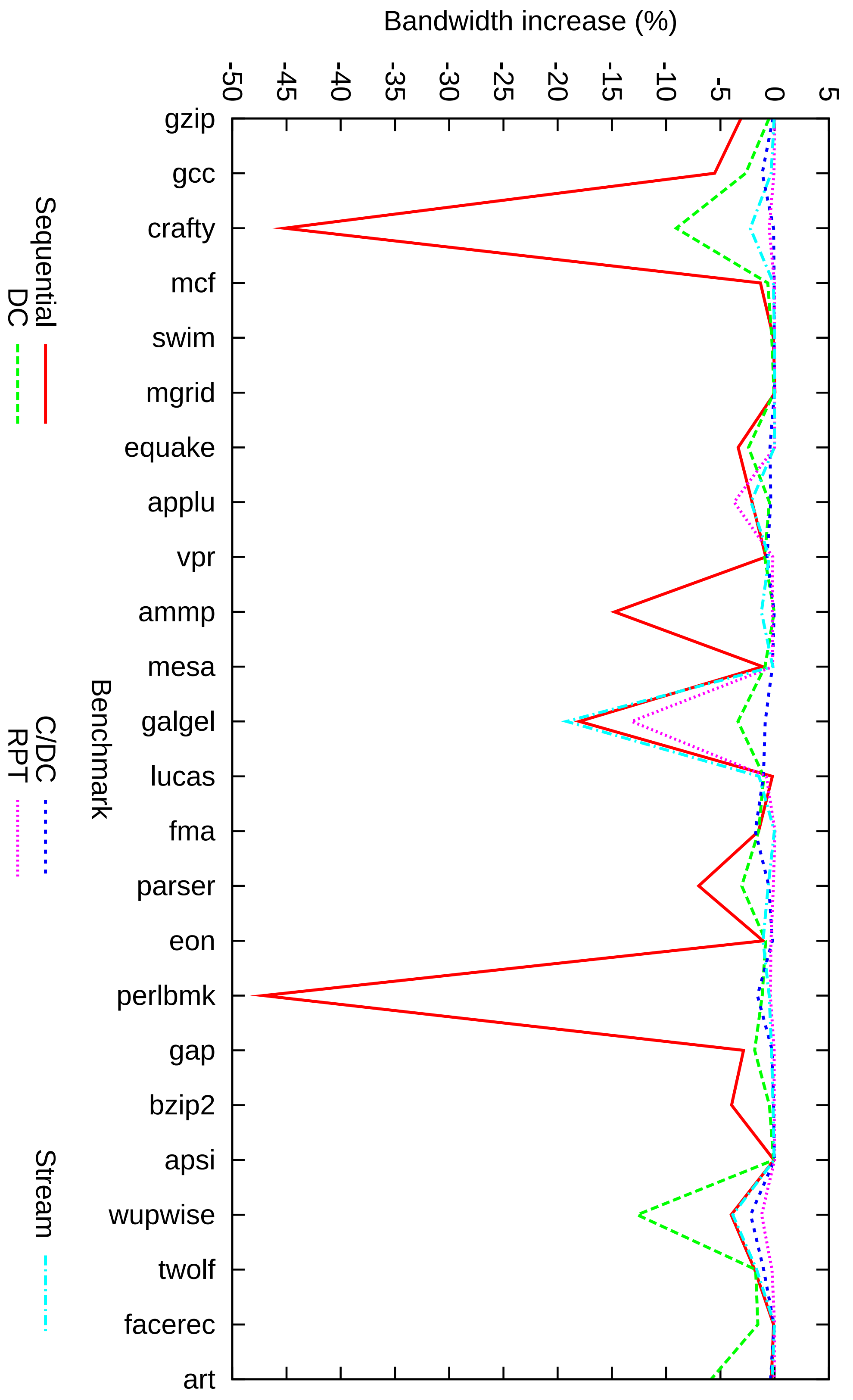

Figure 4.39: Reductions in bandwidth usage. Threshold is set at 400.

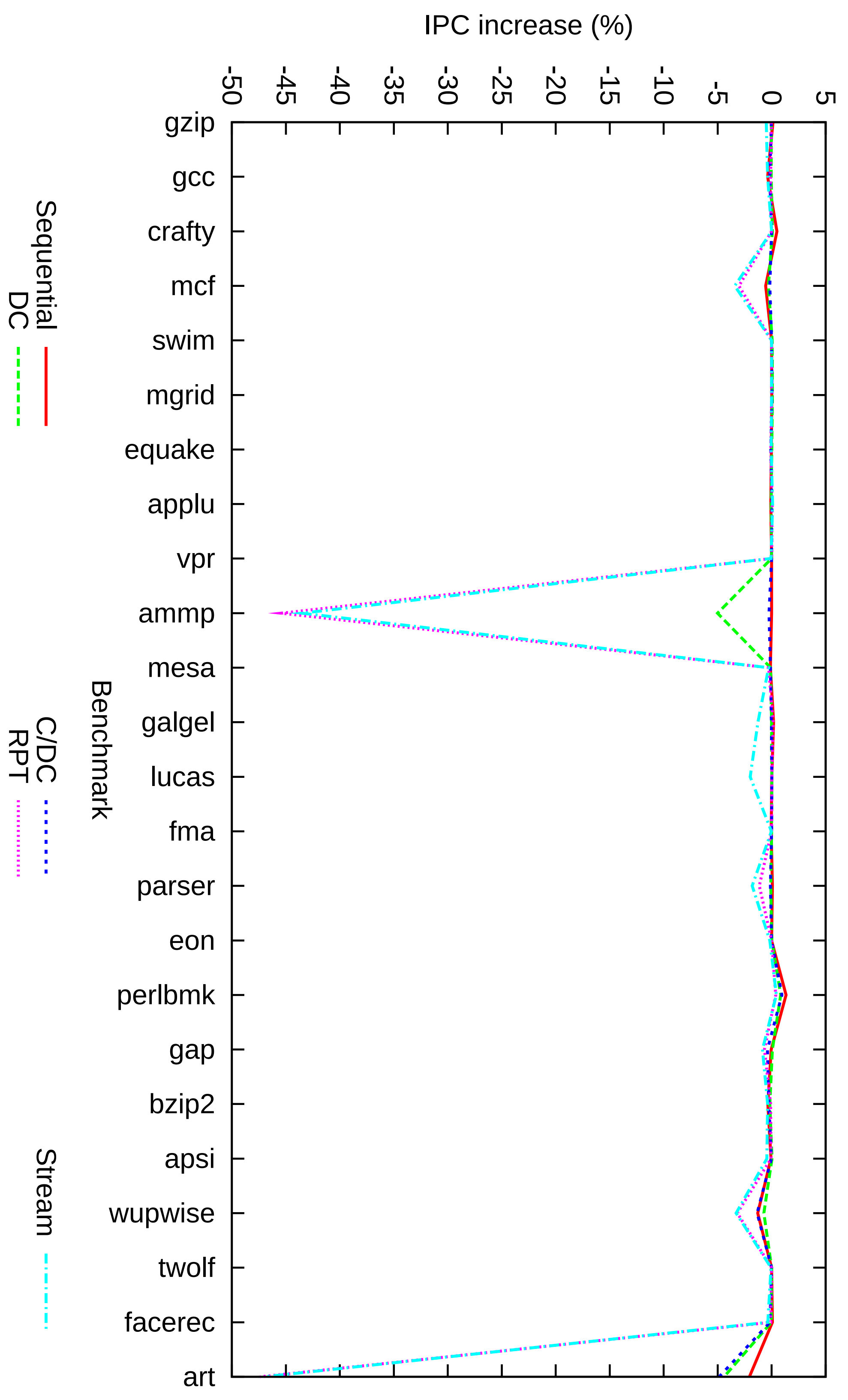

Figure 4.40: Speedup using bandwidth-aware prefetching. Threshold is set at 800.

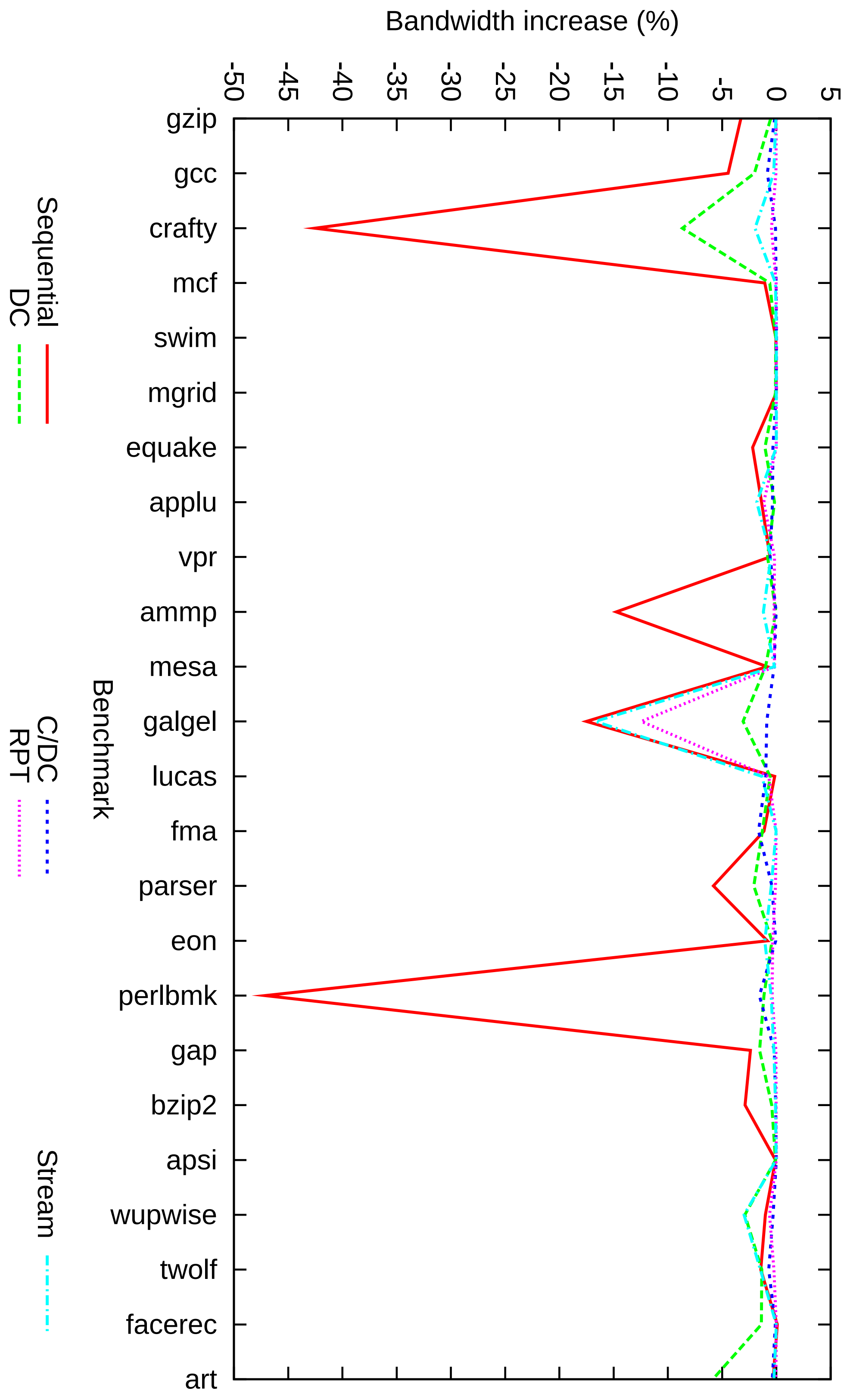

Figure 4.41: Reductions in bandwidth usage. Threshold is set at 800.

4.4 CMP

4.4.1 Plan for the Experiments

In this section I will look at how prefetching work in CMPs. The following experiments will be conducted on a dual core CMP system. By using a 2-way system instead of a system with a larger amount of cores, it becomes easier to analyze the results.

Each core will be configured as in the uniprocessor experiments. However, the L2 cache and memory controller will be shared among the cores, but will retain the same parameters as in the previous experiments, unless otherwise noted.

I will look at three distinct properties of this CMP:

1. How CMP affects performance for single applications.
2. How prefetching affects performance.
3. How Bandwidth-aware prefetching affects performance.

Lgred benchmarks can be very short (50 million instructions) or comparatively long (several billion instructions). This poses a problem when running them simultaneously, because running time would be limited to the shortest running program. If the running time of the experiment was not limited to the running time of the shortest experiment, then it would degenerate into a uniprocessor experiment[1].

A solution to this problem is to use the full SPEC2000 benchmarks, but running them to end would be prohibitive, therefore it is only possible to run a small interval of each. To avoid benchmarking startup code, I will fast-forward (no pipeline or cache simulation) through the first 500 million to 1500 million instructions (randomly selected). After that, I will run the cycle-accurate simulation for 200 million clock cycles.

As mentioned in section 3.3.2, the CMP simulator is not as fast as the uniprocessor simulator. This is both due to locking and due to the fact that 2 cores are being simulated at the same time. This limits the number of benchmarks that can be run. By using the full SPEC2000 benchmark suite there are 676 possible benchmark pairs (assuming the cores are not symmetrical). Fortunately, most of the benchmarks in the SPEC2000 suite are not interesting for our purposes. I have selected 10 benchmarks from two groups. In the first group are the compute bound applications, they were selected because they have relatively few L2 misses per instruction.

- Gzip
- Gcc
- Crafty
- Eon
- Twolf

The second group is the memory bound applications:

[1]It would be possible to restart benchmarks to compensate for this.

- Swim
- Mgrid
- Ammp
- Wupwise
- Art

By making such a distinction we can now look at how benchmarks from the two groups interact.

4.4.2 CMP Performance

In this section, I will look at how benchmarks perform in a CMP with another program present. First, I establish a baseline experiment, which I will use as a base metric, for the rest of the experiments.

In table 4.4a I have run all the experiments combined with all the others and summarized them. In this table the performance in terms of IPC of the benchmark in the first column is shown. The benchmark is combined with the benchmark in the top row. By dividing the benchmarks in such a manner, we see that there are four groups of interest, these are:

1. Compute bound combined with compute bound.
2. Compute bound combined with memory bound.
3. Memory bound combined with compute bound.
4. Memory bound combined with memory bound.

Naturally, combining any application with a memory bound application leads to decreased performance. This is natural, as there are more evictions from the cache as well as bandwidth contention, and is consistent with theory. In addition, compute bound applications shows less performance degradation when combined with memory bound applications than memory bound applications.

In table 4.4b the same experiment was run, but with 2 DRAM channels. This experiment was conducted to establish how much extra bandwidth increases performance for each benchmark. We observe little change in group 1, however, the most bandwidth intensive applications, such as Ammp show a significant improvement. This is natural, and is compatible with the results obtained in the uniprocessor section. It is especially worthwhile to note the changes that occur in group 4. Because, both benchmarks are memory bound, each of them gains from the added bandwidth, but the measured benchmark only gains its fraction. By comparing this to group 3 we see a significant decrease in performance. However, there is still a large performance increase compared to the system with only 1 DRAM channel.

In table 4.5, I examine the effects of increasing the cache size. In subtable a I have doubled the cache size to 2MB by doubling the number of sets. In effect, this will double the area requirements of the cache. However, we do not gain significant speedup by doing

(a) 1 DRAM channel

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art	h-mean
gzip	1.4692	1.4454	1.4276	1.5186	1.5163	1.2669	1.1621	1.2543	1.4620	1.0872	1.3438
gcc	0.6768	0.6672	0.6398	0.7023	0.7195	0.6181	0.5650	0.6343	0.6834	0.5136	0.6358
crafty	0.6905	0.6975	0.5953	0.7280	0.7384	0.6529	0.5988	0.6638	0.7232	0.5554	0.6587
eon	0.8914	0.8877	0.8845	0.8754	0.8949	0.8846	0.8494	0.8634	0.8916	0.8515	0.8771
twolf	0.9341	0.9291	0.9282	0.9390	0.9379	0.9169	0.8671	0.8937	0.9300	0.8511	0.9117
swim	1.3454	1.3025	1.5104	1.5230	1.4012	1.2492	1.5628	1.4623	1.2748	1.3371	1.3889
mgrid	0.7832	0.7884	0.7988	0.9050	0.9158	0.7443	0.6275	0.6125	0.8402	0.5800	0.7419
ammp	0.0636	0.0688	0.5926	0.0782	0.1040	0.0563	0.0495	0.7066	0.0720	0.0342	0.0731
wupwise	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846
art	0.0824	0.0926	0.0913	0.1053	0.0997	0.0693	0.0618	0.0613	0.0933	0.0542	0.0772

(b) 2 DRAM channels

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art	h-mean
gzip	1.4922	1.4752	1.4550	1.5265	1.5258	1.3712	1.2994	1.3078	1.4838	1.1943	1.4044
gcc	0.6983	0.6857	0.6638	0.7101	0.7255	0.6522	0.6267	0.6795	0.6996	0.5622	0.6670
crafty	0.7142	0.7184	0.6285	0.7369	0.7465	0.6850	0.6553	0.7001	0.7337	0.6095	0.6898
eon	0.8923	0.8893	0.8855	0.8758	0.8952	0.8857	0.8610	0.8694	0.8923	0.8555	0.8800
twolf	0.9364	0.9318	0.9302	0.9400	0.9392	0.9218	0.8918	0.9045	0.9313	0.8783	0.9200
swim	1.3748	1.3484	1.5549	1.5598	1.4363	1.2704	1.5882	1.4901	1.3159	1.3691	1.4229
mgrid	0.9879	0.9842	0.9870	1.0301	1.0335	0.9578	0.8320	0.8433	0.9978	0.7853	0.9357
ammp	0.0840	0.0873	0.7027	0.0913	0.1212	0.0802	0.0808	0.7192	0.0909	0.0608	0.1024
wupwise	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846	1.8846
art	0.1266	0.1375	0.1360	0.1494	0.1470	0.1140	0.1064	0.0933	0.1436	0.0883	0.1202

Table 4.4: IPC of the benchmark in the left column in a dual core CMP combined with the benchmarks in the first row.

so. I have compared the performance with the baseline experiment with 1 DRAM channel. The largest increase is for the Ammp benchmark combined with the Twolf benchmark with a 79% increase in performance. The rest of the benchmarks show less than a 24% increase in performance.

In subtable b I have experimented with a 4MB cache, by both doubling the number of sets, and by doubling the associativity of the cache. In this experiment we see a much more significant increase in performance. Again, the compute bound applications in group 1 gain little from the larger cache. In group 2 however, there is a large increase in performance when the benchmarks are combined with some of the more memory intensive benchmarks. The largest increases in performance occurs for groups 3 and 4. Here we observe increases up to 776% for Ammp combined with Gzip. This increase is especially significant for Ammp and Art, which are the most bandwidth intensive benchmarks in the entire SPEC2000 suite.

(a) Double the number of sets (2 MB cache)

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	+4.4 %	+3.6 %	+4.0 %	+0.8 %	+0.7 %	+7.8 %	+12.2 %	+4.6 %	+1.0 %	+25.9 %
gcc	+5.2 %	+5.9 %	+6.8 %	+3.5 %	+2.6 %	+10.5 %	+10.7 %	+3.4 %	+4.4 %	+11.8 %
crafty	+5.7 %	+5.5 %	+16.3 %	+4.6 %	+3.1 %	+4.5 %	+7.5 %	+3.8 %	+2.7 %	+10.3 %
eon	+0.2 %	+0.4 %	+0.6 %	+2.0 %	+0.0 %	+0.4 %	+1.4 %	+0.9 %	+0.3 %	+2.3 %
twolf	+0.4 %	+0.8 %	+0.8 %	+0.1 %	+0.1 %	+0.6 %	+2.0 %	+0.7 %	+0.4 %	+3.9 %
swim	+0.4 %	+0.1 %	+0.1 %	+0.0 %	+0.1 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
mgrid	+20.7 %	+18.3 %	+14.5 %	+11.7 %	+21.8 %	+12.8 %	+13.8 %	+12.5 %	+14.6 %	+1.3 %
ammp	+2.3 %	+2.3 %	+1.0 %	+0.5 %	+79.0 %	+1.6 %	+10.2 %	+6.1 %	+1.9 %	+0.9 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+23.5 %	+13.1 %	+13.4 %	+11.5 %	+17.4 %	+12.0 %	+12.9 %	+12.8 %	+7.8 %	+5.1 %

(b) Double the number of sets and associativity (4MB cache)

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	+5.1 %	+4.4 %	+4.7 %	+0.9 %	+0.9 %	+9.7 %	+14.7 %	+11.5 %	+1.6 %	+40.1 %
gcc	+6.6 %	+8.9 %	+9.3 %	+4.3 %	+3.3 %	+13.6 %	+15.9 %	+6.5 %	+6.9 %	+24.6 %
crafty	+8.2 %	+8.2 %	+21.6 %	+5.9 %	+4.7 %	+7.3 %	+11.3 %	+6.3 %	+4.4 %	+18.3 %
eon	+0.2 %	+0.5 %	+0.6 %	+2.1 %	+0.0 %	+0.6 %	+2.3 %	+1.7 %	+0.4 %	+3.2 %
twolf	+0.6 %	+1.0 %	+0.9 %	+0.1 %	+0.2 %	+1.0 %	+3.0 %	+1.9 %	+0.6 %	+6.0 %
swim	+0.4 %	+0.2 %	+0.2 %	+0.1 %	+0.1 %	+0.0 %	+0.1 %	+0.0 %	+0.0 %	+0.0 %
mgrid	+31.4 %	+27.6 %	+22.7 %	+15.8 %	+26.2 %	+20.6 %	+31.4 %	+39.6 %	+17.5 %	+33.0 %
ammp	+776.8 %	+601.3 %	+18.1 %	+606.2 %	+248.8 %	+307.3 %	+375.9 %	+7.7 %	+523.9 %	+133.3 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+135.1 %	+110.1 %	+120.1 %	+177.8 %	+229.3 %	+76.4 %	+17.5 %	+156.2 %	+114.4 %	+9.8 %

Table 4.5: Speedup in IPC compared to a L2 cache of 1MB.

4.4.3 Prefetching in CMP

In this section I will look at how prefetching works in a CMP.

I will only look at sequential, C/DC and RPT prefetching. I have omitted DC prefetching because it is basically C/DC prefetching with very large CZones. Stream prefetching was omitted because the underlying memory reference detection mechanism is the same as RPT prefetching. AVD prefetching was omitted because it showed little effect on uniprocessor benchmarks.

In table 4.6, I have compared the performance of sequential and C/DC prefetching to the baseline experiment. By using sequential prefetching we see a much more diffuse picture, as some benchmarks experience a slowdown, while other combinations show a speedup. This is most likely due to the inaccuracy of sequential prefetching causing bandwidth contention. This is especially clear in group 2, where the memory intensive applications prefetches too much data and degrades performance for the compute intensive application. This occurs both because the compute bound application evicts useful data from the cache, but also because it causes bandwidth contention.

In subtable b, the performance of C/DC prefetching in a CMP is documented. Here we observe the same performance degradation in group 2, but it is dampened compared to sequential prefetching. In addition, in groups 3 and 4 we no longer observe significant performance decreases. This is again due to C/DCs increased accuracy.

In table 4.7, the performance of the RPT prefetching is shown. RPT prefetching has a very high accuracy, and therefore it is strange to note that performance is not increased as much as one would think. For the Art benchmark, performance is actually decreased by a significant amount in most cases. This might be caused by the relatively short simulation time not being sufficient to properly initialize the reference prediction tables.

(a) Sequential prefetching

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	+1.5 %	-0.5 %	-1.4 %	+1.0 %	+0.8 %	-6.4 %	-8.4 %	-7.2 %	-1.3 %	+7.6 %
gcc	-2.3 %	-1.6 %	-3.1 %	-0.3 %	-0.2 %	-3.6 %	-6.0 %	-7.9 %	-0.3 %	+1.0 %
crafty	-4.3 %	-3.7 %	-7.9 %	-2.1 %	-1.7 %	-7.7 %	-9.6 %	-9.4 %	-1.9 %	-6.6 %
eon	-0.1 %	-0.1 %	-0.4 %	-0.1 %	-0.0 %	-0.6 %	-1.8 %	-1.9 %	-0.1 %	-0.2 %
twolf	+0.0 %	-0.1 %	-0.3 %	+0.0 %	+0.0 %	-0.7 %	-2.8 %	-2.9 %	-0.0 %	-0.7 %
swim	+0.1 %	+0.5 %	+0.4 %	+1.1 %	+0.7 %	+0.4 %	-0.1 %	+0.5 %	-0.2 %	+1.6 %
mgrid	+15.1 %	+18.9 %	+15.0 %	+20.3 %	+26.4 %	+16.6 %	+15.5 %	+3.9 %	+22.8 %	+45.4 %
ammp	-23.5 %	-20.6 %	-15.7 %	-14.5 %	-14.3 %	-23.9 %	-32.1 %	-1.8 %	-19.4 %	-21.4 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	-1.1 %	+0.4 %	-0.9 %	+4.2 %	+5.5 %	-1.3 %	-2.7 %	-11.0 %	+4.8 %	+6.3 %

(b) C/DC prefetching

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	+1.1 %	+0.9 %	+1.2 %	+1.0 %	+1.0 %	-4.2 %	-6.2 %	-0.1 %	-0.1 %	-1.0 %
gcc	-0.1 %	-0.1 %	+0.3 %	+0.0 %	+0.0 %	-3.5 %	-6.0 %	-0.4 %	+0.0 %	-7.0 %
crafty	-0.2 %	-0.3 %	-0.3 %	-0.1 %	-0.0 %	-4.0 %	-5.5 %	-0.9 %	-0.4 %	-5.9 %
eon	-0.0 %	-0.0 %	-0.0 %	-0.3 %	+0.0 %	-0.4 %	-1.3 %	-0.5 %	-0.0 %	-1.0 %
twolf	+0.0 %	+0.0 %	+0.1 %	+0.0 %	+0.0 %	-0.7 %	-1.8 %	-0.4 %	+0.0 %	-1.2 %
swim	+0.8 %	+1.4 %	+1.0 %	+1.2 %	+1.1 %	+0.6 %	+0.2 %	+0.6 %	+0.9 %	+1.4 %
mgrid	+19.1 %	+21.4 %	+20.0 %	+17.2 %	+23.3 %	+17.9 %	+16.1 %	+8.1 %	+18.8 %	+26.1 %
ammp	+9.7 %	+10.7 %	+3.1 %	+11.3 %	+10.5 %	+6.5 %	+0.5 %	+0.1 %	+9.4 %	+7.1 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.8 %	+0.4 %	+0.8 %	+0.6 %	+0.0 %	-1.4 %	-0.8 %	-1.2 %	+2.0 %	+4.7 %

Table 4.6: Prefetching in CMP, speedup compared to a CMP with no prefetching.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	+0.4 %	+0.2 %	+0.3 %	+0.5 %	+0.5 %	-0.7 %	-0.6 %	-1.7 %	+0.1 %	-3.1 %
gcc	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-1.4 %	-0.9 %	-0.8 %	+0.0 %	-6.0 %
crafty	-0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %	-0.5 %	-1.1 %	-1.2 %	-0.0 %	-5.0 %
eon	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.2 %	-0.9 %	+0.0 %	+4.9 %
twolf	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.1 %	-0.4 %	-0.8 %	+0.0 %	-1.7 %
swim	+0.3 %	+0.3 %	+0.2 %	+0.3 %	+0.2 %	+0.2 %	+0.2 %	-0.2 %	+0.4 %	-0.1 %
mgrid	+6.7 %	+8.4 %	+5.8 %	+7.5 %	+8.2 %	+6.7 %	+11.1 %	-5.2 %	+5.4 %	+2.2 %
ammp	+12.8 %	+14.1 %	+3.6 %	+15.4 %	+13.3 %	+13.9 %	+12.5 %	-0.2 %	+13.4 %	+10.1 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	-3.5 %	-9.4 %	-9.9 %	-14.5 %	-11.0 %	-2.8 %	-0.6 %	-4.6 %	-8.4 %	-7.3 %

Table 4.7: Performance of RPT benchmarking in a CMP compared to the case where no prefetching is performed.

4.4.4 Bandwidth-Aware Prefetching in CMP

In this section, I will look at how bandwidth-aware prefetching works in a CMP. Bandwidth-aware prefetching was designed with CMP in mind.

In table 4.8, I have compared the performance of bandwidth aware sequential prefetching with the bandwidth-oblivious version. In this experiment the threshold was set to 40, which again is a very aggressive threshold, but is used to better highlight the properties of bandwidth-aware prefetching. For the Crafty benchmark, we see some significant performance gains, especially when combining two Crafty benchmarks (+ 4%). For Mgrid, there is a significant performance decrease when combining it whit compute-bound applications (up to a 5.3% decrease). For most applications, there is a small speed increase.

However, there are some very large decreases in bandwidth usage, especially for the compute-bound applications, in addition to Swim, Mgrid and Wupwise. The most bandwidth-intensive benchmarks, Art and Ammp, did not gain such large improvements, mostly due to the relatively high accuracy of sequential prefetching on these benchmarks. The largest savings in bandwidth usage was 47.8% which occurred when combining Crafty with Eon.

The next experiment (table 4.9) were conducted with C/DC prefetching. C/DC prefetching is much more accurate than sequential prefetching in general. As accuracy increases, the performance benefits of bandwidth-aware prefetching decreases. We observe the same pattern with C/DC prefetching as we did with sequential prefetching, however, Crafty does not experience a speedup by using bandwidth-aware prefetching. Mgrid does, on the other hand, experience a larger performance decrease than in the sequential case (up to 6.8%).

However, we are still able to significantly reduce the bandwidth required. Especially when combining compute intensive applications together. It is also worth noting that the largest increase in bandwidth usage is a mere 0.1%, while the largest decrease is 26.8%.

RPT prefetching is the most accurate prefetching heuristic studied in this thesis. In table 4.10, I have compared its performance with the bandwidth-aware version. Because RPT prefetching is very accurate it produces few unnecessary prefetches, thus the gains by using bandwidth-aware prefetching is minor. The largest decrease in performance occurs for Eon in combination with Art (5.4%). This is also the only difference larger than 2% observed. However, there is only small reductions in bandwidth, which again is due to the high accuracy of the RPT heuristic.

By increasing the threshold value, one also decreases the effects of bandwidth-aware prefetching. The next experiments are conducted by setting the threshold value to 400. In other words, we allow prefetching to continue, even if there are two or three memory requests on average in the queue.

In table 4.11, I have compared bandwidth-ware sequential prefetching to the bandwidth-oblivious implementation. When using the increased threshold, we observe no significant performance degradation (the largest occurs for Swim combined with Gzip for a 0.3% decrease). On the other hand, we observe performance increases up to 1.7% when combining Crafty with Gzip. There are significant bandwidth reductions as well (up to 38.5%), especially in group 1.

In table 4.12, I have conducted the same experiment, but with C/DC prefetching. Again, we observe little impact on performance and a small, but significant impact on bandwidth usage. The largest decrease in bandwidth usage occurs in group 1. This is

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.5 %	+0.0 %	+1.1 %	-0.4 %	-0.3 %	-0.5 %	+0.9 %	-0.1 %	-0.1 %	-0.0 %
gcc	+0.9 %	+0.6 %	+1.8 %	+0.1 %	+0.1 %	+0.4 %	+0.2 %	+0.1 %	+0.6 %	+0.0 %
crafty	+3.2 %	+2.5 %	+4.0 %	+2.0 %	+1.6 %	+2.5 %	+0.6 %	+0.5 %	+1.6 %	+0.0 %
eon	+0.0 %	+0.0 %	+0.4 %	+0.1 %	+0.0 %	+0.2 %	+0.2 %	-0.0 %	+0.1 %	+0.0 %
twolf	-0.1 %	-0.0 %	+0.2 %	-0.0 %	-0.0 %	+0.0 %	+0.1 %	+0.0 %	+0.1 %	+0.0 %
swim	-0.3 %	+0.0 %	-0.0 %	+0.1 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.5 %	+0.0 %
mgrid	-2.5 %	-2.9 %	-2.4 %	-5.0 %	-5.3 %	-1.1 %	+0.1 %	-0.2 %	-1.0 %	+0.0 %
ammp	+0.0 %	-0.1 %	-0.1 %	-0.1 %	-0.1 %	-0.4 %	+0.2 %	+0.0 %	+1.7 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.2 %	+0.1 %	+0.0 %	+0.0 %	+0.0 %	-0.3 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-6.7 %	-12.5 %	-21.5 %	-3.5 %	-4.0 %	-10.6 %	-2.2 %	-0.5 %	-10.1 %	+0.1 %
gcc	-17.7 %	-16.3 %	-21.6 %	-19.4 %	-18.7 %	-12.7 %	-1.4 %	-3.9 %	-15.0 %	-0.0 %
crafty	-40.9 %	-37.2 %	-23.3 %	-47.8 %	-46.8 %	-26.5 %	-3.2 %	-12.8 %	-37.7 %	-0.1 %
eon	-26.8 %	-26.6 %	-46.5 %	-34.4 %	-14.4 %	-24.7 %	-6.2 %	-0.4 %	-24.8 %	-0.4 %
twolf	-14.2 %	-19.0 %	-33.7 %	-7.6 %	-12.0 %	-13.2 %	-4.4 %	-1.0 %	-24.1 %	-0.2 %
swim	-9.2 %	-9.0 %	-16.2 %	-13.7 %	-10.5 %	-17.1 %	-1.8 %	-0.3 %	-14.9 %	+0.0 %
mgrid	-2.4 %	-2.5 %	-2.2 %	-4.3 %	-4.8 %	-0.9 %	+0.1 %	-0.1 %	-1.2 %	+0.0 %
ammp	-0.1 %	-0.2 %	-1.1 %	-0.2 %	-0.4 %	-0.5 %	+0.2 %	-0.2 %	+1.7 %	-0.0 %
wupwise	-11.3 %	-17.8 %	-14.2 %	-8.5 %	-17.4 %	-14.5 %	-2.1 %	-4.2 %	+0.0 %	+0.5 %
art	-0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.2 %	+0.0 %

Table 4.8: Bandwidth-aware prefetching using sequential prefetching on a CMP. Threshold is set to 40.

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.4 %	-0.3 %	-0.2 %	-0.4 %	-0.5 %	+0.2 %	+0.9 %	-0.1 %	-0.2 %	-0.1 %
gcc	-0.1 %	-0.0 %	-0.2 %	-0.1 %	-0.1 %	+0.7 %	+1.0 %	-0.1 %	+0.0 %	+0.4 %
crafty	+0.0 %	+0.1 %	+0.2 %	+0.1 %	+0.0 %	+0.6 %	+0.6 %	+0.0 %	+0.3 %	+0.0 %
eon	+0.0 %	-0.0 %	+0.0 %	+0.3 %	+0.0 %	+0.1 %	+0.2 %	-0.0 %	+0.0 %	+0.0 %
twolf	-0.1 %	-0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.1 %	+0.5 %	-0.0 %	+0.0 %	+0.1 %
swim	-0.0 %	-0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.5 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %
mgrid	-3.7 %	-4.7 %	-4.7 %	-6.0 %	-6.8 %	-1.9 %	+0.0 %	-0.4 %	-3.2 %	-0.1 %
ammp	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.3 %	+0.0 %	+0.2 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	-0.6 %	-0.2 %	-0.2 %	-0.4 %	-0.4 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-5.3 %	-2.2 %	-1.4 %	-2.1 %	-1.9 %	-2.4 %	-4.1 %	+0.0 %	-3.0 %	-0.2 %
gcc	-3.4 %	-3.1 %	-3.3 %	-4.1 %	-3.6 %	-6.2 %	-1.8 %	-1.1 %	-3.9 %	-0.1 %
crafty	-1.0 %	-1.1 %	-2.8 %	-1.8 %	-0.7 %	-3.4 %	-1.3 %	-0.3 %	-1.4 %	+0.1 %
eon	-13.1 %	-6.2 %	-3.6 %	-26.8 %	-3.7 %	-11.2 %	-4.1 %	-0.4 %	-11.0 %	-0.4 %
twolf	-5.3 %	-4.3 %	-1.9 %	-1.9 %	-3.6 %	-5.3 %	-7.3 %	-0.1 %	-6.8 %	-0.7 %
swim	-6.1 %	-4.2 %	-6.9 %	-7.5 %	-5.6 %	-6.6 %	-0.8 %	-0.4 %	-7.1 %	-0.0 %
mgrid	-3.6 %	-4.0 %	-4.0 %	-8.3 %	-6.1 %	-1.6 %	-0.0 %	-0.4 %	-2.6 %	-0.0 %
ammp	+0.0 %	-0.0 %	-0.1 %	-0.0 %	-0.0 %	-0.0 %	+0.1 %	+0.0 %	+0.2 %	+0.0 %
wupwise	-5.5 %	-3.8 %	-3.4 %	-7.2 %	-6.0 %	+0.1 %	+0.2 %	-0.9 %	-5.0 %	-0.2 %
art	-0.3 %	-0.4 %	-0.5 %	-0.7 %	-1.0 %	-0.1 %	-0.0 %	-0.1 %	-0.1 %	+0.0 %

Table 4.9: Bandwidth-aware prefetching using C/DC prefetching on a CMP. Threshold is set to 40.

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.2 %	-0.1 %	-0.1 %	-0.4 %	-0.4 %	+0.0 %	+0.2 %	-0.1 %	-0.1 %	+0.2 %
gcc	+0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.1 %	+0.0 %	-0.0 %	-0.0 %	+0.2 %
crafty	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	-0.1 %
eon	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	-5.4 %
twolf	-0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.0 %
swim	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %
mgrid	-0.8 %	-0.8 %	-1.0 %	-0.9 %	-1.2 %	-0.1 %	-0.0 %	-0.2 %	-0.1 %	+0.0 %
ammp	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.0 %	+0.0 %	+0.0 %	-0.2 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.1 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.0 %	-0.0 %	-0.4 %	-0.5 %	-0.3 %	-0.2 %	-1.8 %	-0.1 %	+0.9 %	-0.2 %
gcc	-0.4 %	-0.2 %	-0.3 %	-0.1 %	-0.1 %	-0.5 %	+0.1 %	+0.0 %	-0.0 %	+0.3 %
crafty	-0.0 %	-0.0 %	-0.2 %	+0.0 %	-0.1 %	-0.1 %	-0.1 %	-0.0 %	+0.0 %	+0.1 %
eon	-2.9 %	-0.3 %	-0.4 %	-0.1 %	-0.9 %	-0.2 %	-0.9 %	+0.1 %	-0.0 %	+1.7 %
twolf	+0.2 %	-0.3 %	-0.3 %	-0.0 %	-0.1 %	-0.0 %	-1.1 %	+0.0 %	+0.0 %	-0.3 %
swim	-0.3 %	-0.2 %	-0.1 %	-0.0 %	-0.1 %	-0.0 %	-0.8 %	-0.0 %	-0.0 %	+0.3 %
mgrid	-0.7 %	-0.6 %	-0.9 %	-1.4 %	-1.1 %	-0.1 %	+0.0 %	-0.1 %	-0.1 %	+0.0 %
ammp	-0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	-0.0 %	+0.0 %	-0.1 %	-0.0 %	+0.3 %
wupwise	-0.0 %	-0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.0 %
art	-0.1 %	-0.1 %	-0.1 %	-0.3 %	-0.1 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %	+0.0 %

Table 4.10: Bandwidth-aware prefetching using RPT prefetching on a CMP. Threshold is set to 40.

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.3 %	-0.2 %	+0.4 %	-0.2 %	-0.1 %	-0.6 %	-0.0 %	-0.1 %	+0.3 %	+0.0 %
gcc	+0.4 %	+0.1 %	+1.0 %	+0.1 %	+0.1 %	+0.4 %	+0.0 %	+0.1 %	+0.6 %	-0.0 %
crafty	+1.7 %	+1.1 %	+0.6 %	+1.5 %	+1.2 %	+0.9 %	+0.0 %	+0.4 %	+0.9 %	+0.0 %
eon	+0.0 %	+0.0 %	+0.3 %	+0.1 %	+0.0 %	+0.2 %	-0.0 %	+0.0 %	+0.1 %	+0.0 %
twolf	-0.1 %	-0.1 %	+0.2 %	-0.0 %	-0.0 %	+0.3 %	+0.0 %	+0.0 %	+0.1 %	+0.0 %
swim	-0.3 %	+0.0 %	+0.0 %	+0.1 %	+0.0 %	+0.4 %	+0.0 %	-0.1 %	+0.0 %	+0.0 %
mgrid	+0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %
ammp	+0.0 %	-0.1 %	-0.0 %	-0.1 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.8 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+1.3 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-3.5 %	-6.4 %	-11.2 %	-2.7 %	-2.9 %	-6.7 %	+0.2 %	-0.2 %	-6.2 %	+0.0 %
gcc	-9.1 %	-7.1 %	-10.4 %	-12.4 %	-12.4 %	-4.0 %	-0.0 %	-2.3 %	-5.7 %	+0.0 %
crafty	-23.3 %	-19.8 %	-3.6 %	-38.4 %	-39.1 %	-7.8 %	-0.0 %	-9.6 %	-13.3 %	+0.0 %
eon	-21.6 %	-19.4 %	-38.5 %	-33.5 %	-11.1 %	-20.4 %	+0.2 %	-0.4 %	-24.4 %	-0.2 %
twolf	-10.2 %	-11.9 %	-28.0 %	-6.5 %	-10.6 %	-12.3 %	-0.0 %	-0.7 %	-21.7 %	-0.2 %
swim	-5.9 %	-5.4 %	-7.0 %	-12.4 %	-9.4 %	+0.4 %	-0.0 %	-0.2 %	+0.0 %	-0.0 %
mgrid	-0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.1 %	+0.0 %
ammp	-0.0 %	-0.1 %	-0.8 %	-0.1 %	-0.2 %	-0.2 %	+0.0 %	-0.2 %	+0.8 %	+0.0 %
wupwise	-12.7 %	-12.9 %	-9.5 %	-8.4 %	-17.3 %	-1.7 %	+0.1 %	-4.1 %	-0.0 %	-1.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.3 %	+0.0 %

Table 4.11: Bandwidth-aware prefetching using sequential prefetching on a CMP. Threshold is set to 400.

especially true when combining Eon with Eon (26.4% decrease).

In the previous experiment with RPT we observed little impact from using bandwidth-aware prefetching. By increasing the threshold to 400 (table 4.13), there are even less impact. However, there is a significant decrease when combining Gcc with Eon (29%). Again, the minor effect of bandwidth-aware prefetching is due to the high accuracy of the RPT prefetching heuristic.

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.1 %	-0.1 %	-0.1 %	-0.1 %	-0.1 %	+0.1 %	-0.0 %	-0.0 %	-0.1 %	+0.0 %
gcc	-0.1 %	-0.0 %	-0.1 %	-0.0 %	-0.0 %	+0.4 %	-0.0 %	-0.0 %	+0.1 %	+0.0 %
crafty	+0.0 %	+0.0 %	+0.1 %	+0.1 %	+0.0 %	+0.2 %	-0.0 %	+0.0 %	+0.1 %	-0.0 %
eon	+0.0 %	+0.0 %	+0.0 %	+0.3 %	+0.0 %	+0.1 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %
twolf	-0.1 %	-0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.1 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %
swim	-0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.5 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %
mgrid	+0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %
ammp	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.2 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-2.1 %	-1.0 %	-0.8 %	-1.1 %	-1.1 %	-1.2 %	+0.1 %	+0.0 %	-0.8 %	+0.0 %
gcc	-1.4 %	-1.1 %	-1.2 %	-2.0 %	-1.9 %	-2.6 %	-0.1 %	-0.5 %	-1.4 %	+0.0 %
crafty	-0.3 %	-0.3 %	-0.5 %	-1.4 %	-0.4 %	-1.3 %	+0.1 %	-0.2 %	-0.4 %	-0.0 %
eon	-8.2 %	-4.0 %	-3.7 %	-26.4 %	-2.8 %	-7.0 %	+0.1 %	-0.2 %	-9.6 %	-0.0 %
twolf	-2.9 %	-2.3 %	-1.5 %	-1.5 %	-3.1 %	-4.0 %	-0.1 %	-0.1 %	-6.0 %	-0.1 %
swim	-3.0 %	-1.9 %	-2.6 %	-5.3 %	-3.9 %	-1.0 %	+0.0 %	-0.1 %	-0.2 %	-0.0 %
mgrid	+0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %
ammp	-0.0 %	-0.0 %	-0.1 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.1 %	+0.2 %	+0.0 %
wupwise	-4.6 %	-2.4 %	-1.2 %	-7.0 %	-5.8 %	+0.1 %	+0.1 %	-0.7 %	+7.2 %	-0.0 %
art	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %

Table 4.12: Bandwidth-aware prefetching using C/DC prefetching on a CMP. Threshold is set to 400.

(a) Speedup in IPC.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.1 %	-0.0 %	-0.0 %	-0.1 %	-0.1 %	-0.0 %	+0.0 %	-0.1 %	+0.0 %	+0.0 %
gcc	-0.0 %	-0.0 %	+0.0 %	+3.1 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
crafty	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %
eon	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
twolf	-0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
swim	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
mgrid	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
ammp	+0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.1 %	+0.0 %	+0.0 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %

(b) Reductions in bandwidth usage.

	gzip	gcc	crafty	eon	twolf	swim	mgrid	ammp	wupwise	art
gzip	-0.4 %	+0.0 %	-0.1 %	-0.1 %	-0.1 %	+0.1 %	-0.0 %	-0.1 %	-0.1 %	+0.0 %
gcc	+0.2 %	-0.1 %	-0.1 %	-29.0 %	-0.1 %	-0.2 %	+0.0 %	-0.0 %	-0.0 %	+0.0 %
crafty	+0.0 %	-0.0 %	-0.1 %	+0.0 %	-0.1 %	-0.1 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %
eon	-2.1 %	-0.4 %	-0.1 %	-0.1 %	-1.0 %	-0.4 %	-0.2 %	-0.0 %	-0.0 %	+0.0 %
twolf	+0.2 %	-0.1 %	-0.2 %	-0.0 %	-0.1 %	-0.0 %	+0.0 %	-0.0 %	+0.0 %	+0.0 %
swim	-0.4 %	-0.2 %	-0.1 %	-0.0 %	-0.1 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	-0.0 %
mgrid	-0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	+0.0 %	-0.0 %	-0.0 %	+0.0 %
ammp	+0.0 %	+0.0 %	-0.0 %	-0.0 %	-0.0 %	-0.0 %	+0.0 %	-0.1 %	-0.0 %	+0.0 %
wupwise	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	-0.0 %	+0.0 %	-0.0 %	-0.0 %
art	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %	+0.0 %

Table 4.13: Bandwidth-aware prefetching using RPT prefetching on a CMP. Threshold is set to 400.

Chapter 5

Discussion

5.1 The Simulator

Much of the work done in this thesis was implementing the simulator. In the next sections I will look at some of the choices I have made.

5.1.1 Prefetching

I have built upon the framework for prefetching simulation that I developed in the fifth year project. Although the framework can handle a large array of prefetching heuristics, I have only been able to implement 8 reference heuristics (including the perfect L2), due to time constraints. I selected sequential prefetching because it is simple and effective, and has been thoroughly studied in the literature. C/DC- and its predecessor DC-prefetching are relatively new heuristics that have been proposed. RPT prefetching is an old heuristic, but has a very high accuracy (as demonstrated in the previous chapter). AVD was included because it is one of the few heuristics designed to handle pointer-chasing problems.

This selection ranges from the very simple, inaccurate heuristics to the complex and accurate. In addition, the area requirements vary considerably across implementations, in addition to power requirements. For instance, a full RPT-prefetcher would not be the ideal choice for a small embedded processor, as opposed to the much simpler sequential heuristic. In addition, RPT prefetching requires information from inside the core, which might delay the critical path, and reduce system performance. Thus, all prefetching heuristics have their place in modern processor design.

The framework was extended in many ways. It now checks if the prefetched address is within the programs address space, so that the simulator does not crash in the cache-handling code. This was a problem in the fifth year project, especially with DC prefetching and high prefetching degrees. In addition, the framework has been optimized for performance by moving much of the logic so that computation is not performed unnecessary.

5.1.2 DRAM Model

Early in this work it became obvious that a good DRAM model was key to understanding the performance of the various prefetching heuristics. In my previous work I had used a much simpler model that only addressed contention. This was insufficient, especially considering the complexity of modern DRAM system.

An important factor for moving to another model was open pages. Open pages have a significant impact on prefetching. Consider for instance sequential prefetching, because it always fetches the next cacheline, it is likely that it will hit on an open page, and will thus experience a reduced latency. In effect, all prefetching heuristics that work by detecting memory access patterns will benefit from open pages. Thus, this facet of the DRAM model contributes significantly to the performance of prefetching.

In addition, by allowing pipelining of requests, it becomes more favorable to prefetch, simply because the impact of adding another DRAM request is reduced under high bandwidth utilization.

However, because of the increased complexity of the model, it becomes more difficult to analyze the results. In effect, the latency of a DRAM operation is no longer fixed. To compensate for this problem a number of performance counters have been added to highlight the effects of the new DRAM model. The number of accesses that hit open pages have been especially useful for analyzing the results.

5.1.3 CMP Extension

The CMP extension is a result from cooperation with Haakon Dybdahl. Although it works very well, it has it's limitations [11].

CMPs are targeted at parallel applications. It is therefore a serious shortcoming that the simulator is unable to simulate parallel benchmarks. As mentioned in section 3.3.3 this shortcoming is inherit from SimpleScalar. It is possible to execute parallel code, and it has been done in other projects, but it requires a rewrite of the program to use special system calls. Otherwise, this issue would require an almost complete rewrite of the SimpleScalar code, which would be unfeasible.

The other issue with the simulator is its inability to execute operating system code. It has been previously shown that OS code can significantly impact the performance of a system. However, this effect can be compensated, as this factor is equal for all experiments. The problem lies in the inability of the simulator to support dynamic libraries, network code, native compilation and so on.

The simulator uses a very inefficient way to simulate CMP. Because the two cores are simulated in separate processes, there is considerable overhead in context switching. In addition, the two cores are synchronized on every clock cycle. As already mentioned, this is suboptimal, as it is only necessary to serialize accesses to the L2 cache, which does not occur as frequent.

For these three reasons the NCAR group has been looking at other possible simulators. Arnt Jørgen Lande's diploma work analyzes different simulators and will make a recommendation for a new simulator. I will comment more on this topic in the future work section (6.3).

5.2 Methodology

Simulations of CMPs offers a vast experimental space. The uniprocessor cores have 82 parameters each. In addition, there is a large number of possible benchmarks to run. Much of the work in this thesis has been to simply reduce the experimental degrees of freedom, in such a way that prefetching becomes the dominant factor in realistic conditions.

5.2.1 Benchmark Selection

The number of benchmarks that are possible to run are limited due to the constrains imposed by the simulator. Because it is not possible to run true parallel benchmarks it was a satisfactory choice to use the SPEC2000 benchmark suite. I have used this suite previously, and it is very common in the literature. In addition, it is necessary to compare uniprocessor results with CMP results, if both use the same benchmark.

I have used two separate datasets for SPEC2000. The Lgred datasets are reduced datasets which is said to behave in much the same manner as the reference sets. However, in practice there is a significant difference as the reduced sets have a much smaller memory footprint, and thus fits easier into the cache.

5.2.2 Simulation Environment

Most of the simulations in this thesis were conducted on the Clustis2 cluster located at IDI. Clustis2 is composed of 22 nodes, each with a Pentium 4 processor. As my experiments are trivially parallelizable (each experiment is independent from the others), using this computer provides an approximate 22 times speedup.

Python was used extensively, both to construct and submit jobs to the Clustis queue system (OpenPBS). As each experiment produces a lot of data (approx 200 MB), Python was also extensively used to analyze the data. Some of the scripts used can be found in the appendix.

In addition, Musculus (musculus.hpc.ntnu.no) was also used for this project. Musculus is an Cray XD-1. It has 12 64-bit Opteron CPU with 24 GB's of RAM. However, it could only be used for uniprocessor experiments, as two and two CPU's share memory (and would thus conflict when running shared-memory based experiments). A possibility would be to simply randomize or probe shared memory usage. This was not done, as the computing power of Clustis2 was sufficient for this project.

5.3 Results

5.3.1 Uniprocessor Results

By comparing the performance of a system with no prefetching with the perfect L2 case it is clear that for some of the benchmarks there is a very large opportunity for prefetching. As en example, the performance of Art increases by 529 % with a perfect L2. Other benchmarks gain very little from a perfect L2 (such as VPR).

Some of the prefetching methods come very close to this theoretical limit. This is especially visible with RPT and stream prefetching on the most memory-intensive applications such as Ammp and Art. This is due to these heuristics having an accuracy close to 100%. As such it generates very few unnecessary prefetches. In addition, the coverage is also close to 100%, which indicates that it exploits almost all opportunities to prefetch. On the other hand, RPT prefetching is expensive to implement. It requires large cache-like structures that consume both area and power.

A much simpler alternative is sequential prefetching. It is easy to implement and gives considerable performance increases. However, it is not as accurate as RPT prefetching, and therefore generates a lot of unnecessary prefetches.

In between, there are DC and C/DC prefetching. They are more complex than sequential prefetching, but less complex than RPT prefetching. They are more accurate than sequential prefetching, and get better results.

AVD prefetching had little effect on the benchmarks used. This is probably due to the fact that the benchmarks used was not pointer-intensive.

Increasing the size of the tables used by C/DC and RPT prefetching generally increases performance. However, the most significant increases occur for relatively small tables. This indicates that the most memory intensive loads occur in relatively tight loops, which is in-line with previous experiments.

It is also worthwhile to note the optimum CZone size (around 256KB). This indicates how large the different memory regions in these benchmarks are. It is therefore reasonable that the optimum CZone size is application dependant.

Increasing the prefetching degree generally increases performance if there is enough bandwidth available to support it. In other words, the effect of prefetching new data outweighs the disadvantages of displacing old data in the cache. However, in bandwidth-limited situation, sequential prefetching might decrease performance when increasing the prefetching degree. This effect does not occur with the more advanced heuristics such as C/DC and RPT.

5.3.2 CMP Results

When experimenting with a CMP, I have looked at 2-way CMP, instead of a system with more cores. This decision was driven by a need for simplicity of analysis. More cores would necessarily both increase computing time as well as the complexity of the analysis without gaining much new information. Although it is possible to buy systems with up to 64 cores on a single die, they are not general purpose, and is thus not as interesting for this type of research.

I have looked at how increasing the number of DRAM channels, the number of sets and the associativity of the L2 affects performance. Increasing the number of DRAM channels has a significant impact on benchmarks that are memory bound such as Ammp and Art, while increasing the number of sets has a comparatively small impact compared to the area requirements. On the other hand, doubling the associativity has a very large impact on performance. This is in-line with similar research.

Prefetching in a CMP is a double-edged sword. For some benchmarks (mainly memory bound) it can have a large positive impact, but it can also reduce performance for others. This is especially true for the sequential prefetching heuristic.

As accuracy increases, so does the performance. It is interesting to note that the performance of RPT prefetching is not as high as can be expected given the uniprocessor results. This is both due to a change in problem size and a result of the limited time available to initialize the RPT structures.

5.3.3 Bandwidth-Aware Prefetching

Bandwidth-aware prefetching is a new prefetching heuristic proposed in this thesis. It is based on using performance counters to direct prefetching. By predicting future bandwidth requirements, it is possible to use that information to gain a performance benefit. The idea is based on the assumption that it is better to prefetch when there is little bandwidth contention.

In the previous two chapters I have examined the performance of bandwidth-aware prefetching in a uniprocessor setting and in a CMP. It is clear that bandwidth-aware prefetching works best when the prefetchers accuracy is low. As such it would be interesting to combine this heuristic with a heuristic that predicts the accuracy of the prefetching heuristic. This subject will be further discussed in section 6.3.

Taking the average of the last three latency values was a relatively good predictor. The Network Weather Service project [66] use several heuristics to predict future bandwidth usage and use the best heuristic for each prediction. In a processor, one cannot afford to use such advanced methods. Due to area constraints one cannot afford to use a very compute- or storage- intensive predictor.

Bandwidth-aware prefetching worked quite well in most cases. Most benchmarks had relatively little changes in performance due to it being used. However, most applications received a drastic reduction in the number of DRAM accesses (up to 47.8%).

5.4 Workflow

In this section I will briefly discuss the tools that I have used in this thesis and my experience with them. This will hopefully be useful for someone who are planning to do something similar.

For development I have used C and the Gcc compiler. I experimented with Icc (Intel C compiler), and it did provide a speedup of about 20%. However, it produces code specific for the Pentium series of processors, and it's performance is therefore less on AMD processors. In addition, Icc is licensed software[1], which made it easier for me to simply use Gcc on all platforms.

Subversion was used as the version control system. This worked very well, and gave little problems. In effect, I had two repositories, one for CMP development (which I share with the rest of the NCAR group) and a personal repository. This repository was also used for documentation and the preparation of this document.

Latex was used as the primary typesetting tool. As it is a plain-text format I could use subversion for version control for this part of the work as well.

Python was used extensively both to set up and execute simulations, but also for data analysis.

Gnuplot was used to plot all graphs contained in this document. It was easy to integrate it with Python in such a way that getting a graphical representation of the results quickly became possible.

Lastly, OpenPBS was used for batchprocessing on the Clustis2 cluster.

[1]Although free for academic use.

Chapter 6

Conclusion

6.1 Results

In this thesis I have looked at several types of prefetching. Experiments with the simulator have obtained comparable results to those in the literature. This builds confidence in the simulator framework.

The most interesting result is the Bandwidth-aware results. The development of this heuristic was driven through a hypothesis that reducing bandwidth usage of prefetching would lead to increased performance in CMP. Bandwidth-aware prefetching does reduce bandwidth usage significantly (up to 47.8%), but it does not give a large increase in IPC. However, for most applications IPC does not change significantly. The performance of bandwidth-aware prefetching is closely correlated to the accuracy of the prefetching heuristic. If the accuracy of the prefetching heuristic is high, then there is little to gain from rejecting prefetches.

In CMP systems, the largest gains are from compute-bound applications, which is natural, considering that most compute-bound applications have more erratic memory access patterns.

During the experimentation with bandwidth-aware prefetching I have discovered several factors that can be exploited to increase the performance of the heuristic. These will be explored further in future work.

6.2 Contributions

To summarize; In this thesis I have:

- Investigated performance counters in modern processors.
- Developed kernel modules for Linux for exploring performance counters.
- Investigated modern prefetching methods.
- Expanded the simulator used in my fifth year project.
- Expanded the simulator to include a more realistic DRAM model.
- Expanded the simulator to simulate CMP architectures.

- Developed a methodology to benchmark the performance of CMP architectures.
- Developed an understanding of current prefetching heuristics by conducting experiments.
- Developed an understanding of known prefetching heuristics in a CMP setting by performing simulations.
- Developed a *new* prefetching heuristic based on performance counters.
- Documented its performance through experimentation.

6.3 Future work

As this master thesis is the ground work for further work in my PhD thesis, I would like to highlight some possible future work:

6.3.1 Simulator

The CMP simulator based on SimpleScalar has some limitations. There exists many simulators that are already capable of simulating CMPs. Arnt Jørgen Lande is currently working on diploma thesis that evaluates many simulators and I intend to use his work to consider switching simulator. The purposes of such a switch are:

- A larger community using the simulator.
- A possible speedup.
- Ability to run true parallel benchmarks.
- Ability to run OS code.

From Arnt Jørgens initial report, it looks like the M5 simulator [80] is a viable alternative. M5 is loosely based on some SimpleScalar code, which would be beneficial, as I am familiar with this code.

As always, extending the simulator framework to support more types of prefetching would be beneficial when designing heuristics that direct prefetching, such as bandwidth-aware prefetching. Markov predictors [81] would be especially interesting to implement, as it has shown very good results in the past.

6.3.2 Interactions

A computer system is very complex and prefetching interacts with other parts of the system. It would be interesting to see how prefetching interacts with alternative cache replacement algorithms. In a uniprocessor the LRU algorithm has been the dominant replacement algorithm for decades, however it is not certain that this algorithm is the optimal one for CMPs. Therefore a lot of different other algorithms have been recently developed (Haakon Dybdahl has done some work in this area [11]).

Prefetching will also necessarily interact with the cache consistency mechanisms. In a tightly coupled application, this might have a large impact on performance if the prefetcher generates unnecessary invalidates.

Another new and interesting topic in CMP is cache partitioning among cores [82]. These schemes dynamically divides the cache into partitions so that each core has its own subset of the cache. The idea is that some applications require larger caches than others and this can be dynamically determined.

6.3.3 Bandwidth-Aware Prefetching

It is obvious that the efficiency of bandwidth-aware prefetching is determined by the accuracy of the prefetching heuristic. Therefore it would be interesting to estimate the accuracy of prefetching heuristics and use this as an additional input to the bandwidth-aware scheme. However, estimating accuracy is difficult. Prefetched data might not be used for several million clock cycles, and tagging every cache line would be expensive in terms of hardware. A possible solution would be to use shadow tags, that mimic parts of the original cache and are only a few sets big. Such a structure does not hold actual data, only metadata such as tags and LRU placement. By using sampling theory it might be possible to get an estimate of the accuracy of a given algorithm at runtime.

Another possible venue for this type of heuristic is to control the prefetching degree, rather than a binary on/off. Some initial experimentation was conducted based on this approach. It yielded good results for some benchmarks, but performed badly for others. It was therefore dropped in favor of the one presented in this thesis. However, with an accuracy estimator it might be possible to improve this heuristic.

In addition, adding a DRAM controller with a capability of providing priorities and preemption might increase performance [83]. Such a memory controller could be used to give prefetches a relatively low priority, while giving a higher priority to actual loads. This could be expanded to provide fairness across cores.

6.4 Acknowledgments

I would like to acknowledge the following persons for their help with this thesis:

- Lasse Natvig - For his guidance and tutoring.
- Haakon Dybdahl - For working with me with the CMP simulator and exchanging his ideas.
- Cyril Banino-Rokkones - For providing me with example PAPI code and proofreading.
- Thorvald Natvig - For helping me with understanding x86 performance counters.
- Hanne Lian - For her support.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, Apr. 1965.

[2] International Technology Roadmap for Semiconductors, "ITRS roadmap," 2004. `http://www.itrs.net/Common/2004Update/2004Update.htm`.

[3] Wikipedia, "Moore's law," 2006. `http://en.wikipedia.org/wiki/Moore%27s_Law`.

[4] J. L. Hennesey and D. A. Patterson, *Computer Architecture.* 340 Pine Street, Sixth Floor, San Fransisco, CA 94104-3205, USA: Morgan Kaufmann Publishers, 2003.

[5] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM Computer Architecture New*, vol. 23, march 1995.

[6] D. A. Patterson, "Computer science education in the 21st century," *Commun. ACM*, vol. 49, no. 3, pp. 27–30, 2006.

[7] D. A. Patterson, "Latency lags bandwith," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[8] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, pp. 34–44, 1997.

[9] 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)*, pp. 233–244, 2005.

[10] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou, "ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing," in *38th Annual International Symposium on Microarchitecture (MICRO-38)*, pp. 245–256, 2005.

[11] H. Dybdahl and P. Stenström, "Enhancing lower level cache performance by early miss determination and block bypassing," in *Prooceedings of ACSAC*, 2006.

[12] L. Spracklen and S. G. Abraham, "Chip multithreading: Opportunities and challenges," in *11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pp. 248–252, 2005.

[13] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[14] P. Shivakumar and N. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Tech. Rep. 2, Compaq Western Research Laboratory, August 2001.

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

[16] S. VanderWiel, "A survey of data prefetching techniques," Tech. Rep. 5, University of Minnesota, October 1996.

[17] I.-C. K. Chen, C.-C. Lee, and T. Mudge, "Instruction prefetching using branch prediction information," in *IEEE International Conference on Computer Design*, p. 593, 1997.

[18] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, p. 165, 1996.

[19] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *11th International Symposium on High-Performance Computer Architecture*, pp. 225–236, 2005.

[20] 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*, (New York, NY, USA), pp. 40–52, ACM Press, 1991.

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

[22] W. Zhenlin, D. Burger, K. McKinley, S. Reinhardt, and C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 388–398, June 2003.

[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*, vol. 7, pp. 385–398, Apr. 1996.

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

[25] 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*, pp. 135–145, 2004.

[26] 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*, pp. 62–73, 2002.

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

[28] 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*, pp. 111–121, 1999.

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

[30] J. Fritts, "Multi-level memory prefetching for media and stream processing," in *2002 IEEE International Conference on Multimedia and Expo, 2002. ICME '02. Proceedings*, vol. 2, pp. 101–104, Aug. 2002.

[31] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 50, 2002.

[32] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4/5, 2005.

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

[34] Rambus Inc., "XDR dram : System design overview," 2006. `http://www.rambus.com/products/xdr/index.aspx`.

[35] Philips Semiconductors, "Tm-1300 media processor data book."

[36] M. Ekman and P. Stenström, "Performance and power impact of issue-width in chip-multiprocessor cores," in *International Conference on Parallel Processing*, 2003.

[37] L. Benini and G. D. Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, pp. 70–78, Jan. 2002. `http://ieeexplore.ieee.org/iel5/2/21069/00976921.pdf?isnumber=21069\begingroup\let\relax\relax\endgroup[Pleaseinsert\PrerenderUnicode{âĹŔ}intopreamble]=JNL\&arnumber=976921\&arnumber=976921\&arSt=70\&ared=78\&arAuthor=Benini%2C+L.%3B+De+Micheli%2C+G.`

[38] IBM, "Octopiler webpage," 2006. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/cellcompiler.index.html`.

[39] Advanced Micro Devices, Inc, "Amd athlon 64 x2 key architectural features web page," 2006. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9485_13041%5E13043,00.html`.

[40] R. Shrout, "Amd athlon 64 x2 4400+ dual core processor review," 2005. `http://www.pcper.com/article.php?aid=141\&type=expert`.

[41] K. Krewell, "Sun weaves multithreaded future," *Microprocessor Report*, Apr. 2003.

[42] J. D. Gelas, "Sun's ultrasparc t1 - the next generation server cpus," 2005. `http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2657\&p=3`.

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

[44] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation cell processor," in *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 184–185, February 2005.

[45] 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.

[46] N. Blachford, "Cell architecture explained," 2005. `http://www.blachford.info/computer/Cell/Cell0_v2.html`.

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

[48] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer*, 2002.

[49] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," 1997. `http://www.simplescalar.com/docs/users_guide_v2.pdf`.

[50] N. Manjikian, "More enhancements of the simplescalar tool set," *SIGARCH Comput. Archit. News*, vol. 29, no. 4, pp. 5–12, 2001.

[51] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture, 2000*, pp. 83–94, 200.

[52] E. Larson, S. Chatterjee, and T. Austin, "The mase microarchitecture simulation environment," in *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, June 2001.

[53] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, Feb. 2002.

[54] T. F. Wenisch and R. E. Wunderlich, "Simflex: Fast, accurate and flexible simulation of computer systems," November 2005. Tutorial in the International Symposium on Microarchitecture (MICRO-38).

[55] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation, 2nd ed.* 84 Theobalds Road, London WC1X 8RR, UK: Academic Press, 2000.

[56] 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.

[57] Intel corporation, "IA-32 intel architecture software developer's manual," January 2006.

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

[59] Discretix Technologies Ltd, "Introduction to side channel attacks," 2006. `http://www.hbarel.com/publications/Introduction_To_Side_Channel_Attacks.pdf`.

[60] AMD, "AMD Athlon Processor, x86 Code Optimization Guide," February 2002. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf`.

[61] J. Levon, *OProfile manual*, 2004. `http://oprofile.sourceforge.net/doc/index.html`.

[62] Intel corporation, "VTune website." `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm`.

[63] 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*, pp. 222–233, 1999.

[64] Wikipedia, "DDR2," 2005. `http://en.wikipedia.org/wiki/DDR-2`.

[65] Corsair Memory Inc, "CM2X512A-6400." `http://www.corsairmemory.com/corsair/products/specs/CM2X512A-6400.pdf`.

[66] R. Wolski, "Experiences with predicting resource performance on-line in computational grid settings," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, pp. 41–49, March 2003.

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

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

[69] 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.

[70] "Simpoint web page," 2006. `http://www-cse.ucsd.edu/~calder/simpoint/`.

[71] S. Sair and M. Charney, "Memory behavior of the spec2000 benchmark suite." `http://citeseer.ist.psu.edu/431597.html`.

[72] Transaction Processing Performance Council, "TPC-C webpage," 2006. `http://www.tpc.org/tpcc/default.asp`.

[73] SPEC, "SPECWeb web page," 2005. `http://www.spec.org/web2005/`.

[74] SPEC, "SPECjAppServer2004 web page," 2004. `http://www.spec.org/jAppServer2004/`.

[75] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," January 2004. `http://www.netlib.org/benchmark/hpl/`.

[76] R. Biswas and R. F. V. der Wijngaart, "NAS parallel benchmarks home page," 2006. `http://www.nas.nasa.gov/Software/NPB/`.

[77] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, October 1988.

[78] B. Jacob and T. Mudge, "Notes on calculating computer performance," Tech. Rep. 231-95, University of Michigan, March 1995.

[79] Wikipedia, "Harmonic mean," 2006. `http://en.wikipedia.org/wiki/Harmonic_mean`.

[80] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, "Network-oriented full-system simulation using m5," in *CAECW - Computer Architecture Evaluation using Commercial Workloads*, Feb 2003.

[81] P. U. M. Predictors, "D. joseph," in *The 24th Annual International Symposium on Computer Architecture*, pp. 252–263, 1997.

[82] S. Kim, D. Chandra, and Y. Solhin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pp. 111–122, 2004.

[83] L. Wei-Fen, S. Reinhardt, and D. Burger, "Reducing dram latencies with an integrated memory hierarchy design," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 301–312, Jan. 2001.

Appendix A

Cacti Output

This is the output of CACTI simulating a 4-way 8KB cache with 64byte cache lines. It has 1 read and 1 write port, and the technology being used is 65nm.

```
---------- CACTI version 3.2 ----------

Cache Parameters:
  Number of Subbanks: 1
  Total Cache Size: 8192
  Size in bytes of Subbank: 8192
  Number of sets: 32
  Associativity: 4
  Block Size (bytes): 64
  Read/Write Ports: 1
  Read Ports: 1
  Write Ports: 1
  Technology Size: 0.06um
  Vdd: 0.8V

Access Time (ns): 0.57591
Cycle Time (wave pipelined) (ns):  0.273554
Total Power all Banks (nJ): 0.139284
Total Power Without Routing (nJ): 0.139284
Total Routing Power (nJ): 0
Maximum Bank Power (nJ):  0.139284

Best Ndwl (L1): 16
Best Ndbl (L1): 1
Best Nspd (L1): 1
Best Ntwl (L1): 1
Best Ntbl (L1): 4
Best Ntspd (L1): 1
Nor inputs (data): 2
Nor inputs (tag): 1

Area Components:

Cache data
        array          1.002876 (mm^2)
        pred           0.001504 (mm^2)
        colmux pred    0.000754 (mm^2)
        colmux post    0.000053 (mm^2)
        write sig      0.000872 (mm^2)
        total area     1.006058 (mm^2)
Cache tag
        array          0.061271 (mm^2)
        pred           0.000754 (mm^2)
        colmux pred    0.000754 (mm^2)
        colmux post    0.000211 (mm^2)
        out decode     0.002650 (mm^2)
        out sig        0.000872 (mm^2)
        total area     0.066512 (mm^2)
Cache
        total area    1.074315 (mm^2)
        subanked      1.126643 (mm^2)

        aspect ratio     1.43
        data ramcells   93.4%
        tag  ramcells    5.7%
        control/routing  0.9%
        efficiency      95.2%

Time Components:
 decode data      :   273.554ps    50.726pJ
 w&b line  data   :   112.349ps    15.286pJ
        wordline  :    91.638ps     0.284pJ
        bitline   :    20.711ps    15.002pJ
 sense amp data   :    67.600ps    42.025pJ
 decode tag       :    71.852ps     2.525pJ
 w&b line tag     :    55.342ps     4.302pJ
       wordline   :    45.725ps     0.221pJ
       bitline    :     9.617ps     4.080pJ
 sense amp tag    :    21.937ps     7.551pJ
 compare address  :   122.225ps     1.003pJ
 mux driver       :   185.040ps     1.683pJ
 select inverter  :    17.525ps     0.013pJ
 data output drv  :   101.988ps    14.170pJ
 total data ~drv  :   453.503ps   108.037pJ
 total tag (~DM)  :   473.922ps    17.077pJ
Total Data        :   555.491ps   123.904pJ
Total TAG         :   575.910ps    15.381pJ

Read Energy       :               139.284pJ
Write Energy      :               162.399pJ

Access Time       :   575.910ps
Max Precharge     :     0.000ps
Pipe Time (1clk)  :   575.910ps (data=  575.910ps) (tag=  575.910ps)
Pipe Time (2clk)  :   287.955ps (data=  277.746ps) (tag=  287.955ps)
Pipe Time (3clk)  :   273.554ps (data=  185.164ps) (tag=  191.970ps)

Cache bank distributions and energy per access (1 banks)
        bank ctrl       0%      0.000pJ
        decode         15%     53.250pJ
        tag  array      4%      4.302pJ
        data array     65%     15.286pJ
        tag  ctrl    0.29%     10.251pJ
        data ctrl      16%     56.196pJ
```

```
    total                   139.284pJ
```

Appendix B

Notur 2006 Poster

This poster was submitted and accepted to the Notur 2006 conference. It's purpose was to showcase work in progress.

Simulation of Bandwidth-Aware Hardware Based Prefetching in Chip Multiprocessors

NTNU Innovation and Creativity
Department of Computer and Information Science

by Marius Grannæs (`grannas@idi.ntnu.no`)
supervised by Lasse Natvig (`Lasse.Natvig@idi.ntnu.no`)
11th May 2006

NTNU Innovation and Creativity
Department of Computer and Information Science

1 Prefetching

Prefetching is a technique used to *increase the effectiveness of caches* by trying to predict the memory reference stream. By fetching needed data to the caches before it is actually referenced by the processor, it is possible to achieve a significant performance increase.

There are two things that can potentially degrade performance on uniprocessors:

- *Displacing* data from the cache that is still needed
- Causing bandwidth *contention*

There exist very good heuristics for hardware prefetching on uniprocessors, but in a CMP where the cache is shared, prefetching might displace other processors data [1, 2].

2 SimpleScalar simulator

We at the NCAR group at NTNU have developed an advanced simulator based on the open source SimpleScalar simulator. It has been extended with the following extensions:

- Added support for *chip multiprocessing*
- *Shared caches*
- *Improved DRAM-model* that mimics DDR2
- Added support for *hardware prefetching*
- Added system calls for *synchronization and shared memory*

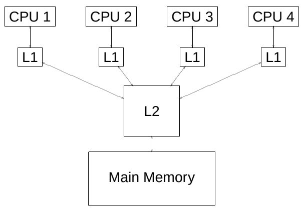

Figure 1: The simulated architecture

3 Uniprocessor results

To understand prefetching in a CMP, one must first understand how the heuristics work in a simpler setting. In figure 2, we see how the most memory intensive benchmarks in the SPEC 2000 suite performs *without any bandwidth limitations*.

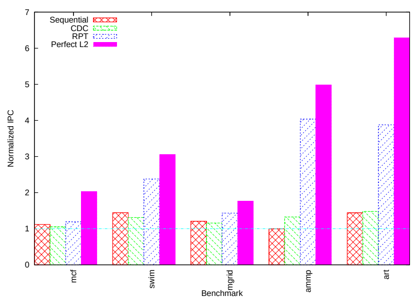

Figure 2: Prefetching on uniprocessors with unlimited bandwidth to memory. IPC is normalized to the case where no prefetching is performed.

In figure 3, we *limit the amount of bandwidth* available:

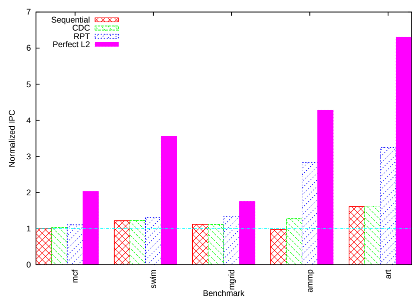

Figure 3: Prefetching on uniprocessors with limited bandwidth to memory. IPC is normalized to the case where no prefetching is performed.

4 CMP results

In figure 4, we run the same prefetching heuristics, but on a 4-way CMP running 4 different SPEC2000 benchmarks.

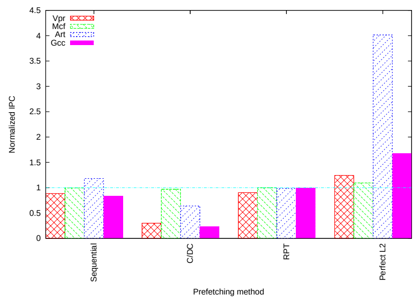

Figure 4: Prefetching on a 4-way CMP. IPC is normalized to the case where no prefetching is performed.

5 Conclusion

It is clear that CMPs offer ample oppertunity for prefetching, but it requires new heuristics. While prefetching might benefit one core, it might seriously degrade performance for another.

5.1 Future work

This work will be continued as part of my PhD thesis, and is a work in progress.

- SimpleScalar cannot execute true parallel programs.
- Switch to the *M5 simulator*.
- *Cache partitioning*
- Study the interaction between the *cache coherence protocol* and prefetching
- More reference prefetching algorithms

Acknowledgments

I would like to thank Lasse Natvig for guidance and support. Haakon Dybdahl for sharing his ideas and improvements to the simulator. Hanne Lian for her support and patience.

References

[1] L. Spracklen and S. G. Abraham, “Chip multithreading: Opportunities and challenges,” in *11th International Symposium on High-Performance Computer Architecture (HPCA’05)*, pp. 248–252, 2005.

[2] L. Spracklen, Y. Chou, and S. G. Abraham, “Effective instruction prefetching in chip multiprocessors for modern commercial applications,” in *11th International Symposium on High-Performance Computer Architecture*, pp. 225–236, 2005.

Appendix C

Performance Counter Code

C.1 Pmc.c

Listing C.1: Pmc.c

```
/*
 *   pcm.c
 *
 *   This linux kernel module enables RDPMC for user level programs.
 *   In addition, it sets up counter #0 to count the number of cache misses.
 *
 *   Written by Marius Grannas 2006
 *
 * NOTE! This is AMD Athlon XP specific code, and will possibly break things
 * badly on any other arch.
 *
 */

#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

/* This function is called when the module is loaded */
int init_module(void)
{
  printk(KERN_INFO "Enabling␣RDPCM...\n");

  // The following is a reimplementation of set_in_cr4() that doesn't
  // require mmu_cr4_features which isn't exported anymore.

  __asm__ __volatile__("movl␣%%cr4,%%eax\n\t"
                       "orl␣%0,%%eax\n\t"
                       "movl␣%%eax,%%cr4\n"
                       : : "irg" (X86_CR4_PCE)
                       :"ax");

  printk(KERN_INFO "Setting␣up␣counter␣0.\n");
  // This part sets up counter 0 to count the number of data cache misses.
  // See the Athlon Optimization guidelines for values
  __asm__ __volatile__ ("mov␣␣$0xC0010000,␣%%ecx␣;␣mov␣$0x00420041,␣%%eax;␣
      wrmsr"
                       :
                         :
                         : "eax", "ecx");

  // A non 0 return means init_module failed; module can't be loaded.
  return 0;
}


/* This function is called when the module is unloaded */

void cleanup_module(void)
{
  printk(KERN_INFO "Disabling␣Performance␣counters.\n");
  // Same as above
  __asm__ __volatile__("movl␣%%cr4,%%eax\n\t"
                         "andl␣%0,%%eax\n\t"
                         "movl␣%%eax,%%cr4\n"
                         : : "irg" (~X86_CR4_PCE)
                         :"ax");

}
```

C.2 Makefile

Listing C.2: Makefile

```
# This makefile creates the kernel module from source
# Created by Marius Grannaes 2006
obj-m += pmc.o


all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

C.3 Performance.c

Listing C.3: Performance.c

```
/*
 * performance.c
 *
 * The purpose of this small program is to show how userspace can access
 * performance registers. This code is pretty useless unless RDPMC is
     enabled
 * for userspace. See the kernel module for additional details.
 *
 * Written by Marius Grannas 2006
 *
 * Note: Some of this code is athlon XP specific and might break on other
 * architectures.
 *
 * Compile with
 * gcc performance.c -o performance
 *
 * Note: Do not turn on optimizing, the compiler might optimize away the
     entire
 * loop body!
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * This function reads the value of the built in clock cycle timer
 * The cpuid instruction is there to ensure serializability (eg it forces
 * a pipeline flush
 */
static inline unsigned long long rdtsctime() {
    unsigned int eax, edx;
    unsigned long long val;
    __asm__ __volatile__("cpuid": : : "ax", "bx", "cx", "dx");
    __asm__ __volatile__("rdtsc":"=a"(eax), "=d"(edx));
    val = edx;
    val = val << 32;
    val += eax;
    return val;
}

/*
 * This function reads the value of performance register 0.
 * Cpuid instruction is again used for serializability
 */
static inline unsigned long long readpc() {
  unsigned int eax, edx;
  unsigned long long val;
  __asm__ __volatile__("cpuid": : : "ax", "bx", "cx", "dx");
  __asm__ __volatile__("xor %%ecx, %%ecx ; rdpmc"
                        :"=a"(eax), "=d"(edx) /* Output */
                        :                     /* Input   */
                        :"ecx"                /* Clobbered */
                        );
  val = edx;
  val = val << 32;
  val += eax;
  return val;
}

/*
 * Sample test program.
 * This shows how the counter works as well as the timer.
 */

int main(void) {
  long long int start,stop;
  int a = 9;
  int c = 3;
  int d[90000];
  int i;

  printf("Clock is %lld\n", rdtsctime());
  start = readpc();

  for (i=0; i< 1000; i++) {
        d[i] = a / c;
  }

  stop = readpc();
  printf("Number of misses: %lld\n", stop-start);
  printf("Clock is %lld\n", rdtsctime());
  return 0;
}
```

Appendix D

Python Scripts

D.1 Clustisrunbench.py

Listing D.1: Clustisrunbench.py

```
#!/usr/bin/python
import sys
import time
import popen2
import config

header = """#!/bin/bash
#PBS␣-N␣sim-out
#PBS␣-l␣walltime=8:00:00
#PBS␣-l␣nodes=1:ppn=1
#PBS␣-m␣bea
#PBS␣-q␣default
#
"""

count = 0

for benchmark in config.benchmarks:
  for configuration in config.configurations:
    output = open(config.temppbs, 'w')
    simulation = config.simulator + '␣' + config.commonconfig + '␣' +
        configuration[1] + '␣'+ config.specbinpath + benchmark[1] + '␣' +
        benchmark[3]
    count = count + 1
    output.write(header)
    # Change directory into the simulation directory
    output.write('cd␣' + config.specdatapath + benchmark[2] +'\n');
    output.write(simulation)
    output.close()
    results = popen2.popen3('qsub␣' + config.temppbs)
    print results[0].readline(),

print 'Number␣of␣submitted␣jobs:',
print count
```

D.2 Config.py

Listing D.2: Sample configuration file

```
#!/usr/bin/python

#Path to binaries

#Full path to executable
simulator = '/home/grannas/hovedoppgave/simplesim-3.0/sim-outorder'
# Full path to simulation executables
specbinpath = '/home/grannas/spec2000binaries/'
# Where you want to store the temporary pbs file
temppbs = 'sim-outorder.pbs'
# Where the data files to the spec benchmarks are
specdatapath = '/home/grannas/SPEC_2000_REDUCED/'

# Common configuration across benchmarks.
commonconfig = '-cache:dl1␣dl1:32:64:4:l␣-cache:dl1lat␣2␣-cache:il1␣il1
    :32:64:4:l␣-cache:il1lat␣2␣-cache:dl2␣ul2:512:128:8:l␣-cache:dl2lat␣7␣␣-
    cache:il2␣dl2␣-cache:il2lat␣7␣-mem:width␣8␣␣-mem:lat␣160␣2␣-fetch:mplat␣
    15␣-ruu:size␣16␣-lsq:size␣8␣␣-tlb:itlb␣itlb:1:4096:128:l␣-tlb:dtlb␣dtlb
    :1:4096:128:l␣␣-bpred␣2lev␣-bpred:bimod␣4096␣-bpred:2lev␣␣1␣1024␣10␣0␣␣␣-
    bpred:comb␣4096␣-dram:block␣128␣-dram:page␣8␣-dram:comm␣40␣-dram:core␣40␣
    -dram:data␣80␣-prefetch:table␣1024␣-dram:chan␣1'

# Experiment name, used to generate plotfiles
experimentname = 'baseline-chan1'

# Parse out this statistic
#grepname = 'sim_IPC'
#grepname = 'dram.accesses'
#grepname = 'ul2.prefetches_ok'
grepname = 'ul2.misses'

# Configuration
# Format : (Name, Parameters)
configurations = [('None', '-prefetch␣none'), \
                  ('Sequ','-prefetch␣sequential'), \
                  ('DC','-prefetch␣DC'), \
                  ('CDC','-prefetch␣CDC'), \
                  ('RPT','-prefetch␣RPT␣-prefetch:loc␣DL1␣-prefetch:type␣
                      access'), \
                  ('Stream','-prefetch␣stream␣-prefetch:loc␣DL1␣-prefetch:
                      type␣access'), \
                  ('AVD','-prefetch␣AVD'), \
                  ('Perfect␣L2', '-perfect:l2␣1')]



# Different benchmarks
# Format : (Name, binary, working dir, parameters)
benchmarks = [ ('gzip' , 'gzip00.peak.ev6', '164.gzip/input/', 'lgred.log␣1'
    ),\
('gcc' ,'gcc00.peak.ev6','176.gcc/input/','lgred.cp-decl.i␣-o␣lgred.cp-decl.
    s'),\
('crafty','crafty00.peak.ev6', '186.crafty/input/lgred/','<␣lgred.in'),\
('mcf','mcf00.peak.ev6', '181.mcf/input/', 'lgred.in'),\
('swim','swim00.peak.ev6', '171.swim/input/lgred/', '<␣swim.in'),\
('mgrid','mgrid00.peak.ev6','172.mgrid/input/lgred/', '<␣mgrid.in'),\
('equake','equake00.peak.ev6','183.equake/input/' ,'<␣lgred/lgred.in'),\
('applu','applu00.peak.ev6','173.applu/input/lgred', '<␣applu.in'),\
('vpr','vpr00.peak.ev6','175.vpr/input/', 'lgred.net␣small.arch.in␣tull1␣
    tull2␣-nodisp␣-place_only␣-init_t␣5␣-exit_t␣0.005␣-alpha_t␣0.9412␣-
    inner_num␣2'),\
('ammp','ammp00.peak.ev6','188.ammp/','<␣./input/lgred.in'),\
('mesa','mesa00.peak.ev6','177.mesa/input/','-frames␣1␣-meshfile␣lgred.in␣-
    ppmfile␣utfil'),\
('galgel','galgel00.peak.ev6','178.galgel/input/lgred/','<␣lgred.in'),\
('lucas','lucas00.peak.ev6','189.lucas/input/lgred/','<␣lgred.in'),\
('fma','fma3d00.peak.ev6','191.fma3d/input/lgred/',''),\
('parser','parser00.peak.ev6','197.parser/input', '2.1.dict␣-batch␣<␣lgred.
    in'),\
('eon','eon00.peak.ev6','252.eon/input/lgred', 'chair.control.kajiya␣chair.
    camera␣chair.surfaces␣chair.kajiya.ppm␣ppm␣pixels_out.kajiya'),\
('perlbmk','perlbmk00.peak.ev6','253.perlbmk/input/lgred','-I.␣-I./lib␣lgred
    .makerand.pl'),\
('gap','gap00.peak.ev6','254.gap/input/lgred','-l␣.␣-q␣-m␣64M␣<␣lgred.in'),\
('bzip2' , 'bzip200.peak.ev6','256.bzip2/input','lgred.source␣1'),\
('apsi' , 'apsi00.peak.ev6', '301.apsi/input/lgred',''),\
('wupwise' , 'wupwise00.peak.ev6', '168.wupwise/input/lgred',''),\
('twolf' , 'twolf00.peak.ev6', '300.twolf/input/','lgred/lgred'),\
('facerec','facerec00.peak.ev6','187.facerec/input/lgred','␣<␣lgred.in'),\
('art','art00.peak.ev6','179.art/input/','-scanfile␣c756hel.in␣-trainfile1␣
    a10.img␣-stride␣5␣-startx␣134␣-starty␣220␣-endx␣184␣-endy␣240␣-objects␣1'
    )\
]

# Ikke fungerende :
# Sixtrack : Manglende filer? Filstoerrelse er ihvertfall 0.
# Vortex: Korrupt fil?
#('sixtrack','sixtrack00.peak.ev6','200.sixtrack/input/lgred', '<␣inp.in'),\
#('vortex','vortex00.peak.ev6','255.vortex/input','lgred.raw'),\
```

D.3 Parsebench.py

Listing D.3: Parsebench.py

```
#!/usr/bin/python
import sys
import time
import popen2
import config
import os

print '#␣This␣is␣an␣autogenerated␣plotfile␣for␣gnuplot'
print '#␣Configuration␣used:␣'
print '#␣' + config.commonconfig
for i in config.configurations:
  print '#' + i[0] + ':␣' + i[1]

sys.stdout.write('#Benchmark\t')

for i in config.configurations:
  sys.stdout.write(i[0])
  sys.stdout.write('\t')

sys.stdout.write('\n')

# Iterate trough different configurations and grep for important information
    .

count = 1

runlog = open('run.log', 'r')

for benchmark in config.benchmarks:
  print benchmark[0] + '\t',
  print count,
  print '\t',
  for configuration in config.configurations:
    logfile = runlog.readline()
    logfile = logfile.split('.')
    try:
      datafile = open('sim-out.e'+logfile[0],'r')
    except IOError:
      print 'N/C\t',
      continue
    sim_results = datafile.readlines()
    flag = 0
    for line in sim_results:
      if line.startswith(config.grepname):
        ipc = line.split()[1]
        flag = 1
        print ipc + '\t',
        break
    if flag == 0:
      print 'ERROR\t',
  count = count + 1
  print '\n',
```

Appendix E

Uniprocessor Code

E.1 Makefile

Listing E.1: Makefile - Unified diff against SimpleScalar 3.0d

```
--- ../simplesim-3.0-orig/Makefile        2003-10-09 04:42:59.000000000 +0200
+++ ../simplesim-3.0/Makefile    2006-05-25 23:42:04.000000000 +0200
@@ -77,8 +77,8 @@
 ##      RS/6000 AIX Unix version 4, GNU GCC version cygnus-2.7-96q4
 ##      Windows NT version 4.0, Cygnus CygWin/32 beta 19
 ##
-CC = gcc
-OFLAGS = -O0 -g -Wall
+CC = gcc-3.4
+OFLAGS = -g
 MFLAGS = `./sysprobe -flags`
 MLIBS  = `./sysprobe -libs` -lm
 ENDIAN = `./sysprobe -s`
@@ -277,7 +277,7 @@
 #
 # all the sources
 #
-SRCS = main.c sim-fast.c sim-safe.c sim-cache.c sim-profile.c \
+SRCS = main.c dram.c prefetch.c sim-fast.c sim-safe.c sim-cache.c sim-
    profile.c \
        sim-eio.c sim-bpred.c sim-cheetah.c sim-outorder.c \
        memory.c regs.c cache.c bpred.c ptrace.c eventq.c \
        resource.c endian.c dlite.c symbol.c eval.c options.c range.c \
@@ -287,7 +287,7 @@
        target-alpha/alpha.c target-alpha/loader.c target-alpha/syscall.c \
        target-alpha/symbol.c

-HDRS = syscall.h memory.h regs.h sim.h loader.h cache.h bpred.h ptrace.h \
+HDRS = dram.h syscall.h memory.h regs.h sim.h loader.h cache.h bpred.h
    ptrace.h \
        eventq.h resource.h endian.h dlite.h symbol.h eval.h bitmap.h \
        eio.h range.h version.h endian.h misc.h \
        target-pisa/pisa.h target-pisa/pisabig.h target-pisa/pisalittle.h \
@@ -305,9 +305,7 @@
 #
 # programs to build
 #
-PROGS = sim-fast$(EEXT) sim-safe$(EEXT) sim-eio$(EEXT) \
-        sim-bpred$(EEXT) sim-profile$(EEXT) \
-        sim-cache$(EEXT) sim-outorder$(EEXT) # sim-cheetah$(EEXT)
+PROGS = sim-outorder$(EEXT)

 #
 # all targets, NOTE: library ordering is important...
@@ -390,8 +388,8 @@
 sim-cache$(EEXT):       sysprobe$(EEXT) sim-cache.$(OEXT) cache.$(OEXT) $(
     OBJS) libexo/libexo.$(LEXT)
        $(CC) -o sim-cache$(EEXT) $(CFLAGS) sim-cache.$(OEXT) cache.$(OEXT)
            $(OBJS) libexo/libexo.$(LEXT) $(MLIBS)

-sim-outorder$(EEXT):    sysprobe$(EEXT) sim-outorder.$(OEXT) cache.$(OEXT)
    bpred.$(OEXT) resource.$(OEXT) ptrace.$(OEXT) $(OBJS) libexo/libexo.$(
    LEXT)
-       $(CC) -o sim-outorder$(EEXT) $(CFLAGS) sim-outorder.$(OEXT) cache.$(
    OEXT) bpred.$(OEXT) resource.$(OEXT) ptrace.$(OEXT) $(OBJS) libexo/libexo
    .$(LEXT) $(MLIBS)
+sim-outorder$(EEXT):    sysprobe$(EEXT) sim-outorder.$(OEXT) dram.$(OEXT)
    cache.$(OEXT) prefetch.$(OEXT) bpred.$(OEXT) resource.$(OEXT) ptrace.$(
    OEXT) $(OBJS) libexo/libexo.$(LEXT)
+       $(CC) -o sim-outorder$(EEXT) $(CFLAGS) sim-outorder.$(OEXT) cache.$(
    OEXT) dram.$(OEXT) prefetch.$(OEXT) bpred.$(OEXT) resource.$(OEXT) ptrace
    .$(OEXT) $(OBJS) libexo/libexo.$(LEXT) $(MLIBS)

 exo libexo/libexo.$(LEXT): sysprobe$(EEXT)
        cd libexo $(CS) \
@@ -499,6 +497,8 @@
 regs.$(OEXT): options.h stats.h eval.h
 cache.$(OEXT): host.h misc.h machine.h machine.def cache.h memory.h options
     .h
 cache.$(OEXT): stats.h eval.h
+prefetch.$(OEXT): host.h misc.h machine.h machine.def cache.h memory.h
    options.h
+prefetch.$(OEXT): stats.h eval.h cache.h
 bpred.$(OEXT): host.h misc.h machine.h machine.def bpred.h stats.h eval.h
 ptrace.$(OEXT): host.h misc.h machine.h machine.def range.h ptrace.h
 eventq.$(OEXT): host.h misc.h machine.h machine.def eventq.h bitmap.h
```

E.2 Sim-outorder.c

Listing E.2: Sim-outorder.c - Unified diff against SimpleScalar 3.0d

```
--- ../simplesim-3.0-orig/sim-outorder.c         2003-10-09
    03:57:25.000000000 +0200
+++ ../simplesim-3.0/sim-outorder.c       2006-05-31 13:32:54.000000000 +0200
@@ -72,6 +72,8 @@
 #include "ptrace.h"
 #include "dlite.h"
 #include "sim.h"
+#include "dram.h"
+#include "prefetch.h"

 /*
  * This file implements a very detailed out-of-order issue superscalar
@@ -91,6 +93,73 @@
  * simulator options
  */

+/* Prefetch options */
+
+/* prefetching type {none|sequential....} */
+static char *prefetch_type;
+
+/* prefetching location {DL1|DL2|IL1|IL2} */
+static char *prefetch_location;
+
+/* prefetching degree */
+static int prefetch_degree_in;
+
+/* prefetching trigger type (Cache miss, cache hit etc) */
+static char *prefetch_target;
+
+/* CZone size (in bits */
+static int czone_size_in;
+
+/* GHB size (in entries) */
+static int table_size_in;
+
+/* Bandwidth-aware prefetching - Treshold */
+int bwa_treshold;
+
+/* Bandwidth-aware prefetching - Enable */
+int bwa_enable;
+
+/* number of channels between cache and DRAM */
+static unsigned int max_dram_chan;
+
+/*
+ * These options are added by Marius Grannaes
+ * The define the DRAM used.
+ * If this model is used the latency specified by -mem:lat is not used.
+ */
+
+/* Dram structure */
+dram_system_t *dram_system;
```

```
+
+/* The number of availabel DRAM channels */
+int num_channels;
+
+/* Block size in DRAM */
+int block_size;
+
+/* Page size in DRAM (in blocks) */
+int page_size;
+
+/* Time to execute command transfer (in clock cycles) */
+int control_time;
+
+/* Time to transfer data from the core to the latches */
+int core_time;
+
+/* Time to transfer data from the latches to the memory controller */
+int data_time;
+
+/* How often to log DRAM trace */
+int dram_trace_interval;
+
+/* Perfect l2 */
+int perfect_l2 = 0;
+
+/*
+ * Original parameters
+ * (not altered)
+ */
+
 /* maximum number of inst's to execute */
 static unsigned int max_insts;

@@ -433,14 +502,19 @@
               tick_t now)               /* time of access */
 {
   unsigned int lat;
-
+  process_prefetch_trigger(Cache_Miss,Cache_DL1,baddr,sim_cycle);
   if (cache_dl2)
     {
-      /* access next level of data cache hierarchy */
-      lat = cache_access(cache_dl2, cmd, baddr, NULL, bsize,
-                         /* now */now, /* pudata */NULL, /* repl addr */NULL
   );
+      if (perfect_l2) {
+        lat = cache_dl2->hit_latency;
+      } else {
+        /* access next level of data cache hierarchy */
+        lat = cache_access(cache_dl2, cmd, baddr, NULL, bsize,
+                           /* now */now, /* pudata */NULL, /* repl addr */
   NULL);
+      }
+      process_prefetch_trigger(Memory_Access, Cache_DL2,baddr,sim_cycle);
       if (cmd == Read)
-        return lat;
```

```
+               return lat;
        else
         {
           /* FIXME: unlimited write buffers */
@@ -450,8 +524,17 @@
   else
     {
       /* access main memory */
-      if (cmd == Read)
-       return mem_access_latency(bsize);
+      if (cmd == Read) {
+        /* If the dram system is defined using 0 channels, then fallback
+         * to old model */
+        if (dram_system->num_channels == 0) {
+               lat = mem_access_latency(bsize);
+        } else {
+          lat = access_dram(dram_system,baddr,bsize,now);
+        }
+        process_prefetch_trigger(Memory_Access,DRAM,baddr,sim_cycle);
+        return lat;
+      }
       else
        {
          /* FIXME: unlimited write buffers */
@@ -468,14 +551,26 @@
               struct cache_blk_t *blk,  /* ptr to block in upper level */
               tick_t now)               /* time of access */
 {
+  int latency;
   /* this is a miss to the lowest level, so access main memory */
-  if (cmd == Read)
-    return mem_access_latency(bsize);
+  if (cmd == Read) {
+    /* If the dram system is defined using 0 channels, then fallback
+     * to old model */
+    if (dram_system->num_channels == 0) {
+          latency = mem_access_latency(bsize);
+        } else {
+          latency = access_dram(dram_system,baddr,bsize,now);
+        }
+      }
   else
     {
       /* FIXME: unlimited write buffers */
-      return 0;
+      latency = 0;
     }
+  set_return_latency(latency);
+  process_prefetch_trigger(Cache_Miss, Cache_DL2,baddr, sim_cycle);
+  process_prefetch_trigger(Memory_Access, DRAM,baddr, sim_cycle);
+  return latency;
 }

 /* l1 inst cache l1 block miss handler function */
@@ -491,21 +586,34 @@
 if (cache_il2)
```

```
      {
        /* access next level of inst cache hierarchy */
-       lat = cache_access(cache_il2, cmd, baddr, NULL, bsize,
+       if (perfect_l2) {
+         lat = cache_il2->hit_latency;
+       } else {
+         lat = cache_access(cache_il2, cmd, baddr, NULL, bsize,
                         /* now */now, /* pudata */NULL, /* repl addr */NULL
                            );
-       if (cmd == Read)
-        return lat;
-       else
-        panic("writes␣to␣instruction␣memory␣not␣supported");
+       }
+       if (cmd != Read) {
+               panic("writes␣to␣instruction␣memory␣not␣supported");
+       }
+       process_prefetch_trigger(Memory_Access, Cache_IL2,baddr, sim_cycle);
      }
    else
      {
-       /* access main memory */
-       if (cmd == Read)
-        return mem_access_latency(bsize);
-       else
-        panic("writes␣to␣instruction␣memory␣not␣supported");
+     /* access main memory */
+     if (cmd == Read) {
+     /* If the dram system is defined using 0 channels, then fallback
+      * to old model */
+     if (dram_system->num_channels == 0) {
+           lat = mem_access_latency(bsize);
+         } else {
+           lat = access_dram(dram_system,baddr,bsize,now);
+         }
+    process_prefetch_trigger(Memory_Access, DRAM, baddr, sim_cycle);
+    } else {
+             panic("writes␣to␣instruction␣memory␣not␣supported");
+       }
      }
+  return lat;
 }

 /* l2 inst cache block miss handler function */
@@ -516,14 +624,27 @@
              struct cache_blk_t *blk,   /* ptr to block in upper level */
              tick_t now)                /* time of access */
 {
+  int latency;
   /* this is a miss to the lowest level, so access main memory */
-  if (cmd == Read)
-    return mem_access_latency(bsize);
-  else
-    panic("writes␣to␣instruction␣memory␣not␣supported");
+    if (cmd == Read) {
+    /* If the dram system is defined using 0 channels, then fallback
+        * to old model */
+      if (dram_system->num_channels == 0) {
+             latency = mem_access_latency(bsize);
+          } else {
+             latency = access_dram(dram_system,baddr,bsize,now);
+          }
+      process_prefetch_trigger(Memory_Access, DRAM, baddr,sim_cycle);
+      }
+      else {
+        panic("writes␣to␣instruction␣memory␣not␣supported");
+      }
+      process_prefetch_trigger(Cache_Miss, Cache_IL2, baddr, sim_cycle);
+      return latency;
 }



+
 /*
  * TLB miss handlers
  */
@@ -580,14 +701,83 @@
 "latency␣of␣all␣pipeline␣operations.\n"
                 );

-   /* instruction limit */
+   /* New DRAM-options */
+
+   opt_reg_uint(odb, "-dram:chan", "number␣of␣DRAM␣channels",
+       &num_channels, /* default - disabled*/0,
+       /* print */TRUE, /* format */NULL);
+
+   /* TODO: Doublecheck these values! */
+
+   opt_reg_uint(odb, "-dram:block", "size␣of␣a␣block␣(in␣bytes)",
+       &block_size, /* default (same as L2 defaults)*/128,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_uint(odb, "-dram:page", "size␣of␣each␣page␣(in␣blocks)",
+       &page_size, /* default */8,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_uint(odb, "-dram:comm", "latency␣of␣command␣transfer",
+       &control_time, /*default*/40,
+       /* print */TRUE, /*format */NULL);
+
+   opt_reg_uint(odb, "-dram:core", "latency␣of␣DRAM␣core",
+       &core_time, /* default */40,
+       /* print */TRUE, /*format */NULL);
+
+   opt_reg_uint(odb, "-dram:data", "latency␣of␣DRAM␣transfer",
+       &data_time, /* default */80,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_uint(odb, "-dram:trace", "Number␣of␣cycles␣between␣each␣sample",
+       &dram_trace_interval, /* default */0,
+       /* print */TRUE, /* format */NULL);
```

```
+
+   /* prefetch options */
+
+   opt_reg_string(odb, "-prefetch",
+       "prefetching␣type␣{none|sequential|DC|CDC|RPT|Stream|AVD}",
+       &prefetch_type, /* default */"none",
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_string(odb, "-prefetch:type",
+       "prefetching␣trigger␣type␣{miss,␣hit,␣access}",
+       &prefetch_target, /* default */"miss",
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_string(odb, "-prefetch:loc",
+       "prefetching␣location␣{DL1|DL2|IL1|IL2}",
+       &prefetch_location, /* default */ "DL2",
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-prefetch:degree", "prefetch␣degree",
+         &prefetch_degree_in, /* default */ 1,
+         /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-prefetch:czone", "size␣of␣each␣CZone␣in␣bits",
+       &czone_size_in, /* default */ 16,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-prefetch:table","size␣of␣the␣prefetching␣table␣in␣
    entries",
+       &table_size_in, /*default */ 1024,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-perfect:l2", "Is␣the␣l2␣perfect?",
+       &perfect_l2, /* default */0,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-bwa", "Enables␣Bandwidth␣Aware␣Prefetching",
+       &bwa_enable, /* default */0,
+       /* print */TRUE, /* format */NULL);
+
+   opt_reg_int(odb, "-bwa:tresh", "Bandwidth␣Aware␣Prefetching␣treshold",
+       &bwa_treshold, /* default */0,
+       /* print */TRUE, /* format */NULL);

    opt_reg_uint(odb, "-max:inst", "maximum␣number␣of␣inst's␣to␣execute",
                 &max_insts, /* default */0,
                 /* print */TRUE, /* format */NULL);

-   /* trace options */
-
    opt_reg_int(odb, "-fastfwd", "number␣of␣insts␣skipped␣before␣timing␣
        starts",
                &fastfwd_count, /* default */0,
                /* print */TRUE, /* format */NULL);
@@ -894,6 +1084,91 @@
    if (fetch_speed < 1)
      fatal("front-end␣speed␣must␣be␣positive␣and␣non-zero");
```

```
+   if (!mystricmp(prefetch_type, "sequential"))
+     {
+       /* Sequential prefetch */
+       register_prefetch_algorithm(&sequential_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "DC"))
+     {
+       /* Delta correlation */
+       register_prefetch_algorithm(&delta_correlation_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "none"))
+     {
+       /* No prefetching */
+       register_prefetch_algorithm(&no_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "CDC"))
+     {
+       /* CZone/Delta Correlation */
+       register_prefetch_algorithm(&czone_delta_correlation_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "RPT"))
+     {
+       /* RPT table */
+       register_prefetch_algorithm(&rpt_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "AVD"))
+     {
+       /* AVD prefetching */
+       register_prefetch_algorithm(&avd_prefetch);
+     }
+   else if (!mystricmp(prefetch_type, "stream"))
+     {
+       /* Stream prefetching */
+       register_prefetch_algorithm(&stream_prefetch);
+     }
+   else
+   {
+     fatal("cannot␣parse␣prefetch␣type␣'%s'", prefetch_type);
+   }
+   if (!mystricmp(prefetch_target, "miss"))
+   {
+     /* Prefetch on miss */
+     register_prefetch_target(Cache_Miss);
+   }
+   else if (!mystricmp(prefetch_target, "hit"))
+   {
+     /* Prefetch on hit */
+     register_prefetch_target(Cache_Hit);
+   }
+   else if (!mystricmp(prefetch_target, "access"))
+   {
+     /* Prefetch on access */
+     register_prefetch_target(Memory_Access);
+   }
+   else
+  {
+    fatal("cannot␣parse␣prefetch␣trigger␣type␣‘%s’", prefetch_target);
+  }
+  if (!mystricmp(prefetch_location, "dl1"))
+  {
+    /* Cache DL1 */
+    register_prefetch_location(Cache_DL1);
+  }
+  else if (!mystricmp(prefetch_location, "dl2"))
+  {
+    /* Cache DL2 */
+    register_prefetch_location(Cache_DL2);
+  }
+  else if (!mystricmp(prefetch_location, "il1"))
+  {
+    /* Cache IL1 */
+    register_prefetch_location(Cache_IL1);
+  }
+  else if (!mystricmp(prefetch_location, "il2"))
+  {
+    /* Cache IL2 */
+    register_prefetch_location(Cache_IL2);
+  }
+  else
+  {
+    fatal("cannot␣parse␣prefetch␣location␣‘%s’", prefetch_location);
+  }
+
+  register_prefetch_degree(prefetch_degree_in);
+
   if (!mystricmp(pred_type, "perfect"))
     {
       /* perfect predictor */
@@ -1292,6 +1567,11 @@
   stat_reg_formula(sdb, "avg_sim_slip",
                    "the␣average␣slip␣between␣issue␣and␣retirement",
                    "sim_slip␣/␣sim_num_insn", NULL);
+
+  /* register DRAM stats */
+  if (dram_system->num_channels !=0) {
+    dram_reg_stats(dram_system,sdb);
+  }

   /* register predictor stats */
   if (pred)
@@ -1368,6 +1648,9 @@
 {
   sim_num_refs = 0;

+  /* create the memory hierachy */
+  dram_system = create_dram(num_channels, block_size, page_size,
    control_time, core_time, data_time, dram_trace_interval);
+
   /* allocate and initialize register file */
   regs_init(&regs);
```

```
@@ -1434,6 +1717,7 @@
   readyq_init();
   ruu_init();
   lsq_init();
+  prefetch_init(cache_il1, cache_il2, cache_dl1, cache_dl2, czone_size_in,
    table_size_in, mem);

   /* initialize the DLite debugger */
   dlite_init(simoo_reg_obj, simoo_mem_obj, simoo_mstate_obj);
@@ -2184,10 +2468,12 @@
                  /* go to the data cache */
                  if (cache_dl1)
                    {
+          set_last_PC_value(rs->PC);
                      /* commit store value to D-cache */
                      lat =
                        cache_access(cache_dl1, Write, (LSQ[LSQ_head].addr
                            &~3),
                                     NULL, 4, sim_cycle, NULL, NULL);
+          process_prefetch_trigger(Memory_Access, Cache_DL1, (rs->addr &
    ~3),sim_cycle);
                      if (lat > cache_dl1_lat)
                        events |= PEV_CACHEMISS;
                    }
@@ -2730,10 +3016,13 @@
                                if (cache_dl1 && valid_addr)
                                  {
                                    /* access the cache if non-faulting */
+          /* Set PC value */
+          set_last_PC_value(rs->PC);
                                    load_lat =
                                      cache_access(cache_dl1, Read,
                                                   (rs->addr & ~3), NULL, 4,
                                                   sim_cycle, NULL, NULL);
+          process_prefetch_trigger(Memory_Access, Cache_DL1, (rs->addr &
    ~3),sim_cycle);
                                    if (load_lat > cache_dl1_lat)
                                      events |= PEV_CACHEMISS;
                                  }
@@ -4230,10 +4519,15 @@
          if (cache_il1)
            {
              /* access the I-cache */
+        /* Set prefetching PC value to -1 to signify that this is not a
+         * data request going through the hierarchy.
+         */
+        set_last_PC_value(-1);
              lat =
                cache_access(cache_il1, Read, IACOMPRESS(fetch_regs_PC),
                             NULL, ISCOMPRESS(sizeof(md_inst_t)), sim_cycle,
                             NULL, NULL);
+        process_prefetch_trigger(Memory_Access, Cache_IL1, IACOMPRESS(
    fetch_regs_PC), sim_cycle);
              if (lat > cache_il1_lat)
                last_inst_missed = TRUE;
            }
```

```
@@ -4428,7 +4722,7 @@
   /* ignore any floating point exceptions, they may occur on mis-speculated
      execution paths */
   signal(SIGFPE, SIG_IGN);
-
+
   /* set up program entry state */
   regs.regs_PC = ld_prog_entry;
   regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);
@@ -4595,6 +4889,9 @@
        RUU_fcount += ((RUU_num == RUU_size) ? 1 : 0);
        LSQ_count += LSQ_num;
        LSQ_fcount += ((LSQ_num == LSQ_size) ? 1 : 0);
+
+       /* Dram trace */
+       dram_trace(dram_system, sim_cycle);

        /* go to next cycle */
        sim_cycle++;
```

E.3 Dram.h

Listing E.3: Dram.h

```
/* This code describes the memory model for DRAM
 *
 * It includes the following properties
 * - Virtual channels (for parallelisation)
 * - Pipelining
 * - Open pages
 *
 * This file was written by Marius Grannaes in 2006.
 */

#ifndef DRAM_H
#define DRAM_H

#include <stdio.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "memory.h"
#include "stats.h"


/* This is the size of the circular buffer containing
   the occupancy of the memory channel */
#define CIRC_BUFFER_SIZE 3

/* This datastructure defines the DRAM system */

typedef struct {
  int num_channels;          /* Number of channels */
  int block_size;            /* Size of each block, usally equal
                              * to l2 block size (in bytes) */
  int page_size;             /* Number of blocks in a page */
  int control_time;          /* Time to transfer data */
  int core_time;             /* Time to transfer from core to
                              * latches */
  int data_time;             /* Time to transfer from latches to
                              * memory controller */
  tick_t *ready_channels;    /* When are the channels ready? */
  tick_t *circbuffer;        /* Circular buffer */

  int trace_interval;        /* Sample interval when tracing */
  md_addr_t *last_address;   /* Last page accessed */

  counter_t accesses;        /* Number of accesses to the memory
                              * system */
  counter_t latency;         /* Total latency imposed by system */
  counter_t stalls;          /* Total number of stalls due to busy system */
  counter_t page_hits;       /* Total number of times accessing an open page
      */
} dram_system_t;

/* Function prototypes - See dram.c for help */
```

```
dram_system_t *create_dram(int number_of_channels, int size_of_block, int
    page_size, int control_time, int core_time, int data_time, int
    trace_interval);

void free_dram(dram_system_t *dram_system);

int block_to_bank(dram_system_t *dram_system, md_addr_t block);

int is_same_page(dram_system_t *dram_system, md_addr_t block1, md_addr_t
    block2);

unsigned int access_dram(dram_system_t *dram_system, md_addr_t block, int
    bsize, tick_t now);

unsigned int get_channels_in_use(dram_system_t *dram_system, tick_t now);

void dram_reg_stats(dram_system_t *dram_system, struct stat_sdb_t *sdb);

void dram_trace(dram_system_t *dram_system, tick_t now);

int get_occupancy(void);

#endif
```

E.4 Dram.c

Listing E.4: Dram.c

```
/* This code describes the memory model for DRAM
 *
 * It includes the following properties
 * - Virtual channels (for parallelisation)
 * - Pipelining
 * - Open pages
 *
 * This file was written by Marius Grannaes in 2006.
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "dram.h"

/* Global circular buffer pointer */

int *circbuffer;

/* This function creates the dram subsystem */

dram_system_t *create_dram(int number_of_channels, int size_of_block, int
    page_size, int control_time, int core_time, int data_time, int
    trace_interval) {
  dram_system_t *dram_system;

  /* Allocate memory for the structure */

  dram_system = calloc(1, sizeof(dram_system_t));
  if (dram_system == 0) {
    printf("Could␣not␣allocate␣memory␣for␣DRAM␣model.\n");
    exit(1);
  }

  /* Set the values given as parameters */
  dram_system->num_channels = number_of_channels;
  dram_system->block_size = size_of_block;
  dram_system->page_size = page_size;
  dram_system->control_time = control_time;
  dram_system->core_time = core_time;
  dram_system->data_time = data_time;
  dram_system->trace_interval = trace_interval;

  /* Reset statistics */
  dram_system->accesses = 0;
  dram_system->latency = 0;
  dram_system->stalls = 0;
  dram_system->page_hits = 0;
  /* Create the arrays */
  /* If the number of channels is 0 then do not create anything */

  if (number_of_channels > 0) {
    dram_system->ready_channels = calloc(number_of_channels, sizeof(tick_t))
        ;
    if (dram_system->ready_channels == 0) {
      printf("Error␣creating␣ready␣channels.\n");
      exit(1);
    }
    dram_system->last_address = calloc(number_of_channels, sizeof(md_addr_t)
        );
    if (dram_system->last_address == 0) {
      printf("Error␣creating␣Address␣array.\n");
      exit(1);
    }
  }
  /* Create the circular buffer as usual */
  circbuffer = calloc(CIRC_BUFFER_SIZE +1, sizeof(int));
  if (circbuffer == 0) {
    printf("Error␣creating␣Circular␣buffer.\n");
    exit(1);
  }
  return(dram_system);
}

/* This function frees up the memory used by the DRAM-model */

void free_dram(dram_system_t *dram_system) {
  free(dram_system->last_address);
  free(dram_system->ready_channels);
  free(dram_system);
}

/* This function maps an address to a DRAM bank */

int block_to_bank(dram_system_t *dram_system, md_addr_t block) {
  int bank = 0;
  bank = (block / dram_system->block_size) % dram_system->num_channels;
  if (bank < 0) {
    printf("Negative␣bank␣value␣from␣hash!␣This␣shouldn't␣happen!");
    exit(1);
  }
  return bank;
}

/* This function determines if two blocks are on the same page
 * Returns 1 if they are on the same page, 0 otherwise
 */

int is_same_page(dram_system_t *dram_system, md_addr_t block1, md_addr_t
    block2) {
  md_addr_t block_no1,block_no2;
  /* Calculate block numbers by dividing by the size of each block */
  block_no1 = block1 / dram_system->block_size;
  block_no2 = block2 / dram_system->block_size;
  /* Divide by the number of banks to get the distance */
  block_no1 /= dram_system->num_channels;
  block_no2 /= dram_system->num_channels;

  /* If the integer division by the page size is equal, the two are on
   * the same page */

  if ((block_no1/dram_system->page_size) == (block_no2/dram_system->
      page_size)) {
    return 1;
  }
  /* Fall-through : No match */
  return 0;
}


/* This function calculates the access time of a singel access based on
 * the state of the DRAM-system.
 * It returns the access time in number of ticks (due to compability issues)
 */

unsigned int access_dram(dram_system_t *dram_system, md_addr_t block, int
    bsize, tick_t now) {
  int dram_bank;      /* The bank in use, calculated based on the address */
  int latency;        /* The calculated latency - in ticks */
  int control_time;   /* Time required to transfer a control word to DRAM */
  int core_time;      /* Time required to transfer data from cells to latches
      */
  int data_time;      /* Time required to transfer data from latches to
                       * controller */
  int overlap;        /* Overlapping time */
  int pipelining;     /* Time that can be overlapped (from 0 to control_time)
      */

  dram_bank = block_to_bank(dram_system, block);

  control_time = dram_system->control_time;
  core_time = dram_system->core_time;
  data_time = dram_system->data_time;

  pipelining = 0 ;     /* Safe initialization */

  dram_system->accesses++;

  /* Update the circular buffer */

  circbuffer[circbuffer[0]+1] = now - dram_system->ready_channels[dram_bank
      ];
  circbuffer[0] = (circbuffer[0] + 1) % CIRC_BUFFER_SIZE;

  /* If the DRAM chip is free */
  if (dram_system->ready_channels[dram_bank] < now) {
    /*Check if we hit an open page */
    if (is_same_page(dram_system, block, dram_system->last_address[dram_bank
        ])) {
```

```
      /* We hit an open page, transfer time is reduced. */
      dram_system->page_hits++;
      latency = control_time + data_time;
    } else {
      /* We hit a closed page */
      latency = control_time + core_time + data_time;
    }
  } else {
    /* The DRAM bank is currently occupied */
    dram_system->stalls++;
    /* Calculate overlapping time (pipelining) */
    overlap = dram_system->ready_channels[dram_bank] - now;
    if (overlap > control_time) {
      pipelining = control_time;     /* Only pipeline control */
    } else {
      pipelining = overlap;
    }
    if (is_same_page(dram_system, block, dram_system->last_address[dram_bank
        ])) {
      /* We hit an open page */
      dram_system->page_hits++;
      latency = (control_time - pipelining) + data_time;
    } else {
      /* We hit a closed page */
      latency = (control_time - pipelining) + core_time + data_time;
    }
    /* Latency observed by the system, wait for other request to complete */
    latency += (dram_system->ready_channels[dram_bank] - now);
  }
  /* Commit changes to the data structure */
  dram_system->ready_channels[dram_bank] = now + latency;
  dram_system->last_address[dram_bank] = block;

  /* Update Statistics */
  dram_system->latency += latency;
  //printf("Accessing %ld at time %ld with latency %d.\n", block, now,
      latency);
  return latency;
}

/* This function gets the current bandwidth useage by returning the
 * number of busy channels
 */

unsigned int get_channels_in_use(dram_system_t *dram_system, tick_t now) {
  int no_busy = 0;
  int i;
  for (i=0; i < dram_system->num_channels; i++) {
    if (dram_system->ready_channels[i] > now) {
      /* Channel is busy */
      no_busy++;
    }
  }
  return no_busy;
}
```

```
/* This function registers DRAM stats based on the data structure */
void dram_reg_stats(dram_system_t *dram_system, struct stat_sdb_t *sdb) {
    stat_reg_counter(sdb, "dram.accesses", "total number of memory accesses"
        ,
                    &(dram_system->accesses), 0, NULL);
    stat_reg_counter(sdb, "dram.latency", "total latency of all accesses",
                    &(dram_system->latency), 0, NULL);
    stat_reg_formula(sdb, "dram.avglatency", "average latency in DRAM",
                    "dram.latency / dram.accesses", NULL);
    stat_reg_counter(sdb, "dram.stalls", "total number of stalls",
                    &(dram_system->stalls), 0, NULL);
    stat_reg_formula(sdb, "dram.percentstall",
                     "percentage of acceses that are stalled",
                     "dram.stalls * 100.0 / dram.accesses", NULL);
    stat_reg_counter(sdb, "dram.page_hits",
                    "total number of hits on open pages",
                    &(dram_system->page_hits), 0, NULL);
    stat_reg_formula(sdb, "dram.percenthits",
                     "percent of accesses hitting open pages",
                     "dram.page_hits * 100.0 / dram.accesses", NULL);
}

/* This function is called every cycle and generates a bandwidth trace if
    needed */
void dram_trace(dram_system_t *dram_system, tick_t now) {
  /* Only trace if non-zero interval is specified */
  if (dram_system->trace_interval > 0) {
    /* Only trace on given interval, use mod to accomplish this */
    if (now % dram_system->trace_interval == 0) {
      printf ("%ld; %d\n", now, get_channels_in_use(dram_system, now));
    }
  }
}

int get_occupancy(void) {
  int i;
  long total = 0;
  for (i=1; i<CIRC_BUFFER_SIZE+1; i++) {
    total += circbuffer[i];
  }
  return (total / CIRC_BUFFER_SIZE);
}
```

E.5 Prefetch.c

Listing E.5: Prefetch.c

```
/* prefetch.c - prefetching module routines */
/* Written by Marius Grannaes 2006 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "cache.h"
#include "prefetch.h"
#include "loader.h"
#include "dram.h"

/* Globals */

extern int bwa_treshold;
extern int bwa_enable;

/* We need to store pointers to the caches so that we can access them */
struct cache_t *datal1;
struct cache_t *datal2;
struct cache_t *instructionl1;
struct cache_t *instructionl2;

/* And a pointer to memory so we can examine the returning data */
struct mem_t *mem;

/* This is where the prefetching occurs */
location_t prefetch_location = None;

/* The type of access that triggers a prefetch */
trigger_type_t target_type = Cache_Miss;

/* This is the actual target cache, selected from the above in the
    prefetch_init function */
struct cache_t *target_cache;

/* This is a flag that is set to 1 if a prefetch is attempted. */
int prefetch_attempt = 0;

/* Function pointer to the active algorithm */
void (*prefetch_algorithm)(prefetch_trigger_t);

/* Prefetch degree */
int prefetch_degree = 1;

/* CZone size in bits */
int czone_size = 16;

/* Table size in entries (GHB, RPT etc) */
int table_size = 64;
```

```
/* Global History buffer */
md_addr_t *ghb;

/* RPT table */
rpt_entry_t *rpt;

/* Stream offset */
int stream_offset = 4;

/* AVD table */
avd_entry_t *avd;

/* AVD treshold (MAXAVD */
long int maxavd = 65535;

/* Delta buffer */
md_addr_t *delta_buffer;

/* Global History buffer top */
int ghb_top = 0;

/* Global History buffer fill */
int ghb_is_full = 0;

/* Return latency - Used when prefetches uses returned data */
int return_latency = 0;

/* Last PC value - Note this is a hack:
 * This is done to avoid passing the current PC value along with
 * each call to cache_access().
 * Thus the code becomes more modular and easier to maintain.
 */

md_addr_t last_PC_value;

/*
 * This function sets the last PC value, this is typically done when the
     first
 * access to level 1 data cache occurs - note: This type of prefetching only
 * makes sense for data-prefetchers!
 */

md_addr_t set_last_PC_value(md_addr_t PC) {
  last_PC_value = PC;
  return last_PC_value;
}

/*
 * This function sets where the prefetching occurs.
 */
int register_prefetch_location(location_t location) {
  prefetch_location = location;
  return(0);
}
```

```
/*
 * This function sets the prefetching event that triggers a prefetching
 */
void register_prefetch_target(trigger_type_t target) {
  target_type = target;
}

/*
 * This function sets the prefetch degree.
 */

void register_prefetch_degree(int degree) {
  prefetch_degree = degree;
}

/*
 * This function sets the function pointer to point to the needed function.
 */

int register_prefetch_algorithm(void (*algorithm)(prefetch_trigger_t)) {
  prefetch_algorithm = algorithm;
}

/*
 * This function sets the return latency
 */
void set_return_latency(int latency) {
  return_latency = latency;
}

/*
 * This function sets up the prefetching. In addition to storing the cache
     pointers, it
 * resolves which location needs to be prefetched
 */

int prefetch_init(struct cache_t *il1, struct cache_t *il2, struct cache_t *
    dl1, struct cache_t *dl2,int c_size,int t_size, struct mem_t *memory) {
  czone_size = c_size;
  table_size = t_size;
  datal1 = dl1;
  datal2 = dl2;
  instructionl1 = il1;
  instructionl2 = il2;
  mem = memory;
  if (prefetch_location == Cache_IL1) {
    target_cache = instructionl1;
  } else if (prefetch_location == Cache_DL1) {
    target_cache = datal1;
  } else if (prefetch_location == Cache_IL2) {
    target_cache = instructionl2;
  } else if (prefetch_location == Cache_DL2) {
    target_cache = datal2;
  }
  /* Set up the refrence prediction table */
  rpt = calloc(table_size,sizeof(rpt_entry_t));
  /* Set up the AVD prediction table */
  avd = calloc(table_size,sizeof(avd_entry_t));
  /* Set up the global history buffer */
  ghb = calloc(table_size,sizeof(md_addr_t));
  /* Allocate a delta buffer of equal size to the GHB */
  delta_buffer = calloc(table_size,sizeof(md_addr_t));
  return (1);
}

/*
 * This is the prefetching function. It issues the actual prefetches.
 * This is used by the algorithms as an abstraction
 */
void prefetch(md_addr_t adress, struct cache_t *target, time_t now) {
  /* Check if the prefetched address is valid */
  if (MD_VALID_ADDR(adress)) {
    /* Check if the prefetched address is allready in the cache */
    if (cache_probe(target, adress) == FALSE) {
      if (bwa_enable) {
        if (get_occupancy() < bwa_treshold) {
          cache_access(target, Prefetch, adress, NULL, 4, now, NULL,NULL);
        }
      } else {
        cache_access(target, Prefetch, adress, NULL, 4, now, NULL,NULL);
      }
    }
  }
}

/*
 * As the name implies this prefetching method does no prefetching.
 * This is the default case.
 */
void no_prefetch(prefetch_trigger_t trigger) {
}

/*
 * This is the sequential prefetching on Miss algorithm.
 * When a miss in the cache occurs on block X, block X+1 is prefetched.
 */
void sequential_prefetch(prefetch_trigger_t trigger) {
  int i;
  for (i=1; i<=prefetch_degree; i++) {
    prefetch(trigger.address + i*target_cache->bsize, target_cache, trigger.
        time);
  }
}

/*
 * This is a delta correlation prefetching algorithm on access
 * IT stores the last two accesses. If the delta is equal in access
 * X,X-1,X-2, a prefetch is issued.
 */

void delta_correlation_prefetch(prefetch_trigger_t trigger) {
  int i,j;
  int delta_count = 0;
  int delta_1;
  int delta_2;
  int delta;
  md_addr_t adress;
  /* Insert reference into GHB */
  ghb_top = (ghb_top + 1) % table_size;
  ghb[ghb_top] = trigger.address;
  /* Construct delta buffer */
  /* GHB is a circular buffer, need two separate for loops */
  for (i = ghb_top; i >= 0; i--) {
    delta_buffer[delta_count] = ghb[i];
    delta_count++;
  }
  for (i = table_size -1; i > ghb_top; i--) {
    delta_buffer[delta_count] = ghb[i];
    delta_count++;
  }
  /* Correlate deltas - Two deltas are used as Nesbit found optimal */
  delta_1 = delta_buffer[0] - delta_buffer[1];
  delta_2 = delta_buffer[1] - delta_buffer[2];
  /* Search for first delta */
  for (i = 2; i < delta_count-2 ; i++) {
    if (delta_buffer[i] - delta_buffer[i+1] == delta_1) {
      if (delta_buffer[i+1] - delta_buffer[i+2] == delta_2) {
        /* Pattern found */
        /* Start prefetching */
        adress = trigger.address;
        for (j=1; j<=prefetch_degree; j++) {
          /* Find next delta */
          i--;
          if (i < 0) {
            break;
          }
          adress += delta_buffer[i] - delta_buffer[i+1];
          prefetch(adress, target_cache, trigger.time);
        }
        /* Break out of the loop */
        break;
      }
    }
  }
}
/*
 * This is the strided prefetching algorithm using a global history buffer
 * and delta correlation.
 */

void czone_delta_correlation_prefetch(prefetch_trigger_t trigger) {
  int i,j;
  int delta_count = 0;
  int delta_1;
  int delta_2;
  int delta;
  md_addr_t adress;
  /* Insert reference into GHB */
  ghb_top = (ghb_top + 1) % table_size;
  ghb[ghb_top] = trigger.address;
  /* Construct delta buffer */
  /* GHB is a circular buffer, need two separate for loops */
  for (i = ghb_top; i >= 0; i--) {
    /* If this entry matches the czone size, put it into the buffer */
    if (ghb[i] >> czone_size == trigger.address >> czone_size) {
      delta_buffer[delta_count] = ghb[i];
      delta_count++;
    }
  }
  for (i = table_size -1; i > ghb_top; i--) {
    /* If this entry matches the czone size, put it into the buffer */
    if (ghb[i] >> czone_size == trigger.address >> czone_size) {
      delta_buffer[delta_count] = ghb[i];
      delta_count++;
    }
  }
  /* We can only prefetch if there is enough data available */
  if (delta_count>3) {
    /* Correlate deltas - Two deltas are used as Nesbit found optimal */
    delta_1 = delta_buffer[0] - delta_buffer[1];
    delta_2 = delta_buffer[1] - delta_buffer[2];
    /* Search for first delta */
    for (i = 2; i < delta_count-2 ; i++) {
      if (delta_buffer[i] - delta_buffer[i+1] == delta_1) {
        if (delta_buffer[i+1] - delta_buffer[i+2] == delta_2) {
          /* Pattern found */
          /* Start prefetching */
          adress = trigger.address;
          for (j=1; j<=prefetch_degree; j++) {
            /* Find next delta */
            i--;
            if (i < 0) {
              break;
            }
            adress += delta_buffer[i] - delta_buffer[i+1];
            prefetch(adress, target_cache, trigger.time);
          }
          /* Break out of the loop */
          break;
        }
      }
    }
  }
}


/* This is the strided prefetching algorithm using an RPT */
void rpt_prefetch(prefetch_trigger_t trigger) {

  int index;          /* Index into the RPT table */
  int i;              /* Loop index */
  tick_t oldest;      /* Used to find a RPT entry to replace */
  int correct;        /* Flag if predictions are correct */
  /* Check if the entry is in the table */
  index = -1;

  /* Linear search trough the RPT */
  for (i = 0; i < table_size; i++) {
    if (last_PC_value == rpt[i].pc) {
      index = i;
      break;
    }
  }
  if (index > -1) {
    /* Entry is in table */
    /* Calculate if prediction is correct */
    if (trigger.address == rpt[index].prev_addr + rpt[index].stride) {
      correct = 1;
    } else {
      correct = 0;
    }

    switch(rpt[index].state) {
      case Initial:
        if (!correct) {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = Transient;
        } else {
          rpt[index].state = Steady;
        }
        break;
      case Transient:
        if (correct) {
          rpt[index].state = Steady;
        } else {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = No_prediction;
        }
        break;
      case Steady:
        if (correct) {
          rpt[index].state = Steady;
        } else {
          rpt[index].state = Initial;
        }
        break;
      case No_prediction:
        if (correct) {
          rpt[index].state = Transient;
        } else {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = No_prediction;
        }
        break;
      default:
        printf("Something weird happened, shouldn't be in this state");
    }
    rpt[index].prev_addr = trigger.address;
    /* Update access time */
    rpt[index].atime = trigger.time;
    /* If we now are in the steady state; issue prefetches! */
    if (rpt[index].state == Steady) {
      for (i=1; i<=prefetch_degree; i++) {
        prefetch(trigger.address + i*rpt[index].stride, datal2, trigger.time
            );
      }
    }
  } else {
    /* This entry is not in the table, so we insert it. */
    /* Find the oldest entry through linear search */
    oldest = rpt[0].atime;
    index = 0;
    for (i = 0; i < table_size; i++) {
      if (oldest > rpt[i].atime) {
        oldest = rpt[i].atime;
        index = i;
      }
    }
    /* Replace the oldest */
    rpt[index].pc = last_PC_value;
    rpt[index].prev_addr = trigger.address;
    rpt[index].stride = 0;
    rpt[index].atime = trigger.time;
    rpt[index].state = Initial;
  }
}



/* This is the streaming prefetching used in the Power 4 processor by IBM
 * It has been modified to fit a memory hierarchy of only two levels
 * For more information see:
 * http://www.research.ibm.com/journal/rd/461/tendler.html
 * NOTE: To detect streams, we use the rpt
 */
void stream_prefetch(prefetch_trigger_t trigger) {

  int index;          /* Index into the RPT table */
  int i;              /* Loop index */
  tick_t oldest;      /* Used to find a RPT entry to replace */
  int correct;        /* Flag if predictions are correct */

  /* Only prefetch if it is a data access */
  if (last_PC_value != -1) {
    /* Check if this is the correct type */

    /* Check if the entry is in the table */
    index = -1;

    /* Linear search trough the RPT */
    for (i = 0; i < table_size; i++) {
      if (last_PC_value == rpt[i].pc) {
        index = i;
        break;
      }
  }
  if (index > -1) {
    /* Entry is in table */
    /* Calculate if prediction is correct */
    if (trigger.address == rpt[index].prev_addr + rpt[index].stride) {
      correct = 1;
    } else {
      correct = 0;
    }

    switch(rpt[index].state) {
      case Initial:
        if (!correct) {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = Transient;
        } else {
          rpt[index].state = Steady;
        }
        break;
      case Transient:
        if (correct) {
          rpt[index].state = Steady;
        } else {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = No_prediction;
        }
        break;
      case Steady:
        if (correct) {
          rpt[index].state = Steady;
        } else {
          rpt[index].state = Initial;
        }
        break;
      case No_prediction:
        if (correct) {
          rpt[index].state = Transient;
        } else {
          rpt[index].stride = trigger.address - rpt[index].prev_addr;
          rpt[index].state = No_prediction;
        }
        break;
      default:
        printf("Something␣weird␣happened,␣shouldn't␣be␣in␣this␣state");
    }
    rpt[index].prev_addr = trigger.address;
    /* Update access time */
    rpt[index].atime = trigger.time;
    /* If we now are in the steady state; issue prefetches! */
    if (rpt[index].state == Steady) {
      /* Stream prefetching */
      /* First access goes to the L1 */
      prefetch(trigger.address + rpt[index].stride, datal1, trigger.time);
      /* Then some data is transferred to the l2 */
      for (i=stream_offset; i< prefetch_degree + stream_offset ; i++) {
          prefetch(trigger.address + i*rpt[index].stride, datal2, trigger.
              time);
        }
      }

    } else {
      /* This entry is not in the table, so we insert it. */
      /* Find the oldest entry through linear search */
      oldest = rpt[0].atime;
      index = 0;
      for (i = 0; i < table_size; i++) {
        if (oldest > rpt[i].atime) {
          oldest = rpt[i].atime;
          index = i;
        }
      }
      /* Replace the oldest */
      rpt[index].pc = last_PC_value;
      rpt[index].prev_addr = trigger.address;
      rpt[index].stride = 0;
      rpt[index].atime = trigger.time;
      rpt[index].state = Initial;
    }
  }
}

/* This is the address-value delta algorithm.
 * It compares the load adress with the data returned
 * If it is similiar, a prefetch is issues
 * See prefetch.h for a reference to litterature.
 */

void avd_prefetch(prefetch_trigger_t trigger) {
  md_addr_t data;    /* Store the returned data */
  int index;         /* Index to the AVD table */
  int i;             /* General purpose loop index */

  /* Get the associated data */
  mem_access(mem, Read, trigger.address, &data, sizeof(md_addr_t));
  /* Check if delta is within bounds (maxavd) */

  if (((trigger.address - data) < maxavd) ||
        ((data - trigger.address) <maxavd)) {
    /* Prefetch into the future!
     * This is due to the fact that the data will not be
     * available until the return latency has passed
     */
    prefetch(((data >>8)<<8), target_cache, trigger.time + return_latency);
  }
}

/*
 * This is the prefetching engine, it packs a trigger into the
      prefetch_trigger_t format
 * and uses a function pointer to send it to the correct place.
 */
```

```
int process_prefetch_trigger(trigger_type_t type, location_t location,
    md_addr_t address, tick_t now) {

  prefetch_trigger_t  prefetch_trigger;

  /* Avoid prefetching algorithms that generate new prefetching */
  if (prefetch_attempt == 0) {
    /* Pack the data into a trigger_type_t */
    prefetch_trigger.type = type;
    prefetch_trigger.location = location;
    prefetch_trigger.address = address;
    prefetch_trigger.time = now;

    /* Send the prefetch trigger to the
     * correct algorithm using function pointers.
     * prefetch_attempt variable is used as a lock to avoid cascading
     * prefetches
     */
    prefetch_attempt = 1;
    if ((type==target_type) && (location == prefetch_location)) {
      prefetch_algorithm(prefetch_trigger);
    }
    prefetch_attempt = 0;
  }
}
```

E.6 Prefetch.h

Listing E.6: Prefetch.h

```
/* prefetch.h - prefetch module interfaces and definitions */
/* Written by Marius Grannaes 2006 */

#ifndef PREFETCH_H
#define PREFETCH_H

#include <stdio.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "memory.h"
#include "stats.h"
#include "cache.h"

/* The various trigger types */

typedef enum {
    Cache_Miss,             /* A miss in the cache */
    Cache_Hit,              /* A hit in the cache */
    Memory_Access,          /* An access to memory by an instruction */
    PC_Update,              /* Program counter is updated */
    No_event                /* Dummy trigger */
} trigger_type_t;

/* This enum handles whare prefetching triggers happen */

typedef enum {
                Cache_IL1,              /* Event happened in the D-I1 cache
                    */
                Cache_DL1,              /* Event happened in the D-L1 cache
                    */
                Cache_IL2,              /* Event happened in the I-L2 cache
                    */
                Cache_DL2,              /* Event happened in the D-L2 cache
                    */
                DRAM,                   /* Event happened in DRAM */
                None                    /* When location doesn't matter - eg
                    PC_Update */
} location_t;

/* The main prefetching structure */

typedef struct {
  trigger_type_t type;      /* What happened */
  location_t location;      /* Where something did happen */
  md_addr_t address;        /* Adress of memory access */
  tick_t time;              /* Time of access */
} prefetch_trigger_t;

/* States for the rpt table */
/* Defined in the paper by Chen and Baer */
```

```
typedef enum {
  Initial,
  Transient,
  Steady,
  No_prediction
} rpt_state_t;

/* RPT table structure */
typedef struct {
  md_addr_t pc;              /* Adress of loading instruction */
  md_addr_t prev_addr;       /* Previous adress loaded by instruction */
  int stride;                /* Recorded stride */
  rpt_state_t state;         /* Current state */
  tick_t atime;              /* Time of last access (in ticks) */
} rpt_entry_t;

/* AVD prefetcher as in the paper:
 * "Adress-Value Delta (AVD) Prediction: Increasing the Effectiveness of
     Runahead
 * Execution by Exploiting Regular Memory Allocation Patterns" by
 * Onur Mutlu, Hyesoon Kim and Uale N. Patt
 */

/* AVD table structure */
typedef struct {
  md_addr_t pc;              /* Adress of loading instruction */
  int avd;                   /* Calculated delta */
  tick_t atime;              /* Time of last access (in ticks) */
  int confidence;            /* Confidence in prediction */
} avd_entry_t;


/* Prototype declarations */
md_addr_t set_last_PC_value(md_addr_t PC);

int register_prefetch_location(location_t location);

void register_prefetch_degree(int degree);

void register_prefetch_target(trigger_type_t target);

int register_prefetch_algorithm(void (*algorithm)(prefetch_trigger_t));

void set_return_latency(int latency);

int prefetch_init(struct cache_t *il1, struct cache_t *il2, struct cache_t *
    dl1, struct cache_t *dl2,int c_size,int t_size, struct mem_t *memory);

void prefetch(md_addr_t adress, struct cache_t *target, time_t now);

void no_prefetch(prefetch_trigger_t trigger);

void sequential_prefetch(prefetch_trigger_t trigger);

void delta_correlation_prefetch(prefetch_trigger_t trigger);
void czone_delta_correlation_prefetch(prefetch_trigger_t trigger);

void rpt_prefetch(prefetch_trigger_t trigger);

void avd_prefetch(prefetch_trigger_t trigger);

void stream_prefetch(prefetch_trigger_t trigger);

int process_prefetch_trigger(trigger_type_t type, location_t location,
    md_addr_t address, tick_t now);

#endif /* PREFETCH_H */
```

E.7 Memory.h

Listing E.7: Memory.h - Unified diff against SimpleScalar 3.0d

```
--- ../simplesim-3.0-orig/memory.h        2003-10-09 03:13:46.000000000 +0200
+++ ../simplesim-3.0/memory.h     2006-05-08 23:14:32.000000000 +0200
@@ -86,7 +86,8 @@
 /* memory access command */
 enum mem_cmd {
   Read,                       /* read memory from target (simulated prog)
       to host */
-  Write                       /* write memory from host (simulator) to
     target */
+  Write,              /* write memory from host (simulator) to target */
+  Prefetch   /* prefetch memory */
 };

 /* memory access function type, this is a generic function exported for the
```

Appendix F

CMP Code

F.1 Makefile

Listing F.1: Makefile - Unified diff against Uniprocessor version

```
--- ../simplesim-3.0/Makefile    2006-05-25 23:42:04.000000000 +0200
+++ ../../../felles-svn/project/branches/grannas_prefetch/Makefile
    2006-02-06 22:55:24.000000000 +0100
@@ -78,7 +78,7 @@
 ##      Windows NT version 4.0, Cygnus CygWin/32 beta 19
 ##
 CC = gcc-3.4
-OFLAGS = -g
+OFLAGS = -g
 MFLAGS = `./sysprobe -flags`
 MLIBS  = `./sysprobe -libs` -lm
 ENDIAN = `./sysprobe -s`
@@ -277,7 +277,7 @@
 #
 # all the sources
 #
-SRCS = main.c dram.c prefetch.c sim-fast.c sim-safe.c sim-cache.c sim-
    profile.c \
+SRCS = dram.c prefetch.c shared.c main.c sim-fast.c sim-safe.c sim-cache.c
    sim-profile.c \
        sim-eio.c sim-bpred.c sim-cheetah.c sim-outorder.c \
        memory.c regs.c cache.c bpred.c ptrace.c eventq.c \
        resource.c endian.c dlite.c symbol.c eval.c options.c range.c \
@@ -287,7 +287,7 @@
        target-alpha/alpha.c target-alpha/loader.c target-alpha/syscall.c \
        target-alpha/symbol.c

-HDRS = dram.h syscall.h memory.h regs.h sim.h loader.h cache.h bpred.h
    ptrace.h \
+HDRS = dram.h prefetch.h syscall.h memory.h regs.h sim.h loader.h cache.h
    bpred.h ptrace.h \
        eventq.h resource.h endian.h dlite.h symbol.h eval.h bitmap.h \
        eio.h range.h version.h endian.h misc.h \
        target-pisa/pisa.h target-pisa/pisabig.h target-pisa/pisalittle.h \
@@ -305,7 +305,7 @@
 #
 # programs to build
 #
-PROGS = sim-outorder$(EEXT)
+PROGS = controller$(EEXT) sim-outorder$(EEXT)

 #
 # all targets, NOTE: library ordering is important...
@@ -388,8 +388,10 @@
 sim-cache$(EEXT):        sysprobe$(EEXT) sim-cache.$(OEXT) cache.$(OEXT) $(
     OBJS) libexo/libexo.$(LEXT)
         $(CC) -o sim-cache$(EEXT) $(CFLAGS) sim-cache.$(OEXT) cache.$(OEXT)
             $(OBJS) libexo/libexo.$(LEXT) $(MLIBS)

-sim-outorder$(EEXT):     sysprobe$(EEXT) sim-outorder.$(OEXT) dram.$(OEXT)
    cache.$(OEXT) prefetch.$(OEXT) bpred.$(OEXT) resource.$(OEXT) ptrace.$(
    OEXT) $(OBJS) libexo/libexo.$(LEXT)
-        $(CC) -o sim-outorder$(EEXT) $(CFLAGS) sim-outorder.$(OEXT) cache.$(
    OEXT) dram.$(OEXT) prefetch.$(OEXT) bpred.$(OEXT) resource.$(OEXT) ptrace
    .$(OEXT) $(OBJS) libexo/libexo.$(LEXT) $(MLIBS)
+sim-outorder$(EEXT):     dram.$(OEXT) prefetch.$(OEXT) sysprobe$(EEXT) sim-
    outorder.$(OEXT) cache.$(OEXT) bpred.$(OEXT) resource.$(OEXT) shared.$(
    OEXT) ptrace.$(OEXT) $(OBJS) libexo/libexo.$(LEXT)
+        $(CC) -o sim-outorder$(EEXT) $(CFLAGS) sim-outorder.$(OEXT) dram.$(
    OEXT) prefetch.$(OEXT) cache.$(OEXT) shared.$(OEXT) bpred.$(OEXT)
    resource.$(OEXT) ptrace.$(OEXT) $(OBJS) libexo/libexo.$(LEXT) $(MLIBS)
+controller$(EEXT): shared.$(OEXT) controller.c
+        $(CC) $(CFLAGS) shared.$(OEXT) controller.c -o controller$(EEXT)

 exo libexo/libexo.$(LEXT): sysprobe$(EEXT)
         cd libexo $(CS) \
@@ -497,8 +499,6 @@
 regs.$(OEXT): options.h stats.h eval.h
 cache.$(OEXT): host.h misc.h machine.h machine.def cache.h memory.h options
     .h
 cache.$(OEXT): stats.h eval.h
-prefetch.$(OEXT): host.h misc.h machine.h machine.def cache.h memory.h
    options.h
-prefetch.$(OEXT): stats.h eval.h cache.h
 bpred.$(OEXT): host.h misc.h machine.h machine.def bpred.h stats.h eval.h
 ptrace.$(OEXT): host.h misc.h machine.h machine.def range.h ptrace.h
 eventq.$(OEXT): host.h misc.h machine.h machine.def eventq.h bitmap.h
@@ -508,6 +508,8 @@
 dlite.$(OEXT): host.h misc.h machine.h machine.def version.h eval.h regs.h
 dlite.$(OEXT): memory.h options.h stats.h sim.h symbol.h loader.h range.h
 dlite.$(OEXT): dlite.h
+prefetch.$(OEXT): host.h misc.h machine.h machine.def cache.h memory.h
    options.h
+prefetch.$(OEXT): stats.h eval.h cache.h
 symbol.$(OEXT): host.h misc.h target-pisa/ecoff.h loader.h machine.h
 symbol.$(OEXT): machine.def regs.h memory.h options.h stats.h eval.h symbol
     .h
 eval.$(OEXT): host.h misc.h eval.h machine.h machine.def
```

F.2 Sim-outorder.c

Listing F.2: Sim-outorder.c - Unified diff against Uniprocessor version

```
--- ../simplesim-3.0/sim-outorder.c      2006-05-31 13:32:54.000000000 +0200
+++ ../../../felles-svn/project/branches/grannas_prefetch/sim-outorder.c
            2006-05-31 13:33:51.000000000 +0200
@@ -54,6 +54,11 @@
 #include <math.h>
 #include <assert.h>
 #include <signal.h>
+#include <errno.h>
+#include <sys/types.h>
+#include <sys/ipc.h>
+#include <sys/sem.h>
+#include <sys/shm.h>

 #include "host.h"
 #include "misc.h"
@@ -72,9 +77,11 @@
 #include "ptrace.h"
 #include "dlite.h"
 #include "sim.h"
+#include "shared.h"
 #include "dram.h"
 #include "prefetch.h"

+
 /*
  * This file implements a very detailed out-of-order issue superscalar
  * processor with a two-level memory system and speculative execution
      support.
@@ -88,11 +95,36 @@
 /* simulated memory */
 static struct mem_t *mem = NULL;

+union semun {
+  int val;
+  struct semid_ds *buf;
+  ushort *array;
+} arg;
+
+/* Global shared memory variables */
+
+int sync_semaphore_id;
+int controller_semaphore_id;
+int report_semaphore_id;
+int l2_semaphore_id;
+struct sembuf sb = {0, -1, 0}; /* set to allocate resource */
+
+int controller_id; // The id of the shared memory segment for the
    controller
+int counter_id; // The id containgin the shared memory segment for the
    counter
+
+int *controller; // Pointer to the controller segment
+counter_t *counter; // Pointer to the counter segment
```

```
 /*
  * simulator options
  */

+/* The number of concurrent processors */
+unsigned int total_cpus;
+
+/* This processors number */
+unsigned int my_cpuid;
+
 /* Prefetch options */

 /* prefetching type {none|sequential....} */
@@ -163,6 +195,9 @@
 /* maximum number of inst's to execute */
 static unsigned int max_insts;

+/* maximum number of cycles to execute */
+tick_t max_cycles;
+
 /* number of insts skipped before timing starts */
 static int fastfwd_count;

@@ -476,6 +511,7 @@
          : (panic("bad␣stat␣class"), 0))))


+
 /* memory access latency, assumed to not cross a page boundary */
 static unsigned int                       /* total latency of access */
 mem_access_latency(int blk_sz)            /* block size accessed */
@@ -503,44 +539,40 @@
 {
   unsigned int lat;
   process_prefetch_trigger(Cache_Miss,Cache_DL1,baddr,sim_cycle);
-  if (cache_dl2)
-    {
+  if (cache_dl2) {
        if (perfect_l2) {
          lat = cache_dl2->hit_latency;
        } else {
+         my_lock(l2_semaphore_id, 0);
          /* access next level of data cache hierarchy */
          lat = cache_access(cache_dl2, cmd, baddr, NULL, bsize,
-                                  /* now */now, /* pudata */NULL, /* repl addr */
    NULL);
+                              /* now */now, /* pudata */NULL, /* repl addr */NULL
    );
+         my_unlock(l2_semaphore_id, 0);
        }
        process_prefetch_trigger(Memory_Access, Cache_DL2,baddr,sim_cycle);
        if (cmd == Read)
                return lat;
-       else
-        {
```

```
-          /* FIXME: unlimited write buffers */
-          return 0;
-        }
-    }
-  else
-    {
+      else {
+              /* FIXME: unlimited write buffers */
+            return 0;
+          }
+    } else {
        /* access main memory */
        if (cmd == Read) {
          /* If the dram system is defined using 0 channels, then fallback
           * to old model */
-         if (dram_system->num_channels == 0) {
-                 lat = mem_access_latency(bsize);
+         if (num_channels == 0) {
+           lat = mem_access_latency(bsize);
          } else {
-           lat = access_dram(dram_system,baddr,bsize,now);
+           lat = access_dram(dram_system,baddr,bsize,my_cpuid,now);
          }
          process_prefetch_trigger(Memory_Access,DRAM,baddr,sim_cycle);
          return lat;
-       }
-       else
-        {
-          /* FIXME: unlimited write buffers */
-          return 0;
-        }
-    }
+       } else {
+              /* FIXME: unlimited write buffers */
+              return 0;
+          }
+   }
 }

 /* l2 data cache block miss handler function */
@@ -556,10 +588,10 @@
   if (cmd == Read) {
     /* If the dram system is defined using 0 channels, then fallback
      * to old model */
-    if (dram_system->num_channels == 0) {
+    if (num_channels == 0) {
          latency = mem_access_latency(bsize);
       } else {
-        latency = access_dram(dram_system,baddr,bsize,now);
+        latency = access_dram(dram_system,baddr,bsize,my_cpuid,now);
       }
     }
   else
@@ -582,18 +614,19 @@
            tick_t now)            /* time of access */
 {
   unsigned int lat;
-
 if (cache_il2)
     {
       /* access next level of inst cache hierarchy */
       if (perfect_l2) {
         lat = cache_il2->hit_latency;
       } else {
+        my_lock(l2_semaphore_id, 0);
         lat = cache_access(cache_il2, cmd, baddr, NULL, bsize,
-                        /* now */now, /* pudata */NULL, /* repl addr */NULL
   );
+       /* now */now, /* pudata */NULL, /* repl addr */NULL);
+        my_unlock(l2_semaphore_id, 0);
       }
       if (cmd != Read) {
-              panic("writes to instruction memory not supported");
+        panic("writes to instruction memory not supported");
       }
       process_prefetch_trigger(Memory_Access, Cache_IL2,baddr, sim_cycle);
     }
@@ -603,19 +636,20 @@
     if (cmd == Read) {
     /* If the dram system is defined using 0 channels, then fallback
      * to old model */
-    if (dram_system->num_channels == 0) {
+    if (num_channels == 0) {
          lat = mem_access_latency(bsize);
        } else {
-         lat = access_dram(dram_system,baddr,bsize,now);
+         lat = access_dram(dram_system,baddr,bsize,my_cpuid,now);
        }
    process_prefetch_trigger(Memory_Access, DRAM, baddr, sim_cycle);
    } else {
-          panic("writes to instruction memory not supported");
+      panic("writes to instruction memory not supported");
       }
     }
   return lat;
 }

+
 /* l2 inst cache block miss handler function */
 static unsigned int                   /* latency of block access */
 il2_access_fn(enum mem_cmd cmd,                /* access cmd, Read or Write
     */
@@ -626,25 +660,24 @@
 {
   int latency;
   /* this is a miss to the lowest level, so access main memory */
-    if (cmd == Read) {
-    /* If the dram system is defined using 0 channels, then fallback
-     * to old model */
-    if (dram_system->num_channels == 0) {
-         latency = mem_access_latency(bsize);
-       } else {
```

```
-           latency = access_dram(dram_system,baddr,bsize,now);
-         }
-     process_prefetch_trigger(Memory_Access, DRAM, baddr,sim_cycle);
-     }
-     else {
-       panic("writes␣to␣instruction␣memory␣not␣supported");
-     }
-     process_prefetch_trigger(Cache_Miss, Cache_IL2, baddr, sim_cycle);
-     return latency;
+   if (cmd == Read) {
+   /* If the dram system is defined using 0 channels, then fallback
+    * to old model */
+   if (num_channels == 0) {
+         latency = mem_access_latency(bsize);
+       } else {
+         latency = access_dram(dram_system,baddr,bsize,my_cpuid,now);
+       }
+   process_prefetch_trigger(Memory_Access, DRAM, baddr,sim_cycle);
+   }
+   else {
+     panic("writes␣to␣instruction␣memory␣not␣supported");
+   }
+   process_prefetch_trigger(Cache_Miss, Cache_IL2, baddr, sim_cycle);
+   return latency;
 }


-
 /*
  * TLB miss handlers
  */
@@ -701,6 +734,16 @@
 "latency␣of␣all␣pipeline␣operations.\n"
                 );

+   /* Multiprocessors parameters */
+
+   opt_reg_uint(odb, "-cpu:total", "The␣total␣number␣of␣cpu's",
+          &total_cpus, /*default */1,
+          /* print */TRUE, /* format */NULL);
+
+   opt_reg_uint(odb, "-cpu:this", "The␣cpuid␣for␣this␣processor␣(start␣with␣
    zero)",
+          &my_cpuid, /*default */0,
+          /* print */TRUE, /* format */NULL);
+
    /* New DRAM-options */

    opt_reg_uint(odb, "-dram:chan", "number␣of␣DRAM␣channels",
@@ -778,6 +821,10 @@
                  &max_insts, /* default */0,
                  /* print */TRUE, /* format */NULL);

+   opt_reg_int(odb, "-max:cycles", "maximum␣numer␣of␣cycles␣to␣execute",
+           &max_cycles, /* default */0,
+           /* print */TRUE, /* format */NULL);
```

```
+
   opt_reg_int(odb, "-fastfwd", "number␣of␣insts␣skipped␣before␣timing␣
       starts",
              &fastfwd_count, /* default */0,
              /* print */TRUE, /* format */NULL);
@@ -927,7 +974,7 @@

   opt_reg_string(odb, "-cache:dl1",
                 "l1␣data␣cache␣config,␣i.e.,␣{<config>|none}",
-                &cache_dl1_opt, "dl1:128:32:4:l",
+                &cache_dl1_opt, "dl1:8:64:2:l",
                 /* print */TRUE, NULL);

   opt_reg_note(odb,
@@ -1302,9 +1349,15 @@
                    name, &nsets, &bsize, &assoc, &c) != 5)
            fatal("bad␣l2␣D-cache␣parms:␣"
                  "<name>:<nsets>:<bsize>:<assoc>:<repl>");
-          cache_dl2 = cache_create(name, nsets, bsize, /* balloc */FALSE,
+     l2_semaphore_id = get_semaphore_set(SEMAPHORE_L2_KEY, 1);
+
+     if (my_cpuid !=0) {
+       my_lock(l2_semaphore_id, 0);
+     }
+          cache_dl2 = cache_create_shared(name, nsets, bsize, /* balloc */
    FALSE,
                                   /* usize */0, assoc, cache_char2policy(c)
                                   ,
-                                  dl2_access_fn, /* hit lat */cache_dl2_lat
    );
+                                  dl2_access_fn, /* hit lat */cache_dl2_lat
    , /* process_id */my_cpuid);
+    my_unlock(l2_semaphore_id, 0);
        }
     }

@@ -1320,9 +1373,10 @@
     }
   else if (!mystricmp(cache_il1_opt, "dl1"))
     {
-      if (!cache_dl1)
+      /*if (!cache_dl1)
        fatal("I-cache l1 cannot access D-cache l1 as it's undefined");
-      cache_il1 = cache_dl1;
+      cache_il1 = cache_dl1; */
+      fatal("Haakon␣states␣that␣I-cache␣l1␣cannot␣access␣D-cache␣due␣to␣
    varying␣D␣cache␣size␣etc.");

      /* the level 2 I-cache cannot be defined */
      if (strcmp(cache_il2_opt, "none"))
@@ -1569,7 +1623,7 @@
                   "sim_slip␣/␣sim_num_insn", NULL);

   /* register DRAM stats */
-  if (dram_system->num_channels !=0) {
+  if (num_channels !=0) {
      dram_reg_stats(dram_system,sdb);
    }

@@ -1649,7 +1703,9 @@
    sim_num_refs = 0;

    /* create the memory hierachy */
-   dram_system = create_dram(num_channels, block_size, page_size,
     control_time, core_time, data_time, dram_trace_interval);
+   if (num_channels > 0) {
+     dram_system = create_dram(num_channels, block_size, page_size,
     control_time, core_time, data_time, dram_trace_interval,my_cpuid);
+   }

    /* allocate and initialize register file */
    regs_init(&regs);
@@ -4719,6 +4775,22 @@
  void
  sim_main(void)
  {
+   /* Set up semaphore locking for synchronization, One semaphore per cpu */
+   sync_semaphore_id = get_semaphore_set(SEMAPHORE_SYNCH_KEY, total_cpus);
+
+   /* The controller semaphore */
+   controller_semaphore_id = get_semaphore_set(SEMAPHORE_CONTROLLER_KEY, 1);
+
+   /* The report semaphore */
+   report_semaphore_id = get_semaphore_set(SEMAPHORE_REPORT_KEY, 1);
+
+   /* Attatch to shared memory segments */
+   controller_id = get_shmem(SHM_CONTROLLER_KEY, total_cpus*sizeof(int));
+   counter_id = get_shmem(SHM_COUNTER_KEY, total_cpus*sizeof(counter_t));
+
+   controller = (int*) shmem_attatch(controller_id);
+   counter = (counter_t*) shmem_attatch(counter_id);
+
    /* ignore any floating point exceptions, they may occur on mis-speculated
       execution paths */
    signal(SIGFPE, SIG_IGN);
@@ -4824,13 +4896,38 @@
       to eliminate this/next state synchronization and relaxation problems
         */
    for (;;)
      {
+       /* Connect to controller every 10000 clock ticks */
+
+       if (sim_cycle%RESOLUTION == 0) {
+         /* Report status into shared memory */
+         controller[my_cpuid] = WAITING_FOR_COMMAND;
+         counter[my_cpuid] = cache_dl1->misses;
+
+         /* Signal the controller */
+         my_unlock(controller_semaphore_id,0);
+
+         /* Wait for signal from controller to continue */
+         my_lock(report_semaphore_id,my_cpuid);
+
+          /* Read command from shared memory */
+          switch(controller[my_cpuid]) {
+            case   RUN_COMMAND:
+              break;
+            default:
+              printf("Something␣bad␣happened␣in␣command␣transfer!\n");
+              break;
+          }
+        }
+        /* Wait for own semaphore before continuing on next clock cycle */
+        my_lock(sync_semaphore_id, my_cpuid);
+
         /* RUU/LSQ sanity checks */
         if (RUU_num < LSQ_num)
-          panic("RUU_num␣<␣LSQ_num");
+            panic("RUU_num␣<␣LSQ_num");
         if (((RUU_head + RUU_num) % RUU_size) != RUU_tail)
-          panic("RUU_head/RUU_tail␣wedged");
+            panic("RUU_head/RUU_tail␣wedged");
         if (((LSQ_head + LSQ_num) % LSQ_size) != LSQ_tail)
-          panic("LSQ_head/LSQ_tail␣wedged");
+            panic("LSQ_head/LSQ_tail␣wedged");

         /* check if pipetracing is still active */
         ptrace_check_active(regs.regs_PC, sim_num_insn, sim_cycle);
@@ -4850,37 +4947,35 @@
         /* ==> inserts operations into ready queue --> register deps resolved
              */
         ruu_writeback();

-        if (!bugcompat_mode)
-          {
-            /* try to locate memory operations that are ready to execute */
-            /* ==> inserts operations into ready queue --> mem deps resolved
    */
-            lsq_refresh();
-
-            /* issue operations ready to execute from a previous cycle */
-            /* <== drains ready queue <-- ready operations commence execution
    */
-            ruu_issue();
-          }
+        if (!bugcompat_mode) {
+          /* try to locate memory operations that are ready to execute */
+          /* ==> inserts operations into ready queue --> mem deps resolved */
+          lsq_refresh();
+
+          /* issue operations ready to execute from a previous cycle */
+          /* <== drains ready queue <-- ready operations commence execution
    */
+          ruu_issue();
+        }

         /* decode and dispatch new operations */
```

```
       /* ==> insert ops w/ no deps or all regs ready --> reg deps resolved
           */
       ruu_dispatch();

-      if (bugcompat_mode)
-       {
-         /* try to locate memory operations that are ready to execute */
-         /* ==> inserts operations into ready queue --> mem deps resolved
   */
-         lsq_refresh();
-
-         /* issue operations ready to execute from a previous cycle */
-         /* <== drains ready queue <-- ready operations commence execution
   */
-         ruu_issue();
-       }
+      if (bugcompat_mode) {
+        /* try to locate memory operations that are ready to execute */
+        /* ==> inserts operations into ready queue --> mem deps resolved */
+        lsq_refresh();
+
+        /* issue operations ready to execute from a previous cycle */
+        /* <== drains ready queue <-- ready operations commence execution
   */
+        ruu_issue();
+      }

       /* call instruction fetch unit if it is not blocked */
       if (!ruu_fetch_issue_delay)
-       ruu_fetch();
+        ruu_fetch();
       else
-       ruu_fetch_issue_delay--;
+        ruu_fetch_issue_delay--;

       /* update buffer occupancy stats */
       IFQ_count += fetch_num;
@@ -4889,15 +4984,70 @@
       RUU_fcount += ((RUU_num == RUU_size) ? 1 : 0);
       LSQ_count += LSQ_num;
       LSQ_fcount += ((LSQ_num == LSQ_size) ? 1 : 0);
-
-      /* Dram trace */
-      dram_trace(dram_system, sim_cycle);
+

       /* go to next cycle */
       sim_cycle++;

+      /* Signal chained CPU that it can continue execution */
+      my_unlock(sync_semaphore_id,(my_cpuid +1) % total_cpus);
+
       /* finish early? */
       if (max_insts && sim_num_insn >= max_insts)
-       return;
+             return;
+        /* maximum number of cycles reached */
+        if (max_cycles && sim_cycle >= max_cycles)
+          return;
      }
 }
+
+/* This function is called after a simulation finishes so that it keeps in
+ * synch with everyone else
+ */
+
+void sim_continue_ticking() {
+  /* This clock cycle is over */
+  sim_cycle++;
+  /* Signal chained CPU that it can continue execution */
+
+  my_unlock(sync_semaphore_id,(my_cpuid +1) % total_cpus);
+
+  /* Only way out of this loop is to get a command from the controller */
+
+  for(;;) {
+    /* Connect to controller every 10000 clock ticks */
+    if (sim_cycle%RESOLUTION == 0) {
+
+      /* Report status into shared memory */
+      controller[my_cpuid] = SIMULATION_COMPLETED_COMMAND;
+      counter[my_cpuid] = cache_dl1->misses;
+
+      /* Signal the controller */
+      my_unlock(controller_semaphore_id,0);
+
+      /* Wait for signal from controller to continue */
+      my_lock(report_semaphore_id,my_cpuid);
+
+      /* Read command from shared memory */
+      switch(controller[my_cpuid]) {
+        case RUN_COMMAND:
+          break;
+        case HALT_COMMAND:
+          return;
+          break;
+        default:
+          printf("Something␣bad␣happened␣in␣command␣transfer!\n");
+          break;
+      }
+    }
+
+    /* Wait for own semaphore before continuing on next clock cycle */
+    my_lock(sync_semaphore_id,my_cpuid);
+
+    /* go to next cycle */
+    sim_cycle++;
+
+    /* Signal chained CPU that it can continue execution */
+    my_unlock(sync_semaphore_id,(my_cpuid +1) % total_cpus);
+  }
+}
```

F.3 Controller.c

Listing F.3: Controller.c

```
/* This program controls the execution of parallell simplescalar
 * in a CMP enviroment.
 * Control is done trough shared memory segments.
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <signal.h>

#include "host.h" // For counter_t definition
#include "shared.h" // For the keys

#define SCHEDULER_HISTORY_LENGHTH 5

/* Global variables */

int *controller; // Pointer to the controller segment
counter_t *counter; // Pointer to the counter segment

int cpu_count; // The number of CPU's to control

int controller_semaphore; // The ID of the semaphore for controlling the
    invocation
                          // of the controller loop

int sync_semaphore; // The ID of the synchronizing semaphores
int report_semaphore; // Used for locking the CPUs while reporting in
    progress.
int l2_semaphore;    // The semaphore controlling access to the L2 cache

int controller_id; // The id of the shared memory segment for the controller
int counter_id; // The id containgin the shared memory segment for the
    counter

/* This is the main controller loop */

void controller_loop() {
  int command;
  int i;
  int finished = 0; // Set this flag when all simulators have finished.
  int flag;          // Used to check if everyone is finished.
  struct sembuf sb = {0, 0, 0}; /* Semaphore control operation */
  int cycle = 0;     // Cycle counter

  /* Initialization: Allow cpu #0 to start */
  my_unlock(sync_semaphore,0);
  while (finished != 1) {
    // Wait until all cpus have flagged the controller semaphore
    sb.sem_num = 0;
    sb.sem_op = 0-(short)cpu_count;
    if (semop(controller_semaphore, &sb, 1) == -1) {
      perror("Something went wrong while getting the semaphore: ");
      exit(1);
    }
    printf("%d cycles have elapsed.\n", RESOLUTION);
    /* Read the misses from each instance */
    /*for (i = 0; i < cpu_count;i++) {
      printf("Cpu %d has %d misses.\n", i, counter[i]);
    }*/
    flag = 1;
    for (i=0;i<cpu_count;i++) {
      if (controller[i] != SIMULATION_COMPLETED_COMMAND) {
        flag = 0;
      }
    }
    /* If everyone is finished then set finished flag and let everyone halt
       */
    if (flag == 1) {
      finished = 1;
      for (i = 0; i < cpu_count;i++) {
        controller[i] = HALT_COMMAND;
      }
    } else {
      /* Set run command on all cpu's */
      for (i = 0; i < cpu_count;i++) {
        controller[i] = RUN_COMMAND;
      }
    }
    /* Unlock all report locks */
    for (i = 0; i < cpu_count;i++) {
      my_unlock(report_semaphore, i);
    }
    cycle++;
  }
  sleep(5); /* Don't deallocate resources too soon */
}

/* This function frees all allocated shared resources */
void cleanup(void) {
  int shared_id;

  /* Detatch from the shared memory segments */
  printf("Detatching segments.\n");
  shmdt(controller);
  shmdt(counter);

  /* Deallocate shared memory segments */
  shmctl(controller_id, IPC_RMID, NULL);
  shmctl(counter_id, IPC_RMID, NULL);

  /* Remove semaphores */
  printf("Detatch sucessful.\nRemoving semaphores.\n");
  destroy_semaphore(controller_semaphore);
  destroy_semaphore(sync_semaphore);
  destroy_semaphore(report_semaphore);
  destroy_semaphore(l2_semaphore);
  /* Try to detatch the cache */
  /* This is somewhat dirty */
  shared_id = shmget(SHM_L2_KEY,0,0);
  shmctl(shared_id, IPC_RMID, 0);
  shared_id = shmget(SHM_L2_KEY+1,0,0);
  shmctl(shared_id, IPC_RMID, 0);
  shared_id = shmget(SHM_DRAM_KEY,0,0);
  shmctl(shared_id, IPC_RMID, 0);
  shared_id = shmget(SHM_DRAM_KEY+1,0,0);
  shmctl(shared_id, IPC_RMID, 0);
  shared_id = shmget(SHM_DRAM_KEY+2,0,0);
  shmctl(shared_id, IPC_RMID, 0);
  shared_id = shmget(SHM_DRAM_KEY+3,0,0);
  shmctl(shared_id, IPC_RMID, 0);
}

/* Interrupt signal handler */
/* Used to clean up mess when quitting */
/* first, here is the signal handler */
void catch_int(int sig_num) {
  printf("Ctrl-C caugt, cleaning up.\n");
  cleanup();
  printf("Cleanup complete.\n");
  exit(2);
}

int main(int argc, char *argv[]) {
  int i;

  /* Check command line */
  if (argc != 2) {
    printf("Usage:\n./controller <number_of_cpus>\n");
    exit(1);
  }

  /* Set signal handler to own */
  signal(SIGINT, catch_int);

  /* Parse the number of CPUS */
  if ((cpu_count = atoi(argv[1])) < 1) {
    printf("You have started too few cpus\n");
  }

  printf("Starting simulation of %d cpus.\n", cpu_count);

  /* Create the shared memory areas */
  controller_id = create_shmem(SHM_CONTROLLER_KEY, cpu_count *sizeof(int));
  counter_id = create_shmem(SHM_COUNTER_KEY, cpu_count*sizeof(counter_t));

  /* Attatch to the shared segments */
  controller = (int*) shmem_attatch(controller_id);
  counter = (counter_t*) shmem_attatch(counter_id);

  printf("All␣shared␣memory␣created␣and␣attatched.\n");
  printf("Creating␣semaphores.\n");

  /* create a semaphore set with 1 semaphore for controller */
  controller_semaphore = create_semaphore_set(SEMAPHORE_CONTROLLER_KEY, 1);

  /* Initialize the semaphore to 0 */
  semaphore_set_value(controller_semaphore, 0, 0);

  /* Create the synch semaphore, one semaphore per CPU */
  sync_semaphore = create_semaphore_set(SEMAPHORE_SYNCH_KEY, cpu_count);

  /* Initialize all sync semaphores to 0 (no go!) */
  for (i=0; i<cpu_count; i++) {
    semaphore_set_value(sync_semaphore, i, 0);
  }

  /* Create the report semaphore, one semaphore per CPU */
  report_semaphore = create_semaphore_set(SEMAPHORE_REPORT_KEY, cpu_count);

  /* Initialize all report semaphores to 0 (no go!) */
  for (i=0; i<cpu_count; i++) {
    semaphore_set_value(report_semaphore, i, 0);
  }

  /*Create the L2 cache lock and initialize it to 0 */
  l2_semaphore = create_semaphore_set(SEMAPHORE_L2_KEY, 1);
  semaphore_set_value(l2_semaphore, 0, 0);

  printf("Semaphores␣created.\nStarting␣controller␣looop\n");

  controller_loop();

  printf("Simulation␣done.\n");

  cleanup();

  /* Successful return */
  return 0;
}
```

F.4 Shared.c

Listing F.4: Shared.c

```
/* This file is a collection of abstractions for shared memory control
 * It was made to make it easier to use semaphores in the rest of the
     program
 * and make things more readable
 */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "shared.h"


/* This function simply locks a semaphore */
void my_lock(int semaphore_id, int semaphore_number) {
  struct sembuf sb = {0, 0, 0};
  sb.sem_num = semaphore_number;
  sb.sem_op = -1;
  if (semop(semaphore_id, &sb, 1) == -1) {
    perror("Something went wrong while locking a semaphore: ");
    exit(1);
  }
}

/* This function unlocks a semaphore */
void my_unlock(int semaphore_id, int semaphore_number) {
  struct sembuf sb = {0, 0, 0};
  sb.sem_num = semaphore_number;
  sb.sem_op = 1;
  if (semop(semaphore_id, &sb, 1) == -1) {
    perror("Something went wrong while unlocking a semaphore: ");
    exit(1);
  }
}

/* This function destroys a semaphore set */
void destroy_semaphore(int semaphore_id) {
  union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
  } arg;
  if (semctl(semaphore_id, 0, IPC_RMID, arg) == -1) {
    perror("Could not remove  semaphore: ");
    exit(1);
  }
}

/* This function creates a new semaphore set and returns the id */
```

```
int create_semaphore_set(key_t semaphore_key, int setsize) {
  int semaphore_id;
  if ((semaphore_id = semget(semaphore_key, setsize, DEFAULT_PERMISSIONS |
      IPC_CREAT)) == -1) {
    perror("Could not create semaphore :");
    exit(1);
  }
  return semaphore_id;
}

/* This function gets a semaphore set based on the key */
int get_semaphore_set(key_t semaphore_key, int setsize) {
  int semaphore_id;
  if ((semaphore_id = semget(semaphore_key, setsize, DEFAULT_PERMISSIONS))
      == -1) {
    perror("Could not get semaphore, are they created? ");
    exit(1);
  }
  return semaphore_id;
}

/* This function sets the value of a semaphore (useful for initialization */
void semaphore_set_value(int semaphore_id, int semaphore_number, int value)
    {
  union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
  } arg;
  arg.val = value;
  if (semctl(semaphore_id,semaphore_number, SETVAL, arg) == -1) {
    perror("Could not set value of semaphore: ");
    exit(1);
  }
}

/* Shared memory functions */
/* Create a shared memory segment based on a key with a set size*/
int create_shmem(key_t key, int size) {
  int shmem_id;
  if ((shmem_id = shmget(key, size, IPC_CREAT | IPC_EXCL |
      DEFAULT_PERMISSIONS)) < 0) {
    printf("Faulting key is %d with size = %d\n", key, size);
    perror("Could not allocate shared memory: ");
    exit(1);
  }
  return shmem_id;
}

/* Get a shared memory segment based on a key with a set size*/
int get_shmem(key_t key, int size) {
  int shmem_id;
  if ((shmem_id = shmget(key, size, DEFAULT_PERMISSIONS)) < 0) {
    perror("Could not get shared memory: ");
    exit(1);
  }
  return shmem_id;
}

/* This function returns a pointer to the shared memory segment */
void *shmem_attatch(int shmem_id) {
  void *pointer;
  if ((pointer = shmat(shmem_id, NULL, 0)) == NULL) {
    perror("Could not attatch to shared memory: ");
    exit(1);
  }
  return pointer;
}
```

F.5 Shared.h

Listing F.5: Shared.h

```
/* This header file contains the keys for the shared memory segments as
 * well as the keys for the semaphores used
 */

#define RESOLUTION 1000000
#define DEFAULT_PERMISSIONS 0644

#define SHM_CONTROLLER_KEY 13380
#define SHM_COUNTER_KEY 13390
#define SHM_DRAM_KEY 135

#define SHM_L2_KEY 20000

#define SEMAPHORE_CONTROLLER_KEY 9987
#define SEMAPHORE_SYNCH_KEY 8901
#define SEMAPHORE_REPORT_KEY 7891
#define SEMAPHORE_L2_KEY 197

/* Commands issued through shared memory */

#define WAITING_FOR_COMMAND 0
#define RUN_COMMAND 1
#define SIMULATION_COMPLETED_COMMAND 2
#define HALT_COMMAND 3

/* Some useful functions for locking */

/* This function simply locks a semaphore */
void my_lock(int semaphore_id, int semaphore_number);

/* This function unlocks a semaphore */
void my_unlock(int semaphore_id, int semaphore_number);

/* This function destroys a semaphore */
void destroy_semaphore(int semaphore_id);

/* This function creates a new semaphore set and returns the id */
int create_semaphore_set(key_t semaphore_key, int setsize);

/* This function gets a semaphore set based on the key */
int get_semaphore_set(key_t semaphore_key, int setsize);

/* This function sets the value of a semaphore (useful for initialization */
void semaphore_set_value(int semaphore_id, int semaphore_number, int value);

/* Shared memory operations */
/* Create a shared memory segment based on a key with a set size*/
int create_shmem(key_t key, int size);

/* Get a shared memory segment based on a key with a set size*/
int get_shmem(key_t key, int size);

/* This function returns a pointer to the shared memory segment */
```

```
void *shmem_attatch(int shmem_id);
```

F.6 Cache.c

Listing F.6: Cache.c - Unified diff against SimpleScalar 3.0d

```
--- ../simplesim-3.0-orig/cache.c        2003-10-08 17:50:34.000000000 +0200
+++ ../../../felles-svn/project/branches/grannas_prefetch/cache.c
    2006-05-20 13:07:15.000000000 +0200
@@ -48,15 +48,22 @@
   * Copyright (C) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
   */

-
 #include <stdio.h>
 #include <stdlib.h>
 #include <assert.h>
+#include <sys/types.h>
+#include <sys/ipc.h>
+#include <sys/shm.h>
+

 #include "host.h"
 #include "misc.h"
 #include "machine.h"
 #include "cache.h"
+#include "shared.h"
+
+extern unsigned int total_cpus;
+extern unsigned int my_cpuid;

 /* cache access macros */
 #define CACHE_TAG(cp, addr)    ((addr) >> (cp)->tag_shift)
@@ -136,6 +143,40 @@
      }\
    }

+/* The following variables are used as local variables in a shared
+ * cache_t structure
+ */
+
+char *cp__name;                          /* cache name */
+  /* miss/replacement handler, read/write BSIZE bytes starting at BADDR
+     from/into cache block BLK, returns the latency of the operation
+     if initiated at NOW, returned latencies indicate how long it takes
+     for the cache access to continue (e.g., fill a write buffer), the
+     miss/repl functions are required to track how this operation will
+     effect the latency of later operations (e.g., write buffer fills),
+     if !BALLOC, then just return the latency; BLK_ACCESS_FN is also
+     responsible for generating any user data and incorporating the latency
+     of that operation */
+  unsigned int                                  /* latency of block access
    */
+    (*cp__blk_access_fn)(enum mem_cmd cmd,             /* block access
    command */
+                        md_addr_t baddr,           /* program address to access
     */
+                        int bsize,                 /* size of the cache block
    */
```

```
+                    struct cache_blk_t *blk,    /* ptr to cache block struct
     */
+                    tick_t now);                /* when fetch was initiated
    */
+
+
+
+  counter_t cp__hits;          /* total number of hits */
+  counter_t cp__misses;                /* total number of misses */
+  counter_t cp__replacements;  /* total number of replacements at misses */
+  counter_t cp__writebacks;            /* total number of writebacks at
    misses */
+  counter_t cp__invalidations; /* total number of external invalidations */
+  counter_t cp__prefetches;   /* total number of prefetches */
+  counter_t cp__prefetches_ok; /* total number of prefetches that worked */
+
+
+
+
 /* bound sqword_t/dfloat_t to positive int */
 #define BOUND_POS(N)           ((int)(MIN(MAX(0, (N)), 2147483647)))

@@ -258,7 +299,7 @@

 /* create and initialize a general cache structure */
 struct cache_t *                        /* pointer to cache created */
-cache_create(char *name,                /* name of the cache */
+cache_create_shared(char *name,                 /* name of the cache */
             int nsets,                  /* total number of sets in cache */
             int bsize,                  /* block (line) size of cache */
             int balloc,                 /* allocate data space for blocks?
                 */
@@ -270,11 +311,14 @@
                                            md_addr_t baddr, int bsize,
                                            struct cache_blk_t *blk,
                                            tick_t now),
-            unsigned int hit_latency)  /* latency in cycles for a hit */
+            unsigned int hit_latency /* latency in cycles for a hit */,
+          int processID)
 {
+  key_t key = SHM_L2_KEY;
   struct cache_t *cp;
   struct cache_blk_t *blk;
   int i, j, bindex;
+  int shmid;

   /* check all cache parameters */
   if (nsets <= 0)
@@ -296,13 +340,30 @@
     fatal("must␣specify␣miss/replacement␣functions");

   /* allocate the cache structure */
-  cp = (struct cache_t *)
-    calloc(1, sizeof(struct cache_t) + (nsets-1)*sizeof(struct cache_set_t)
    );
-  if (!cp)
-     fatal("out␣of␣virtual␣memory");
-
+   switch(processID) {
+     case -1 :
+       cp = (struct cache_t *)
+         calloc(1, sizeof(struct cache_t) + (nsets-1)*sizeof(struct
    cache_set_t));
+       if (!cp)
+         fatal("out␣of␣virtual␣memory:");
+       break;
+     case 0 :
+       shmid = create_shmem(key, sizeof(struct cache_t) + (nsets-1)*sizeof(
    struct cache_set_t));
+       cp = shmem_attatch(shmid);
+       memset(cp,0,sizeof(struct cache_t) + (nsets-1)*sizeof(struct
    cache_set_t));
+       break;
+     default :
+       shmid = get_shmem(key, sizeof(struct cache_t) + (nsets-1)*sizeof(
    struct cache_set_t));
+       cp = (struct cache_t *) shmem_attatch(shmid);
+       break;
+   }
+
    /* initialize user parameters */
-   cp->name = mystrdup(name);
+   cp->name = NULL;
+   if(processID==-1)
+       cp->name = mystrdup(name);
+   else
+     cp__name = mystrdup(name);
    cp->nsets = nsets;
    cp->bsize = bsize;
    cp->balloc = balloc;
@@ -312,7 +373,11 @@
    cp->hit_latency = hit_latency;

    /* miss/replacement functions */
-   cp->blk_access_fn = blk_access_fn;
+   cp->blk_access_fn = NULL;
+   if(processID==-1)
+       cp->blk_access_fn = blk_access_fn;
+   else
+       cp__blk_access_fn = blk_access_fn;

    /* compute derived parameters */
    cp->hsize = CACHE_HIGHLY_ASSOC(cp) ? (assoc >> 2) : 0;
@@ -325,12 +390,13 @@
    cp->bus_free = 0;

    /* print derived parameters during debug */
-   debug("%s:␣cp->hsize␣␣␣␣␣=␣%d", cp->name, cp->hsize);
-   debug("%s:␣cp->blk_mask␣␣=␣0x%08x", cp->name, cp->blk_mask);
-   debug("%s:␣cp->set_shift␣=␣%d", cp->name, cp->set_shift);
-   debug("%s:␣cp->set_mask␣␣=␣0x%08x", cp->name, cp->set_mask);
-   debug("%s:␣cp->tag_shift␣=␣%d", cp->name, cp->tag_shift);
-   debug("%s:␣cp->tag_mask␣␣=␣0x%08x", cp->name, cp->tag_mask);
+   //debug("%s: cp->hsize      = %d", cp__name, cp->hsize);
+   //debug("%s: cp->blk_mask   = 0x%08x", cp__name, cp->blk_mask);
+   //debug("%s: cp->set_shift  = %d", cp__name, cp->set_shift);
+   //debug("%s: cp->set_mask   = 0x%08x", cp__name, cp->set_mask);
+   //debug("%s: cp->tag_shift  = %d", cp__name, cp->tag_shift);
+   //debug("%s: cp->tag_mask   = 0x%08x", cp__name, cp->tag_mask);
+

    /* initialize cache stats */
    cp->hits = 0;
@@ -338,32 +404,83 @@
    cp->replacements = 0;
    cp->writebacks = 0;
    cp->invalidations = 0;
+   cp->prefetches = 0;
+   cp->prefetches_ok = 0;
+
+   if(processID!=-1) {
+        cp->hits = -1; //Flag that this counters are not used
+        cp__hits = 0;
+        cp__misses = 0;
+        cp__replacements = 0;
+        cp__writebacks = 0;
+        cp__invalidations = 0;
+        cp__prefetches = 0;
+        cp__prefetches_ok = 0;
+   }

    /* blow away the last block accessed */
    cp->last_tagset = 0;
    cp->last_blk = NULL;

-   /* allocate data blocks */
-   cp->data = (byte_t *)calloc(nsets * assoc,
-                               sizeof(struct cache_blk_t) +
-                               (cp->balloc ? (bsize*sizeof(byte_t)) : 0));
-   if (!cp->data)
-     fatal("out␣of␣virtual␣memory");
+   key++;
+
+   switch(processID) {
+     case -1 :
+       /* allocate data blocks */
+       cp->data = (byte_t *)calloc(nsets * assoc,
+                  sizeof(struct cache_blk_t) +
+                  (cp->balloc ? (bsize*sizeof(byte_t)) : 0));
+       if (!cp->data)
+         fatal("out␣of␣virtual␣memory:b");
+       break;
+    case 0 :
+      shmid = create_shmem(key, nsets * assoc * (
+                     sizeof(struct cache_blk_t) +
+                     (cp->balloc ? (bsize*sizeof(byte_t)) : 0)));
+
+      cp->data = (byte_t *)shmem_attatch(shmid);
+       memset(cp->data,0,sizeof(struct cache_blk_t) +
+             (cp->balloc ? (bsize*sizeof(byte_t)) : 0));
+
+
+       break;
+   default :
+     shmid = get_shmem(key, nsets * assoc * (sizeof(struct cache_blk_t) +
+                       (cp->balloc ? (bsize*sizeof(byte_t)) : 0)));
+     cp->data = (byte_t *) shmem_attatch(shmid);
+     break;
+ }
+

+
    /* slice up the data blocks */
    for (bindex=0,i=0; i<nsets; i++)
      {
        cp->sets[i].way_head = NULL;
        cp->sets[i].way_tail = NULL;
-       /* get a hash table, if needed */
-       if (cp->hsize)
-         {
-           cp->sets[i].hash =
-             (struct cache_blk_t **)calloc(cp->hsize,
-                                           sizeof(struct cache_blk_t *));
+       /* get a hash table, if needed */
+    if (cp->hsize) {
+      switch(processID) {
+        case -1 :
+               cp->sets[i].hash = (struct cache_blk_t **)calloc(cp->hsize,
+                                                             sizeof(struct
    cache_blk_t *));
+            break;
+        case 0 :
+          key ++;
+          shmid = create_shmem(key, cp->hsize * (sizeof(struct cache_blk_t
    *)));
+          cp->sets[i].hash = (struct cache_blk_t **)shmem_attatch(shmid);
+          memset(cp->sets[i].hash ,0,cp->hsize * (sizeof(struct cache_blk_t
    *)));
+          break;
+        default :
+          key ++;
+          shmid = get_shmem(key, cp->hsize * (sizeof(struct cache_blk_t *)))
    ;
+          cp->sets[i].hash = (struct cache_blk_t **)shmem_attatch(shmid);
+          break;
+      }
           if (!cp->sets[i].hash)
-            fatal("out␣of␣virtual␣memory");
+            fatal("out␣of␣virtual␣memory:c");
         }
+
        /* NOTE: all the blocks in a set *must* be allocated contiguously,
           otherwise, block accesses through SET->BLKS will fail (used
           during random replacement selection) */
@@ -376,13 +493,34 @@
          /* locate next cache block */
          blk = CACHE_BINDEX(cp, cp->data, bindex);
          bindex++;
-
+
          /* invalidate new cache block */
          blk->status = 0;
          blk->tag = 0;
          blk->ready = 0;
-         blk->user_data = (usize != 0
-                           ? (byte_t *)calloc(usize, sizeof(byte_t)) : NULL
    );
+     blk->prefetched = 0;
+     blk->procNo = -1;
+
+     if(usize==0)
+       blk->user_data = NULL;
+     else {
+       switch(processID) {
+         case -1 :
+           blk->user_data =  (byte_t *)calloc(usize, sizeof(byte_t));
+           break;
+        case 0 :
+           key ++;
+           shmid = create_shmem(key, usize * sizeof(byte_t));
+           blk->user_data = (byte_t *)shmem_attatch(shmid);
+           memset(blk->user_data ,0,usize * sizeof(byte_t));
+           break;
+        default :
+           key ++;
+           shmid = get_shmem(key, usize * sizeof(byte_t));
+           blk->user_data = (byte_t *)shmem_attatch(shmid);
+           break;
+       }
+     }

          /* insert cache block into set hash table */
          if (cp->hsize)
@@ -396,8 +534,8 @@
          cp->sets[i].way_head = blk;
          if (!cp->sets[i].way_tail)
            cp->sets[i].way_tail = blk;
-       }
-     }
+         }
+  }
   return cp;
 }

@@ -413,17 +551,50 @@
   }
 }

+
+
+/* create and initialize a general cache structure */
+struct cache_t *                        /* pointer to cache created */
+cache_create(char *name,                 /* name of the cache */
+             int nsets,                  /* total number of sets in cache */
+             int bsize,                  /* block (line) size of cache */
+             int balloc,                 /* allocate data space for blocks?
     */
+             int usize,                  /* size of user data to alloc w/blks
     */
+             int assoc,                  /* associativity of cache */
+             enum cache_policy policy,   /* replacement policy w/in sets */
+             /* block access function, see description w/in struct cache def
     */
+             unsigned int (*blk_access_fn)(enum mem_cmd cmd,
+                                           md_addr_t baddr, int bsize,
+                                           struct cache_blk_t *blk,
+                                           tick_t now),
+             unsigned int hit_latency)/* latency in cycles for a hit */
+{ return cache_create_shared(name,              /* name of the cache */
+             nsets,                      /* total number of sets in cache */
+             bsize,                      /* block (line) size of cache */
+             balloc,            /* allocate data space for blocks? */
+             usize,                      /* size of user data to alloc w/blks
     */
+             assoc,                      /* associativity of cache */
+             policy,    /* replacement policy w/in sets */
+             /* block access function, see description w/in struct cache def
     */
+             blk_access_fn,
+             hit_latency,
+           -1);/* latency in cycles for a hit */
+}
+
+
 /* print cache configuration */
 void
 cache_config(struct cache_t *cp,        /* cache instance */
             FILE *stream)               /* output stream */
 {
+  char * threadSafeName = cp->name;
+  if(cp->name==NULL) threadSafeName = cp__name;
   fprintf(stream,
           "cache:␣%s:␣%d␣sets,␣%d␣byte␣blocks,␣%d␣bytes␣user␣data/block\n",
-          cp->name, cp->nsets, cp->bsize, cp->usize);
+          threadSafeName , cp->nsets, cp->bsize, cp->usize);
   fprintf(stream,
           "cache:␣%s:␣%d-way,␣`%s'␣replacement␣policy,␣write-back\n",
-          cp->name, cp->assoc,
+          threadSafeName , cp->assoc,
           cp->policy == LRU ? "LRU"
           : cp->policy == Random ? "Random"
           : cp->policy == FIFO ? "FIFO"
@@ -436,13 +607,54 @@
                 struct stat_sdb_t *sdb) /* stats database */
 {
   char buf[512], buf1[512], *name;
```

```
+   char * threadSafeName = cp->name;
+   if(cp->name==NULL) threadSafeName = cp__name;

    /* get a name for this cache */
-   if (!cp->name || !cp->name[0])
+   if (!threadSafeName || !threadSafeName [0])
      name = "<unknown>";
    else
-     name = cp->name;
+     name = threadSafeName ;

+   if(cp->hits==-1) {
+   sprintf(buf, "%s.accesses", name);
+   sprintf(buf1, "%s.hits␣+␣%s.misses", name, name);
+   stat_reg_formula(sdb, buf, "total␣number␣of␣accesses", buf1, "%12.0f");
+   sprintf(buf, "%s.hits", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣hits", &cp__hits , 0, NULL);
+   sprintf(buf, "%s.misses", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣misses", &cp__misses , 0, NULL
    );
+   sprintf(buf, "%s.replacements", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣replacements",
+                  &cp__replacements , 0, NULL);
+   sprintf(buf, "%s.writebacks", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣writebacks",
+                  &cp__writebacks , 0, NULL);
+   sprintf(buf, "%s.invalidations", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣invalidations",
+                  &cp__invalidations , 0, NULL);
+   sprintf(buf, "%s.miss_rate", name);
+   sprintf(buf1, "%s.misses␣/␣%s.accesses", name, name);
+   stat_reg_formula(sdb, buf, "miss␣rate␣(i.e.,␣misses/ref)", buf1, NULL);
+   sprintf(buf, "%s.repl_rate", name);
+   sprintf(buf1, "%s.replacements␣/␣%s.accesses", name, name);
+   stat_reg_formula(sdb, buf, "replacement␣rate␣(i.e.,␣repls/ref)", buf1,
    NULL);
+   sprintf(buf, "%s.wb_rate", name);
+   sprintf(buf1, "%s.writebacks␣/␣%s.accesses", name, name);
+   stat_reg_formula(sdb, buf, "writeback␣rate␣(i.e.,␣wrbks/ref)", buf1, NULL
    );
+   sprintf(buf, "%s.inv_rate", name);
+   sprintf(buf1, "%s.invalidations␣/␣%s.accesses", name, name);
+   stat_reg_formula(sdb, buf, "invalidation␣rate␣(i.e.,␣invs/ref)", buf1,
    NULL);
+   /* Prefetching statistics */
+   sprintf(buf, "%s.prefetches", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣prefetches", &cp__prefetches ,
+        0, NULL);
+   sprintf(buf, "%s.prefetches_ok", name);
+   stat_reg_counter(sdb, buf, "total␣number␣of␣sucessfull␣prefetches",
+        &cp__prefetches_ok , 0, NULL);
+   }
+   else
+{
    sprintf(buf, "%s.accesses", name);
    sprintf(buf1, "%s.hits␣+␣%s.misses", name, name);
```

```
   stat_reg_formula(sdb, buf, "total␣number␣of␣accesses", buf1, "%12.0f");
@@ -471,6 +683,14 @@
   sprintf(buf, "%s.inv_rate", name);
   sprintf(buf1, "%s.invalidations␣/␣%s.accesses", name, name);
   stat_reg_formula(sdb, buf, "invalidation␣rate␣(i.e.,␣invs/ref)", buf1,
       NULL);
+  /* Prefetching statistics */
+  sprintf(buf, "%s.prefetches", name);
+  stat_reg_counter(sdb, buf, "total␣number␣of␣prefetches", &cp->prefetches,
+        0, NULL);
+  sprintf(buf, "%s.prefetches_ok", name);
+  stat_reg_counter(sdb, buf, "total␣number␣of␣sucessfull␣prefetches",
+        &cp->prefetches_ok, 0, NULL);
+  }
 }

 /* print cache stats */
@@ -479,16 +699,35 @@
             FILE *stream)                  /* output stream */
 {
   double sum = (double)(cp->hits + cp->misses);
+  if(cp->hits==-1)
+        sum = (double)(cp__hits + cp__misses);
+  char * threadSafeName = cp->name;
+  if(cp->name==NULL) threadSafeName = cp__name;

+  if(cp->hits==-1) {
   fprintf(stream,
          "cache:␣%s:␣%.0f␣hits␣%.0f␣misses␣%.0f␣repls␣%.0f␣invalidations\n"
            ,
-          cp->name, (double)cp->hits, (double)cp->misses,
+          threadSafeName , (double)cp__hits, (double)cp__misses,
+          (double)cp__replacements, (double)cp__invalidations);
+  fprintf(stream,
+          "cache:␣%s:␣miss␣rate=%f␣␣repl␣rate=%f␣␣invalidation␣rate=%f\n",
+          threadSafeName ,
+          (double)cp__misses/sum, (double)(double)cp__replacements/sum,
+          (double)cp__invalidations/sum);
+  }
+  else
+        {
+  fprintf(stream,
+          "cache:␣%s:␣%.0f␣hits␣%.0f␣misses␣%.0f␣repls␣%.0f␣invalidations\n"
    ,
+          threadSafeName , (double)cp->hits, (double)cp->misses,
          (double)cp->replacements, (double)cp->invalidations);
   fprintf(stream,
          "cache:␣%s:␣miss␣rate=%f␣␣repl␣rate=%f␣␣invalidation␣rate=%f\n",
-          cp->name,
+          threadSafeName ,
          (double)cp->misses/sum, (double)(double)cp->replacements/sum,
          (double)cp->invalidations/sum);
+  }
+
 }
 /* access a cache, perform a CMD operation on cache CP at address ADDR,
@@ -513,6 +752,15 @@
   struct cache_blk_t *blk, *repl;
   int lat = 0;

+  /* Increment prefetch counter */
+  if (cmd == Prefetch) {
+    if (cp->hits == -1) {
+      cp__prefetches++;
+    } else {
+      cp->prefetches++;
+    }
+  }
+
   /* default replacement address */
   if (repl_addr)
     *repl_addr = 0;
@@ -526,7 +774,6 @@
      ((addr + (nbytes - 1)) > ((addr & ~cp->blk_mask) + (cp->bsize - 1)))
          */
   if ((addr + nbytes) > ((addr & ~cp->blk_mask) + cp->bsize))
     fatal("cache:␣access␣error:␣access␣spans␣block,␣addr␣0x%08x", addr);
-
   /* permissions are checked on cache misses */

   /* check for a fast hit: access to same block */
@@ -546,8 +793,9 @@
            blk;
            blk=blk->hash_next)
        {
-          if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
+     if (blk->tag == tag && (blk->status & CACHE_BLK_VALID) && (blk->procNo
    ==my_cpuid)) {
             goto cache_hit;
+     }
        }
     }
   else
@@ -557,15 +805,17 @@
            blk;
            blk=blk->way_next)
        {
-          if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
+     if (blk->tag == tag && (blk->status & CACHE_BLK_VALID) && (blk->procNo
    ==my_cpuid))
             goto cache_hit;
        }
     }

   /* cache block not found */
-
   /* **MISS** */
-  cp->misses++;
+  if(cp->hits==-1)
+      cp__misses++;
+  else
+        cp->misses++;

   /* select the appropriate block to replace, and re-link this entry to
      the appropriate place in the way list */
@@ -596,7 +846,10 @@
   /* write back replaced block data */
   if (repl->status & CACHE_BLK_VALID)
     {
-      cp->replacements++;
+      if(cp->hits==-1)
+           cp__replacements++;
+      else
+            cp->replacements++;

       if (repl_addr)
       *repl_addr = CACHE_MK_BADDR(cp, repl->tag, set);
@@ -613,20 +866,34 @@
       if (repl->status & CACHE_BLK_DIRTY)
        {
          /* write back the cache block */
-         cp->writebacks++;
-         lat += cp->blk_access_fn(Write,
+       if(cp->hits==-1)
+            cp__writebacks++;
+       else
+            cp->writebacks++;
+       if(cp->blk_access_fn==NULL)
+         lat += cp__blk_access_fn(Write,
                                       CACHE_MK_BADDR(cp, repl->tag, set),
                                       cp->bsize, repl, now+lat);
+       else
+       lat += cp->blk_access_fn(Write,
+                     CACHE_MK_BADDR(cp, repl->tag, set),
+                     cp->bsize, repl, now+lat);
+
       }
     }

   /* update block tags */
+  repl->procNo = my_cpuid;
   repl->tag = tag;
   repl->status = CACHE_BLK_VALID;       /* dirty bit set on update */

   /* read data block */
-  lat += cp->blk_access_fn(Read, CACHE_BADDR(cp, addr), cp->bsize,
-                           repl, now+lat);
+  if(cp->blk_access_fn==NULL)
+       lat += cp__blk_access_fn(Read, CACHE_BADDR(cp, addr), cp->bsize,
+                               repl, now+lat);
+   else
+       lat += cp->blk_access_fn(Read, CACHE_BADDR(cp, addr), cp->bsize,
+                               repl, now+lat);

   /* copy data out of cache block */
   if (cp->balloc)
@@ -638,6 +905,12 @@
```

```
    if (cmd == Write)
      repl->status |= CACHE_BLK_DIRTY;

+   /* Update prefetch status */
+   if (cmd == Prefetch)
+     repl->prefetched = 1;
+   else
+     repl->prefetched = 0;
+
    /* get user block data, if requested and it exists */
    if (udata)
      *udata = repl->user_data;
@@ -656,7 +929,21 @@
  cache_hit: /* slow hit handler */

    /* **HIT** */
-   cp->hits++;
+   if(cp->hits==-1)
+         cp__hits++;
+   else
+      cp->hits++;
+
+
+   if (blk->prefetched == 1) {
+     if (cp->hits == -1) {
+       cp__prefetches_ok++;
+     } else {
+       cp->prefetches_ok++;
+     }
+     blk->prefetched = 0;
+   }
+

    /* copy data out of cache block, if block exists */
    if (cp->balloc)
@@ -691,8 +978,21 @@
  cache_fast_hit: /* fast hit handler */

    /* **FAST HIT** */
-   cp->hits++;
+   if(cp->hits==-1)
+         cp__hits++;
+   else
+      cp->hits++;
+
+   if (blk->prefetched == 1) {
+     if (cp->hits==-1) {
+       cp__prefetches_ok++;
+     } else {
+       cp->prefetches_ok++;
+     }
+     blk->prefetched = 0;
+   }

+
    /* copy data out of cache block, if block exists */
    if (cp->balloc)
      {
@@ -741,7 +1041,7 @@
           blk;
           blk=blk->hash_next)
      {
-       if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
+       if (blk->tag == tag && (blk->status & CACHE_BLK_VALID) && (blk->
    procNo==my_cpuid))
          return TRUE;
      }
    }
@@ -752,7 +1052,7 @@
           blk;
           blk=blk->way_next)
      {
-       if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
+       if (blk->tag == tag && (blk->status & CACHE_BLK_VALID) && (blk->
    procNo==my_cpuid))
          return TRUE;
      }
    }
@@ -780,16 +1080,29 @@
          {
            if (blk->status & CACHE_BLK_VALID)
              {
-               cp->invalidations++;
+              if(cp->hits==-1)
+                   cp__invalidations++;
+              else
+                 cp->invalidations++;
+
               blk->status &= ~CACHE_BLK_VALID;

               if (blk->status & CACHE_BLK_DIRTY)
                 {
                   /* write back the invalidated block */
-                  cp->writebacks++;
-                  lat += cp->blk_access_fn(Write,
+                  if(cp->hits==-1)
+                       cp__writebacks++;
+                  else
+                       cp->writebacks++;
+                  if(cp->blk_access_fn==NULL)
+                  lat += cp__blk_access_fn(Write,
                                            CACHE_MK_BADDR(cp, blk->tag, i),
                                            cp->bsize, blk, now+lat);
+                  else
+                  lat += cp->blk_access_fn(Write,
+                                           CACHE_MK_BADDR(cp, blk->tag, i),
+                                           cp->bsize, blk, now+lat);
+
                 }
             }
         }
@@ -848,10 +1161,19 @@
```

```
        if (blk->status & CACHE_BLK_DIRTY)
         {
           /* write back the invalidated block */
-            cp->writebacks++;
-          lat += cp->blk_access_fn(Write,
-                                   CACHE_MK_BADDR(cp, blk->tag, set),
-                                   cp->bsize, blk, now+lat);
+        if(cp->hits==-1)
+              cp__writebacks++;
+        else
+              cp->writebacks++;
+          if(cp->blk_access_fn==NULL)
+                lat += cp__blk_access_fn(Write,
+                                        CACHE_MK_BADDR(cp, blk->tag, set),
+                                        cp->bsize, blk, now+lat);
+          else
+              lat += cp->blk_access_fn(Write,
+                            CACHE_MK_BADDR(cp, blk->tag, set),
+                            cp->bsize, blk, now+lat);
+
         }
        /* move this block to tail of the way (LRU) list */
        update_way_list(&cp->sets[set], blk, Tail);
```

F.7 Dram.c

Listing F.7: Dram.c - Unified diff against Uniprocessor version

```
--- ../simplesim-3.0/dram.c        2006-06-01 23:10:03.000000000 +0200
+++ ../../../felles-svn/project/branches/grannas_prefetch/dram.c
    2006-05-31 13:55:19.000000000 +0200
@@ -16,6 +16,7 @@
 #include "misc.h"
 #include "machine.h"
 #include "dram.h"
+#include "shared.h"

 /* Global circular buffer pointer */

@@ -23,17 +24,21 @@

 /* This function creates the dram subsystem */

-dram_system_t *create_dram(int number_of_channels, int size_of_block, int
    page_size, int control_time, int core_time, int data_time, int
    trace_interval) {
+dram_system_t *create_dram(int number_of_channels, int size_of_block, int
    page_size, int control_time, int core_time, int data_time, int
    trace_interval, int procid) {
   dram_system_t *dram_system;
-
-  /* Allocate memory for the structure */
+  int dram_id;
+  int channels_id;
+  int available_id;
+  int last_procid;
+  int i;
+  int circbuffer_id;

   dram_system = calloc(1, sizeof(dram_system_t));
   if (dram_system == 0) {
     printf("Could␣not␣allocate␣memory␣for␣DRAM␣model.\n");
     exit(1);
   }
-
+
   /* Set the values given as parameters */
   dram_system->num_channels = number_of_channels;
   dram_system->block_size = size_of_block;
@@ -50,25 +55,44 @@
   dram_system->page_hits = 0;

   /* Create the arrays */
+  if (procid == 0) {
+     channels_id = create_shmem(SHM_DRAM_KEY+1, number_of_channels * sizeof
    (tick_t));
+     available_id = create_shmem(SHM_DRAM_KEY+2, number_of_channels *
    sizeof(md_addr_t));
+     last_procid = create_shmem(SHM_DRAM_KEY+3, number_of_channels * sizeof
    (int));
```

```
+      circbuffer_id = create_shmem(SHM_DRAM_KEY+4, (CIRC_BUFFER_SIZE +1) *
    sizeof(int));
+    } else {
+      channels_id = get_shmem(SHM_DRAM_KEY+1, number_of_channels * sizeof(
    tick_t));
+      available_id = get_shmem(SHM_DRAM_KEY+2, number_of_channels * sizeof(
    md_addr_t));
+      last_procid = get_shmem(SHM_DRAM_KEY+3, number_of_channels * sizeof(
    int));
+      circbuffer_id = get_shmem(SHM_DRAM_KEY+4, (CIRC_BUFFER_SIZE +1) *
    sizeof(int));
+    }
+
+   dram_system->ready_channels = shmem_attatch(channels_id);
+   dram_system->last_address = shmem_attatch(available_id);
+   dram_system->last_proc = shmem_attatch(last_procid);
+
+   circbuffer = shmem_attatch(circbuffer_id);
+
    /* If the number of channels is 0 then do not create anything */

-   if (number_of_channels > 0) {
-     dram_system->ready_channels = calloc(number_of_channels, sizeof(tick_t)
    );
-     if (dram_system->ready_channels == 0) {
-       printf("Error␣creating␣ready␣channels.\n");
-       exit(1);
-     }
-     dram_system->last_address = calloc(number_of_channels, sizeof(md_addr_t
    ));
-     if (dram_system->last_address == 0) {
-       printf("Error␣creating␣Address␣array.\n");
-       exit(1);
-     }
-   }
-   /* Create the circular buffer as usual */
-   circbuffer = calloc(CIRC_BUFFER_SIZE +1, sizeof(int));
-   if (circbuffer == 0) {
-     printf("Error␣creating␣Circular␣buffer.\n");
-     exit(1);
+   if (procid == 0) {
+     if (number_of_channels > 0) {
+       if (dram_system->ready_channels == 0) {
+         printf("Error␣creating␣ready␣channels.\n");
+         exit(1);
+       }
+       for (i=0; i< number_of_channels; i++) {
+         dram_system->ready_channels[i] = 0;
+       }
+       if (dram_system->last_address == 0) {
+         printf("Error␣creating␣Address␣array.\n");
+         exit(1);
+       }
+       for (i=0; i< number_of_channels; i++) {
+         dram_system->last_address[i] = 0;
+         dram_system->last_proc[i] = -1;
+       }
+     }
   }
   return(dram_system);
 }
@@ -78,6 +102,7 @@
 void free_dram(dram_system_t *dram_system) {
   free(dram_system->last_address);
   free(dram_system->ready_channels);
+  free(dram_system->last_proc);
   free(dram_system);
 }

@@ -123,7 +148,7 @@
  * It returns the access time in number of ticks (due to compability issues
     )
  */

-unsigned int access_dram(dram_system_t *dram_system, md_addr_t block, int
    bsize, tick_t now) {
+unsigned int access_dram(dram_system_t *dram_system, md_addr_t block, int
    bsize, int procid, tick_t now) {
   int dram_bank;      /* The bank in use, calculated based on the address */
   int latency;        /* The calculated latency - in ticks */
   int control_time;   /* Time required to transfer a control word to DRAM */
@@ -144,14 +169,14 @@
   dram_system->accesses++;

   /* Update the circular buffer */
-
-  circbuffer[circbuffer[0]+1] = now - dram_system->ready_channels[dram_bank
    ];
+
+  circbuffer[circbuffer[0]+1] = now - dram_system->ready_channels[dram_bank
    ];
   circbuffer[0] = (circbuffer[0] + 1) % CIRC_BUFFER_SIZE;

   /* If the DRAM chip is free */
   if (dram_system->ready_channels[dram_bank] < now) {
     /*Check if we hit an open page */
-    if (is_same_page(dram_system, block, dram_system->last_address[
    dram_bank])) {
+    if ((is_same_page(dram_system, block, dram_system->last_address[
    dram_bank])) && (dram_system->last_proc[dram_bank] == procid)) {
       /* We hit an open page, transfer time is reduced. */
       dram_system->page_hits++;
       latency = control_time + data_time;
@@ -183,6 +208,7 @@
   /* Commit changes to the data structure */
   dram_system->ready_channels[dram_bank] = now + latency;
   dram_system->last_address[dram_bank] = block;
+  dram_system->last_proc[dram_bank] = procid;

   /* Update Statistics */
   dram_system->latency += latency;
@@ -246,3 +272,4 @@
```

```
    }
    return (total / CIRC_BUFFER_SIZE);
  }
+
```