NTNU

Innovation and Creativity

Improving the Performance of Parallel Applications in Chip Multiprocessors with Architectural Techniques

Magnus Jahre

Master of Science in Computer Science
Submission date: July 2007
Supervisor: Lasse Natvig, IDI

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

Problem Description

Chip Multiprocessors (CMPs) are becoming increasingly popular, both in industry and academia. However, most applications are still single-threaded. The paradox is that these applications will not experience improved performance when run on a CMP platform. In fact, the performance is often worse due to competition for shared resources. Consequently, the only way to achieve the performance potential of CMPs is to run parallel applications.

The candidate must investigate the performance of communication intensive multi-threaded workloads on a CMP platform. The result of this investigation should be the identification of performance bottlenecks. Furthermore, the candidate should propose and evaluate architectural techniques that alleviate these performance issues.

The proposed techniques should be evaluated with the M5 simulator. In addition, both multi-threaded and multi-programmed workloads should be simulated. The multi-threaded workloads can be used to explore the merits of the proposed technique, while multi-programmed workloads can be used for sensitivity analysis. It is advisable to use the multi-threaded SPLASH-2 benchmark suite to investigate communication effects, and programs from the SPEC-2000 benchmark suite to create multi-programmed workloads.

Assignment given: 13. September 2006
Supervisor: Lasse Natvig, IDI

Abstract

Chip Multiprocessors (CMPs) or *multi-core architectures* are a new class of processor architectures. Here, multiple processing cores are placed on the same physical chip. To reach the performance potential of these architectures with a single application, it must be multi-threaded. In these applications, the processing cores cooperate to solve a single task, and this requires a large amount of inter-processor communication in many cases. Consequently, CMPs need to support this communication in an efficient manner.

To investigate inter-processor communication in CMPs, a good understanding of the state-of-the-art of CMP design options, interconnect network design and cache coherence protocol solutions is required. Furthermore, a good computer architecture simulator is needed to evaluate both new and conventional architectural solutions. The M5 simulator [BDH$^+$06] is used for this purpose and has been extended with a generic split transaction bus, a crossbar based on the IBM Power 5 crossbar [KZT05], a butterfly network and an ideal interconnect. The unrealistic ideal interconnect provides an upper bound on the performance improvement available from enhancing the interconnect. In addition, a directory-based coherence protocol proposed by Stenström has been implemented [Ste89].

The performance of 2-, 4- and 8-core CMPs with crossbar and bus interconnects, private L1 caches and shared L2 caches is investigated. The bus and the crossbar are the conventional ways of implementing the L1 to L2 cache interconnect. These configurations have been evaluated with multiprogrammed workloads from the SPEC2000 benchmark suite [SPEa] and parallel, scientific benchmarks from the SPLASH-2 benchmark suite [WOT$^+$95]. With multiprogrammed workloads, the crossbar interconnect configurations perform nearly as well as a configuration with an ideal interconnect. However, the performance of the crossbar CMPs is similar to the performance of the bus CMPs when there is intensive L1 to L1 cache communication. The reason is limited L1 to L1 bandwidth. The bus CMPs experience a severe performance degradation with some benchmarks for all processor counts and workload classes.

A butterfly interconnect is proposed to alleviate the L1 to L1 communication bottleneck. The butterfly CMP performs on average 3.9 times better than the bus CMP and 3.8 times better than the crossbar CMP when there are 8 processor cores. These numbers are based on the performance of the *WaterNSquared*, *Raytrace*, *Radix* and *LUNoncontig* benchmarks. The reason is that the other SPLASH-2 benchmarks had issues with the M5 thread implementation for these configurations. For the multiprogrammed workloads, the butterfly CMPs are a bit slower than the crossbar CMPs.

Contents

List of Figures

List of Tables

List of Abbreviations

ACK	Acknowledgement
CAS	Column Access Strobe
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip Multiprocessor
CMT	Chip Multithreading
CPU	Central Processing Unit
flit	flow control digit
IDI	Department of Computer and Information Science, NTNU
ILP	Instruction-Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architectures
ITRS	International Technology Roadmap for Semiconductors
KB	Kilobyte
LRU	Least Recently Used
MB	Megabyte
MIN	Multistage Interconnection Network
MPSoC	Multi-Processor System on Chip
MSHR	Miss Status Holding Register
MSI	Modified, Shared, Invalid
NACK	Negative Acknowledgement
NCAR	NTNU Computer Architecture Research Group
NUCA	Non-Uniform Cache Access
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
SoC	System on Chip
SPEC	Standard Performance Evaluation Corporation

SPLASH	Stanford Parallel Applications for Shared Memory
SRAM	Static Random Access Memory
SW	Switch
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism
VC	Virtual Channel

Chapter 1

Introduction

Chip Multiprocessors (CMPs) or *multi-core architectures* are a new class of high-performance processor architectures. Here, multiple processing cores are placed on one physical chip, and the result is a single-chip multiprocessor. To realise the performance potential of these architectures, it is important to support efficient inter-processor communication. This report investigates architectural techniques for providing this much needed capability.

Figure 1.1 shows the high-level CMP architecture which is the main focus this report. This architecture is known as a shared-cache CMP and is one of many ways to design a CMP. Other possible architectures will be discussed when necessary. Since the task at hand is complex, it is helpful to focus on one high-level architecture.

There are three main reasons for focusing on shared-cache CMPs:

- In this work, the most important reason is that the architecture enables very fast communication between the processor cores as communication can be carried out with L1 to L1 cache data transfers. In other CMP architectures, this communication might need to access the off-chip memory bus or go via both processors' L2 caches. Both options add considerable delay.

- The recent, commercial Intel Core Duo dual-core processor is a shared-cache CMP [GMNR06]. Consequently, a major commercial actor has made a considerable investment into this high-level architecture.

- As more cores are added to the chip, the off-chip memory bus can become a bottleneck. Consequently, this shared resource should not be used more than necessary. An important part in achieving this is to use the on-chip L2 cache efficiently. A shared cache will in most cases result in better chip-wide utilisation than per-core private L2 caches.

This chapter has the following outline:

- First, section 1.1 discusses the assignment and formulates the tasks that must be carried out in order to answer it. Furthermore, it explains in which part of the report the tasks are answered.
- Then, section 1.2 discusses the main contributions of this work.
- Finally, the structure of the report is presented in section 1.3.

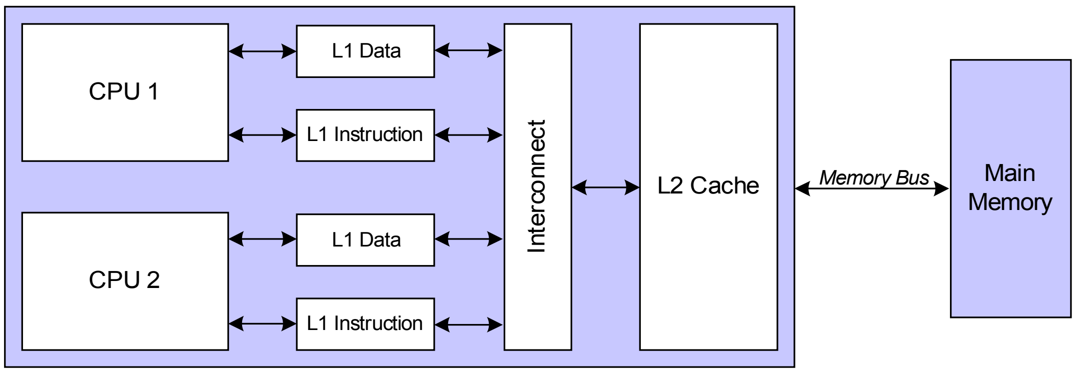


Figure 1.1: A Shared-Cache CMP

1.1 Assignment Interpretation

This section discusses the assignment text and divides it into subtasks. In addition, it highlights where in the report the subtasks are answered. Consequently, this section clarifies the relationship between the assignment text and the report.

The assignment consists of the following tasks:

T1 Investigate the performance of communication intensive multi-threaded benchmarks.

T2 Identify performance bottlenecks.

T3 Propose techniques that alleviate the bottlenecks.

Task T1 is answered by the experiments discussed in chapter 6. Here, the SPLASH-2 benchmark suite [WOT$^+$95] is used to investigate the performance of parallel programs on 2-, 4- and 8-core shared-cache CMPs. Furthermore, the results are supported by the multiprogrammed workload experiments with programs from the SPEC2000 benchmark suite [SPEa] which are discussed in chapter 5.

This investigation has identified a number of bottlenecks:

- Most importantly, efficient L1 to L1 communication is needed for parallel programs. The state-of-the-art crossbar interconnect simulated in this work uses a shared bus for L1 to L1 traffic. This crossbar is based on the crossbar used in the IBM Power 5 CMP [KZT05]. The results in chapter 6 shows that congestion in this bus severely limits performance for a number of parallel programs.
- For multiprogrammed workloads, the off-chip memory bus become congested in some cases and this limits performance.
- Lastly, the number of misses that can be serviced simultaneously in the L1 cache is a bottleneck for the processor simulated in this report. This bottleneck has been investigated further by us in a different work [JN07].

These bottlenecks demonstrate that task T2 is answered. Since this work focuses on CMP communication, the L1 to L1 communication bottleneck is prioritised. A butterfly interconnect is proposed because it enables efficient communication both between the L1 caches and between the L1 caches and the L2 cache. Although the hardware cost of the butterfly is somewhat higher than the cost of the simulated crossbar, it is less than the cost of a full crossbar. In this context, the term full crossbar is used to describe an interconnect where all nodes have a direct connection to all other nodes. The butterfly interconnect results are discussed in chapter 7 and show that the butterfly is very successful in alleviating the L1 to L1 communication bottleneck.

In addition to these tasks, the assignment text makes two suggestions:

- The experiments should be carried out on the M5 simulator [BDH$^+$06].
- The multi-threaded SPLASH-2 [WOT$^+$95] and multi-programmed SPEC2000 [SPEa] benchmark suites should be used.

Although these suggestions are not absolute demands, it is probably advantageous to follow them. The M5 simulator is a recent, feature-rich computer architecture simulator and a considerable improvement compared to the SimpleScalar simulator [ALE02] previously used at *NTNU Computer Architecture Research Group (NCAR)*. For instance, M5 accurately models finite miss status buffers and an L1 to L2 interconnect. In SimpleScalar, these resources have an infinite capacity. Sadly, M5 does have some issues with its thread implementation in system call emulation mode, but these were not known at the time the simulator was chosen. These limitations will be discussed in detail in section 4.1.2. All in all, M5's advantages outweigh its disadvantages.

The SPLASH-2 and SPEC2000 benchmark suites are a good match for the M5 simulator. Since the M5 simulator uses the Alpha instruction set, they are both available as precompiled binaries. This is a great advantage as compiling benchmarks is a non-negligible amount of work.

In addition to the tasks specified in the assignment, the interconnect performance of multi-programmed workloads is investigated. The reason is that this is a simpler task because it does not require a cache coherence protocol. Consequently, some insights can be gained earlier in the work. Hopefully, this leads to a better understanding of the problem at hand than if the assignment text was followed to the letter.

Answering the tasks given in the assignment text within the time frame of master thesis is ambitious. There are a number of reasons for this. Firstly, the M5 simulator has not been used by the NCAR group earlier. Consequently, there is no help available locally on how to use and configure it. This is a challenge, as the M5 simulator is a complex piece of software. Secondly, M5 only supports a bus interconnect between the L1 and L2 caches. The M5 software architecture makes it difficult to add other interconnects. Lastly, it is preferable to simulate a directory cache coherence protocol as this gives full freedom in choosing the types of interconnects to investigate. Although it is considerably easier to implement a directory protocol in a simulator than in real hardware, it is still a challenge. The reason is that there are numerous rare race conditions that must be handled correctly.

1.2 Main Contributions

The main contributions in this work are:

C1 A study of the most important interconnects in a shared-cache CMP has been carried out.

C2 A good understanding of the M5 simulator has been gained. Furthermore, a split transaction bus, a crossbar, a butterfly, an ideal interconnect and a directory-based cache coherence protocol have been developed.

C3 A problem with the M5 system call emulation thread implementation that makes using this library for research less attractive than previously known was identified. This problem was reported to the M5 development team.

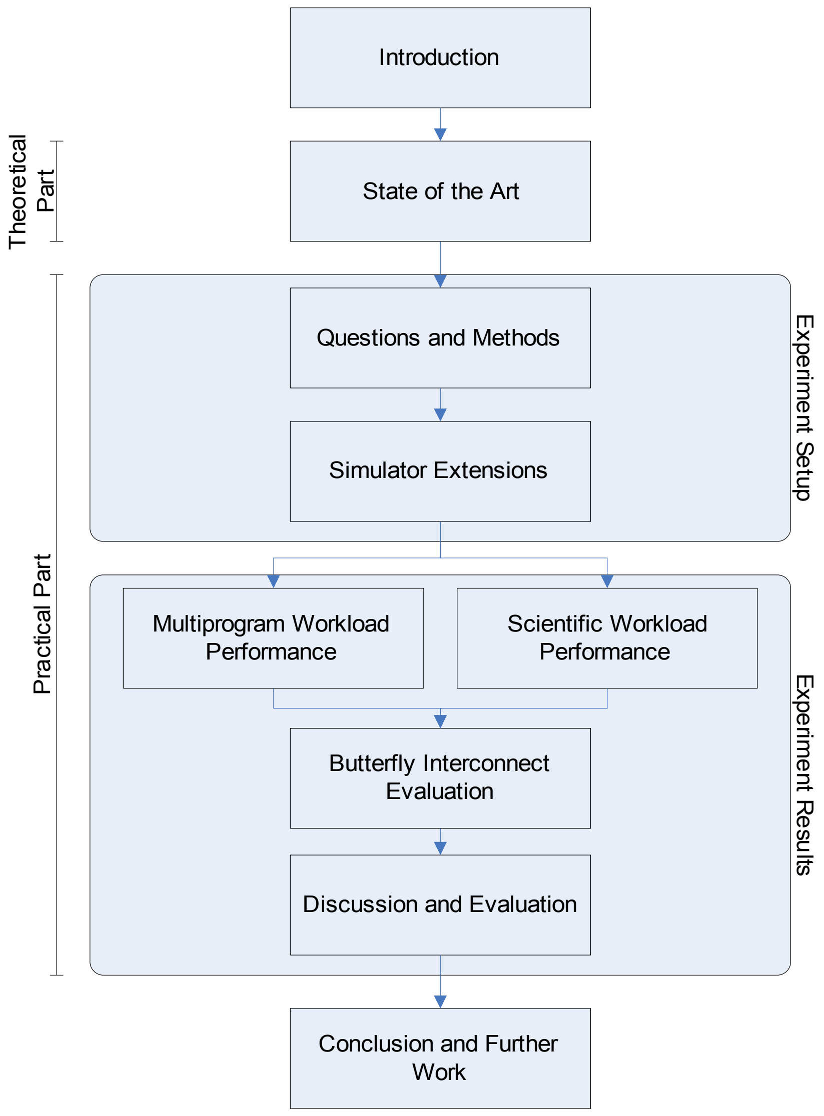

Introduction
State of the Art
Theoretical Part
Questions and Methods
Simulator Extensions
Experiment Setup
Practical Part
Multiprogram Workload Performance
Scientific Workload Performance
Butterfly Interconnect Evaluation
Discussion and Evaluation
Experiment Results
Conclusion and Further Work

Figure 1.2: Report Outline

C4 A review of the state-of-the-art of CMP design with a particular focus on interconnect and cache coherence solutions is given.

Contribution C1 is the answer to the assignment text and therefore the most important one for this work. However, contributions C2 and C3 are important for the NCAR group. Detailed knowledge about a state-of-the-art computer architecture simulator is clearly an asset for future research. Lastly, contribution C4 is also helpful for future research. In addition, it contains discussions of the most important, recently published works on interconnects in CMPs.

1.3 Report Outline

Figure 1.2 is a graphical depiction of the structure of this report. The report consists of two main parts: a theoretical part and a practical part. The practical part is further subdivided into the experimental setup and experimental results parts. The theoretical part comes first and is covered by the State-of-the-art chapter (chapter 2). This chapter contains an introduction to CMP design options, a discussion of CMP on-chip interconnects and an introduction to cache coherence protocols in a CMP context.

The practical part starts with the Questions and Methods chapter (chapter 3). This chapter states the research questions that form the basis for the practical part of the report. In addition, the baseline CMP architecture used in the simulations is discussed. A general introduction to the M5 simulator is given in the Simulator Extensions chapter (chapter 4). The implementation of the new interconnects and the directory protocol is also described here. Furthermore, the problems with the M5 system call emulation thread library are discussed.

There are three chapters describing the simulation results. First, the Multiprogrammed Workload Performance chapter (chapter 5) discusses the simulation results from the experiments with the bus, crossbar and ideal interconnects and multiprogrammed workloads created with SPEC2000 benchmarks [SPEa]. Then, the Scientific Workload Performance chapter (chapter 6) discusses the results obtained when using the SPLASH-2 benchmark suite [WOT$^+$95]. Finally, the Butterfly Interconnect Evaluation chapter (chapter 7) evaluates the butterfly interconnect with both multiprogrammed and scientific workloads. The butterfly interconnect was chosen because it should alleviate the L1 to L1 cache bandwidth bottleneck discovered in chapter 6.

The two final chapters wrap up the report. First, the Discussion and Evaluation chapter (chapter 8) answers the research questions. Furthermore, it discusses possible threats to the validity of the experimental results. Finally, the Conclusion chapter (chapter 9) concludes and gives indications for possible further work.

Chapter 2

State-of-the-art

It is necessary to have a good understanding of previously proposed techniques to be able to propose new ones. In this report, there is a need to understand CMP design possibilities, interconnects and cache coherence solutions. As mentioned in the introduction, this report focuses on shared L2 cache CMPs. Consequently, the techniques reviewed in this chapter will mainly be related to this high-level architecture. However, other architectures will be considered when it is appropriate.

This chapter has the following outline:

- Firstly, the CMP-design space is explored by discussing academic CMP design proposals and commercial CMP implementations. Section 2.1 covers this point and does not focus on a specific CMP architecture.
- Section 2.2 discusses the interconnection network design options. The focus is on interconnect solutions for shared L2 cache CMPs.
- Finally, section 2.3 presents possible cache coherence protocol implementations. Again, shared L2 cache CMPs are the main focus.

2.1 Chip Multiprocessor Background

This section explores the design space of single-chip multiprocessors. In the academic community, these designs are commonly referred to as *Chip Multiprocessors (CMPs)*. However, commercial CMPs with two cores are known as dual-core processors. This has led to the adoption of the new term *multi-core architectures*. These terms are synonymous, and the CMP term will be used in the remainder of this report.

A CMP or multi-core architecture can be *homogeneous* or *heterogeneous* [KTJR05, KTR+04, KFJ+03]. This refers to how similar the different processing cores in the system are to each other. In a heterogeneous CMP, the processing cores have different properties. For instance, some cores can be simple in-order cores and other cores can be speculative and out-of-order. Here, applications that can efficiently utilise an out-of-order core are run on an out-of-order core. Applications with limited *Instruction-Level Parallelism (ILP)* can run on the in-order core as they experience only a small speed-up when run on an out-of-order core. Depending on the design constraints, this can result in lower power consumption, higher area efficiency or both.

Multi-Processor System on Chip (MPSoC) is an important class of heterogeneous single-chip multiprocessors [Wol04]. A *System on Chip (SoC)* is an embedded system where all or most components are placed on the same chip. If this system uses more than one general-purpose processor, it is referred to as a MPSoC. In this context, each processor is often given responsibility for a small number of tasks. Then, a processor that is well suited for these tasks is selected. The main difference between CMPs and MPSoCs is that MPSoCs are usually application specific. In addition, MPSoCs often have a strict power budget, area budget and real time demands. Furthermore, each processor often has an *Instruction Set Architectures (ISA)* that are different from the other processors in the system. Therefore, they differ from the general-purpose single-chip multiprocessors investigated in this report and will not be discussed further.

This report will focus on homogeneous CMPs. In other words, the CMP's processing cores are identical.

The rest of this section is organised as follows:

- Section 2.1.1 discusses the motivation for implementing CMPs.
- Then, section 2.1.2 discuss possible future limitations to the CMP architectures.
- Section 2.1.3 investigates possible high-level architectural choices when implementing a CMP. Both CMP architectures proposed in academia and commercial implementations are discussed.

2.1.1 CMP Motivation

The single-chip multiprocessor concept has been around for some time. For instance, Olukotun et al. [ONH$^+$96] proposed CMPs as a way to increase the processor clock rate in the late 90s. Although increasing the clock rate is not an important motivation any more, CMPs have recently gained popularity. A number of commercial vendors now produce CMPs [KST04, KAO05, AMD, GMNR06].

The recent popularity of CMPs is due to the following factors:

- Technology scaling has made placing multiple cores on one chip feasible.
- It has become increasingly difficult to improve performance by techniques that exploit *Instruction-Level Parallelism (ILP)* beyond what is common today.
- The power consumption of single-core, high-performance processors is high. Consequently, expensive packaging and noisy cooling solutions are needed. This limitation is known as the *power wall*.
- Processor performance has been improving at a faster rate than the main memory access time for over 20 years. Consequently, the performance difference between the processor and the memory is large and techniques that hide this latency are needed. This limitation is known as the *memory wall*.
- When designing a CMP, a processor core is designed once and reused as many times as there are cores on the chip. Furthermore, these cores can be simpler than their single-core counterparts. Consequently, CMPs facilitate design reuse and reduce time-to-market.

From 1987 to 2004 the performance of a microprocessor was increased by around 55% per year [HP07]. This high performance improvement was primarily due to two factors. Firstly, the number of transistors per chip increased as the production technology scaled down. Secondly, the clock frequency was increased faster than what was natural given the reduction in feature size.

In 2000, Agarwal et al. argued that the techniques used to exploit ILP in aggressive out-of-order

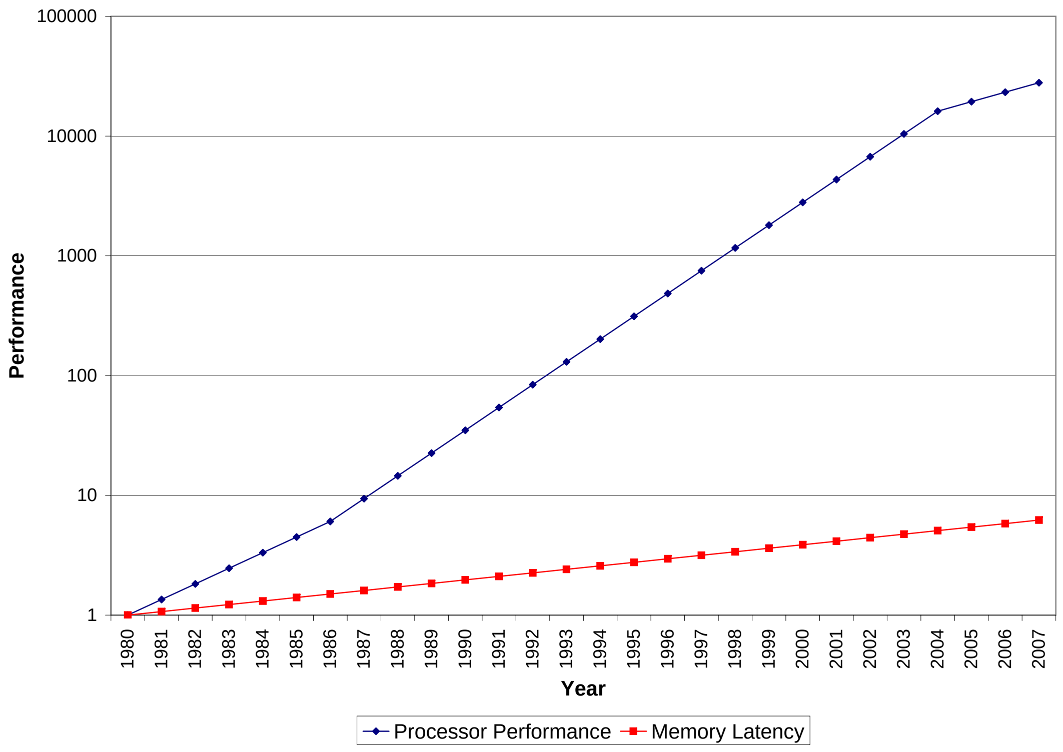


Figure 2.1: Processor and Memory Performance [HP07]

processors could only support an annual performance improvement of 12.5% [AHKB00]. The reason is that global wire delays grow faster than gate delay. Consequently, designers can choose between deeper pipelines, smaller structures or slower clocks. None of these design options will result in scalable performance. Therefore, there is a need to look into new architectures. In CMPs, the problem of wire delays is confined to the interconnect between cores. This point will be discussed further in section 2.1.2.

The power consumption of a processor must be controlled. However, achieving processor performance improvements by increasing the clock frequency results in higher power consumption. Furthermore, many techniques that exploit ILP are power hungry. The reason is that they do more work than is needed. This increases performance, but has a non-negligible power cost. In CMPs, *Thread-Level Parallelism (TLP)* can be used to achieve high performance. Consequently, more power efficient cores running on a lower clock frequency can be used. However, achieving a speed-up comes at the price of parallelising the application.

Figure 2.1 shows the relative difference between microprocessor and memory performance with 1980 as a baseline. According to Hennessy and Patterson [HP07], microprocessor performance increased by on average 25% per year from 1980 to 1986. Then, processor performance increased by 55% on average from 1987 to 2004. As discussed earlier, this growth is attributed to the advances in computer architecture as well as the scaling of technology. However, from 2004 the average performance improvement per year has been reduced to 20%. This is due to the reasons noted at the beginning of this section: power limitations, limitations to ILP-based techniques, long memory latency and high design cost.

The average improvements in memory latency are also shown in figure 2.1. This improvement has remained at 7% per year on average. However, the memory density has improved at roughly the same rate as the processor performance. Consequently, the challenge is to feed the powerful processors from a large and slow memory. The main technique to combat this problem is to create large on-chip caches. Currently, CMPs make this problem worse. When it is difficult to feed one processor, why should it be easier to feed for instance four processors? The next section will discuss this point further.

The last reason for introducing CMPs is that processors which are good at exploiting ILP, are very complex. In other words, they are expensive to design. In homogeneous CMPs, the processor core is made one time and then reused throughout the design. Consequently, CMPs make more business sense than conventional ILP-based processors.

2.1.2 Future Limitations to CMP Architectures

CMPs address a number of challenges that face microprocessor designers but not all. In addition, the physical limitations of the production technology apply regardless of architectural choices. Consequently, it is interesting to discuss potentially limiting factors for CMP designs.

The following potential limits to CMP performance will be discussed in this section:

- A single-threaded application will in some cases run slower on a CMP than on a comparable single-core processor
- Memory latency and off-chip bandwidth are constrained resources
- The latency of global wires does not scale down with technology

2.1.2.1 CMP Single Thread Performance

The great paradox of CMPs is that a single threaded program never runs faster on a CMP than on a single-core design with the same processor core. Sadly, it will in some cases run slower. The reason is that CMPs either share caches or have smaller per-core caches. In the shared cache case, different caches might compete for space in the same cache set or bank. This problem is known as *hot-sets* and *hot-banks* [SA05]. Consequently, the application is slowed down in unpredictable ways. In the private cache case, the private cache size is usually divided equally between the processors. If this result in the application's working set not fitting in the cache, the application is slowed down.

This problem can be solved by parallelising the application. However, single-threaded programs will probably be the norm for many years. Consequently, there is a need for techniques that make these programs run efficiently.

2.1.2.2 Memory Latency and Bandwidth

As mentioned in the previous section, there is a need to hide the memory latency in modern processors. This is done by using caches. Figure 2.2 shows growth trends for processor performance, memory density, pins per chip and memory latency. The performance numbers have been set to one in 2005 to make comparison of the trends easier. Processor performance (55% per year), memory latency (7% per year) and memory density (55% per year) are based on the growth trends used by Hennessy and Patterson [HP07]. The average pin count growth of 6.5% is based on the ITRS Roadmap [ITR06].

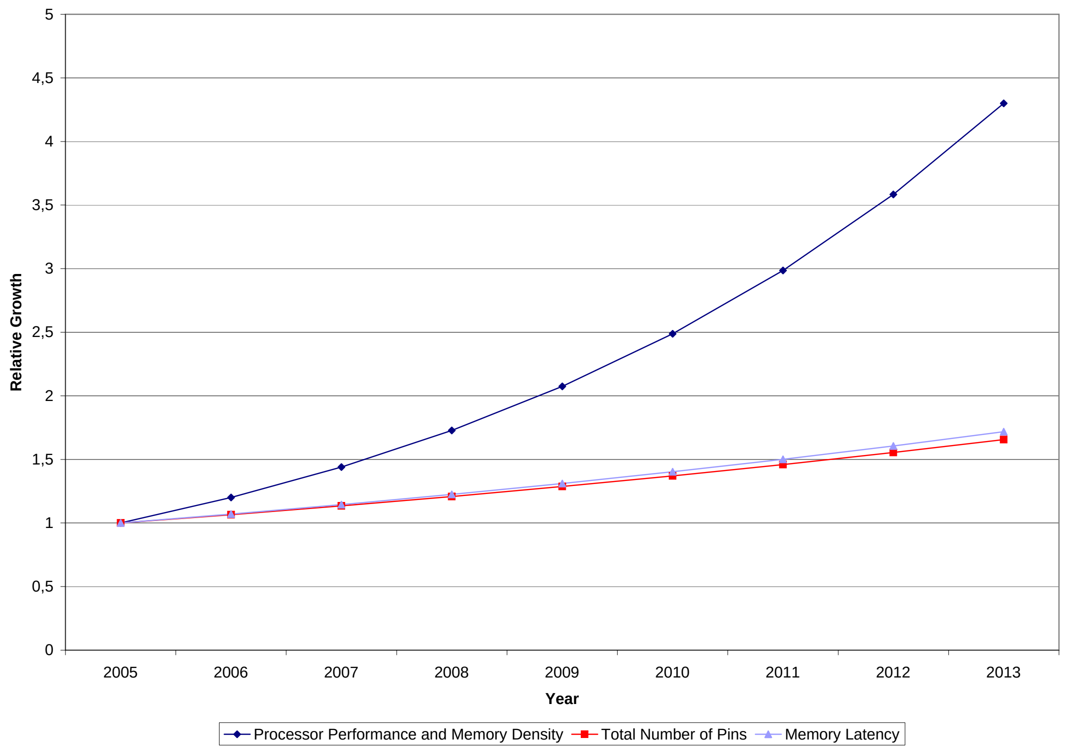


Figure 2.2: CMP Memory System Scalability

Figure 2.2 shows that processor performance grows faster than pin count and memory latency. The memory latency can be hidden by using caches. However, the memory bandwidth depends on the width and clock frequency of the memory bus. Figure 2.2 shows that the number of pins per chip is expected to grow at roughly the same rate as the memory latency. Although the clock frequency of the bus can be increased, off-chip bandwidth is likely to become a constrained resource [HBK01]. Consequently, a good CMP design must use the memory bus in an intelligent manner.

2.1.2.3 Delay of Global Wires

The delay of a wire depends on its resistance and its capacitance. Both these quantities depend on wire length. Consequently, the delay of a wire depends on its length. The delay of a gate also depends on its size, and a smaller gate has a lower delay than a larger gate. When the production technology improves, gates become smaller and the circuits become faster.

If the wire length is scaled with technology, the relative delay of a gate and the wire stays roughly the same [HHM99]. Figure 2.3 illustrates this point. However, the long wire connecting the two modules in the figure does not decrease in length when the technology is scaled down. Consequently, the delay of this wire is increased relative to gate delay. On the other hand, the length of the short wire decreases when technology is scaled. In other words, the delay of *global* wires is expected to increase relative to gate delay when technology scales down.

CMPs cope well with this challenge. As technology is scaled, the size of the individual core is decreased. The local wiring scales down with core size. However, inter-processor communication uses global wires. Consequently, this on-chip communication will become more expensive as

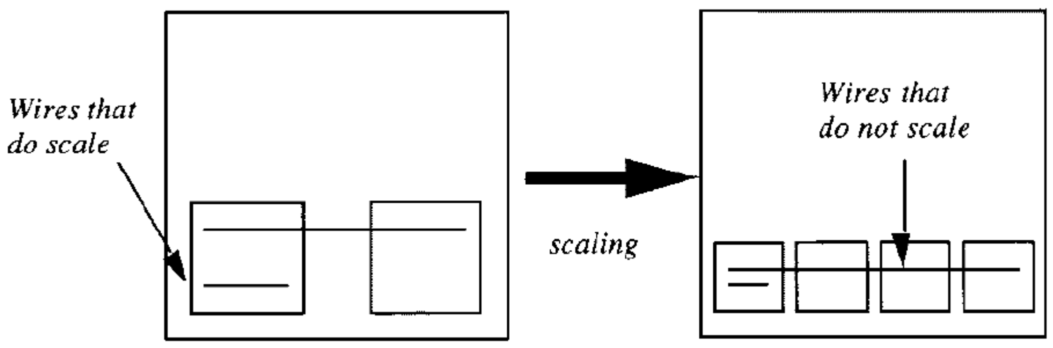


Figure 2.3: Wire Scaling and Technology Scaling (Reproduced from [HHM99])

technology scales down. CMP researchers should take this trend into account when proposing new techniques.

2.1.3 CMP Design Options

This section investigates the high-level architectural choices that can be made when designing a CMP. The placement and sharing status of the last-level cache is especially important. In this report, the last-level cache is the on-chip cache closest to memory. This section is based on the *Chip Multiprocessor (CMP)* discussion in my preliminary project [Jah06].

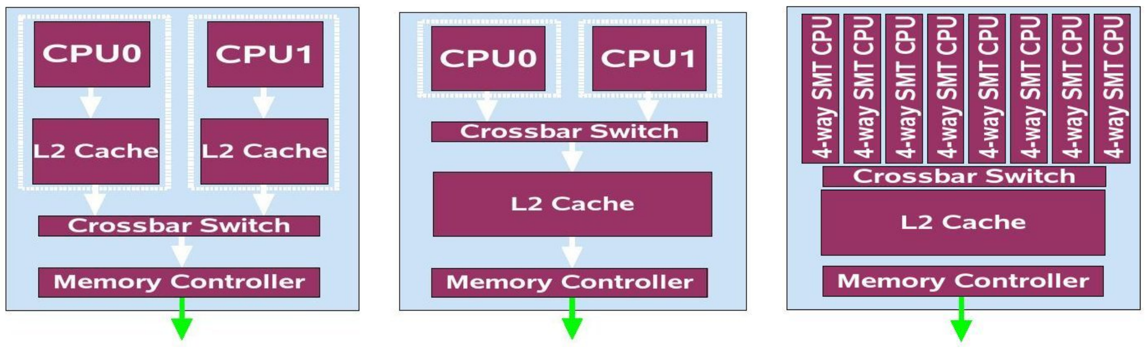
This section is organised as follows:

- Firstly, *conventional CMPs* are discussed. These CMPs are extensions of traditional single-core designs.
- *Tiled CMPs* are discussed next. Here, the processor core, caches and communication routers are allocated on a *tile*. This tile is then replicated throughout the chip.
- It is also possible to share some functional units between adjacent processing cores. This type of CMPs is called *Conjoined Core CMPs.*
- Recently, CMP-designs that distribute over several stacked wafers have been proposed. These are called *3D CMPs.*
- Lastly, a number of commercial CMP designs is discussed. Since the details of these designs often are closely guarded secrets, they will receive less attention than their academic counterparts.

2.1.3.1 Conventional CMPs

The term Conventional CMP does not appear in the literature. However, this important class of CMPs needs a name. They will be called Conventional CMPs in this report.

Conventional CMPs have evolved from single-core processors. According to Spracklen and Abraham [SA05], CMPs have so far gone through three generations. These generations are shown in figure 2.4. In the first generation of CMPs, ease of design was the primary constraint. Consequently, two nearly independent single-core designs where added to the same chip. As shown in figure 2.4(a), only the off-chip memory controller and memory bus is shared.



(a) First Generation (b) Second Generation (c) Third Generation

Figure 2.4: Chip Multiprocessor Generations (Reproduced from [SA05])

Figure 2.4(b) shows a second generation CMP-design. Here, the processing cores have been designed specifically for inclusion in a CMP. Furthermore, the L2 cache is shared. The move to the third generation is carried out by increasing the number of cores and adding *Simultaneous Multithreading (SMT)*. In SMT, instructions are fetched from more than one thread simultaneously. In this case, a new thread can start execution the clock cycle after the running thread encountered a long latency event. Spracklen and Abraham coined the term *Chip Multithreading (CMT)* for this approach [SA05].

A third generation CMP can achieve very high *throughput* when many threads are available. However, to fit many cores on a chip, the cores must be reasonably simple. Consequently, the execution time for a single thread can be quite long.

In server workloads, handling many threads efficiently is more important than the execution time of a single thread. Therefore, third generation CMPs are probably a good choice in these machines. Many threads are also available in a desktop machine. However, the execution time of a single thread does matter in this case. Therefore, a second generation CMP with good single-thread performance might be more appropriate. This highlights the important trade-off between throughput and single-thread performance.

The last-level cache in second and third generation CMPs can be private or shared. A private cache design has a lower latency because the cache is smaller than a shared one. Furthermore, conflicts between two cores are not possible as it is used by only one core.

However, there are two disadvantages. Firstly, global cache utilisation might be low. For example, the tread running on core A might only use a small amount of its cache space while the thread running on core B uses all its cache space. In this case, thread B would run faster with a shared cache because it could use the free space in A's cache. Secondly, inter-core communication is slow. The reason is that sharing is implemented between the last-level caches and memory. Consequently, inter-core communication uses the off-chip memory bus.

It is possible to achieve the advantages of both designs. In the shared case, the key idea is that cache blocks should be placed in cache banks that are physically close to the processor that uses this block. This reduces hit latency. Exposing this difference in delay between cache banks result in what Kim et al. refers to as a *Non-Uniform Cache Access (NUCA)* architecture [KBK02]. In the private case, the idea is that a processor core can borrow space in a neighbouring core's cache. This increases cache utilisation. Chishti et al. [CPV05], Chang and Sohi [CS06] as well as Dybdahl and Stenström [DS07] have proposed such architectures.

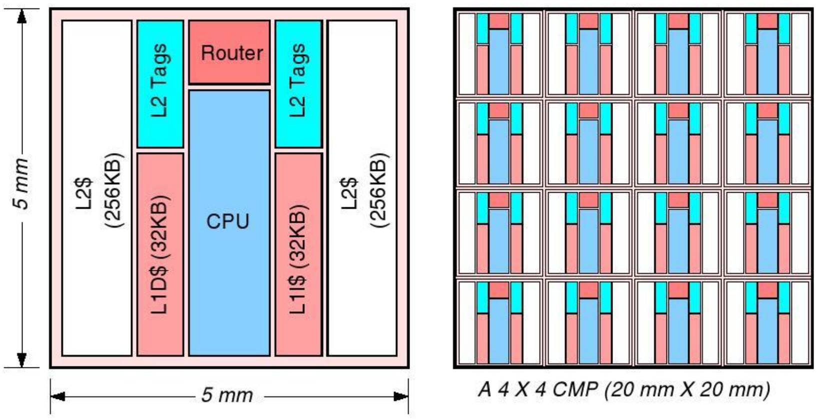


Figure 2.5: A Tiled Chip Multiprocessor (Reproduced from [ZA05])

2.1.3.2 Tiled CMPs

A tiled CMP is shown in figure 2.5. These CMPs are very similar to second or third generation CMPs with private caches. The main difference is that in tiled CMPs each tile is designed once and then replicated throughout the chip. This makes design and floor planning easier. However, there is no clean division in the terminology so the classification is somewhat overlapping.

In a plain tiled CMP architecture, each core has its private last-level cache. This might result in poor global cache utilisation. Again, this can be improved if a core is allowed to borrow space in a neighbouring core's cache. Zhang and Asanović' Victim Replication scheme makes this possible [ZA05]. Victim Replication is a way of implementing a shared cache from a number of per-tile private caches and differs from the previous techniques in that it is designed specifically for tiled CMPs. In this technique, a cache block is first brought into the L2 cache bank that is responsible for its address. Furthermore, the block is stored in the local L1 of the processor that requested it. When the cache block is evicted from the local L1 cache, a replica of the block is kept in the local L2 cache. In this way, the block migrates towards where it is used. Consequently, fast access time and good global cache utilisation is achieved.

2.1.3.3 Conjoined Core CMPs

Kumar et al. has proposed Conjoined Core CMPs [KJT04]. In this case, cores that are physically close to each other can share resources. It might be more area efficient to share these resources between cores if they are used rarely. However, the additional wiring complexity must be taken into account. The reason is that the wiring overhead can quickly outweigh any area benefits. Kumar et al. investigated the sharing of floating-point units, crossbar ports, data caches and instruction caches. They showed that processor core area can be substantially reduced with a small performance degradation.

2.1.3.4 3D Chip Multiprocessors

As feature size is decreased, the relative impact of interconnect delay grows as discussed in section 2.1.2. 3D CMPs, as shown in figure 2.6, have been proposed to reduce this problem. Here, multiple wafers are placed on top of each other and vertical vias provide inter-wafer

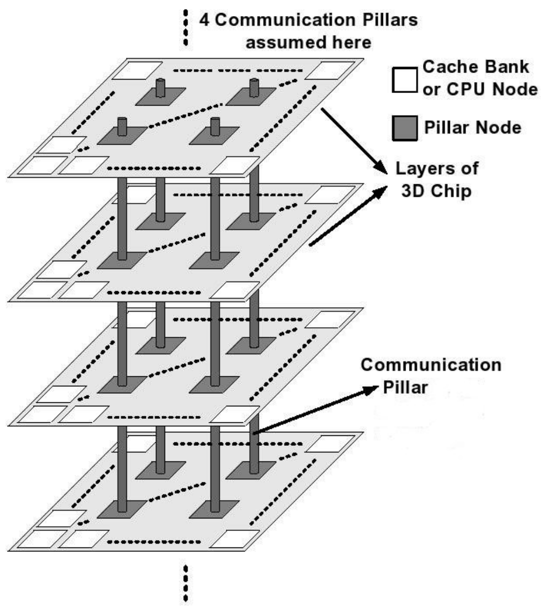


Figure 2.6: A 3D Chip Multiprocessor (Adapted from [LNR+06])

communication. Since the vertical connections are short, this creates the possibility of placing many cache banks close to each core. This reduces interconnect delay because the delay through a wire mainly depends on its length. Li et al. [LNR+06] proposed and evaluated this design option and report promising results. However, a number of implementation issues must be resolved before this type of CMPs can be implemented.

2.1.3.5 Commercial CMPs

As mentioned earlier, the information commercial companies publicly provide about their CMP implementations is in many cases intentionally vague. Consequently, this section will focus on relatively high level architectural choices.

A number of commercial companies sell CMPs:

- The Intel Core Duo processor is a shared L2 cache, two-core CMP [GMNR06].
- AMD Athlon 64 X2 is a two-core CMP with per-core private L2 caches [AMD].
- Another dual-core processor with a shared L2 cache is the IBM Power 5 processor [SKT+05, KST04]. Each core is two-way multithreaded.
- Sun Microsystems has implemented an 8-core CMP with a shared L2 cache called UltraSPARC T1 or Niagara [KAO05]. Here, each core is 4-way multithreaded. Recently, Sun has announced the 8-core Niagara 2 processor [McG06]. The cores in this processor are 8-way multithreaded so this processor can execute 64 threads simultaneously.

The processors are located at different solution points in the continuum between single-thread performance and throughput. For instance, the UltraSPARC T1 has a large throughput focus. The number of advanced out-of-order features must be limited when eight 4-way multithreaded cores are placed on a single chip. In fact, when a thread issues a multiply or divide instruction, the thread is suspended. Consequently, single-thread performance is likely to be low. However, high throughput is expected when a sufficient number of threads are available.

The Intel Core Duo and AMD Athlon 64 X2 processors emphasise single-thread performance. Here, powerful, out-of-order cores harvest ILP from the instruction stream. Consequently, a single-threaded application will run faster. The Power 5 processor is somewhat of a compromise as it has two, two-way multithreaded cores.

Another difference between the processors is whether the L2 cache should be shared or not. The AMD Athlon X2 processor has private L2 caches while the other processors have shared caches. This indicates that there is no agreement of what constitutes the best solution to this problem at this time.

2.2 Interconnect

As mentioned earlier, this section will focus on the interconnect between the private L1 caches and the shared L2 cache in a shared cache CMP. Consequently, the interconnected nodes are the L1 caches and the L2 cache banks. However, the interconnection networks are general and can be used in other CMP architectures as well.

Balfour and Dally investigated the design trade-offs in tiled CMP interconnection networks [BD06]. They focused on the mesh, concentrated mesh, torus, fat-tree and tapered fat-tree topologies. In addition, Kumar et al. [KZT05] has investigated bus, hierarchical bus and crossbar topologies. These works seem to be the most important works investigating interconnection networks in CMPs. Consequently, they will play a central part in this section.

The section follows this outline:

- Section 2.2.1 discusses how on-chip interconnects differ from off-chip interconnects and to what extent the large body of research carried out on interconnection networks can be reused in the CMP context.
- Section 2.2.2 discusses network topology. A topology is a description of how network nodes are connected to each other.
- Routing is the task of choosing how to get from one node to another and is discussed in section 2.2.3.
- Finally, section 2.2.4 discusses flow control. Flow control manages the allocation of resources to data that flow through the network.

2.2.1 On-Chip and Off-Chip Interconnects

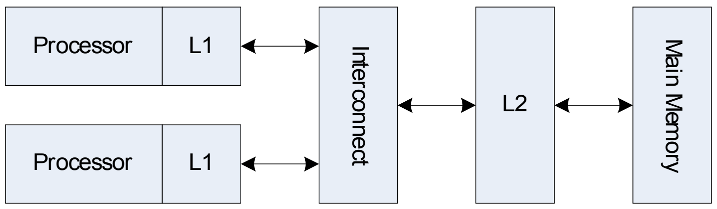
This section is primarily based on Dally's presentation at the "Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems" [Dal06]. Furthermore, a quick note on terminology is in order. As will be discussed in section 2.2.3, the main area consuming parts of a router are the buffers and switches. Consequently, when buffers and switches are discussed in this section, they refer to functional units within routers.

Off-chip interconnects have the following characteristics:

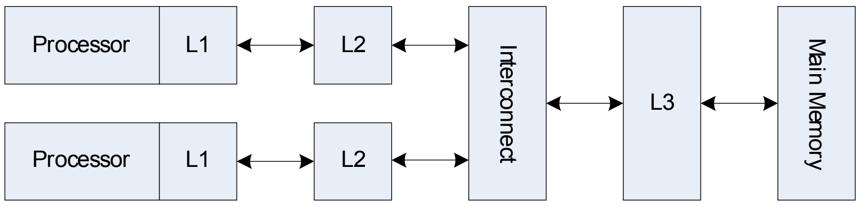
- The cost of the interconnect is determined by the number of transmission channels. This determines the number of pins used for the different chips, which connectors can be used, the number of cables and the number of optical units. Optical channels have lower latency than electrical channels but are more expensive. Consequently, they should only be used for channels where low latency is important.
- Network nodes are relatively far apart. Consequently, latency is high in general.

On-chip interconnects are characterised by:

- The cost of the interconnect is primarily the area used for buffers and switches.
- A lot of wires can easily be added. However, repeaters must be added to avoid signal decay. In addition, pipelining transfers makes it possible for the network to work at a clock frequency similar to the processor clock frequency. In this case, flip-flops or latches are inserted. It might be advantageous to add some additional hardware such that these devices can handle certain network functions.
- Network nodes can communicate in a few processor clock cycles. Consequently, the latency is considerably lower than for off-chip networks.



(a) Shared L2 Cache



(b) Private L2 and Shared L3 Cache

Figure 2.7: Possible Levels of Sharing in a CMP

- The properties of the channels, buffers and switches influence the performance of a given topology. Consequently, finding the most suitable interconnection network for a given design requires optimising all these components together.
- Power consumption is a first order design constraint.

From these lists, two effects are apparent. Firstly, routers are expensive and wires are cheap in on-chip networks. This differs from traditional multiprocessor networks. Consequently, network topologies with few routers and many wires are likely to perform well on CMPs. However, the size of the router depends on the width of the transmission channels. Therefore, wide channels also create large, expensive routers. Balfour and Dally [BD06] found that providing two independent networks gave good performance.

Moving communication closer to the processor creates the opportunity for fast interprocessor communication. However, it also creates additional design constraints. Even though the communication traffic depends on the application and is independent of where in the CMP communication is implemented, non-communication traffic is more frequent closer to the processor cores. This point is illustrated by figure 2.7 which shows two possible shared cache CMP designs. In figure 2.7(a), the interconnect has to handle all regular L1 cache misses in addition to the communication traffic created by the application. The pressure on the interconnect can be traded off against an increased communication latency by inserting private L2 caches as shown in figure 2.7(b). Since this cache can be larger than the L1 cache, the number of misses will be reduced. Consequently, the impact of interconnect latency on overall system performance is likely to be larger when a shared L2 solution is chosen since congestion is more likely to occur in this case.

As mentioned, a large amount of research has been carried out on interconnection networks in traditional multiprocessors. The discussion in this section highlights that this research needs to be taken into account when designing on-chip interconnects. However, the trade-offs will be different and there is probably still room for innovation.

Network-on-chip is a popular research topic in the embedded systems domain [BM02]. Here, a main concern is to create an optimised interconnection network for a given SoC design. In this case, the workloads and communication patterns are known. This enables radical optimisations

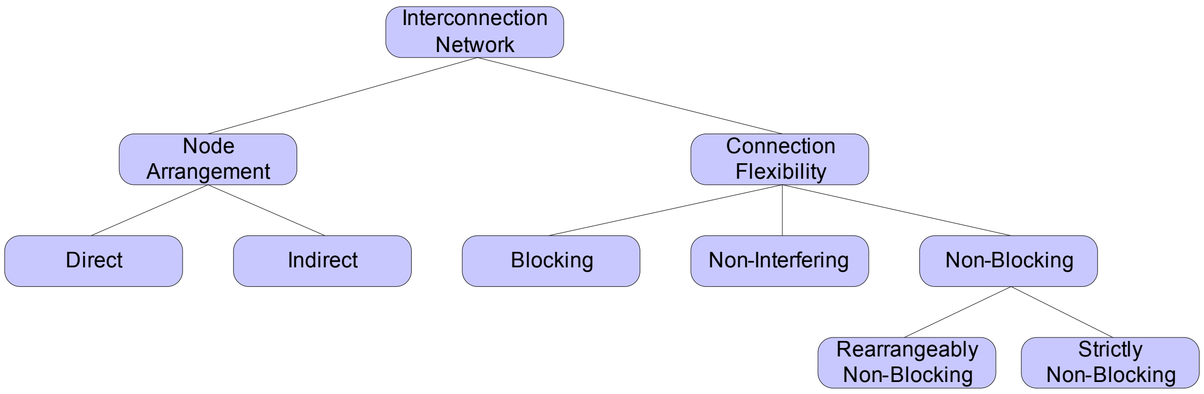


Figure 2.8: Interconnection Network Terminology

that can not be used in a general CMP interconnect. However, CMP interconnects and on-chip networks use similar building blocks like for instance power efficient routers. In other words, the components used are similar, but the embedded system designer often knows more about the workload.

2.2.2 Topology

This section discusses some of the possible topologies for an L1 cache to L2 cache interconnection network in a shared-cache CMP. In this case, the *nodes* of the network are the L1 caches and L2 banks and the *channels* are point-to-point links. A topology refers to how the channels and nodes of a network are laid out. Furthermore, a network can be *direct* or *indirect*. In a direct network, all channels run between network terminals. An indirect network has nodes that only perform a switching function and do not have a terminal associated with them. Indirect networks are also known as *multistage interconnection networks (MINs)*.

There is also a distinction between *blocking* and *non-blocking* networks. In a *non-blocking* network, a path can be formed between all network inputs and all network outputs without a conflict occurring. In this context, a conflict is a busy channel. If new connections can be set up incrementally without rerouting an existing connection for all input permutations, the network is said to be *strictly non-blocking*. Alternatively, a *rearrangeably non-blocking* network might reroute established connections to accommodate new ones. If conflicts can occur, the interconnection network is blocking.

According to Dally and Towels, creating non-blocking networks is overkill in packet switched networks [DT03]. The reason is that by allocating network resources in a good way, it is possible to guarantee that one flow does not deny the service of another flow for more than a short time period. Consequently, it is possible to place an upper bound on the delay. If the network meets these criteria, Dally and Towels refer to it as a *non-interfering* network.

The terminology introduced in this introduction is summarised in figure 2.8.

A frequently used term when describing topologies is *bisection bandwidth*. To understand this term, the notion of a *cut* must be understood first. A cut is a set of channels that partition the nodes of a network into two groups. A cut is a *bisection* if two conditions are met. Firstly, half of the nodes must be in one partition and the other half must be in the other partition. Secondly, half of the terminal nodes must be in one partition and the other half must be in the

Term	**Explanation**
Radix	The number of inputs or outputs to each switching node. In other words, if a switch has 2 inputs and 2 outputs, its radix is 2.
Throughput	The data rate in bits per second that the network accepts per input port.
Ideal Throughput	The throughput of the network with perfect routing and flow control. These concepts will be discussed in sections 2.2.3 and 2.2.4, respectively.
Maximum channel load	The load on the most heavily loaded channel in the network under a particular traffic pattern.
Head latency (T_h)	The time it takes for the head of the packet to traverse the network
Serialisation latency (T_s)	The time it takes for a packet of length L to traverse a channel with bandwidth b, i.e. $T_s = \frac{L}{b}$
Total Latency	$T = T_h + T_s = T_h + \frac{L}{b}$
Path diversity	The number of minimal paths between two nodes in the network

Table 2.1: Topology Terminology

other partition. Of course, the number of nodes might be odd so one partition can have one more node than the other. The *bisection bandwidth* is then the *minimum* bandwidth over all possible bisections of the network.

The interconnection network field of research is terminology-rich, and a detailed discussion of all terminology is beyond the scope of this report. However, table 2.1 describes a few additional terms.

2.2.2.1 Direct Networks

This section discusses some of the direct interconnect networks used in recent CMP publications. It has the following outline:

- First, the *shared bus* topology is discussed.
- Then, the *crossbar* topology is presented.
- Lastly, the *mesh*, *torus*, *hypercube* and *ring* topologies are discussed together because they are variations of a single connection scheme.

Shared Bus

A bus is a simple topology where all network nodes are connected to the same channel. An important reason for using a bus interconnect is that it is simple to construct. Furthermore, it makes it possible to use a snooping coherence protocol. This further simplifies the design. The main drawback is that it can become a performance bottleneck.

Kumar et al. have evaluated bus interconnects in a CMP context [KZT05]. Their CMP has private L2 caches and 4, 8 or 16 cores, and the bus design is shown in figure 2.9. Here, four processors and L2 caches are connected to the bus interconnect. Note that there is one *arbitration queue* and one *data queue* for each L2 cache. Naturally, there is only one *address arbiter* and one *data arbiter* in the system. The bus implemented in this work is considerably simpler than

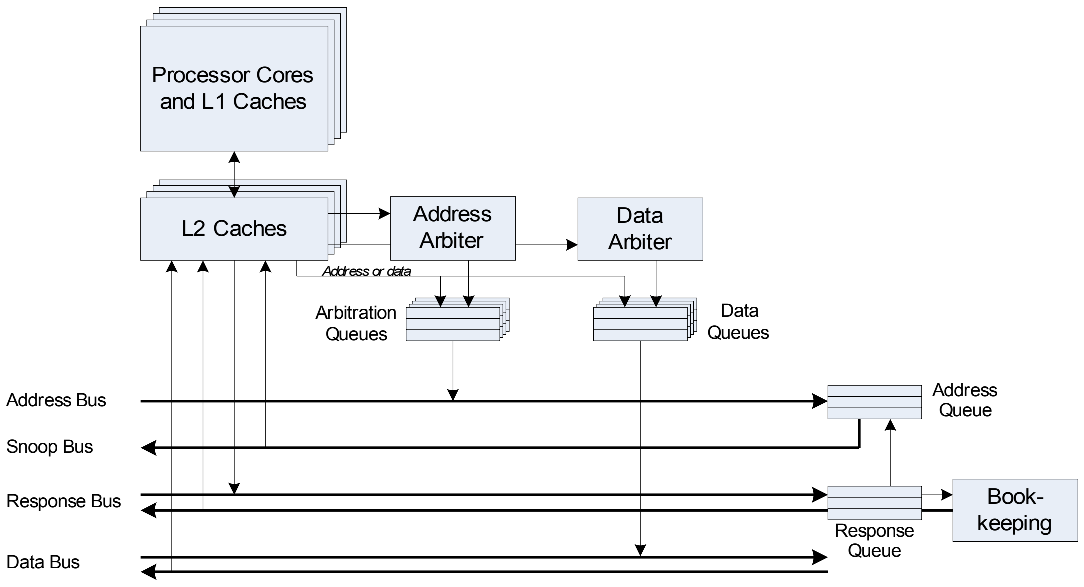


Figure 2.9: The Bus Interconnect from Kumar et al. [KZT05]

this design as this simplifies both the implementation and the result interpretation. However, this might lead to the performance of the bus being overestimated.

A typical read transaction on this bus proceeds as follows:

1. The requester tells the *address arbiter* that it wants to access the bus.
2. When it is granted access, it sends its request over the *address bus*.
3. The request arrives in the *address queue* and is sent over the *snoop bus* when this is free.
4. All nodes listen to the snoop bus and all nodes that have information about the cache block in question put a response on the *response bus* after a fixed delay. Then, a message is broadcasted over the response bus from the *bookkeeping* unit. This message takes into account the responses from each of the caches and informs the caches of which action they should take next. Examples of possible actions are data transfers and invalidations.
5. Finally, the responding node asks the *data arbiter* for access to the *data bus*. When access is granted, the data is sent to the original requester.

Both the address bus and the snooping bus are broadcast buses. Therefore, it is possible for the nodes to snoop on the address bus instead of the snoop bus. However, the snoop bus might be shared with other chips as the bus is located on the memory side of the last-level cache. The needed data might be cached on a different chip, and this chip is only allowed access to the snoop bus. In other words, the snoop bus is the point of serialisation. These broadcast buses have a significant delay. However, the bidirectional, pipelined data bus ensures that data can be transferred quickly when the administration tasks are finished.

Kirman et al. [KKD$^+$06] have evaluated the use of optical buses for private last-level cache CMPs. Optical interconnects provide low-latency, high bandwidth channels. Kirman et al. maintain that technological advances in CMOS compatible optical components make optical interconnects a potential replacement for electrical components around 2013. Obviously, there is a great deal of uncertainty associated with this number. As expected, optical buses outperform electrical buses in Kirman et al.'s evaluation.

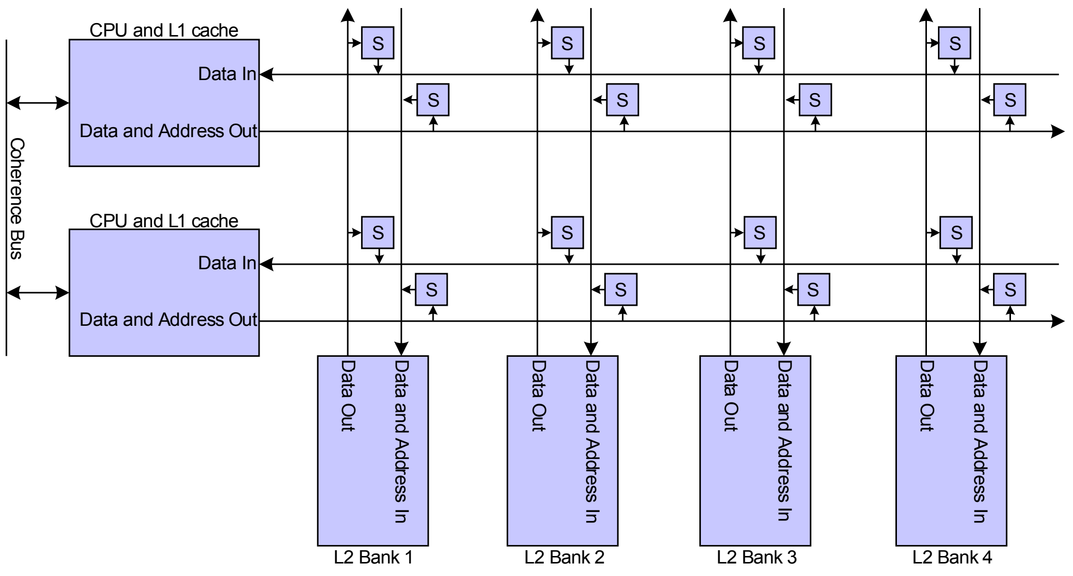


Figure 2.10: Crossbar Topology

Crossbar

A crossbar directly connects n inputs to m outputs. If $n = m$, the crossbar is *square*. Otherwise, it is *rectangular*. Furthermore, a crossbar is strictly non-blocking as every input can be connected to any output incrementally without influencing any other connections [DT03].

Crossbar interconnects are very popular in shared last-level cache CMPs. They are used in IBM's Power 4 and Power 5 as well as Sun's Niagara [KZT05, KAO05]. However, it is difficult to find precise technical information about these interconnects. Luckily, Kumar et al. evaluates a crossbar interconnect based on the interconnect used in the Power 4 and Power 5 processors [KZT05]. Their crossbar design is shown in figure 2.10. Here, there are data and address lines in L1 to L2 direction and data lines in the L2 to L1 direction. This crossbar differs slightly from the crossbar implemented in this work as will be discussed in chapters 3 and 4.

The crossbar shown in figure 2.10, is actually two crossbars. First, there is a rectangular crossbar with two inputs and four outputs in the CPU to L2 cache direction. In addition, there is a rectangular crossbar with four inputs and two outputs in the L2 to CPU direction. By closing the appropriate switch, any input can be connected to any output. Furthermore, all channels only send data in one direction as this makes pipelined data transfer possible. If the electrical drivers are sufficiently powerful, multicast can be enabled by closing more than one switch.

Of course, the high performance of a crossbar does not come for free. Since each network node is connected to all other nodes, there are n connections per node. If there are n nodes in the system, this gives a growth on the order of n^2. This theoretical analysis was validated experimentally by Kumar et al. They found that if the die size is kept constant, the area overhead of including a crossbar reduces overall performance. The reason is that the L2 cache size must be reduced, and this reduction hurts performance more than the performance gain by introducing cache sharing.

In a shared L2 cache CMP, the frequent case is L2 access. Therefore, this must be supported in an efficient manner. However, inter-processor communication requires transferring data between L1 caches. There are three ways of doing this. Firstly, the crossbar can be extended to allow L1 to L1 transfers as well. Secondly, data from one L1 to another can be sent via an L2 cache. The first solution is expensive in terms of area while the second option is slow. A compromise

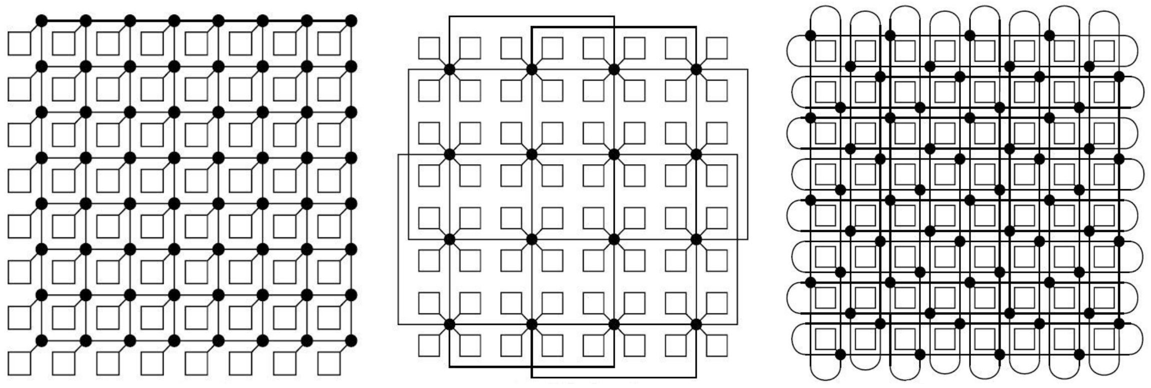

(a) Mesh (b) Concentrated Mesh (c) Torus

Figure 2.11: Mesh and Torus Topologies (Reproduced from [BD06])

is to add a bus between all L1 caches. This is the solution used in the Power 4 and Power 5.

Mesh and Torus

Mesh and torus networks are classes of a network family called *cubes*. Here, the network consists of $N = k^n$ nodes which are allocated into a n-dimensional grid with k nodes in each dimension. All nodes are connected to its nearest neighbours. If $n = 1$ the topology is known as a *ring* and if $k = 2$ the topology is called a *hypercube*. Otherwise, it is known as a *mesh* or a *torus*. The difference between these two is that a torus has connections from each node on the edge of the network to another edge node. In a mesh, these end nodes have fewer connections than nodes in the middle of the network. Furthermore, a ring is a 1-dimensional torus and a hypercube is a mesh where the node count is a power of 2 [DT03].

The mesh, concentrated mesh and torus topologies are shown in figure 2.11. Here, each square is a processor tile and each dot is a network router. The lines connecting the dots are channels. Balfour and Dally evaluated these topologies for a tiled CMP [BD06].

Figure 2.11(a) shows a mesh topology. Here, a network node that is not on the edge of the network is connected to all its neighbours. Edge nodes are connected to all available neighbours.

A torus topology is shown in figure 2.11(c). For nodes that are not on the edge of the network, the torus topology and the mesh topology are the same. However, in the torus case, edge nodes have the same number of connections as the internal nodes. Figure 2.11(c) shows a folded topology. In this case, the connections from the edge nodes do not go around the whole network, but to a neighbouring node. Here, all channels have the same length but the average channel length is longer.

Figure 2.11(b) shows a concentrated mesh topology. This is a variation on a mesh topology where four processors share a router. In addition, *express channels* are added for the edge nodes. These channels reduce the number of hops needed in the worst case for communication between an arbitrary pair of nodes.

Balfour and Dally introduced this topology for tiled-CMPs. They found that a concentrated mesh is superior to a regular mesh and a torus in terms of performance, power and area. The main reason is that an average transaction requires fewer hops when the concentrated mesh is used. In addition, the concentrated mesh perform better than the fat-tree and tapered fat-tree topologies discussed later in this section. The fat-tree and tapered fat-tree topologies have a low average hop count but their wiring complexity make them less desirable.

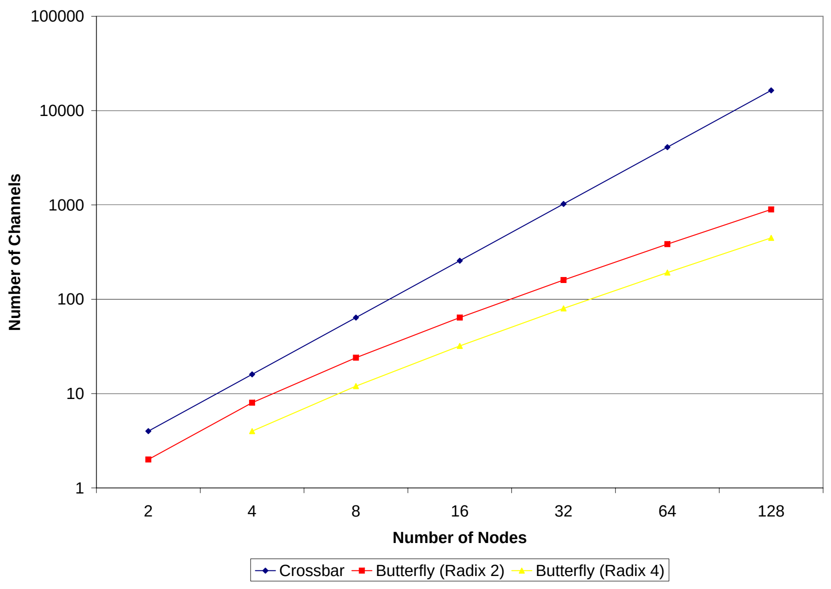


Figure 2.12: Number of Channels in a Butterfly and a Crossbar

Another reason for these good results is that Balfour and Dally was able to create two independent copies of the mesh and concentrated mesh networks while hiding the area overhead of the second network. The reason was that they had allocated a specific portion of the total chip area for interconnect use. Since the mesh and concentrated mesh use little area, they were able to create two networks in the allocated space. This area could have been used to create larger L2 caches, but Balfour and Dally did not pursue this.

It is unclear to what extent the findings of Balfour and Dally can be extended to shared L2 CMPs. The first reason is that the topologies have only been evaluated for tiled CMPs. In tiled CMPs, the L2 caches are private and co-located with the processors. This results in less pressure on the interconnect as there will be fewer misses in these L2 caches than in the private L1 caches in a shared L2 CMP. Secondly, only synthetic benchmarks where used in their evaluation. It is unclear to what extent these represent real workloads.

As mentioned earlier, a ring is a 1-dimensional torus. Marty and Hill has proposed an extended snooping cache coherence protocol that relies on the ordering properties of a ring interconnect [MH06]. Their solution is discussed in section 2.3.1. The main point here is that this simpler coherence solution makes a ring more attractive as the complexity of implementing a directory cache coherence protocol can be avoided.

2.2.2.2 Indirect Networks

The choice of topology for a given design does to some extent depend on how well the topology can be mapped to the components of the system. In this respect, meshes and tori map well to tiled CMPs. A shared last-level cache CMP has a less regular structure and mapping meshes and tori to this CMP type is more difficult. Consequently, it might be worthwhile to look into indirect networks.

In general, indirect networks provide all-to-all connectivity at a $O(n \log n)$ cost [DT03]. In comparison, realising this with a crossbar has a cost of $O(n^2)$. This relationship is illustrated

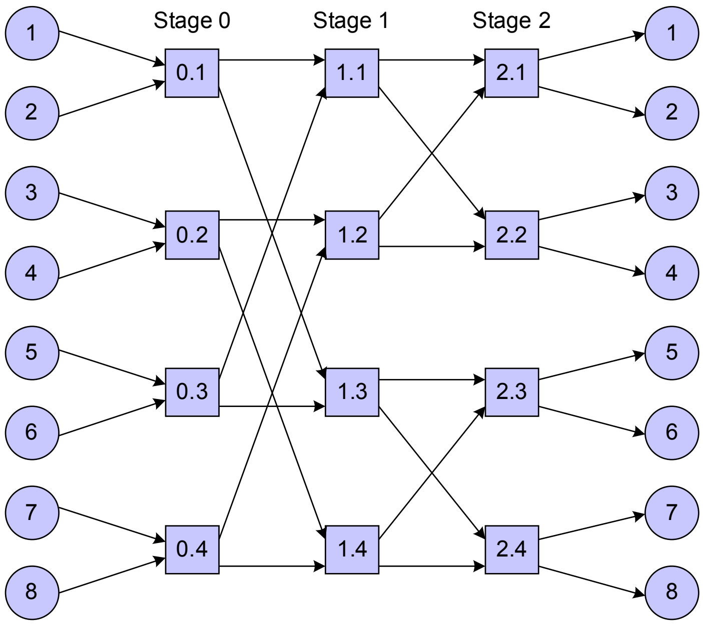


Figure 2.13: A Radix 2 Butterfly with 8 Nodes

by figure 2.12 which plots the number of channels in the network against the number of nodes for two different butterfly networks and a crossbar. The number of channels is plotted on a logarithmic scale to enhance readability. Recall from the beginning of this section that the radix of a network is the number of inputs or outputs of a switch. The function for computing the number of channels in a butterfly network will be discussed shortly.

This section discusses the butterfly and fat-tree topologies. The butterfly topology is chosen because it has the minimum possible diameter for an N node network with switches of degree δ [DT03]. Here, the diameter is the largest, minimal hop count for any pair of network nodes, and the degree is the number of channels that terminate on a node. In other words, $degree = \delta = 2 \cdot radix$ because the radix is the number of input or output channels to a node. Fat-tree topologies are discussed because they have been evaluated for tiled CMPs by Balfour and Dally [BD06].

Butterfly

A butterfly network topology is determined by the radix of its switches and the number of stages. Here, a stage is a group of switches. The letter k is often used to describe the radix and n indicates the number of stages. These numbers must be chosen such that the relation $N = k^n$ holds. In this case, N is the number of nodes in the network.

A short example will make this clear. If $N = 16$, the following topologies are possible:

$$N = 16 = k^n = 16^1$$
$$N = 16 = k^n = 4^2$$
$$N = 16 = k^n = 2^4$$

Consequently, butterflies with radix 16, 4 and 2 are possible. The radix 16 butterfly will have 1 stage, the radix 4 butterfly will have 2 stages and the radix 2 butterfly will have 4 stages. A 1 stage butterfly is simply one crossbar switch.

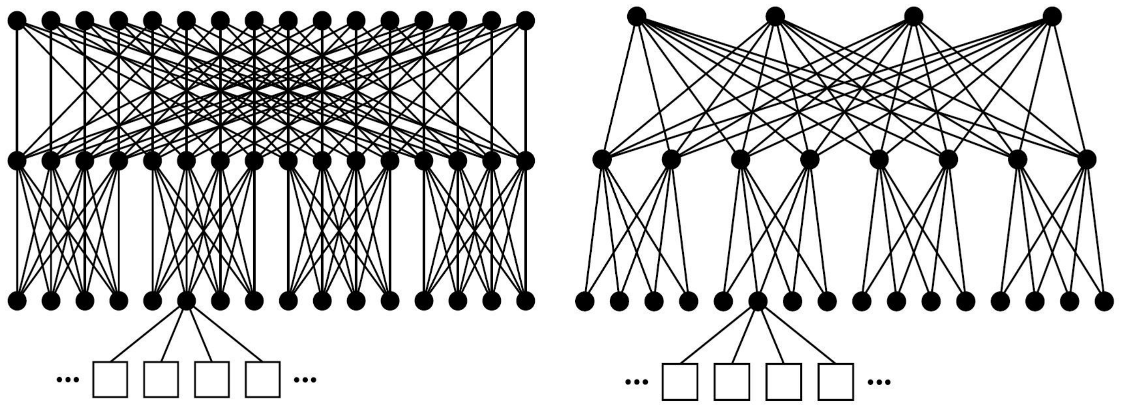

(a) Fat-Tree (b) Tapered Fat-Tree

Figure 2.14: Fat-Tree and Tapered Fat-Tree Topologies (Reproduced from [BD06])

Figure 2.13 shows a butterfly network with 8 nodes, a radix of 2 and 3 stages. This figure illustrates how the switches are connected. First, the terminal nodes are placed in groups of 2 which are connected to one switch. The middle switch stage is then partitioned into two groups and each of the first switches is connected to one switch in each group. This procedure is then repeated for each of the two groups at the middle switch stage, creating four groups at the last switch level. The number of switch groups created at each stage depends on the radix of the switches which is 2 in this example. Consequently, this butterfly construction method is general.

From the above discussion and figure 2.13, we can see that the cost of a butterfly topology is given by the following equations:

$$Number\ of\ switches = n_s = n \cdot \frac{N}{k} = log_k N \cdot \frac{N}{k}$$

$$Number\ of\ channels = n_c = k \cdot n_s = k \cdot log_k N \cdot \frac{N}{k} = N \cdot log_k N$$

A problem with the butterfly network is that it does not have path diversity. In other words, there is only one path from one node to another. For example, assume that node 1 and 6 in figure 2.13 attempts to send data to 7 and 8 at the same time. This is not possible as they both need to traverse the link from switch 1.3 to 2.4. Adding an extra stage increases the path diversity to 2 because there are now two possible paths between two nodes. If n extra stages are added, the butterfly becomes a non-blocking Benes network and the problem is solved [DT03].

Fat-Tree

The fat-tree and tapered fat-tree topologies are shown in figure 2.14(a) and 2.14(b), respectively. The fat-tree is actually a number of interconnected trees. All nodes in each of the interconnected trees have four children, and each sub-tree contains all nodes in the fat-tree. In other words, the radix of the fat-trees seen here is 4. Consequently, there are many different ways to get from one processor to another. The squares in the figures are processor tiles, the points are switches and the lines are channels.

A more detailed look at figure 2.14(a) illustrates how the fat-tree topology is constructed. In this work, an informal discussion is sufficient, and the precise mathematical definition is therefore not discussed. Firstly, the leftmost root node is connected to the leftmost node in each group of four nodes on the next tree level. This node is then connected to its four “closest” children.

In this context, closest means the four nodes directly below the four nodes in the group the parent belongs to. Lastly, each node at the last switch layer is connected to four processors. The processors are the leaf nodes of the tree.

As the previous example illustrates, the wiring complexity of a fat-tree is considerable. The tapered fat-tree reduces this problem by reducing the bandwidth available towards the root. As shown in figure 2.14(b), there are only four root nodes. These are in turn connected to all nodes at the next tree level. Each of the lowest level intermediate nodes has two parents. By comparing figure 2.14(a) and 2.14(b), we can see the available bandwidth in the tapered fat-tree is comparable to the fat-tree when close to the leaves.

Balfour and Dally [BD06] evaluated the fat-tree and the tapered fat-tree topologies for a tiled CMP. They found that the fat-tree has higher performance than the tapered fat-tree. The downside is that the fat-tree uses considerably more area and power than the tapered fat-tree.

In addition, both topologies are inferior to the concentrated mesh topology described in section 2.2.2.1 in terms of performance, power and area. The reason is that the wiring complexity of the tree topologies results in long transmission channels on the chip. Furthermore, these channels must be narrower than the channels in the concentrated mesh to fit in within the assigned area budget. Naturally, longer lines lead to longer delays and more area used. In addition, more power is needed to drive longer lines. Since these lines are narrower, each packet must be split into smaller units than in the concentrated mesh case which makes the problems worse.

It is unclear whether Balfour and Dally's results extend to a shared cache CMP as discussed in section 2.2.2.1. The main problems are that the evaluation is based on tiled CMPs and that only synthetic benchmarks where used in the evaluation.

2.2.3 Routing

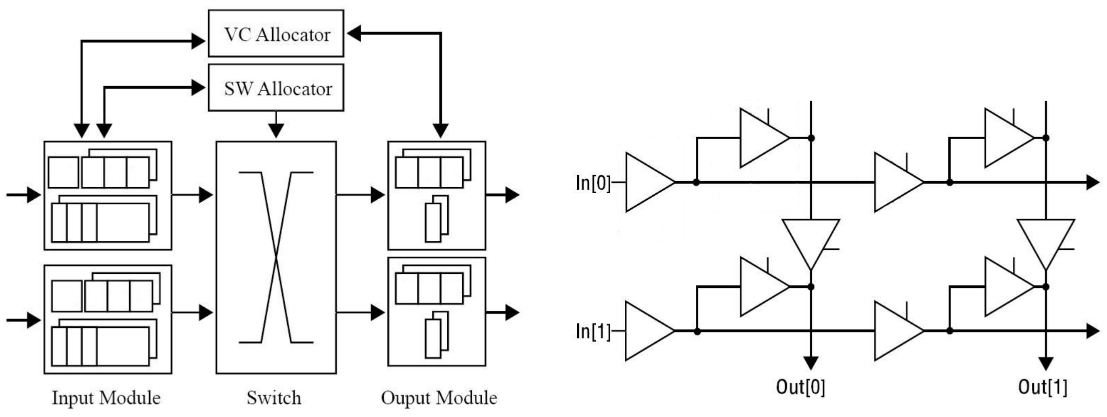
Routing is the task of selecting which channel a given packet should traverse in order to get to its destination. The routing task is highly dependent on the topology. This section introduces the routing problem and how it can be solved. Then, a state-of-the art router used by Balfour and Dally is discussed [BD06].

2.2.3.1 Routing Algorithms

According to Dally and Towels, there are three classes of routing algorithms [DT03]:

- *Deterministic routing algorithms* always choose the same route between two nodes. The main advantage of this class of algorithms is that they are easy to construct. If the underlying topology has multiple paths from one node to another, this is ignored. Consequently, the primary disadvantage is poor load balancing.
- *Oblivious routing algorithms* distribute the network traffic over a set of possible paths without taking the state of the network into account.
- *Adaptive routing algorithms* consider the network status when making a routing decision. Examples of such status are if a link is up or down, queue length and history status. Dally states that the worst-case performance of these algorithms is often poor compared to oblivious routing because status information is mostly local.

Furthermore, a routing algorithm is classified according to which paths it considers legal paths between two nodes. A routing algorithm is *minimal* if it only considers the shortest paths



(a) Router Architecture (b) Crossbar Switch

Figure 2.15: An Example Router Architecture (Adapted from [BD06])

between to nodes as candidate paths. More flexibility is gained by considering longer paths as well. In this case, the algorithm is called *non-minimal*.

2.2.3.2 Router Design

An important contribution from Balfour and Dally's work on tiled CMP interconnection networks, was their area and power models of the interconnect network [BD06]. The router used in their work is shown in figure 2.15(a). Since the router is a basic building block of an interconnection network, it is helpful to have an idea of how it is constructed.

As shown in figure 2.15(a), the router consists of three main parts:

- The *input module* is responsible for receiving incoming data and buffering it.
- The *switch* is a crossbar connecting all input modules to all output modules. The details of its design are shown in figure 2.15(b).
- Finally, the *output module* contains a register that temporarily stores the data. The reason is that traversing the switch takes nearly a clock cycle. Consequently, the data must wait for the next clock cycle before it can be sent over the channel.

The crossbar switch shown in figure 2.15(b) is called a *segmented crossbar*. Its main advantages is a compact layout and low power dissipation [WPM03]. Here, the transmission lines are broken into segments of approximately equal length with tri-state buffers on the boundary between segments. Control signals are chosen in a way that minimises the number of active segments. Consequently, power dissipation is reduced.

The *Virtual Channel (VC)* allocator and *Switch (SW)* allocator in figure 2.15(a) are part of the virtual channel flow control policy implemented in this router. This controls the allocation of resources to packets and is discussed in the next section.

2.2.4 Flow Control

The flow control task can be viewed in two ways. Firstly, it can be seen as a resource allocation task. In this case, resources like buffers and channels are allocated to packets as they advance through the network. In addition, it can be viewed as a conflict resolution task. If two packets arrive at a router at the same time, the flow control policy decides which packet should be forwarded first and how the blocked packet should be handled.

There are a number of different ways to do flow control [DT03]:

- In *Bufferless flow control* there is no temporary storage for packets. Consequently, they are either dropped or routed along a different path. This simple solution is inefficient as it uses network resources for packets that are dropped.
- *Circuit Switching* first reserves resources along a path through the network and then sends one or more packets down this path. This policy is also simple to implement and the buffers needed are small as only the packet header must be buffered. The downside is that setting up a circuit is a considerable overhead if the circuit is only used by a few packets.
- In *store-and-forward flow control*, buffers are added to the routers. Furthermore, each router waits until a packet is completely received before it is forwarded.
- *Cut-through flow control* improves store-and-forward flow control. In this case, the packet is forwarded immediately if there are no conflicts. Consequently, latency is reduced.
- *Wormhole flow control* further improves cut-through flow control by introducing the notion of a *flow control digit (flit)*. Here, a packet is divided into multiple flits. The first flit allocates a virtual channel through the network, and the rest of the flits follow this path. This reduces the buffer space required as only a small number of flits needs to be buffered per virtual channel.
- *Virtual channel flow control* improves wormhole flow control. The problem is that a channel is owned by a packet but that buffers are allocated on a flit-by-flit basis. Consequently, if the buffer goes full, a channel goes idle even though it could be used by flits belonging to a different packet. Virtual channel flow control solves this problem by adding a flit-queue for each router output.

The cut-through and wormhole flow control policies are often referred to as cut-through routing and wormhole routing. However, Dally and Towels maintain that this terminology is imprecise as cut-through and wormhole flow control are flow control policies and have nothing to do with routing [DT03]. Hennessy and Patterson have adopted the improved terminology in the new edition of their book [HP07].

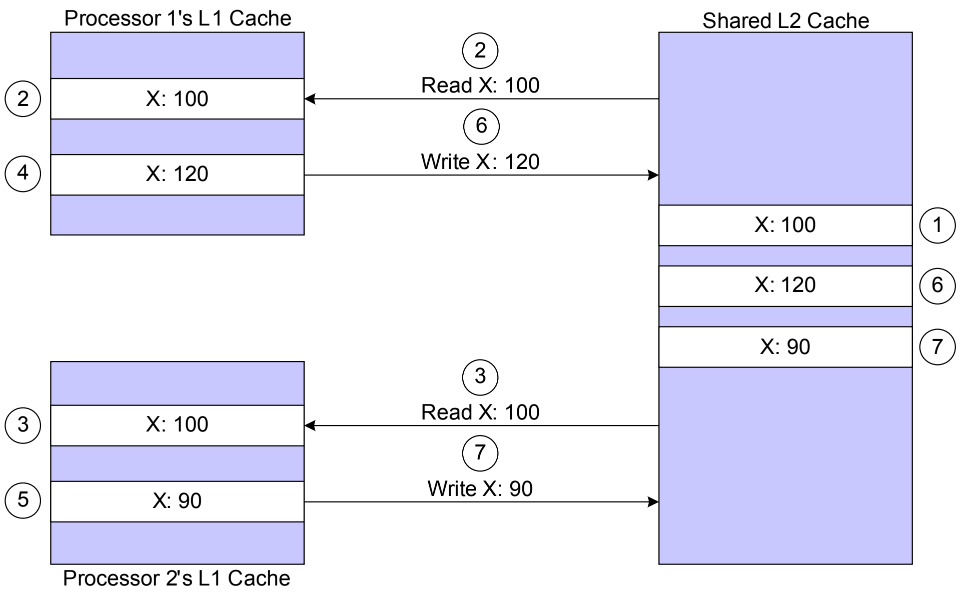


Figure 2.16: Illustration of the Cache Coherence Problem

2.3 Cache Coherence Protocols

A cache coherence protocol solves the *cache coherence problem*. This problem arises when at least two processors write to the same cache block in two different caches at the same time. Figure 2.16 is an example of this situation in a shared L2 cache CMP with write-back caches. The numbers in the figure correspond to the numbers in the list below and define the ordering of the actions.

The actions in figure 2.16 are:

1. First, the L2 cache is the only cache with a copy of block X which has the value 100.
2. Processor 1 reads the value and stores it in its own cache.
3. Then, processor 2 reads and stores the value.
4. Processor 1 writes the value 120 to its copy of the cache line. This update is invisible to both processor 2 and the L2 cache.
5. Processor 2 writes the value 90 to its cache line.
6. Some time later, the block is replaced from processor 1's cache. It is then written back to the L2 cache and the value is updated to 120.
7. Then, processor 2 writes back its copy of X. The value in the L2 cache becomes 90 and the modification made by processor 1 disappears.

This behaviour can not be allowed in a useful system. Consequently, the writing to shared values must be controlled, and a cache coherence protocol is a popular way of providing this control.

For a programmer to be able to use the memory system, a precise definition is needed of how memory requests are ordered. This definition is called a *memory consistency model* [AG96, GLL^{+}98]. Intuitively, a read of a value should return the last value written. In sequential programs this last write is defined by the order of operations in the program. For parallel programs it is more complicated. One possibility is to implement *sequential consistency*. In this case, the operations appear to execute one at the time and in one of the possible sequential orderings

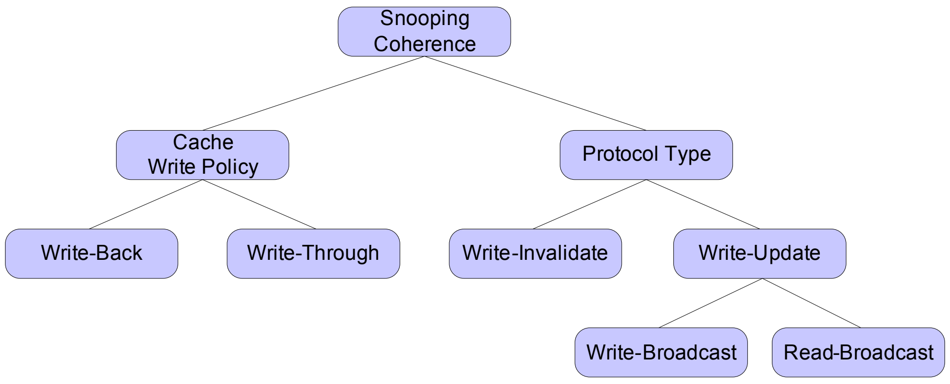


Figure 2.17: Snooping Protocol Possibilities

defined in the program. However, this model disallows a number of hardware and compiler optimisations. Therefore, *relaxed consistency models* have been proposed. Here, varying degrees of freedom is given to reorder reads and writes in ways that do not affect program output.

This section will focus on coherence protocols between private and shared caches in shared last-level cache CMPs. However, coherence protocols are also needed if different CMPs are used to make a larger *Symmetric Multiprocessor (SMP)*. Consequently, solutions like the recent proposal by Marty et al. [MBH$^+$05] regarding intra-CMP coherence is outside the scope of this report. Furthermore, only pure hardware coherence solutions are considered.

This section has the following outline:

- Section 2.3.1 is a high-level introduction to snooping cache coherence protocols. Furthermore, three recent optimisations of snooping coherence in CMPs are discussed.
- Then, section 2.3.2 discusses directory-based cache coherence protocols in depth. The main part of this section is a detailed discussion of a directory protocol proposed by Stenström [Ste89].
- Finally, section 2.3.3 briefly discusses *Token Coherence* [MHW03]. This scheme is an improvement over unoptimised snooping and directory protocols.

2.3.1 Snooping-based Cache Coherence Protocols

Snooping coherence protocols relies on changes to cache blocks being broadcast to all sharers. Furthermore, these broadcasts must be seen in the same order by all sharers. For these reasons, snooping coherence protocols are often used in connection with bus interconnects where these properties come for free. However, it is possible to implement snooping protocols over other interconnects as well, but this requires some additional features.

2.3.1.1 Snooping Protocol Design Space

Figure 2.17 illustrates a few high-level decisions that must be taken when choosing a snooping protocol for a system. The first choice is the write policy of the cache, which is whether the cache is write-back or write-through. Then, a coherence protocol type is selected. The possible choices are *write-invalidate* and *write-update*. In a write-invalidate protocol, writes are carried out locally and all other copies of the block are invalidated. A write-update protocol updates the data stored in other caches when the block is written to. In both protocols, only one cache is allowed to write to a block at the time. There is also a choice of whether this update is carried out directly when the data is written or later when the cache sees a read on the bus. This is known as *write-broadcast* and *read-broadcast*, respectively. According to Culler et al., the write-invalidate protocols are more robust and most vendors provide these protocols as the default [CGS97].

There are many different snooping cache coherence protocols. Consequently, conducting a full survey of these proposals is beyond the scope of this report. Furthermore, this subject matter is discussed and presented in survey articles and textbooks. For instance, both Stenström [Ste90] and Archibald and Baer [AB86] have written survey articles on the subject. In addition, Hennessy and Patterson [HP07] and Culler at al. [CGS97] discuss it in their textbooks. Consequently, this report will focus more on recently proposed enhancements to snooping protocols than the protocols themselves.

2.3.1.2 Recent Enhancements to Snooping Coherence

Snooping cache coherence protocols are attractive because they are considerably easier to implement than their directory-based counterparts. However, they need to check if a block is in a different processor's cache before they retrieve the block from the shared L2 cache. This is most often accomplished by broadcasting the request to all caches and creates two problems. Firstly, this broadcasting consumes considerable interconnect bandwidth [CLS05]. Furthermore, a request must be broadcasted even if the needed cache block is not shared. The reason is that the cache does not know if the cache is shared before it has checked the other caches. Consequently, the latency of all memory requests is increased.

In other words, there are at least two possible ways of enhancing a snooping cache coherence protocol:

- It is possible to improve the protocol in a way that reduces the number of broadcasts issued. Cantin et al. [CLS05] takes this approach with their *Coarse-grain coherence tracking* technique. Here, each core maintains aggregate coherence information for a region of the address space. If no other processors have cached blocks in this address space, there is no need to broadcast the data.
- The protocol can be adapted such that more powerful interconnects can be used. Marty et al. [MH06] extend the snooping protocols so that they can work with a ring interconnect. Another possibility is to map different coherence messages to on-chip wires with different electrical characteristics as proposed by Cheng et al. [CMR$^+$06]. Here, latency critical protocol actions are mapped to fast wires while less critical actions use power efficient, slower wires.

2.3.2 Directory-based Cache Coherence Protocols

A cache coherence protocol needs to enforce an ordering of memory accesses to shared blocks. This is enabled by the interconnect in a snooping protocol. However, this does make it difficult to use a number of interconnection networks. A directory-based protocol employs a directory that stores which cores have cached copies of which cache blocks. In this case, the ordering of memory accesses is enforced by the directory.

The directory can be either *centralised* or *distributed.* A centralised directory stores the status for all cache blocks. However, it can become a bottleneck. Consequently, distributed directories are more common. Here, each memory module is responsible for a part of the global address space and has a directory that keeps track of the caching status of its cache blocks [LLG+90].

There is a hardware overhead associated with storing the sharing status of all cache blocks in the system. Chaiken et al. classifies directory protocols according to their directory implementation [CFKA90]:

- *Full-map directories* make it possible for all caches to have a copy of a given cache block. This is enabled by using one bit per processor to indicate that a processor has a copy of the line.
- *Limited directories* have a fixed number of processor pointers allocated. Consequently, only a subset of the processors can have a copy of a given line. If all pointers are in use and a processor requests the cache block, a cached copy must be invalidated in one of the processor caches. Each pointer use $\log_2 N$ bits where N is the number of processors.
- In *Chained directories* the directory only has a pointer to one sharer. In addition, each cache has a pointer to the next cache that has a copy of a block. In other words, a linked list of sharers is maintained. The main advantage of this directory implementation is the low hardware overhead, and the main disadvantage is that keeping the list updated creates a latency overhead.

The main challenge in implementing a cache coherence protocol is to handle *race conditions* correctly [HP07]. For instance, two or more processors can initiate protocol actions for the same block at the same time in a directory protocol. This race will be resolved when the messages reach the directory since there is only one directory for a given cache block. However, the loser must be notified so that it can take an appropriate action. This is accomplished by sending a *Negative Acknowledgement (NACK)* message to the loser. In addition, some protocol actions require *Acknowledgement (ACK)* messages so that the directory or processor knows that it went through.

Another source of possible deadlocks is finite buffering in the interconnect. The key to avoid these is to ensure that all replies can be accepted and that all requests are eventually serviced. When implementing a coherence protocol in a simulator, it is possible to model infinite buffers. This assumption is made in the coherence implementation described later in the report as it simplifies the implementation.

The rest of this section is a case study of two possible directory protocol solutions:

- First, a recently proposed technique by Eisley et al. called *In-Network Cache Coherence* is discussed [EPS06]. Here, directory protocol requests are optimised while traversing the interconnection network by protocol agents embedded in network routers.

- Then, an old directory protocol proposed by Stenström [Ste89] is discussed in detail. The reason for focusing on this protocol is that it makes frequent protocol actions fast by storing sharer status in the private caches. In a traditionally multiprocessor, this entails a large hardware overhead as the size of the directory depends on the number of different cache blocks that can be stored in private caches at the same time. In a shared L2 cache CMP, the overhead is lower because the L1 cache size is small.

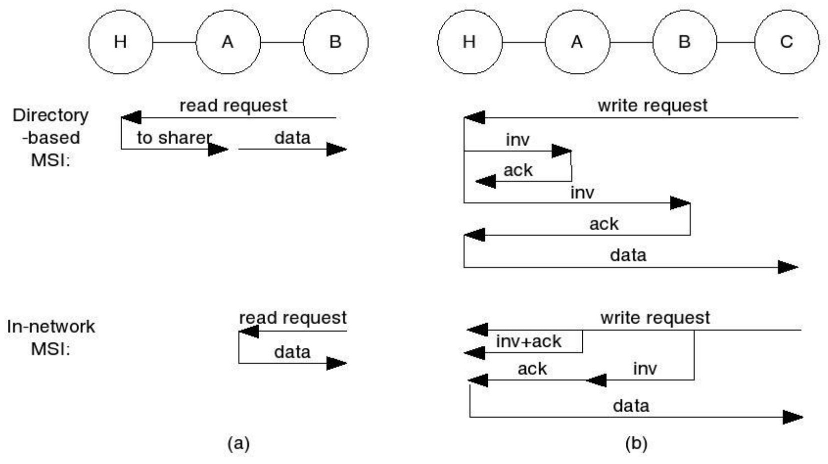


Figure 2.18: In-Network Cache Coherence Optimisation Example (Reproduced from [EPS06])

2.3.2.1 In-Network Cache Coherence

The In-Network Cache Coherence technique by Eisley et al. propose to improve the performance of a directory protocol by reducing the latency of protocol actions [EPS06]. In their approach, both the coherence protocol and directories are embedded within network routers.

Figure 2.18 illustrates how In-Network Cache Coherence reduces the latency of protocol actions. In figure 2.18(a), processor B issues a read request for a block currently cached by A. First, consider a conventional *Modified, Shared, Invalid (MSI)* directory protocol. Here, B's request is sent to the directory H, also known as the *home node*. H then instructs A to supply the data to B and the transaction is finished. In an in-network MSI directory protocol, A intercepts the directory request while it traverses the interconnection network and supplies the data directly. Consequently, latency is reduced.

Figure 2.18(b) shows the protocol actions when processor C wants to write to a cache block present in the caches of processor A and processor B. In the conventional MSI protocol, C sends a request to the directory H. H sends invalidate messages to A and B and sends the data to processor C when it receives the ACK messages from A and B. The in-network protocol optimises the protocol actions by making the write request pass all sharers. A and B then start processing the invalidations when they see C's request. Then, H can supply the data as soon as it has received the request and all ACK messages.

The in-network coherence protocol works by creating a virtual tree for each cache block where the root of the tree is the directory and each leaf is a sharer. Intermediate nodes are routers on the path from the sharer to the directory. To simplify the discussion, we assume that the directory is located at a shared L2 cache bank and the sharers are private L1 caches.

Figure 2.19(a) shows a read request to a cache block that is not cached by any cache. Here, bold arrows represent protocol actions and normal arrows represent virtual tree edges. First, the read is sent to the directory. Then, the L2 cache retrieves the data from memory and sends a reply to the requester. As this reply traverses the network, a virtual tree is constructed. The virtual tree is created by storing the block address and a direction pointer in a small cache in each router. This cache is known as a *virtual tree cache*. The direction pointer points to the next router on the path towards the sharer. Choosing a large virtual tree cache makes it possible for many virtual trees to exist at the same time. However, the size of this cache must be traded off

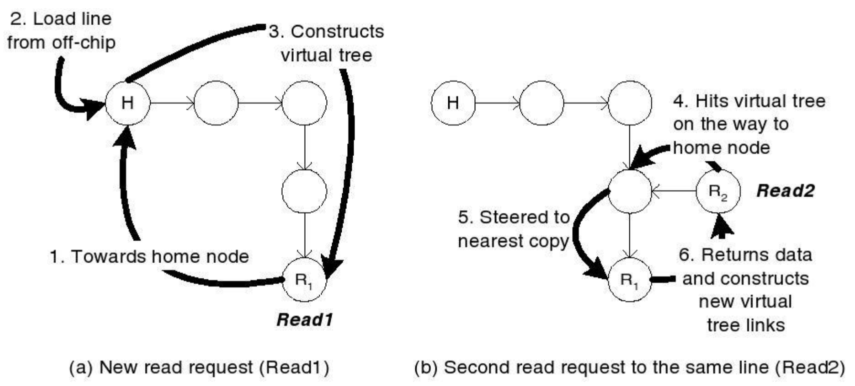

2. Load line from off-chip
3. Constructs virtual tree
H
1. Towards home node
R1
Read1
(a) New read request (Read1)
H
4. Hits virtual tree on the way to home node
R2
Read2
5. Steered to nearest copy
R1
6. Returns data and constructs new virtual tree links
(b) Second read request to the same line (Read2)

Figure 2.19: In-Network Cache Coherence Virtual Tree (Reproduced from [EPS06])

against the impact of its access time on overall router delay.

Figure 2.19(b) illustrates the protocol actions when another cache requests read access to the cache block. Here, the read request is forwarded towards the home node until it arrives at a router which is a member of the virtual tree. This router recognises the block address and redirects the message towards the nearest copy. The sharer cache then provides the requested data and the virtual tree is extended as this message traverses the interconnect.

The In-Network Cache Coherence technique does a good job at optimising the baseline protocol. However, it is unclear how this protocol performs in comparison to other protocol optimisations like for instance the Stenström protocol discussed in the next section. Furthermore, it is designed for a torus network and might need modifications to be adapted to other interconnection networks.

2.3.2.2 The Stenström Directory Protocol

The Stenström Directory Protocol was proposed by Per Stenström [Ste89] and was primarily designed for multiprocessors with multistage interconnection networks. It differs from other directory protocols in that if cache A owns a cache block X, cache A knows which other caches have copies of X. In addition, all other caches that have a copy of X know that cache A owns it. Consequently, they can access cache A directly when they need to read block X. Tang, Censier and Feautrier, Yen and Fu as well as Archibald and Baer have all proposed protocols that require the caches to access the directory before a cache-to-cache transfer. These protocols have all been reviewed by Agarwal et al. [ASHH88] in their survey article and differ mainly in the area overhead of their directory implementation.

In summary, there is a trade-off between area and performance. The traditional directory protocols use less area, but require one access to the directory before all cache-to-cache transfers. The Stenström Protocol uses more area but can send a read request directly to the owner if the owner information is stored in the cache. For writes, the Stenström protocol works similarly to the traditional directory protocols.

This section will discuss how the Stenström protocol can be used in a CMP. To make the discussion easier to understand, a two-level cache hierarchy with private L1 caches and a shared L2 cache is assumed. Implementing the cache coherence protocol between private L2 caches and a shared L3 cache would work in exactly the same way. However, since private L2 caches are normally larger than private L1, the area overhead would be larger.

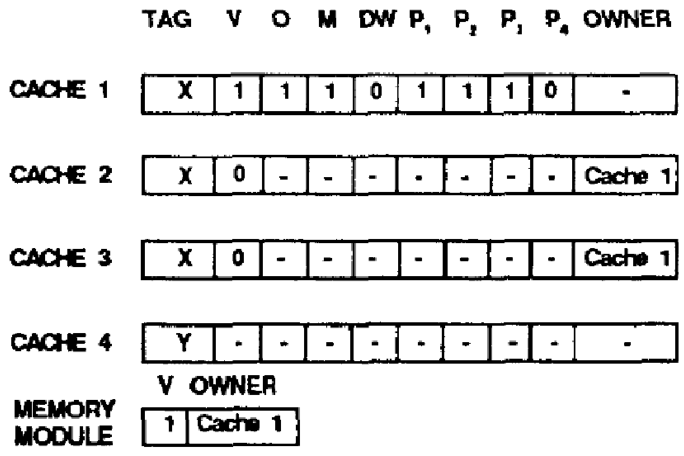

TAG V O M DW P1 P2 P3 P4 OWNER
CACHE 1 X 1 1 1 0 1 1 1 0 -
CACHE 2 X 0 - - - - - - - Cache 1
CACHE 3 X 0 - - - - - - - Cache 1
CACHE 4 Y - - - - - - - - -
V OWNER
MEMORY MODULE 1 Cache 1

Figure 2.20: Status Information used in the Stenström Protocol (Adapted from [Ste89])

Figure 2.20 shows the hardware data structures needed for the Stenström protocol. The *Tag* field has the same meaning as in a normal cache. V denotes the valid bit. A cache can only read or write the block if the valid bit is set to 1. This is only the case for cache 1 in the figure. The reason is that cache 1 owns block X, which is given by the fact that the owned bit (O) is set. The last two single bit fields in the figure are the modified bit (M) and the distributed write mode bit (DW). The modified bit is set when the block data is altered and signifies that the block must be written back when the block is replaced. The distributed write bit selects whether the *Distributed Write* or the *Global Read* mode should be used. The difference between these modes will be explained later in this section. Since the discussions in this report are limited to the *Global Read* mode, this bit will always be set to 0. The owner, modified and distributed write bits only carry meaning for the cache that owns a block. In all sharer caches these bits are don't cares.

The remaining data fields in the caches are the present flags and the owner identification. The present flags contain one bit for each processor. If a processors bit is set to 1, there is an invalid copy of the block in this processors cache. The owner identification field identifies the cache that owns the block. Consequently, read requests to block X from cache 2 or 3 in figure 2.20 can be forwarded directly to cache 1. The present flags are only used in the owner cache, while the owner identification is only used in sharer caches.

The memory module or L2 cache needs to know who owns each cache block. This information is stored in a structure which Stenström calls the *block store*. The reason is that new requests to block X must be redirected to the L1 cache that has the updated copy.

Since only the present flags and owner identification sizes grow with the number of processors in the system, the area overhead of the Stenström protocol in a CMP context is given by the equations:

$$\begin{aligned}\textit{L1 Cache Area Overhead} &= O(\textit{Number of Blocks in DL1} \cdot (P + \log_2 P)) \\ &= O(\textit{Number of Blocks in DL1} \cdot P)\end{aligned}$$

$$\textit{Shared L2 Cache Area Overhead} = O(\textit{Number of Blocks in L2} \cdot \log_2 P)$$

$$\textit{Total Area Overhead} = O((P \cdot \textit{L1 Cache Area Overhead}) + \textit{Shared L2 Cache Area Overhead})$$

The above equations assume that the directory state is stored with the cache blocks in the L2 cache. However, it is possible to use a dedicated memory for this purpose. The rationale is that the worst-case situation occurs when all L1 caches own all blocks stored in them. In other words, this is the situation where all L1 caches are full and there is no sharing. Depending on the cache sizes, this dedicated memory might be larger or smaller than storing the owner information with the L2 blocks. Note that coherence state is only needed for the L1 data cache since the L1 instruction cache is read only.

The area overhead with a dedicated directory memory is given by:

$$\textit{Dedicated Directory Memory Size} = O(P \cdot \textit{Number of Blocks in DL1} \cdot \log_2 P)$$

The shared L2 cache area overhead and dedicated directory memory size computations assume that a full-map directory is used. As mentioned earlier, this area overhead can be reduced by using limited or chained directories at the cost of increased latency for protocol actions.

Figure 2.20 is reproduced from Stenström's original article, but has been changed to reflect the state of the *Global Read* mode. In Stenström's paper, the figure exemplifies the operation of the *Distributed Write* mode.

The area overhead of the Stenström protocol is less important in a CMP-context than in a traditional multiprocessor. The reason is that the number of state bits depend on the size of the caches and the size of the memory or cache at the level where sharing starts. In a CMP with private L1 caches and a shared L2 cache, both the L1 data cache and the shared L2 cache is considerably smaller than the size of the off-chip memory. Consequently, an on-chip realisation of a directory protocol will use significantly less area than a directory protocol between the last level cache and memory. Furthermore, the area difference between a conventional protocol and the Stenström scheme will be small. However, the increase in L1 cache size must be implemented without increasing the access time of this cache. The reason is that the L1 cache often is on the processors critical path and consequently the impact of increasing its hit latency would be large.

This subsection focuses on the *Global Read* mode of the Stenström Protocol. The reason is that it only allows one valid copy of a cache line in the system at any time. In contrast, the *Distributed Write* mode can have one writer and many readers. The key idea is that every write is distributed to all readers. Consequently, the cache block is updated when the reader needs it. In summary, the *Global Read* mode is probably easier to understand while the *Distributed Write* mode has the potential of giving better performance.

The reason for including two different modes in the protocol is that the application can select the mode that gives the best performance for its communication pattern. Furthermore, different phases of the program might use different modes. The *Global Read* mode is simple, but it adds the extra latency of accessing a remote cache for the readers. In the *Distributed Write* mode, writes to a shared block are distributed to all caches that have a copy of it. Consequently, the block might already be in the reader's cache when it needs it. However, all updates are forwarded to all copies, and this might create a lot of unnecessary network traffic. In other words, the protocol is optimised for the communication pattern with one writer and many readers. Stenström states that many supercomputing applications follow this communication pattern [Ste89].

There are six possible states in the protocol, but only three of them are used in the Global Read mode:

- *Invalid* - The cache block is not valid. If the owner information is present, reads are forwarded to the cache that owns the block. This point is the key to understanding how the protocol works and the reason for naming the protocol mode global read. On writes, ownership must be acquired before the write can be carried out.
- *Owned Exclusively Global Read* - This cache owns the cache block, and *it is the only cached copy.* Both reads and writes can be carried out without delay.
- *Owned NonExclusively Global Read* - This cache owns the cache block, but *at least one other cache has a copy in the Invalid state.* Since all other copies of the cache line are invalid, reads and writes can proceed without delay.

The other three states are *UnOwned*, *Owned Exclusively Distributed Write* and *Owned NonExclusively Distributed Write.* These states are only used in the *Distributed Write* mode.

The rest of this section discusses the protocol actions for read hits, read misses, write hits, write misses and cache block replacements from the L1 data caches.

Read Hit

A request is only a hit in the cache if the valid bit of the cache block is set to 1. This can only happen in the states *Owned Exclusively Global Read* and *Owned NonExclusively Global Read* in the *Global Read* mode. In other words, this is the most recent data copy and the read can proceed without delay.

We do not need to inform any other sharers for two reasons. Firstly, we are reading the cache and consequently not changing the data. Also, since we are in the *Global Read* mode, this cache has the only valid copy of the block. Consequently, other sharers will access this cache when they need the data.

Read Miss

There are three main cases for a read miss when using the *Global Read* policy. The first possibility is that the cache block is not present in the L1 cache. Furthermore, the block might be present in a different L1 cache. In addition, the cache block might be present but invalid. These cases are handled differently by the protocol, and they are shown in figure 2.21.

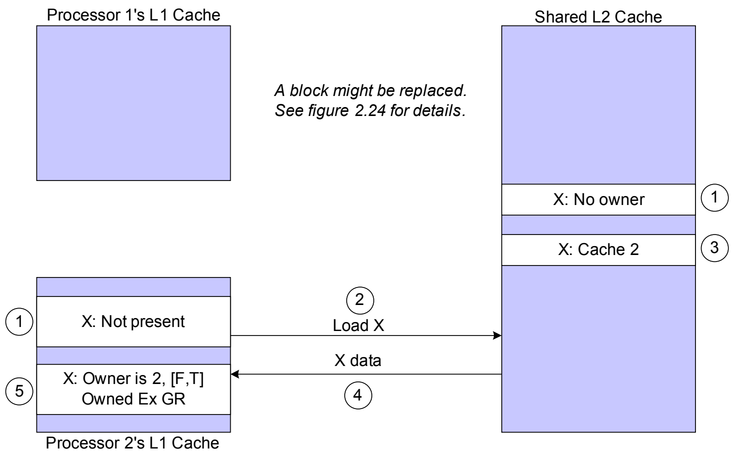
In the two first cases a block might be replaced in the requesting cache. The protocol actions in this case are described later in this section.

Consider the case where the only available copy of cache block X is in the shared L2 cache. This case is shown in figure 2.21(a). The protocol actions are described in the following list, and the numbers in the list correspond to the numbers in the figure:

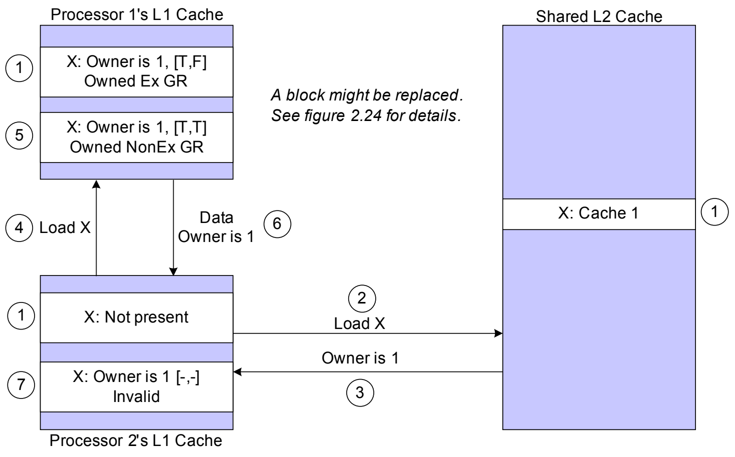
1. At first, the only valid copy of X resides in the shared L2 cache.
2. Processor 2's L1 cache does not have a copy of cache block X and issues a load request to the L2 cache.
3. The L2 cache sets Cache 2 as the owner of block X.
4. Block X is sent to Cache 2.
5. Cache 2 stores the data in its cache, initialises the present flags and sets the state to *Owned Exclusively Global Read.*

The actions taken when block X is present in a different L1 cache are shown in figure 2.21(b):

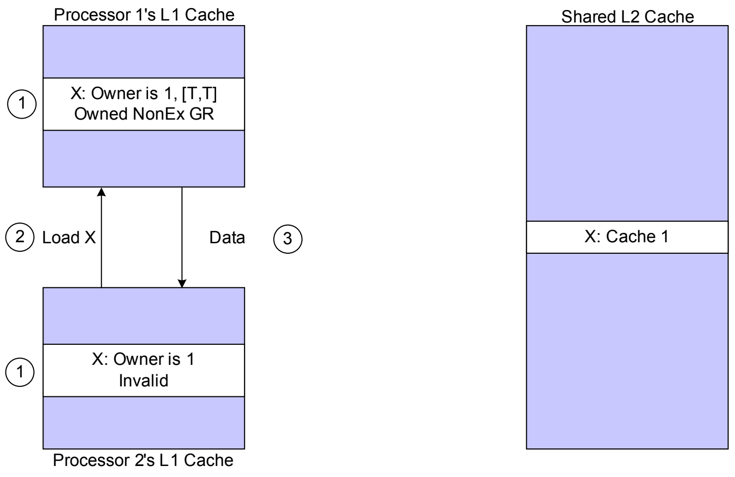
1. Initially, the valid cached copy is in processor 1's cache and the L2 cache has recorded processor 1 as the owner of block X. Block X is not present in processor 2's cache.



(a) The other caches does not have a copy of X



(b) At least one other cache has a copy of X



(c) The requesting cache has an invalid copy of X

Figure 2.21: Stenström Protocol Read Miss Handling

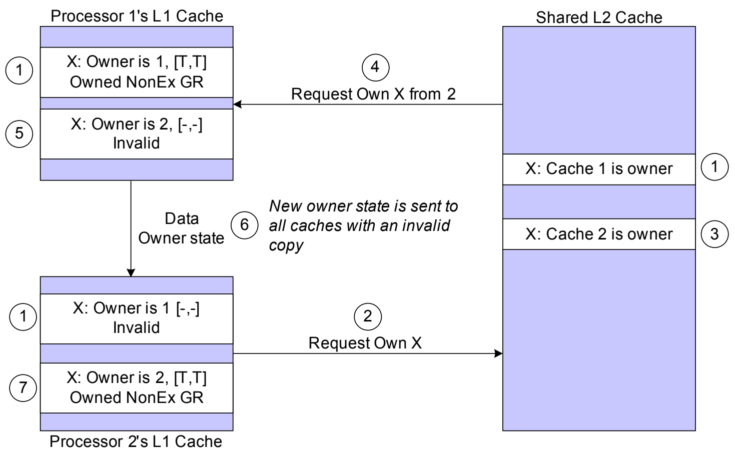


Figure 2.22: Stenström Protocol Write Hit Handling

2. Processor 2 issues a load request for block X to the L2 cache.
3. The L2 cache tells processor 2 that processor 1 is the owner. Consequently, processor 2 will redirect its request to processor 1.
4. Processor 2 sends the load request to processor 1.
5. Processor 1 changes the state of block X from *Owned Exclusively Global Read* to *Owned NonExclusively Global Read*. In addition it sets processor 2's present flag. If the state of block X in processor 1's cache was *Owned NonExclusively Global Read*, block X would remain in this state and only the present flag would be set. This could happen in a system with more than 2 processors if one of the other processors previously had read block X.
6. Processor 1 sends the data and the owner identification to processor 2. The owner identification is attached so that processor 2 does not need to remember which processor the message was sent to. In other words, a buffer is avoided.
7. Processor 2 stores block X in its cache and uses the received data. The state of block X is set to *Invalid* to ensure that subsequent reads of the block is redirected to the owner cache. In other words, the protocol does not migrate data.

When block X is present in the cache with state *Invalid*, the protocol actions shown in figure 2.21(c) are taken:

1. At first, block X is present in both processor 1's and processor 2's L1 caches. Processor 1 owns block X. Since there is more than one copy, the state is *Owned NonExclusively Global Read*. Processor 2's copy of X is invalid since all copies that are not owned must be invalid in the *Global Read* mode.
2. Processor 2 knows that processor 1 is the owner and issues the load request directly to processor 1 without accessing the L2 cache. Other directory based protocols would access the L2 cache first in this case.
3. Processor 1 sends the data to processor 2. There are no changes to the protocol state in any of the L1 caches.

Write Hit

In the *Global Read* mode there are three possible protocol actions on a write hit. Firstly, the cache block can be in the state *Owned Exclusively Global Read*. In this case, the write can proceed without delay as there is no other cached copy of the cache block. Secondly, the cache block can have the state *Owned NonExclusively Global Read.* Here, the write can also proceed without delay because all other cached copies of the block are invalid. Consequently, reads to these blocks will be redirected to this cache.

The last possibility is that the cache line is invalid. In this case, the cache needs to obtain ownership of the block before it can write to it. The protocol actions needed are shown in figure 2.22 and described in the following list:

1. First, processor 2's cache has an invalid copy of block X. The valid copy is present in processor 1's cache in the state *Owned NonExclusively Global Read.* In addition, the L2 cache has stored that processor 1 is the owner of cache block X.
2. Processor 2 requests ownership of cache block X from the L2 cache. The request is sent to the directory as this is the point of serialisation.
3. The L2 cache sets processor 2 as the owner of block X.
4. The L2 cache informs processor 1 that processor 2 is the new owner of block X
5. Processor 1 sets the state of block X to *Invalid* and sets processor 2 as the new owner.
6. Processor 1 then sends the data and the present flags for block X to processor 2. In addition, it informs all other caches that have a copy of X that the new owner is processor 2. The present flags tell processor 1 which processors to inform.
7. Finally, processor 2 stores the data and the present flags of block X in its cache. The state is set to *Owned NonExclusively Global Read.* The write operation is carried out and the modified bit is set to 1.

Write Miss

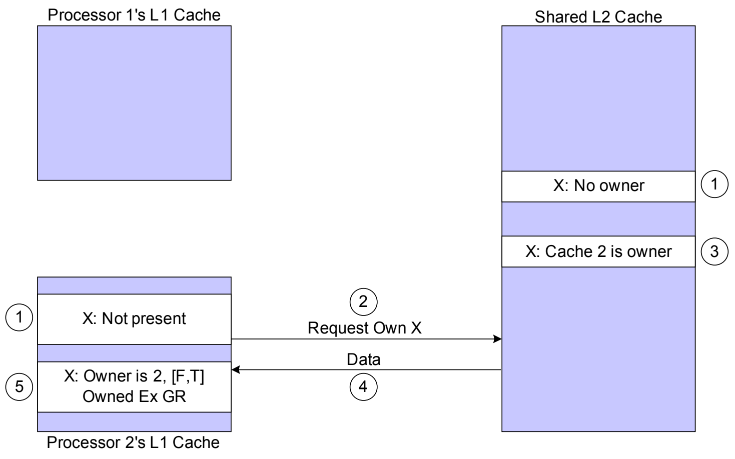
The protocol actions on a write miss are similar to the actions taken on a write hit. Again, only the owner can write to a block. Consequently, a cache must obtain ownership of the block before the write can proceed. There are two possibilities. Either the block is already present in a L1 cache or it must be retrieved from the L2 cache.

Consider the possibility where the cache block is not present in any L1 caches. The protocol actions for this case are shown in figure 2.23(a) and are described in the following list:

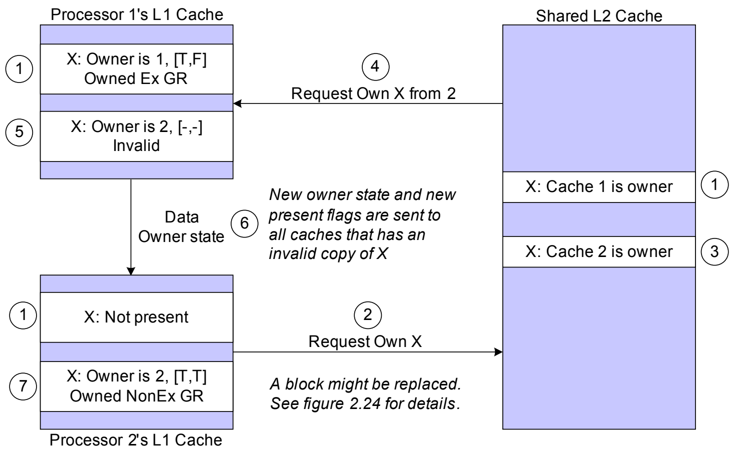
1. Initially, block X is neither present in processor 2's L1 cache or in the L2 cache.
2. Processor 2 requests ownership of block X from the shared L2 cache.
3. The L2 cache retrieves block X from memory if necessary and sets processor 2 as the owner.
4. The L2 cache sends the data to processor 2.
5. Processor 2 stores block X in its cache and sets the state to *Owned Exclusive Global Read.* The present flags are initialised with processor 2's flag set to 1 and all other flags set to 0. The write operation is carried out and the modified bit is set to 1.

The other write miss possibility is that the block is present in a different L1 cache. Here, it does not matter whether the block is not present or invalid in the requesting processor's L1 cache. Figure 2.23(b) shows the case where block X is not present in the requesting processor's cache. In either case, the following protocol actions are carried out:

1. At first, block X is not present in processor 2's cache, and processor 1 owns block X. Block X is in the state *Owned Exclusive Global Read*, but the protocol actions would be essentially the same if there where other sharers. The only difference is that the owner information is forwarded to all sharers in point 6.
2. Processor 2 requests ownership of block X from the L2 cache.
3. The L2 cache changes the owner of block X to processor 2.



(a) The other caches does not have a copy of X



(b) At least one other cache has a copy of X

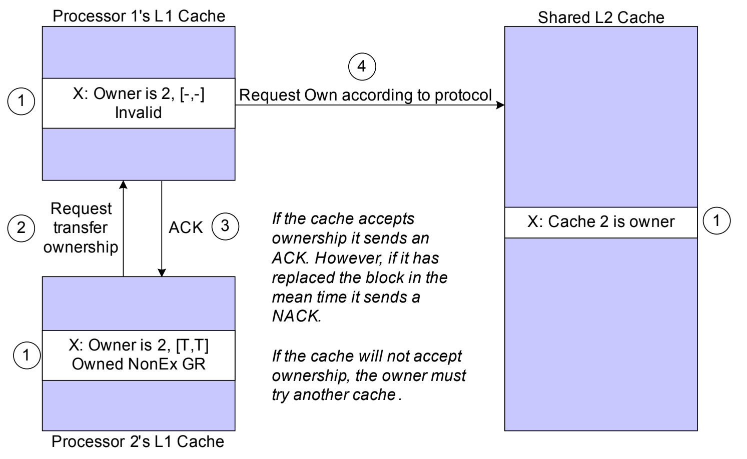
Figure 2.23: Stenström Protocol Write Miss Handling

4. The L2 cache issues a request to processor 1 telling it that processor 2 is the new owner of block X.
5. Processor 1 sets processor 2 as the new owner of block X and changes the state to *Invalid*.
6. Processor 1 sends the data and the present flags of block X to processor 2. If there are other processors with a copy of block X in the system, processor 1 forwards the new owner information to them.
7. Processor 2 creates or updates the cache entry for block X with the data and present flags received from processor 1. The new state of block X is *Owned NonExclusively Global Read*. The write operation is carried out and the modified bit is set to 1.

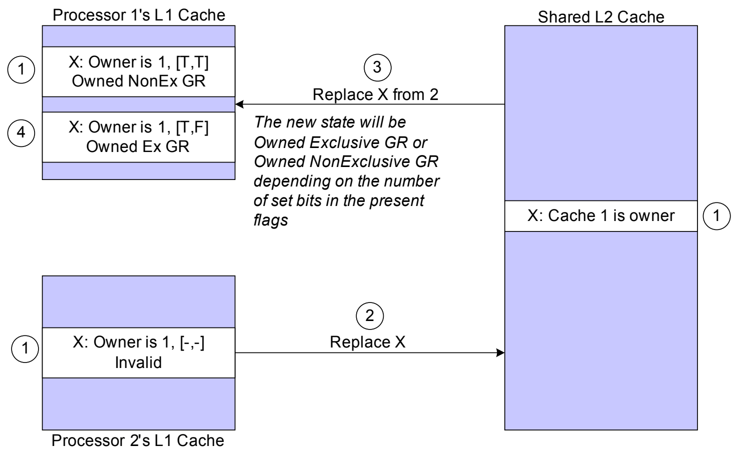
Block replacement

There are three possible protocol actions on a block replacement. If the block is owned exclusively, it is the only cached copy. In this case, the L2 cache must be notified that the cache no longer owns the block. Furthermore, if the block is modified it must be written back.

In addition, the block might be non-exclusively owned or invalid. These cases are shown in


Processor 1's L1 Cache
Shared L2 Cache
1 X: Owner is 2, [-,-] Invalid
4 Request Own according to protocol
2 Request transfer ownership
ACK 3
If the cache accepts ownership it sends an ACK. However, if it has replaced the block in the mean time it sends a NACK.
If the cache will not accept ownership, the owner must try another cache.
X: Cache 2 is owner 1
1 X: Owner is 2, [T,T] Owned NonEx GR
Processor 2's L1 Cache

(a) The to-be-replaced block X is in the state *Owned NonExclusively Global Read*


Processor 1's L1 Cache
Shared L2 Cache
1 X: Owner is 1, [T,T] Owned NonEx GR
3 Replace X from 2
4 X: Owner is 1, [T,F] Owned Ex GR
The new state will be Owned Exclusive GR or Owned NonExclusive GR depending on the number of set bits in the present flags
X: Cache 1 is owner 1
2 Replace X
1 X: Owner is 1, [-,-] Invalid
Processor 2's L1 Cache

(b) The replaced block X is invalid

Figure 2.24: Stenström Protocol Block Replacement Handling

figures 2.24(a) and 2.24(b) respectively.

Consider first the case where the block is non-exclusively owned. Here, ownership of the cache block must be transferred to a different L1 cache. The new owner can be chosen arbitrarily from the present flags. The following actions are carried out:

1. At first, processor 2 is the owner of cache block X and processor 1 has an invalid copy.
2. Processor 2 wants to replace block X, and requests processor 1 to become the new owner.
3. (a) If processor 1 still has block X in its cache, it responds with an *Acknowledgement (ACK)* message. Processor 2 still can not replace block X, as processor 1 will go through the usual protocol steps for acquiring ownership of block X.
   (b) If processor 1 does not have block X in its cache, it responds with a *Negative Acknowledgement (NACK)* message. Processor 2 must then set processor 1's present flag to 0 and try to transfer ownership to a different cache. If there are no more sharers, processor 2 can change the state to *Owned Exclusively Global Read* and follow the protocol actions for the owned exclusively case.

4. In figure 2.24(a), processor 1 accepts ownership of block X. Then, it requests ownership for the block using the normal protocol. When processor 2 has delivered the block data and present flags to processor 1, it can replace block X.

The protocol actions when the block is invalid are considerably simpler as shown in figure 2.24(b):

1. Initially, block X is owned by processor 1, and processor 2 has an invalid copy. Processor 2 wants to replace block X.
2. Processor 2 informs the L2 cache that it will replace block X. There is no need to issue a writeback as an invalid copy never can be modified. When this request is sent, processor 2 is free to replace block X.
3. The L2 cache informs processor 1 that processor 2 no longer has a copy of block X.
4. Processor 1 sets processor 2's bit in the present flags to 0. If this results in processor 1 being the only processor with a copy of X, the state is changed to *Owned Exclusively Global Read*.

2.3.3 Alternative Cache Coherence Solutions

The cache coherence problem can be solved in other ways than by snooping or directory-based cache coherence protocols. Although this report mainly focuses on these traditional approaches, a brief look at an alternative solution is in order.

Martin et al. maintain that snooping and directory-based protocols solve the coherence problem in an inefficient way [MHW03]:

- Snooping protocols require a totally ordered interconnection network. This avoids protocol race conditions, but can limit performance.
- Traditional directory protocols resolve races by accessing the directory in an ordered fashion. Although it removes the totally ordered interconnect restriction, it adds overhead by requiring an access to the directory before a cache-to-cache transfer. Martin et al. refers to this property as adding *indirection*.

According to Martin et al., the fastest way to service a request for shared data is to broadcast it and let the cache with updated data respond. The problem is that this might lead to race conditions on an unordered interconnect. However, these race conditions will be rare.

Token Coherence uses these properties to solve the cache coherence problem efficiently [MHW03]. Here, a number of *tokens* equal to the number of processors in the system are added for each cache block. If a cache has all the tokens for a block, it can safely write to the block as there are no other cached copies. Similarly, a cache can read a block if it has at least one token. Furthermore, all requests containing tokens must contain valid data. Consequently, the protocol enforces that only one processor writes to a block and that multiple processors can read a block.

If two processors attempt to write to the same block simultaneously, they will compete for the same tokens. This situation can lead to starvation. Martin et al. use *persistent requests* to handle this situation. This special request type results in that all tokens belonging to a given block is sent to the requester. There can only be one persistent request for a given cache block in the system at the time. Since the requester gets all tokens, the request is guaranteed to complete. A cache detects possible starvation by measuring the time it takes to service a cache miss. If this time increases beyond a given threshold, a persistent request is issued. For this mechanism to guarantee freedom from starvation, a fair mechanism must be provided to decide which cache should be allowed to issue the persistent request.

Token Coherence has been compared to a snooping and a directory-based protocol [MHW03]. It performs better than a snooping protocol if it is run on an interconnect that does not order requests. Furthermore, it outperforms a directory-based protocol at the cost of a somewhat higher interconnect bandwidth requirement. The scalability of token coherence is somewhat limited because it uses broadcasts. However, Martin et al. state that they expect the protocol to scale well enough to be used in a 32 or 64 processor SMP provided that the interconnect is powerful enough.

Chapter 3

Research Questions and Methods

This chapter is a discussion of the research questions this report is based on and how they will be answered. The research questions lay the foundation for the practical work described in this report. The chapter has the following outline:

- Section 3.1 states and discusses the research questions.
- Section 3.2 describes the baseline CMP architecture used in this report.
- Section 3.3 discusses possible simulators and explains why the M5 simulator [BDH^{+}06] was chosen.
- Finally, section 3.4 discusses possible benchmarks and how they will be used.

3.1 Research Questions

The research questions should be well-formulated and within the boundaries given by the assignment text. Consequently, they are a tool for focusing the work as well as giving the reader a feeling of where the report is heading.

This report will focus on the following research questions:

1. How does the CMP on-chip interconnect between private and shared caches influence overall system performance for *multiprogrammed workloads*?
2. How does the CMP on-chip interconnect between private and shared caches influence overall system performance for *scientific workloads*?
3. Can improvements to the private to shared cache interconnect improve performance for both multiprogrammed and scientific workloads?

The research questions follow one suggestion for further work outlined in my 5th year project report [Jah06]. In this work, the interconnect, the cache coherence protocol and *Non-Uniform Cache Access (NUCA)* cache designs were identified as the most promising areas for CMP communication research. Since it is probably a good idea to start with a reasonably simple system, NUCA architectures are not considered here.

The cache coherence protocol depends heavily on the properties of the interconnect. In particular, a snooping protocol can be used if the interconnect guarantees a global ordering of requests. One problem with a snooping protocol is that it broadcasts requests, and this can create a large strain on the interconnect. Furthermore, creating a global ordering might prevent an interconnect from reaching its full performance potential. In addition, it is difficult to create an ordering

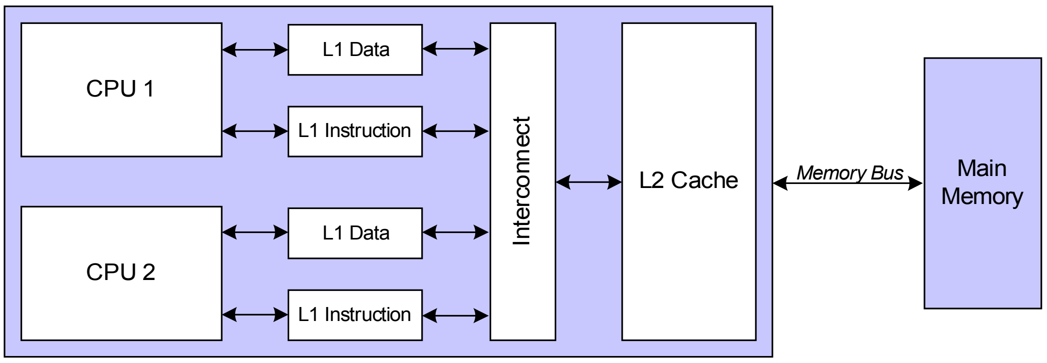


Figure 3.1: High-level Chip Multiprocessor Architecture

in some interconnects. Choosing a directory protocol creates great freedom in which interconnects can be investigated. Furthermore, this will give insights into cache coherence protocols in general. In this sense, the research questions attack both the interconnect and the cache coherence protocol research fields.

The assignment text states that this report should investigate the performance of communication intensive workloads in CMPs and identify performance bottlenecks. Furthermore, architectural techniques that alleviate these bottlenecks should be proposed. The first issue is addressed by research question 2, and the second issue is addressed by question 3. Furthermore, these questions focus the work towards the private to shared cache interconnect. However, question 1 does not address any of these issues. The reason for including this question is that multiprogrammed workloads are an easier problem to deal with as they do not require a cache coherence protocol. Consequently, the proposed interconnects can be evaluated before the cache coherence protocol is ready. The effect is that it is possible to investigate architectural effects early on. Hopefully, this will result in a better understanding of the problem at hand than if the assigned problem was addressed right away.

For these reasons, answering the research questions will keep the report within the further work identified in my project report and fulfil the requirements given in the assignment text.

3.2 CMP Architecture Model

This section presents the CMP architecture used in this report. As shown in figure 3.1, each core has a private instruction cache and a private data cache. These caches are connected to a shared L2 cache through an interconnect. The L2 cache is then connected to the main memory with a memory bus.

A possible design alternative is to have private per-core L2 caches as used in AMD's Athlon 64 X2 processor [AMD]. In general, these private caches will have lower access times which in many cases will result in increased performance [CS06]. However, if the working set for one processor is larger than the local L2 cache size, the result will be many costly off-chip memory accesses and reduced performance. Another important drawback of a pure private cache scheme is that inter-core communication must go via the memory bus. Since the aim of this work is to investigate communicating workloads, a shared cache scheme will probably be more appropriate.

Many decisions must be taken when configuring a CMP simulator. For instance, the type and

Parameter	**Value**
Clock frequency	3.2 GHz
Reorder Buffer Size	128 entries
Store Buffer Size	32 entries
Instruction Queue Size	64 instructions
Instruction Fetch Queue Size	32 entries
Load/Store Queue Size	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs 2 Integer Multipy/Divide 4 Floating Point ALUs 2 Floating Point Multiply/Divide 4 Memory Read/Write Ports 1 Internal Register Access Port
Branch predictor	Hybrid, 2048 local history registers, 2-way 2048 entry Branch Target Buffer (BTB)
Decode to Dispatch latency	10 cycles
Dispatch to Issue latency	1 cycle

Table 3.1: Baseline Processor Model Parameters

size of the branch predictor, the size of the issue queue and the L1 cache hit latency must be decided. This is a difficult task and the numbers used in academic publications differ widely. Consequently, values from existing processors will be used in this report if they are available. The goal is that the simulated CMP should be representative of current CMP implementations.

The scripts used to configure the M5 simulator can be found in appendix D.

The rest of this section has the following outline:

- Section 3.2.1 discusses the parameters chosen for the processor core.
- Section 3.2.2 discusses the parameters chosen for the caches and the main memory.
- Section 3.2.3 presents the modelled interconnects and their parameters.
- Finally, section 3.2.4 presents the cache coherence protocols and their parameters.

3.2.1 Processor Parameters

The work is not primarily aimed at the processor cores. However, the design of the processor cores has a significant impact on the results of the memory system research. For instance, a simple in-order processor core will not create as many simultaneous memory requests as a powerful out-of-order core. Consequently, it would be advantageous to have powerful processor cores as this would most likely put a large strain on the memory system.

This design philosophy is apparent in the processor core parameters shown in table 3.1. Firstly, the different queues and buffers are relatively large. Therefore, the processor can issue many parallel memory requests. In addition, the processor has many functional units available. This makes the memory accesses more frequent as the CPU-bound portions of the programs are executed faster than on a less powerful design.

The clock frequency of 3.2 GHz is taken from the Intel Core 2 Extreme shared L2 cache 2-core processor [GMNR06]. Choosing this low but realistic clock frequency has the added benefit that the interconnect and the processor cores can be clocked at the same frequency. If the clock frequency was higher, each interconnect clock cycle might need to be 2 or more processor clock cycles. This simplifies the simulator and is further discussed in section 3.2.3. In addition, 3.2 GHz makes one processor cycle correspond to 4 effective memory bus cycles with an 800 MHz PC6400 DDR2 memory bus. This point is discussed further in section 3.2.2.

With the deep pipelines of recent processors, branch prediction is an important issue. Consequently, a lot of research has been done on this subject. The main requirement for the branch predictor in this context is that it is sufficiently powerful to not stall the processor pipeline too often. Again, the reason is that many pipeline stalls will result in less strain on the memory system.

Using the terminology of Smith [Smi81], a branch prediction strategy can be static or dynamic. A dynamic strategy takes into account the run-time behaviour of the branch while a static strategy does not. Naturally, a dynamic strategy is more accurate, but it does need more hardware support. Furthermore, McFarling has shown that the best accuracy for a given predictor size is attainable by combining different dynamic strategies [McF93].

McFarling based his work on three different schemes:

- A *bimodal predictor* assumes that a branch will go in the same direction as it did the last few times it was executed
- A *local predictor* stores the history of a one branch instruction and makes predictions for it based on this history information
- A *global predictor* uses the combined history of all recent branches when making a prediction

The M5 simulator supports the local and global branch predictor schemes. The simulated CMP uses the scheme that performs best for each branch at any given time. Furthermore, the Branch Target Buffer (BTB) and the number of local history registers are both fairly large. Consequently, the simulated branch predictor is reasonably powerful.

The last parameters in table 3.1 are decode-to-dispatch latency and dispatch-to-issue latency. These parameters make it possible to simulate a deeper pipeline than the 5-stage Alpha pipeline M5 is based on. Consequently, by setting the minimum time an instruction must use from the decode to the dispatch stage to 10 cycles, the pipeline would behave as if it has approximately 15 stages. 15 pipeline stages are more reasonable than 5 stages given a clock rate of 3.2 GHz.

The M5 simulator does not support *Translation Lookaside Buffer (TLB)* simulation in system call emulation mode. In this mode, the benchmark's system calls are executed by the operating system the simulator is run on. Consequently, the modelled CMP does not contain a TLB. Alternatively, the full system mode of M5 could be used. Here, the operating system is simulated as well as the benchmark. The reason for not choosing full system simulation is that carrying out the simulations takes more time and interpreting the results becomes more difficult. Not simulating the TLB is probably a small price to pay to avoid these difficulties. This point is further discussed in section 3.3.

3.2.2 Memory System Parameters

The cache latencies in table 3.2 are based on the cache parameters for Intel's new Core 2 Duo architecture [Int06]. This architecture has a similar memory system to the simulated CMP with

Parameter	**Value**
Level 1 Data Cache	32 KB 8-way set associative 64 KB blocks LRU replacement policy Write-Back 3 cycles latency 1 bank 4 MSHRs
Level 1 Instruction Cache	32 KB 8-way set associative 64 KB blocks LRU replacement policy Write-Back 1 cycle latency 1 bank 4 MSHRs
Level 2 Unified Shared Cache	4 MB 8-way set associative 64 KB blocks LRU replacement policy Write-Back 14 cycles latency 4 banks 8 MSHRs per bank
Main memory	112 cycles access time 8 byte wide, DDR2-800 memory bus

Table 3.2: Baseline Memory System Parameters

one L1 instruction cache and one L1 data cache per core. A large L2 cache is shared between all cores. The cache sizes and latencies as well as the write strategy in table 3.2 are are all identical to the Intel architecture.

The L2 cache is similar to the static non-uniform access cache architecture (S-NUCA) design described by Kim et al. [CKB03]. The reason for it having a non-uniform access time in the work by Kim et al., is that each bank will be at a different distance from each core. Consequently, the transfer times through the interconnect can be different between different cores and banks. In this report, all banks have the same interconnect delay and access times. This will probably make the results easier to interpret, but the delay must be large enough to enable the cores to reach the farthest bank within this time. Consequently, some performance might be lost.

Kim et al. argue that up to 32 banks are reasonable for a 4MB L2 cache in a 100 nm technology. However, the IBM Power 4 L2 cache has only 3 banks [KZT05]. Consequently, it is somewhat unclear which number of banks is the best choice for the simulated CMP. Following the commercial designs, the bank count is set to 4.

The caches in M5 are non-blocking. In other words, they can service more requests while outstanding misses are being serviced by a unit further down in the memory hierarchy. This cache design scheme was first proposed by Kroft [Kro81], and became a necessity with the deeply pipelined superscalar out-of-order designs of the 90's. The key idea in this scheme is to add a hardware structure called a *Miss Status Holding Register (MSHR)* to keep track of the outstanding misses. Furthermore, by searching the MSHRs on each miss, a missing cache block is only requested once.

According to Kroft [Kro81], 4 MSHRs per bank is a good balance between allowing a sufficient number of outstanding misses and hardware cost. Sohi and Franklin also used 4 MSHRs per bank in their evaluation of a superscalar processor [SF91]. Consequently, the simulated CMP will use 4 MSHRs per bank in the L1 caches. However, it is unclear if these results extend to a large L2 cache in a CMP. Consequently, 8 MSHRs per bank will be used here as the shared L2 cache probably should tolerate more outstanding misses than the private L1 caches.

The focus of the research in this report is the interaction between the different L1 caches and the shared L2 cache. Consequently, it is not necessary to go into to much detail on how the main memory and the memory bus are modelled. However, the parameters chosen to describe these units must be representative.

The memory bus performance is modelled on the DDR2-800 memory bus standard. The reasons for choosing this standard is that it is a relatively recent standard that fits well with a core clock of 3.2 GHz. M5 requires the effective bus clock frequency to be a multiple of the processor clock frequency. In the DDR2-800 standard, the memory bus is clocked twice as fast as the memory itself, and data is transferred on both rising and falling clock edges. The 800 part of the DDR2-800 name indicates that the bus has an effective clock frequency of 800 MHz. However, the real bus frequency is 400 MHz and the memory is clocked at 200 MHz. This relation is important as vendors provide the memory latency measured in the number of memory clock cycles used to supply the data. However, the memory chip suppliers use the 800 number when they market their chips. Therefore, knowledge about the bus interface is needed to compute the actual latency.

The latencies used in this report are taken from the Corsair TWIN2X2048-6400 memory chip [Cor07]. This chip was chosen because it has the DDR2-800 interface and the manufacturers documentation is sufficiently detailed to compute the access latency. However, it is only an example of a memory chip and not representative of memory chips in general.

Ali [Ali06] states that the read latency of a DDR2 chip can be found with the following formula:

$$Read\ latency = CAS\ latency + CAS\ additive\ latency$$

The *Column Access Strobe (CAS)* latency is the number of memory cycles that passes from the last part of a read command is received to the data is ready. The CAS additive latency is added to avoid that different commands use the same resources internally in the DRAM. This makes it possible to use every bus cycle to supply data when the reads are directed to adjacent banks. However, this effect is not taken into account in this work, and the details of the DDR2 scheme are beyond the scope of this report.

The Corsair TWIN2X2048-6400 memory chip [Cor07] has a CAS latency of 5 memory cycles and a CAS additive latency of 2 cycles. Consequently, the memory latency of this chip is 7 memory cycles. From the preceding discussion, we know that the memory clock frequency is 200 MHz. Since the processor frequency is 3200 MHz, a bus clock cycle is $\frac{3200MHz}{200MHz} = 16$ processor cycles. Consequently, the memory latency in processor cycles is $16 \cdot 7 = 112$.

The caches used in M5 do not enforce inclusion. In other words, the blocks kept in an L1 cache is not necessarily a subset of the blocks stored in the L2 cache. This was an implementation choice taken by the M5 developers and changing it would involve a considerable amount of work. Consequently, inclusion is not enforced in this report. However, this does create some problems for the cache coherence protocol implementation as discussed in chapter 4.

Parameter	**Value**
Transfer latency	4 processor clock cycles
Arbitration latency	5 processor clock cycles
Width	64 byte

Table 3.3: Baseline Interconnect Parameters

3.2.3 Interconnect Parameters

When investigating the interconnect in a CMP, a possible starting point is to investigate what types of interconnects are used in commercial CMPs. However, it is difficult to find accurate and credible information about this. For instance, the interconnect is part of the *Intel Advanced Smart Cache* used in the *Intel Core Microarchitecture*, but it is unclear how it is actually implemented [Int06]. However, Intel has disclosed that one core can retrieve data from a different core's L1 cache. Consequently, there must be some form of communication channel between the cores as well as between the cores and the L2 cache.

AMD's Athlon X2 has a private L2 cache for each core [AMD05]. Consequently, there is no need for an advanced interconnect between the core's L1 and L2 cache. A bus would suffice.

The IBM Power4 and Power5 CMPs use a crossbar interconnect called the *Core Interface Unit* between the private L1 caches and the shared L2 cache [KZT05]. In this case, each core has address and data lines that can be connected to each L2 cache bank. However, one core can only access one bank each clock cycle. In addition, each L2 cache bank has data lines that can be connected to all cores. Again, one L2 bank can only deliver data to one core each clock cycle. The crossbar model used in this report will be based on IBM's crossbar implementation. This model is further discussed later in this section. Sun's Niagara processor also use a crossbar interconnect [KAO05].

All interconnects consists of a number of transmission channels and a way of controlling access to these channels. The interconnects differ in how many transmission channels are available and how they are organised. The interconnect latencies used in this report are shown in table 3.3. By keeping the total delay the same for all configurations, it becomes easy to compare the queue delay experienced in each of the interconnects. In reality, these delays differ from interconnect to interconnect, and taking this into account is interesting further work. Furthermore, these values do not make much sense for butterfly interconnect as it does not have an explicit arbitration step. However, its parameters are chosen such that the baseline delay is approximately the same. This point is discussed further in section 3.2.3.3.

Kumar et al. [KZT05] have carried out comprehensive simulations of bus and crossbar interconnects. In particular, they present detailed calculations of the transfer and arbitration latencies of a 5 GHz CMP bus. They found that the transfer latency is 6 processor cycles if the wires are routed through the 8X plane. In addition, they used an arbitration latency of 8 processor cycles. By scaling these values to a bus clock frequency of 3 GHz, we get the values shown in table 3.3. For instance, $\lceil 6\ clock\ cycles \cdot \frac{3GHz}{5GHz} \rceil = 4\ clock\ cycles$ computes the transfer time. The sum of these delays is higher than the 5.5 bus cycles used in Intel's Advanced Smart Cache [Int06]. However, it is unclear how this value is computed. Consequently, Kumar's numbers will be used in this report. All channels are wide enough to accommodate one cache line each clock cycle. Again, this choice has been made to make it easier to compare the interconnects. In reality, the width of the channels would most likely be adjusted to provide maximum performance for a given area budget. Investigating these trade-offs is further work.

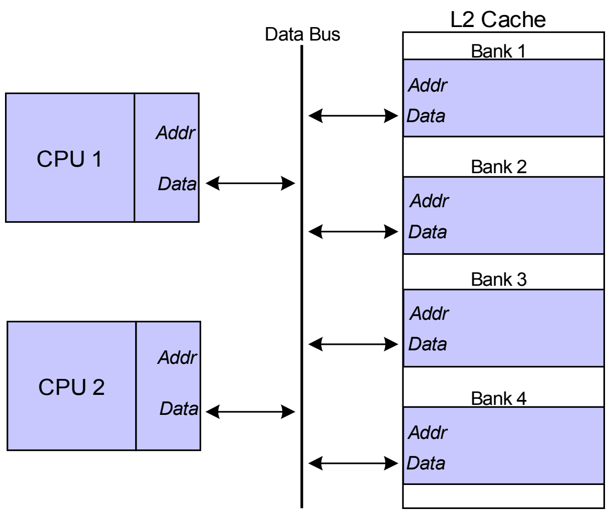


Figure 3.2: Shared Bus Model - Data Bus

The rest of this section will describe the following interconnect models:

- Split transaction bus
- Crossbar
- Butterfly
- Ideal interconnect

3.2.3.1 Split Transaction Bus Model

The data bus of the split transaction bus for a 2-core CMP and a L2 cache with 4 banks, is shown in figure 3.2. As there is only one transmission channel and it is shared among all cores and L2 banks, only one of these units can use the bus at one time.

A read transaction will need arbitration both when sending the address to the L2 cache and when the requested cache line is sent to the L1 cache. Note that this is an advantage as other transactions can use the bus while the L2 cache is fetching the requested cache line. When a unit is granted access, it gets both the data and the address bus regardless of whether it needs both. Consequently, only one arbitration unit is needed.

Since the bus is bi-directional, pipelining of transfers is not possible. However, this makes it easy to use a snooping cache coherence protocol.

3.2.3.2 Crossbar Model

The crossbar model used in this thesis is based on the IBM crossbar implementation used in the Power4 and Power5 CMPs [KZT05] and is shown in figure 3.3. As in the IBM scheme, the crossbar is really two crossbars: one in the L1 to L2 cache direction and one in the L2 to L1 direction. However, it differs in two respects. Firstly, all transmission channels have both address lines and data lines. Furthermore, there is one set of channels for the data cache and one set of channels for the instruction cache. Adding these connections significantly increases the area overhead of the crossbar. In addition, it makes the crossbar indicate an upper bound on the achievable performance with this type of crossbars.

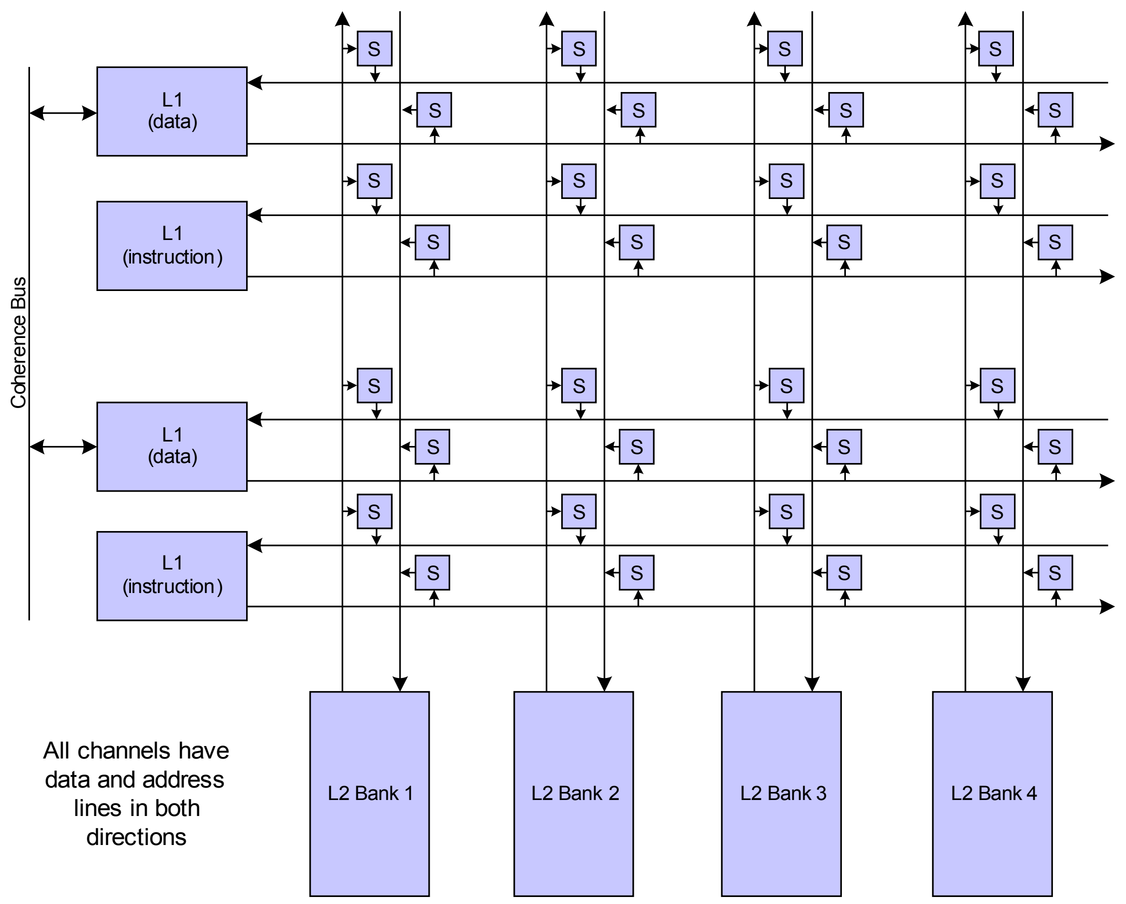


Figure 3.3: Crossbar Model

Since the transmission lines are uni-directional, pipelined transfer is possible. The boxes marked *S* in figure 3.3 are switches.

As mentioned, the IBM scheme only has address lines in the L1 cache to L2 cache direction. However, the L1 caches in the simulated CMP can have several outstanding requests. Consequently, we need a way to communicate which cache line is being delivered to the L1 cache. A simple way to do this is to have address lines in both directions. It is possible to find more area efficient ways of communicating this information but this simple solution will be used in this report.

Arbitration in the crossbar is carried out by setting the appropriate control signals for the switches and resolving contention. Since each cache only has one input line, only one unit can send to it at a time. This task is less work than the arbitration carried out with the split transaction bus. However, it is set to take the same amount of time to make it easier to compare the two interconnects. Furthermore, arbitration for the address and data lines are done simultaneously. Consequently, one request from a core will result in it being granted both the data and the address lines.

Data transfer from the private L1 caches to the shared L2 cache is the common case in a CMP. However, adding the possibility of transferring data from one L1 cache to another can be used to get faster inter-processor communication. As shown in figure 3.3, this is accomplished by adding a bus between the L1 caches. According to Kumar, this is the solution used in the IBM

Power processors but the details are unclear [KZT05].

3.2.3.3 Butterfly

The Butterfly interconnect differs from the other interconnects in that it is not helpful to divide the total delay into an arbitration latency and a transfer latency. The reason is that arbitration is done in each switch in the butterfly. If the data at two different switch input ports need the same output channel, one is granted access and the other is blocked. Consequently, the arbitration delay is set to 0 in the butterfly interconnect.

Furthermore, a butterfly is a really a class of interconnects. In this report, only radix 2 butterflies are investigated. In other words, all switches have 2 input ports and 2 output ports. Different radix configurations change the trade-off between switch delay and the number of hops needed to cross the network and is left as further work.

Setting realistic values for the delay of the different butterfly channels and switches requires making a detailed floorplan. Consequently, it is probably a bit too ambitious for this work. However, by making the assumption that the total transfer delay through all interconnects is approximately the same, we can investigate the impact of congestion. This is the main focus of this report. In other words, the end-to-end delay of the butterfly should be as close to the 9 processor clock cycle latency used in the other interconnects as possible.

The butterfly models for the two and four-core CMPs are shown in figure 3.2.3.3. The 8-core butterfly is not shown as it is relatively large and quite similar to the butterflies shown. Consequently, it is only discussed in the text.

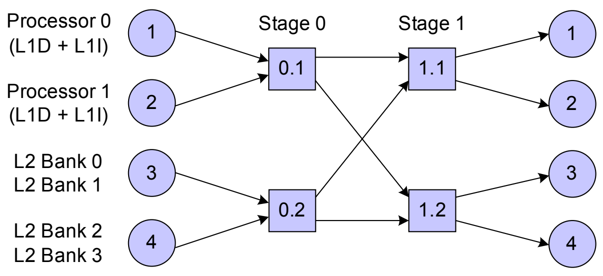
Figure 3.4(a) shows the 2-core CMP butterfly used in this work. Here, the data cache and instruction cache of one processor is mapped to one terminal node in the butterfly. In addition, two L2 banks are mapped to the same terminal node. An alternative way of mapping caches to nodes is to assign one node to each instruction cache, data cache and L2 bank. This results in a total of 8 nodes which is significantly more expensive in terms of chip area than a four-node butterfly. The downside of choosing the four-node butterfly is that congestion is probably more likely.

The four-core butterfly is shown in figure 3.4(b). Here, an eight-node butterfly is a perfect fit when each processor and L2 bank is a terminal node. The situation is more complex for the eight-core butterfly. Here, the chosen solution is to map each processor and L2 bank to a terminal node. However, this results in 12 terminal nodes while the next possible butterfly network has 16 terminal nodes. Consequently, four terminal nodes are left unconnected. Since all terminal nodes will not inject traffic, congestion is probably less likely than in a butterfly where all nodes are in use. An alternative scheme would be to add four more L2 banks. However, this would make it difficult to compare the butterfly to the other interconnects.

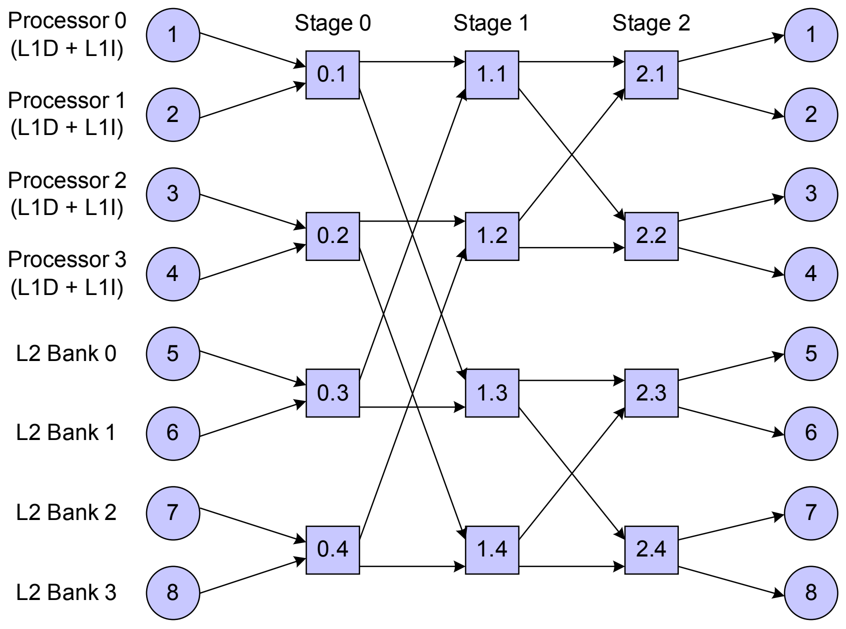
The transmission latency through a butterfly is given by the equation:

$$\begin{aligned} \textit{total latency} = {} & (\textit{number of switches traversed} \cdot \textit{switch latency}) \\ & + (\textit{number of channels traversed} \cdot \textit{channel latency}) \end{aligned}$$

The number of switches and channels traversed is the same regardless of origin and destination node. Consequently, this equation can be used to compute the end-to-end delay of a given



(a) 2 Core Butterfly Model



(b) 4 Core Butterfly Model

Figure 3.4: Examples of Butterfly Models

butterfly interconnect. For instance, in the 2-core CMP butterfly shown in figure 3.4(a), each request must traverse two switches and three channels.

The switch and channel latencies for the different butterflies are given in table 3.4. These values are chosen to get as close as possible to a total delay of 9 processor clock cycles. The reason is that this is the end-to-end delay of the other interconnects investigated in this report. For the 8-core butterfly where each request traverses 5 channels and 4 switches, this corresponds to a switch latency and channel latency of 1 processor clock cycle. For the other two butterflies, it is not possible to get exactly 9 clock cycles total delay by manipulating the switch and channel latencies. Here, the values are chosen such that the total delay is 10 clock cycles. Consequently, the butterfly has a small latency penalty compared to the other interconnect in the 2 and 4-core CMPs.

Parameter	Value
2-core channel latency	2 processor clock cycles
2-core switch latency	2 processor clock cycles
4-core channel latency	1 processor clock cycles
4-core switch latency	2 processor clock cycles
8-core channel latency	1 processor clock cycles
8-core switch latency	1 processor clock cycles
Arbitration latency	0 processor clock cycles
Width	64 byte

Table 3.4: Baseline Butterfly Interconnect Parameters

3.2.3.4 Ideal Interconnect

To estimate the potential performance improvement available for the different simulated benchmarks, it is useful to compare to an ideal interconnect. Here, transmission and arbitration takes the same number of clock cycles as in the bus and crossbar interconnects. The reason is that these are to some extent physical limits that can not be avoided. A request is granted access as soon as the specified delay has passed. In other words, there is no queueing delay. Consequently, the ideal interconnect gives an indication of the maximum possible speedup achievable from interconnect improvements.

3.2.4 Coherence Protocol Parameters

The possible cache coherence protocols was discussed extensively in chapter 2. Furthermore, the research questions focus on the private cache to shared cache interconnect. A directory protocol will create full freedom for the choice of interconnect.

The only problem is that the M5 simulator only supports snooping protocols. Consequently, M5 must be extended with a directory protocol. The Stenström protocol discussed in section 2.3.2 was chosen. The reason is that it optimises L1 to L1 communication by storing which other L1 caches have copies of a cache block in the owner's cache. Therefore, it is likely that this protocol will outperform traditional directory protocols that require a directory access before all cache to cache transfers.

The choices taken regarding the Stenström protocol are largely implementation choices that will be discussed in section 4.3. Consequently, the protocol will not be discussed further in this chapter.

3.3 Simulator

There are two simulators that are the most likely candidates for being used in this work:

- The SimpleScalar [ALE02] based IDI CMP Simulator developed by Haakon Dybdahl and Marius Grannæs
- The M5 Simulator [BDH+06]

The reason for the IDI CMP simulator being a candidate is that it was the simulator used in my 5th year project [Jah06]. Furthermore, a number of useful extensions to this simulator were

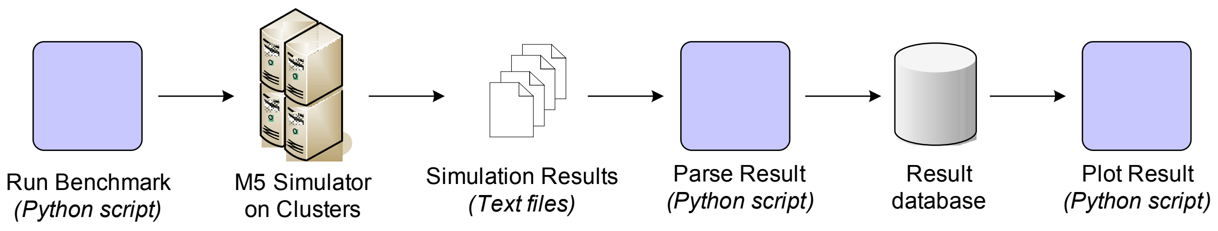


Figure 3.5: Typical Experiment Work-Flow

developed during this project. However, it is not trivial to get scientific benchmark suites like SPLASH-2 to run on this simulator. As this is crucial for the research described in this report, the IDI CMP simulator is probably not a good choice.

Arnt Jørgen Lande investigated possible CMP simulators for use at the *NTNU Computer Architecture Research Group (NCAR)* at IDI [Lan06]. He investigated the Rsim, Asim, SimOS, Simics, TFSim, SimFlex, GEMS and M5 simulators and concluded that M5 was the most appropriate for the NCAR group. The most important point for this work is that M5 supports cache coherent CMP simulation. Although this feature will not be used directly, some of the facilities needed to implement a coherence protocol will be available. Furthermore, pre-compiled binaries for the SPLASH-2 benchmark suite [WOT$^+$95] are distributed with the simulator.

Another feature of M5 is that it supports both full system simulation and system call emulation simulation. In full system simulation, the simulated program is run on a simulated operating system. Consequently, the behaviour of the operating system can be studied as well as the behaviour of the benchmark program. In system call emulation simulation, the operating system calls in the benchmark are executed by the operating system of the computer running the simulator. Because the operating system is not simulated, system call emulation simulation is faster than full system simulation. However, the simulation results will say nothing about the impact of the operating system on performance. This is not necessarily a drawback as it probably makes the simulation results easier to understand. Consequently, system call emulation simulation is used in this report. Sadly, some problems with M5's thread implementation discovered late in the work made choosing system call emulation a bad choice. These problems will be discussed in chapter 4.

3.3.1 Experiment Tool-chain

Carrying out experiments consists of running the simulator with different parameters, recording the results and analysing them. Furthermore, this procedure must often be iterated to gain a good understanding of the results. Consequently, there is a need for automating this process. This typical experiment work-flow is illustrated in figure 3.5.

The following processes have been automated with Python scripts:

- Running experiments with different parameters on the Clustis2 [Clu] and Norgrid [Nor] clusters.
- Retrieving selected values from the large simulation result text files and storing them in a SQLite [SQL] database.
- Plotting the results in a graphical form to ease analysis.

This experiment set-up turned out to be too complicated in practice. The main problem was that it was too cumbersome to make it retrieve other statistics than the ones identified when the system was first made. A better way was to use a number of short python scripts that retrieve statistics according to a regular expression. These short scripts can be easily modified to extract different simulation data.

3.4 Benchmarks

In this report, there is a need for two different benchmark types:

- Multi-threaded benchmarks make it possible to investigate application communication behaviour
- Single-threaded benchmarks can be combined to create multiprogrammed workloads. This makes it possible to investigate CMP interconnects without adding a cache coherence protocol.

In the assignment text, it is suggested to use the SPLASH-2 benchmark suite [WOT+95] to investigate application communication behaviour. Furthermore, it suggests that the SPEC CPU2000 benchmark suite [SPEa] should be used for sensitivity analysis. In other words, the SPEC2000 benchmarks should be used to verify that new schemes aimed at scientific workloads do not reduce the performance of single-threaded applications. The main problem with only using these two benchmark suites, is that they do not include commercial applications. However, scientific shared memory workloads and single-threaded workloads are represented. Consequently, this will limit the generality of the conclusions in this report to these application classes. In addition, the SPEC CPU2006 benchmark suite [SPEb] has recently been introduced. Consequently, this should replace SPEC2000 in future work.

3.4.1 SPEC CPU2000 Multiprogrammed Workloads

SPEC CPU2000 has been used extensively by the computer architecture research community. Sadly, Citron points out that it has also been misused [Cit03]. He makes the following points:

- Many papers do not simulate all applications in the benchmark suite. Consequently, average performance measures might be misleading. In this report, all benchmarks from the SPEC2000 suite are used.
- The benchmarks are not run to completion with the reference datasets.
- In 2003 when the paper was written, many researchers still used the retired SPEC CPU95 benchmark suite.

As Citron acknowledges, running the SPEC CPU2000 benchmarks to completion with the reference datasets is intractable in practise because of long simulation times. However, it is possible to run the CPU2000 benchmarks to completion with the reduced datasets. On the other hand, Perelman [PHC03] maintains that these datasets either put to much emphasis on program initialisation or are still too large to be simulated to completion.

Perleman advocates the use of *simulation points (SimPoints)* [PHC03]. Here, the benchmark is analysed to find some sections of the program that together represent the behaviour of the whole program. These sections are then simulated in detail. Since fast-forwarding is used between simulation points to keep for instance the cache state up to date, it is advantageous to choose simulation points early in the program's execution. This technique is not used in this report. The reason is that when simulating multiprogrammed workloads, the interaction between benchmarks is just as interesting as one benchmark on its own. SimPoints does not take this into account.

Another possible technique is to fast-forward past the initialisation phase of the benchmark. This fast-forwarding is followed by a warm-up phase to reduce the cold-start bias from execution history aware units such as caches and branch predictors. Then, all performance counters are reset and the benchmark is simulated for a fixed number of clock cycles. Although this technique removes the start-up phase from the results, the simulated part of the benchmark might still not be representative. However, since each SPEC2000 benchmark is part of a multiprogrammed workload it is the interplay of these benchmarks that is the main concern. By choosing both benchmarks in a workload and the number of fast-forward cycles per benchmark in the workload at random, we can get interesting workloads. This approach was inspired by the experiment methodology used by Haakon Dybdahl [DS07].

In this report, 40 workloads were created for 2, 4 and 8 CPUs by randomly choosing SPEC2000 benchmarks. Consequently, there are 120 workloads in total. All SPEC2000 benchmarks are represented in at least one workload for each number of CPUs. The benchmarks are fast-forwarded a random number of clock cycles between 1 and 1.1 billion and then simulated in detail for 100 million clock cycles measured from when the last benchmark finished fast-forwarding. There is no need for warm-up in the M5 simulator because the memory hierarchy is simulated in detail while fast-forwarding. The multiprogrammed workloads used in this report can be found in appendix A.

It is an open question to what extent these workloads will be representative of real-life workloads. However, they will probably serve to identify bottlenecks in the CMP memory system which is the main aim of this work. Consequently, this methodology will be used in the simulations in this report.

To compare two architecture schemes, it is important that the exact same part of the benchmark is simulated. Creating samples by specifying clock cycles is not ideal in this respect. If one scheme performs substantially better than another scheme, it might move into a different execution phase which makes the results less accurate. An alternative would be to use the number of committed instructions to create samples. The problem here is that the different applications in a workload commit instructions at very different rates. Consequently, a situation where one benchmark is running alone can occur. This is even worse than simulating slightly different parts of the benchmark. Therefore, samples are defined in terms of clock cycles in this work.

3.4.2 SPLASH-2 Communicating Workloads

To simulate the SPLASH-2 benchmarks [WOT+95] to completion takes less time than with the SPEC2000 benchmarks. However, it still takes too long to be practical. Consequently, the fast-forwarding techniques described for the SPEC2000 benchmarks can also be useful here. Sadly, flaws in the M5 thread implementation makes it difficult to use fast-forwarding. Furthermore, they limit how much of the SPLASH-2 benchmarks can be simulated. These flaws are related to the system call emulation thread implementation and will be discussed in detail in section 4.1.2. Sadly, the flaws are difficult to fix. The reason is that the benchmarks and the M5 specific pthreads library only compile with a library only found on a Tru64 UNIX operating system in an Alpha-based system.

The best way to get round these problems is to switch to full system simulation. However,

Benchmark	**2 CPUs**	**4 CPUs**	**8 CPUs**
Barnes	200 million	200 million	200 million
Cholesky	200 million	100 million	200 million
FFT	150 million	125 million	100 million
FMM	80 million	80 million	15 million
LUContig	200 million	150 million	100 million
LUNoncontig	200 million	100 million	75 million
OceanContig	180 million	60 million	30 million
OceanNoncontig	170 million	50 million	30 million
Radix	50 million	25 million	15 million
Raytrace	180 million	To completion	To completion
WaterNSquared	200 million	150 million	75 million
WaterSpatial	200 million	125 million	75 million

Table 3.5: Number of Instructions Simulated with SPLASH-2 Benchmarks

there was not enough time left to do this when the problem was detected. Consequently, a workaround is needed for the system call emulation mode. One possibility is to detect the problem and abandon simulation with an error message. Sadly, some benchmarks deadlock very early in their execution and this result in very short simulations. Another option is to detect the situation and start a waiting processor. This is most likely not the correct behaviour, but it makes it possible to simulate for a reasonable number of clock cycles. Consequently, the applicability of the results will be limited, but at least some results can be gathered and analysed.

Table 3.5 shows the simulation lengths chosen. These where found by trial-and-error and are chosen such that all simulation runs finish within 16 hours simulation time on the Norgrid cluster. Consequently, it is difficult to say if they are representative for the whole program execution or not. This further limits the applicability of the experimental results. However, some insight can be gained into the performance of parallel programs.

In addition to the benchmarks listed in table 3.5, *radiosity* and *volrend* are also part of the SPLASH-2 benchmark suite. These where not included with the binaries shipped with M5. Since a Tru64 UNIX operating system and Alpha processor is needed to compile the library, it is also difficult to compile these benchmarks for the M5 simulator.

The default problem size for the *LUContig* and *LUNoncontig* benchmarks is a 512×512 matrix with 16 element blocks. However, the precompiled *LUNoncontig* benchmark was compiled with a default matrix size of 128×128. The effect of this was that simulation terminated after 10 to 15 million clock cycles. This is to short for it to be useful as a benchmark. The problem was fixed by setting the matrix size to 512×512 with a command line parameter.

Chapter 4

Simulator Extensions

Ideally, the chosen simulator would provide all necessary features for investigating the problem at hand. However, this is rarely the case. Consequently, there is a need to extend the simulator with the needed features. This chapter describes the extensions to M5 simulator [BDH+06] developed as a part of this work. The complete simulator source code used in this work is attached as a digital appendix. Furthermore, the extension source code can be found in appendix C.

This chapter has the following outline:

- Section 4.1 contains a quick introduction to the M5 simulator.
- Then, section 4.2 describes the extensions to the private to shared cache interconnect.
- Section 4.3 describes the implementation of the Stenström directory protocol.
- Finally, section 4.4 describes the steps taken to make sure that the new components behave as specified.

4.1 The M5 simulator

The M5 simulator is a computer architecture simulator originally developed at the University of Michigan for modelling networked systems [BDH+06]. This requires a large number of features and results in M5 being a large software system. Consequently, a full review of the simulator is beyond the scope of this report. On the other hand, it is necessary to provide enough detail to understand the extensions presented in this chapter.

The M5 team has at the time of writing just released the third beta version of M5 2.0. Beta 1 of version 2.0 was released in the late autumn of 2006. Sadly, all beta versions lack coherence protocol support and this makes them poorly suited to communication research. Although a directory protocol would have to be developed anyway, it is preferable to work on a simulator where a coherence protocol is supported. The reason is that the developers have been thinking about coherence support while developing the simulator. Consequently, it was considered too risky to move to version 2.0, and the simulator used in this report is M5 version 1.1.

4.1.1 M5 Overview

Understanding the simulator extensions requires a basic understanding of how the M5 simulator works. This section starts with describing how the M5 simulator is configured. The reason is that

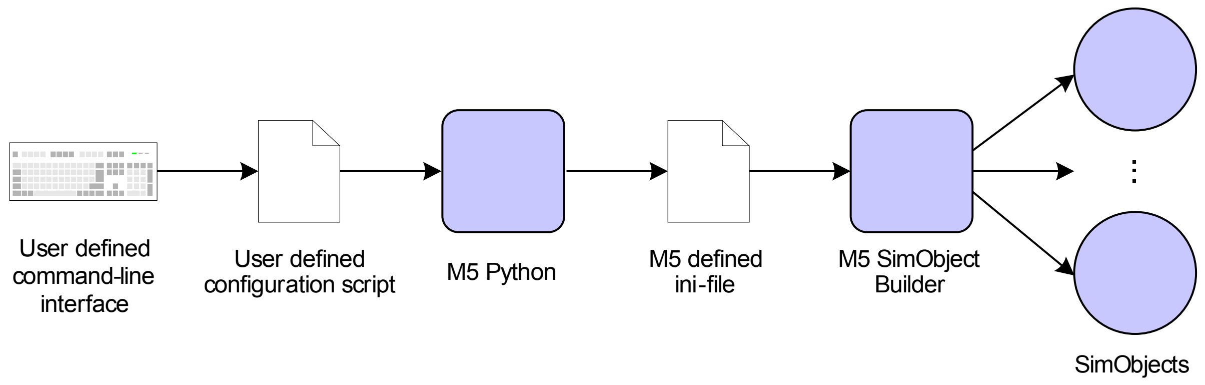

User defined command-line interface
User defined configuration script
M5 Python
M5 defined ini-file
M5 SimObject Builder
SimObjects

Figure 4.1: M5 Simulator Configuration

this is done in a special way that gives some insights into how the simulator is constructed. Then, the basic simulator architecture is described with a strong emphasis on the memory system.

4.1.1.1 M5 Simulator Configuration

A simulator is really a collection of components that can be interconnected in different ways. In M5, these components are known as *SimObjects*. More precisely, a SimObject is a simulator component that can be configured outside the simulator. If these components can be connected in many ways, it is more likely that the simulator can be used for many different research projects. However, this flexibility also makes the simulator difficult to use. This is less of a concern as researchers are willing to invest a considerable amount of time in understanding a simulator.

Figure 4.1 shows how M5 is configured. The simulator is run from the command line. Here, a number of user defined options are input to a user defined configuration script. This configuration is written in Python and sets the parameters for the different SimObjects and their connections. Each SimObject has an internal M5 python file that defines its external interface. The user defined configuration script uses this interface to set the parameters of the SimObjects. These parameters are then used to create a configuration tree which is flattened and written to an ini-style configuration file.

This configuration file is the interface between the input parsing part written in Python and the actual simulator which is written in C++. In the simulator, the configuration file is read and the configuration tree is recreated. The SimObject Builder then creates the SimObjects as specified in the configuration file. Each SimObject also has an object builder class associated with it. This class is created in a standardised way, and instantiates the simulator object with the parameters defined in the configuration file. When this configuration phase is finished, simulation is started.

Consequently, each SimObject has three classes associated with it: the actual C++ SimObject class, a C++ SimObject builder class and a Python class. This makes the elaborate simulator configuration scheme used in M5 possible. The main advantage of this scheme is that only the necessary configuration options are made available on the command line. In other simulators like for instance SimpleScalar [ALE02], all parameters are set with command line options. Consequently, a long command line is needed and this makes it difficult to start the simulator without a script. The downside of this scheme is that implementing new SimObjects takes time.

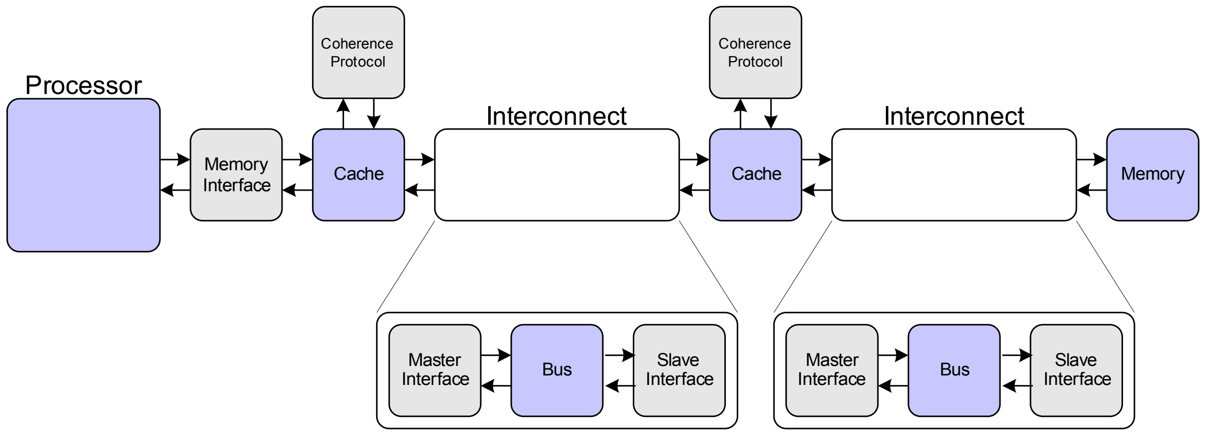


Figure 4.2: M5 Memory System Example

4.1.1.2 M5 Memory Hierarchy

Figure 4.2 shows an example of a memory system configuration in the M5 simulator. Here, a processor is connected to two levels of caches and a main memory. The blue boxes are SimObjects and the grey boxes are helper classes used in the simulator. The processor core is shown at the left side of figure 4.2. In M5, this can be a detailed or a simple processor. The simple processor is used for fast-forwarding while the detailed processor is used when measurements are taken. In M5 version 1.1, which is the version used in this work, the detailed CPU implements an *execute-in-fetch* model. In other words, an instruction is executed in fetch stage of the simulated pipeline and only timing analysis is done in later stages. In contrast, M5 version 2.0 uses an *execute-in-execute* model. This model provides accurate simulation of time dependent instructions as for instance synchronisation operations [BDH$^+$06]. The processor core communicates with the memory system through the *MemoryInterface* class which hides the memory system access details.

The cache implementation used in M5 is the same for first and second level caches. Furthermore, it makes heavy use of the C++ template construct. This creates difficulties for the M5 configuration system. Consequently, the components used by the cache are not SimObjects and their external interface is made available through the cache's external interface. This is the reason for the coherence protocol class in figure 4.2 not being a SimObject.

The M5 cache-to-cache interconnect is shown in the middle of figure 4.2. The only available interconnect is a split transaction bus. This bus is accessed through two interface classes, namely *MasterInterface* and *SlaveInterface*. The master term signifies that the interface is on the processor side of the interconnect. Conversely, the slave term means that it is on the memory side of the interconnect. The interconnect can be extended by modifying the files in the interconnect box in figure 4.2.

The main memory implementation used in this report is very simple. If a response is needed, it is returned after the user specified memory latency. For this work, this simple memory model is sufficient. In research that focuses mainly on main memory access, it is probably too simplistic and should be extended.

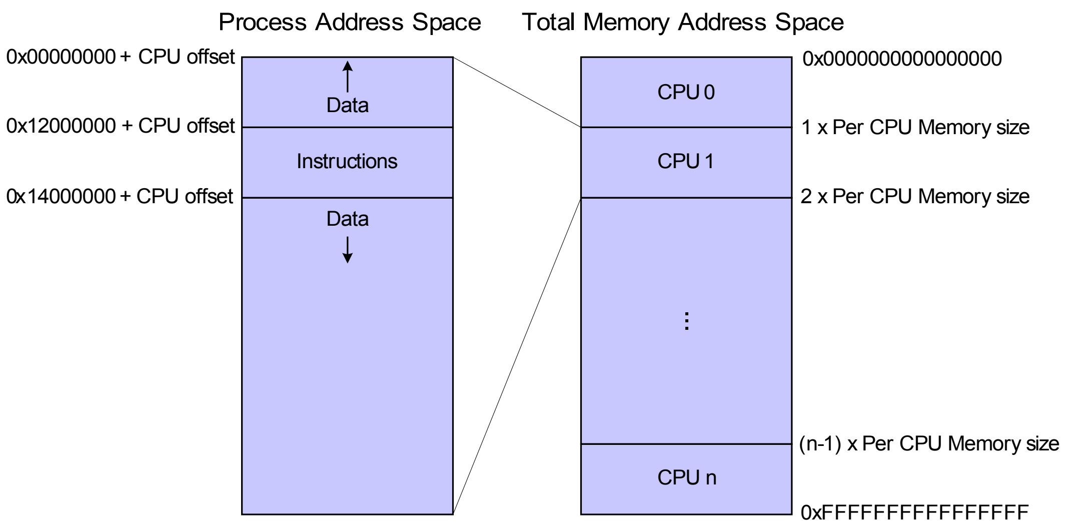


Figure 4.3: M5 Memory Layout for Multiprogrammed Workloads

4.1.2 Flaws in M5

Unfortunately, the M5 simulator is not perfect. Even worse, there are a few flaws in M5 that may influence the simulation results. The first flaw is due to the lack of address translation and is described in section 4.1.2.1. This results in all applications being mapped to the same address space when running multiprogrammed workloads. Consequently, they might warm up the cache for each other. The mapping of requests to L2 banks makes all requests go to one L2 cache bank when the scientific benchmarks are run. This problem is discussed in section 4.1.2.2. The third flaw is the thread implementation used in the system call emulation mode. This implementation has several problems, and these are discussed in section 4.1.2.3.

4.1.2.1 Multiprogrammed Workload Address Translation Flaw

In a multiprogrammed workload, all benchmarks end up using the same address space in M5. The reason is that there is no address translation support in the system call emulation mode. The benchmarks will still run correctly, but the statistics gathered may not be accurate. For instance, one processor can move a cache block into a shared cache. If this block is accessed from a different processor, the result is a cache hit. However, it should have been a cache miss.

Luckily, this flaw is easy to fix. The key idea is to intercept all memory requests and move them to a part of address space reserved for the processor the application is running on. When the request is returned to the processor, the address is changed back to the original address.

Figure 4.3 illustrates how this fix works. Basically, each processor is given a part of the total address space as given by the equation:

$$Per\ CPU\ Memory\ Size = \frac{Number\ of\ CPUs}{2^{64}}$$

Then, the start address and new address can be computed:

$$CPU\ offset = CPU\ ID \cdot Per\ CPU\ Memory\ Size$$

$$New\ address = Old\ address + CPU\ offset$$

The border between the different address spaces are guarded by assertions. This makes the simulator quit with an error message if a request is relocated into another processors address space. The address boundaries in figure 4.3 have been estimated by tracing which addresses the different caches access when running various SPEC2000 benchmarks. It seems like the address space between addresses 0x12000000 and 0x14000000 is used for instructions and that the rest of the address space is used for data.

This fix is not very realistic. The reason is that address translation is normally carried out by the hardware in cooperation with the operating system. Consequently, full system simulation is needed to do address translation realistically. In other words, this fix is sufficient for system call emulation simulation.

4.1.2.2 Address to L2 Cache Bank Mapping

In standard M5, each L2 cache bank is responsible for a contiguous part of the address space. For multiprogrammed workloads, the address translation described in section 4.1.2.1 is sufficient to distribute accesses across banks. However, all accesses are mapped to the same bank for the scientific workloads. This leads to a very low cache utilisation and can be fixed by using the least significant bits of the cache block address to select the bank. When the number of banks is a power of two, this is equivalent to using the modulo operator. This solution is used in the the scientific workload experiments in this report.

4.1.2.3 Faulty System Call Emulation Thread Implementation

The M5 system call emulation thread implementation actually contains at least two different flaws:

- Firstly, the thread implementation is prone to deadlocks. These are not protected by assertions and their only effect is very strange simulation results.
- Furthermore, some communication system calls are not implemented. Here, the only action taken is to write a message to the standard output stream.

These problems were discovered for the first time during this work and has been reproduced on an unmodified M5 simulator. Consequently, it is not a problem with the extensions developed in this work. The thread implementation only compiles on a Tru64 UNIX machine. As such a machine is not available to the NCAR group, it is impossible to fix these problems. According to Steve Reinhardt, the best option is to move to full system simulation where the thread implementation is better. Reinhardt is one of the main developers of the M5 simulator. The e-mail correspondence with Reinhardt can be found in appendix B.

According to Reinhardt, the pthreads library used in M5 allocates $P + 1$ threads where P is the number of processors. The extra thread handles management tasks. When the system call emulation thread support was developed, it was assumed that this thread was not used. Consequently, it was not created since this makes it possible to allocate one thread to each processor. In other words, there is no need for a thread scheduler. The deadlocks are probably due to all processing threads waiting for this manager thread.

To make things worse, the only effect of this problem is inaccurate simulation results, and this makes it difficult to discover. The reason is that the deadlock situation is not protected by any assertions. Consequently, it looks like the simulation finishes successfully if the experiment is terminated after a fixed number of clock cycles. As an example, consider the situation where this problem arises in configuration A and not in configuration B. Then, some architectural effect can be blamed for creating the performance difference which is really due to a simulator bug.

According to Reinhardt, moving to full system simulation is the best way to avoid these problems. Here, the faulty thread implementation is not used. Sadly, there was no time to change simulation mode when the problem was eventually diagnosed. Instead, the decision was made to go with the system call emulation mode and write the occurrence of the known problems to a tracefile. The idea is to use this problem trace to discard results where the problems might have influenced the results. Since some benchmarks deadlock very early in their execution, simply detecting the problem and exiting with an error message would lead to very short simulations. Therefore, the processor that has been waiting for the longest time is started when a deadlock is detected. The rationale is that when all processors are waiting, it is safe to start one of them. This can clearly lead to wrong results so we need to know how often this happens. Consequently, these events are noted in the problem trace file. Discussing the occurrences of these problems will be a central part of chapters 6 and 7 where the scientific benchmark results are presented. In addition, verifying the findings of this report in full system mode of M5 is very important further work.

4.2 Interconnect Extensions

This section describes the extensions developed to the cache-to-cache interconnect. First, the general software architecture is discussed with a focus on how the extensions communicate with the existing M5 code. Then, the new split transaction bus, crossbar, butterfly and ideal interconnects are discussed. The source code for the interconnect extensions can be found in appendix C.1.

4.2.1 Software Architecture

Figure 4.4 shows the classes that implement the interconnect extension. The blue classes in the figure have been developed in this work. The other classes are needed to glue the extensions to the rest of the simulator.

The interconnect extensions can be divided into two types: interconnects and interfaces. Both types inherit from the *BaseHier* and *SimObject* classes. In M5, all classes that can be configured from outside the simulator must inherit from the *SimObject* class. Although the M5 interfaces are not configured themselves, the *BaseHier* class is. The purpose of this class is to contain user configurable variables that define if data should be transported by the memory requests and if events should be scheduled in the memory system. In this report, event simulation is always on and data is not transferred. The reason for not transferring data is that the data is not used by the processor core anyway. Therefore, nothing is gained by explicitly transferring data.

M5 expects that the interconnect can be called through an interface. Both a master and a slave interface are needed. As mentioned earlier, the master term tells us that the interface is on the processor side of the interconnect, and the slave term signifies that the interface is on the memory side. These both inherit from the *InterconnectInterface* class. This class implements a few features that are needed by both the *InterconnectMaster* and *InterconnectSlave*

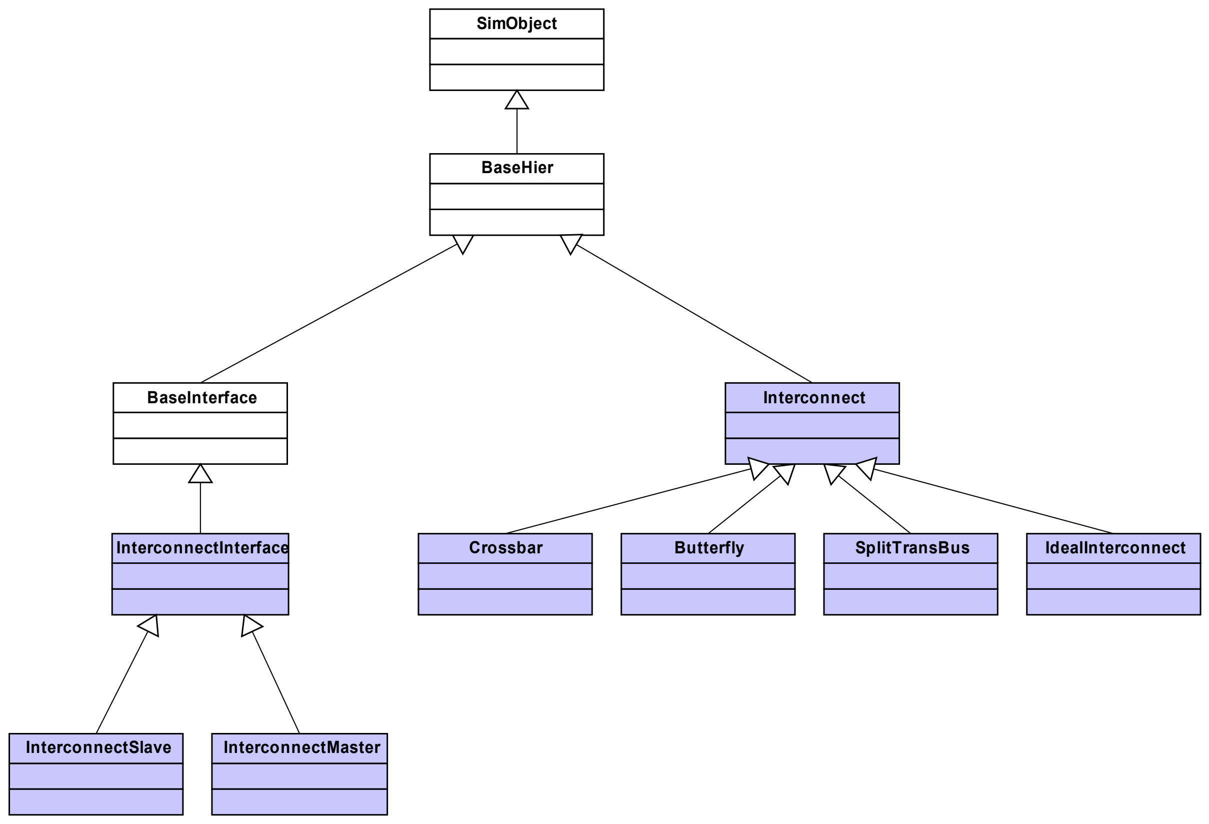


Figure 4.4: Interconnect Extension Software Architecture

classes. *InterconnectInterface* then inherits from *BaseInterface* which implements methods that are needed in all memory system interfaces in M5.

It is a great advantage to be able to switch between interconnects easily. Furthermore, it should be easy to add new ones. The software architecture in figure 4.4 meets both these design constraints. All interfaces have references to the abstract *Interconnect* class and this defines the methods that must be implemented by all interconnects. Furthermore, it declares the measurement variables used to record the performance of the interconnect. This ensures that all interconnects provide compatible statistics. Of course, the interconnects can add more measurement variables if these are needed. Adding a new interconnect is easy as only one new class that inherits from *Interconnect* and implements the abstract methods must be created. An additional advantage of this architecture is that all interconnects have the same configuration parameters which simplifies the configuration scripts.

Figure 4.5 should make the function of the different interconnect classes clearer. Here, a typical call sequence on a transfer from a L1 cache to a L2 cache is illustrated. The return of data from the L2 cache to the L1 cache is not shown. First, the L1 cache calls the request method of its *MasterInterface* and the master interface passes on the request to the *Interconnect*. The *Interconnect* object stores the request and schedules an arbitration event. When this arbitration event is serviced, the current requests are checked and at least one request is granted access. The specifics of the arbitration method depends on the interconnect used.

When the *Interconnect* object decides to grant access to a request, it calls the *GrantData* method on the *MasterInterface* object. The *MasterInterface* then retrieves the current request from the

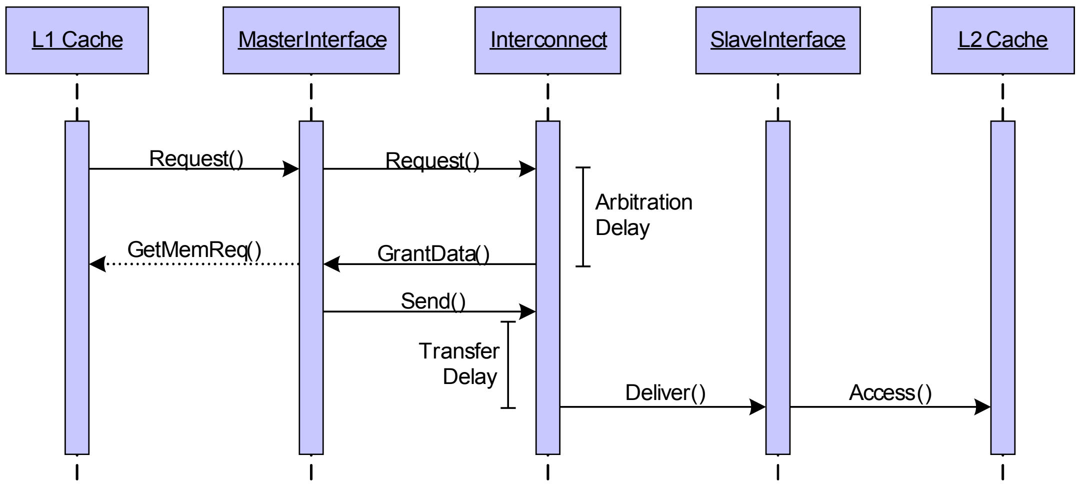


Figure 4.5: Interconnect Extension L1 to L2 Cache Transfer Example

L1 cache and calls the interconnect's *Send* method. This results in a deliver event being scheduled after the number of clock cycles specified by the transfer delay parameter. The transfer event calls the *Deliver* method in the slave interface. Then, the *Access* method for the *L2 Cache* object is called and the request is delivered.

The rest of this section is a quick introduction to the different interconnect extensions. The main part of the implementation is the method that does arbitration. Consequently, the discussion of each interconnect will focus on this method.

4.2.2 Split Transaction Bus

As noted earlier in this section, standard M5 has a split transaction bus interconnect. Consequently, it does seem strange to extend the simulator with an interconnect that is already supported. However, there are three good reasons to do just that. First, it is a good starting point to implement the simplest possible component. Then, this can be tested thoroughly and form the basis for other more advanced components.

Secondly, it is an advantage to have a clean interface to the other interconnects. This is especially important for simulator statistics as one need to measure the same thing for all interconnects. The measurements taken in the original M5 bus are very bus specific and not well suited to other interconnects. In addition, a common configuration interface simplifies the configuration scripts. Thirdly, it is an advantage to have implemented the key components yourself. Then, you know exactly how they work. This is a great advantage when analysing the results.

As mentioned in section 3.2.3, the split transaction bus does not implement pipelined arbitration or transfer. Consequently, carrying out one arbitration operation takes the number of clock cycles specified by the user. If a request is received while an arbitration is in progress, it is not serviced until the previous arbitration operation is finished. Therefore, the arbitration time for a given request can be more than the arbitration delay even if the bus is not very busy. Naturally, only one request is granted access to the bus after one arbitration operation.

The bus only supports an arbitration delay that is less than the transfer delay. In this case, the arbitration delay determines the delay of the arbitration operation and the transfer delay determines the transmission time. If the transfer time is longer than the arbitration delay, the arbitration delay is determined by the transfer delay. The reason is that the bus would be occupied when the arbitration is finished in this case.

4.2.3 Crossbar

The implemented crossbar simulates one channel from all L1 caches to all L2 banks and was shown in figure 3.3. In other words, each instruction cache and each data cache has its own channel to all L2 banks. Furthermore, there is a channel from all L2 banks to all L1 caches. Both arbitration and transmission is pipelined with each stage taking one processor clock cycle. L1 to L1 traffic is enabled by adding a split transaction bus between these. This crossbar was described in section 3.2.3 and is based on the design used by Kumar et al. [KZT05].

This design differs from the crossbar used by Kumar et al. in two ways. Firstly, Kumar et al. only use one channel from each core to each L2 bank. In this implementation, there is one channel for the data cache and one for the instruction cache. The reason for doing this is that the implementation in the simulator becomes easier. Furthermore, it creates an upper bound on the achievable performance with a crossbar. The downside is that the area overhead of this crossbar is significantly higher than the area overhead of Kumar's design. The other difference is that my implementation has both data and address channels in both directions.

When one L2 bank blocks, the crossbar blocks. An L2 cache bank will block if it can not guarantee that it has space to store the state of an additional cache miss. Since the crossbar blocks as well, access to all banks are blocked when one cache bank blocks. This is realistic if we assume that there is no buffering in the crossbar. Consequently, a request might go to a blocked bank and it is not possible to guarantee delivery. However, changing this implementation choice and exploring the results is possible further work.

4.2.4 Butterfly

The butterfly is an implementation of the model discussed in section 3.2.3.3. Because of time constraints, it can only handle the input combinations used in this work. In other words, only radix 2 butterflies with 2, 4 and 8 processor cores is implemented. The blocking behaviour is identical to the crossbar's blocking behaviour as this makes it easier to compare them to each other. Removing the blocking restriction and supporting other butterflies is left as further work.

4.2.5 Ideal Interconnect

The ideal interconnect is an interconnect which can issue an unlimited number of requests in parallel. However, these requests experience an arbitration delay and a transmission delay. Consequently, it gives an upper bound on the performance attainable when a request is never queued due to interconnect capacity constraints. The implementation keeps a sorted list of requests and delivers them when the specified number of clock cycles has passed. Again, the whole interconnect blocks if one L2 bank blocks.

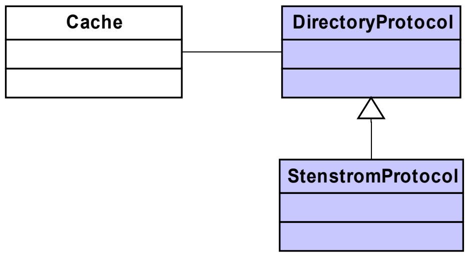


Figure 4.6: Directory Protocol Extension

4.3 Cache Coherence Protocol Extensions

This section describes the implementation of the Stenström directory-based cache coherence protocol [Ste89] discussed in section 2.3.2 and has the following outline:

- Section 4.3.1 discusses the software architecture chosen for the directory protocol implementation.
- The non-blocking caches implemented in M5 create a few additional challenges which are discussed in section 4.3.2.
- Protocol race conditions are often cited as the most challenging part of implementing a directory-based protocol. Section 4.3.3 presents two examples of such races in the Stenström protocol and describes how they are handled in this implementation.
- Finally, section 4.3.4 discusses a few additional implementation choices.

The source code for the coherence protocol implementation can be found in appendix C.2.

4.3.1 Software Architecture

Figure 4.3.1 shows the software architecture of the directory protocol implementation. The *Cache* class has a pointer to a *DirectoryProtocol* class. Furthermore, the cache implementation was modified to call the directory protocol methods at certain places. The abstract *DirectoryProtocol* class defines an interface which the classes that inherit from it must implement. Consequently, it is easy to interface a new coherence protocol with the rest of the simulator.

As noted earlier, the *DirectoryProtocol* does not inherit from *SimObject*. Consequently, it can not be configured directly from outside the simulator. The reason is that the M5 cache implementation uses the C++ template construct heavily which result in it not interfacing cleanly with the simulator configuration scripts. The original M5 code works around this problem by using the cache's external interface to configure the objects that are associated with it. For instance, the existing prefetcher and snooping cache coherence protocols are configured in this way. The directory protocol implementation uses the same workaround.

4.3.2 Handling Non-blocking Caches

The memory system used in the M5 simulator uses non-blocking or lockup-free caches [Kro81]. This means that the state of the miss is stored in special-purpose register called a *Miss Status*

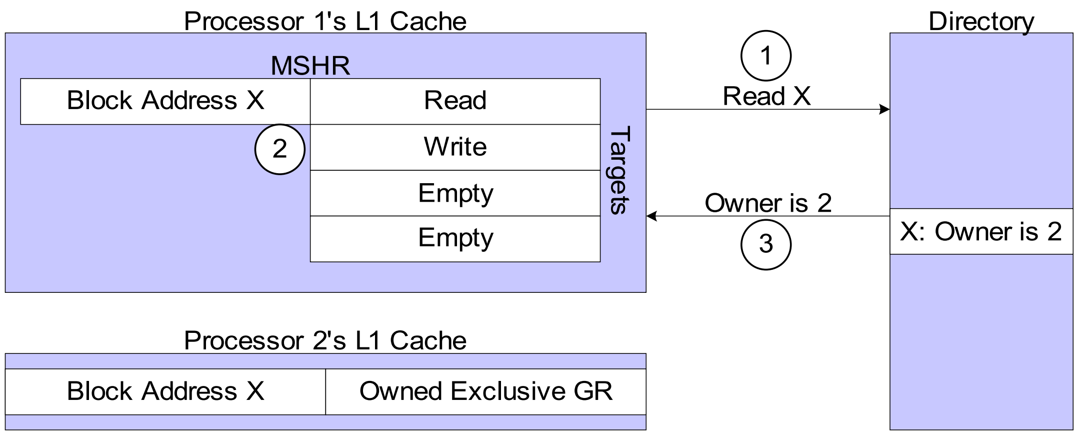


Figure 4.7: Non-blocking Cache and Coherence Challenge

Holding Register (MSHR) and that the cache continues to service hits and misses even if several earlier requests have missed in the cache. If there are more than one miss to the same cache block, this information is stored in the MSHR. However, a new request is not sent to the next memory hierarchy level. The number of misses to the same cache block that can be handled without blocking is called the number of targets of a MSHR in M5.

There is a limited amount of registers and targets. Consequently, the cache must *block* if it can not guarantee that it will have space to allocate an additional miss. In this case, the cache does not accept any new requests until an outstanding request has been serviced. There are two situations in M5 where the cache might block. Firstly, all MSHRs might be in use. In this case, the cache will be unable to service a miss to a cache block which has no previous miss allocated. Secondly, a MSHR might use all its targets. Here, the cache can not service a new miss to this cache block.

Non-blocking caches create additional complications for a cache coherence protocol. The reason is that a write request can be hidden behind a read request. If the read does not result in the requesting cache becoming the owner, the write must not complete.

Figure 4.7 exemplifies this problem. The numbers in the figure correspond to the numbers in the following list:

1. First, processor 1 has a read miss to block X. This results in the allocation of one MSHR. In addition, a read request is sent to the directory. Recall from section 2.3.2 that the directory is co-located with the shared L2 cache.
2. Processor 1 writes to block X before the read has completed. Since there is an outstanding miss for this block, this write is allocated as a target in the MSHR.
3. The directory answers that block X is owned by processor 2. This results in the read being redirected to processor 2's L1 cache. However, care must be taken to avoid that the waiting write is carried out on the received block. If the write goes through, it is not noticed by the directory protocol and is lost.

There are a number of possible ways of avoiding this erroneous behaviour:

- Firstly, we could use blocking caches. In this case, there is only one outstanding miss at any time and the problem disappears. The downside is reduced performance and reduced pressure on the interconnect. Since this work is aimed at investigating communication

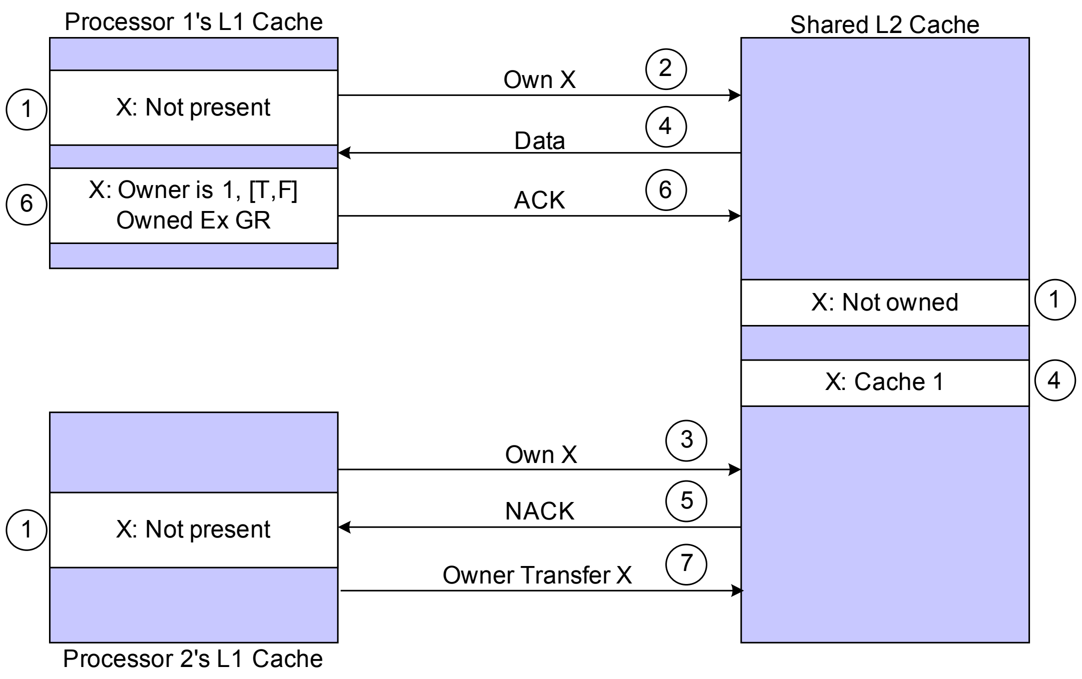


Figure 4.8: Two Processors Request Block Ownership Simultaneously

performance, this option would severely limit the applicability of the results.

- Reads and writes to the same address could use different MSHRs.
- Additional hardware can be used to detect the hazard situation shown in figure 4.7 and resend the hidden write request if needed.

Applications that do not share data should not be slowed down by the cache coherence protocol. This makes the option of allocating different MSHRs to reads and writes less attractive. The reason is that it will increase the number of memory requests in the system. Consequently, the strain on the interconnect and the shared cache will be larger than necessary.

The implemented protocol detects the hazard and resends the write request. This ensures that requests are only resent when it is necessary. Furthermore, the performance non-communicating applications are not affected.

4.3.3 Possible Race Conditions

During the implementation of the Stenström protocol, many race conditions were observed. These race conditions are rare [MHW03]. For example, some race conditions were observed after simulating for only a few million clock cycles with 2 processors while others were only observed after over 100 million clock cycles with 8 processors. Consequently, high performance is not a key issue when handling them. However, they must be handled correctly. This section presents two observed races and describes how they are handled in the protocol implementation.

Figure 4.8 illustrates a race condition where two processors attempt to acquire ownership of a block simultaneously. The protocol actions are described in the following list:

1. At first, block X is not present in any of the caches and the L2 cache has recorded the

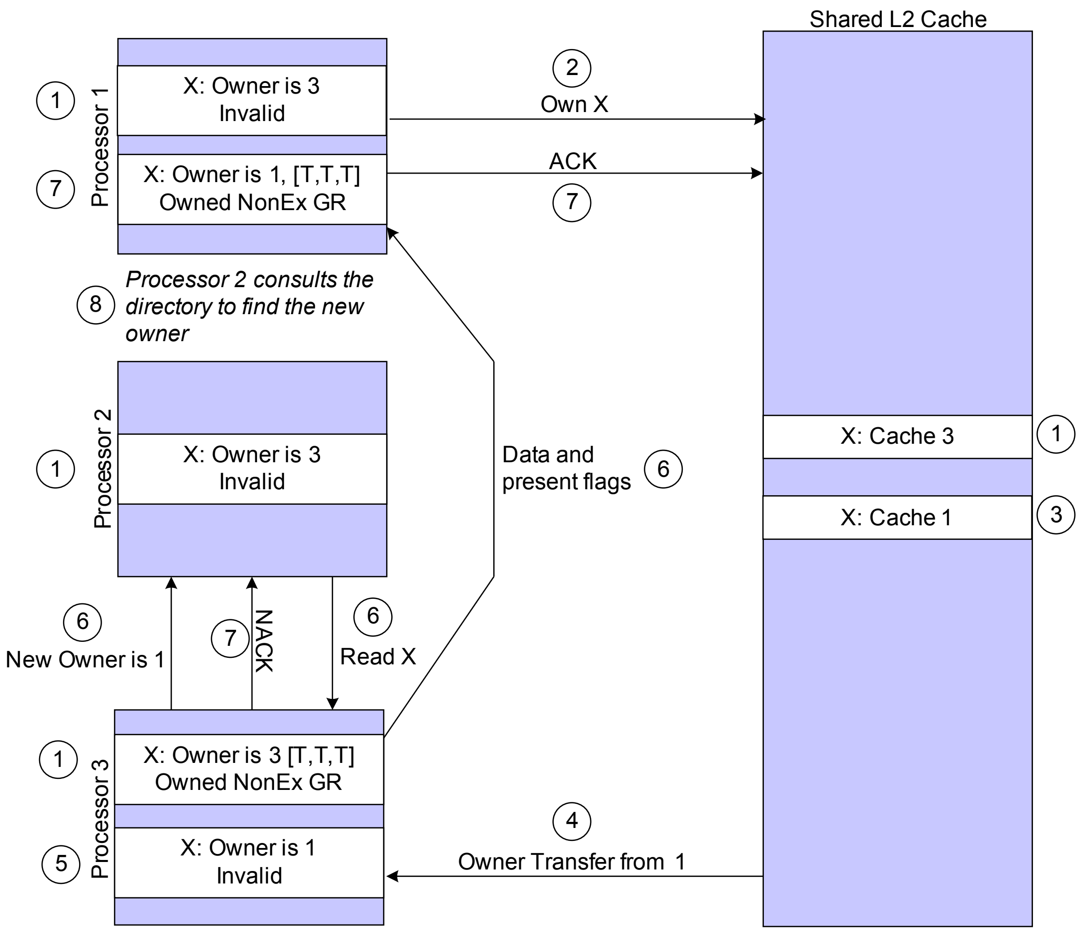


Figure 4.9: A Processor Issues a Redirected Read while Owner Transfer in Progress

block as not being owned.
2. Then, processor 1 issues a owner request for block X.
3. Moments later, processor 2 issues a owner request as well.
4. The directory receives the owner request from processor 1 first and stores processor 1 as the new owner. Furthermore, it sends a message to processor 1 with the data for block X.
5. Then, the request from processor 2 is received by the directory. Since the directory must create a globally consistent ordering of the requests, the owner request can not be granted until processor 1 has finished its write to block X. Consequently, a NACK message is sent to processor 2.
6. Processor 1 receives the block and stores it in its cache. In addition, it sends an ACK message to the directory to signify that the owner request is finished. This tells the directory that it can accept other owner transfer requests for block X.
7. When processor 2 reissues the request, a normal owner transfer operation is carried out.

In summary, two or more simultaneous owner transfer requests are handled by granting one and sending NACK messages to all other processors. This way of handling races is for instance described by Hennessy and Patterson [HP03].

Figure 4.9 illustrates a more complicated race condition where a processor issues a redirected read while two other processors are in the middle of an owner transfer. The example uses three processors as this is the minimum number of processors needed for this case to occur.

The following list describes the situation:

1. First, block X is owned by processor 3 and both processor 1 and processor 2 have invalid copies of it.
2. Processor 1 wants to write to the block and issues an owner transfer request.
3. The directory receives the request and sets processor 1 as the owner of block X.
4. A message is sent to processor 3 that instructs it to send the updated block data to processor 1.
5. Processor 3 prepares the message and invalidates its own copy.
6. Processor 3 sends the block data and present flags to processor 1. Furthermore, it sends a message to processor 2 to inform it that processor 1 is now the owner of block X. At the same time, processor 2 attempts to read block X and issues a redirected read to processor 3. As far as processor 2 is considered, processor 3 is still the owner of block X.
7. Processor 3 has sent the updated data to processor 1 and can not answer processor 2's request. Consequently, it answers by sending a NACK message. At the same time, processor 1 has received the data, updated its cache and sent an ACK message to the directory.
8. Processor 2 receives the NACK message. Now, there are two possible implementations. We can assume that the new owner information has been received and redirect the request there. However, the updated data and present flags might not have been received yet. Consequently, it is safer to consult the directory first and then forward the request. This safe option is taken in this implementation.

4.3.4 Implementation Choices

Implementing a directory coherence protocol carries with it a number of problems. This section discusses a few of them:

- The M5 cache implementation does not enforce inclusion and this must be handled.
- M5 supports software prefetches.
- Some protocol operations require accessing the cache. What is the latency of these operations?
- Finally, requests that are issued to acquire ownership of a given cache block are not buffered in the cache in this implementation. Consequently, multiple owner transfer requests for the same cache block to the same cache can circulate in the interconnect at the same time.

The inclusion property is that all cache blocks stored in the L1 cache are also stored in the L2 cache [HP07]. The effect of not enforcing inclusion is that the coherence state of a cache block can not be stored together with the L2 cache data. Fixing this requires decoupling the directory from the L2 cache data storage. This is easy in a simulator as one can create a map that stores the owner of a given cache block. Solving the problem in hardware requires an upper bound on the storage required. The worst case storage requirement is that all L1 caches are full and that they own every block they store. In other words, there is no sharing. Then, this structure can be stored in an SRAM memory. Consequently, this implementation assumes that such a memory is available. Of course, the area requirements of this memory can be reduced by using techniques inspired by limited or chained directories as discussed in section 2.3.2 and by Chaiken et al. [CFKA90].

A few of the SPLASH-2 benchmarks issue software prefetches. If these miss in the L1 cache, they are simply discarded. This is not a problem if they hit in the cache. However, there might be a performance penalty on discarding them when they miss in the L1 cache. One problem is that it is difficult to know whether the cache should request ownership of the block or simply register itself as a sharer in this case. Exploring different strategies with software prefetches is left as further work.

Some protocol actions require retrieving data from a cache. For instance, when a read is redirected to the owner cache, the updated data must be retrieved from this cache and returned to the requesting processor. The latency of this check is set to the hit latency of the cache. This is realistic if the only overhead associated with this look-up is retrieving the data. However, this request might happen at the same time as the owner processor requests data from the cache. If the cache only has one port, one of the requests must be delayed. Investigating the impact of such real world effects is also possible further work.

In this protocol implementation, owner transfer requests are not buffered in the caches. As only one cache can be the owner at one time, the directory will only grant one and send *Negative Acknowledgement (NACK)* messages to the other caches. Consequently, the writes are processed in a globally consistent order. However, this lack of buffering creates the possibility that one processor can issue more than one owner transfer request for a given block. This might lead to it requesting ownership to a block that it already owns, and this must be handled by the protocol. An alternative solution would be to only allow one outstanding owner transfer request per block and block the cache if it tries to issue more requests. Investigating this solution is left as further work.

4.4 Simulator Extension Testing

This section discusses how the simulator extensions have been tested. When writing simulator code, testing is especially important for two reasons. Firstly, programming errors can create inaccuracies that only show up in the simulator results. These inaccuracies might be difficult to discover and can lead to erroneous conclusions. Secondly, the simulators often run for a long time to gather the needed results. An implementation that fails unpredictably after a few hours of simulation is consequently not acceptable.

The testing carried out in this work is mainly based on placing assertions in the code. These assertions contain a boolean expression that is true if the simulator is in a correct state. If this expression evaluates to false, the simulator exits with an error message. Consequently, the simulator has executed a correct sequence of states if simulation finishes without errors. Of course, this only helps if there are assertions that guard against the error that arises. Therefore, the simulator extensions contain a large number of assertions.

A test script was developed to facilitate automatic testing of the simulator. This script simply runs all available benchmarks in all relevant configurations. When one run finishes, it checks if the simulation finished successfully. If the simulator exited with an error message, the simulator output is written to a file and the test is reported as failed. As the simulator code matures, the parts of the benchmarks simulated in a test are increased.

This approach was very successful as only two simulator extension bugs were found during simulation on the clusters. Furthermore, these errors were caught by assertions and did not result in wrong simulator statistics being reported.

This testing scheme created a need for additional computing power. Therefore, the *aocdev* computer has been used as a dedicated test machine. This computer is normally used to test new features added to the Age of Computers (AoC) teaching system before they are added to the production server. Luckily, there has not been any AoC development this term so *aocdev* has been available to run tests.

Chapter 5

CMP Performance with Multiprogrammed Workloads

Single-threaded programs will probably be common for many years to come. Consequently, it is important that CMPs perform well with multiprogrammed workloads. This section investigates the performance of multiprogrammed workloads consisting of SPEC2000 [SPEa] benchmarks with a split transaction bus and a state-of-the-art crossbar interconnect. CMPs with 2, 4 and 8 cores are considered.

The workloads are created by randomly adding SPEC2000 benchmarks to each workload. The only guidance provided to the random selection is that a SPEC2000 benchmark must be present in at least one workload for each CPU count. Each benchmark is fast forwarded for a random number of clock cycles between 1 and 1.1 billion. Then, detailed simulation is carried out for 100 million clock cycles after the last benchmark has finished fast forwarding. 40 different workloads were generated for each number of cores, and the workloads can be found in appendix A.

Figure 5.1 shows the number of L1 and L2 cache misses the SPEC2000 benchmarks encounter when they are run on a single core processor. The main point of this graph is that most SPEC benchmarks have relatively few misses. However, *gap*, *ammp* and *apsi* all miss in the L1 cache in more than 7.4% of their instructions. Consequently, they are classified as cache intensive. The experimental analysis will focus specifically on these applications as it is natural to expect that they will be sensitive to the properties of the interconnect. Of course, the sections of the benchmarks that are simulated with the single-core processor are not identical to the sections used in the CMP simulations so the actual number of misses might differ. However, the qualitative trends should hold.

As mentioned, the analysis will focus on *gap*, *ammp* and *apsi*. In addition, other benchmarks will be discussed if the results make this necessary. However, an exhaustive analysis of the performance of all benchmarks will be too lengthy to fit in this report.

The performance measurements will be reported relative to the performance of an ideal interconnect. In this context, an ideal interconnect is an interconnect where each request experiences a transmission delay and an unlimited number of requests can be sent in parallel as described in section 3.2.3. Consequently, it represents a golden standard that the other interconnects can be compared to. In addition, the performance measurements are reported per benchmark. The number reported is the harmonic mean of the *Instructions per Cycle (IPC)* this benchmark achieved in each workload. This makes it possible to compare the performance of a benchmark in

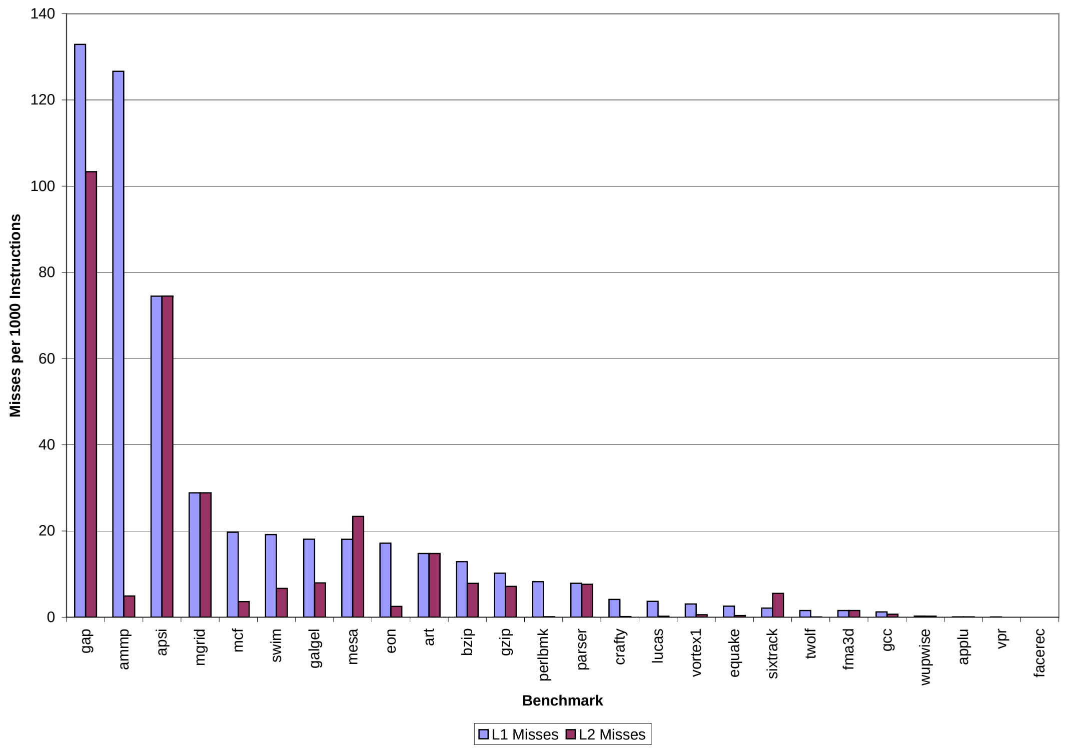


Figure 5.1: Single-core SPEC Benchmark Cache Performance

the different CMP configurations. Since IPC is a rate, the harmonic mean is used [Smi88, JM95]. Furthermore, the benchmarks are sorted after their bus performance in the figures so that similar results are shown next to each other. This makes the general trends easier to see but comes at the cost of making comparisons between the same benchmark in different CMP configurations more difficult.

The rest of this chapter has the following outline:

- Section 5.1 discusses the results from the 2-core CMP.
- Section 5.2 presents the 4-core CMP results.
- Finally, section 5.3 discusses the 8-core CMP results.

Each section will first discuss the performance of the miss intensive *apsi*, *ammp* and *gap* benchmarks. Then, benchmarks that either perform better or worse than expected will be discussed.

5.1 2-core CMP Configuration

Figure 5.2 shows the per benchmark average IPC relative to the ideal interconnect for the 2-core CMP. As expected, the bus performs a lot worse than the crossbar and the ideal interconnect in some cases.

This section has the following outline:

- Section 5.1.1 discusses the *apsi*, *ammp* and *gap* benchmarks.
- Then, *art*, *wupwise*, *swim*, *eon* and *fma3d* are discussed in section 5.1.1. The reason for choosing these benchmarks, is that *art* performs worse than expected while the realistic

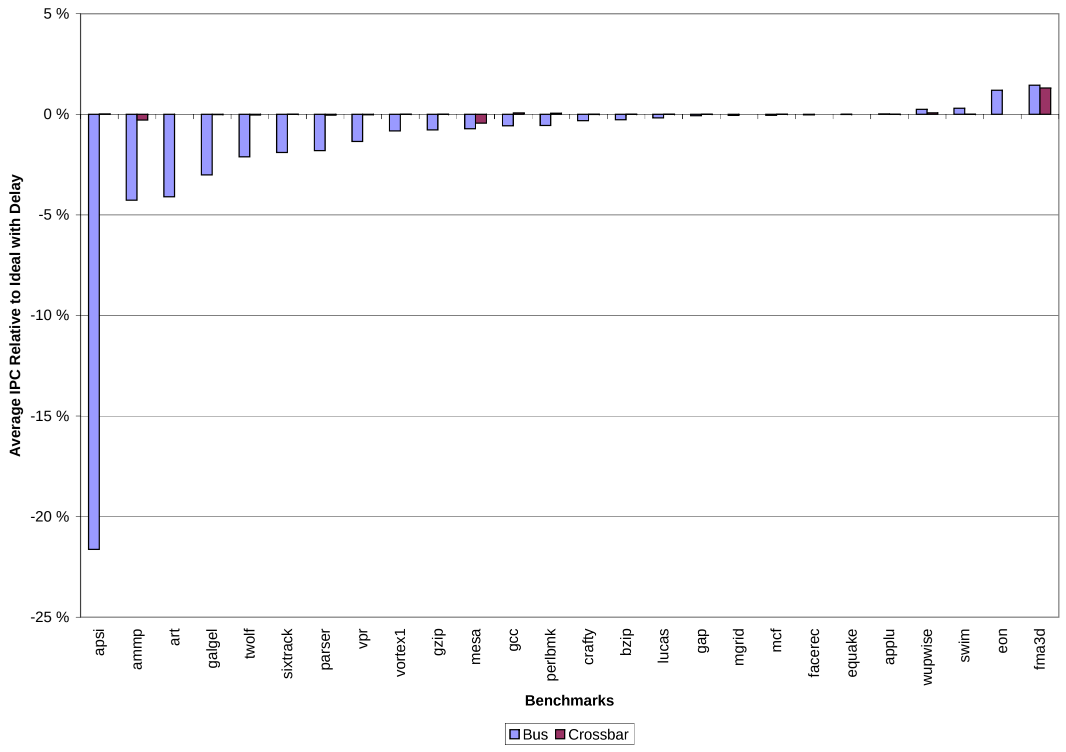


Figure 5.2: 2-core CMP Interconnect Performance

interconnects outperform the ideal interconnect in the other benchmarks.

5.1.1 Miss Intensive Benchmarks

The cache miss intensive *apsi* benchmark has the worst bus performance with a speed degradation of over 20%. The reason for this poor performance is severe congestion in the L1 to L2 bus. Workload 18 is an example of this and the workload where *apsi* performs worst. With the bus interconnect, a request spends on average 10.4 clock cycles in queue. In comparison, the average queue time is 0.15 clock cycles with the crossbar interconnect.

The miss intensive *ammp* and *gap* benchmarks do not experience a performance degradation proportional to their L1 cache misses. In the *ammp* case, workload 18 can offer some insights. Here, *ammp* is run together with *apsi* and experiences a considerable speed degradation compared to the other workloads it is a member of. In this case, the bus, crossbar and ideal interconnects are all slowed down. The reason is that the memory bus is badly congested. Consequently, the L1 to L2 interconnect only has a secondary effect on performance. However, this effect is large enough to create a small performance difference between the crossbar and ideal interconnects.

All interconnects have identical performance for the *gap* benchmark. This is not expected, as the large number of cache misses should put considerable strain on the interconnect. By carefully studying the simulator statistics, it turns out that these misses does not reach the interconnect in sufficient numbers. The reason is that the L1 cache blocks. This will happen if there are 4 misses to different cache blocks or 4 misses to the same cache block in the L1 cache. In the *gap* benchmark this becomes the predominant bottleneck. Consequently, the misses are injected into the interconnect in bursts, but these bursts have sufficiently large idle periods between them.

Therefore, even the bus is able to handle the traffic load.

5.1.2 Other Results

The performance degradation of *art* is due to the low bus performance in workload 9. Here, *art* is run together with *twolf*. Although these benchmarks do not miss too much by themselves, the sum of the misses creates the problem. Profiling the bus utilisation reveals that it occupied 30% of the time for most of the detailed simulation. However, at some points the utilisation reaches around 60%. This high utilisation results in long queues and long wait times. Consequently, the combined miss behaviour of *art* and *twolf* are the cause of their low performance.

The bus and crossbar interconnects do perform better than the ideal interconnect in some cases. This might seem counter-intuitive at first. The key idea is that even if the interconnect is ideal, other parts of the processor are not ideal. If the performance difference due to the interconnect is small, then small changes in the timing of misses can make other parts of the processor perform better or worse.

This is the case for *eon*, *fma3d* and *swim*. Here, all memory system statistics indicate that the ideal interconnect is better than the crossbar which again is better than the bus. However, the bus interconnect leads to less speculative instructions being rolled back in the processor core. In other words, the crossbar and the ideal interconnects execute further down a wrong path. The reason is that memory accesses are delivered faster. In this case, waiting for memory accesses to complete is better since this reduces the number of speculative instructions issued. Consequently, fewer instructions need to be rolled back and the bus outperforms the other interconnects.

The good bus performance of *wupwise* is due to the blocking behaviour of the L1 cache. Here, the ideal interconnect has the largest number of cycles where the cache is blocked. Recall that when a cache is blocked, it can not service any requests. In the bus case, the processor execution is slowed down due to longer interconnect delays. Consequently, it generates memory requests at a slower rate that with the ideal interconnect. This slow rate result in the sequence of misses that cause cache blocking to be further spaced in time. The misses might even be slowed down so much that the first miss is serviced before the blocking miss arrives. This effect makes *wupwise* perform better with a bus interconnect than with the ideal interconnect.

5.2 4-core CMP Configuration

Figure 5.3 shows the interconnect performance on a 4-core CMP. There is a large performance degradation with the bus interconnect while the crossbar performs close to the ideal. In other words, the trends from the 2-core experiment still hold, but the performance impact of the bus is larger. A new trend is that the crossbar in some cases performs slightly better than the ideal interconnect. The reason is that the crossbar can avoid some cache blocking by reducing the memory access rate as discussed in the previous section.

This section has the following outline:

- Section 5.2.1 discusses the miss intensive *apsi*, *ammp* and *gap* benchmarks.
- Then, section 5.2.2 discusses the surprising performance degradation of *vortex1* and *parser* as well as the counter-intuitive results from the *wupwise*, *gcc*, *applu* and *facerec* applications.

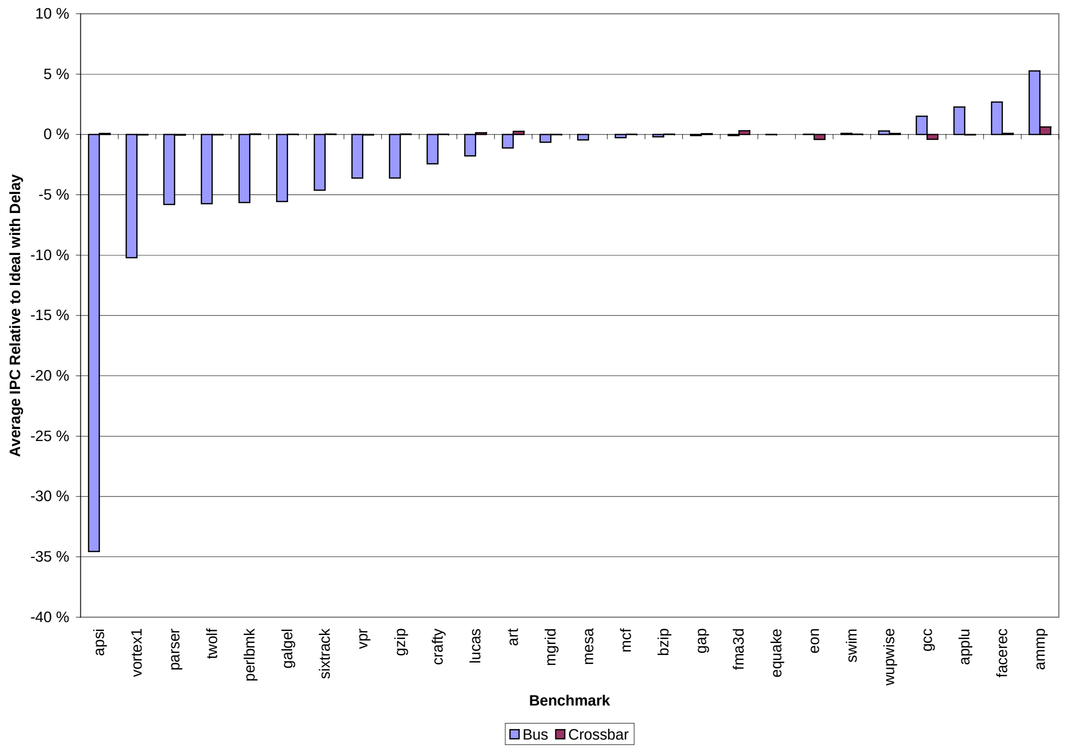


Figure 5.3: 4-core CMP Interconnect Performance

5.2.1 Miss Intensive Benchmarks

Apsi has the largest performance degradation for the bus interconnect of all the benchmarks. Again, this is due to bus congestion. For instance, the average queue time with the bus interconnect was 20.1 clock cycles and only 0.1 with the crossbar in workload 40.

In this experiment, *ammp* is over 5% faster with the bus interconnect than with the ideal interconnect. Given the large number of misses in *ammp*, it is very unlikely that this is actually due to the bus being fast. This intuition is confirmed by the simulator statistics. In fact, the only workload with *ammp* where the bus beat the other interconnects is workload 20. In all other workloads, the performance with the bus is lower than with the other interconnects. The reason why the harmonic mean over these values is better for the bus is that it emphasises the lowest number. When measuring performance, this is an advantage as it makes the harmonic mean somewhat of a worst case measure. If the arithmetic mean was used, the bus performance would be worse than the other interconnects and this important workload might not have been discovered.

Since the bus is not actually performing any better than the other interconnects, a different property of the CMP must be responsible for slowing down execution for all configurations. The simulator statistics show that *ammp* causes the L1 cache to be blocked for a considerable period. Furthermore, the off-chip memory bus is congested. It is unlikely that congestion alone makes the bus perform better than the ideal interconnect. To achieve this, the time in the memory bus queue in the ideal case must be larger than the combined time in queue in the L1 to L2 bus and the off-chip memory bus with the bus interconnect. Although this effect would probably not lead to the bus outperforming the ideal interconnect, it will certainly reduce the performance difference between them. An important consequence of this queueing is that it influences the memory access issue rate and therefore the blocking behaviour of the cache. Consequently, the end result is probably a combination of these two factors. To verify this hypothesis, an additional experiment was carried out in the 8-core CMP where this problem is more pronounced. This experiment will be discussed in section 5.3.2.1.

The performance of *gap* is similar to the 2-core case. Here, the choice of interconnect does not have a large impact on overall performance. Again, the reason is that the L1 cache is blocked for a considerable time period and this has a larger performance impact. Furthermore, the crossbar interconnect makes the L1 cache block a few less times than the ideal interconnect. Consequently, it performs slightly better.

5.2.2 Other Results

Vortex1 and *parser* experience a considerable slowdown with the bus interconnect. This was not expected as they have relatively few cache misses. However, looking into the results reveals an interesting situation. *Vortex1* is mainly slowed down in workload 9 and workload 29, while *parser* performs worst in workload 17. Furthermore, the miss intensive *apsi* benchmark is a member of all these workloads. The result is bus congestion which delays the few misses *vortex1* and *parser* have enough to slow them down. In other words, *apsi* creates problems for the applications it is run together with.

As in the 2-core case, the strange results where the bus or the crossbar performs better than the ideal is due to either cache blocking or wrong-path execution. In *swim*, *wupwise* and *applu*, the cache lock-up time is the reason for the bus or crossbar outperforming the ideal interconnect. In *eon* and *facerec*, the bus outperforms ideal because it has fewer instructions that are rolled back. Both these effects were discussed in section 5.1.2.

5.3 8-core CMP Configuration

This section explores the experimental results with the 8-core CMP:

- First, the experimental results with the baseline CMP architecture are presented in section 5.3.1. Here, the L2 cache becomes a large performance bottleneck. Consequently, the performance of the interconnect becomes secondary and only has a small impact on overall performance.
- Section 5.3.2 applies this observation by increasing the L2 cache size to 8 MB and the number of MSHRs to 16. Here, the performance impact of the interconnect is much larger.

5.3.1 Baseline 8-core CMP Results

Figure 5.4 shows the results from the 8-core baseline architecture. The general trend is that the performance degradation with a bus interconnect is very small. For instance, *apsi* has a degradation of only just over 5%. This is very low compared to the degradation of almost 35% seen in the 4-core CMP.

A detailed look at the simulation statistics reveals that the benchmarks that use the bus most are slowed down by an increased number of L2 misses. The reason is that two cores now compete

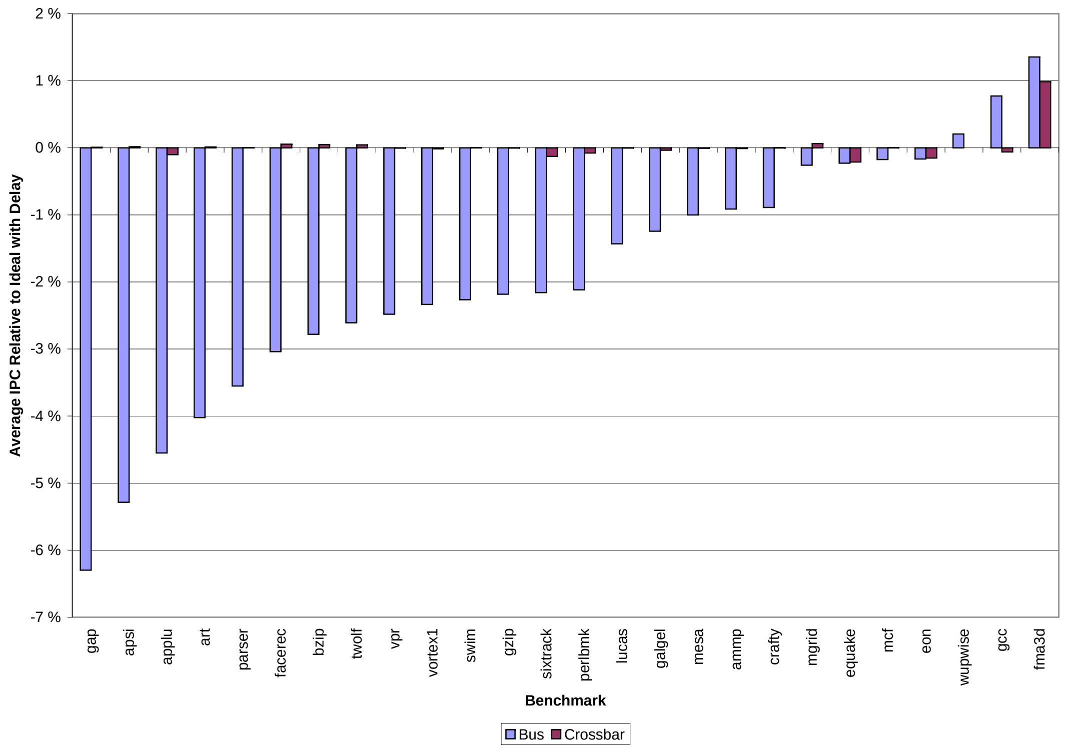


Figure 5.4: 8-core CMP Interconnect Performance

for space in the same cache bank because of the way address translation is done in the M5 simulator. This was described in detail in section 4.1.2.

This hypothesis is validated by running the experiments again with a larger cache size and comparing the results. Workload 4 illustrates the problem. Here, *apsi* and *mgrid* compete for space in the same cache bank. The larger L2 cache size reduces the miss rate from 28% to 20% for the ideal and crossbar interconnects. For the bus interconnect, the miss rate is reduced from 28% to 25%. The reduced L2 cache performance is probably due to a lower access rate in the bus case. A possible explanation is that the memory access rate is slowed down enough that frequently accessed blocks get thrown out of the cache. Here, the accesses that would keep them in the cache with the fast interconnects arrive too rarely to keep the block in the cache with the bus interconnect.

Another strange result is that *gap* suddenly has become sensitive to interconnect performance. Again, the reason is a hot L2 bank which it shares with *ammp*. When the L2 cache size is increased, the miss rate stays the same. However, the number of hits is increased. This makes *gap* run fast enough to make L1 cache blocking the main bottleneck.

The results shown in figure 5.4 will not be discussed further in this report. The reason is that they primarily illustrate L2 cache performance which is not the focus of this work. Instead, the next section discusses the results gathered from the 8 MB 8-core configuration in detail.

5.3.2 8-core CMP with Larger Cache

Figure 5.5 shows the results from the 8-core CMP with 8 MB L2 cache and 16 MSHRs per L2 bank. Here, the overall performance with the bus interconnect is considerably worse than

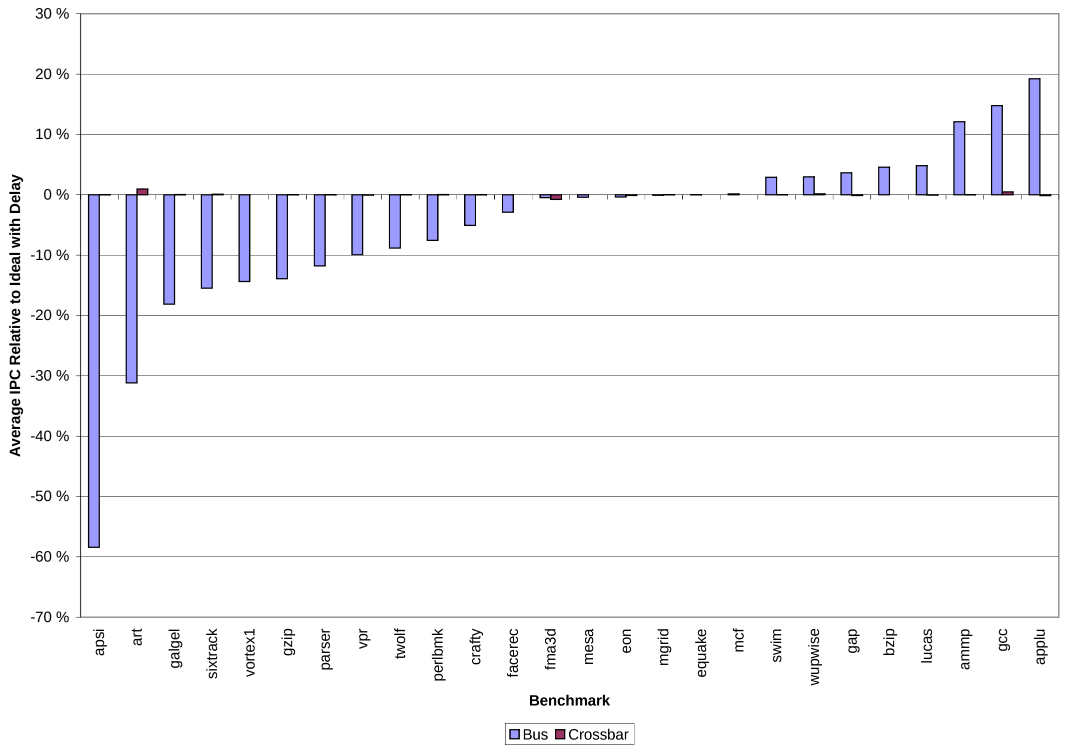


Figure 5.5: 8-core CMP with Large Cache

with the other interconnects as expected. The crossbar scales well in terms of performance and performs close to the ideal interconnect for all benchmarks.

5.3.2.1 Miss Intensive Benchmarks

As usual, *apsi* is sensitive to the performance of the interconnect. For instance, in workload 4 the average queue delay of a request was 23.3 clock cycles with the bus interconnect and 0.1 with the crossbar interconnect.

Both *gap* and *ammp* perform better with a bus interconnect than with the ideal interconnect. In *gap*, this happens in 10 out of the 14 workloads it is a member of. Again, this is due to the amount of time the L1 cache was blocked. With the bus interconnect, the L1 cache is blocked for fewer clock cycles. Therefore, it can service cache hits during this time and the result is higher overall performance. In addition, the off-chip memory bus is congested and this reduces the advantage of having fast L1 to L2 transfers.

The bus interconnect results in better performance than the ideal interconnect for all workloads containing the *ammp* benchmark. This pattern was also seen in one *ammp* workload in the 4-core configuration. The hypothesis is that memory bus congestion and L1 cache blocking combined make the bus outperform the ideal interconnect. To investigate this, workloads 9, 20, 21, 24 and 31 were run with a memory bus that was clocked at the same rate as the processor. Obviously, this is not realistic but it will tell us whether or not the memory bus is the performance bottleneck. The chosen workloads are a subset of the workloads that contain *ammp*, and they were chosen at random.

This experiment resulted in the overall performance of *ammp* with the bus being 22% worse than the performance with the ideal interconnect. Consequently, the strange speed-up was removed. The main reason is that there is less contention for the memory bus. For example, the fraction of the time the memory bus is idle is increased from 0% to 40% for the ideal interconnect in workload 20. In addition, the time the L1 cache is blocked is reduced. These results support the hypothesis that memory bus congestion and L1 cache blocking create the strange results for the *ammp* benchmark.

5.3.2.2 Other Results

The *art* benchmark has a very large performance degradation in the 8 MB 8-core CMP. This is probably caused by a combination of memory bus congestion and the time the L1 cache was blocked. The simulation statistics show that for some workloads the L1 cache blocks more with the bus interconnect than with the ideal interconnect. This is the opposite effect of what we observed earlier. The reason is that it takes a long time to service the misses that have caused the cache to block. In other workloads, the performance of *art* is determined by the sum of delay through the L1 to L2 interconnect and the off-chip interconnect. With a bus interconnect, L1 to L2 delay is large while the memory bus not as badly congested as with the other interconnects. With the crossbar and ideal interconnects, there is virtually no extra delay in the L1 to L2 interconnect but the memory bus is severely congested.

Mcf, *swim*, *wupwise*, *bzip*, *lucas* and *applu* all perform better with the bus interconnect that with the ideal interconnect. The reason is that the L1 cache is blocked for a smaller amount of time with the bus interconnect. This effect was discussed in section 5.1.2.

Gcc also perform better with a bus interconnect than with the other interconnects. However, pinning down the exact cause of this has proved difficult. It is not L1 cache blocking as it is blocked a much larger fraction of the time with the bus interconnect than with the ideal. Wrong path execution is also ruled out as the number of rolled back instructions are greater with the bus interconnect. By examining the workloads where the bus beats the other interconnects, a number of possible reasons are found. However, the most likely reason differs from workload to workload. Most likely, the good performance of the bus is due to complex interactions between the L1 miss rate, L1 lock-up time, L1 to L2 interconnect delay, L2 miss rate and memory bus congestion. Looking further into this problem has not been prioritised.

Chapter 6

CMP Performance with Communicating Workloads

The main aim of this work is to investigate the performance of parallel programs on a CMP platform. In this chapter, the performance of the bus and crossbar interconnects is investigated.

As noted in section 4.1.2, there are some problems with the thread implementation in the M5 simulator's system call emulation mode. Since these flaws might influence the results, we need to record how often they happen. This is done by writing some text describing the occurrence of a problem to a trace file when the simulation is run. Table 6.1 shows the benchmarks where the known flaws were observed. This information is taken into account when the results are analysed. Note that there are a number of cases where no problems are observed for all core counts.

6.1 Parallel Benchmark Performance with 2 CPUs

This section discusses the results from the experiments with the SPLASH-2 benchmarks on a 2-core CMP. Since the main focus of this work is on the interconnect, it is helpful to quantify the interconnect bandwidth demands of the different benchmarks. Then, the interconnect performance results are discussed.

6.1.1 Bandwith Demand with 2 Cores

The number of interconnect requests per committed instruction for all benchmarks and configurations are shown in figure 6.1. In this figure, the benchmarks are sorted in ascending order after the number of requests for the bus interconnect. If the interconnects have a large difference in request rate for the same benchmark, it must be taken into account when the experiments are analysed. The reason is that a difference in network load can make some of the interconnects seem better or worse than they really are.

There are three different causes for the differences in network load shown in figure 6.1:

- The distribution of data to the different L1 caches is controlled by the cache coherence protocol. Consequently, the cache that first requests a cache block becomes the owner of that block. If this block is only read, the cache will continue to be the owner until it writes

Configuration	**Benchmark**	**Bus**	**Crossbar**	**Ideal**
2 CPUs	LUContig	X	X	X
	OceanContig	X	X	-
	OceanNoncontig	X	X	X
	WaterSpatial	-	X	X
4 CPUs	LUContig	X	X	X
	OceanContig	X	X	X
	OceanNoncontig	X	X	X
	WaterNSquared	X	X	X
	WaterSpatial	-	X	X
8 CPUs	Cholesky	X	-	-
	LUContig	X	X	X
	LUNoncontig	X	X	X
	OceanContig	X	X	X
	OceanNoncontig	X	X	X
	Radix	X	X	X
	Raytrace	X	-	-
	WaterNSquared	X	X	X
	WaterSpatial	X	X	X

Table 6.1: Splash Benchmarks where M5 Flaws were Observed

back the block. For instance, say that cache A reads block X one time and that cache B reads block X five times. If the access from cache A arrives at the directory first, cache B will have to issue five reads to A's cache. If B reads block X first, cache A issues one access to its cache. Therefore, it is best to put the block in B's cache in this case. However, in the simulated protocol the block simply ends up in the cache that requested it first.

- Simulation is finished when one processor reaches a certain number of committed instructions. However, the number of instructions the other processor manages to commit varies with interconnect performance.
- The M5 flaws can manifest themselves in different ways for the different configurations. In this case, it is difficult to know how to interpret the results, and they should be discarded.

The differences in request rate for the *Barnes*, *FMM* and *LUContig* benchmarks are all because of different data distributions. In particular, the number of reads from the other processor's L1 cache is different. The M5 flaws manifest themselves a few times for the *LUContig* benchmark, but in a similar way for all configurations. A possible way to reduce the effects of the initial data distribution is to migrate cache blocks such that they are close to the processor that uses them the most. However, a more through investigation is needed to fully understand how this affects protocol and benchmark performance. Pursuing this interesting opportunity is left as further work.

The variation in request rate for the *Cholesky* benchmark is probably not due to data distribution. The reason is that the number of reads to the other processor's L1 cache is about the same for all configurations. However, the number of instructions committed on the processor that does not reach the maximum instruction count, is considerably less for the crossbar than for the two other interconnects. None of the known bugs manifest themselves for this benchmark.

The *OceanContig* result is a typical example of results that must be discarded due to M5 thread implementation problems. Here, two unimplemented methods are called a lot when

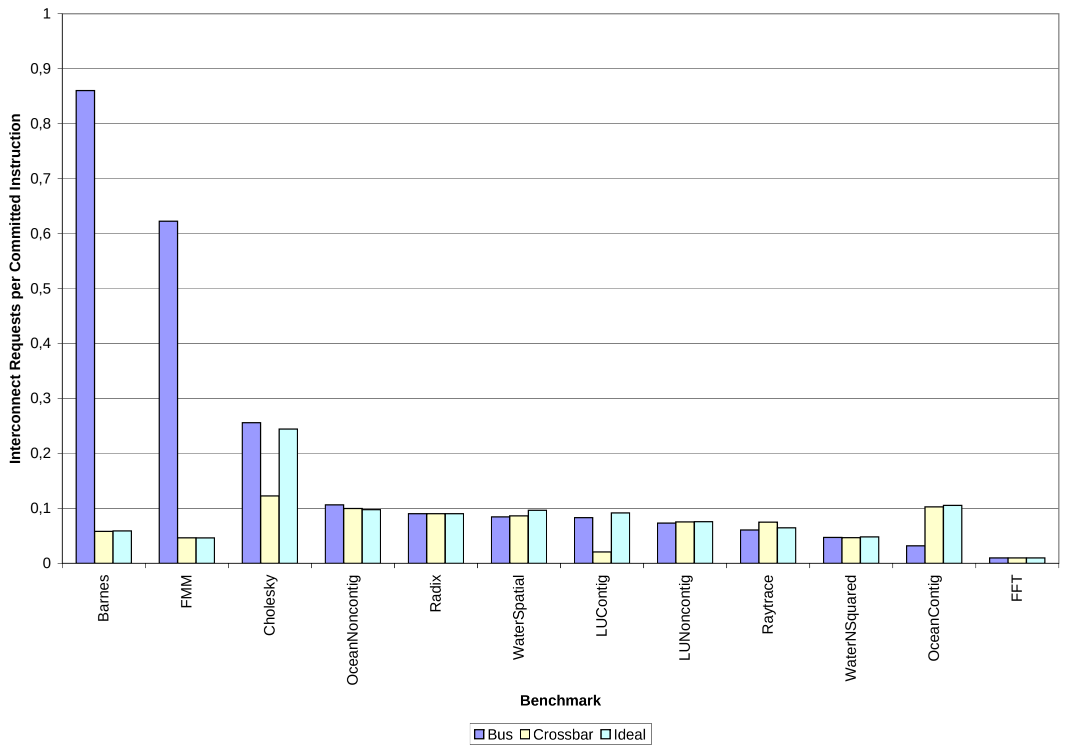


Figure 6.1: Interconnect Requests Rate in Sample for a 2-core CMP

the bus interconnect is used and there are a number of deadlocks when the crossbar is used. Consequently, it is difficult to know how to interpret the results.

6.1.2 2-core Interconnect Performance

Figure 6.2 shows the sum of the *Instructions per Cycle (IPC)* for each benchmark and configuration relative to the performance of configuration with an ideal interconnect. As earlier, the ideal interconnect can service an unlimited number of requests in parallel but all requests experience a constant transmission delay. The sum of IPC is used because it is throughput metric. For a parallel application, it is the total amount work carried out that matter, and the sum of IPC measures this.

The results in figure 6.2 show that the interconnect has a considerable performance impact for the SPLASH-2 benchmarks. In chapter 5, only *apsi* had a performance degradation of more than 20% for the bus compared to the ideal interconnect. Here, 7 out of 12 benchmarks have more than 20% performance degradation for the bus interconnect. Furthermore, the bus interconnect configuration performs over 90% worse than the ideal interconnect for the *FMM* and *Barnes*.

The large performance degradation for the bus interconnect with the *FMM* and *Barnes* benchmarks is due to bus congestion. However, the crossbar and ideal interconnects have considerably less requests to deal with. The reason is that they have a more favourable data distribution and therefore less coherence traffic. Consequently, it is difficult to know whether the crossbar could handle the load the bus received. Conversely, the bus might perform better if it only had to handle the load the crossbar got.

For the *Cholesky* benchmark, there is only a small performance difference between the bus and

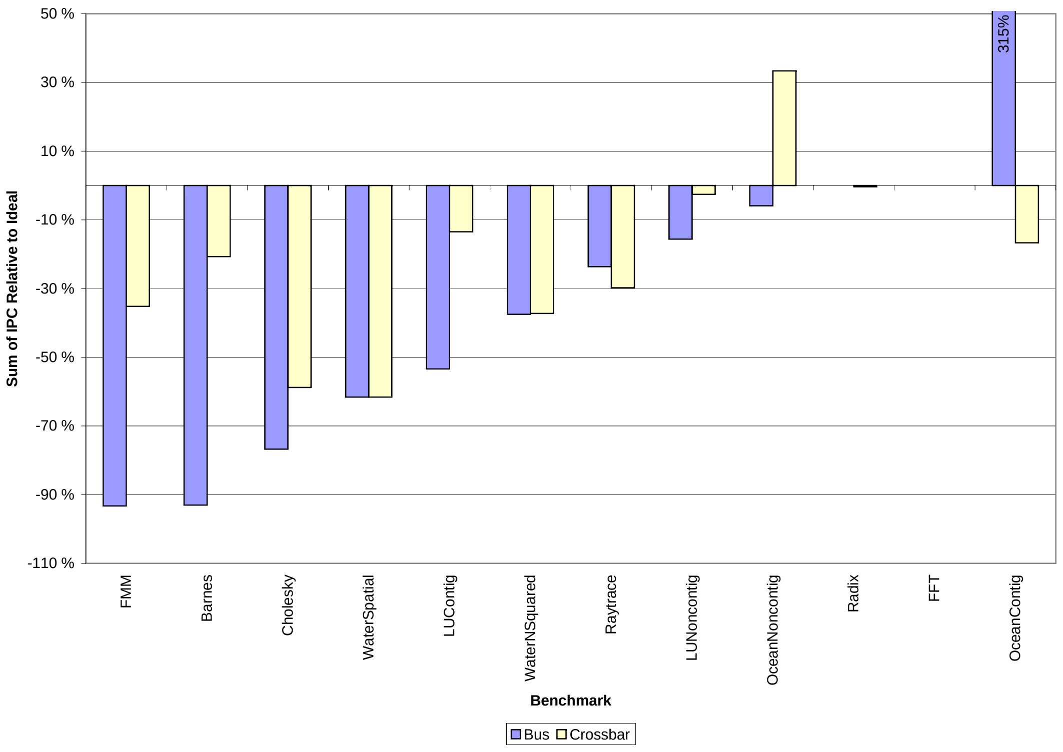


Figure 6.2: Scientific Workload Performance in a 2-core CMP

the crossbar interconnects. The reason is that the bus used for L1 to L1 communication in the crossbar becomes congested. Furthermore, the network load in the crossbar case is considerably less than with the bus and ideal interconnects. The *WaterSpatial* and *WaterNSquared* results also support this theory. Here, the bus and crossbar configurations have very similar performance. In this case, the request rates are similar and only very few M5 flaws were observed with the *WaterSpatial* benchmark.

The *LUContig* benchmark performance with the crossbar is nearly as good as its performance with the ideal interconnect. However, the crossbar has far fewer requests to deal with than the bus and ideal interconnects in this case. Consequently, it is difficult to say whether the performance difference is real or created by the more favourable data distribution in the crossbar case. The M5 flaws manifest themselves in a similar way for the different interconnects. The opposite situation happens with the *Raytrace* benchmark. Here, the crossbar interconnect gets the largest load and therefore performs worst. It is unlikely that the bus could handle a similar load.

Radix and *FFT* do not create enough interconnect requests for the bus or crossbar to limit performance. Consequently, there is nearly no performance difference. The *OceanNoncontig* and *OceanContig* experience a large number M5 deadlocks. Consequently, the results can not be used and they are only shown for completeness.

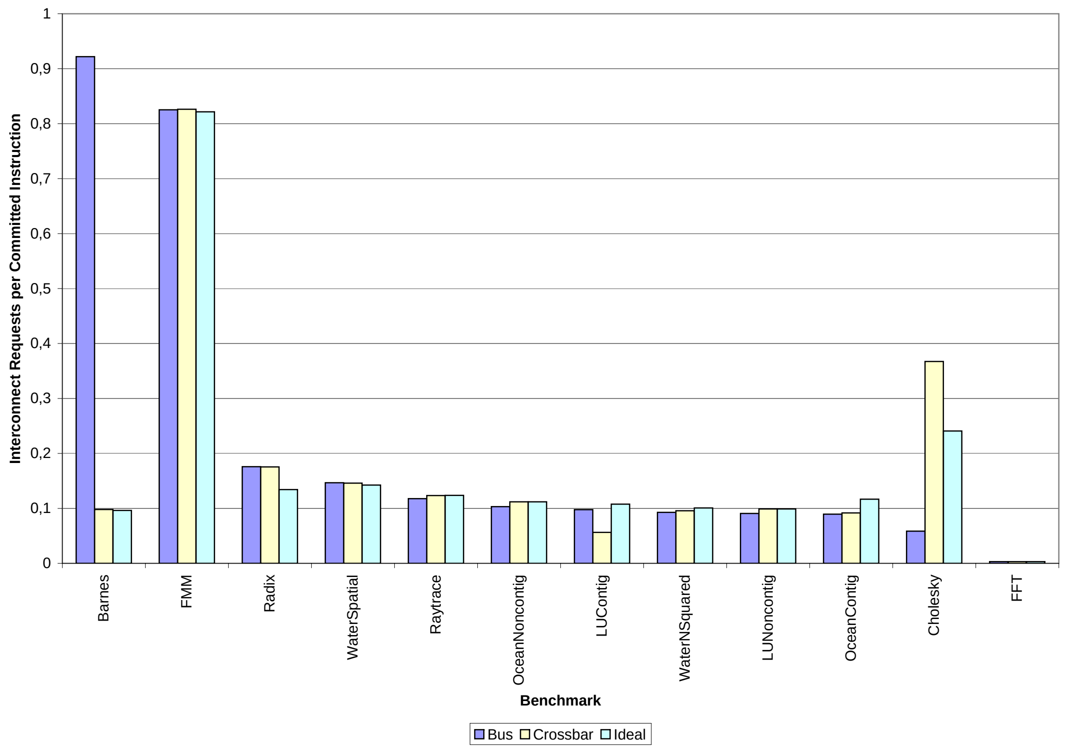


Figure 6.3: Interconnect Request Rate in Sample for a 4-core CMP

6.2 Parallel Benchmark Performance with 4 CPUs

This section discusses the performance of the SPLASH-2 benchmarks on a 4-core CMP. First, the bandwidth demands of the different benchmarks are quantified. Then, the performance results are presented and discussed.

6.2.1 Bandwith Demand with 4 Cores

Figure 6.3 shows the number of interconnect requests per committed instruction for the 4-core CMP. Here, most of the benchmarks have similar request rates with all interconnects. However, *Barnes*, *Radix*, *LUContig* and *Cholesky* have considerable differences and need to be discussed further.

In the 2-core CMP, a similar request pattern was observed with the *Barnes* benchmark. There, the difference was due to a different data distribution, and this is also the case here. With the bus interconnect, two processors initiate a large number of reads to other L1 caches while all processors initiate some reads to other L1 caches with the crossbar interconnect. The total number of remote reads is larger with the bus interconnect and this creates the difference in request rate.

The *Cholesky* benchmark also behaves similarly in the 4-core and 2-core CMP experiments. Again, a larger number of instructions are committed in total with the bus and ideal interconnects. By inspecting the benchmark source code, it becomes clear that there is relatively little global synchronisation. Basically, the processors have relatively large sections of computation

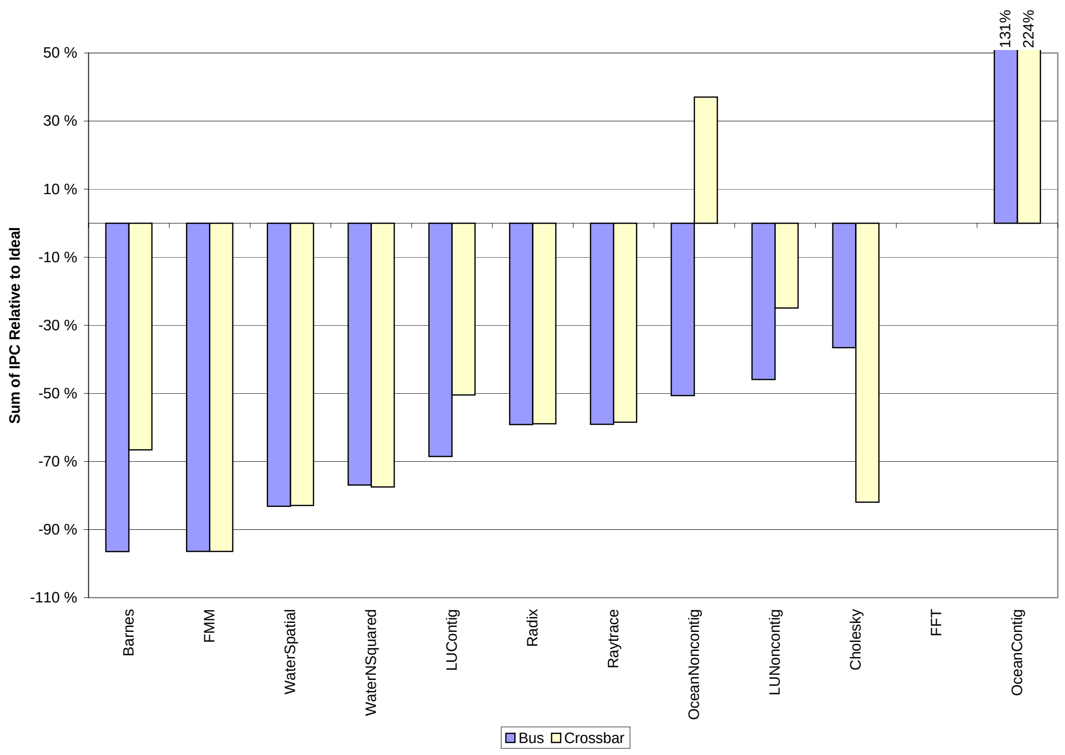


Figure 6.4: Scientific Workload Performance in a 4-core CMP

divided by barriers. This can result in some processors entering a phase with more interconnect usage for some interconnects and might explain the difference observed. However, further investigation is needed to pin down the exact cause of this situation.

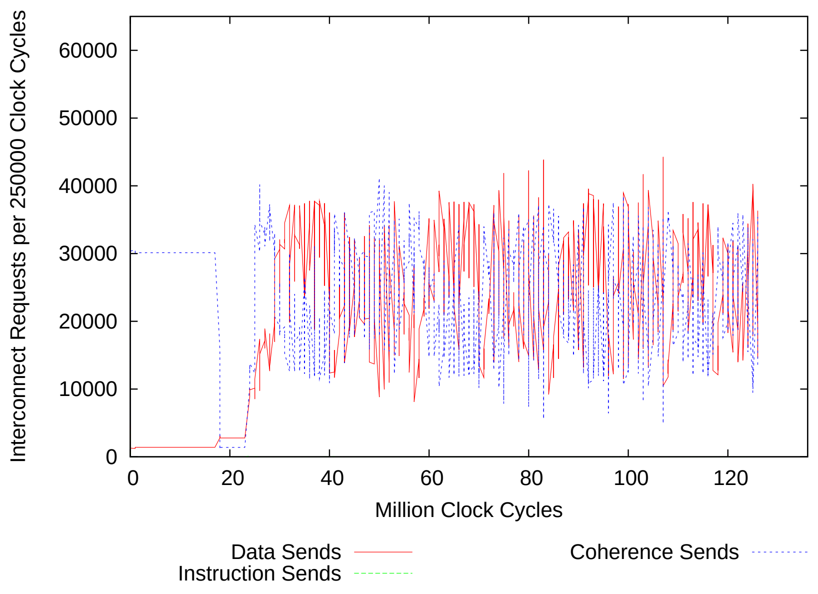
The *Radix* benchmark has one processor that commits very few instructions with the bus and crossbar interconnects. Consequently, the number of requests per committed instruction is larger than with the ideal interconnect. It is unclear whether this difference is the correct behaviour or due to an unknown problem with the M5 system call emulation thread implementation. The *FFT* benchmark is only simulated in the serial section. This is the farthest it is possible to simulate without the simulation taking more than 16 hours on the Norgrid cluster. Sadly, the simulation finishes quickly when the maximum number of instructions is set appropriately. This suggests that it is another problem with the M5 thread implementation. Rather than investigating these problems, full system simulation should be used. There, the faulty thread library is not used and the problems are hopefully avoided.

6.2.2 4-core Performance Results

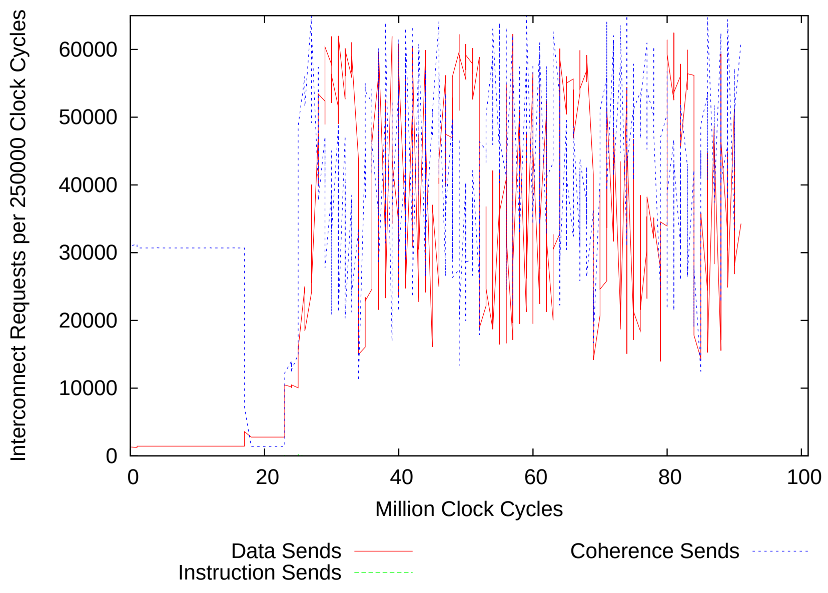
Figure 6.4 shows the performance results for the SPLASH-2 benchmarks on the 4-core CMP. Again, the interconnect has a considerable performance impact. For instance, *FFM* has a performance degradation of over 90% with both the bus and the crossbar interconnects. As expected, the general performance degradation is larger with four cores than with two cores.

According to the results, the benchmarks can be grouped into four classes:

- For the *LUNoncontig* benchmark, the crossbar interconnect outperforms the bus.



(a) Bus Interconnect



(b) Crossbar Interconnect

Figure 6.5: *LUNoncontig* Request Profile

- There are a number of benchmarks where the bus and the crossbar have nearly identical performance. *FMM*, *WaterSpatial*, *WaterNSquared*, *Radix* and *Raytrace* are in this category.
- *Barnes* and *Cholesky* have a large variation in the request rate between the different interconnects.
- Lastly, a few benchmarks must be discarded due to problems with the M5 system call emulation thread implementation. *OceanNoncontig*, *OceanContig* and *FFT* fall into this category.

Figure 6.5 illustrates why the crossbar outperforms the bus with the *LUNoncontig* benchmark.

Here, the lines represent the number of requests to the interconnect over time. The blue lines represent coherence requests and the red lines represent requests for data from the L2 cache. In the bus case, L1 to L1 transfers and data transfers from the L2 cache share the same transmission channel. This can be seen by noticing that when there are many coherence requests, there are few data requests and vice-versa. In the crossbar case, these requests can be handled in parallel. Consequently, the variations of the graph follow the variation in bandwidth demand. An additional point is that there are very few instruction requests as shown by the green line being practically invisible.

For *FFM*, *WaterNSquared*, *WaterSpatial*, *Raytrace* and *Radix* the performance of the bus and the crossbar is nearly the same. The reason is that the interconnect requests are predominately L1 to L1 transfers. Consequently, most requests to the crossbar use its L1 to L1 bus. This bus is very similar to the bus interconnect, so it is no surprise that they have similar performance. The small differences in performance are due to the small differences in network load shown in figure 6.3. As noted earlier, it is unclear if these results are representative for the *Raytrace* benchmark. In addition, a few M5 bugs manifest themselves with the *WaterNSquared* and *WaterSpatial* benchmarks but in a similar way for all interconnects.

Barnes, *Cholesky* and *LUContig* have large differences in interconnect bandwidth demand between different configurations. Consequently, it is difficult to know how the interconnects perform relative to each other. In the *Barnes* and *LUContig* cases, the crossbar outperforms the bus. However, the bandwidth demand is also higher with the bus than with the crossbar. Conversely, the bus performs better than the crossbar with the *Cholesky* application. Here, the crossbar has a significantly larger amount of requests to deal with. In summary, the performance difference is probably due to variations in bandwidth demand and does not provide evidence either way on which interconnect is better.

The last group of benchmarks is the one where the results must be discarded. With *OceanContig* and *OceanNoncontig*, there are large performance differences and unlikely results. These are due to the known M5 system call emulation thread implementation flaws. The *FFT* benchmark is simulated for a very short period of time to avoid what is probably a different M5 problem. Analysing these benchmarks with the better thread implementation in the full system simulation mode of M5, is left as further work.

6.3 Parallel Benchmark Performance with 8 CPUs

In this section, the simulation results for the 8-core CMP is presented and discussed. As usual, it starts with a discussion of the bandwidth needs of the different applications before the actual performance measurements are discussed.

6.3.1 Bandwith Demand with 8 Cores

Figure 6.6 shows the number of requests per committed instruction. Again, most benchmarks have similar request rates across the different configurations. However, *Barnes* and *Cholesky* stand out. Here, one processor has a lot of reads from a different processor's L1 cache for both the bus and crossbar interconnects. Consequently, this processor is slowed down and it commits fewer instructions. Since the total number of instructions committed is reduced, the number of requests per committed instructions is also less in these cases. Note that this might suggest

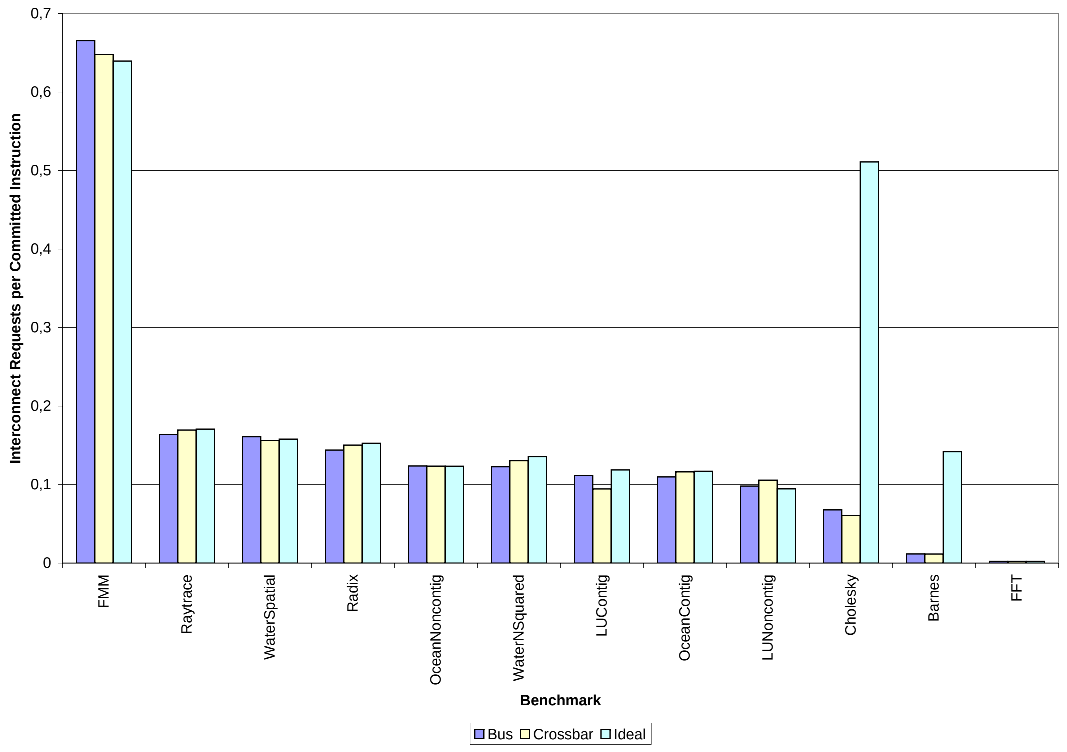


Figure 6.6: Interconnect Requests Rate in Sample for a 8-core CMP

that the impact of data distribution is reduced when the number of processors are increased for these interconnects. Investigating this theory is interesting further work.

6.3.2 8-core Performance Results

Figure 6.7 shows the performance results for the 8-core CMP. Again, the choice of interconnect has a considerable impact on overall system performance. Furthermore, the performance of the bus and crossbar interconnects are more similar to each other than with the 4-core CMP. It is somewhat surprising that the performance degradation compared to the ideal interconnect is not larger. When comparing figure 6.7 to the 4-core results in figure 6.4, the performance degradations experienced are reasonably similar. This might be due to the problem sizes being kept constant when the number for processors is increased. Consequently, the total amount of data communication is the same in both cases. Since the interconnects are already operating at their maximum capacity in the 4-core case, this communication can not be carried out any faster with eight processors. However, more work is needed to verify this theory.

The division of the benchmarks into categories presented in section 6.2.2 for 4-core CMP, holds for the 8-core CMP as well. Again, *FFM*, *WaterNSquared*, *WaterSpatial* and *Radix* all have nearly identical performance for the bus and crossbar interconnects. However, the M5 thread library problems manifest themselves more often with 8 processor cores. Consequently, these results are probably not exact. Taking these limitations into account, the results still support the theory that the performance of the crossbar approaches the performance of a bus when there are a lot of L1 to L1 transfers.

LUNoncontig is again the only benchmark where the crossbar outperforms the bus. In addition,

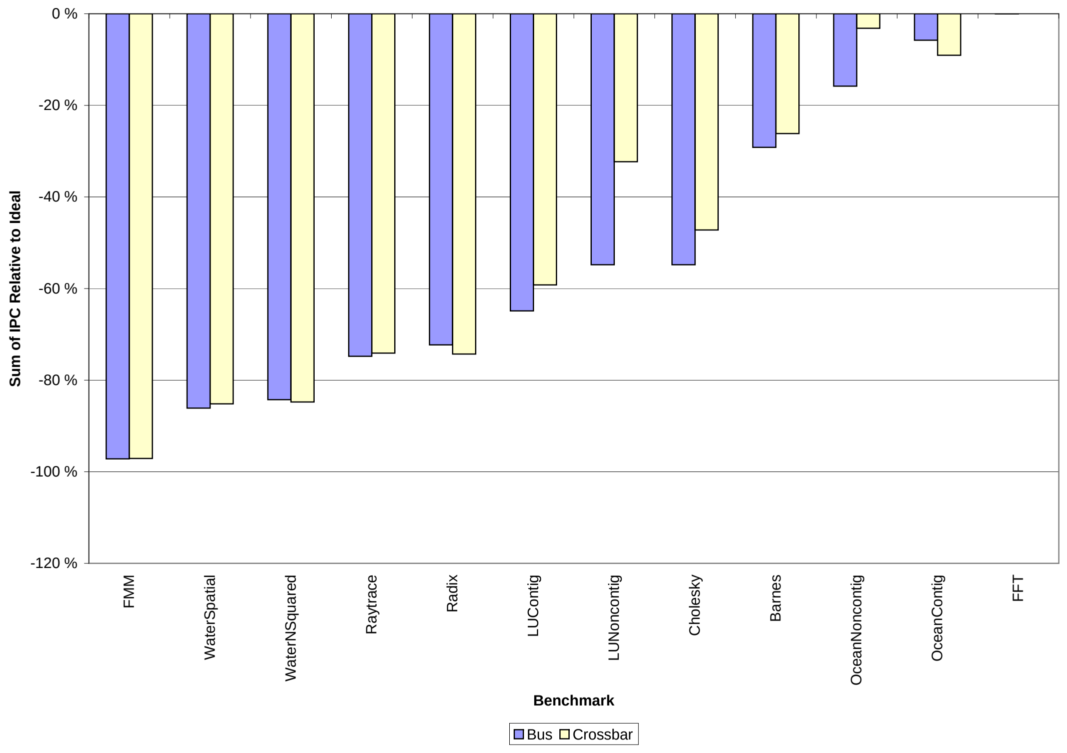


Figure 6.7: Scientific Workload Performance in a 8-core CMP

there is a higher bandwidth demand for the crossbar than the bus. Consequently, the results support the theory that this benchmark benefits from an interconnect where L1 to L2 and L1 to L1 transfers can be handled in parallel. A possible source of error is the M5 thread library flaws, but these manifest themselves in a similar way for all interconnects for this benchmark.

As in the 4-core case, the *LUContig*, *Barnes* and *Cholesky* benchmarks have large variations in the bandwidth demand between interconnects. For *LUContig*, there is only a small performance difference between the bus and crossbar interconnects, and this is due to the difference in bandwidth demand. *Cholesky* and *Barnes* both have a surprisingly low performance degradation with the bus and crossbar compared to the ideal interconnect. The reason is that the bandwidth demand with the ideal interconnect is much larger than for the other interconnects. Although it can handle an unlimited number of requests in parallel, they all experience a transmission delay. The result is that the performance degradation with the realistic interconnects is underestimated. The small performance difference between the bus and crossbar interconnects are caused by a small difference in bandwidth demand in the *Cholesky* case and the presence of a number of L1 to L2 transfers for *Barnes*.

The results with the *OceanContig*, *OceanNoncontig* and *FFT* benchmarks must be discarded in this case as well. For the *OceanContig* and *OceanNoncontig* benchmarks, there are so many deadlocks that it is likely that a lot of the execution of the benchmark is serial. Consequently, the bandwidth demands are quite limited. Again, *FFT* is only executed in the serial section to avoid what is probably a M5 thread library problem.

Chapter 7

Butterfly Interconnect Evaluation

The aim of this chapter is to apply some of the lessons learnt in the experiments carried out in this work. Because of time constraints, a full exploration of possible performance improvement techniques will not be carried out. Instead, the focus will be on one technique: a multistage butterfly network.

To achieve good performance with communicating workloads in a shared L2 cache CMP, efficient L1 to L1 communication should be supported. A conventional CMP crossbar can not do this as the previous chapter shows. The reason is that this network optimises the single-thread common case of efficient L2 access and has limited L1 to L1 bandwidth. A simple solution to this problem would be to use a full crossbar network. In this case, all network nodes can communicate with all other nodes through point-to-point links. However, the number of channels in a full crossbar is $O(N^2)$ where N is the number of nodes in the network. In other words, the hardware cost is prohibitive when there are many nodes.

Multistage networks support this all-to-all interconnection with $O(N \log N)$ channels. An important multistage interconnect is the butterfly network [DT03], which was discussed in chapter 2. This chapter evaluates the butterfly network in a CMP context.

It is unlikely that the butterfly network will outperform the crossbar for the multiprogrammed workloads. As shown in chapter 5, the crossbar performs close to the ideal interconnect in all cases with multiprogrammed workloads. However, chapter 6 showed that the performance of a crossbar approaches that of a shared bus when there is frequent L1 to L1 communication. Therefore, a performance improvement is expected for this application type.

This chapter is organised as follows:

- First, the performance of the butterfly with multiprogrammed workloads is presented and discussed in section 7.1.
- Then, section 7.2 discusses performance of the butterfly with communicating programs from the SPLASH-2 benchmark suite.

The results from chapters 5 and 6 are shown again in the graphs in this chapter. This makes it easy for the reader to compare the CMP performance with the butterfly to the performance with the other interconnects.

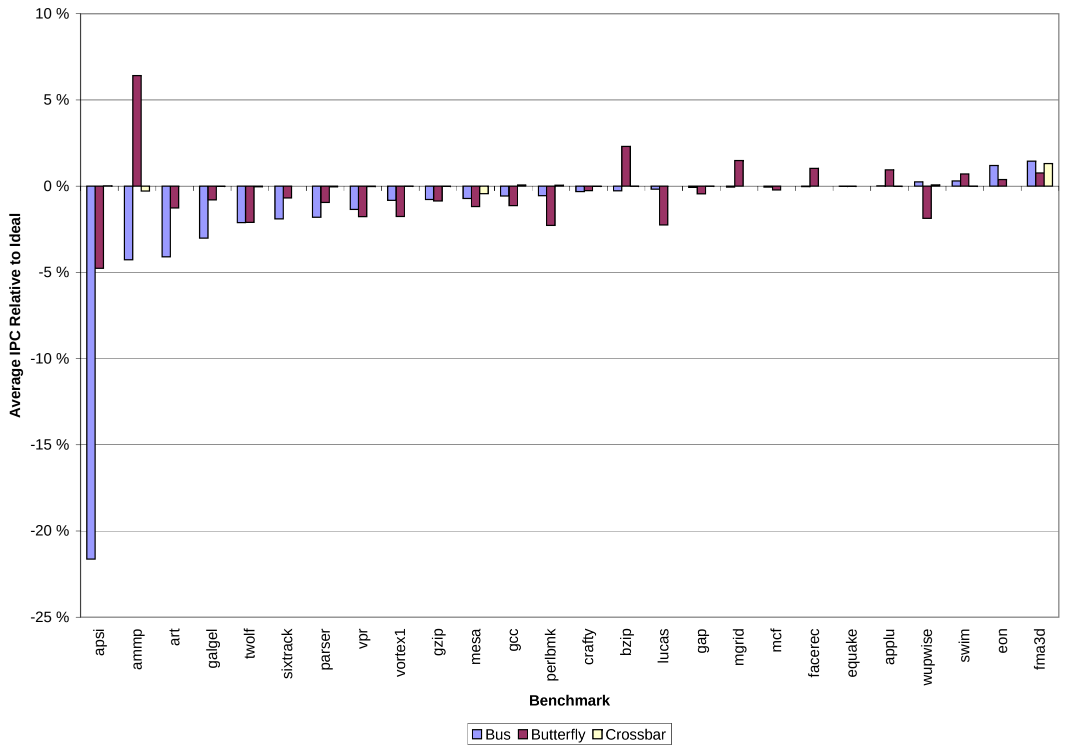


Figure 7.1: Butterfly Performance in a 2-core CMP

7.1 Butterfly Performance with Multiprogrammed Workloads

This section discusses the performance of a CMP with a butterfly interconnect and multiprogrammed workloads from the SPEC 2000 benchmark suite. It has the following outline:

- Section 7.1.1 discusses the results from simulating a 2-core CMP.
- Then, 7.1.2 presents the results from the 4-core CMP experiments.
- Finally, 7.1.3 investigates the butterfly performance in an 8-core CMP.

7.1.1 2-core CMP Multiprogrammed Performance

Figure 7.1 shows the performance of the CMP with bus, crossbar and butterfly interconnects relative to the performance of an ideal interconnect. Recall from chapter 3 that the butterfly has one clock cycle larger minimum transfer latency than the other interconnects for the 2-core CMP. As mentioned in chapter 5, *apsi* is the only benchmark where the interconnect is critical for performance. Here, the butterfly performs much better than the bus but worse than the crossbar as expected. This difference is mostly due to that all transfers take one clock cycle more with the butterfly than the crossbar. For the other benchmarks, the choice of interconnect has only a small performance impact.

The CMP performance with the butterfly interconnect is better than with the ideal interconnect for the *ammp*, *bzip*, *mgrid*, *facerec* and *applu* benchmarks. Obviously, this is not due to the delay of the butterfly being lower. On the contrary, the lower performance of the interconnect causes the caches to perform better. For *ammp*, *bzip*, *mgrid* and *facerec*, there are fewer L2 cache misses with the butterfly than with the ideal interconnect. With *applu*, there are less L1

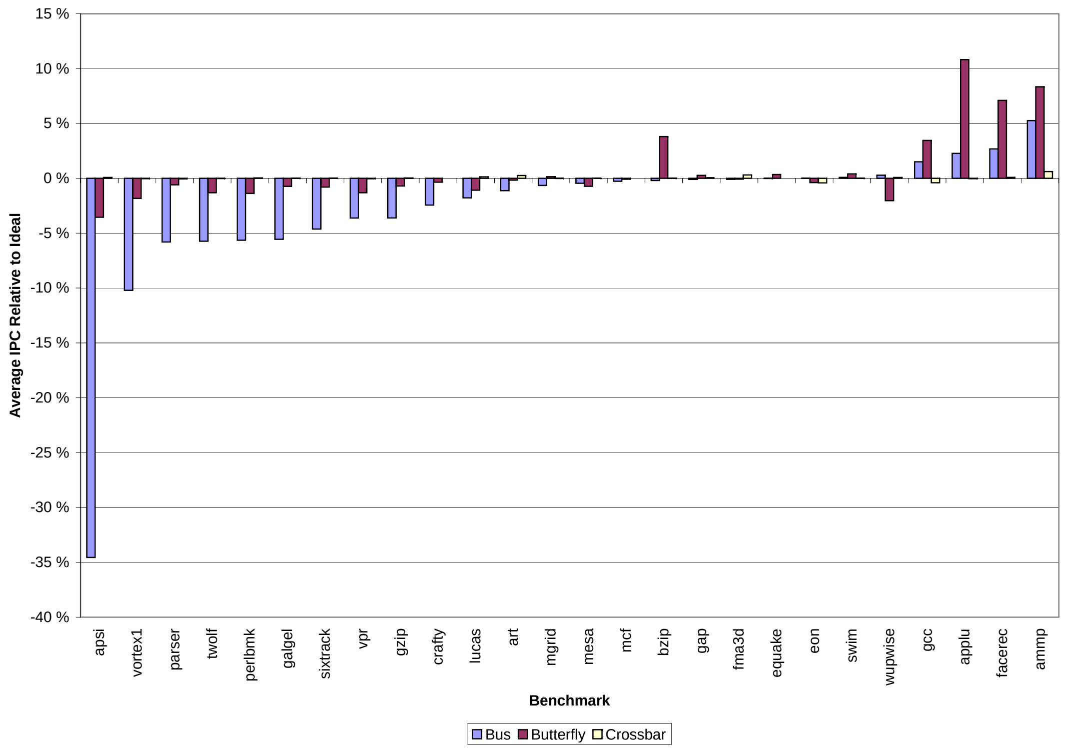


Figure 7.2: Butterfly Performance in a 4-core CMP

instruction cache misses. A likely explanation is that the butterfly delays the requests enough to keep some frequently used data in the caches.

The CMP performance is worse with the butterfly than with all other interconnects for 10 benchmarks. With the *lucas*, *vortex*, *perlbmk*, *wupwise*, *gap* and *mcf* benchmarks, the reason is that the there are not enough requests to make the queue delay of the bus large enough for the total delay to be greater than the minimum butterfly delay. For *vpr*, *gzip*, *mesa* and *gcc* the delay of bus is very similar to the butterfly delay. Consequently, small changes in the timing of requests create the small effects that create the performance difference. Since the differences are small, it is time-consuming to track down the exact causes. It is better to spend this time on effects relevant to the problem at hand. Mainly, these effects highlight that the choice of interconnect only has a small performance impact for a 2-core CMP for most of the SPEC2000 benchmarks.

This point is further backed up by the performance of *swim*, *eon* and *fma3d*. Here, the *CMP* performance with the butterfly is better than with the ideal interconnect. This time, the reason is that the better performance of the ideal interconnect result in the programs continuing further down a wrong execution path.

7.1.2 4-core CMP Multiprogrammed Performance

This section discusses the performance of the butterfly interconnect with multiprogrammed workloads and a 4-core CMP. The results are shown in figure 7.2. As expected, the performance with the butterfly is better than the performance with the bus and worse than the performance with the crossbar. Again, the minimum transfer delay of the butterfly is one clock cycle longer than the minimum latency of the other interconnects.

The performance with the butterfly is better than the performance with the ideal interconnect in a few cases. Again, the reason is that the somewhat slower performance of the butterfly makes some other part of the architecture perform better. With *bzip* and *ammp*, the reason is that the butterfly avoids a few cache misses. The situation is more complex with *gcc*, *applu* and *facerec* because the butterfly configuration performs better than the bus in some workloads and worse in others. Since the harmonic mean emphasises the lowest number, the configurations with the lowest performance have a stronger impact on the result. For these benchmarks, the good performance is due to fewer cache misses, less cache blocking or a combination of these. In some workloads, the memory bus is badly congested and this amplifies the performance effect of these small changes.

With *wupwise*, the butterfly configuration performs worse than all other interconnects. This is due to workload 32 where the L1 cache is blocked for longer time with the butterfly than with the other interconnects. Consequently, the performance of the butterfly configuration is degraded.

7.1.3 8-core CMP Multiprogrammed Performance

This section discusses the performance of the butterfly with multiprogrammed workloads and an 8-core CMP. First, the performance results from the 8-core CMP described in chapter 3 is presented. This configuration provides too little L2 cache space per core, and this becomes the predominant bottleneck. Consequently, the interconnect only has a secondary impact on performance. Therefore, a CMP where the L2 cache size is increased to 8 MB and the number of MSHRs per bank is increased to 16 is simulated. For both configurations, the minimum delay of the butterfly is the same as the minimum delay for the other configurations.

7.1.3.1 Original 8-core CMP Configuration

Figure 7.3 shows the simulation results for the original 8-core configuration. Here, the performance with the butterfly is chaotic. For some benchmarks a large speed-up is observed and for others a large performance degradation. A possible explanation is that the butterfly can give very good performance for a favourable traffic pattern while it can give low performance for an unfavourable pattern. When this is coupled with a strained cache system, the effects can be severe.

Again, this experiment does not give too much insight into interconnect performance as the caches are the predominant bottleneck. Therefore, only the performance of the *applu* and the *gap* benchmarks will be discussed. *Applu* has very good performance with the butterfly network while *gap* experiences a severe performance degradation. In the *applu* case, the performance improvement is due to *applu* being very lucky with how its L1 cache misses are serviced. In workload 8, *applu's* L1 cache nearly does not block at all. Since the available resources have not been increased, this results in much more blocking for the other applications in this workload.

Gap has very low performance with the butterfly in some workloads and very good performance in others. Workload 17 in an example where *gap* experiences a performance degradation. Here, *gap's* L1 cache blocks a lot more than with the other interconnects. Furthermore, there are more L2 cache misses in the L2 bank that *gap* uses.

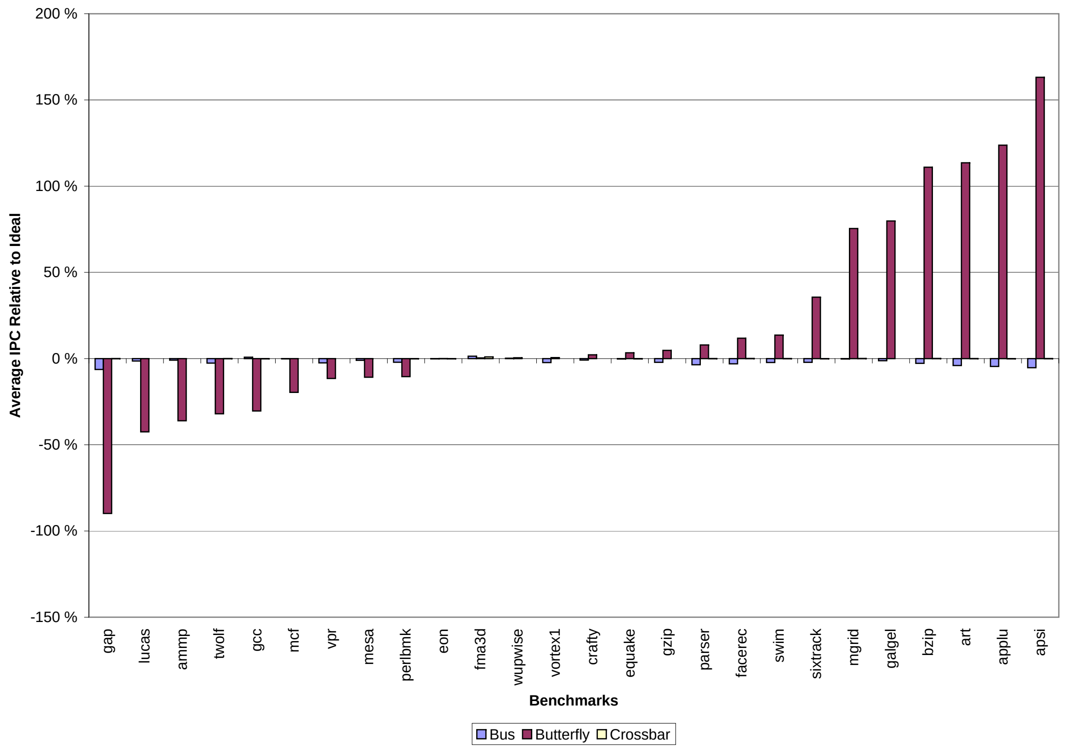


Figure 7.3: Butterfly Performance in a 8-core CMP

These examples indicate an interesting point. When there are not enough resources to serve the needs of all applications, the achieved performance can be quite random with the butterfly interconnect. A possible explanation is that conflicts in the butterfly impacts some benchmarks more than others. Consequently, the benchmarks that have few conflicts will get more of their data into the L2 cache as they are able to access it more often. This leads to other applications data being thrown out and therefore they experience a performance degradation. Ideally, performance should be predictable when the processor is heavily loaded. Therefore, the butterfly seems badly suited to a CMP where this situation is expected to occur relatively often. Investigating techniques that ensure that all cores are given a fair share of the shared resources is interesting further work.

7.1.3.2 8-core CMP Configuration with Large Cache

Figure 7.4 shows the performance results from the experiments with the large cache, 8-core CMP. Here, the performance with the butterfly is very close to the performance with the crossbar and the ideal interconnects. This is actually a bit better than expected, and two factors contribute to these good results. Firstly, the minimum delay through the butterfly is the same as through the other interconnects with the 8-core CMP. Secondly, there are some unused nodes in the butterfly. The reason is that only 12 nodes are needed, but the number of nodes in a butterfly must be a power of two. This point was discussed in section 3.2.3.3, and the reason is that there are 8 cores and 4 L2 banks. Consequently, the butterfly has 16 nodes, and this reduces the probability of conflicts in the butterfly.

The only strange result that can be seen in the graph is that the butterfly configuration performs worse than all other interconnects with *gcc*. This is mainly due more L1 misses with the butterfly

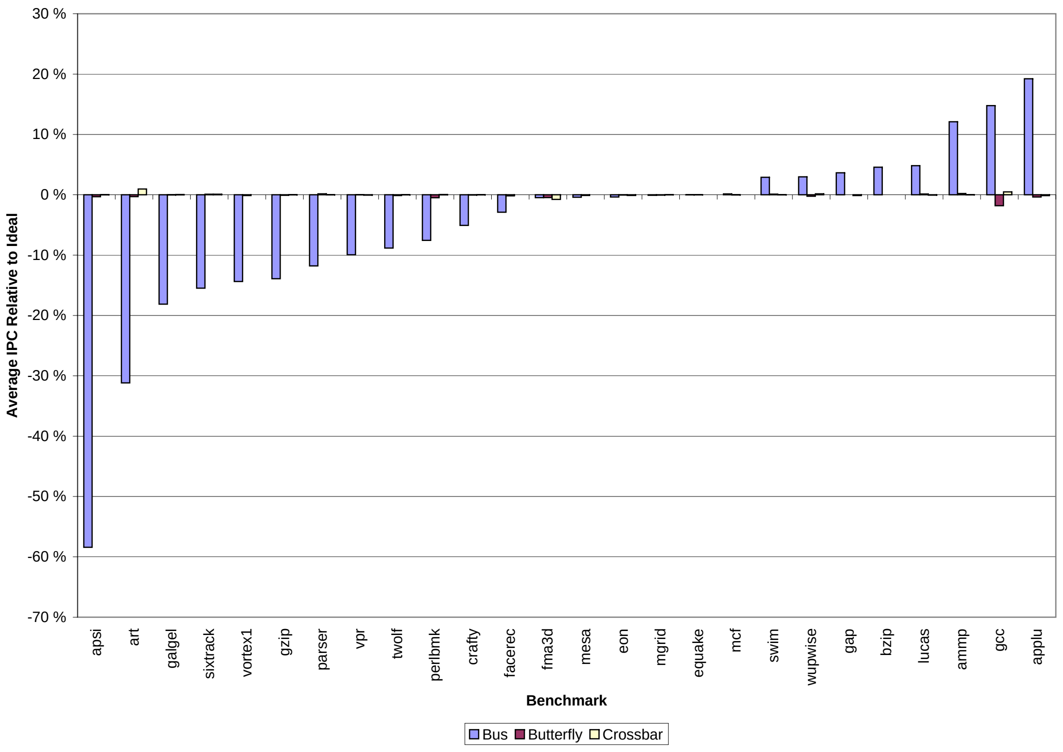


Figure 7.4: Butterfly Performance in a 8-core CMP with Large Cache

than with the other interconnects in a few workloads. Since the harmonic mean emphasises the lowest number, these workloads make the performance degradation show up in the graph. The exact cause of the good bus performance is unclear, but it is probably due to complex interactions between the number of L1 misses, the number of L2 misses and cache lock-up time. This point was discussed more thoroughly in section 5.3.2.2.

With *mcf*, *swim*, *wupwise*, *gap*, *bzip* and *applu* the bus outperforms the other interconnects due to less L1 cache lock-up time. This point was also discussed in section 5.3.2.2.

7.2 Butterfly Performance with Scientific Workloads

This section discusses the CMP performance with the butterfly interconnect and applications from the SPLASH-2 benchmark suite. It discusses the results from experiments with 2-, 4- and 8-core CMPs.

Table 7.1 shows the configurations where the M5 system call emulation thread library flaws were observed. The bus, crossbar and ideal values are repeated for completeness. These flaws were discussed in chapter 4. As in chapter 6, the results from the *OceanContig* and *OceanNoncontig* benchmarks must be discarded. The reasons are that the flaws are observed very often or in a different way for each interconnect with one benchmark. For the other benchmarks, the flaws are observed rarely enough that the results can be used. However, it is likely that some error is introduced. Consequently, important future work is to rerun the experiments in the full system simulation mode of M5 where the thread implementation allegedly is better. Furthermore, the results for *FFT* are discarded as it is only simulated in the serial section. This is due to what

Configuration	Benchmark	Bus	Butterfly	Crossbar	Ideal
2 CPUs	LUContig	X	-	X	X
	OceanContig	X	X	X	-
	OceanNoncontig	X	X	X	X
	WaterNSquared	-	X	-	-
	WaterSpatial	-	-	X	X
4 CPUs	LUContig	X	-	X	X
	OceanContig	X	X	X	X
	OceanNoncontig	X	X	X	X
	WaterNSquared	X	X	X	X
	WaterSpatial	-	-	X	X
8 CPUs	Cholesky	X	-	-	-
	LUContig	X	-	X	X
	LUNoncontig	X	X	X	X
	OceanContig	X	X	X	X
	OceanNoncontig	X	X	X	X
	Radix	X	-	X	X
	Raytrace	X	-	-	-
	WaterNSquared	X	X	X	X
	WaterSpatial	X	X	X	X

Table 7.1: Splash Benchmarks where M5 Flaws were Observed

is probably another M5 thread library bug. Section 6.2.1 contains a more detailed discussion of this problem.

7.2.1 2-core CMP SPLASH-2 Performance

This section discusses the performance of the 2-core CMP with the different interconnects and benchmarks from the SPLASH-2 benchmark suite. Figure 7.5 shows the interconnect request rate and figure 7.6 shows the performance results. The coherence protocol, parallel phase behaviour of the benchmark and the M5 thread library flaws can create variation in the interconnect request rate between different interconnects for the same benchmark. This point was thouroughly discussed in section 6.1.1. Furthermore, the minimum transfer delay of the butterfly is one clock cycle larger than in the other interconnects.

For most benchmarks, the butterfly configuration outperforms the bus and crossbar configurations. Furthermore, it performs close to the ideal interconnect with the *FFM* and *LUContig* benchmarks. For *FFM*, this is due to actual good performance, but for *LUContig* it is probably a bit flattering. The reason is that the butterfly has less requests to handle than the ideal interconnect. With *Cholesky* the variation in requests make the crossbar look better than it really is because it has many fewer requests to handle than the other interconnects.

With the *Barnes* and *LUNoncontig* benchmarks, the butterfly configuration performs better than the bus but worse than the crossbar. In the *Barnes* case, the explanation is simply that the crossbar has significantly fewer requests to deal with. However, the situation is somewhat more complicated with *LUNoncontig*. Here, the average delay each request experiences is actually only 10.2 clock cycles with the butterfly while it is 11.0 clock cycles with the crossbar. The performance difference is due to more cache blocking with the butterfly.

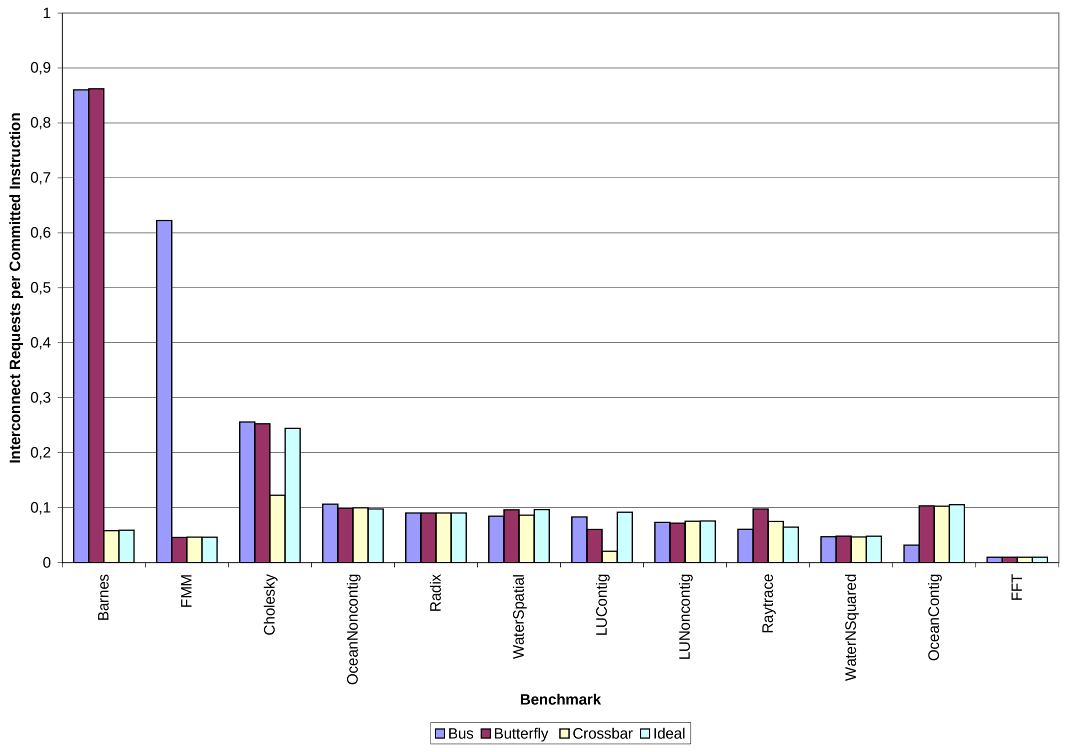


Figure 7.5: Total Interconnect Requests in Sample for a 2-core CMP

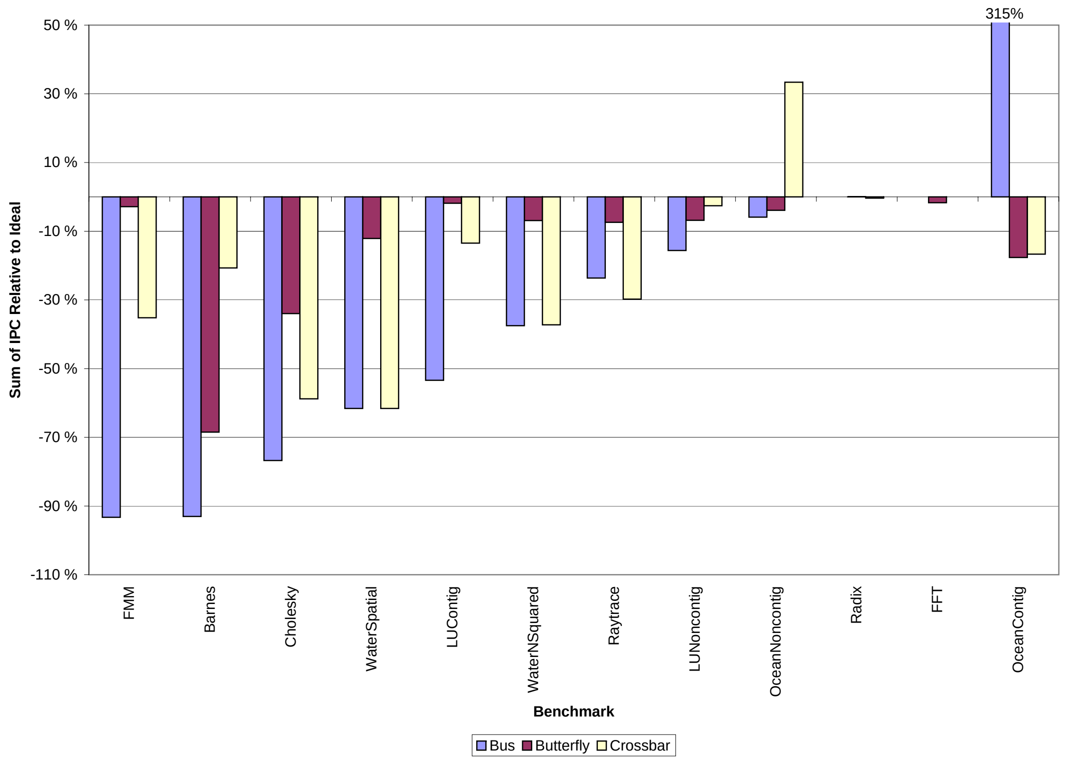


Figure 7.6: Butterfly Communication Performance in a 2-core CMP

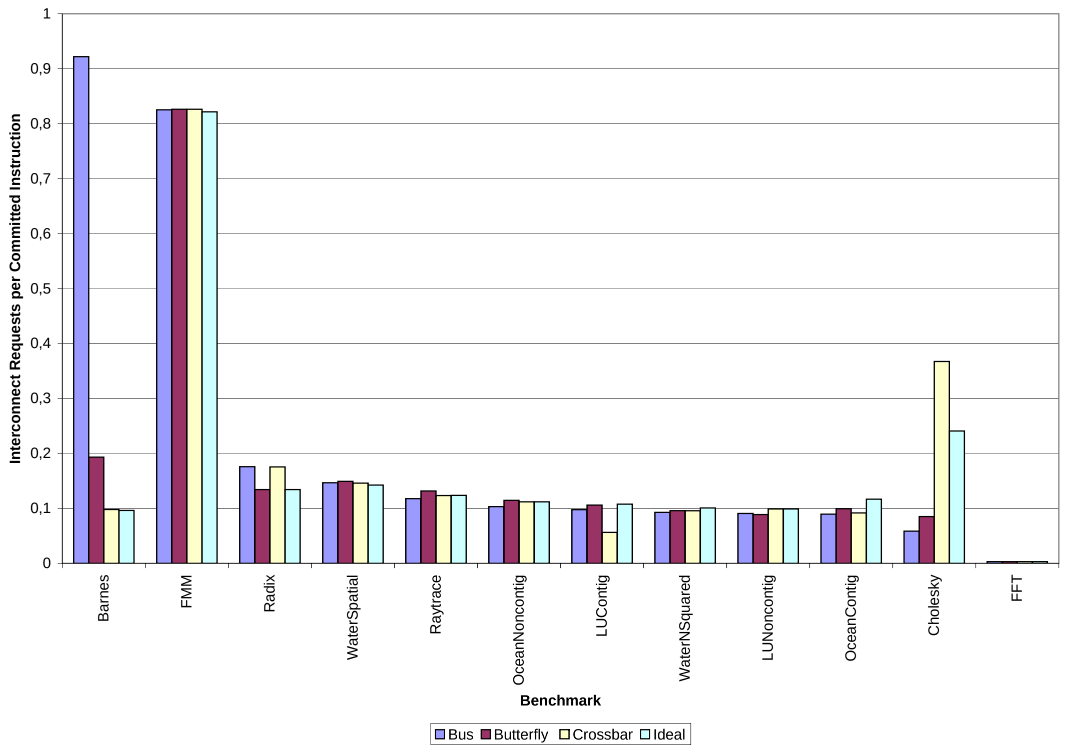


Figure 7.7: Total Interconnect Requests in Sample for a 4-core CMP

FFT and *Radix* both have little communication and all interconnects can handle it. Consequently, there is no performance difference. The results from the experiments with *OceanContig* and *OceanNoncontig* are discarded as these benchmarks have been severely exposed to the M5 thread library flaws. They are shown for completeness.

7.2.2 4-core CMP SPLASH-2 Performance

Figure 7.7 shows the number of requests per committed instruction for all benchmarks and configurations, and figure 7.8 shows the 4-core CMP performance with the different interconnects. Again, the minimum latency through the butterfly is one clock cycle larger than the minimum latency through the other interconnects.

The most interesting result in figure 7.8 is probably the *Radix* performance with the butterfly interconnect. Here, it actually performs as well as the ideal interconnect. This highlights the abundant bandwidth available with a butterfly when the traffic pattern is favourable.

The butterfly configuration outperforms the ideal with the *Barnes* and *Cholesky* benchmarks. For *Cholesky*, a part of the explanation is that the butterfly has much less requests to deal with than the ideal interconnect. Furthermore, the processors that do not reach the maximum instruction count get more work done before simulation is finished with the butterfly. Consequently, the sum of IPC is larger and it seems like the butterfly outperforms the ideal interconnect. For *Barnes*, the butterfly actually has more requests to deal with than the ideal interconnect. However, the lower performance of the butterfly reduces the number of cache misses and cache lock-up time compared to the ideal configuration.

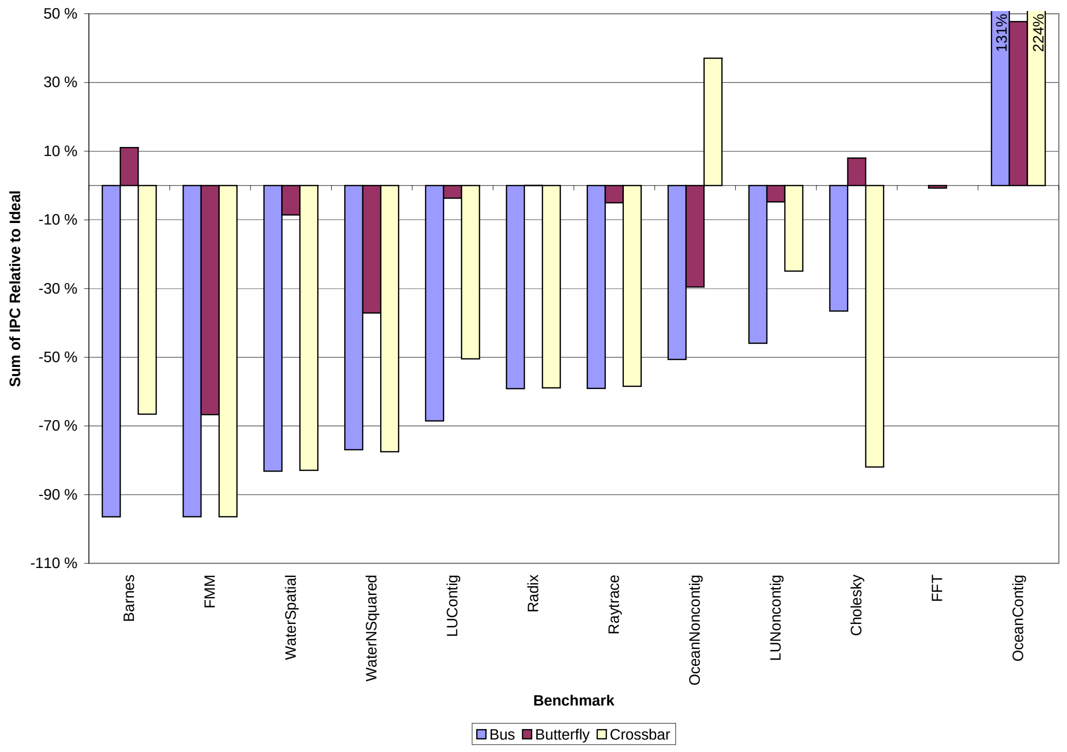


Figure 7.8: Butterfly Communication Performance in a 4-core CMP

Again, the results from the *OceanContig*, *OceanNoncontig* and *FFT* benchmarks are discarded due to problems with M5.

7.2.3 8-core CMP SPLASH-2 Performance

This section discusses the experimental results with the SPLASH-2 benchmarks and the 8-core CMP. Here, the applications are so communication intensive that the interconnect continues to be the predominate bottleneck. Consequently, there was no need to increase the L2 cache size. Figure 7.9 shows the interconnect request rate for this configuration and figure 7.10 shows the performance results.

With *FMM*, *WaterNSquared*, *Raytrace* and *LUContig*, the butterfly configurations outperform the bus and crossbar configurations due to a better performing interconnect. Furthermore, the butterfly configurations are slower than the ideal interconnect configurations for these benchmarks. Consequently, they perform as expected in these cases. However, the butterfly configuration should perform closer to the ideal configuration with the *FMM* benchmark. The reason for this low performance is that an unfavourable traffic pattern creates a hot channel in the butterfly. This result in the average request delay being as large as 126 clock cycles. With the ideal interconnect this delay is 9 clock cycles, and the bus and crossbar has an average request delay of 1819 and 1820 clock cycles, respectively. This illustrates how the parallelism available within the butterfly can be wasted for some traffic patterns.

There are also a number of cases where the butterfly configuration outperforms the ideal configuration. With the *Radix* benchmark, the butterfly slows down the execution of the processor that reaches the maximum instruction count. Consequently, the other processors get more work

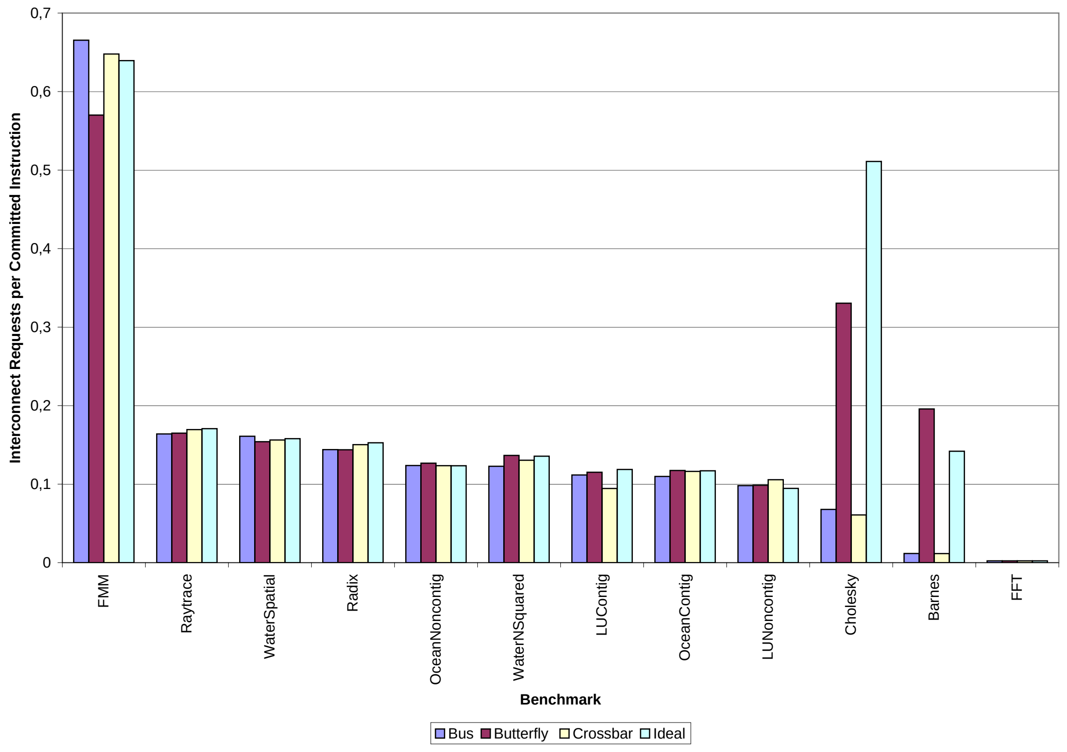


Figure 7.9: Total Interconnect Requests in Sample for a 8-core CMP

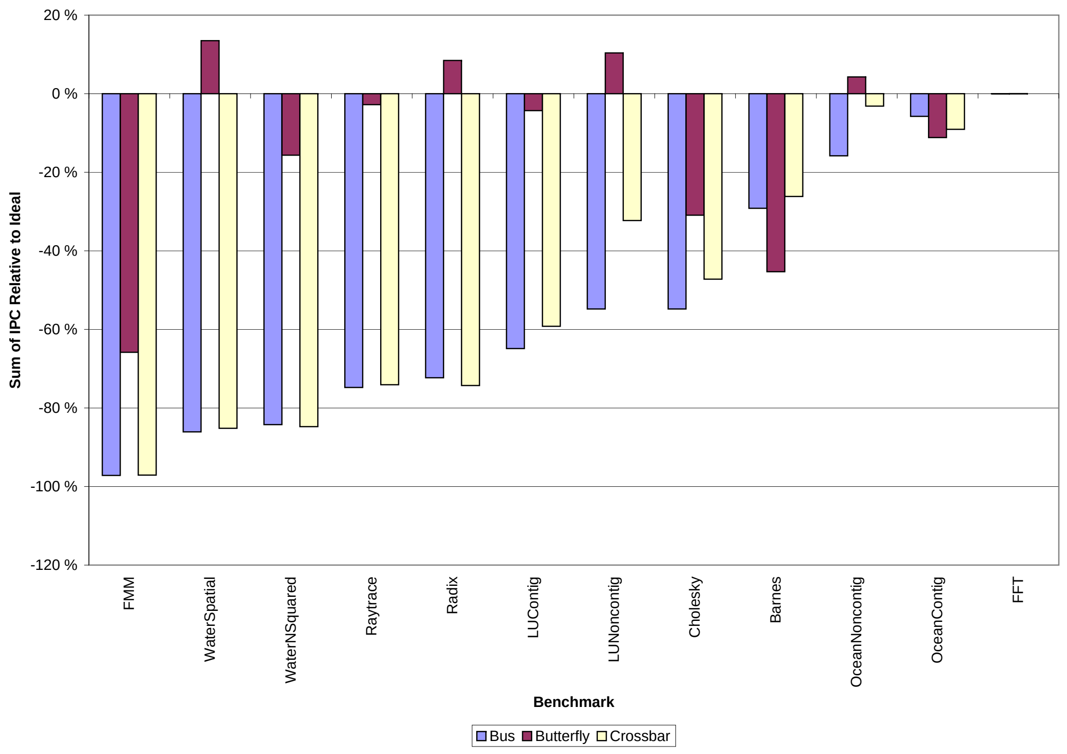


Figure 7.10: Butterfly Communication Performance in a 8-core CMP

done and it seems like the butterfly performs better than the ideal interconnect. *LUNoncontig* has between 1000 and 2000 less L2 misses in each bank when the butterfly interconnect is used. This increased cache performance causes the good butterfly performance in this case. For *WaterSpatial*, the difference is probably due to the M5 problems manifesting themselves in different ways. In particular, there are a few more deadlocks with the butterfly than with the other interconnects. Consequently, the results from this benchmark can probably not be trusted.

With the *Barnes* benchmark, the butterfly performs worse than both the bus and the crossbar. The reason is that it has a lot more requests to deal with. Consequently, it is difficult to compare the different interconnects in this case. The situation with the *Cholesky* benchmark is similar with the butterfly and ideal interconnects having many more requests to deal with. However, the butterfly still outperforms the crossbar and the bus. It is likely that this performance difference would be much larger if the bus and crossbar had to handle the same number of requests as the butterfly.

As usual, the results from the *OceanContig*, *OceanNoncontig* and *FFT* benchmarks are discarded due to problems with M5.

Chapter 8

Discussion and Evaluation

The main purpose of this chapter is to answer the research questions stated in section 3.1. Section 8.1 takes care of this. Then, section 8.2 discusses a few choices made while carrying out this work. With the benefit of hindsight, they do not seem as clever as they did when they were made. Consequently, it is important to document them so that better choices can be made in the future.

8.1 Discussion

8.1.1 Multiprogrammed Workload Performance

How does the CMP on-chip interconnect between private and shared caches influence overall system performance for multiprogrammed workloads?

The answer to this question is highly dependent on the benchmark and interconnect used. A CMP that uses the crossbar interconnect performs very close to the ideal interconnect for all multiprogrammed workloads and CMP configurations investigated in this report. In this case, the performance impact of the interconnect is very small. If the bus interconnect is used, the performance impact can be severe. For instance, the bus configuration performs 58% worse than the ideal interconnect with the *apsi* benchmark on the 8-core, large cache CMP. Consequently, the choice of interconnect is important for overall system performance as a bad choice can severely limit performance.

8.1.2 Parallel Workload Performance

How does the CMP on-chip interconnect between private and shared caches influence overall system performance for scientific workloads?

For the scientific workloads, the impact of the interconnect on overall system performance is large. The *FFM* benchmark experiences a 93% performance degradation with the bus interconnect and a 62% degradation with the crossbar interconnect compared to the ideal interconnect configuration on the 2-core CMP. In other words, the interconnect performance is critical even with only two processing cores. With the 8-core CMP, the performance degradation is 97% for both the bus and crossbar interconnects. The reason for the small difference is probably that the interconnects are operating at their maximum capacity and that the same problem size is used for the 2, 4 and 8 processor experiments. In other words, the total amount of communication is not increased when the number of processors is increased. Investigating the effects of scaling the problem sizes with the number of processors is possible further work.

8.1.3 Performance Impact of Interconnect Enhancements

Can improvements to the private to shared cache interconnect improve performance for both multiprogrammed and scientific workloads?

This question is covered by the evaluation of the butterfly interconnect with both multiprogrammed workloads and scientific applications. For the scientific applications, overall system performance with the butterfly interconnect is much better than with the crossbar and bus interconnects. If we discard the results from the applications where the comparison is not valid due to variation in request rate or problems with the M5 system call emulation thread library[1], the butterfly configuration on average perform 3.9 times better than the bus and 3.8 times better than the crossbar on the 8-core CMP. This impressive speed-up is probably to a larger extent an indication of the low L1 to L1 cache communication performance of the crossbar than the merits of the butterfly interconnect. In other words, the results should be interpreted as a large speed-up being available if sufficient L1 to L1 cache bandwidth is provided rather than the butterfly network being the definitive answer to the problem. Furthermore, the hardware cost of the butterfly networks used in this work is probably higher than the cost of the crossbars used.

With the multiprogrammed workloads, the crossbar is difficult to beat in terms of performance. The reason is that overall system performance with the crossbar is close to the ideal interconnect in all cases simulated in this work. However, the butterfly is not much worse, performance wise. Sadly, conflicts in the butterfly impact benchmarks in a non-uniform way and this leads to some benchmarks in a workload experiencing a speed-up and others a considerable performance degradation. This effect is evident when there is extensive competition for space in the L2 cache as shown in the small L2 cache, 8-core CMP experiments. Consequently, some fairness constraints are needed before the butterfly can be applied in this context.

The good performance of the crossbar and butterfly interconnects is due to overprovisioning of resources. Consequently, the utilisation of many channels in these interconnects can be very low. In other words, we have a hardware structure that uses a lot of area, and much of this hardware is only rarely used. This is hardly efficient. By using the allocated hardware in a more efficient manner, it should be possible to reduce the hardware needs at the cost of a modest performance degradation. This saved area can then be used to provide more cache space and this might make up for the performance loss in the interconnect. Pursuing this further is promising further work.

8.2 Evaluation

This section discusses a few choices made in this work that were not the best possible choices. First, the choice of a few CMP model parameters is discussed. Then, a couple of improvements to the use of the M5 simulator are presented. Finally, some implementation decisions regarding the simulator extensions is discussed.

[1]The discarded benchmarks are *FFM*, *LUContig*, *Cholesky*, *Barnes*, *OceanContig*, *OceanNoncontig*, *WaterSpatial* and *FFT*.

8.2.1 CMP Model Configuration

The main problem with the CMP model is that only 4 MSHRs are available in each L1 data cache. This hardware structure determines the number of outstanding misses the cache can handle without blocking. For instance, the Intel Pentium 4 processor has 8 MSHRs in its L1 data cache [BBH$^+$04]. Choosing a too low number of MSHRs results in the cache being blocked for long periods of time. Furthermore, this behaviour depends on the timing of cache misses, and therefore it is often different for the same benchmark and different interconnects. In addition, it limits the pressure on the interconnect. Consequently, increasing the number of MSHRs will probably increase performance impact of the choice of interconnect. In a different work, we found that 8 or 16 MSHRs in the L1 cache is a good compromise between area and performance for a 4-core CMP [JN07].

The latencies of the different interconnects were based on the numbers computed by Kumar et al. [KZT05]. However, these numbers are highly dependent on the CMP floorplan and properties of the interconnect. Consequently, more work is needed to explore the realism of these numbers. The impact of this uncertainty is reduced by using the same transfer latency for all interconnects. This is probably not an unreasonable assumption as this delay is mostly given by the distance the signals must travel on the chip. Since the same units are connected in all cases, this delay is likely to be reasonably uniform across interconnects. To summarise, it is the available parallelism in the interconnect that is investigated in this work and not the impact of end-to-end transfer latency.

8.2.2 Use of the M5 Simulator

This section discusses how the M5 simulator can be used better. The main point here is that using the M5 system call emulation thread library is a bad idea. If parallel applications are simulated, it is better to use M5's full system mode. Sadly, the extent of the problems with this library was not even known to the M5 development team when this choice was made. Changing from system call emulation to full system simulation is time consuming, and there was simply not enough time to do this when the problem was discovered. Therefore, the system call emulation mode was used for the experiments. To control the introduction of errors, extensive tracing of the occurrence of bugs was implemented. This strategy worked well and made it possible to remove many obviously erroneous results.

The choice of which cache bank to use in M5 is carried out by allocating a contiguous address range to each bank. For the scientific benchmarks, this results in all accesses going to the same bank. A better way to choose the bank is to use the least significant address bits. This is equivalent to the modulo operation as long as the number of banks is a power of two. The experiments with the scientific workloads use modulo bank selection while the multiprogrammed workloads use the M5 default. This might have resulted in poor chip-wide cache utilisation for the multiprogrammed workloads. Consequently, modulo bank selection should be used in future experiments.

The memory model and memory bus model used in this work are simple. For instance, all accesses to main memory have the same latency. In reality, accesses to adjacent memory blocks can be carried out in parallel which reduce the access latency. Furthermore, the memory bus is a generic bus design which does not model any real memory bus standard. This is probably an advantage in this work as it makes the results easier to interpret. However, a more advanced model is needed for work that has the memory and memory bus as its main focus.

Lastly, simulation of scientific workloads is terminated when one processor has committed a certain number of instructions. This introduces some error as the number of instructions committed by the other processors might vary between configurations. It might be better to finish simulation when the total number of committed instructions reaches a certain number. Another possibility is to simulate for a fixed number of clock cycles. More work is needed to find the best solution to this problem.

8.2.3 Implementation Decisions

There is no throttling mechanism implemented for the coherence protocol. This was not a good choice as it makes the problem of congestion in the interconnect worse. Since the scientific workloads are communication intensive, congestion was observed with all realistic interconnects. A better solution might be to block the L1 cache when there are many active coherence messages. Figuring out the details of this scheme is further work.

The crossbar implemented in this work consider the L1 data cache and L1 instruction cache as different nodes and give both of them a full set of transmission channels. It is likely that these two sets of channels can be replaced by one set with only a minute performance impact. The reason is that there are very few cache misses in the instruction cache. Verifying this assumption is left as further work.

Chapter 9

Conclusion and Further Work

9.1 Conclusion

This report started with a review of the state-of-the-art of CMP design, on-chip CMP interconnects and cache coherence solutions for CMPs. Although a lot of previous work has focused on interconnection networks and cache coherence protocols, only relatively few researches have addressed this problem in a CMP context. Consequently, it is probably still possible to develop better solutions to these problems. Furthermore, older techniques as for instance the Stenström protocol, might prove to be a good match for the new multi-core processor architectures.

The most important part of this work is the exploration of the CMP interconnect design space. In particular, the bus and crossbar interconnects were evaluated with multiprogrammed workloads created from the SPEC2000 [SPEa] benchmark suite and SPLASH-2 scientific applications [WOT^{+}95]. With multiprogrammed workloads, a CMP with a crossbar interconnect performs almost as good as one using the ideal interconnect. Consequently, the crossbar interconnect should be used if this is the predominant application class and the hardware cost can be tolerated. However, this must be balanced against the other parts of the system as optimising the parts independently might not yield the best solution [KZT05]. The bus interconnect does not perform well for some benchmarks and should not be used.

The impact of the interconnect on overall system performance is severe for scientific applications. Unfortunately, some of these results had to be discarded due to problems with the M5 system call emulation thread library implementation. With scientific workloads, both the configurations with the bus and crossbar interconnects experience a considerable performance degradation compared to the ideal interconnect configuration. Furthermore, the performance with the crossbar approaches the performance with the bus when the L1 to L1 communication intensity is high. The reason is that the crossbar has very limited L1 to L1 bandwidth available.

A butterfly interconnection network was proposed to counter the problems observed with the bus and crossbar for scientific applications. For the 8-core CMP used in this work, the butterfly configuration on average performs 3.9 times and 3.8 times better than the bus and crossbar configurations, respectively. These numbers are based on the results from the benchmarks where the M5 problems had small effects and there was little variation in interconnect request rate. For the multiprogrammed workloads, the butterfly configurations perform a bit worse than the crossbar configurations. This is expected, as the crossbar performs very close to the ideal configuration in all experiments with multiprogrammed workloads. However, there is large variation in the performance of the benchmarks within a workload when there is extensive competition for space in the shared L2 cache. Consequently, some fairness scheme is needed if the butterfly is used for this class of workloads.

9.2 Further Work

This section describes possible further work. It is organised as three lists where the first one discusses some further work of a general nature. Then, further work regarding the on-chip interconnection network is discussed. Finally, a few possible future directions for cache coherence protocol investigation is presented. Since this work will be continued in a PhD, it is especially useful to write down these ideas.

The following list describes future work of a general nature:

- The most important further work is probably to rerun the scientific workload experiments in M5's full system mode. In particular, it is interesting to confirm whether the large variation in interconnect request rate between configurations in the *Cholesky* and *Barnes* benchmarks is an architectural effect or due to a problem with M5's system call emulation mode thread implementation.
- *Non-Uniform Cache Access (NUCA)* caches [KBK02] make the difference in access time to cache banks due to the distances between a given core and the banks visible at the architecture level. It is interesting to see how moving to this type of architecture influences the findings in this report.
- There is a considerable body of research on the behaviour of the Splash-2 benchmarks in the SMP context [WOT$^+$95, CGS97]. Do these findings still hold on a CMP platform?

The interconnect investigations have also given some ideas for further work. These are described in the following list:

- The butterfly and crossbar perform well because most channels are lightly loaded. However, this is inefficient in terms of area. By mapping requests to channels in a more intelligent fashion, it should be possible to use these channels more efficiently. A related possibility is to investigate the impact of mapping the instruction and data traffic to the same crossbar transmission channels.
- It is expected that the number of cores in a CMP will grow in the future. Since we already have commercial implementations of 8-core CMPs, it is interesting to investigate the impact increasing the number of cores has on the interconnect.
- The crossbar and butterfly implementations used in this work, blocks when one of the cache banks blocks. However, it is possible to continue to deliver requests to the banks that are not blocked. Quantifying the performance impact of this optimisation is possible further work. In addition, more intelligent blocking strategies can possibly be used to ensure fair use of shared resources.
- The butterfly interconnection network used in this work is only one out of a number of possible butterfly networks. A possible future direction is to investigate the performance of butterflies with a different radix. In this case, the actual chip area consumption and latencies should be estimated. It might be necessary to make a floorplan to get sufficiently good estimates.

Although cache coherence protocol solutions only have had a secondary focus in this work, a few possibilities for future work has been discovered. These are discussed in the following list:

- The Stenström directory protocol is an improvement over the simple MSI directory protocol. Consequently, possible further work is to compare these protocols. This will give an idea of the merits of the Stenström scheme in a CMP.
- Although the Stenström protocol has not been compared to any other directory protocols, some improvements might be possible. Firstly, it might be possible to alleviate the congestion in the interconnect by reducing the number of requests injected into the network. Furthermore, the experimental results seem to indicate the protocol is sensitive to the initial data distribution. A possible strategy is to improve on this distribution by migrating cache blocks to the cache of the processor that most frequently use the data. Furthermore, software prefetches are discarded in the current implementation. It is unclear if it is better to retrieve the block for reading or for writing in this case.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[AG96] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259, New York, NY, USA, 2000. ACM Press.

[ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modelling. *IEEE Computer*, 35(2):59–67, 2002.

[Ali06] Razak Mohammed Ali. DDR2 SDRAM Interfaces for Next-Gen Systems. *Electronic Engineering Times-Asia*, 2006.

[AMD] AMD. AMD Athlon 64 X2 Dual-Core Processor for Desktop. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html`.

[AMD05] AMD. Software Optimization Guide for AMD64 Processors. `http://www.amd.com/gb-uk/Processors/TechnicalResources/0,,30_182_739_7203,00.html`, 2005.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *SIGARCH Comput. Archit. News*, 16(2):280–298, 1988.

[BBH$^+$04] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), 2004.

[BD06] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198, New York, NY, USA, 2006. ACM Press.

[BDH$^+$06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[BM02] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49–58, 1990.

[CGS97] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Cit03] Daniel Citron. MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 52–61, New York, NY, USA, 2003. ACM Press.

[CKB03] S.W. Changkyu Kim; Burger, D.; Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23(6):99–107, 2003.

[CLS05] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[Clu] Clustis2 Cluster Web Page. `http://clustis2.idi.ntnu.no/`.

[CMR$^+$06] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 339–351, Washington, DC, USA, 2006. IEEE Computer Society.

[Cor07] Corsair. TWIN2X2048-6400. `http://www.corsairmemory.com/corsair/products/specs/TWIN2X2048-6400.pdf`, 2007.

[CPV05] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 357–368, Washington, DC, USA, 2005. IEEE Computer Society.

[CS06] Jinchuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. *33nd Annual International Symposium on Computer Architecture (ISCA'06)*, 2006.

[Dal06] William J. Dally. Future Directions for On-Chip Interconnection Networks - Presentation at Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems. `http://www.ece.ucdavis.edu/~ocin06/talks/dally.pdf`, 2006.

[DS07] Haakon Dybdahl and Per Stenström. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[EPS06] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-Network Cache Coherence. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332, Washington, DC, USA, 2006. IEEE Computer Society.

[GLL$^+$98] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 376–387, New York, NY, USA, 1998. ACM Press.

[GMNR06] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to Intel Core Duo Processor Architecture. *Intel Technology Journal*, 10(2), 2006.

[HBK01] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the Design Space of Future CMPs. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society.

[HHM99] M. Horowitz, R. Ho, and K. Mai. The Future of Wires, 1999.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture - A quantitative approach, Third Edition.* Morgan Kaufmann Publishers, 2003.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A quantitative approach, Fourth Edition.* Morgan Kaufmann Publishers, 2007.

[Int06] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://developer.intel.com/products/processor/manuals/index.htm`, 2006.

[ITR06] ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net/`, 2006.

[Jah06] Magnus Jahre. Interprocessor Communication in Chip Multiprocessors. Project Report in TDT 4720 Computer Design and Architecture, Specialisation, 2006.

[JM95] Bruce Jacob and Trevor Mudge. Notes on Calculating Computer Performance. Technical Report 231-95, University of Michigan, March 1995.

[JN07] Magnus Jahre and Lasse Natvig. Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor. *Submitted to Norsk Informatikkonferanse (NIK-2007)*, 2007.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, New York, NY, USA, 2002. ACM Press.

[KFJ$^+$03] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[KJT04] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-Core Chip Multiprocessing. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM Inter-*

national Symposium on Microarchitecture, pages 195–206, Washington, DC, USA, 2004. IEEE Computer Society.

[KKD+06] Nevin Kirman, Meyrem Kirman, Rajeev K. Dokania, Jose F. Martinez, Alyssa B. Apsel, Matthew A. Watkins, and David H. Albonesi. Leveraging Optical Technology in Future Bus-based Chip Multiprocessors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 492–503, Washington, DC, USA, 2006. IEEE Computer Society.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 2004.

[KTJR05] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, 2005.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[KZT05] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. *32nd Annual International Symposium on Computer Architecture (ISCA'05)*, 2005.

[Lan06] Arnt Jørgen Lande. Evaluering av Chip Multiprocessor simulatorer. Master Thesis, 2006.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM Press.

[LNR+06] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 130–141, Washington, DC, USA, 2006. IEEE Computer Society.

[MBH+05] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *HPCA*, pages 328–339, 2005.

[McF93] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[McG06] Harlan McGahn. Niagara 2 Opens the Floodgates. *Microprocessor Report*, 2006.

[MH06] Michael R. Marty and Mark D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 309–320, Washington, DC, USA, 2006. IEEE Computer Society.

[MHW03] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: decoupling performance and correctness. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 182–193, New York, NY, USA, 2003. ACM Press.

[Nor] Norgrid Cluster Web Page. `http://norgrid.ntnu.no/`.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM Press.

[PHC03] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points, 2003.

[SA05] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, Washington, DC, USA, 2005. IEEE Computer Society.

[SF91] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 53–62, New York, NY, USA, 1991. ACM Press.

[SKT+05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[Smi81] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[Smi88] James E. Smith. Characterizing Computer Performance With a Single Number. *Communications of the ACM*, 31(10), October 1988.

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

[SPEb] SPEC CPU 2006 Web Page. `http://www.spec.org/cpu2006/`.

[SQL] SQLite Web Page. `http://www.sqlite.org/`.

[Ste89] P. Stenström. A cache consistency protocol for multiprocessors with multistage networks. *SIGARCH Comput. Archit. News*, 17(3):407–415, 1989.

[Ste90] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, 1990.

[Wol04] Wayne Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 681–685, New York, NY, USA, 2004. ACM Press.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[WPM03] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven Design of Router Microarchitectures in On-chip Networks. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 105, Washington, DC, USA, 2003. IEEE Computer Society.

[ZA05] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Randomly Generated Multiprogram Workloads

A.1 Multiprogram Workloads for 2 CPUs

The randomly generated multiprogrammed workloads used in experiments with 2-way CMPs are shown in table A.1.

A.2 Multiprogram Workloads for 4 CPUs

The randomly generated multiprogrammed workloads used in experiments with 4-way CMPs are shown in table A.2.

A.3 Multiprogram Workloads for 8 CPUs

The randomly generated multiprogrammed workloads used in experiments with 8-way CMPs are shown in table A.3.

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

Table A.1: Randomly Generated Multiprogram Workloads for 2 CPUs

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

Table A.2: Randomly Generated Multiprogram Workloads for 4 CPUs

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

Table A.3: Randomly Generated Multiprogram Workloads for 8 CPUs

Appendix B

Mail Correspondence with M5 Development Team

B.1 First Bug Report

From: magnus.jahre@idi.ntnu.no
To: m5-users@m5sim.org
Subject: Deadlock with Splash benchmarks in SE mode in version 1.1
Date: 27. May 2007

Hi,

I'm using the precompiled Splash benchmarks for M5 version 1.1 and the SE mode. The problem is that many of these benchmarks eventually deadlock. I have traced the problem to the synchronization functions in alpha_tru64_process.cc. Here, all processors end up calling the m5_cond_waitFunc() method and they all suspend. The really bad thing is that if you simulate for a fixed number of clock cycles, it is very difficult to see that something has gone wrong.

I've added the following code to the end of the m5_cond_waitFunc() method to detect the problem:

```
if(process->waitList.size() == process->numCpus()){
fatal("We have a deadlock");
}
```

Have you seen this problem before?

I think it might be a problem with the thread library. However, I have not been able to look into this as I do not have a working cross compiler. Have you been able to build a cross compiler for the Splash benchmarks and the thread libraries?

Regards,
Magnus Jahre

B.2 Reply from Steve Reinhardt

From: stever@eecs.umich.edu
To: m5-users@m5sim.org
Subject: Re: [m5-users] Deadlock with Splash benchmarks in SE mode in version 1.1
Date: 28. May 2007

Hi Magnus,

What inputs are you using for the benchmarks? I don't recall this happening, but they haven't been tested extensively, so it wouldn't be a huge surprise if this happened when you use different input sizes or numbers of processors. The Tru64 pthreads library (like the Linux pthreads library) has a "manager" thread in addition to the application threads. In order to keep the m5 support manageable, we don't create a manager thread (because then you'd have N+1 threads trying to run on only N CPUs). If the application ever gets into the situation where all the worker threads are waiting on the manager thread then you're hosed. We don't see that happen under SPLASH (since they're pretty simple in their use of threads) but it could be you've run into a situation where that's happening.

I don't believe it's possible to build a cross-compiler for Tru64 Alpha binaries; at least we've never been able to. Thus unless you have a native Tru64 Alpha machine you can't really generate new binaries for the existing SPLASH support.

Unfortunately supporting Linux pthreads in SE mode is even harder than under Tru64 (and the situation with Tru64 is that my goal was to support pthreads but I gave up partway through, which is why it's kind of a mess). This question comes up a lot, so if you're interested you can probably find more detailed answers in the mailing list archive.

Actually this question comes up often enough that I'm just going to create a wiki page for it:

http://www.m5sim.org/wiki/index.php/Splash_benchmarks

The bottom line is that you're probably best off running with Linux pthreads in FS mode.

Steve

B.3 Elaboration on First Bug Report

From: magnus.jahre@idi.ntnu.no
To: m5-users@m5sim.org
Subject: Deadlock with Splash benchmarks in SE mode in version 1.1
Date: 28. May 2007

Hi,

Thanks for the quick reply!

First, I agree that switching to FS mode is the better option. I will get FS mode up and running as soon as I can find the time :-)

I'll elaborate a bit on the deadlock problem with Splash in SE mode. I'm using the standard inputs except for with LUNoncontig where I have increased the problem size to 512. The Ocean benchmark is the one that is causing the most trouble. With my extensions (different L1-L2 interconnects and directory coherence) it simulates less than 1 million instructions before all threads suspend. This happens for 2, 4 and 8 cores. To makes sure that it is not a problem with my code, I ran a test with vanilla M5, 8 cores and the MSI, MESI or MOESI coherence protocols. In this case, the problem arises for Barnes, both Ocean benchmarks, both LU benchmarks, WaterSpatial and WaterNSquared with all coherence protocols. Furthermore, FMM has the problem with the MSI protocol and Radix with the MOESI protocol.

If we assume that all threads are waiting for the scheduler thread, a possible dirty hack would be to start up the thread at the head of the wait queue. Then, it should be possible to get at least some results. Of course, it is far from ideal and might create more problems than it solves. What do you think?

In addition, I notice that some benchmarks call the nxm_thread_blockFunc, nxm_blockFunc and nxm_unblockFunc. However, these methods only print their arguments to stdout and return 0. What should these methods do?

Regards,
Magnus

Appendix C

Simulator Extension Code

C.1 Interconnect Extension Code

C.1.1 Interconnect Header File

```
#ifndef __INTERCONNECT_HH__
#define __INTERCONNECT_HH__

#include <iostream>
#include <vector>
#include <queue>
#include <fstream>

#include "mem/base_hier.hh"
#include "interconnect_interface.hh"
#include "interconnect_profile.hh"
#include "sim/eventq.hh"
#include "sim/stats.hh"

#include "cpu/exec_context.hh" // for ExecContext, needed for cpu_id
#include "cpu/base.hh" // for BaseCPU, needed for cpu_id


/** The maximum value of type Tick. */
#define TICK_T_MAX ULL(0x3FFFFFFFFFFFFF)

class InterconnectInterface;
class InterconnectArbitrationEvent;
class InterconnectDeliverQueueEvent;
class InterconnectProfile;

/**
* This class is the parent class of all interconnect extensions. In other
* words, all interconnects are sub-classes of this class.
*
* It has three funcions:
* - Firstly, it defines the interface which all interconnects must
*   implement
* - Secondly, it takes care of registration and administration of
*   interconnect interfaces. This functionality is common to all interconnects.
* - In addition, defines some classes that the interconnects need. These
*   classes are the event objects used to create delays and a two convenience
*   classes that represent requests and deliveries.
*
* @author Magnus Jahre
*/
class Interconnect : public BaseHier
{
    private:

        int masterInterfaceCount;
        int slaveInterfaceCount;
        int totalInterfaceCount;

    protected:

        bool blocked;
        int waitingFor;
        Tick blockedAt;

        int cpu_count;

        InterconnectProfile* profiler;

        std::map<int, int> processorIDToInterconnectIDMap;
        std::map<int, int> interconnectIDToProcessorIDMap;
        std::map<int, int> interconnectIDToL2IDMap;

        std::vector<InterconnectInterface* > masterInterfaces;
        std::vector<InterconnectInterface* > slaveInterfaces;
        std::vector<InterconnectInterface* > allInterfaces;

        /* Statistics variables */
        Stats::Scalar<> totalArbitrationCycles;
        Stats::Scalar<> totalArbQueueCycles;
        Stats::Formula avgArbCyclesPerRequest;
        Stats::Formula avgArbQueueCyclesPerRequest;

        Stats::Scalar<> totalTransferCycles;
        Stats::Scalar<> totalTransQueueCycles;
        Stats::Formula avgTransCyclesPerRequest;
        Stats::Formula avgTransQueueCyclesPerRequest;

        Stats::Vector<> perCpuTotalTransferCycles;
        Stats::Vector<> perCpuTotalTransQueueCycles;

        Stats::Formula avgTotalDelayCyclesPerRequest;

        Stats::Scalar<> requests;
        Stats::Scalar<> arbitratedRequests;
        Stats::Scalar<> sentRequests;
        Stats::Scalar<> nullRequests;
//          Stats::Scalar<> duplicateRequests;
        Stats::Scalar<> numClearBlocked;
        Stats::Scalar<> numSetBlocked;

        /**
        * Convenience class that represents a transfer request.
        */
        class InterconnectRequest{
            public:
                Tick time;
                int fromID;

                /**
                * Default constructor
                *
                * @param _time    The tick the request was issued
                * @param _fromID The ID of the requesting interface
                */
                InterconnectRequest(Tick _time, int _fromID){
                    time = _time;
                    fromID = _fromID;
                }
        };

        /**
         * Convenience class that represents a granted request which is in the
         * process of being delivered.
         */
        class InterconnectDelivery{
            public:
                Tick grantTime;
                int fromID;
                int toID;
                MemReqPtr req;


                /**
                 * Default constructor
                 *
                 * @param _grantTime The tick the request was granted access
                 *                   and the delivery object created
                 * @param _fromID     The ID of the requesting interface
                 * @param _toID       The ID of the destination interface
                 * @param _req        The MemReqPtr that will be delivered
                 */
                InterconnectDelivery(Tick _grantTime,
                                     int _fromID,
                                     int _toID,
                                     MemReqPtr& _req)
                {
                    grantTime = _grantTime;
                    fromID = _fromID;
                    toID = _toID;
                    req = _req;
                }
        };

    public:
        int clock;
        int width;
        int transferDelay;
        int arbitrationDelay;

        std::vector<InterconnectArbitrationEvent *> arbitrationEvents;
        std::vector<InterconnectDeliverQueueEvent* > deliverEvents;

    protected:

        /**
        * Checks that a list of InterconnectRequest objects is sorted
        * in ascending order according to their request times. This is
```

```
        * important because the arbitration methods usually assume that the
        * request list is sorted. It is used in assertions in the subclasses.
        *
        * @param inList The list to check
        *
        * @return True if the list is sorted
        *
        * @see InterconnectRequest
        */
        bool isSorted(std::list<InterconnectRequest*>* inList);

        /**
        * Checks that a list of InterconnectDelivery objects is sorted
        * in ascending order according to their grant times. This is
        * important because the arbitration methods usually assume that the
        * request list is sorted. It is used in assertions in the subclasses.
        *
        * @param inList The list to check
        *
        * @return True if the list is sorted
        *
        * @see InterconnectDelivery
        */
        bool isSorted(std::list<InterconnectDelivery*>* inList);


    public:

        /**
        * This is the default constructor for the Interconnect class. It stores
        * the arguments and initialises some member variables and does some
        * input checking.
        *
        * The interconnect only supports running at the same frequency as the
        * processor core, and there must be at least one CPU in the system.
        *
        * @param _name       The object name from the configuration file. This
        *                    is passed on to BaseHier and SimObject
        * @param _width      The bit width of the transmission lines in the
        *                    interconnect
        * @param _clock      The number of processor cycles in one interconnect
        *                    clock cycle.
        * @param _transDelay The end-to-end transfer delay through the
        *                    interconnect in CPU cycles
        * @param _arbDelay   The lenght of an arbitration in CPU cycles
        * @param _cpu_count  The number of processors in the system
        * @param _hier       Hierarchy parameters for BaseHier
        *
        */
        Interconnect(const std::string &_name,
                     int _width,
                     int _clock,
                     int _transDelay,
                     int _arbDelay,
                     int _cpu_count,
                     HierParams *_hier)
            : BaseHier(_name, _hier){

            width = _width;
            clock = _clock;
            transferDelay = _transDelay;
            arbitrationDelay = _arbDelay;
            cpu_count = _cpu_count;

            if(clock != 1){
                fatal("The interconnects are only implemented to run "
                      "at the same frequency as the CPU core");
            }

            if(cpu_count < 1){
                fatal("There must be at least one CPU in the system");
            }

            masterInterfaceCount = -1;
            slaveInterfaceCount = -1;
            totalInterfaceCount = -1;

            blocked = false;
            blockedAt = -1;
            waitingFor = -1;

        }

        ~Interconnect(){ /* does nothing */ }

        /**
        * This method registers a InterconnectProfiler. The profiler is used to
        * dump selected statistics to a file at regular time intervals.
        *
        * @param _profiler The InterconnectProfiler to use
        */
        void registerProfiler(InterconnectProfile* _profiler){
            profiler = _profiler;
        }

        /**
        * This method is called from the M5 statistics package and initialises
        * the statistics variables used in all interconnects.
        */
        void regStats();

        /**
        * This method is supposed to reset the statistics values. However, it
        * is not used any set-up used in this work and is not implemented.
        */
        void resetStats();

        /**
        * An InterconnectInterface must register with the interconnect to be
        * able to use it. This function is handled by this method.
        *
        * @param interface   A pointer to the interconnect interface that is
        *                    registering itself
        * @param isL2        Is true if the interface represents an L2 cache,
        *                    false otherwise
        * @param processorID The ID of the processor connected to the
        *                    interface. If the cache is not connected to any
        *                    particular processor, -1 should be supplied.
        *
        * @return The ID the interface is given
        */
        int registerInterface(InterconnectInterface* interface,
                              bool isL2,
                              int processorID);

        /**
        * This method makes all registers interconnects reevaluate which
        * address ranges they are responsible for.
        */
        void rangeChange();

        /**
        * The cache might issue requests that are later squashed. The
        * interfaces might detect this situation when they try to retrieve the
        * current request from the cache. This situation needs to be measured
        * as it does cause empty issue slots in the interconnect.
        *
        * The InterconnectInterface calls this method when a squashed request
        * was encountered. In increments a statistic variable that is printed
        * when simulation is finished.
        */
        void incNullRequests();

//          void incDuplicateRequests();

        /**
        * To enable transfers between caches at the same level, a means of
        * translating from interconnect IDs to processor IDs is needed.
        * This information is stored in a map when the interface registers
        * itself and is retrieved through this method.
        *
        * @param processorID The processor ID to translate
        *
        * @return The interconnect ID of the data cache belonging to this
        *         processor
        */
        int getInterconnectID(int processorID);

        /**
        * This method provides statistic values to the InterconnectProfile
        * object. The values are stored in the memory pointed to by the
        * arguments, and the internal counters are reset.
        *
        * @param dataSends      Pointer to a memory area where the number of
        *                       data sends can be stored
        * @param instSends      Pointer to a memory area where the number of
        *                       instruction sends can be stored
        * @param coherenceSends Pointer to a memory area where the number of
        *                       coherence sends can be stored
        * @param totalSends     Total number of sends which is the sum of the
        *                       other request types
        *
        * @see InterconnectProfile
        */
        void getSendSample(int* dataSends,
                           int* instSends,
                           int* coherenceSends,
                           int* totalSends);

        /**
        * Convenience method that finds the ID of the slave interface that is
        * responsible answering requests related to a given address.
        *
        * @param address The address in question
        *
        * @return The interface ID of the interface responsible for the address
        */
        int getTarget(Addr address);

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void request(Tick time, int fromID) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void send(MemReqPtr& req, Tick time, int fromID) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void arbitrate(Tick cycle) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void deliver(MemReqPtr& req,
                                Tick cycle,
                                int toID,
                                int fromID) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void setBlocked(int fromInterface) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void clearBlocked(int fromInterface) = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual int getChannelCount() = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual std::vector<int> getChannelSample() = 0;

        /**
         * This method is commented in the subclasses where it is implemented
         */
        virtual void writeChannelDecriptor(std::ofstream &stream) = 0;
};

/**
* This class creates an arbitation event that is compatible with the M5 event
* queue. It is used by the Interconnect classes to create a time delay from a
* request is recieved untill the arbitration is carried out.
*
* @see Interconnect
* @see SplitTransBus
* @see Crossbar
* @see Butterfly
* @see IdealInterconnect
*
* @author Magnus Jahre
*/
class InterconnectArbitrationEvent : public Event
{

    public:
        Interconnect *interconnect;

        /**
        * Default constructor.
        *
        * @param _interconnect A pointer to the associated interconnect
        */
        InterconnectArbitrationEvent(Interconnect *_interconnect)
            : Event(&mainEventQueue), interconnect(_interconnect)
        {
        }

        /**
        * This method is called when the event is serviced. It searches through
        * the Interconnects arbitration tick queue to find the current clock
        * tick, removes this and calls the arbitrate method in Interconnect.
        * Then, it deletes itself.
        *
        * @see Interconnect
        */
        void process();

        /**
        * @return A textual description of the event
        */
        virtual const char *description();
};

/**
 * This class creates a deliver event that is compatible with the M5 event
 * queue. It is used by the Interconnect classes to create a time delay from a
 * request is granted access untill it is delivered.
 *
 * This event is for use in interconnects that do _not_ use delivery queue.
 * Such classes should use InterconnectDeliverQueueEvent instead.
 *
 * @see InterconnectDeliverQueueEvent
 * @see Interconnect
 * @see SplitTransBus
 * @see Crossbar
 * @see Butterfly
 * @see IdealInterconnect
 *
 * @author Magnus Jahre
 */
class InterconnectDeliverEvent : public Event
{

    public:
        Interconnect *interconnect;
        MemReqPtr req;
        int toID;
        int fromID;

        /**
        * Constructs a delivery event for interconnects that do not use a
        * delivery queue.
        *
        * @param _interconnect A pointer to the interconnect that created the
                               event
        * @param _req          The request to deliver
        * @param _toID         The interface ID the request will be delivered to
        * @param _fromID       The interface ID the request was sent from
        */
        InterconnectDeliverEvent(Interconnect *_interconnect,
                                 MemReqPtr& _req,
                                 int _toID,
                                 int _fromID)
            : Event(&mainEventQueue)
        {
            interconnect = _interconnect;
            req = _req;
            toID = _toID;
            fromID = _fromID;
        }

        /**
        * This method is called when the event is serviced and calls the
        * deliver method in an Interconnect class. Afterwards, it deletes
        * itself.
        *
        * @see Interconnect
        */
        void process();

        /**
         * @return A textual description of the event
         */
        virtual const char *description();
};


/**
 * This class creates a deliver event that is compatible with the M5 event
 * queue. It is used by the Interconnect classes to create a time delay from a
 * request is granted access untill it is delivered.
 *
 * This event is for use in interconnects that do _not_ use delivery queue.
 * Such classes should use InterconnectDeliverEvent instead.
 *
 * @see InterconnectDeliverEvent
 * @see Interconnect
 * @see SplitTransBus
 * @see Crossbar
 * @see Butterfly
 * @see IdealInterconnect
 *
 * @author Magnus Jahre
 */
class InterconnectDeliverQueueEvent : public Event
```

```
{

    public:

        Interconnect *interconnect;

        /**
         * Constructs a delivery event for interconnects that uses a delivery
         * queue.
         *
         * @param _interconnect A pointer to the interconnect that created the
         *                       event
         */
        InterconnectDeliverQueueEvent(Interconnect* _interconnect)
            : Event(&mainEventQueue) {
            interconnect = _interconnect;
    }

    /**
     * This method is called when the event is serviced. First, it removes
     * itself from the delivery queue. Then it calls the deliver method in an
     * Interconnect subclass.
     *
     * Only the tick argument to deliver is provided when this method is
     * serviced. The memory request provided is NULL and the from and to IDs
     * are set to -1.
     *
     * @see Interconnect
    */
    void process(){
        bool found = false;
        int foundIndex = -1;
        for(int i=0;i<interconnect->deliverEvents.size();i++){
            if((InterconnectDeliverQueueEvent*) interconnect->deliverEvents[i]
                == this){
                foundIndex = i;
                found = true;
            }
        }
        assert(found);
        interconnect->deliverEvents.erase(
                interconnect->deliverEvents.begin()+foundIndex);

        MemReqPtr noReq = NULL;
        interconnect->deliver(noReq, this->when(), -1, -1);
        delete this;
    }

    /**
     * @return A textual description of the event
     */
    virtual const char *description(){
        return "InterconnectDeliverQueueEvent";
    }
};

#endif // __INTERCONNECT_HH__
```

C.1.2 Interconnect Code File

```
#include "interconnect.hh"
#include "sim/builder.hh"
#include "mem/base_hier.hh"

using namespace std;


void
Interconnect::regStats(){

    using namespace Stats;


    /* Arbitration */
    totalArbitrationCycles
        .name(name() + ".total_arbitration_cycles")
        .desc("total number of arbitration cycles for all requests")
        ;

    avgArbCyclesPerRequest
        .name(name() + ".avg_arbitration_cycles_per_req")
        .desc("average number of arbitration cycles per requests")
        ;
    avgArbCyclesPerRequest = totalArbitrationCycles / arbitratedRequests;

    totalArbQueueCycles
        .name(name() + ".total_arbitration_queue_cycles")
        .desc("total number of cycles in the arbitration queue "
              "for all requests")
        ;

    avgArbQueueCyclesPerRequest
        .name(name() + ".avg_arbitration_queue_cycles_per_req")
        .desc("average number of arbitration queue cycles per requests")
        ;
    avgArbQueueCyclesPerRequest = totalArbQueueCycles / arbitratedRequests;


    /* Transfer */
    totalTransferCycles
        .name(name() + ".total_transfer_cycles")
        .desc("total number of transfer cycles for all requests")
        ;

    avgTransCyclesPerRequest
        .name(name() + ".avg_transfer_cycles_per_request")
        .desc("average number of transfer cycles per requests")
        ;

    avgTransCyclesPerRequest = totalTransferCycles / sentRequests;

    totalTransQueueCycles
        .name(name() + ".total_transfer_queue_cycles")
        .desc("total number of transfer queue cycles for all requests")
        ;

    avgTransQueueCyclesPerRequest
        .name(name() + ".avg_transfer_queue_cycles_per_request")
        .desc("average number of transfer queue cycles per request")
        ;

    avgTransQueueCyclesPerRequest = totalTransQueueCycles / sentRequests;

    perCpuTotalTransferCycles
        .init(cpu_count)
        .name(name() + ".per_cpu_total_transfer_cycles")
        .desc("total number of transfer cycles per cpu for all requests")
        .flags(total)
        ;

    perCpuTotalTransQueueCycles
        .init(cpu_count)
        .name(name() + ".per_cpu_total_transfer_queue_cycles")
        .desc("total number of cycles in the transfer queue per cpu "
              "for all requests")
        .flags(total)
        ;


    /* Other statistics */
    avgTotalDelayCyclesPerRequest
        .name(name() + ".avg_total_delay_cycles_per_request")
        .desc("average number of delay cycles per request")
        ;

    avgTotalDelayCyclesPerRequest =   avgArbCyclesPerRequest
                                    + avgArbQueueCyclesPerRequest
                                    + avgTransCyclesPerRequest
                                    + avgTransQueueCyclesPerRequest;

    requests
            .name(name() + ".requests")
            .desc("total number of requests")
            ;

    arbitratedRequests
            .name(name() + ".arbitrated_requests")
            .desc("total number of requests that reached arbitration")
            ;

    sentRequests
            .name(name() + ".sent_requests")
            .desc("total number of requests that are actually sent")
            ;

    nullRequests
            .name(name() + ".null_requests")
            .desc("total number of null requests")
            ;

    nullRequests
            .name(name() + ".null_requests")
            .desc("total number of null requests")
            ;

//      duplicateRequests
//              .name(name() + ".duplicate_requests")
//              .desc("total number of duplicate requests")
//              ;
```

```
    numSetBlocked
            .name(name() + ".num_set_blocked")
            .desc("the number of times the interconnect has been blocked")
            ;

    numClearBlocked
            .name(name() + ".num_clear_blocked")
            .desc("the number of times the interconnect has been cleared")
            ;

}

void
Interconnect::resetStats(){
    /* seems like function is not needed as measurements only are taken in the
     * second phase when using fast forwarding. Consequently, it is not
     * implemented.
     */
}

int
Interconnect::registerInterface(InterconnectInterface* interface,
                                bool isL2,
                                int processorID){


    ++totalInterfaceCount;
    allInterfaces.push_back(interface);
    assert(totalInterfaceCount == (allInterfaces.size()-1));

    if(isL2){
        // This is a slave interface (i.e. interface to a L2 bank)
        ++slaveInterfaceCount;
        slaveInterfaces.push_back(interface);
        assert(slaveInterfaceCount == (slaveInterfaces.size()-1));
    }
    else{
        // This is a master interface (i.e. interface to a L1 cache)
        ++masterInterfaceCount;
        masterInterfaces.push_back(interface);
        assert(masterInterfaceCount == (masterInterfaces.size()-1));
    }

    if(processorID != -1){
        assert(processorID >= 0);
        processorIDToInterconnectIDMap.insert(
                make_pair(processorID, totalInterfaceCount));
        interconnectIDToProcessorIDMap.insert(
                make_pair(totalInterfaceCount, processorID));
    }
    else{
        assert(isL2);
        interconnectIDToL2IDMap.insert(
                make_pair(totalInterfaceCount, slaveInterfaceCount));
    }

    return totalInterfaceCount;
}

void
```

```
Interconnect::rangeChange(){
    for(int i=0;i<allInterfaces.size();++i){
        list<Range<Addr> > range_list;
        allInterfaces[i]->getRange(range_list);
    }
}

void
Interconnect::incNullRequests(){
    nullRequests++;
}

// void
// Interconnect::incDuplicateRequests(){
//     duplicateRequests++;
// }

int
Interconnect::getInterconnectID(int processorID){
    if(processorID == -1) return -1;

    map<int,int>::iterator tmp =
            processorIDToInterconnectIDMap.find(processorID);

    // make sure at least one result is returned
    assert(tmp != processorIDToInterconnectIDMap.end());

    return tmp->second;
}

void
Interconnect::getSendSample(int* dataSends,
                            int* instSends,
                            int* coherenceSends,
                            int* totalSends){

    assert(*dataSends == 0);
    assert(*instSends == 0);
    assert(*coherenceSends == 0);
    assert(*totalSends == 0);

    int tmpSends, tmpInsts, tmpCoh, tmpTotal;

    for(int i=0;i<allInterfaces.size();i++){
        allInterfaces[i]->getSendSample(&tmpSends,
                                        &tmpInsts,
                                        &tmpCoh,
                                        &tmpTotal);
        *dataSends += tmpSends;
        *instSends += tmpInsts;
        *coherenceSends += tmpCoh;
        *totalSends += tmpTotal;
    }
}

int
Interconnect::getTarget(Addr address){
    int toID = -1;
    int hitCount = 0;

    for(int i=0;i<allInterfaces.size();i++){
        if(allInterfaces[i]->isMaster()) continue;

        if(allInterfaces[i]->inRange(address)){
            toID = i;
            hitCount++;
        }
    }
    if(hitCount == 0) fatal("No supplier for address in interconnect");
    if(hitCount > 1) fatal("More than one supplier for address in interconnect");

    return toID;
}

bool
Interconnect::isSorted(list<InterconnectDelivery*>* inList){
    InterconnectDelivery* prev = NULL;
    bool first = true;
    bool nonSeqDataExists = false;
    for(list<InterconnectDelivery*>::iterator i=inList->begin();
        i!=inList->end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            if(prev->grantTime > (*i)->grantTime) nonSeqDataExists = true;
            prev = *i;
    }
    return !nonSeqDataExists;
}

bool
Interconnect::isSorted(list<InterconnectRequest*>* inList){
    InterconnectRequest* prev = NULL;
    bool first = true;
    bool nonSeqDataExists = false;
    for(list<InterconnectRequest*>::iterator i=inList->begin();
        i!=inList->end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            if(prev->time > (*i)->time) nonSeqDataExists = true;
            prev = *i;
    }
    return !nonSeqDataExists;
}

void
InterconnectArbitrationEvent::process(){

    int foundIndex = -1;
    int eventHitCount = 0;
    for(int i=0;i<interconnect->arbitrationEvents.size();++i){
        if(interconnect->arbitrationEvents[i] == this){
                foundIndex = i;
                eventHitCount++;
            }
        }
        assert(foundIndex >= 0);
        assert(eventHitCount == 1);
        interconnect->arbitrationEvents.erase(
                interconnect->arbitrationEvents.begin()+foundIndex);

        interconnect->arbitrate(this->when());
        delete this;
}

const char*
InterconnectArbitrationEvent::description(){
        return "Interconnect arbitration event";
}

void
InterconnectDeliverEvent::process(){
        interconnect->deliver(this->req, this->when(), this->toID, this->fromID);
        delete this;
}

const char*
InterconnectDeliverEvent::description(){
        return "Interconnect deliver event";
}


#ifndef DOXYGEN_SHOULD_SKIP_THIS

DEFINE_SIM_OBJECT_CLASS_NAME("Interconnect", Interconnect);

#endif
```

C.1.3 Split Transaction Bus Header File

```
#ifndef __SPLIT_TRANS_BUS_HH__
#define __SPLIT_TRANS_BUS_HH__

#include <iostream>
#include <vector>
#include <queue>

#include "interconnect.hh"

#define DEBUG_SPLIT_TRANS_BUS

/**
* This class implements a Split Transaction Bus interconnect. Here, all
* interfaces are connected to one transmission channel. After arbitration,
* a request is granted both the address bus and the data bus.
*
* Two bus types have been implemented. One version has pipelined arbitration
* and pipelined transfer while the other one is not pipelined. The pipelined
* version is not realistic as it assumes that a request can be injected into
* the data bus from any interface each clock cycle.
*
* @author Magnus Jahre
*/
class SplitTransBus : public Interconnect
{
    private:

        std::list<InterconnectRequest* > requestQueue;
        std::list<InterconnectDelivery* > deliverQueue;

        /* in a pipelined, bi-directional bus we can issue one request
           in each direction each clock cycle */
        std::list<InterconnectRequest* >* slaveRequestQueue;

        bool pipelined;

        void addToList(std::list<InterconnectRequest*>* inList,
                       InterconnectRequest* icReq);

        typedef enum{STB_MASTER, STB_SLAVE, STB_NOT_PIPELINED} grant_type;
        void grantInterface(grant_type gt, Tick cycle);

        void scheduleArbitrationEvent(Tick possibleArbCycle);
        void scheduleDeliverEvent(Tick possibleArbCycle);

        bool doProfile;
        int useCycleSample;

#ifdef DEBUG_SPLIT_TRANS_BUS
        void checkIfSorted(std::list<InterconnectRequest*>* inList);
        void printRequestQueue();
        void printDeliverQueue();
#endif //DEBUG_SPLIT_TRANS_BUS

    public:

        /**
        * This constructor creates a split transaction bus object. If the bus
        * is not pipelined the arbitration delay must be longer or equal to the
        * transfer delay. The reason is that the arbitration method assumes
        * that the previous bus transfer has finished when an arbitration
        * operation finishes.
        *
        * @param _name        The name provided in the config file
        * @param _width       The bus width in bytes
        * @param _clock       The number of processor clock in one bus cycle
        * @param _transDelay  The number of bus cycles one transfer takes
        * @param _arbDelay    The number of bus cycles one arbitration takes
        * @param _cpu_count   The number of processors in the system
        * @param _hier        Hierarchy params for BaseHier
        */
        SplitTransBus(const std::string &_name,
                      int _width,
                      int _clock,
                      int _transDelay,
                      int _arbDelay,
                      int _cpu_count,
                      bool _pipelined,
                      HierParams *_hier)
            : Interconnect(_name,
                           _width,
                           _clock,
                           _transDelay,
                           _arbDelay,
                           _cpu_count,
                           _hier){

            pipelined = _pipelined;

            if(arbitrationDelay < transferDelay && !pipelined){
                fatal("This bus implementation requires the arbitration "
                      "delay to be longer than or equal to the transfer "
                      "delay");
            }

            doProfile = false;
            useCycleSample = 0;

            if(pipelined){
                slaveRequestQueue = new std::list<InterconnectRequest*>;
            }
        }

        /**
        * Default destructor. Deletes the dynamically allocated
        * slaveRequestQueue if the bus is pipelined.
        */
        ~SplitTransBus(){
            if(pipelined){
                delete slaveRequestQueue;
            }
        }

        /**
        * This method is called when a interface needs to use the bus. It adds
        * the request to a queue and schedules an arbitration event. If the bus
        * is pipelined, there are two request queues. The reason is that there
        * are two buses in this case. One runs from the slave interfaces to the
        * master interfaces and one in the opposite direction.
```

```
 *
 * @param time   The clock cycle the request is requested
 * @param fromID The ID of the interface requesting access
 */
void request(Tick time, int fromID);

/**
 * This methods takes creates an InterconnectDelivery object based on
 * the arguments given. Then, an InterconnectDeliverQueueEvent is
 * scheduled after the specified transmission delay. The request
 * queue(s) are kept sorted in ascending order with the oldest
 * request first.
 *
 * @param req    The memory request
 * @param time   The clock cycle the method is called in
 * @param fromID The ID of the interface sending the request
 *
 * @see InterconnectDelivery
 * @see InterconnectDeliverQueueEvent
 */
void send(MemReqPtr& req, Tick time, int fromID);

/**
 * This method is called when an arbitration event is serviced. Each
 * time it is called, it issues at least one request. In the
 * non-pipelined version the oldest request is granted access.
 * In the pipelined version, the oldest master request and the oldest
 * slave request are granted access.
 *
 * The method assumes that the request queues are sorted.
 *
 * @param cycle The clock cycle the arbitration method is called
 */
void arbitrate(Tick cycle);

/**
 * This method is called when a InterconnectDeliverQueueEvent is
 * serviced. It delivers one request each time it is called. If the
 * cache does not block and there are more requests that need to be
 * delivered, it checks whether a delivery event has been registered.
 * This is needed because there might be requests waiting from an
 * earlier cache blocking.
 *
 * @param req    The request to deliver
 * @param cycle  The clock tick the method was called
 * @param toID   The interface ID of the destination interface
 * @param fromID The interface ID of the sender interface
 */
void deliver(MemReqPtr& req, Tick cycle, int toID, int fromID);

/**
 * This method is called if one of the slave cache banks blocks. Then,
 * it removes all arbitration and deliver events. Requests that arrive
 * while a cache bank is blocked are simply queued.
 *
 * @param fromInterface The interface that is blocked
 */
void setBlocked(int fromInterface);

/**
 * This method is called when a slave cache can recieve requests again.
```

```
        *
        * @param fromInterface The interface that is no longer blocked
        */
        void clearBlocked(int fromInterface);


        /**
        * This method is called from the InterconnectProfile class when it
        * needs to know how many transmission channels the bus has.
        *
        * @return 1 since the bus only has one channel
        *
        * @see InterconnectProfile
        */
        int getChannelCount(){
            return 1;
        }

        /**
        * This method is called from the InterconnectProfile class and returns
        * the number of clock cycles the bus was in use since the last time it
        * was called.
        *
        * @return The number of clock cycles the bus was used since the last
        *         time the method was called.
        *
        * @see InterconnectProfile
        */
        std::vector<int> getChannelSample();

        /**
        * This method writes a desciption of the transmission channels used to
        * the provided stream.
        *
        * @param stream The stream to write to
        */
        void writeChannelDecriptor(std::ofstream &stream){
            stream << "0: The shared bus\n";
        }

};
#endif // SPLIT_TRANS_BUS_HH
```

C.1.4 Split Transaction Bus Code File

```
#include "sim/builder.hh"
#include "split_trans_bus.hh"

using namespace std;

void
SplitTransBus::request(Tick time, int fromID){

    requests++;

    assert(fromID >= 0);

    // keep linked list of requests sorted at all times
    // first request takes priority over later requests at same cycle
    InterconnectRequest* newReq = new InterconnectRequest(time, fromID);
    if(pipelined){
        if(allInterfaces[fromID]->isMaster()){
            addToList(&requestQueue, newReq);
        }
        else{
            addToList(slaveRequestQueue, newReq);
        }
    }
    else{
        addToList(&requestQueue, newReq);
    }


#ifdef DEBUG_SPLIT_TRANS_BUS
    checkIfSorted(&requestQueue);
    if(pipelined) checkIfSorted(slaveRequestQueue);
#endif //DEBUG_SPLIT_TRANS_BUS

    if(!blocked){

        if(!pipelined){
            if(arbitrationEvents.empty()){
                scheduleArbitrationEvent(time + arbitrationDelay);
            }
            else{
                Tick nextArbCycle = TICK_T_MAX;
                int hitIndex = -1;
                for(int i=0;i<arbitrationEvents.size();i++){
                    if(arbitrationEvents[i]->when() < nextArbCycle){
                        nextArbCycle = arbitrationEvents[i]->when();
                        hitIndex = i;
                    }
                }
                assert(nextArbCycle < TICK_T_MAX);

                if(nextArbCycle > (time + arbitrationDelay)){
                    /* the arbitration events are out of synch */
                    for(int i=0;i<arbitrationEvents.size();i++){
                        if(arbitrationEvents[i]->scheduled()){
                            arbitrationEvents[i]->deschedule();
                        }
                        delete arbitrationEvents[i];
                    }
                    arbitrationEvents.clear();

                    scheduleArbitrationEvent(time + arbitrationDelay);
                }

            }
        }
        else{
            scheduleArbitrationEvent(time + arbitrationDelay);
        }
    }
}

void
SplitTransBus::addToList(std::list<InterconnectRequest*>* inList,
                         InterconnectRequest* icReq){

    list<InterconnectRequest*>::iterator findPos;
    for(findPos=inList->begin();
        findPos!=inList->end();
        findPos++){
            InterconnectRequest* tempReq = *findPos;
            if(icReq->time < tempReq->time) break;
        }

    inList->insert(findPos, icReq);
}

void
SplitTransBus::scheduleArbitrationEvent(Tick possibleArbCycle){

    assert(!blocked);

    bool addArbCycle = true;
    for(int i=0;i<arbitrationEvents.size();++i){
        if(arbitrationEvents[i]->when() == possibleArbCycle){
            addArbCycle = false;
        }
    }

    if(addArbCycle){
        InterconnectArbitrationEvent* event =
                new InterconnectArbitrationEvent(this);
        event->schedule(possibleArbCycle);
        arbitrationEvents.push_back(event);
    }
}

void
SplitTransBus::arbitrate(Tick cycle){

    assert(!blocked);
    if(pipelined) assert(!requestQueue.empty() || !slaveRequestQueue->empty());
    else assert(!requestQueue.empty());

    if(pipelined){
        grantInterface(STB_SLAVE, cycle);
        grantInterface(STB_MASTER, cycle);
    }
    else{
        grantInterface(STB_NOT_PIPELINED, cycle);
    }


    if(!requestQueue.empty()){
        Tick nextReqTime = requestQueue.front()->time;

        if(pipelined){
            if(nextReqTime <= (cycle - arbitrationDelay)){
                scheduleArbitrationEvent(cycle + 1);
            }
            else{
                scheduleArbitrationEvent(nextReqTime + arbitrationDelay);
            }
        }
        else{
            if(nextReqTime <= cycle){
                scheduleArbitrationEvent(cycle + arbitrationDelay);
            }
            else{
                scheduleArbitrationEvent(nextReqTime + arbitrationDelay);
            }
        }
    }

    if(pipelined){
        if(!slaveRequestQueue->empty()){
            Tick nextReqTime = slaveRequestQueue->front()->time;

            if(nextReqTime <= (cycle - arbitrationDelay)){
                scheduleArbitrationEvent(cycle + 1);
            }
            else{
                scheduleArbitrationEvent(nextReqTime + arbitrationDelay);
            }
        }
    }
}

void
SplitTransBus::grantInterface(grant_type gt, Tick cycle){

    Tick goodReqTime = cycle - arbitrationDelay;
    InterconnectRequest* grantReq;

    /* remove the request from the correct queue */
    switch(gt){
        case STB_NOT_PIPELINED:
            grantReq = requestQueue.front();
            requestQueue.pop_front();
            break;

        case STB_MASTER:
            if(requestQueue.empty()) return;
            grantReq = requestQueue.front();

            /* check if requests are available */
            if(grantReq->time > goodReqTime) return;
            requestQueue.pop_front();
            break;

        case STB_SLAVE:
            if(slaveRequestQueue->empty()) return;
            grantReq = slaveRequestQueue->front();

            /* check if requests are available */
            if(grantReq->time > goodReqTime) return;
            slaveRequestQueue->pop_front();
            break;

        default:
            fatal("Unknown grant_type encountered");

    }

    /* grant access */
    allInterfaces[grantReq->fromID]->grantData();

    /* update statistics */
    arbitratedRequests++;
    totalArbQueueCycles += ((cycle - grantReq->time) - arbitrationDelay);
    totalArbitrationCycles += arbitrationDelay;

    delete grantReq;
}


void
SplitTransBus::send(MemReqPtr& req, Tick time, int fromID){

    assert(!blocked);
    assert((req->size / width) <= 1);

    bool isFromMaster = false;
    if(allInterfaces[fromID]->isMaster()) isFromMaster = true;

    if(req->toInterfaceID != -1){
        // L1 to L1 request
        deliverQueue.push_back(
                new InterconnectDelivery(time,
                                         fromID,
                                         req->toInterfaceID,
                                         req));
    }
    else if(isFromMaster){
        // Try all slaves and check if they can supply the needed data
        int successCount = 0;
        int toID = -1;
        for(int i=0;i<allInterfaces.size();++i){

            if(allInterfaces[i]->isMaster()) continue;

            if(allInterfaces[i]->inRange(req->paddr)){
                successCount++;
                toID = i;
            }
        }

        if(successCount == 0){
            fatal("No supplier for data on SplitTransBus");
        }
        if(successCount > 1){
            fatal("More than one supplier for data on SplitTransBus");
        }

        /* deliver to L2 cache */
        deliverQueue.push_back(
                new InterconnectDelivery(time, fromID, toID, req));
    }
    else{
        /* deliver to L1 cache */
        deliverQueue.push_back(
                new InterconnectDelivery(time,
                                         fromID,
                                         req->fromInterfaceID,
                                         req));
    }

#ifdef DEBUG_SPLIT_TRANS_BUS
    /* check that the queue is sorted */
    InterconnectDelivery* prev = NULL;
    bool first = true;
    for(list<InterconnectDelivery*>::iterator i=deliverQueue.begin();
        i!=deliverQueue.end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            assert(prev->grantTime <= (*i)->grantTime);
            prev = *i;
        }
#endif //DEBUG_SPLIT_TRANS_BUS

    if(doProfile) useCycleSample += transferDelay;

    scheduleDeliverEvent(time + transferDelay);
}

void
SplitTransBus::scheduleDeliverEvent(Tick possibleArbCycle){

    bool addEvent = true;
    for(int i=0;i < deliverEvents.size();i++){
        if(deliverEvents[i]->when() == possibleArbCycle){
            addEvent = false;
        }
    }

    if(addEvent){

        InterconnectDeliverQueueEvent* event =
                new InterconnectDeliverQueueEvent(this);
        event->schedule(possibleArbCycle);
        deliverEvents.push_back(event);
    }
}


void
SplitTransBus::deliver(MemReqPtr& req, Tick cycle, int toID, int fromID){
    assert(!blocked);
    assert(!deliverQueue.empty());

    InterconnectDelivery* delivery = deliverQueue.front();
    deliverQueue.pop_front();

    /* update statistics */
    sentRequests++;
    int queueTime = (cycle - delivery->grantTime) - transferDelay;
    totalTransQueueCycles += queueTime;
    totalTransferCycles += transferDelay;

    int curCpuId = delivery->req->xc->cpu->params->cpu_id;
    perCpuTotalTransQueueCycles[curCpuId] += queueTime;
    perCpuTotalTransferCycles[curCpuId] += transferDelay;

    int retval = BA_NO_RESULT;
    assert(delivery->toID > -1);
    if(allInterfaces[delivery->toID]->isMaster()){
        allInterfaces[delivery->toID]->deliver(delivery->req);
    }
    else{
        retval = allInterfaces[delivery->toID]->access(delivery->req);
    }

    delete delivery;

    if(retval != BA_BLOCKED){
        /* see if we need to schedule another delivery */
        if(!deliverQueue.empty()){
            InterconnectDelivery* nextDelivery = deliverQueue.front();
            if(nextDelivery->grantTime <= (cycle - transferDelay)){
                if(pipelined) scheduleDeliverEvent(cycle + 1);
                else scheduleDeliverEvent(cycle + transferDelay);
            }
            else{
                scheduleDeliverEvent(nextDelivery->grantTime + transferDelay);
            }
        }
    }
}

void
SplitTransBus::setBlocked(int fromInterface){

    if(blocked) warn("SplitTransBus blocking on a second cause");

    blocked = true;
    numSetBlocked++;
    waitingFor = fromInterface;

    /* remove all scheduled arbitration events */
    for(int i=0;i<arbitrationEvents.size();++i){
        if (arbitrationEvents[i]->scheduled()) {
            arbitrationEvents[i]->deschedule();
        }
        delete arbitrationEvents[i];
    }
    arbitrationEvents.clear();

    /* remove all deliver events */
    for(int i=0;i<deliverEvents.size();i++){
        if(deliverEvents[i]->scheduled()){
            deliverEvents[i]->deschedule();
        }
        delete deliverEvents[i];
    }
    deliverEvents.clear();

    blockedAt = curTick;
}

void
SplitTransBus::clearBlocked(int fromInterface){

    assert(blocked);
    assert(blockedAt >= 0);
    if (blocked && waitingFor == fromInterface) {
        blocked = false;

        if(!requestQueue.empty()){
            Tick min = requestQueue.front()->time;

            if(min >= curTick){
                scheduleArbitrationEvent(min + arbitrationDelay);
            }
            else{
                scheduleArbitrationEvent(curTick + arbitrationDelay);
            }

        }

        if(pipelined){
            if(!slaveRequestQueue->empty()){
                Tick min = slaveRequestQueue->front()->time;
                if(min >= curTick){
                    scheduleArbitrationEvent(min + arbitrationDelay);
                }
                else{
                    scheduleArbitrationEvent(curTick + arbitrationDelay);
                }
            }
        }

        if(!deliverQueue.empty()){
            Tick min = deliverQueue.front()->grantTime;
            if(min >= curTick){
                scheduleDeliverEvent(min + transferDelay);
            }
            else{
                scheduleDeliverEvent(curTick + transferDelay);
            }
        }

        numClearBlocked++;

        blockedAt = -1;
    }
}

vector<int>
SplitTransBus::getChannelSample(){
    if(!doProfile) doProfile = true;

    std::vector<int> retval(1, 0);
    retval[0] = useCycleSample;
    useCycleSample = 0;

    return retval;
}

#ifdef DEBUG_SPLIT_TRANS_BUS

void
SplitTransBus::checkIfSorted(std::list<InterconnectRequest* >* inList){
    /* check that the queue is sorted */
    InterconnectRequest* prev = NULL;
    bool first = true;
    for(list<InterconnectRequest*>::iterator i=inList->begin();
        i!=inList->end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            assert(prev->time <= (*i)->time);
            prev = *i;
    }
}

void
SplitTransBus::printRequestQueue(){
    cout << "Request queue: ";
    for(list<InterconnectRequest*>::iterator i = requestQueue.begin();
        i != requestQueue.end();
        i++){
        cout << "("
                << (*i)->fromID
                << ", "
                << (*i)->time
                << ") ";
    }
    cout << "\n";

    if(pipelined){
        cout << "Slave request queue: ";
        for(list<InterconnectRequest*>::iterator i = slaveRequestQueue->begin();
            i != slaveRequestQueue->end();
            i++){
                cout << "("
                        << (*i)->fromID
                        << ", "
                        << (*i)->time
                        << ") ";
        }
        cout << "\n";
    }
}

void
```

```
SplitTransBus::printDeliverQueue(){
    cout << "Deliver queue: ";
    for(list<InterconnectDelivery*>::iterator i = deliverQueue.begin();
        i != deliverQueue.end();
        i++){
            cout << "("
                   << (*i)->fromID
                   << ", "
                   << (*i)->toID
                   << ", "
                   << (*i)->grantTime
                   << ") ";
    }
    cout << "\n";
}

#endif //DEBUG_SPLIT_TRANS_BUS

#ifndef DOXYGEN_SHOULD_SKIP_THIS

BEGIN_DECLARE_SIM_OBJECT_PARAMS(SplitTransBus)
        Param<int> width;
        Param<int> clock;
        Param<int> transferDelay;
        Param<int> arbitrationDelay;
        Param<int> cpu_count;
        Param<bool> pipelined;
        SimObjectParam<HierParams *> hier;
END_DECLARE_SIM_OBJECT_PARAMS(SplitTransBus)

BEGIN_INIT_SIM_OBJECT_PARAMS(SplitTransBus)
        INIT_PARAM(width, "bus width in bytes"),
        INIT_PARAM(clock, "bus clock"),
        INIT_PARAM(transferDelay, "bus transfer delay in CPU cycles"),
        INIT_PARAM(arbitrationDelay, "bus arbitration delay in CPU cycles"),
        INIT_PARAM(cpu_count, "the number of CPUs in the system"),
        INIT_PARAM(pipelined, "true if the bus has pipelined arbitration "
                              "and transmission"),
        INIT_PARAM_DFLT(hier,
                        "Hierarchy global variables",
                        &defaultHierParams)
END_INIT_SIM_OBJECT_PARAMS(SplitTransBus)

CREATE_SIM_OBJECT(SplitTransBus)
{
    return new SplitTransBus(getInstanceName(),
                             width,
                             clock,
                             transferDelay,
                             arbitrationDelay,
                             cpu_count,
                             pipelined,
                             hier);
}

REGISTER_SIM_OBJECT("SplitTransBus", SplitTransBus)

#endif //DOXYGEN_SHOULD_SKIP_THIS
```

C.1.5 Butterfly Header File

```
#ifndef __BUTTERFLY_HH__
#define __BUTTERFLY_HH__

#include "interconnect.hh"


/**
* This class implements a butterfly interconnect. It was only developed to
* investigate the performance of a multistage interconnection network.
* Consequently, is only possible to configure it to represent a subset of all
* possible butterfly networks.
*
* In particular, it handles 2, 4 or 8 processor cores. The reason is that the
* mapping from interface to butterfly node is defined in the constructor. Since
* 8 processors are the maximum number used in this work, it was not prioritised
* to add support for more processors. Furthermore, only radix 2 switches is
* supported and only L2 caches with 4 banks.
*
* Path diversity can be added to a butterfly by adding extra stages. This
* implementation has no path diversity.
*
* @author Magnus Jahre
*/
class Butterfly : public Interconnect
{
    private:
        int switchDelay;
        int radix;
        int butterflyCpuCount;
        int butterflyCacheBanks;
        int terminalNodes;
        int stages;
        int switches;
        int butterflyHeight;
        int hopCount;
        int chanBetweenStages;

        std::map<int, int> cpuIDtoNode;
        std::map<int, int> l2IDtoNode;

        std::list<InterconnectRequest*> requestQueue;
        std::list<InterconnectDelivery*> deliverQueue;

        std::vector<bool> butterflyStatus;
        std::vector<int> channelUsage;

        std::vector<int> blockedInterfaces;

    public:

        /**
        * This constructor creates the interface to node mapping for a given
        * number of CPUs. Furthermore, a number of convenience values are
        * computed. Examples of such values are the width and height of the
        * butterfly.
        *
        * @param _name         The name given in the configuration file
        * @param _width        The width of the transmission channels
        * @param _clock       The number of processor clock cycles in one
        *                     interconnect clock cycle
        * @param _transDelay  The transfer delay _per_ _channel_ in the
        *                     butterfly
        * @param _arbDelay    Arbitration delay for the Interconnect
        *                     constructor. This should be set to 0 as there is
        *                     no explicit arbitration in a butterfly.
        * @param _cpu_count   The number of cpus in the system
        * @param _hier        Hierarchy parameters for BaseHier
        * @param _switchDelay The delay through the switches in the butterfly
        * @param _radix       The number of inputs or outputs for each switch
        *                     (only 2 are supported in this implementation).
        * @param _banks       The number of L2 banks (only 4 are supported in
        *                     this implementation).
        */
        Butterfly(const std::string &_name,
                  int _width,
                  int _clock,
                  int _transDelay,
                  int _arbDelay,
                  int _cpu_count,
                  HierParams *_hier,
                  int _switchDelay,
                  int _radix,
                  int _banks);

        /**
        * This destructor does nothing.
        */
        ~Butterfly(){
            /* noop */
        }

        /**
        * This method puts the request into a queue and schedules an
        * arbitration event if needed. The request queue is kept sorted in
        * ascending order on the clock cycle it was recieved as this simplifies
        * the arbitration method.
        *
        * @param time   The clock cycle the method was called
        * @param fromID The interface ID of the requesting interface
        */
        void request(Tick time, int fromID);

        /**
        * This method is called from an interface when it is granted access. It
        * computes the interface ID of the recipient based on the request given
        * and adds this to a delivery queue. Then, it schedules a delivery
        * event if needed.
        *
        * @param req    The memory request to send.
        * @param time   The clock cycle the method was called at.
        * @param fromID The interface ID of the sender interface.
        */
        void send(MemReqPtr& req, Tick time, int fromID);

        /**
        * This method is called when an arbitration event is serviced. It
        * attempts to grant access to as many interfaces as possible given the
        * limitations of the butterfly interconnect. Since the request queue
        * is sorted, the older requests are prioritised.
```

```
*
* If all requests can not be granted at a given cycle, an arbitration
* event is scheduled at the next clock cycle if at least one request is
* old enough to be scheduled at this cycle. If not, an arbitration
* event is added at the request time + arbitration delay.
*
* @param cycle The clock cycle the method is called.
*/
void arbitrate(Tick cycle);

/**
* This method tries to deliver as many requests as possible to its
* destination. Only, requests that have experienced the defined delay
* can be delivered. However, if an L2 bank blocks, all requests that
* are old enough might not be delivered. Since the delivery queue
* is kept sorted, the oldest requests are delivered first.
*
* Since this class uses a delivery queue, all parameters except
* cycle are discarded.
*
* @param req     Not used, must be NULL.
* @param cycle   The clock cycle the method is called.
* @param toID    Not used, must be -1.
* @param fromID  Not used, must be -1.
*/
void deliver(MemReqPtr& req, Tick cycle, int toID, int fromID);

/**
* This method is called when a L2 bank blocks. It deschedules all
* arbitration events and delivery events. Consequently, no requests
* are delivered to interfaces that are not blocked either.
*
* @param fromInterface The ID of the interface that has blocked
*/
void setBlocked(int fromInterface);

/**
* This method is called when a L2 bank becomes unblocked. If there are
* waiting requests or deliveries, new arbitration events or deliver
* events are scheduled respectively.
*
* @param fromInterface The ID of the interface that has blocked
*/
void clearBlocked(int fromInterface);

/**
* This method returns the number of transmission channels in the
* interconnect and is used by the InterconnectProfile class.
*
* @return The number of transmission channels
*
* @see InterconnectProfile
*/
int getChannelCount();

/**
* This method returns the number of cycles the different channels was
* occupied since it was called last.
*
* @return The number of clock cycles each channel was used since last
*         time the method was called.
```

```
        *
        * @see InterconnectProfile
        */
        std::vector<int> getChannelSample();

        /**
        * This method writes a description of the different channels to
        * the provided stream.
        *
        * @param stream The output stream to write to.
        *
        * @see InterconnectProfile
        */
        void writeChannelDecriptor(std::ofstream &stream);

    private:

        void scheduleArbitrationEvent(Tick candidateTime);

        bool setChannelsOccupied(int fromInterfaceID, int toInterfaceID);

        int getDestinationId(int fromID);

        void printChannelStatus();
};

#endif //__BUTTERFLY_HH__
```

C.1.6 Butterfly Code File

```
#include "sim/builder.hh"
#include "butterfly.hh"

#include <math.h>

using namespace std;

Butterfly::Butterfly(const std::string &_name,
                     int _width,
                     int _clock,
                     int _transDelay,
                     int _arbDelay,
                     int _cpu_count,
                     HierParams *_hier,
                     int _switchDelay,
                     int _radix,
                     int _banks)
                : Interconnect(_name,
                               _width,
                               _clock,
                               _transDelay,
                               _arbDelay,
                               _cpu_count,
                               _hier)
{
    switchDelay = _switchDelay;
    radix = _radix;
    butterflyCpuCount = _cpu_count;
    butterflyCacheBanks = _banks;

    if(butterflyCacheBanks != 4){
        fatal("mappings only implemented for 4 L2 cache banks");
    }

    if(cpu_count == 2){

        cpuIDtoNode[0] = 0;
        cpuIDtoNode[1] = 1;

        l2IDtoNode[0] = 2;
        l2IDtoNode[1] = 2;
        l2IDtoNode[2] = 3;
        l2IDtoNode[3] = 3;

        terminalNodes = 4;
    }
    else if(cpu_count == 4){

        cpuIDtoNode[0] = 0;
        cpuIDtoNode[1] = 1;
        cpuIDtoNode[2] = 2;
        cpuIDtoNode[3] = 3;

        l2IDtoNode[0] = 4;
        l2IDtoNode[1] = 5;
        l2IDtoNode[2] = 6;
        l2IDtoNode[3] = 7;
        terminalNodes = 8;
    }
    else if(cpu_count == 8){

        cpuIDtoNode[0] = 0;
        cpuIDtoNode[1] = 1;
        cpuIDtoNode[2] = 2;
        cpuIDtoNode[3] = 3;
        cpuIDtoNode[4] = 4;
        cpuIDtoNode[5] = 5;
        cpuIDtoNode[6] = 6;
        cpuIDtoNode[7] = 7;

        cpuIDtoNode[8] = -1;
        cpuIDtoNode[9] = -1;
        cpuIDtoNode[10] = -1;
        cpuIDtoNode[11] = -1;

        l2IDtoNode[0] = 12;
        l2IDtoNode[1] = 13;
        l2IDtoNode[2] = 14;
        l2IDtoNode[3] = 15;

        terminalNodes = 16;
    }
    else{
        fatal("The butterfly only supports 2, 4 or 8 processors");
    }


    // compute topology utility vars

    // ceil needed because the answer is not completely accurate
    double tmp = (log10((double) terminalNodes) / log10((double) radix));
    stages = (int) ceil(tmp-1e-9);

    switches = stages * (terminalNodes / radix);
    butterflyHeight = (terminalNodes / radix);
    hopCount = stages + 1;
    chanBetweenStages = butterflyHeight * radix;
    int totalChannels = hopCount * chanBetweenStages;

    butterflyStatus.insert(butterflyStatus.begin(), totalChannels, false);
    channelUsage.insert(channelUsage.begin(), totalChannels, 0);

    if(radix != 2) fatal("Only radix 2 butterflies are implemented");
}

void
Butterfly::request(Tick time, int fromID){

    requests++;

    if(requestQueue.empty() || requestQueue.back()->time < time){
        requestQueue.push_back(new InterconnectRequest(time, fromID));
    }
    else{
        list<InterconnectRequest*>::iterator pos;
        for(pos = requestQueue.begin();
            pos != requestQueue.end();
            pos++){
            if((*pos)->time > time) break;
        }
        requestQueue.insert(pos, new InterconnectRequest(time, fromID));
    }

    assert(isSorted(&requestQueue));

    if(!blocked) scheduleArbitrationEvent(time + arbitrationDelay);
}

void
Butterfly::send(MemReqPtr& req, Tick time, int fromID){

    assert(!blocked);
    assert((req->size / width) <= 1);

    int toID = -1;
    if(allInterfaces[fromID]->isMaster() && req->toInterfaceID != -1){
        toID = req->toInterfaceID;
    }
    else if(allInterfaces[fromID]->isMaster()){
        toID = getTarget(req->paddr);
    }
    else{
        toID = req->fromInterfaceID;
    }

    deliverQueue.push_back(new InterconnectDelivery(time, fromID, toID, req));

    /* check if we need to schedule a deliver event */
    Tick deliverTime = time + (stages*switchDelay + hopCount*transferDelay);
    bool found = false;
    for(int i=0;i<deliverEvents.size();i++){
        if(deliverEvents[i]->when() == deliverTime) found = true;
    }

    if(!found){
        InterconnectDeliverQueueEvent* event =
                new InterconnectDeliverQueueEvent(this);
        event->schedule(deliverTime);
        deliverEvents.push_back(event);
    }
}

void
Butterfly::arbitrate(Tick cycle){

    // reset internal state
    for(int i=0;i<butterflyStatus.size();i++) butterflyStatus[i] = false;

    list<InterconnectRequest* > notGrantedReqs;
    Tick legalRequestTime = cycle - arbitrationDelay;

    list<InterconnectRequest*>::iterator pos;
    while(!requestQueue.empty()){
        if(requestQueue.front()->time <= legalRequestTime){

            int toInterface = getDestinationId(requestQueue.front()->fromID);
            if(toInterface == -1){
                // null request, remove
                delete requestQueue.front();
                requestQueue.pop_front();
                continue;
            }

            if(setChannelsOccupied(
                    requestQueue.front()->fromID,
                    toInterface)){

                // update statistics
                arbitratedRequests++;
                totalArbQueueCycles +=
                    (cycle - requestQueue.front()->time) - arbitrationDelay;
                totalArbitrationCycles += arbitrationDelay;

                // grant access
                allInterfaces[requestQueue.front()->fromID]->grantData();
                delete requestQueue.front();
                requestQueue.pop_front();

            }
            else{
                notGrantedReqs.push_back(requestQueue.front());
                requestQueue.pop_front();
            }

        }
        else{
            notGrantedReqs.push_back(requestQueue.front());
            requestQueue.pop_front();
        }
    }

    if(!notGrantedReqs.empty()){
        // there where requests we could not issue
        // put them back in the queue and schedule new arb event
        assert(requestQueue.empty());
        requestQueue.splice(requestQueue.begin(), notGrantedReqs);

        if(requestQueue.front()->time <= cycle){
            scheduleArbitrationEvent(cycle+1);
        }
        else{
            scheduleArbitrationEvent(
                    requestQueue.front()->time + arbitrationDelay);
        }
    }

    assert(butterflyStatus.size() == channelUsage.size());
    for(int i=0;i<butterflyStatus.size();i++){
        if(butterflyStatus[i]) channelUsage[i]++;
    }
}

bool
Butterfly::setChannelsOccupied(int fromInterfaceID, int toInterfaceID){

    assert(fromInterfaceID >= 0 && toInterfaceID >=  0);

    // translate into butterfly node IDs
    int fromNodeId = (allInterfaces[fromInterfaceID]->isMaster() ?
                          cpuIDtoNode[interconnectIDToProcessorIDMap[fromInterfaceID]]
                    : l2IDtoNode[interconnectIDToL2IDMap[fromInterfaceID]]);

    int toNodeId = (allInterfaces[toInterfaceID]->isMaster() ?
                    cpuIDtoNode[interconnectIDToProcessorIDMap[toInterfaceID]]
                  : l2IDtoNode[interconnectIDToL2IDMap[toInterfaceID]]);

    // store old state in case we can't grant the request
    vector<bool> tmpState = butterflyStatus;

    int atSwitch = -1;
    for(int i=0;i<hopCount;i++){

        if(i == 0){

            if(butterflyStatus[fromNodeId]){
                butterflyStatus = tmpState;
                return false;
            }

            butterflyStatus[fromNodeId] = true;

            atSwitch = fromNodeId / radix;

        }
        else if(i == hopCount-1){
            int lastStageChanID = (chanBetweenStages * i) + toNodeId;
            if(butterflyStatus[lastStageChanID]){
                butterflyStatus = tmpState;
                return false;
            }
            butterflyStatus[lastStageChanID] = true;
        }
        else{

            int useChannelNum = -1;
            if((toNodeId & (1 << (stages - i))) > 0) useChannelNum = 1;
            else useChannelNum = 0;
            int channelID = (atSwitch * 2) + useChannelNum;

            int offset = 1 << (stages - i - 1);
            int nextSwitch = -1;
            if((atSwitch & offset) == 0 && useChannelNum == 1){
                nextSwitch = atSwitch + offset;
            }
            else if((atSwitch & offset) > 0 && useChannelNum == 0){
                nextSwitch = atSwitch - offset;
            }
            else{
                nextSwitch = atSwitch;
            }

            int stageOffset = chanBetweenStages * i;
            if(butterflyStatus[stageOffset + channelID]){
                butterflyStatus = tmpState;
                return false;
            }

            butterflyStatus[stageOffset + channelID] = true;

            atSwitch = nextSwitch;
        }
    }
    return true;
}

void
Butterfly::deliver(MemReqPtr& req, Tick cycle, int toID, int fromID){

    assert(!blocked);

    assert(!req);
    assert(toID == -1);
    assert(fromID == -1);

    assert(isSorted(&deliverQueue));

    int butterflyTransDelay = stages*switchDelay + hopCount*transferDelay;
    Tick legalGrantTime = cycle - butterflyTransDelay;

    /* attempt to deliver as many requests as possible */
    /* since the queue is sorted, starvation is not possible */
    while(!deliverQueue.empty()){
        InterconnectDelivery* delivery = deliverQueue.front();

        /* check if this grant has experienced the proper delay */
        /* since the requests are sorted, we know that all other are later */
        if(delivery->grantTime > legalGrantTime) break;

        deliverQueue.pop_front();

        /* update statistics */
        sentRequests++;
        int curCpuId = delivery->req->xc->cpu->params->cpu_id;
        int queueCycles = (cycle - delivery->grantTime) - butterflyTransDelay;

        totalTransQueueCycles += queueCycles;
        totalTransferCycles += butterflyTransDelay;
        perCpuTotalTransQueueCycles[curCpuId] += queueCycles;
        perCpuTotalTransferCycles[curCpuId] += butterflyTransDelay;


        int retval = BA_NO_RESULT;
        if(allInterfaces[delivery->toID]->isMaster()){
            allInterfaces[delivery->toID]->deliver(delivery->req);
        }
        else{
            retval = allInterfaces[delivery->toID]->access(delivery->req);
        }

        delete delivery;

        /* if the cache returns blocked we cannot deliver any more data */
        if(retval == BA_BLOCKED) break;
    }
}

void
Butterfly::setBlocked(int fromInterface){
    if(blocked) warn("blocking on a second cause");
    blocked = true;
    waitingFor = fromInterface;
    blockedAt = curTick;
    numSetBlocked++;

    blockedInterfaces.push_back(fromInterface);

    for(int i=0;i<arbitrationEvents.size();++i){
        if(arbitrationEvents[i]->scheduled()){
            arbitrationEvents[i]->deschedule();
        }
        delete arbitrationEvents[i];
    }
    arbitrationEvents.clear();

    for(int i=0;i<deliverEvents.size();++i){
        if(deliverEvents[i]->scheduled()){
            deliverEvents[i]->deschedule();
        }

        delete deliverEvents[i];
    }
    deliverEvents.clear();
}

void
Butterfly::clearBlocked(int fromInterface){
    assert(blocked);
    assert(blockedAt > -1);

    int hitIndex = -1;
    int hitCount = 0;
    for(int i=0;i<blockedInterfaces.size();i++){
        if(blockedInterfaces[i] == fromInterface){
            hitIndex = i;
            hitCount++;
        }
    }
    assert(hitCount == 1 && hitIndex > -1);
    blockedInterfaces.erase(blockedInterfaces.begin()+hitIndex);

    if(blockedInterfaces.empty()){

        blocked = false;
        numClearBlocked++;

        assert(arbitrationEvents.empty());
        if (!requestQueue.empty()){

            /* schedule new arbitration event */
            Tick firstReq = (requestQueue.front())->time;

            InterconnectArbitrationEvent* event =
                    new InterconnectArbitrationEvent(this);
            arbitrationEvents.push_back(event);

            if((firstReq + arbitrationDelay) <= curTick){
                event->schedule(curTick);
            }
            else{
                event->schedule(firstReq + arbitrationDelay);
            }
        }
        assert(deliverEvents.empty());
        if(!deliverQueue.empty()){

            /* schedule new deliver event */
            Tick firstGrant = (deliverQueue.front())->grantTime;

            InterconnectDeliverQueueEvent* event =
                    new InterconnectDeliverQueueEvent(this);
            deliverEvents.push_back(event);

            if((firstGrant + transferDelay) <= curTick){
                event->schedule(curTick);
            }
            else event->schedule(firstGrant + transferDelay);
        }

        blockedAt = -1;
    }
}

int
Butterfly::getChannelCount(){
    return hopCount * chanBetweenStages;
}

vector<int>
Butterfly::getChannelSample(){
    vector<int> copy = channelUsage;
    assert(channelUsage.size() == getChannelCount());
    for(int i=0;i<channelUsage.size();i++) channelUsage[i] = 0;
    return copy;
}

void
Butterfly::writeChannelDecriptor(std::ofstream &stream){
    stream << "Interfaces:\n";
    for(int i=0;i<allInterfaces.size();i++){
        stream << "Interface " << i
               << " (" << allInterfaces[i]->getCacheName() << "): "
               << " mapped to node id "
               << (allInterfaces[i]->isMaster() ?
                   cpuIDtoNode[interconnectIDToProcessorIDMap[i]] :
                   l2IDtoNode[interconnectIDToL2IDMap[i]] )
               << "\n";
    }

    stream << "\nChannels:\n";

    int chanSet = 0;
    for(int i=0;i<(hopCount * chanBetweenStages);i++){

        if(i != 0 && i % chanBetweenStages == 0){
            chanSet++;
            stream << "\n";
        }

        stream << "Channel ID " << i << ": In set "
                << chanSet << ", id in set "
                << (i % chanBetweenStages) << "\n";
    }
}
```

```
void
Butterfly::printChannelStatus(){

    cout << "ID:          ";
    for(int i=0;i<chanBetweenStages;i++){
        if(i<=10) cout << "     " << i << " ";
        else cout << "    " << i << " ";
    }
    cout << "\n";

    int chanGroup = 0;
    cout << "Channel Group " << chanGroup << ": ";
    for(int i=0;i<butterflyStatus.size();i++){
        cout << (butterflyStatus[i] ? " true" : "false") << " ";
        if(i != 1 && (i+1) % chanBetweenStages == 0){
            chanGroup++;
            cout << "\n";
            if(i != butterflyStatus.size()-1){
                cout << "Channel Group " << chanGroup << ": ";
            }
        }
    }
}

void
Butterfly::scheduleArbitrationEvent(Tick candidateTime){

    int found = false;
    for(int i=0;i<arbitrationEvents.size();i++){
        if(arbitrationEvents[i]->when() == candidateTime) found = true;
    }

    if(!found){
        InterconnectArbitrationEvent* event =
                new InterconnectArbitrationEvent(this);
        event->schedule(candidateTime);
        arbitrationEvents.push_back(event);
    }
}

int
Butterfly::getDestinationId(int fromID){

    if(allInterfaces[fromID]->isMaster()){

        pair<Addr,int> tmp = allInterfaces[fromID]->getTargetAddr();
        Addr targetAddr = tmp.first;
        int toInterfaceId = tmp.second;

        // The request was a null request, remove
        if(targetAddr == 0) return -1;

        // we allready know the to interface id if it's an L1 to L1 transfer
        if(toInterfaceId != -1) return toInterfaceId;

        return getTarget(targetAddr);
    }

    int retID = allInterfaces[fromID]->getTargetId();
    assert(retID != -1);
    return retID;
}


#ifndef DOXYGEN_SHOULD_SKIP_THIS

BEGIN_DECLARE_SIM_OBJECT_PARAMS(Butterfly)
    Param<int> width;
    Param<int> clock;
    Param<int> transferDelay;
    Param<int> arbitrationDelay;
    Param<int> cpu_count;
    SimObjectParam<HierParams *> hier;
    Param<int> switch_delay;
    Param<int> radix;
    Param<int> banks;
END_DECLARE_SIM_OBJECT_PARAMS(Butterfly)

BEGIN_INIT_SIM_OBJECT_PARAMS(Butterfly)
    INIT_PARAM(width, "the width of the crossbar transmission channels"),
    INIT_PARAM(clock, "butterfly clock"),
    INIT_PARAM(transferDelay, "butterfly transfer delay in CPU cycles"),
    INIT_PARAM(arbitrationDelay, "butterfly arbitration delay in CPU cycles"),
    INIT_PARAM(cpu_count, "the number of CPUs in the system"),
    INIT_PARAM_DFLT(hier,
                    "Hierarchy global variables",
                    &defaultHierParams),
    INIT_PARAM_DFLT(switch_delay,
                    "The delay of a switch in CPU cycles",
                    1),
    INIT_PARAM(radix, "The switching-degree of each switch"),
    INIT_PARAM(banks, "the number of last-level cache banks")
END_INIT_SIM_OBJECT_PARAMS(Butterfly)

CREATE_SIM_OBJECT(Butterfly)
{
    return new Butterfly(getInstanceName(),
                         width,
                         clock,
                         transferDelay,
                         arbitrationDelay,
                         cpu_count,
                         hier,
                         switch_delay,
                         radix,
                         banks);
}

REGISTER_SIM_OBJECT("Butterfly", Butterfly)

#endif //DOXYGEN_SHOULD_SKIP_THIS
```

C.1.7 Crossbar Header File

```
#ifndef __CROSSBAR_HH__
#define __CROSSBAR_HH__

#include <iostream>
#include <vector>
#include <queue>

#include "interconnect.hh"

#define DEBUG_CROSSBAR

/**
* This class implements a crossbar interconnect inspired by the crossbar used
* in IBM's Power 4 and Power 5 processors. Here, two crossbar connects all L1
* caches to all L2 banks. One crossbar is added in the L1 to L2 direction and
* the other crossbar runs in the L2 to L1 direction. L1 to L1 transfers are
* made possible by connecting all L1 caches to a shared bus.
*
* The crossbar modelled in this class differs from the IBM design in a few
* ways:
* - The IBM crossbar only has address lines in the L2 to L1 direction. This
*   implementation has address lines in both directions.
* - The data and instruction caches share transmission channels in the IBM
*   design. In this implementation, the instruction and data caches have
*   separate transmission channels.
*
* Arbitration and transfer in the crossbar are pipelined.
*
* @author Magnus Jahre
*/
class Crossbar : public Interconnect
{

    private:

        bool isFirstRequest;
        Tick nextBusFreeTime;

        std::vector<std::list<InterconnectRequest*>* > requestQueues;
        std::list<InterconnectDelivery*> deliverQueue;

        void insertIntoList(std::list<InterconnectRequest* >* inList,
                            Tick reqTime,
                            int fromID);

        bool moreRequestsAvailable();

        int getDestinationId(int fromID);

        void scheduleArbitrationEvent(Tick reqTime);

        std::vector<int> blockedInterfaces;

        bool doProfiling;
        std::vector<int> channelUseCycles;

#ifdef DEBUG_CROSSBAR
        void printRequests();
```

```
#endif //DEBUG_CROSSBAR

    public:

        /**
        * This constructor initialises a few member variables, but sends all
        * parameters to the Interconnect constructor.
        *
        * @param _name       The object name from the configuration file. This
        *                    is passed on to BaseHier and SimObject
        * @param _width      The bit width of the transmission lines in the
        *                    interconnect
        * @param _clock      The number of processor cycles in one interconnect
        *                    clock cycle.
        * @param _transDelay The end-to-end transfer delay through the
        *                    interconnect in CPU cycles
        * @param _arbDelay   The lenght of an arbitration in CPU cycles
        * @param _cpu_count  The number of processors in the system
        * @param _hier       Hierarchy parameters for BaseHier
        *
        * @see Interconnect
        */
        Crossbar(const std::string &_name,
                 int _width,
                 int _clock,
                 int _transDelay,
                 int _arbDelay,
                 int _cpu_count,
                 HierParams *_hier)
            : Interconnect(_name,
                           _width,
                           _clock,
                           _transDelay,
                           _arbDelay,
                           _cpu_count,
                           _hier){

            isFirstRequest = true;
            nextBusFreeTime = 0;
            doProfiling = false;
        }

        /**
        * This destructor deletes the request queues that are dynamically
        * allocated when the first request is recieved.
        */
        ~Crossbar(){
            for(int i=0;i<requestQueues.size();i++){
                delete requestQueues[i];
            }
        }

        /**
        * The request method administrates one queue for each transmission
        * channel. All channels are kept sorted to simplify arbitration. If an
        * arbitration event is needed, this method adds one.
        *
        * @param time   The clock cycle the method was called
        * @param fromID The interface ID of the requesting interface
        */
        void request(Tick time, int fromID);
```

```
/**
* The send method is called when an interface is granted access and
* finds the destination interface from the values in the request. Then,
* it adds the request to a delivery queue and schedules a delivery
* event if needed.
*
* @param req     The memory request to send.
* @param time    The clock cycle the method was called at.
* @param fromID  The interface ID of the sender interface.
*/
void send(MemReqPtr& req, Tick time, int fromID);

/**
* The crossbar arbitration method removes the oldest request from each
* request queue each cycle. The request must have experienced the
* specified arbitration delay to be eligible for being granted access.
* If all requests can not be granted, it attempts to schedule a new
* arbitration event.
*
* @param cycle The clock cycle the method was called.
*/
void arbitrate(Tick cycle);

/**
* This method tries to deliver as many requests as possible to its
* destination. Only, requests that have experienced the defined delay
* can be delivered. However, if an L2 bank blocks, all requests that
* are old enough might not be delivered. Since the delivery queue
* is kept sorted, the oldest requests are delivered first.
*
* Since this class uses a delivery queue, all parameters except
* cycle are discarded.
*
* @param req     Not used, must be NULL.
* @param cycle   The clock cycle the method is called.
* @param toID    Not used, must be -1.
* @param fromID  Not used, must be -1.
*/
void deliver(MemReqPtr& req, Tick cycle, int toID, int fromID);

/**
* This method is called when a L2 bank blocks. It deschedules all
* arbitration events and delivery events. Consequently, no requests
* are delivered to interfaces that are not blocked either.
*
* @param fromInterface The ID of the interface that has blocked
*/
void setBlocked(int fromInterface);


/**
* This method is called when a L2 bank becomes unblocked. If there are
* waiting requests or deliveries, new arbitration events or deliver
* events are scheduled respectively.
*
* @param fromInterface The ID of the interface that has blocked
*/
void clearBlocked(int fromInterface);

/**
```

```
        * This method returns the number of transmission channels and is used
        * by the InterconnectProfile class. In this crossbar implementation,
        * the number of channels is the number of interfaces plus the shared
        * bus.
        *
        * @return The number of transmission channels
        *
        * @see InterconnectProfile
        */
        int getChannelCount(){
            //one channel for all interfaces and one coherence bus
            channelUseCycles.resize(allInterfaces.size()+1,0);
            return allInterfaces.size() + 1;
        }

        /**
        * This method returns the number of cycles the different channels was
        * occupied since it was called last.
        *
        * @return The number of clock cycles each channel was used since last
        *         time the method was called.
        *
        * @see InterconnectProfile
        */
        std::vector<int> getChannelSample();

        /**
        * This method writes a description of the different channels to
        * the provided stream.
        *
        * @param stream The output stream to write to.
        *
        * @see InterconnectProfile
        */
        void writeChannelDecriptor(std::ofstream &stream);
};

#endif // __CROSSBAR_HH__
```

C.1.8 Crossbar Code File

```
#include "sim/builder.hh"
#include "crossbar.hh"

using namespace std;

void
Crossbar::request(Tick time, int fromID){

    requests++;

    if(isFirstRequest){
        //NOTE: This can not be initalised before alle interfaces has registered
        //      Consequently, doing it when the first request arrives is safe
        for(int i=0;i<allInterfaces.size();i++){
            requestQueues.push_back(new list<InterconnectRequest*>);
        }

        isFirstRequest = false;
    }

    insertIntoList(requestQueues[fromID], time, fromID);

    if(!blocked) scheduleArbitrationEvent(time + arbitrationDelay);

}

void
Crossbar::insertIntoList(list<InterconnectRequest* >* inList,
                         Tick reqTime,
                         int fromID){

    if(inList->empty() || inList->back()->time <= reqTime){
        // fast common case;
        inList->push_back(new InterconnectRequest(reqTime, fromID));
    }
    else{
        list<InterconnectRequest*>::iterator pos;
        for(pos = inList->begin();
            pos != inList->end();
            pos++){
                if((*pos)->time > reqTime) break;
        }
        inList->insert(pos, new InterconnectRequest(reqTime, fromID));
    }

#ifdef DEBUG_CROSSBAR
    /* Make sure that the queues are sorted */
    for(int i=0;i<requestQueues.size();i++){
        assert(isSorted(requestQueues[i]));
    }
#endif //DEBUG_CROSSBAR
}

void
Crossbar::scheduleArbitrationEvent(Tick candidateTime){

    int found = false;
    for(int i=0;i<arbitrationEvents.size();i++){
        if(arbitrationEvents[i]->when() == candidateTime) found = true;
    }

    if(!found){
        InterconnectArbitrationEvent* event =
                new InterconnectArbitrationEvent(this);
        event->schedule(candidateTime);
        arbitrationEvents.push_back(event);
    }
}

void
Crossbar::arbitrate(Tick cycle){

    assert(!blocked);

    Tick legalRequestTime = cycle - arbitrationDelay;

    /* create storage for the arbitraion process */
    vector<bool> deliverBusy(allInterfaces.size(), false);
    vector<bool> senderBusy(allInterfaces.size(), false);
    bool toOtherL1Busy = false;

    if(cycle < nextBusFreeTime) toOtherL1Busy = true;


    vector<list<InterconnectRequest*> >
            delayedRequests(allInterfaces.size());
    for(int i=0;i<delayedRequests.size();i++){
        delayedRequests[i] = list<InterconnectRequest*>();
    }

    while(moreRequestsAvailable()){

        /* start with the oldest request, regardless of which queue it is in
           Consequently, starvation is avoided */
        int smallestId = -1;
        Tick smallest = TICK_T_MAX;
        for(int i=0;i<requestQueues.size();i++){
            if(!requestQueues[i]->empty()
               && requestQueues[i]->front()->time < smallest){
                smallest = requestQueues[i]->front()->time;
                smallestId = i;
            }
        }
        assert(smallestId >= 0);

        InterconnectRequest* tempReq = requestQueues[smallestId]->front();
        requestQueues[smallestId]->pop_front();

        if(tempReq->time <= legalRequestTime){

            int toID = getDestinationId(tempReq->fromID);

            if(toID == -1){
                /* this was a null request, remove it and move on */
                delete tempReq;
                continue;
            }

            if(allInterfaces[toID]->isMaster()
                && allInterfaces[tempReq->fromID]->isMaster()){

                 if(toOtherL1Busy){
                     delayedRequests[tempReq->fromID].push_back(tempReq);
                     continue;
                 }
                 else{
                     toOtherL1Busy = true;
                     nextBusFreeTime = (cycle + arbitrationDelay);

                     /* update statistics */
                     arbitratedRequests++;
                     totalArbQueueCycles +=
                             (cycle - tempReq->time) - arbitrationDelay;
                     totalArbitrationCycles += arbitrationDelay;

                     allInterfaces[tempReq->fromID]->grantData();
                     delete tempReq;

                     continue;
                 }
             }

             if(!deliverBusy[toID] && !senderBusy[tempReq->fromID]){
                 /* request can be delivered*/

                 /* update statistics */
                 arbitratedRequests++;
                 totalArbQueueCycles +=
                         (cycle - tempReq->time) - arbitrationDelay;
                 totalArbitrationCycles += arbitrationDelay;

                 allInterfaces[tempReq->fromID]->grantData();
                 delete tempReq;


                 deliverBusy[toID] = true;
                 senderBusy[tempReq->fromID] = true;
                 continue;
             }
         }
         delayedRequests[tempReq->fromID].push_back(tempReq);
     }

     /* put not granted requests back in the request queues */
     assert(requestQueues.size() == delayedRequests.size());
     for(int i=0;i<requestQueues.size();i++){
         assert(requestQueues[i]->empty());
         (*requestQueues[i]) = delayedRequests[i];
     }

     /* check if we need to schedule new arbitration events */
     Tick nextArbTime = TICK_T_MAX;
     bool addArbEvent = false;
     for(int i=0;i<requestQueues.size();i++){
         if(!requestQueues[i]->empty()){
             InterconnectRequest* first = requestQueues[i]->front();
             Tick candidateTime = first->time + arbitrationDelay;
             if(candidateTime <= cycle){
                 if((cycle + 1) < nextArbTime){
                     addArbEvent = true;
```

```
                    nextArbTime = cycle + 1;
                }
            }
            else{
                if(candidateTime < nextArbTime){
                    addArbEvent = true;
                    nextArbTime = candidateTime;
                }
            }
        }
    }

    if(addArbEvent){
        assert(nextArbTime != TICK_T_MAX);
        scheduleArbitrationEvent(nextArbTime);
    }
}

bool
Crossbar::moreRequestsAvailable(){
    bool moreReqs = false;
    for(int i=0;i<requestQueues.size();i++){
        if(!requestQueues[i]->empty()) moreReqs = true;
    }
    return moreReqs;
}

int
Crossbar::getDestinationId(int fromID){

    if(allInterfaces[fromID]->isMaster()){

        pair<Addr,int> tmp = allInterfaces[fromID]->getTargetAddr();
        Addr targetAddr = tmp.first;
        int toInterfaceId = tmp.second;

        // The request was a null request, remove
        if(targetAddr == 0) return -1;

        // we allready know the to interface id if it's an L1 to L1 transfer
        if(toInterfaceId != -1) return toInterfaceId;

        return getTarget(targetAddr);
    }

    int retID = allInterfaces[fromID]->getTargetId();
    assert(retID != -1);
    return retID;
}

void
Crossbar::send(MemReqPtr& req, Tick time, int fromID){

    assert(!blocked);
    assert((req->size / width) <= 1);

    int toID = -1;
    bool busIsUsed = false;
    if(allInterfaces[fromID]->isMaster() && req->toInterfaceID != -1){
        busIsUsed = true;
        toID = req->toInterfaceID;
    }
    else if(allInterfaces[fromID]->isMaster()){
        toID = getTarget(req->paddr);
    }
    else{
        toID = req->fromInterfaceID;
    }

    //update profile stats
    if(doProfiling){
        if(busIsUsed){
            // the coherence bus is not pipelined
            channelUseCycles[allInterfaces.size()] += transferDelay;
        }
        else{
            // regular pipelined crossbar channels used
            // one pipeline slot is allocated in both sender
            // and recievers channel
            channelUseCycles[fromID] += 1;
            channelUseCycles[toID] += 1;
        }
    }

    deliverQueue.push_back(new InterconnectDelivery(time, fromID, toID, req));

    /* check if we need to schedule a deliver event */
    Tick deliverTime = time + transferDelay;
    bool found = false;
    for(int i=0;i<deliverEvents.size();i++){
        if(deliverEvents[i]->when() == deliverTime) found = true;
    }

    if(!found){
        InterconnectDeliverQueueEvent* event =
                new InterconnectDeliverQueueEvent(this);
        event->schedule(deliverTime);
        deliverEvents.push_back(event);
    }
}


void
Crossbar::deliver(MemReqPtr& req, Tick cycle, int toID, int fromID){
    assert(!blocked);

    assert(!req);
    assert(toID == -1);
    assert(fromID == -1);

#ifdef DEBUG_CROSSBAR
    assert(isSorted(&deliverQueue));
#endif //DEBUG_CROSSBAR

    Tick legalGrantTime = cycle - transferDelay;

    /* attempt to deliver as many requests as possible */
    /* since the queue is sorted, starvation is not possible */
    while(!deliverQueue.empty()){
        InterconnectDelivery* delivery = deliverQueue.front();

        /* check if this grant has experienced the proper delay */
        /* since the requests are sorted, we know that all other are later */
        if(delivery->grantTime > legalGrantTime) break;

        deliverQueue.pop_front();

        /* update statistics */
        sentRequests++;
        int curCpuId = delivery->req->xc->cpu->params->cpu_id;
        int queueCycles = (cycle - delivery->grantTime) - transferDelay;

        totalTransQueueCycles += queueCycles;
        totalTransferCycles += transferDelay;
        perCpuTotalTransQueueCycles[curCpuId] += queueCycles;
        perCpuTotalTransferCycles[curCpuId] += transferDelay;


        int retval = BA_NO_RESULT;
        if(allInterfaces[delivery->toID]->isMaster()){
            allInterfaces[delivery->toID]->deliver(delivery->req);
        }
        else{
            retval = allInterfaces[delivery->toID]->access(delivery->req);
        }

        delete delivery;

        /* if the cache returns blocked we cannot deliver any more data */
        if(retval == BA_BLOCKED) break;
    }
}

void
Crossbar::setBlocked(int fromInterface){

    if(blocked) warn("blocking on a second cause");
    blocked = true;
    waitingFor = fromInterface;
    blockedAt = curTick;
    numSetBlocked++;

    blockedInterfaces.push_back(fromInterface);

    for(int i=0;i<arbitrationEvents.size();++i){
        if(arbitrationEvents[i]->scheduled()){
            arbitrationEvents[i]->deschedule();
        }
        delete arbitrationEvents[i];
    }
    arbitrationEvents.clear();

    for(int i=0;i<deliverEvents.size();++i){
        if(deliverEvents[i]->scheduled()){
            deliverEvents[i]->deschedule();
        }

        delete deliverEvents[i];
    }
    deliverEvents.clear();

}
```

```
void
Crossbar::clearBlocked(int fromInterface){

    assert(blocked);
    assert(blockedAt > -1);

    int hitIndex = -1;
    int hitCount = 0;
    for(int i=0;i<blockedInterfaces.size();i++){
        if(blockedInterfaces[i] == fromInterface){
            hitIndex = i;
            hitCount++;
        }
    }
    assert(hitCount == 1 && hitIndex > -1);
    blockedInterfaces.erase(blockedInterfaces.begin()+hitIndex);

    if(blockedInterfaces.empty()){

        blocked = false;
        numClearBlocked++;

        assert(arbitrationEvents.empty());
        if(moreRequestsAvailable()){

            /* schedule new arbitration event */
            Tick firstReq = TICK_T_MAX;
            for(int i=0;i<requestQueues.size();i++){
                if(!requestQueues[i]->empty()
                   && requestQueues[i]->front()->time < firstReq){
                    firstReq = requestQueues[i]->front()->time;
                }
            }
            assert(firstReq != TICK_T_MAX);

            InterconnectArbitrationEvent* event =
                    new InterconnectArbitrationEvent(this);
            arbitrationEvents.push_back(event);

            if(firstReq <= curTick) event->schedule(curTick);
            else event->schedule(firstReq);
        }

        assert(deliverEvents.empty());
        if(!deliverQueue.empty()){

            /* schedule new deliver event */
            Tick firstGrant = (deliverQueue.front())->grantTime;

            InterconnectDeliverQueueEvent* event =
                    new InterconnectDeliverQueueEvent(this);
            deliverEvents.push_back(event);

            if(firstGrant <= curTick) event->schedule(curTick);
            else event->schedule(firstGrant);
        }
        blockedAt = -1;
    }


}
```

```
vector<int>
Crossbar::getChannelSample(){

    if(!doProfiling) doProfiling = true;

    std::vector<int> retval(channelUseCycles);

    for(int i=0;i<channelUseCycles.size();i++){
        channelUseCycles[i] = 0;
    }

    return retval;
}

void
Crossbar::writeChannelDecriptor(std::ofstream &stream){

    for(int i=0;i<allInterfaces.size();i++){
        stream << "Channel " << i << ": "
               << allInterfaces[i]->getCacheName() << "\n";
    }
    stream << "Channel " << allInterfaces.size() << ": Coherence bus\n";
}


#ifdef DEBUG_CROSSBAR

void
Crossbar::printRequests(){
    for(int i=0;i<requestQueues.size();i++){
        cout << i << ": ";
        for(list<InterconnectRequest*>::iterator j=requestQueues[i]->begin();
            j != requestQueues[i]->end();
            j++){
                InterconnectRequest* current = *j;
                cout << "("
                        << current->fromID
                        << ", "
                        << current->time
                        << ") ";
        }
        cout << "\n";
    }
    cout << "\n";
}

#endif //DEBUG_CROSSBAR

#ifndef DOXYGEN_SHOULD_SKIP_THIS

BEGIN_DECLARE_SIM_OBJECT_PARAMS(Crossbar)
    Param<int> width;
    Param<int> clock;
    Param<int> transferDelay;
    Param<int> arbitrationDelay;
    Param<int> cpu_count;
    SimObjectParam<HierParams *> hier;
END_DECLARE_SIM_OBJECT_PARAMS(Crossbar)

BEGIN_INIT_SIM_OBJECT_PARAMS(Crossbar)
```

```
    INIT_PARAM(width, "the width of the crossbar transmission channels"),
    INIT_PARAM(clock, "crossbar clock"),
    INIT_PARAM(transferDelay, "crossbar transfer delay in CPU cycles"),
    INIT_PARAM(arbitrationDelay, "crossbar arbitration delay in CPU cycles"),
    INIT_PARAM(cpu_count, "the number of CPUs in the system"),
    INIT_PARAM_DFLT(hier,
                    "Hierarchy global variables",
                    &defaultHierParams)
END_INIT_SIM_OBJECT_PARAMS(Crossbar)

CREATE_SIM_OBJECT(Crossbar)
{
    return new Crossbar(getInstanceName(),
                        width,
                        clock,
                        transferDelay,
                        arbitrationDelay,
                        cpu_count,
                        hier);
}

REGISTER_SIM_OBJECT("Crossbar", Crossbar)

#endif //DOXYGEN_SHOULD_SKIP_THIS
```

C.1.9 Ideal Interconnect Header File

```
#ifndef __IDEAL_INTERCONNECT_HH__
#define __IDEAL_INTERCONNECT_HH__

#include <iostream>
#include <vector>
#include <list>

#include "interconnect.hh"

#define DEBUG_IDEAL_INTERCONNECT

/**
* This class implements an ideal interconnect. Ideal in this context means that
* a request will experience the specified delay, but an unlimited number of
* requests will be granted access in parallel. The rationale for this choice
* it that transmission delays are mainly due to physical factors that can not
* be changed by architectural techniques.
*
* @author Magnus Jahre
*/
class IdealInterconnect : public Interconnect
{

    private:

        std::list<InterconnectRequest* > requestQueue;
        std::list<InterconnectDelivery* > grantQueue;
        std::vector<int> blockedInterfaces;

        void scheduleArbitrationEvent(Tick cycle);

#ifdef DEBUG_IDEAL_INTERCONNECT
        void printRequestQueue();
        void printGrantQueue();
#endif //DEBUG_IDEAL_INTERCONNECT

    public:

        /**
        * This constructor initialises a few member variables and passes on
        * parameters to the Interconnect constructor. The cache implementation
        * requires that the interconnect has a finite width. Consequently, a
        * width equal to the cache block size should be provided.
        *
        * @param _name       The object name from the configuration file. This
        *                    is passed on to BaseHier and SimObject
        * @param _width      The bit width of the transmission lines in the
        *                    interconnect
        * @param _clock      The number of processor cycles in one interconnect
        *                    clock cycle.
        * @param _transDelay The end-to-end transfer delay through the
        *                    interconnect in CPU cycles
        * @param _arbDelay   The lenght of an arbitration in CPU cycles
        * @param _cpu_count  The number of processors in the system
        * @param _hier       Hierarchy parameters for BaseHier
        *
        * @see Interconnect
        */
```

```
IdealInterconnect(const std::string &_name,
                  int _width,
                  int _clock,
                  int _transDelay,
                  int _arbDelay,
                  int _cpu_count,
                  HierParams *_hier)
    : Interconnect(_name,
                   _width,
                   _clock,
                   _transDelay,
                   _arbDelay,
                   _cpu_count,
                   _hier){

    if(_width <= 0){
        fatal("The idealInterconnect must have a finite width, "
              "or else the cache implementation won't work");
    }

    transferDelay = _transDelay;
    arbitrationDelay = _arbDelay;

}

/**
* Empty destructor.
*/
~IdealInterconnect(){ /* does nothing */ }

/**
* This method puts the requests in a queue according to the clock cycle
* the request was recieved. Consequently, the queue is kept sorted in
* ascending order. This simplifies arbitration.
*
* @param time    The clock cycle the method was called
* @param fromID The interface ID of the requesting interface
*/
void request(Tick time, int fromID);


/**
* The send method is called when an interface is granted access and
* finds the destination interface from the values in the request. Then,
* it adds the request to a delivery queue and schedules a delivery
* event if needed.
*
* @param req     The memory request to send.
* @param time    The clock cycle the method was called at.
* @param fromID The interface ID of the sender interface.
*/
void send(MemReqPtr& req, Tick time, int fromID);

/**
* The ideal interconnect arbitration method grants access to all
* requests that can be granted access every time it runs. A request
* can be granted if it has experienced the specified arbitration
* delay.
*
* @param cycle The clock cycle the method was called.
*/
```

```
void arbitrate(Tick cycle);

/**
* This method tries to deliver as many requests as possible to its
* destination. Only, requests that have experienced the defined delay
* can be delivered. However, if an L2 bank blocks, all requests that
* are old enough might not be delivered. Since the delivery queue
* is kept sorted, the oldest requests are delivered first.
*
* Since this class uses a delivery queue, all parameters except
* cycle are discarded.
*
* @param req      Not used, must be NULL.
* @param cycle    The clock cycle the method is called.
* @param toID     Not used, must be -1.
* @param fromID   Not used, must be -1.
*/
void deliver(MemReqPtr& req, Tick cycle, int toID, int fromID);

/**
* When this method is called, all arbitration events and delivery
* events are descheduled.
*
* @param fromInterface The interface that is blocked
*/
void setBlocked(int fromInterface);

/**
* This method is called when a L2 cache unblocks. Depending on the
* requests and deliveries waiting, a new arbitration event and
* delivery event are sheduled.
*
* @param fromInterface The interface that unblocks
*/
void clearBlocked(int fromInterface);

/**
* Since there are an unlimited number of channels in the ideal
* interconnect, this method returns -1.
*
* @return Always -1
*
* @see InterconnectProfile
*/
int getChannelCount(){
    return -1;
}

/**
* It makes no sense to get a channel sample in an ideal interconnect,
* so if this method is called it prints a fatal error message.
*
* @return An empty vector
*/
std::vector<int> getChannelSample(){
    fatal("Ideal Interconnect has no channels");
    std::vector<int> retval;
    return retval;
}

/**
```

```
        * The ideal interconnect has an unlimited number of channels.
        * Consequently, this method reports a fatal error if it is called.
        *
        * @param stream The stream that is never used
        */
        void writeChannelDecriptor(std::ofstream &stream){
            fatal("Ideal Interconnect has no channel descriptor");
        }
};

#endif // __IDEAL_INTERCONNECT_HH__
```

C.1.10 Ideal Interconnect Code File

```
#include "sim/builder.hh"
#include "ideal_interconnect.hh"

using namespace std;

void
IdealInterconnect::request(Tick time, int fromID){

    requests++;

    // keep linked list of requests sorted at all times
    // first request takes priority over later requests at same cycle
    list<InterconnectRequest*>::iterator findPos;
    for(findPos=requestQueue.begin();
        findPos!=requestQueue.end();
        findPos++){

            InterconnectRequest* tempReq = *findPos;
            if(time < tempReq->time) break;
    }

    requestQueue.insert(findPos, new InterconnectRequest(time, fromID));

#ifdef DEBUG_IDEAL_INTERCONNECT

    /* check that the queue is sorted */
    InterconnectRequest* prev = NULL;
    bool first = true;
    for(list<InterconnectRequest*>::iterator i=requestQueue.begin();
        i!=requestQueue.end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            assert(prev->time <= (*i)->time);
            prev = *i;
    }

//      printRequestQueue();
#endif //DEBUG_IDEAL_INTERCONNECT

    /* add arbitration event if we are not blocked */
    if(!blocked){
        scheduleArbitrationEvent(time + arbitrationDelay);
    }
}

void
IdealInterconnect::send(MemReqPtr& req, Tick time, int fromID){

    assert(!blocked);
    assert((req->size / width) <= 1);

    bool isFromMaster = allInterfaces[fromID]->isMaster();
    if(req->toInterfaceID != -1){
        /* cache-to-cache transfer */
        assert(req->toProcessorID != -1);
        assert(req->toInterfaceID == getInterconnectID(req->toProcessorID));
        grantQueue.push_back(new InterconnectDelivery(time,
                                                      fromID,
                                                      req->toInterfaceID,
                                                      req));
    }
    else if(isFromMaster){
        /* reciever is a slave interface */
        int recvCount = 0;
        int recvID = -1;

        for(int i=0;i<allInterfaces.size();++i){
            if(allInterfaces[i]->isMaster()) continue;

            if(allInterfaces[i]->inRange(req->paddr)){
                recvCount++;
                recvID = i;
            }
        }

        /* check for errors */
        if(recvCount > 1){
            fatal("More than one supplier for address in IdealInterconnect");
        }
        if(recvCount != 1){
            fatal("No supplier for address in IdealInterconnect");
        }
        assert(recvID >= 0);

        grantQueue.push_back(new InterconnectDelivery(time,
                                                      fromID,
                                                      recvID,
                                                      req));
    }
    else{
        /* reciever is a master interface*/
        grantQueue.push_back(new InterconnectDelivery(time,
                                                      fromID,
                                                      req->fromInterfaceID,
                                                      req));
    }

#ifdef DEBUG_IDEAL_INTERCONNECT
    /* check that the queue is sorted */
    InterconnectDelivery* prev = NULL;
    bool first = true;
    for(list<InterconnectDelivery*>::iterator i=grantQueue.begin();
        i!=grantQueue.end();
        i++){
            if(first){
                first = false;
                prev = *i;
                continue;
            }

            assert(prev->grantTime <= (*i)->grantTime);
            prev = *i;
        }
```

```
#endif //DEBUG_IDEAL_INTERCONNECT

    bool found = false;
    for(int i=0;i<deliverEvents.size();i++){
        if(deliverEvents[i]->when() == (time + transferDelay)) found = true;
    }

    if(!found){
        InterconnectDeliverQueueEvent* event =
                new InterconnectDeliverQueueEvent(this);
        event->schedule(time + transferDelay);
        deliverEvents.push_back(event);
    }

}

void
IdealInterconnect::arbitrate(Tick cycle){

    assert(!blocked);

    list<InterconnectRequest*> tempGrantQueue;

    InterconnectRequest* lastReq = requestQueue.back();
    if((lastReq->time + arbitrationDelay) <= cycle){
        //all requests can be issued
        tempGrantQueue.splice(tempGrantQueue.end(), requestQueue);
    }
    else{

        list<InterconnectRequest*>::iterator firstOldReq;
        for(firstOldReq = requestQueue.begin();
            firstOldReq != requestQueue.end();
            firstOldReq++){
                if(((*firstOldReq)->time + arbitrationDelay) > cycle) break;
        }

        tempGrantQueue.splice(tempGrantQueue.end(),
                              requestQueue,
                              requestQueue.begin(),
                              firstOldReq);
    }

    for(list<InterconnectRequest*>::iterator i = tempGrantQueue.begin();
        i != tempGrantQueue.end();
        i++){

            /* update statistics */
            arbitratedRequests++;
            totalArbQueueCycles += (cycle - (*i)->time) - arbitrationDelay;
            totalArbitrationCycles += arbitrationDelay;

            allInterfaces[(*i)->fromID]->grantData();
            delete *i;
    }

    if(!requestQueue.empty()){
        InterconnectRequest* firstReq = requestQueue.front();
        scheduleArbitrationEvent(firstReq->time + arbitrationDelay);
    }
}
```

```
void
IdealInterconnect::scheduleArbitrationEvent(Tick time){

    bool arbEventExists = false;
    for(int i=0;i<arbitrationEvents.size();++i){
        if(time == arbitrationEvents[i]->when()) arbEventExists = true;
    }

    if(!arbEventExists){
        InterconnectArbitrationEvent* event =
                new InterconnectArbitrationEvent(this);
        event->schedule(time);
        arbitrationEvents.push_back(event);
    }

}

void
IdealInterconnect::deliver(MemReqPtr& req, Tick cycle, int toID, int fromID){

    assert(!blocked);

    assert(!req);
    assert(toID == -1);
    assert(fromID == -1);

    list<InterconnectDelivery* > grantsThisCycle;
    InterconnectDelivery* lastReq = grantQueue.back();
    if((lastReq->grantTime + transferDelay) <= cycle){
        //all requests can be issued
        grantsThisCycle.splice(grantsThisCycle.end(), grantQueue);
    }
    else{

        list<InterconnectDelivery*>::iterator firstOldReq;
        for(firstOldReq = grantQueue.begin();
            firstOldReq != grantQueue.end();
            firstOldReq++){
                if(((*firstOldReq)->grantTime + transferDelay) > cycle) break;
            }

            grantsThisCycle.splice(grantsThisCycle.end(),
                                   grantQueue,
                                   grantQueue.begin(),
                                   firstOldReq);
    }


    while(!grantsThisCycle.empty()){
        InterconnectDelivery* tempGrant = grantsThisCycle.front();
        grantsThisCycle.pop_front();

        /* update statistics */
        sentRequests++;
        int queueCycles = (cycle - tempGrant->grantTime) - transferDelay;
        int curCpuId = tempGrant->req->xc->cpu->params->cpu_id;

        totalTransQueueCycles += queueCycles;
        totalTransferCycles += transferDelay;
        perCpuTotalTransQueueCycles[curCpuId] += queueCycles;
        perCpuTotalTransferCycles[curCpuId] += transferDelay;

        int retval = BA_NO_RESULT;
        if(allInterfaces[tempGrant->toID]->isMaster()){
            allInterfaces[tempGrant->toID]->deliver(tempGrant->req);
        }
        else{
            retval = allInterfaces[tempGrant->toID]->access(tempGrant->req);
        }

        /* this delivery got through, free memory */
        delete tempGrant;

        /* if the cache returns blocked we cannot deliver any more data */
        if(retval == BA_BLOCKED) break;
    }

    if(!grantsThisCycle.empty()){
        grantQueue.splice(grantQueue.begin(),
                          grantsThisCycle,
                          grantsThisCycle.begin(),
                          grantsThisCycle.end());
    }
}

void
IdealInterconnect::setBlocked(int fromInterface){

    if(blocked) warn("blocking on a second cause");
    blocked = true;
    waitingFor = fromInterface;
    blockedAt = curTick;
    numSetBlocked++;

    blockedInterfaces.push_back(fromInterface);

    for(int i=0;i<arbitrationEvents.size();++i){
        if(arbitrationEvents[i]->scheduled()){
            arbitrationEvents[i]->deschedule();
        }
        delete arbitrationEvents[i];
    }
    arbitrationEvents.clear();

    for(int i=0;i<deliverEvents.size();++i){
        if(deliverEvents[i]->scheduled()){
            deliverEvents[i]->deschedule();
        }

        delete deliverEvents[i];
    }
    deliverEvents.clear();

}

void
IdealInterconnect::clearBlocked(int fromInterface){

    assert(blocked);
    assert(blockedAt > -1);
    int hitIndex = -1;
    int hitCount = 0;
    for(int i=0;i<blockedInterfaces.size();i++){
        if(blockedInterfaces[i] == fromInterface){
            hitIndex = i;
            hitCount++;
        }
    }
    assert(hitCount == 1 && hitIndex > -1);
    blockedInterfaces.erase(blockedInterfaces.begin()+hitIndex);

    if(blockedInterfaces.empty()){

        blocked = false;
        numClearBlocked++;

        assert(arbitrationEvents.empty());
        if(!requestQueue.empty()){

            /* schedule new arbitration event */
            Tick firstReq = (requestQueue.front())->time;

            InterconnectArbitrationEvent* event =
                    new InterconnectArbitrationEvent(this);
            arbitrationEvents.push_back(event);

            if((firstReq + arbitrationDelay) <= curTick){
                event->schedule(curTick);
            }
            else{
                event->schedule(firstReq + arbitrationDelay);
            }
        }

        assert(deliverEvents.empty());
        if(!grantQueue.empty()){

            /* schedule new deliver event */
            Tick firstGrant = (grantQueue.front())->grantTime;

            InterconnectDeliverQueueEvent* event =
                    new InterconnectDeliverQueueEvent(this);
            deliverEvents.push_back(event);

            if((firstGrant + transferDelay) <= curTick){
                event->schedule(curTick);
            }
            else{
                event->schedule(firstGrant + transferDelay);
            }
        }

        blockedAt = -1;
    }
}

#ifdef DEBUG_IDEAL_INTERCONNECT

void
IdealInterconnect::printRequestQueue(){
    cout << "ReqQueue: ";
    for(list<InterconnectRequest*>::iterator i = requestQueue.begin();
        i != requestQueue.end();
        i++){
            cout << "(" << (*i)->time << ", " << (*i)->fromID << ") ";
    }
    cout << "\n";
}

void
IdealInterconnect::printGrantQueue(){
    cout << "GrantQueue: ";
    for(list<InterconnectDelivery*>::iterator i = grantQueue.begin();
        i != grantQueue.end();
        i++){
            cout << "("
                    << (*i)->grantTime
                    << ", "
                    << (*i)->fromID
                    << ", "
                    << (*i)->toID
                    << ") ";
    }
    cout << "\n";
}

#endif //DEBUG_IDEAL_INTERCONNECT

#ifndef DOXYGEN_SHOULD_SKIP_THIS

BEGIN_DECLARE_SIM_OBJECT_PARAMS(IdealInterconnect)
    Param<int> width;
    Param<int> clock;
    Param<int> transferDelay;
    Param<int> arbitrationDelay;
    Param<int> cpu_count;
    SimObjectParam<HierParams *> hier;
END_DECLARE_SIM_OBJECT_PARAMS(IdealInterconnect)

BEGIN_INIT_SIM_OBJECT_PARAMS(IdealInterconnect)
    INIT_PARAM(width, "ideal interconnect width, set this to the cache line "
                      "width"),
    INIT_PARAM(clock, "ideal interconnect clock"),
    INIT_PARAM(transferDelay, "ideal interconnect transfer delay in CPU "
                              "cycles"),
    INIT_PARAM(arbitrationDelay, "ideal interconnect arbitration delay in CPU "
                                 "cycles"),
    INIT_PARAM(cpu_count, "the number of CPUs in the system"),
    INIT_PARAM_DFLT(hier,
                    "Hierarchy global variables",
                    &defaultHierParams)
END_INIT_SIM_OBJECT_PARAMS(IdealInterconnect)

CREATE_SIM_OBJECT(IdealInterconnect)
{
    return new IdealInterconnect(getInstanceName(),
                                 width,
                                 clock,
                                 transferDelay,
                                 arbitrationDelay,
                                 cpu_count,
                                 hier);
```

```
}

REGISTER_SIM_OBJECT("IdealInterconnect", IdealInterconnect)

#endif //DOXYGEN_SHOULD_SKIP_THIS
```

C.1.11 Interconnect Interface Header File

```
#ifndef __INTERCONNECT_INTERFACE_HH__
#define __INTERCONNECT_INTERFACE_HH__

#include <iostream>

#include "base/range.hh"
#include "targetarch/isa_traits.hh" // for Addr
#include "mem/bus/base_interface.hh"

#include "interconnect.hh"

class Interconnect;

/**
* This class glues the interconnect extensions together with the rest of the M5
* memory system.
*
* This class is based on the bus_interface.hh class in standard M5 but has been
* rewritten from scratch. Consequently, a number of methods that are not used
* are not implemented.
*
* @author Magnus Jahre
*/
class InterconnectInterface : public BaseInterface
{

    protected:
        int interfaceID;
        Interconnect* thisInterconnect;
        bool trace_on;

        int dataSends;
        int instSends;
        int coherenceSends;
        int totalSends;
        bool doProfiling;

    public:

        /**
        * This constructor creates an interconnect interface and initialises
        * a few member variables.
        *
        * @param _interconnect A pointer to the interface this class interfaces
        *                      to.
        * @param name          The name of the class from the config file
        * @param hier          Hierarchy parameters for BaseHier
        */
        InterconnectInterface(Interconnect* _interconnect,
                              const std::string &name,
                              HierParams *hier)
            : BaseInterface(name, hier)
        {
            blocked = false;
            thisInterconnect = _interconnect;
            trace_on = false;

            dataSends = 0;
            instSends = 0;
            coherenceSends = 0;
            totalSends = 0;
            doProfiling = false;
        }

        /**
        * Mark this interface as blocked.
        */
        void setBlocked();

        /**
        * Mark this interface as unblocked.
        */
        void clearBlocked();

        /**
        * Access the connected memory and make it perform a given request.
        *
        * @param req The request to perform.
        *
        * @return The result of the access.
        */
        virtual MemAccessResult access(MemReqPtr &req) = 0;

        /**
        * Request access to the interconnect.
        *
        * @param time The time to request the bus.
        */
        virtual void request(Tick time) = 0;

        /**
        * Respond to the given request at the given time.
        *
        * @param req The request being responded to.
        * @param time The time the response is ready.
        */
        virtual void respond(MemReqPtr &req, Tick time) = 0;

        /**
        * This method must be implemented to fit into the the rest of the M5
        * memory system. In this implemetation it is never used and issues
        * a fatal error message if it is called.
        *
        * @return Always false
        */
        bool grantAddr(){
            fatal("CrossbarInterface grantAddr() method not implemented\n");
            return false;
        }

        /**
        * This method is used when an interface is granted access to the bus.
        *
        * @return True if another request is outstanding.
        */
        virtual bool grantData() = 0;

        /**
        * The interconnects does not support snooping, so this method issues a
```

```
* fatal error message if it is called.
*
* @param req Not used
*/
void snoop(MemReqPtr &req){
    fatal("CrossbarInterface snoop not implemented");
}

/**
* The interconnects does not support snooping, so this method issues a
* fatal error message if it is called.
*
* @param req Not used
*/
void snoopResponse(MemReqPtr &req){
    fatal("CrossbarInterface snoopResponse not implemented");
}

/**
* The interconnects does not support snooping, so this method issues a
* fatal error message if it is called.
*
* @param req Not used
*/
void snoopResponseCall(MemReqPtr &req){
    fatal("CrossbarInterface snoopResponse not implemented");
}

/**
* This method is never called with the configurations used in this
* work. Consequently, it is not used.
*
* @param req     Not used.
* @param update Not used.
*
* @return Nothing
*/
Tick sendProbe(MemReqPtr &req, bool update){
    fatal("CrossbarSlave sendProbe() method not implemented");
    return -1;
}

/**
* This method is never called with the configurations used in this
* work. Consequently, it is not used.
*
* @param req     Not used.
* @param update Not used.
*
* @return Nothing
*/
Tick probe(MemReqPtr &req, bool update){
    fatal("CrossbarSlave probe() method not implemented");
    return -1;
}

/**
* This method is never called in the configuration used in this work
* and is not implemented.
*
* @param range_list Not used
```

```
*/
void collectRanges(std::list<Range<Addr> > &range_list){
    fatal("CrossbarSlave collectRanges() method not implemented");
}

/**
* Adds the address ranges of this interface to the provided list.
*
* @param range_list The list of ranges.
*/
void getRange(std::list<Range<Addr> > &range_list);

/**
* Notify this interface of a range change in the interconnect.
*/
void rangeChange();

/**
* Set the address ranges of this interface to the list provided. This
* function removes any existing ranges.
*
* @param range_list List of addr ranges to add.
*/
void setAddrRange(std::list<Range<Addr> > &range_list);

/**
* Add an address range for this interface.
*
* @param range The addres range to add.
*/
void addAddrRange(const Range<Addr> &range);

/**
* This method returns the number of requests sent since the last time
* it was called. Furthermore, it divides the sends into data sends,
* instruction sends and coherence sends as well as providing a grand
* total.
*
* @param dataSends  A pointer a memory location where the number of
*                   data sends can be stored
* @param instSends  A pointer a memory location where the number of
*                   instruction sends can be stored
* @param instSends  A pointer a memory location where the number of
*                   coherence sends can be stored
* @param totalSends A pointer a memory location where the total number
*                   of sends can be stored
*
* @see InterconnectProfile
*/
void getSendSample(int* dataSends,
                   int* instSends,
                   int* coherenceSends,
                   int* totalSends);

/**
* This method ensures that the profile values are updated consistently
* from all interfaces. It is called by the subclasses.
*
* @param req The current memory request
*
* @see InterconnectProfile
```

```
        */
        void updateProfileValues(MemReqPtr &req);

        /**
        * This method delivers a reponse to the interconnect. In a slave
        * interconnect, this sends the request over the interconnect. In a
        * master interconnect, the request is delivered to the cache.
        *
        * @param req The request that will be delivered
        */
        virtual void deliver(MemReqPtr &req) = 0;

        /**
        * The interconnects often need to distinguish between a master and a
        * slave interface in an efficient manner. This method enables this.
        *
        * @return True if the interface is a master interface
        */
        virtual bool isMaster() = 0;

        /**
        * This method accesses the cache and finds out which interface the
        * next request should be sent to. Then, it returns a pair of the
        * address and the destination interface.
        *
        * Note that the destination interface might be -1. In this case, the
        * interconnect must find the destination itself by inspecting the
        * destination address.
        *
        * @return The address and the destination interface. If the destination
        *         is -1, the interconnect must derive the destination based on
        *         the requested address.
        */
        virtual std::pair<Addr, int> getTargetAddr() = 0;

        /**
        * This method returns the ID of the destination interface of the
        * request at the front of the request queue in a slave interface.
        *
        * @return The destination of the next request to be sent from a slave
        *         interface.
        */
        virtual int getTargetId() = 0;

        /**
        * Convenience method that returns the name of the cache associated with
        * a given interface.
        *
        * @return The name of the cache associated with a given interface.
        */
        virtual std::string getCacheName() = 0;

};

#endif // INTERCONNECT_INTERFACE_HH__
```

C.1.12 Interconnect Interface Code File

```
#include <iostream>
#include <vector>

#include "interconnect_interface.hh"

using namespace std;

void
InterconnectInterface::setBlocked(){
    if(!blocked){
        blocked = true;
        thisInterconnect->setBlocked(interfaceID);
    }
}

void
InterconnectInterface::clearBlocked(){
    if(blocked){
        blocked = false;
        thisInterconnect->clearBlocked(interfaceID);
    }
}

void
InterconnectInterface::getRange(std::list<Range<Addr> > &range_list)
{
    for (int i = 0; i < ranges.size(); ++i) {
        range_list.push_back(ranges[i]);
    }
}

void
InterconnectInterface::rangeChange(){
    thisInterconnect->rangeChange();
}


void
InterconnectInterface::setAddrRange(list<Range<Addr> > &range_list){
    ranges.clear();
    while (!range_list.empty()) {
        ranges.push_back(range_list.front());
        range_list.pop_front();
    }
    rangeChange();
}

void
InterconnectInterface::addAddrRange(const Range<Addr> &range){
    ranges.push_back(range);
    rangeChange();
}

void
InterconnectInterface::getSendSample(int* data,
                                     int* inst,
                                     int* coherence,
                                     int* total){
```

```
    // start profiling if this is the first call to this function
    if(!doProfiling) doProfiling = true;

    // return the sampled values
    *data = dataSends;
    *inst = instSends;
    *coherence = coherenceSends;
    *total = totalSends;

    // reset counters
    dataSends = 0;
    instSends = 0;
    coherenceSends = 0;
    totalSends = 0;
}

void
InterconnectInterface::updateProfileValues(MemReqPtr &req){

    if(doProfiling){
        if(req->cmd.isDirectoryMessage()){
            coherenceSends++;
        }
        else if(req->readOnlyCache){
            instSends++;
        }
        else{
            dataSends++;
        }
        totalSends++;
    }
}
```

C.1.13 Interconnect Master Interface Header File

```
#ifndef __INTERCONNECT_MASTER_HH__
#define __INTERCONNECT_MASTER_HH__

#include <iostream>

#include "base/range.hh"
#include "targetarch/isa_traits.hh" // for Addr
#include "mem/bus/base_interface.hh"

#include "interconnect_interface.hh"
#include "interconnect.hh"

class Interconnect;

/**
* This class implements a master interface as needed by the M5 cache
* implementation. In the terminiology of M5, master is synonymous with 'on the
* processor side of an interconnect'.
*
* This class is based on the master_interface files in the original M5 memory
* system but has been rewritten from scratch.
*
* @author Magnus Jahre
*/
template <class MemType>
class InterconnectMaster : public InterconnectInterface
{
    private:

        MemType* thisCache;

        Addr currentAddr;
        int currentToCpuId;
        bool currentValsValid;

        //debug
        std::vector<std::pair<Addr, Tick>* > outstandingRequestAddrs;

    public:
        /**
        * This constructor creates a master interface and register it with the
        * associated interconnect. In the terminiology of M5, master is
        * synonymous with 'on the processor side of an interconnect'
        *
        * @param name         The name of the interface from the config file
        * @param interconnect A pointer to the associated interconnect
        * @param cache        A pointer to the associated cache
        * @param hier         Hierarchy parameters for BaseHier
        */
        InterconnectMaster(const std::string &name,
                           Interconnect* interconnect,
                           MemType* cache,
                           HierParams *hier);

        /**
        * Access the connect memory to perform the given request.
        *
        * @param req The request to perform.
```

```
*
* @return The result of the access.
*/
MemAccessResult access(MemReqPtr &req);

/**
* Request access to the interconnect at the given time.
*
* @param time The time to request the bus.
*/
void request(Tick time);

/**
* Responses are carried out through the deliver method in the master
* interface. Consequently, this method exits with an error message if
* it is called.
*
* @param req  Not used.
* @param time Not used.
*/
void respond(MemReqPtr &req, Tick time){
    fatal("CrossbarMaster respond method not implemented");
}

/**
* When the interface is granted access to the interconnect, this method
* is called. It retrieves the request with the highest priority from
* the cache and provides it to the send method in the interconnect.
*
* @return True if another request is outstanding.
*/
bool grantData();

/**
* This method delivers the request to the associated cache.
*
* @param req The memory request to deliver
*/
void deliver(MemReqPtr &req);

/**
* Convenience method that identifies this interface as a master
* interface.
*
* @return True, since this is a master interface
*/
bool isMaster(){
    return true;
}

/**
* This method accesses the cache and finds out which interface the
* next request should be sent to. Then, it returns a pair of the
* address and the destination interface.
*
* Note that the destination interface might be −1. In this case, the
* interconnect must find the destination itself by inspecting the
* destination address.
*
* @return The address and the destination interface. If the destination
*         is −1, the interconnect must derive the destination based on
```

```
        *          the requested address.
        */
        std::pair<Addr, int> getTargetAddr();

        /**
        * This method is only valid for slave interfaces and produces a fatal
        * error message if it is called.
        *
        * @return Nothing
        */
        int getTargetId(){
            fatal("getTargetId() not valid for a MasterInterface");
            return -1;
        }

        /**
        * Convenience method that returns the name of the associated cache.
        *
        * @return The name of associated cache
        */
        std::string getCacheName(){
            return thisCache->name();
        }

};

#endif // __INTERCONNECT_MASTER_HH__
```

C.1.14 Interconnect Master Interface Code File

```
#include <iostream>
#include <vector>


#include "interconnect.hh"
#include "interconnect_master.hh"


using namespace std;

template<class MemType>
InterconnectMaster<MemType>::InterconnectMaster(const string &name,
                                                Interconnect* interconnect,
                                                MemType* cache,
                                                HierParams *hier)
    : InterconnectInterface(interconnect, name, hier)
{
    thisCache = cache;

    interfaceID = thisInterconnect->registerInterface(this,
                                                      false,
                                                      cache->getProcessorID());
    thisCache->setInterfaceID(interfaceID);

    currentAddr = 0;
    currentToCpuId = -1;
    currentValsValid = false;

    if(trace_on) cout << "InterconnectMaster with id "
                      << interfaceID << " created\n";
}

template<class MemType>
MemAccessResult
InterconnectMaster<MemType>::access(MemReqPtr &req){

    // NOTE: copied from MasterInterface
    int satisfied_before = req->flags & SATISFIED;
    // Cache Coherence call goes here
    //mem->snoop(req);
    if (satisfied_before != (req->flags & SATISFIED)) {
        return BA_SUCCESS;
    }

    return BA_NO_RESULT;
}

template<class MemType>
void
InterconnectMaster<MemType>::deliver(MemReqPtr &req){

    if(!req->cmd.isDirectoryMessage()){
        pair<Addr, Tick>* hitPair = NULL;
        int hitIndex = -1;
        for(int i=0;i<outstandingRequestAddrs.size();++i){
            pair<Addr, Tick>* tmpPair = outstandingRequestAddrs[i];
            if(req->paddr == tmpPair->first){
                hitIndex = i;
                hitPair = tmpPair;
            }
        }

        /* check if this actually was an answer to something we requested */
        assert(hitIndex >= 0);
        outstandingRequestAddrs.erase(outstandingRequestAddrs.begin()+hitIndex);
        delete hitPair;
    }

    if(trace_on){
        cout << "Master "<< interfaceID <<" is waiting for: ";
        for(int i=0;i<outstandingRequestAddrs.size();++i){
            cout << "(" << outstandingRequestAddrs[i]->first
                 << ", " << outstandingRequestAddrs[i]->second << ") ";
        }
        cout << "at tick " << curTick << "\n";
    }

    if(trace_on) cout << "TRACE: MASTER_RESPONSE id " << interfaceID
                      << " addr " << req->paddr << " at " << curTick << "\n";
    thisCache->handleResponse(req);
}

template<class MemType>
void
InterconnectMaster<MemType>::request(Tick time){

    if(trace_on) cout << "TRACE: MASTER_REQUEST id " << interfaceID << " at "
                      << curTick << "\n";

    thisInterconnect->request(time, interfaceID);
}

template<class MemType>
bool
InterconnectMaster<MemType>::grantData(){

    MemReqPtr req = thisCache->getMemReq();

    if(!req){
        thisInterconnect->incNullRequests();
        return false;
    }

    if(trace_on) cout << "TRACE: MASTER_SEND " << req->cmd.toString()
                      << " from id " << interfaceID << " addr " << req->paddr
                      << " at " << curTick << "\n";

    if(!req->cmd.isNoResponse()){
        if(!req->cmd.isDirectoryMessage()){
            outstandingRequestAddrs.push_back(
                    new pair<Addr, Tick>(req->paddr, curTick));
        }
    }

    req->fromInterfaceID = interfaceID;
    req->firstSendTime = curTick;
    req->readOnlyCache = thisCache->isInstructionCache();
    req->toInterfaceID =
            thisInterconnect->getInterconnectID(req->toProcessorID);
    // make sure destination was set properly
    if(req->toProcessorID != -1) assert(req->toInterfaceID != -1);

    if(currentValsValid){
        assert(currentAddr == req->paddr);
        assert(currentToCpuId == req->toProcessorID);
        currentValsValid = false;
    }

    // Update send profile
    updateProfileValues(req);

    //Currently sends can't fail, so all reqs will be a success
    thisCache->sendResult(req, true);
    thisInterconnect->send(req, curTick, interfaceID);

    return thisCache->doMasterRequest();
}

template<class MemType>
pair<Addr, int>
InterconnectMaster<MemType>::getTargetAddr(){
    MemReqPtr currentRequest = thisCache->getMemReq();

    if(!currentRequest) return pair<Addr,int>(0,-2);
    assert(currentRequest->paddr != 0);

    int toInterface =
            thisInterconnect->getInterconnectID(currentRequest->toProcessorID);
    if(currentRequest->toProcessorID != -1) assert(toInterface != -1);

    currentAddr = currentRequest->paddr;
    currentToCpuId = toInterface;
    currentValsValid = true;

    return pair<Addr,int>(currentRequest->paddr, toInterface);
}
```

C.1.15 Interconnect Slave Interface Header File

```
#ifndef __INTERCONNECT_SLAVE_HH__
#define __INTERCONNECT_SLAVE_HH__

#include <iostream>

#include "base/range.hh"
#include "targetarch/isa_traits.hh" // for Addr
#include "mem/bus/base_interface.hh"

#include "interconnect_interface.hh"
#include "interconnect.hh"

class Interconnect;

/**
* This class implements a slave interface as needed by the M5 cache
* implementation. In the terminiology of M5, slave is synonymous with 'on the
* memory side of an interconnect'
*
* This class is based on the slave_interface files in the original M5 memory
* system but has been rewritten from scratch.
*
* @author Magnus Jahre
*/
template <class MemType>
class InterconnectSlave : public InterconnectInterface
{
    private:

        /**
        * Convenience class for storing cache responses from they are requested
        * until they are granted access.
        *
        * @author Magnus Jahre
        */
        class InterconnectResponse{

            public:
                MemReqPtr req;
                Tick time;

                /**
                * Stores the request and the time it was requested in this
                * object.
                *
                * @param _req  The memory request waiting for access to the
                *              interconnect
                * @param _time The clock cycle the response was recieved
                *
                * @author Magnus Jahre
                */
                InterconnectResponse(MemReqPtr &_req, Tick _time)
                {
                    req = _req;
                    time = _time;
                }
        };
        MemType* thisCache;

        std::vector<InterconnectResponse* > responseQueue;

    public:
        /**
        * This constructor creates an interconnect slave interface and
        * registers it with the provided interconnect.
        *
        * @param name         The name from the config file
        * @param interconnect A pointer to the interconnect
        * @param cache        A pointer to the cache
        * @param hier         Hierarchy parameters for BaseHier
        */
        InterconnectSlave(const std::string &name,
                          Interconnect* interconnect,
                          MemType* cache,
                          HierParams *hier);

        /**
        * Access the connect memory to perform the given request.
        *
        * @param req The request to perform.
        *
        * @return The result of the access.
        */
        MemAccessResult access(MemReqPtr &req);

        /**
        * The request method is not needed in slave interfaces and is not
        * implemented. If it is called, it issues a fatal error message.
        *
        * @param time Not used.
        */
        void request(Tick time){
            fatal("InterconnectSlave request(Tick time) not implemented");
        }


        /**
        * The respond method is called when the connected cache responds to an
        * access. Then, an InterconnectResponse object is allocated and put
        * into a queue. Furthermore, access to the interconnect is requested.
        *
        * @param req  The request being responded to.
        * @param time The time the response is ready.
        *
        * @see InterconnectResponse
        */
        void respond(MemReqPtr &req, Tick time);

        /**
        * Called when this interface is granted access to the interconnect.
        *
        * @return True if another request is outstanding.
        */
        bool grantData();

        /**
        * The deliver method is not used in a slave interface and issues a
        * fatal error message if it is called.
```

```
*
* @param req Not used.
*/
void deliver(MemReqPtr &req){
    fatal("InterconnectSlave deliver() not implemented");
}

/**
* Since this is a slave interface, this method always returns false.
*
* @return False, since this is a slave interface.
*/
bool isMaster(){
    return false;
}

/**
* This method has no relevance for a slave interface and issues a
* fatal error message if it is called.
*
* @return Nothing.
*/
std::pair<Addr, int> getTargetAddr(){
    fatal("getTargetAddr() not valid for slave interfaces");
    return std::pair<Addr, int>(-42,-42);
}

/**
* This method is used to find the destination interface of the request
* at the head of response queue. This information is stored in the
* request, so this method simply accesses this information.
*
* @return The destination interface.
*/
int getTargetId(){
    assert(!responseQueue.empty());
    return responseQueue.front()->req->fromInterfaceID;
}

/**
* Retrieves the name of the associated cache.
*
* @return The name of the associated cache.
*/
std::string getCacheName(){
    return thisCache->name();
}

/**
* This method overloaded here to implement modulo bank addressing.
* The default in M5 is that each bank is responsible for a contigous
* part of the address space. This functionality is retained, but in
* addition it implements modulo addressing. In this case the address
* modulo the number of banks is used to decide which bank should
* service a given request.
*
* A configuration option in the cache selects which bank addressing
* type should be used.
*
* @param addr The address to be checked
*
```

```
        * @return True if this interface is responsible for this address.
        */
        virtual bool inRange(Addr addr);

};

#endif // __INTERCONNECT_SLAVE_HH__
```

C.1.16 Interconnect Slave Interface Code File

```cpp
#include <iostream>
#include <vector>

#include "interconnect.hh"
#include "interconnect_slave.hh"

using namespace std;

template<class MemType>
InterconnectSlave<MemType>::InterconnectSlave(const string &name,
                                               Interconnect* interconnect,
                                               MemType* cache,
                                               HierParams *hier)
    : InterconnectInterface(interconnect, name, hier)
{
    thisCache = cache;

    interfaceID = thisInterconnect->registerInterface(this,
                                                       true,
                                                       cache->getProcessorID());
    thisCache->setInterfaceID(interfaceID);

    if(trace_on) cout << "InterconnectSlave with id " << interfaceID
                      << " created\n";

}

template<class MemType>
MemAccessResult
InterconnectSlave<MemType>::access(MemReqPtr &req){

    bool already_satisfied = req->isSatisfied();
    if (already_satisfied && !req->cmd.isDirectoryMessage()) {
        warn("Request is allready satisfied (InterconnectSlave: access())");
        return BA_NO_RESULT;
    }

    if (this->inRange(req->paddr)) {
        assert(!blocked);

        if(trace_on) cout << "TRACE: SLAVE_ACCESS from id "
                          << req->fromInterfaceID << " addr " << req->paddr
                          << " at " << curTick << "\n";

        thisCache->access(req);

        assert(!this->isBlocked() || thisCache->isBlocked());

        if (this->isBlocked()) {
            //Out of MSHRS, now we block
            return BA_BLOCKED;
        } else {
            // This transaction went through ok
            return BA_SUCCESS;
        }
    }

    return BA_NO_RESULT;
```

```
}

template<class MemType>
void
InterconnectSlave<MemType>::respond(MemReqPtr &req, Tick time){

    if (!req->cmd.isNoResponse()) {
        if(trace_on) cout << "TRACE: SLAVE_RESPONSE " << req->cmd.toString()
                          << " from id " << req->fromInterfaceID
                          << " addr " << req->paddr
                          << " at " << curTick << "\n";

        // handle directory requests
        if(req->toProcessorID != -1){
            // the sender interface recieves slave responses
            req->fromInterfaceID =
                    thisInterconnect->getInterconnectID(req->toProcessorID);
            assert(req->fromInterfaceID != -1);
        }

        responseQueue.push_back(new InterconnectResponse(req, time));
        thisInterconnect->request(time, interfaceID);
    }
}


template<class MemType>
bool
InterconnectSlave<MemType>::grantData(){

    assert(responseQueue.size() > 0);
    InterconnectResponse* response = responseQueue.front();
    responseQueue.erase(responseQueue.begin());

    MemReqPtr req = response->req;

    // Update send profile
    updateProfileValues(req);

    thisInterconnect->send(req, curTick, interfaceID);

    delete response;

    return !responseQueue.empty();

}

template<class MemType>
bool
InterconnectSlave<MemType>::inRange(Addr addr)
{
    assert(thisCache != NULL);

    if(thisCache->isModuloAddressedBank()){

        int bankID = thisCache->getBankID();
        int bankCount = thisCache->getBankCount();

        int localBlkSize = thisCache->getBlockSize();
        int bitCnt = 1;
        assert(localBlkSize != 0);
        while((localBlkSize >>= 1) != 1) bitCnt++;

        assert(bankID != -1);
        assert(bankCount != -1);

        Addr effectiveAddr = addr >> bitCnt;

        if((effectiveAddr % bankCount) == bankID) return true;
        return false;
    }
    else{
        for (int i = 0; i < ranges.size(); ++i) {
            if (addr == ranges[i]) {
                return true;
            }
        }
        return false;
    }
}
```

C.1.17 Interconnect Profiler Header File

```
#ifndef __INTERCONNECT_PROFILE_HH__
#define __INTERCONNECT_PROFILE_HH__

#include "sim/sim_object.hh"
#include "sim/eventq.hh"
#include "interconnect.hh"

/** The number of clock cycles between each profile event */
#define RESOLUTION 250000

class Interconnect;
class InterconnectProfileEvent;

/** Differentiates send profile events from channel profile events */
typedef enum {SEND, CHANNEL} INTERCONNECT_PROFILE_TYPE;

/**
* This class implements a simple profiler for interconnects. It retrieves some
* statistics from the interconnect object and writes it to a file at a regular
* interval. This interval is decided by the RESOLUTION definition at the
* begining of this file.
*
* Currently, this class provides two forms of profiles:
* - Firstly, it profiles the number of sends injected into the interconnect.
*   This gives a measure of the external pressure on the interconnect as a
*   function of time.
* - Secondly, it prints the utilisation of each channel inside the
*   interconnect at regular interval. This feature can be used to identify
*   bottlenecks inside a given interconnect.
*
* @author Magnus Jahre
*/
class InterconnectProfile : public SimObject
{
    private:
        Interconnect* interconnect;
        bool traceSends;
        bool traceChannelUtil;
        Tick startTick;

        InterconnectProfileEvent* sendEvent;
        InterconnectProfileEvent* channelEvent;

        std::string sendFileName;
        std::string channelFileName;
        std::string channelExplFileName;

    public:
        /**
        * This constructor creates the interconnect profiler and schedules the
        * first profile event.
        *
        * @param _name              The name from the config file
        * @param _traceSends        Wheter or not sends should be traced or not
        * @param _traceChannelUtil Wheter or not sends should be traced or not
        * @param _startTick         The clock cycle the profiling will start
        * @param _interconnect      A pointer to the interconnect that will be
        *                           profiled
        */
        InterconnectProfile(const std::string &_name,
                            bool _traceSends,
                            bool _traceChannelUtil,
                            Tick _startTick,
                            Interconnect* _interconnect);

        /**
        * This method initialises the send profile file.
        */
        void initSendFile();

        /**
        * This method is called when a profile event i serviced. It retrieves
        * the current profile values from the interconnect and writes them to
        * the send profile file.
        */
        void writeSendEntry();

        /**
        * This method initialises the channel file. It checks if the
        * interconnect supports channel profiling and initalises the file if it
        * does. The reason for this check is that channel profiling makes no
        * sense with an ideal interconnect.
        *
        * @return True, if the interconnect supports channel profiling.
        */
        bool initChannelFile();

        /**
        * This method retrieves the updated channel utilisation from the
        * interconnect and writes these values to the channel utilisation file.
        */
        void writeChannelEntry();
};

/**
* This class implements a profile event. It schedules itself at regular
* intervals when it has been started and calls the appropriate methods for the
* statistics to be written to the profile files.
*
* @author Magnus Jahre
*/
class InterconnectProfileEvent : public Event
{

    public:

        InterconnectProfile* profiler;
        INTERCONNECT_PROFILE_TYPE traceType;

        /**
        * Initalises the member variables.
        *
        * @param _profiler A pointer to the associated profiler object
        * @param _type     The type of profiler
        */
        InterconnectProfileEvent(InterconnectProfile* _profiler,
                                 INTERCONNECT_PROFILE_TYPE _type)
            : Event(&mainEventQueue)
        {
            profiler = _profiler;
            traceType = _type;
        }

        /**
        * This method is called when the event is serviced. It calls a method
        * of the profiler object according to the event type and schedules
        * itself RESOLUTION ticks later.
        */
        void process(){

            switch(traceType){
                case SEND:
                    profiler->writeSendEntry();
                    break;
                case CHANNEL:
                    profiler->writeChannelEntry();
                    break;
                default:
                    fatal("Unimplemented interconnect trace type");
            }

            this->schedule(curTick + RESOLUTION);
        }

        /**
        * @return A textual description of the event
        */
        virtual const char *description(){
            return "InterconnectProfileEvent";
        }
};

#endif //__INTERCONNECT_PROFILE_HH__
```

C.1.18 Interconnect Profiler Code File

```
#include "interconnect_profile.hh"
#include "sim/builder.hh"

#include <fstream>

using namespace std;

class Interconnect;

InterconnectProfile::InterconnectProfile(const std::string &_name,
                                         bool _traceSends,
                                         bool _traceChannelUtil,
                                         Tick _startTick,
                                         Interconnect* _interconnect)
    : SimObject(_name)
{
    assert(_interconnect != NULL);

    traceSends = _traceSends;
    traceChannelUtil = _traceChannelUtil;
    startTick = _startTick;
    interconnect = _interconnect;

    interconnect->registerProfiler(this);

    sendFileName = "interconnectSendProfile.txt";
    channelFileName = "interconnectChannelProfile.txt";
    channelExplFileName = "interconnectChannelExplanation.txt";

    initSendFile();

    sendEvent = new InterconnectProfileEvent(this, SEND);
    sendEvent->schedule(startTick);

    bool doChannelTrace = initChannelFile();

    if(doChannelTrace){
        channelEvent = new InterconnectProfileEvent(this, CHANNEL);
        channelEvent->schedule(startTick);
    }
}

void
InterconnectProfile::initSendFile(){
    ofstream sendfile(sendFileName.c_str());
    sendfile << "Clock Cycle;Data Sends;Instruction Sends;"
             << "Coherence Sends;Total Sends\n";
    sendfile.flush();
    sendfile.close();
}

void
InterconnectProfile::writeSendEntry(){

    // get sample
    int data = 0, insts = 0, coherence = 0, total = 0;
    interconnect->getSendSample(&data, &insts, &coherence, &total);
    assert(data + insts + coherence == total);
    // write to file
    ofstream sendfile(sendFileName.c_str(), ofstream::app);
    sendfile << curTick << ";"
             << data << ";"
             << insts << ";"
             << coherence << ";"
             << total << "\n";
    sendfile.flush();
    sendfile.close();
}

bool
InterconnectProfile::initChannelFile(){

    int channelCount = interconnect->getChannelCount();

    if(channelCount != -1){

        // Write first line in tracefile
        ofstream chanfile(channelFileName.c_str());
        chanfile << "Clock Cycle; ";

        for(int i =0;i<channelCount;i++){
            chanfile << "Channel " << i;

            if(i == channelCount-1) chanfile << "\n";
            else chanfile << "; ";
        }

        chanfile.flush();
        chanfile.close();

        // Write channel explanation
        ofstream explFile(channelExplFileName.c_str());
        interconnect->writeChannelDecriptor(explFile);
        explFile.flush();
        explFile.close();

        return true;
    }

    //interconnect does not support channel profiling
    return false;

}

void
InterconnectProfile::writeChannelEntry(){

    int channelCount = interconnect->getChannelCount();
    vector<int> res = interconnect->getChannelSample();
    assert(res.size() == channelCount);

    ofstream channelfile(channelFileName.c_str(), ofstream::app);
    channelfile << curTick << ";";

    for(int i=0;i<res.size();i++){
        channelfile << ((double) res[i] / (double) RESOLUTION);
        if(i == res.size()-1) channelfile << "\n";
        else channelfile << ";";
    }

    channelfile.flush();
    channelfile.close();
}

#ifndef DOXYGEN_SHOULD_SKIP_THIS

BEGIN_DECLARE_SIM_OBJECT_PARAMS(InterconnectProfile)
    Param<bool> traceSends;
    Param<bool> traceChannelUtil;
    Param<Tick> traceStartTick;
    SimObjectParam<Interconnect*> interconnect;
END_DECLARE_SIM_OBJECT_PARAMS(InterconnectProfile)

BEGIN_INIT_SIM_OBJECT_PARAMS(InterconnectProfile)
    INIT_PARAM(traceSends, "Trace number of sends?"),
    INIT_PARAM(traceChannelUtil, "Trace channel utilisation?"),
    INIT_PARAM(traceStartTick, "The clock cycle to start the trace"),
    INIT_PARAM(interconnect, "The interconnect to profile")
END_INIT_SIM_OBJECT_PARAMS(InterconnectProfile)

CREATE_SIM_OBJECT(InterconnectProfile)
{
    return new InterconnectProfile(getInstanceName(),
                                   traceSends,
                                   traceChannelUtil,
                                   traceStartTick,
                                   interconnect);
}

REGISTER_SIM_OBJECT("InterconnectProfile", InterconnectProfile)

#endif //DOXYGEN_SHOULD_SKIP_THIS
```

C.2 Coherence Protocol Extension Code

C.2.1 Directory Protocol Header File

```
#ifndef __DIRECTORY_PROTOCOL_HH__
#define __DIRECTORY_PROTOCOL_HH__

#include "sim/sim_object.hh"
#include "mem/cache/base_cache.hh"
#include "mem/mem_req.hh"
#include "mem/cache/cache_blk.hh"
#include "mem/cache/tags/cache_tags.hh"

#define OUTFILENAME "coherencetrace.txt"

class BaseCache;
template <class TagStore> class DirectoryProtocolDumpEvent;

/**
* This class implements an interface between the directory protocol
* implementation added in this work and the M5 cache implementation. To add
* more directory implementations a new subclass can be added to this file.
*
* The responsibility of this class is to specify convenience methods that are
* needed for more than one protocol. In the current implementation, this
* functionality is limited to protocol trace facilities for debugging,
* coherence message tracing and access to the directory.
*
* @author Magnus Jahre
*/
template <class TagStore>
class DirectoryProtocol
{
    protected:

        BaseCache *cache;
        typedef typename TagStore::BlkType BlkType;
        std::string cacheName;

        std::string protocol;
        int directoryCpuCount;
        int directoryCpuID;

        std::map<Addr, int> blockStore;

        bool doTrace;
        int traceStart;

        DirectoryProtocolDumpEvent<TagStore>* dumpEvent;

        double lastNumRedirectedReads;
        double lastNumOwnerRequests;
        double lastNumOwnerWritebacks;
        double lastNumSharerWritebacks;
        double lastNumNACKs;

        std::string dumpFileName;

    public:
        MemReqList directoryRequests;

        // Stats
        Stats::Scalar<> numRedirectedReads;
        Stats::Scalar<> numOwnerRequests;
        Stats::Scalar<> numOwnerWritebacks;
        Stats::Scalar<> numSharerWritebacks;
        Stats::Scalar<> numNACKs;

    public:

        /**
        * This constructor creates the message trace and message profile if
        * needed.
        *
        * @param _name          The name from the config file.
        * @param _protocol      The name of the protocol that will be used
        * @param _doTrace       If true, the protocol actions are written to a
        *                       trace file
        * @param _dumpInterval The number of clock cycles between each protocol
        *                       profile event
        * @param _traceStart    The clock cycle to start tracing protocol
        *                       actions
        */
        DirectoryProtocol(const std::string &_name,
                          const std::string &_protocol,
                          bool _doTrace,
                          int _dumpInterval,
                          int _traceStart);

        /**
        * Empty destructor.
        */
        virtual ~DirectoryProtocol(){}

        /**
        * This method is called in the cache builder and makes sure the
        * cache knows which cache it is associated with.
        *
        * @param _cache A pointer to the associated cache
        */
        void setCache(BaseCache* _cache){
            cache = _cache;
            assert(cache != NULL);
        }

        /**
        * This method sets the number of cpus and the cpu id of the associated
        * cache for the directory protocol. It is called in the cache
        * constructor.
        *
        * @param num_cpus The number of CPUs in the system.
        * @param cpuId    The CPU ID of the cache associated with this
        *                 protocol.
        */
        void setCpuCount(int num_cpus, int cpuId){
            directoryCpuCount = num_cpus;
            directoryCpuID = cpuId;
        }
```

```
/**
* Registers the statistics variables with the M5 statistics module.
*/
void regStats();

/**
* Writes a trace line into the trace file.
*
* @param cachename    The name of the cache that called this method
* @param message      The message to write to the file
* @param owner        The ID of the cache that owns the block
* @param state        The state of the cache block
* @param paddr        The address of the cache block
* @param blkSize      The block size of the cache
* @param presentFlags A bool array showing which caches have a copy of
*                     the cache block
*/
void writeTraceLine(const std::string cachename,
                    const std::string message,
                    const int owner,
                    const DirectoryState state,
                    const Addr paddr,
                    const Addr blkSize,
                    bool* presentFlags);

/**
* When this method is called, the coherence message profile is written
* to the profile file and the counters are reset.
*/
void dumpStats();

/**
* This method returns checks if a given address is owned.
*
* @param address The cache block address
*
* @return True if the cache block is owned
*/
bool isOwned(Addr address);

/**
* Retrieves the CPU ID of the cache currently owning a block.
*
* @param address The cache block address.
*
* @return The CPU ID of the cache block owner.
*/
int getOwner(Addr address);

/**
* This method sets the owner of a cache block.
*
* @param address  The cache block address.
* @param newOwner The CPU ID of the new owner.
*/
void setOwner(Addr address, int newOwner);

/**
* This method removes the owner of a given cache block.
*
* @param address The cache block address.
```

```
        */
        void removeOwner(Addr address);

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual void sendDirectoryMessage(MemReqPtr& req, int lat) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual void sendNACK(MemReqPtr& req,
                              int lat,
                              int toID,
                              int fromID) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual bool doDirectoryAccess(MemReqPtr& req) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual bool doL1DirectoryAccess(MemReqPtr& req, BlkType* blk) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual bool handleDirectoryResponse(MemReqPtr& req,
                                             TagStore *tags) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual bool handleDirectoryFill(MemReqPtr& req,
                                         BlkType* blk,
                                         MemReqList& writebacks,
                                         TagStore* tags) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual bool doDirectoryWriteback(MemReqPtr& req) = 0;

        /**
        * This method is a part of the directory coherence interface and is
        * documented in the subclasses.
        */
        virtual MemAccessResult handleL1DirectoryMiss(MemReqPtr& req) = 0;


};
```

```
/**
* This class implements an event that dumps message statistics to a file.
*
* @author Magnus Jahre
*/
template <class TagStore>
class DirectoryProtocolDumpEvent : public Event
{

    public:

        DirectoryProtocol<TagStore>* protocol;
        int dumpInterval;

        /**
        * This constructor initialises the member variables with the provided
        * arguments.
        *
        * @param _protocol     A pointer to the directory protocol used
        * @param _dumpInterval The number of clock cycles between each dump
        */
        DirectoryProtocolDumpEvent(DirectoryProtocol<TagStore>* _protocol,
                                   int _dumpInterval)
            : Event(&mainEventQueue)
        {
            protocol = _protocol;
            dumpInterval = _dumpInterval;
        }

        /**
        * This method is called when the event is serviced. It makes the
        * protocol dump the gathered statistics and reschedules itself.
        */
        void process(){
            protocol->dumpStats();
            this->schedule(curTick + dumpInterval);
        }

        /**
        * @return A textual description of the event.
        */
        virtual const char *description(){
            return "DirectoryProtocol dump event";
        }
};

#endif //__DIRECTORY_PROTOCOL_HH__
```

C.2.2 Directory Protocol Code File

```
#include "directory.hh"
#include "sim/builder.hh"

#include <fstream>

using namespace std;
// using namespace __gnu_cxx;


template<class TagStore>
DirectoryProtocol<TagStore>::DirectoryProtocol(const std::string &_cacheName,
                                               const std::string &_protocol,
                                               bool _doTrace,
                                               int _dumpInterval,
                                               int _traceStart){

    protocol = _protocol;
    cacheName = _cacheName;

    doTrace = _doTrace;
    traceStart = _traceStart;

    if(_dumpInterval != 0){
        dumpEvent =
                new DirectoryProtocolDumpEvent<TagStore>(this, _dumpInterval);
        dumpEvent->schedule(curTick + _dumpInterval);

        dumpFileName = "coherencedump." + _cacheName + ".txt";

        ofstream dumpfile(dumpFileName.c_str());
        dumpfile << "Clock Cycle;Redirected Reads; Transfer Owner Request; "
                 << "Owner Writebacks; Sharer Writebacks; NACKs\n";
        dumpfile.flush();
        dumpfile.close();
    }
    else{
        dumpEvent = NULL;
    }
    lastNumRedirectedReads = 0;
    lastNumOwnerRequests = 0;
    lastNumOwnerWritebacks = 0;
    lastNumSharerWritebacks = 0;
    lastNumNACKs = 0;

    if(doTrace){
        ofstream tracefile(OUTFILENAME);
        tracefile << "M5 coherence trace:\n\n";
        tracefile.flush();
        tracefile.close();
    }
}

template<class TagStore>
void
DirectoryProtocol<TagStore>::regStats(){

    using namespace Stats;
    numRedirectedReads
            .name(name() + ".num_redirected_reads")
            .desc("total number of reads redirected to a different cache")
            ;

    numOwnerRequests
            .name(name() + ".num_owner_requests")
            .desc("total number of owner requests issued to allready owned"
                  " blocks")
            ;

    numOwnerWritebacks
            .name(name() + ".num_owner_writebacks")
            .desc("total number of times this cache has written back an owned"
                  " block")
            ;

    numSharerWritebacks
            .name(name() + ".num_sharer_writebacks")
            .desc("total number of times this cache has written back a block"
                  " owned by a different cache")
            ;

    numNACKs
            .name(name() + ".num_nacks")
            .desc("total number of negative acknowledgements sent from this"
                  " cache")
            ;
}

template<class TagStore>
void
DirectoryProtocol<TagStore>::writeTraceLine(const std::string cachename,
                                            const std::string message,
                                            const int owner,
                                            const DirectoryState state,
                                            const Addr paddr,
                                            const Addr blkSize,
                                            bool* presentFlags){
    if(doTrace && curTick >= traceStart){

        Addr blkAddr = (paddr & ~((Addr)blkSize - 1));

        ofstream tracefile(OUTFILENAME, ofstream::app);
        tracefile << curTick
                  << "; " << cachename
                  << "; " << blkAddr << ", " << hex
                  << showbase << blkAddr << dec
                  << "; Owner " << owner
                  << "; " << state
                  << "; " << message;

        if(presentFlags != NULL){
            tracefile << "; [";
            for(int i=0;i<directoryCpuCount;i++){
                tracefile << (presentFlags[i] ? "1" : "0");
                if(i != (directoryCpuCount-1)) tracefile << ",";
            }
            tracefile << "]";
        }
        tracefile << "\n";
        tracefile.flush();
        tracefile.close();
    }
}

template<class TagStore>
void
DirectoryProtocol<TagStore>::dumpStats(){

    ofstream dumpfile(dumpFileName.c_str(), ofstream::app);

    dumpfile << curTick << "; "
             << (numRedirectedReads.value() - lastNumRedirectedReads) << "; "
             << (numOwnerRequests.value() - lastNumOwnerRequests) << "; "
             << (numOwnerWritebacks.value() - lastNumOwnerWritebacks) << "; "
             << (numSharerWritebacks.value() - lastNumSharerWritebacks) << "; "
             << (numNACKs.value() - lastNumNACKs) << "\n";

    lastNumRedirectedReads = numRedirectedReads.value();
    lastNumOwnerRequests = numOwnerRequests.value();
    lastNumOwnerWritebacks = numOwnerWritebacks.value();
    lastNumSharerWritebacks = numSharerWritebacks.value();
    lastNumNACKs = numNACKs.value();

    dumpfile.flush();
    dumpfile.close();
}

template<class TagStore>
bool
DirectoryProtocol<TagStore>::isOwned(Addr address){
    if(blockStore.find(address) == blockStore.end()){
        return false;
    }
    return true;
}

template<class TagStore>
int
DirectoryProtocol<TagStore>::getOwner(Addr address){
    map<Addr, int>::iterator found = blockStore.find(address);
    if(found != blockStore.end()){
        return found->second;
    }
    return -1;
}

template<class TagStore>
void
DirectoryProtocol<TagStore>::setOwner(Addr address, int newOwner){
    blockStore[address] = newOwner;
}

template<class TagStore>
void
DirectoryProtocol<TagStore>::removeOwner(Addr address){
    map<Addr, int>::iterator eraseIt = blockStore.find(address);
    assert(eraseIt != blockStore.end());
    blockStore.erase(eraseIt);
}
```

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS

// Include config files
// Must be included first to determine which caches we want
#include "mem/config/cache.hh"
#include "mem/config/compression.hh"


// Tag Templates
#if defined(USE_CACHE_LRU)
#include "mem/cache/tags/lru.hh"
#endif

#if defined(USE_CACHE_FALRU)
#include "mem/cache/tags/fa_lru.hh"
#endif

#if defined(USE_CACHE_IIC)
#include "mem/cache/tags/iic.hh"
#endif

#if defined(USE_CACHE_SPLIT)
#include "mem/cache/tags/split.hh"
#endif

#if defined(USE_CACHE_SPLIT_LIFO)
#include "mem/cache/tags/split_lifo.hh"
#endif

// Compression Templates
#include "base/compression/null_compression.hh"
#if defined(USE_LZSS_COMPRESSION)
#include "base/compression/lzss_compression.hh"
#endif

#if defined(USE_CACHE_FALRU)
    template class DirectoryProtocol<CacheTags<FALRU, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class DirectoryProtocol<CacheTags<FALRU, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_IIC)
    template class DirectoryProtocol<CacheTags<IIC, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class DirectoryProtocol<CacheTags<IIC, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_LRU)
    template class DirectoryProtocol<CacheTags<LRU, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class DirectoryProtocol<CacheTags<LRU, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_SPLIT)
    template class DirectoryProtocol<CacheTags<Split, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class DirectoryProtocol<CacheTags<Split, LZSSCompression> >;
```

```
#endif
#endif

#if defined(USE_CACHE_SPLIT_LIFO)
    template class DirectoryProtocol<CacheTags<SplitLIFO,NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class DirectoryProtocol<CacheTags<SplitLIFO,LZSSCompression> >;
#endif
#endif

#endif // DOXYGEN_SHOULD_SKIP_THIS
```

C.2.3 Stenström Protocol Header File

```
#include "directory.hh"


/**
* This class implements the Stenstrom directory cache coherence protocol.
*
* @author Magnus Jahre
*/
template <class TagStore>
class StenstromProtocol : public DirectoryProtocol<TagStore>
{
    using DirectoryProtocol<TagStore>::cache;
    using DirectoryProtocol<TagStore>::directoryRequests;
    using DirectoryProtocol<TagStore>::cacheName;

    using DirectoryProtocol<TagStore>::numRedirectedReads;
    using DirectoryProtocol<TagStore>::numOwnerRequests;
    using DirectoryProtocol<TagStore>::numOwnerWritebacks;
    using DirectoryProtocol<TagStore>::numSharerWritebacks;
    using DirectoryProtocol<TagStore>::numNACKs;

    typedef typename TagStore::BlkType BlkType;

    private:
        DirectoryProtocol<TagStore>* parentPtr;

        std::map<Addr, bool*> outstandingWritebackWSAddrs;
        std::map<Addr, int> outstandingOwnerTransAddrs;

    public:

        /**
        * This constructor creates a stenstrom protocol object.
        *
        * @param _name          The name from the config file.
        * @param _protocol      The name of the protocol that will be used
        * @param _doTrace       If true, the protocol actions are written to a
        *                       trace file
        * @param _dumpInterval  The number of clock cycles between each protocol
        *                       profile event
        * @param _traceStart    The clock cycle to start tracing protocol
        *                       actions
        */
        StenstromProtocol(const std::string &_name,
                          const std::string &_protocol,
                          bool _doTrace,
                          int _dumpInterval,
                          int _traceStart):
            DirectoryProtocol<TagStore>(_name,
                                        _protocol,
                                        _doTrace,
                                        _dumpInterval,
                                        _traceStart)
        {
            parentPtr = dynamic_cast<DirectoryProtocol<TagStore>* >(this);
            assert(parentPtr != NULL);
        }
```

```
/**
* Depending on whether the cache is a L1 cache or an L2 cache, the
* details of sending a message is different. This method hides
* these details.
*
* @param req The request to send.
* @param lat The number of clock cycles before the request should be
*            sent.
*/
void sendDirectoryMessage(MemReqPtr& req, int lat);

/**
* Negative Acknowledge messages (NACKs) are issued quite often by the
* protocol. This method hides the details of how they are sent.
*
* @param req    The request to send
* @param lat    The latency before the request is sent
* @param toID   The reciever CPU ID
* @param fromID The sender CPU ID
*/
void sendNACK(MemReqPtr& req, int lat, int toID, int fromID);

/**
* This message handles directory accesses in the L2 cache access
* method.
*
* @param req The current request
*
* @return True, if the request was handled by the protocol.
*/
bool doDirectoryAccess(MemReqPtr& req);

/**
* This message handles directory accesses in the L1 cache access
* method. It is only called if it is a hit in the cache.
*
* @param req The current request
* @param blk A pointer to the current cache block
*
* @return True, if the request was handled by the protocol.
*/
bool doL1DirectoryAccess(MemReqPtr& req, BlkType* blk);

/**
* When a L1 cache recieves a response, some of these must be handled
* without actually accessing the cache. These situations are handled
* by this method.
*
* Some of these cases require accessing the tag store to aquire updated
* information on a cache block. Consequently, the tag store is provided
* as a parameter.
*
* @param req  The current request
* @param tags A pointer to the cache tag store
*
* @return True, if the request was handled by the protocol.
*/
bool handleDirectoryResponse(MemReqPtr& req, TagStore *tags);

/**
* When a cache fill is recieved, this must be checked by the coherence
```

```
        * protocol. These cases are handled by this method.
        *
        * @param req        The current request.
        * @param blk        The current cache block.
        * @param writebacks The current list of writebacks.
        * @param tags       A pointer to the cache's tag store.
        *
        * @return True, if the request was handled by the protocol.
        */
        bool handleDirectoryFill(MemReqPtr& req,
                                 BlkType* blk,
                                 MemReqList& writebacks,
                                 TagStore* tags);

        /**
        * Writebacks of shared blocks require special handling. This method
        * checks the writebacks and handles them according to the protocol.
        *
        * @param req The writeback request
        *
        * @return True, if the request was handled by the protocol.
        */
        bool doDirectoryWriteback(MemReqPtr& req);

        /**
        * Some race conditions cause directory messages to miss in the L1
        * cache. These cases are handled by this method.
        *
        * @param req The current memory request
        *
        * @return If the return value is different from BA_NO_RESULT, the
        *         result should be used directly.
        */
        MemAccessResult handleL1DirectoryMiss(MemReqPtr& req);

    private:

        void setUpRedirectedRead(MemReqPtr& req,
                                 int fromProcessorID,
                                 int toProcessorID);

        void setUpRedirectedReadReply(MemReqPtr& req,
                                      int fromProcessorID,
                                      int toProcessorID);

        void setUpOwnerTransferInL2(MemReqPtr& req,
                                    int oldOwner,
                                    int newOwner);

        void setUpACK(MemReqPtr& req, int toID, int fromID);
};
```

C.2.4 Stenström Protocol Code File

```
#include "stenstrom.hh"

using namespace std;

template<class TagStore>
void
StenstromProtocol<TagStore>::sendDirectoryMessage(MemReqPtr& req, int lat){

    if(cache->isDirectoryAndL1DataCache()){
        directoryRequests.push_back(req);
        cache->setMasterRequest(Request_DirectoryCoherence, curTick + lat);
        return;
    }

    if(cache->isDirectoryAndL2Cache()){
        cache->respond(req, curTick + lat);
    }
}

template<class TagStore>
void
StenstromProtocol<TagStore>::sendNACK(MemReqPtr& req,
                                      int lat,
                                      int toID,
                                      int fromID){

    req->toProcessorID = toID;
    req->fromProcessorID = fromID;
    req->toInterfaceID = -1;
    req->fromInterfaceID = -1;
    req->dirNACK = true;

    numNACKs++;

    writeTraceLine(cacheName,
                   "NACK Sent",
                   -1,
                   DirNoState,
                   (req->paddr & ~((Addr)cache->getBlockSize() - 1)),
                   cache->getBlockSize(),
                   NULL);

    sendDirectoryMessage(req, lat);
}

template<class TagStore>
bool
StenstromProtocol<TagStore>::doDirectoryAccess(MemReqPtr& req){

    int lat = cache->getHitLatency();
    int fromCpuId = req->xc->cpu->params->cpu_id;
    Addr tmpL2BlkAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);

    // Directory info is only stored for data blocks
    if(!req->readOnlyCache){

        if(!parentPtr->isOwned(tmpL2BlkAddr)){
```

```
if(req->dirNACK){

    if(req->cmd == DirSharerWriteback){

        // the block is not shared anymore, discard message
        writeTraceLine(cacheName,
                       "Recieved NACK to block that is no longer "
                       "shared, discarding message",
                       -1,
                       DirNoState,
                       tmpL2BlkAddr,
                       cache->getBlockSize(),
                       NULL);
        return true;
    }
    else if(req->cmd = DirOwnerTransfer){

        // the old owner wrote back the block
        // make the requester retransmit the request
        assert(outstandingOwnerTransAddrs.find(tmpL2BlkAddr)
                != outstandingOwnerTransAddrs.end());
        outstandingOwnerTransAddrs.erase(
                outstandingOwnerTransAddrs.find(tmpL2BlkAddr));

        req->ownerWroteBack = true;
        sendNACK(req, cache->getHitLatency(), fromCpuId, -1);
        return true;
    }
    else{
        fatal("Unimplemented NACK type in doDirectoryAccess()");
        return true;
    }
}

if(req->cmd == DirRedirectRead){
    parentPtr->setOwner(tmpL2BlkAddr, fromCpuId);
    req->owner = fromCpuId;

    writeTraceLine(cacheName,
                   "Redirected Read to not owned block recieved, "
                   "requester is new owner",
                   parentPtr->getOwner(tmpL2BlkAddr),
                   DirNoState,
                   tmpL2BlkAddr,
                   cache->getBlockSize(),
                   NULL);

    req->toProcessorID = fromCpuId;
    req->fromProcessorID = -1;
    req->toInterfaceID = -1;
    req->fromInterfaceID = -1;

    sendDirectoryMessage(req, cache->getHitLatency());

    return true;
}

if(req->cmd == DirOwnerTransfer){
    // the previous owner wrote back the block
    // in the middle of the transfer
    // make the requester resend as a read
```

```
            assert(outstandingOwnerTransAddrs.find(tmpL2BlkAddr)
                    == outstandingOwnerTransAddrs.end());
            req->ownerWroteBack = true;
            sendNACK(req, cache->getHitLatency(), fromCpuId, -1);
            return true;
        }

        assert(!req->cmd.isDirectoryMessage());

        // The requesting cache is now the owner, do normal response
        parentPtr->setOwner(tmpL2BlkAddr, fromCpuId);
        req->owner = fromCpuId;
        req->fromProcessorID = -1;
    }
    else{

        if(req->dirACK){
            if(req->cmd == DirOwnerTransfer){

                // owner transfer to this block is allowed again
                outstandingOwnerTransAddrs.erase(
                        outstandingOwnerTransAddrs.find(tmpL2BlkAddr));

                writeTraceLine(cacheName,
                               "Owner Transfer ACK recieved",
                               parentPtr->getOwner(tmpL2BlkAddr),
                               DirNoState,
                               tmpL2BlkAddr,
                               cache->getBlockSize(),
                               NULL);

                return true;
            }
            else{
                fatal("ACK type not implemented");
            }
        }
        else if(req->dirNACK){
            if(req->cmd == DirOwnerTransfer){
                // the owner had not recieved the block yet
                // clean up and send a NACK to the requester
                // NOTE: use fromProcessorID here
                // fromCPUId identifies the original requester
                assert(fromCpuId == parentPtr->getOwner(tmpL2BlkAddr));
                assert(req->fromProcessorID > -1);

                // reset too original owner and remove address
                // from blocked list
                int requesterID = parentPtr->getOwner(tmpL2BlkAddr);
                parentPtr->setOwner(tmpL2BlkAddr, req->fromProcessorID);
                outstandingOwnerTransAddrs.erase(
                        outstandingOwnerTransAddrs.find(tmpL2BlkAddr));

                sendNACK(req, lat, requesterID, -1);
                return true;
            }
            else if(req->cmd == DirSharerWriteback){

                req->dirNACK = false;
                req->toProcessorID = parentPtr->getOwner(tmpL2BlkAddr);
                req->fromProcessorID = -1;
                req->toInterfaceID = -1;

                writeTraceLine(cacheName,
                               "Recieved NACK on sharer writeback, "
                               "retransmitting",
                               parentPtr->getOwner(tmpL2BlkAddr),
                               DirNoState,
                               tmpL2BlkAddr,
                               cache->getBlockSize(),
                               NULL);

                sendDirectoryMessage(req, lat);

                return true;
            }
            else{
                fatal("Unimplemented NACK type (in L2)");
            }
        }
        else if(req->writeMiss){

            if(outstandingOwnerTransAddrs.find(tmpL2BlkAddr)
               != outstandingOwnerTransAddrs.end()){
                //destination was req->fromProcessorID
                sendNACK(req, lat, fromCpuId, -1);
                return true;
            }
            outstandingOwnerTransAddrs[tmpL2BlkAddr] = fromCpuId;

            // write to allready owned block
            int oldOwner = parentPtr->getOwner(tmpL2BlkAddr);
            parentPtr->setOwner(tmpL2BlkAddr, fromCpuId);

            writeTraceLine(cacheName,
                           "Write miss to owned block",
                           oldOwner,
                           DirNoState,
                           tmpL2BlkAddr,
                           cache->getBlockSize(),
                           NULL);

            /* update and send request */
            setUpOwnerTransferInL2(req, oldOwner, fromCpuId);
            sendDirectoryMessage(req, lat);

            return true;

        }
        else if(req->cmd == Writeback || req->cmd == DirWriteback){

            if(outstandingOwnerTransAddrs.find(tmpL2BlkAddr)
               == outstandingOwnerTransAddrs.end()){
                // not part of an owner transfer, must be from owner
                assert(parentPtr->getOwner(tmpL2BlkAddr) == fromCpuId);
            }

            // this block is not in any L1 cache, reset owner status
            parentPtr->removeOwner(tmpL2BlkAddr);

            // carry out normal writeback actions in the write back case
            if(req->cmd == DirWriteback){
                return true;
            }

        }
        else if(req->cmd == DirOwnerTransfer){

            if(outstandingOwnerTransAddrs.find(tmpL2BlkAddr)
               != outstandingOwnerTransAddrs.end()){
                //destination was req->fromProcessorID
                sendNACK(req, lat, fromCpuId, -1);
                return true;
            }
            outstandingOwnerTransAddrs[tmpL2BlkAddr] = fromCpuId;

            // change owner status and forward to old owner
            int oldOwner = parentPtr->getOwner(tmpL2BlkAddr);
            int newOwner = req->fromProcessorID;

            parentPtr->setOwner(tmpL2BlkAddr, newOwner);

            writeTraceLine(cacheName,
                           "Owner Change Request Granted",
                           parentPtr->getOwner(tmpL2BlkAddr),
                           DirNoState,
                           tmpL2BlkAddr,
                           cache->getBlockSize(),
                           NULL);

            /* update and send request */
            setUpOwnerTransferInL2(req, oldOwner, newOwner);
            sendDirectoryMessage(req, lat);

            return true;
        }
        else if(req->cmd == Read){

            // return the owner state to the requesting cache
            int owner = parentPtr->getOwner(tmpL2BlkAddr);

            assert(fromCpuId != owner);

            req->owner = owner;
        }
        else if(req->cmd == DirSharerWriteback){
            // block replaced from a non-owner cache, inform owner
            assert(req->replacedByID == -1);

            req->toProcessorID = parentPtr->getOwner(tmpL2BlkAddr);
            req->fromProcessorID = -1;
            req->toInterfaceID = -1;
            req->fromInterfaceID = -1;
            req->replacedByID = fromCpuId;

            writeTraceLine(cacheName,
                           "Forwarding sharer writeback to owner",
                           req->owner,
                           DirNoState,
                           tmpL2BlkAddr,
                           cache->getBlockSize(),
                           req->presentFlags);
                sendDirectoryMessage(req, lat);
                return true;
            }
            else if(req->cmd == DirRedirectRead){
                // a redirected read got NACKed
                // return the owner state to the requester
                req->owner = parentPtr->getOwner(tmpL2BlkAddr);
                req->toProcessorID = req->fromProcessorID;
                req->fromProcessorID = -1;
                req->toInterfaceID = -1;
                req->fromInterfaceID = -1;

                writeTraceLine(cacheName,
                               "Got Redirected Read, "
                               "informing requester of current owner",
                               req->owner,
                               DirNoState,
                               tmpL2BlkAddr,
                               cache->getBlockSize(),
                               NULL);

                sendDirectoryMessage(req, lat);
                return true;
            }
            else{
                fatal("In L2: access to a block that is allready owned, "
                      "unimplemented request type");
            }
        }
    }
    return false;
}

template<class TagStore>
bool
StenstromProtocol<TagStore>::doL1DirectoryAccess(MemReqPtr& req, BlkType* blk){

    int lat = cache->getHitLatency();
    Addr tmpL1BlkAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);

    if(!req->cmd.isDirectoryMessage()){
        assert(req->xc->cpu->params->cpu_id == cache->getCacheCPUid());
    }

    switch(req->cmd){

        case Write:
            if(blk->dirState == DirOwnedExGR
               || blk->dirState == DirOwnedNonExGR){
                // Write is OK
            }
            else{
                // we need to request ownership for this block
                req->oldCmd = req->cmd;
                req->cmd = DirOwnerTransfer;
                req->toProcessorID = -1;
                req->toInterfaceID = -1;
                req->fromProcessorID = cache->getCacheCPUid();

                numOwnerRequests++;
                writeTraceLine(cacheName,
                               "Issuing owner transfer request",
                               blk->owner,
                               blk->dirState,
                               tmpL1BlkAddr,
                               cache->getBlockSize(),
                               blk->presentFlags);

                sendDirectoryMessage(req, lat);

                return true;
            }
            break;

        case Read:
            if(blk->dirState == DirOwnedExGR
               || blk->dirState == DirOwnedNonExGR){
                // Read is OK
            }
            else{
                setUpRedirectedRead(req, cache->getCacheCPUid(), blk->owner);

                numRedirectedReads++;

                writeTraceLine(cacheName,
                               "Issuing Redirected Read (1)",
                               blk->owner,
                               blk->dirState,
                               tmpL1BlkAddr,
                               cache->getBlockSize(),
                               blk->presentFlags);

                sendDirectoryMessage(req,lat);

                return true;
            }
            break;

        case Soft_Prefetch:
            // discard, since it is a hit in the L1 cache we don't care
            break;

        case DirRedirectRead:

            if(blk->dirState == DirOwnedExGR){

                // make sure it is really one owner
                int presentCount = 0;
                for(int i=0;i<cache->cpuCount;i++){
                    if(blk->presentFlags[i]) presentCount++;
                }
                assert(presentCount == 1);

                // update block state
                int fromCpuID = req->fromProcessorID;
                blk->presentFlags[fromCpuID] = true;

                // this request might be from ourselves
                int newSharerCount = 0;
                for(int i=0;i<cache->cpuCount;i++){
                    if(blk->presentFlags[i]) newSharerCount++;
                }
                if(newSharerCount == 1){
                    // the block is present in our cache, answer the request
                    writeTraceLine(cacheName,
                                   "Answering Redirected Read from myself "
                                   "(return to cache) (1)",
                                   blk->owner,
                                   blk->dirState,
                                   tmpL1BlkAddr,
                                   cache->getBlockSize(),
                                   blk->presentFlags);

                    blk->dirState = DirOwnedExGR;
                    return false;
                }


                blk->dirState = DirOwnedNonExGR;

                // update request and send it
                setUpRedirectedReadReply(req,
                                         cache->getCacheCPUid(),
                                         fromCpuID);

                sendDirectoryMessage(req, lat);

            }
            else if(blk->dirState == DirOwnedNonExGR){

                int fromCpuID = req->fromProcessorID;

                // sending redirected reads to ourselves
                // will cause a deadlock, let it finish
                if(fromCpuID == cache->getCacheCPUid()){
                    writeTraceLine(cacheName,
                                   "Answering Redirected Read from myself "
                                   "(return to cache) (2)",
                                   blk->owner,
                                   blk->dirState,
                                   tmpL1BlkAddr,
                                   cache->getBlockSize(),
                                   blk->presentFlags);

                    return false;
                }
                assert(fromCpuID != cache->getCacheCPUid());

                blk->presentFlags[fromCpuID] = true;
                setUpRedirectedReadReply(req,
                                         cache->getCacheCPUid(),
                                         fromCpuID);

                sendDirectoryMessage(req, lat);
            }
            else{
                sendNACK(req,
                         lat,
                         req->fromProcessorID,
                         cache->getCacheCPUid());
                return true;
            }
```

```
            writeTraceLine(cacheName,
                           "Answering Redirected Read Request",
                           blk->owner,
                           blk->dirState,
                           tmpL1BlkAddr,
                           cache->getBlockSize(),
                           blk->presentFlags);

            return true;

            break;

        case DirOwnerTransfer:
        {

            // Owner transfer is finished
            // Update the block state and let the request go through
            assert(req->owner == cache->getCacheCPUid());
            assert(req->presentFlags != NULL);
            // ownership might be requested by more than one request
            // consequently, we might be transfering ownership to ourselves
            // i.e. no blk->owner != req->owner assertion
            assert(blk->presentFlags == NULL);

            blk->presentFlags = req->presentFlags;
            // the flags must not be removed when the request is deleted
            req->presentFlags = NULL;
            blk->owner = req->owner;

            // the previous owner does not necessarily know about this cache
            blk->presentFlags[cache->getCacheCPUid()] = true;

            int sharerCount = 0;
            for(int i=0;i<cache->cpuCount;i++){
                if(blk->presentFlags[i]) sharerCount++;
            }

            assert(sharerCount > 0);
            if(sharerCount == 1) blk->dirState = DirOwnedExGR;
            else blk->dirState = DirOwnedNonExGR;

            writeTraceLine(cacheName,
                           "Owner transfer complete",
                           blk->owner,
                           blk->dirState,
                           tmpL1BlkAddr,
                           cache->getBlockSize(),
                           blk->presentFlags);

            // Send ACK to the L2 cache
            MemReqPtr tmpReq = buildReqCopy(req,
                                            cache->cpuCount,
                                            DirOwnerTransfer);
            setUpACK(tmpReq, -1, cache->getCacheCPUid());
            sendDirectoryMessage(tmpReq, cache->getHitLatency());
        }

        break;

        case DirOwnerWriteback:
```

```
        {
            // hit on owner writeback
            assert(req->presentFlags == NULL);
            assert(req->paddr ==
                    (req->paddr & ~((Addr)cache->getBlockSize() - 1)));

            // make a copy that will become the ownership request
            MemReqPtr ownershipReq = buildReqCopy(req,
                                                  cache->cpuCount,
                                                  DirOwnerTransfer);

            // set addressing info
            ownershipReq->toProcessorID = -1;
            ownershipReq->fromProcessorID = cache->getCacheCPUid();
            ownershipReq->toInterfaceID = -1;

            // send an ACK to the current owner
            req->toProcessorID = req->fromProcessorID;
            req->fromProcessorID = cache->getCacheCPUid();
            req->toInterfaceID = -1;
            req->fromInterfaceID = -1;
            req->dirACK = true;

            // send the messages
            sendDirectoryMessage(req, lat);
            sendDirectoryMessage(ownershipReq, lat);

            writeTraceLine(cacheName,
                           "Accepting ownership, ACK and ownership request sent",
                           blk->owner,
                           blk->dirState,
                           tmpL1BlkAddr,
                           cache->getBlockSize(),
                           blk->presentFlags);

            return true;
        }
        break;

        default:
            cout << req->cmd.toString() << "\n";
            fatal("L1: cache access(), unknown request type");
    }

    return false;
}

template<class TagStore>
bool
StenstromProtocol<TagStore>::handleDirectoryResponse(MemReqPtr& req,
                                                     TagStore *tags){

    if(req->dirNACK){

        if(req->cmd == DirOwnerTransfer){

            req->dirNACK = false;
            req->flags &= ~SATISFIED;

            BlkType* tmpBlk = tags->findBlock(req);
        if(req->ownerWroteBack){
            assert(req->writeMiss);
            req->cmd = Read;

            writeTraceLine(cacheName,
                           "Owner Transfer NACK recieved, owner wrote back,"
                           " retransmitting as read",
                           req->owner,
                           DirNoState,
                           req->paddr,
                           cache->getBlockSize(),
                           NULL);
        }
        else if(tmpBlk != NULL &&
                (tmpBlk->dirState == DirOwnedExGR
                || tmpBlk->dirState == DirOwnedNonExGR)){

            writeTraceLine(cacheName,
                           "Owner Transfer NACK recieved, "
                           "we have become the owner",
                           tmpBlk->owner,
                           tmpBlk->dirState,
                           req->paddr,
                           cache->getBlockSize(),
                           NULL);

            // we have become the owner, return request to processor
            if(req->mshr == NULL){
                assert(req->completionEvent != NULL);
                cache->respond(req, curTick + cache->getHitLatency());
                return true;
            }
            else{
                assert(req->mshr != NULL);
                cache->missQueueHandleResponse(req,
                        curTick + cache->getHitLatency());
                return true;
            }
        }
        else{
            writeTraceLine(cacheName,
                           "Owner Transfer NACK recieved, "
                           "retransmitting to L2 cache",
                           req->owner,
                           DirNoState,
                           req->paddr,
                           cache->getBlockSize(),
                           NULL);
        }

        req->toProcessorID = -1;
        req->fromProcessorID = cache->getCacheCPUid();
        req->fromInterfaceID = -1;
        req->toInterfaceID = -1;

        sendDirectoryMessage(req, cache->getHitLatency());

        return true;
    }
    else if(req->cmd == DirRedirectRead){
        req->dirNACK = false;
        req->flags &= ~SATISFIED;

        writeTraceLine(cacheName,
                       "Redirected Read NACK recieved, "
                       "retransmitting to L2 cache",
                       req->owner,
                       DirNoState,
                       req->paddr,
                       cache->getBlockSize(),
                       NULL);

        req->toProcessorID = -1;
        req->fromProcessorID = cache->getCacheCPUid();
        req->fromInterfaceID = -1;
        req->toInterfaceID = -1;

        sendDirectoryMessage(req, cache->getHitLatency());

        return true;
    }
    else if(req->cmd == DirOwnerWriteback){

        assert(req->presentFlags == NULL);

        Addr tmpBlkAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);
        assert(outstandingWritebackWSAddrs.find(tmpBlkAddr)
                != outstandingWritebackWSAddrs.end());

        outstandingWritebackWSAddrs[tmpBlkAddr][req->fromProcessorID]
                = false;
        bool* tmpPresentFlags = outstandingWritebackWSAddrs[tmpBlkAddr];

        int presentCount = 0;
        for(int i=0;i<cache->cpuCount;i++){
            if(tmpPresentFlags[i]) presentCount++;
        }

        req->dirNACK = false;

        if(presentCount == 0){
            // no other sharers left, send to L2
            req->toProcessorID = -1;
            req->fromProcessorID = cache->getCacheCPUid();
            req->toInterfaceID = -1;
            req->fromInterfaceID = -1;

            req->cmd = DirWriteback;

            sendDirectoryMessage(req, cache->getHitLatency());

            // writeback has been handled, remove it
            outstandingWritebackWSAddrs.erase(
                    outstandingWritebackWSAddrs.find(tmpBlkAddr));

            writeTraceLine(cacheName,
                           "No sharers left, doing normal writeback",
                           req->owner,
                           DirNoState,
                           req->paddr,
                           cache->getBlockSize(),
                        tmpPresentFlags);

            return true;
        }

        else{
            int nextSharer = -1;
            for(int i=0;i<cache->cpuCount;i++){
                if(tmpPresentFlags[i]){
                    nextSharer = i;
                    break;
                }
            }
            assert(nextSharer != -1);

            req->toProcessorID = nextSharer;
            req->fromProcessorID = cache->getCacheCPUid();
            req->toInterfaceID = -1;
            req->fromInterfaceID = -1;

            writeTraceLine(cacheName,
                        "Attempting to transfer ownership to "
                        "different sharer",
                        req->owner,
                        DirNoState,
                        req->paddr,
                        cache->getBlockSize(),
                        tmpPresentFlags);

            sendDirectoryMessage(req, cache->getHitLatency());

            return true;
        }
    }
    else if(req->cmd == Read){
        assert(req->fromProcessorID == -1);

        if(req->writeMiss){
            assert(req->writeMiss);

            // change the request to an owner transfer and resend
            req->cmd = DirOwnerTransfer;
            req->dirNACK = false;
            req->fromProcessorID = cache->getCacheCPUid();
            req->toProcessorID = -1;
            req->toInterfaceID = -1;

            writeTraceLine(cacheName,
                        "Write miss, recieved NACK, retransmitting",
                        -1,
                        DirNoState,
                        req->paddr,
                        cache->getBlockSize(),
                        NULL);


            //TODO: have a different retransmit delay?
            sendDirectoryMessage(req, cache->getHitLatency());
            return true;
        }
        else{
            fatal("NACK on read miss not implemented");
        }
    }
    else{
        fatal("Recieved NACK, request type not implemented");
    }
}
else if(req->cmd == DirOwnerWriteback){

    assert(!req->isDirectoryNACK());

    if(req->isDirectoryACK()){
        assert(req->presentFlags == NULL);

        writeTraceLine(cacheName,
                       "Owner with sharers, recieved ACK",
                       req->owner,
                       DirNoState,
                       req->paddr,
                       cache->getBlockSize(),
                       NULL);

        return true;
    }
    else{
        // recieved a request to take over ownership of this block
        assert(req->owner != cache->getCacheCPUid());
        cache->access(req);
        return true;
    }

}
else if(req->cmd == DirSharerWriteback){

    // a sharer has written back a block owned by this cache
    BlkType* tmpBlk = tags->findBlock(req);
    if(tmpBlk == NULL){
        // we are in the process of writing back the block
        Addr tmpBlkAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);
        if(outstandingWritebackWSAddrs.find(tmpBlkAddr)
           == outstandingWritebackWSAddrs.end()){
            // we have written back the block
            // or the block has been given a new owner
            // send NACK to L2 and let it handle it
            sendNACK(req,
                     cache->getHitLatency(),
                     -1,
                     cache->getCacheCPUid());
            return true;
        }

        assert(outstandingWritebackWSAddrs.find(tmpBlkAddr)
                != outstandingWritebackWSAddrs.end());
        outstandingWritebackWSAddrs[tmpBlkAddr][req->replacedByID] = false;

        writeTraceLine(cacheName,
                       "Sharer writeback recieved to block that is "
                       "being written back",
                       -1,
                       DirNoState,
                       req->paddr,
                       cache->getBlockSize(),
                       outstandingWritebackWSAddrs[tmpBlkAddr]);

        return true;
    }

    if(tmpBlk->dirState == DirInvalid){
        // we are in the middle of an owner transfer and
        // haven't recieved the data yet
        // make the L2 resend the request
        sendNACK(req, cache->getHitLatency(), -1, cache->getCacheCPUid());
        return true;
    }

    assert(tmpBlk != NULL);
    assert(tmpBlk->dirState == DirOwnedNonExGR
            || tmpBlk->dirState == DirOwnedExGR);
    assert(tmpBlk->presentFlags != NULL);

    assert(req->replacedByID >= 0);
    if(req->replacedByID == cache->cacheCpuID){
        // if a sharer and the owner writes back a block at the same time
        // this can happen, discard this update
        writeTraceLine(cacheName,
                       "Recieved sharer writeback from ourselves",
                       tmpBlk->owner,
                       tmpBlk->dirState,
                       req->paddr,
                       cache->getBlockSize(),
                       tmpBlk->presentFlags);
        return true;
    }

    tmpBlk->presentFlags[req->replacedByID] = false;

    int sharers = 0;
    for(int i=0;i<cache->cpuCount;i++){
        if(tmpBlk->presentFlags[i]) sharers++;
    }

    assert(sharers > 0);
    if(sharers == 1) tmpBlk->dirState = DirOwnedExGR;
    else tmpBlk->dirState = DirOwnedNonExGR;

    writeTraceLine(cacheName,
                   "Sharer writeback recieved and handled",
                   tmpBlk->owner,
                   tmpBlk->dirState,
                   req->paddr,
                   cache->getBlockSize(),
                   tmpBlk->presentFlags);

    return true;
}
else if(req->cmd == DirNewOwnerMulticast){

    BlkType* tmpBlk = tags->findBlock(req);

    if(tmpBlk == NULL){
        //we have written back this block and don't care who owns it
        writeTraceLine(cacheName,
                       "Owner info discarded, block written back",
                       -1,
                       DirNoState,
                       req->paddr,
                       cache->getBlockSize(),
                       NULL);
        return true;
    }

    assert(tmpBlk != NULL);
    assert(tmpBlk->dirState == DirInvalid);
    assert(tmpBlk->presentFlags == NULL);
    assert(req->owner >= 0);

    tmpBlk->owner = req->owner;

    writeTraceLine(cacheName,
                   "New owner info recieved and stored",
                   tmpBlk->owner,
                   tmpBlk->dirState,
                   req->paddr,
                   cache->getBlockSize(),
                   tmpBlk->presentFlags);

    return true;
}
else if(req->owner != -1
        && req->owner != cache->getCacheCPUid()
        && req->fromProcessorID == -1
        && req->cmd != DirOwnerTransfer){
    // This is an L1 cache, but is not the owner

    // redirected request to owner, must be a read
    assert(!req->writeMiss);

    setUpRedirectedRead(req, cache->getCacheCPUid(), req->owner);

    numRedirectedReads++;

    writeTraceLine(cacheName,
                   "Issuing Redirected Read (2)",
                   req->owner,
                   DirNoState,
                   req->paddr,
                   cache->getBlockSize(),
                   NULL);

    sendDirectoryMessage(req, cache->getHitLatency());

    return true;
}
else if(req->fromProcessorID != -1
        && req->owner == cache->getCacheCPUid()){
    // Case 1: request from a different L1 cache and we are the owner
    // Case 2: we have recieved the owner state at the end of
    //         a owner transfer request
    if(req->writeMiss){
        // this is a cache fill, let it through
    }
    else{
        assert(req->cmd == DirRedirectRead || req->cmd == DirOwnerTransfer);
        if(req->cmd == DirOwnerTransfer){
            // the block must be in the cache for this forwarding to work
            BlkType* tmpBlk = tags->findBlock(req);
            if(tmpBlk == NULL){
                // we have replaced this block, let it back in
                assert(req->mshr == NULL);
                return false;
            }
        }

        cache->access(req);
        return true;
    }
}
else if(req->cmd == DirRedirectRead
        && req->owner == cache->getCacheCPUid()
        && req->fromProcessorID == -1){

    if(req->mshr != NULL){
        // a mshr is allocated
        // let it go through and handle the fill normally
        return false;
    }
    else{
        // no MSHR is allocated because the block is allready in our cache

        // this code assumes that we have become the owner while the
        // Redirected Read  has been transported through the system
        BlkType* tmpBlk = tags->findBlock(req);
        assert(tmpBlk != NULL);
        if(tmpBlk->dirState == DirInvalid){
            // we have not become the owner yet, send nack to ourselves
            sendNACK(req,
                     cache->getHitLatency(),
                     cache->getCacheCPUid(),
                     cache->getCacheCPUid());
            return true;
        }
        assert(tmpBlk->dirState == DirOwnedExGR
               || tmpBlk->dirState == DirOwnedNonExGR);
        assert(tmpBlk->owner == cache->getCacheCPUid());

        writeTraceLine(cacheName,
                       "This cache is the owner, send response to CPU",
                       req->owner,
                       DirNoState,
                       req->paddr,
                       cache->getBlockSize(),
                       NULL);

        assert(req->completionEvent != NULL);
        cache->respond(req, curTick + cache->getHitLatency());

        return true;
    }
}

return false;
}
```

```
template<class TagStore>
bool
StenstromProtocol<TagStore>::handleDirectoryFill(MemReqPtr& req,
                                                  BlkType* blk,
                                                  MemReqList& writebacks,
                                                  TagStore* tags){

    // This is an L1 data cache
    if(req->cmd == DirOwnerTransfer){

        if(req->fromProcessorID == -1){

            int newOwner = -1;
            bool* oldFlags = NULL;

            if(outstandingWritebackWSAddrs.find(
               req->paddr & ~((Addr)cache->getBlockSize() - 1))
               != outstandingWritebackWSAddrs.end()){

                //remove this entry
                Addr tmpAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);
                oldFlags = outstandingWritebackWSAddrs[tmpAddr];
                outstandingWritebackWSAddrs.erase(
                        outstandingWritebackWSAddrs.find(tmpAddr));

                // no need to check for other sharers
                // (end of ownership replacement with sharers)
                newOwner = req->owner;
                req->toProcessorID = newOwner;
                req->fromProcessorID = cache->getCacheCPUid();
                req->presentFlags = oldFlags;
                req->owner = newOwner;

                writeTraceLine(cacheName,
                               "Transfering Owner State "
                               "(end of owner with sharers replacement)",
                               newOwner,
                               DirInvalid,
                               req->paddr,
                               cache->getBlockSize(),
                               oldFlags);

            }
            else{

                // Owner transfer recieved from L2, we must be the owner
                if(blk == NULL){
                    // The block hasn't been delivered to us yet
                    // or we have written it back, send NACK
                    sendNACK(req,
                             cache->getHitLatency(),
                             -1,
                             cache->getCacheCPUid());
                    return true;
                }

                assert(blk->dirState == DirOwnedExGR
                       || blk->dirState == DirOwnedNonExGR);
                assert(blk->presentFlags != NULL);

                newOwner = req->owner;
                blk->presentFlags[newOwner] = true;
                oldFlags = blk->presentFlags;

                // update local block
                blk->presentFlags = NULL;
                blk->dirState = DirInvalid;
                blk->status &= ~BlkDirty;
                blk->owner = newOwner;

                writeTraceLine(cacheName,
                               "Transfering Owner State",
                               blk->owner,
                               blk->dirState,
                               req->paddr,
                               cache->getBlockSize(),
                               oldFlags);

                // send owner state to new owner
                req->toProcessorID = newOwner;
                req->fromProcessorID = cache->getCacheCPUid();
                req->presentFlags = oldFlags;
                req->owner = newOwner;
            }

            assert(oldFlags != NULL);
            assert(newOwner != -1);
            for(int i=0;i<cache->cpuCount;i++){
                if(oldFlags[i]
                   && i != cache->getCacheCPUid()
                   && i != newOwner){
                    // send request to all other sharers
                    MemReqPtr tmpReq = buildReqCopy(req,
                                                    cache->cpuCount,
                                                    DirNewOwnerMulticast);
                    assert(tmpReq->cmd == DirNewOwnerMulticast);
                    tmpReq->toProcessorID = i;
                    tmpReq->presentFlags = NULL;

                    writeTraceLine(cacheName,
                                   "Informing sharer of new owner",
                                   (blk != NULL) ? blk->owner : -1,
                                   (blk != NULL) ? blk->dirState : DirNoState,
                                   req->paddr,
                                   cache->getBlockSize(),
                                   oldFlags);

                    sendDirectoryMessage(tmpReq, cache->getHitLatency());
                }
            }

            // send request to new owner
            sendDirectoryMessage(req, cache->getHitLatency());

            return true;
        }
        else if(req->writeMiss){

            // We have recieved ownership of a block because of a L1 write miss
            assert(cache->getCacheCPUid() == req->owner);

            int sharerCount = 0;
            for(int i=0;i<cache->cpuCount;i++){
                if(req->presentFlags[i]) sharerCount++;
            }

            // we must get the block, because the the previous call is bypassed
            CacheBlk::State old_state = (blk) ? blk->status : 0;
            blk = tags->handleFill(blk,
                                   req->mshr,
                                   cache->getNewCoherenceState(req, old_state),
                                   writebacks);
            blk->owner = cache->getCacheCPUid();
            blk->presentFlags = req->presentFlags;

            if(sharerCount > 1) blk->dirState = DirOwnedNonExGR;
            else blk->dirState = DirOwnedExGR;

            // remove the reference to these flags, so they are
            // not deleted together with the request
            req->presentFlags = NULL;

            writeTraceLine(cacheName,
                           "Recieved owner state (write miss)",
                           blk->owner,
                           blk->dirState,
                           req->paddr,
                           cache->getBlockSize(),
                           blk->presentFlags);

            // Send ACK to the L2 cache
            MemReqPtr tmpReq = buildReqCopy(req,
                                            cache->cpuCount,
                                            DirOwnerTransfer);
            setUpACK(tmpReq, -1, cache->getCacheCPUid());
            sendDirectoryMessage(tmpReq, cache->getHitLatency());

        }
        else{
            // the block was replaced in the middle of an owner transfer
            // put it back in and update the stats
            assert(req->mshr == NULL);
            assert(blk == NULL);

            blk = tags->handleFill(blk,
                                   req,
                                   BlkValid | BlkWritable,
                                   writebacks);

            blk->owner = cache->getCacheCPUid();
            blk->presentFlags = req->presentFlags;
            blk->presentFlags[cache->getCacheCPUid()] = true;
            blk->dirState = DirOwnedNonExGR;

            // remove the reference to these flags, so they are
            // not deleted together with the request
            req->presentFlags = NULL;

            writeTraceLine(cacheName,
                           "Owner transfer complete, needed block was replaced",
                           blk->owner,
                           blk->dirState,
                           req->paddr,
                        cache->getBlockSize(),
                        blk->presentFlags);

            // check that the cache fill worked
            BlkType* checkBlk = tags->findBlock(req->paddr, req->asid);
            assert(checkBlk != NULL);

            // Send ACK to the L2 cache
            MemReqPtr tmpReq = buildReqCopy(req,
                                            cache->cpuCount,
                                            DirOwnerTransfer);
            setUpACK(tmpReq, -1, cache->getCacheCPUid());
            sendDirectoryMessage(tmpReq, cache->getHitLatency());

            return true;
        }

    }
    else if(req->cmd == DirRedirectRead){
        //response from a redirected read recieved

        if(blk->owner == cache->getCacheCPUid()
           || req->owner == cache->getCacheCPUid()){
            // CASE 1: we have become the owner while
            //         the redirected read was in transit
            // CASE 2: the previous owner wrote the line back while the RR was
            //         in transit and we are the new owner

            // the block might be brought into the cache so it might not have a state yet
        if(blk->dirState == DirNoState){

            assert(req->presentFlags == NULL);
            blk->presentFlags = new bool[cache->cpuCount];
        for(int i=0;i<cache->cpuCount;i++){
                    blk->presentFlags[i] = false;
                }
        blk->presentFlags[cache->getCacheCPUid()] = true;
        blk->dirState = DirOwnedExGR;
                blk->owner = cache->getCacheCPUid(); // needed in case 2
        }
        assert(blk->dirState == DirOwnedExGR
                  || blk->dirState == DirOwnedNonExGR);
            assert(blk->presentFlags != NULL);
            assert(req->presentFlags == NULL);

            writeTraceLine(cacheName,
                        "Redirected Read Response Recieved,"
                        " we have become the owner",
                        blk->owner,
                        blk->dirState,
                        req->paddr,
                        cache->getBlockSize(),
                        blk->presentFlags);
        }
        else{
            assert(blk->owner != cache->getCacheCPUid());
            assert(blk->presentFlags == NULL);

            blk->dirState = DirInvalid;
            blk->owner = req->owner;
            writeTraceLine(cacheName,
                                   "Redirected Read Response Recieved",
                                   blk->owner,
                                   blk->dirState,
                                   req->paddr,
                                   cache->getBlockSize(),
                                   blk->presentFlags);
        }

        if(req->mshr == NULL){

            assert(req->completionEvent != NULL);
            cache->respond(req, curTick);
            return true;
        }
    }
    else if(req->cmd == Read
            && (blk->dirState == DirOwnedExGR
            || blk->dirState == DirOwnedNonExGR)){
        // command is a read or write and we are the owner
        // let it go through
        assert(req->mshr != NULL);
        return false;
    }
    else if(req->fromProcessorID == -1
            && req->owner == cache->getCacheCPUid()){

        assert(blk->presentFlags == NULL);
        assert(blk->dirState != DirOwnedExGR);
        assert(blk->dirState != DirOwnedNonExGR);

        blk->owner = req->owner;
        blk->dirState = DirOwnedExGR;

        if(blk->presentFlags == NULL){
            blk->presentFlags = new bool[cache->cpuCount];
        }

        for(int i=0;i<cache->cpuCount;i++){
            blk->presentFlags[i] = false;
        }
        blk->presentFlags[cache->getCacheCPUid()] = true;
    }
    else{
        fatal("response type not implemented (handleResponse())");
    }

    return false;
}

template<class TagStore>
bool
StenstromProtocol<TagStore>::doDirectoryWriteback(MemReqPtr& req){

    if(req->cmd == DirWriteback){
        // Directory writeback of non-modified block
        // latency has allready been counted
        sendDirectoryMessage(req, 0);
        return true;
    }
    else if(req->cmd == DirOwnerWriteback){
        assert(req->presentFlags != NULL);

        // set our present flag to false
        req->presentFlags[cache->getCacheCPUid()] = false;

        int foundCount = 0;
        int newOwner = -1;
        for(int i=0;i<cache->cpuCount;i++){
            if(i != cache->getCacheCPUid() && req->presentFlags[i]){
                newOwner = i;
                foundCount++;
                break;
            }
        }
        assert(foundCount == 1);
        assert(newOwner >= 0);

        //update the request stats
        req->toProcessorID = newOwner;
        req->fromProcessorID = cache->getCacheCPUid();
        req->toInterfaceID = -1;
        req->owner = cache->getCacheCPUid();

        bool* tmpFlags = req->presentFlags;
        req->presentFlags = NULL;

        Addr tmpBlkAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);
        if(outstandingWritebackWSAddrs.find(tmpBlkAddr)
           == outstandingWritebackWSAddrs.end()){
            // there is no outstanding writeback to this address
            outstandingWritebackWSAddrs[tmpBlkAddr] = tmpFlags;
        }
        else{
            fatal("We are writing back the same block twice,"
                  " this is not nice...");
        }

        numOwnerWritebacks++;

        writeTraceLine(cacheName,
                       "Replacing owned block with sharers",
                       cache->getCacheCPUid(),
                       DirInvalid,
                       req->paddr,
                       cache->getBlockSize(),
                       tmpFlags);

        //forward the request to the new owner
        sendDirectoryMessage(req, 0);

        return true;
    }
    else if(req->cmd == DirSharerWriteback){

        // send it to the L2 cache, it will inform the owner
        req->toProcessorID = -1;
        req->fromProcessorID = cache->getCacheCPUid();
        req->toInterfaceID = -1;
        req->fromInterfaceID = -1;

        numSharerWritebacks++;
        writeTraceLine(cacheName,
                       "Replacing not owned block",
                       -1,
                       DirInvalid,
                       req->paddr,
                       cache->getBlockSize(),
                       req->presentFlags);

        //forward the request to the L2 cache
        sendDirectoryMessage(req, 0);

        return true;
    }
    return false;
}

template<class TagStore>
MemAccessResult
StenstromProtocol<TagStore>::handleL1DirectoryMiss(MemReqPtr& req){

    if(req->cmd == Soft_Prefetch){
        return MA_CACHE_MISS;
    }
    else if(req->cmd == DirRedirectRead){
        // Miss on a redirected read, we have written back the block, send NACK
        sendNACK(req,
                 cache->getHitLatency(),
                 req->fromProcessorID,
                 cache->getCacheCPUid());
        return MA_CACHE_MISS;
    }
    else if(req->cmd == DirOwnerWriteback){
        // Miss on a owner transfer request,
        // we have written back the block, send NACK
        sendNACK(req,
                 cache->getHitLatency(),
                 req->fromProcessorID,
                 cache->getCacheCPUid());
        return MA_CACHE_MISS;
    }
    else if(req->cmd == Read || req->cmd == Write){
        Addr tmpAddr = req->paddr & ~((Addr)cache->getBlockSize() - 1);
        if(outstandingWritebackWSAddrs.find(tmpAddr)
           != outstandingWritebackWSAddrs.end()){
            // this cache still has updated state for this cache
            // respond to request
            cache->respond(req, curTick + cache->getHitLatency());
            return MA_HIT;
        }
    }

    return BA_NO_RESULT;
}


/* Private helper methods */

template<class TagStore>
void
StenstromProtocol<TagStore>::setUpRedirectedRead(MemReqPtr& req,
                                                            int fromProcessorID,
                                                            int toProcessorID){
    req->oldCmd = req->cmd;
    req->cmd = DirRedirectRead;
    req->fromProcessorID = fromProcessorID;
    req->toProcessorID = toProcessorID;
    //must be updated if the req did not come from L2 just now
    req->owner = toProcessorID;
    req->toInterfaceID = -1;
}

template<class TagStore>
void
StenstromProtocol<TagStore>::setUpRedirectedReadReply(MemReqPtr& req,
                                                              int fromProcessorID,
                                                              int toProcessorID){
    req->toInterfaceID = -1;
    req->fromProcessorID = fromProcessorID;
    req->toProcessorID = toProcessorID;
    //only owners reply to redirected reads
    req->owner = fromProcessorID;
}

template<class TagStore>
void
StenstromProtocol<TagStore>::setUpOwnerTransferInL2(MemReqPtr& req,
                                                            int oldOwner,
                                                            int newOwner){
    req->cmd = DirOwnerTransfer;
    req->toProcessorID = oldOwner;
    req->owner =  newOwner; //blk->owner;
    req->fromProcessorID = -1;
    req->toInterfaceID = -1;
}

template<class TagStore>
void
StenstromProtocol<TagStore>::setUpACK(MemReqPtr& req, int toID, int fromID){
    req->toProcessorID = toID;
    req->fromProcessorID = fromID;
    req->toInterfaceID = -1;
    req->fromInterfaceID = -1;
    req->presentFlags = NULL;
    req->owner = -1;
    req->dirACK = true;
}

/* The rest of this file consists of template definitions */

#ifndef DOXYGEN_SHOULD_SKIP_THIS

// Include config files
// Must be included first to determine which caches we want
#include "mem/config/cache.hh"
#include "mem/config/compression.hh"


// Tag Templates
#if defined(USE_CACHE_LRU)
#include "mem/cache/tags/lru.hh"
#endif
```

```
#if defined(USE_CACHE_FALRU)
#include "mem/cache/tags/fa_lru.hh"
#endif

#if defined(USE_CACHE_IIC)
#include "mem/cache/tags/iic.hh"
#endif

#if defined(USE_CACHE_SPLIT)
#include "mem/cache/tags/split.hh"
#endif

#if defined(USE_CACHE_SPLIT_LIFO)
#include "mem/cache/tags/split_lifo.hh"
#endif

// Compression Templates
#include "base/compression/null_compression.hh"
#if defined(USE_LZSS_COMPRESSION)
#include "base/compression/lzss_compression.hh"
#endif

#if defined(USE_CACHE_FALRU)
    template class StenstromProtocol<CacheTags<FALRU, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class StenstromProtocol<CacheTags<FALRU, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_IIC)
    template class StenstromProtocol<CacheTags<IIC, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class StenstromProtocol<CacheTags<IIC, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_LRU)
    template class StenstromProtocol<CacheTags<LRU, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class StenstromProtocol<CacheTags<LRU, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_SPLIT)
    template class StenstromProtocol<CacheTags<Split, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class StenstromProtocol<CacheTags<Split, LZSSCompression> >;
#endif
#endif

#if defined(USE_CACHE_SPLIT_LIFO)
    template class StenstromProtocol<CacheTags<SplitLIFO, NullCompression> >;
#if defined(USE_LZSS_COMPRESSION)
    template class StenstromProtocol<CacheTags<SplitLIFO, LZSSCompression> >;
#endif
#endif

#endif // DOXYGEN_SHOULD_SKIP_THIS
```

Appendix D

Simulator Configuration Scripts

D.1 run.py

```
from m5 import *
import Splash2
import TestPrograms
import Spec2000
import workloads
from DetailedConfig import *

################################################################################
# Constants
################################################################################

L2_BANK_COUNT = 4
all_protocols = ['none', 'msi', 'mesi', 'mosi', 'moesi', 'stenstrom']
snoop_protocols = ['msi', 'mesi', 'mosi', 'moesi']
directory_protocols = ['stenstrom']

################################################################################
# Check command line options
################################################################################

if env['PROTOCOL'] not in all_protocols:
  panic('No/Invalid cache coherence protocol specified!')

if 'BENCHMARK' not in env:
    panic("The BENCHMARK environment variable must be set!\ne.g. \
     -EBENCHMARK=Cholesky\n")

# Multi-programmed workloads (numbered 1 to N) reads fast-forward cycles
# from a config file
# Splash benchmarks can read from config file
if not ((env['BENCHMARK'].isdigit()) or (env['BENCHMARK']
        in Splash2.benchmarkNames)):
    if 'FASTFORWARDTICKS' not in env:
        panic("The FASTFORWARDTICKS environment variable must be set!\n\
        e.g. -EFASTFORWARDTICKS=10000\n")

if 'SIMULATETICKS' not in env and 'SIMINSTS' not in env \
and 'ISEXPERIMENT' not in env:
    panic("One of the SIMULATETICKS/SIMINSTS/ISEXPERIMENT environment \
    variable must be set!\ne.g. -ESIMULATETICKS=10000\n")
```

```
if 'INTERCONNECT' not in env:
    panic("The INTERCONNECT environment variable must be set!\ne.g. \
    -EINTERCONNECT=bus\n")

if 'STATSFILE' not in env:
    panic('No statistics file name given! (-ESTATSFILE=foobar.txt)')

coherenceTrace = False
coherenceTraceStart = 0
if 'TRACE' in env:
    if env['PROTOCOL'] not in directory_protocols:
        panic('Tracing is only supported for directory protocols');
    coherenceTrace = True
    coherenceTraceStart = env['TRACE']
    print >>sys.stderr, 'warning: Protocol tracing is turned on!'

inDumpInterval = 0
if 'DUMPCCSTATS' in env:
    inDumpInterval = int(env['DUMPCCSTATS'])

icProfileStart = -1
if 'PROFILEIC' in env:
    icProfileStart = int(env['PROFILEIC'])

progressInterval = 0
if 'PROGRESS' in env:
    progressInterval = int(env['PROGRESS'])

# MSHR parameters
l1mshrTargets = -1
l1mshrsData = -1
if 'MSHRSL1D' in env and 'MSHRL1TARGETS' in env:
    l1mshrsData = int(env['MSHRSL1D'])
    l1mshrTargets = int(env['MSHRL1TARGETS'])

l1mshrsInst = -1
if 'MSHRSL1I' in env and 'MSHRL1TARGETS' in env:
    l1mshrsInst = int(env['MSHRSL1I'])
    l1mshrTargets = int(env['MSHRL1TARGETS'])

l2mshrTargets = -1
l2mshrs = -1
if 'MSHRSL2' in env and 'MSHRL2TARGETS' in env:
    l2mshrs = int(env['MSHRSL2'])
    l2mshrTargets = int(env['MSHRL2TARGETS'])

###############################################################################
# Root, CPUs and L1 caches
###############################################################################

root = DetailedStandAlone()
if progressInterval > 0:
    root.progress_interval = progressInterval

# Create CPUs
BaseCPU.workload = Parent.workload
root.simpleCPU = [ CPU(defer_registration=True, cpu_id=i)
                   for i in xrange(int(env['NP'])) ]
root.detailedCPU = [ DetailedCPU(defer_registration=True, cpu_id=i)
                     for i in xrange(int(env['NP'])) ]
```

```
# Create L1 caches
if env['INTERCONNECT'] == 'bus':
    root.L1dcaches = [ DL1(out_bus=Parent.interconnect)
                       for i in xrange(int(env['NP'])) ]
    root.L1icaches = [ IL1(out_bus=Parent.interconnect)
                       for i in xrange(int(env['NP'])) ]
else:
    root.L1dcaches = [ DL1(out_interconnect=Parent.interconnect)
                       for i in xrange(int(env['NP'])) ]
    root.L1icaches = [ IL1(out_interconnect=Parent.interconnect)
                       for i in xrange(int(env['NP'])) ]

if env['PROTOCOL'] != 'none':
    if env['PROTOCOL'] in snoop_protocols:
        for cache in root.L1dcaches:
            cache.protocol = CoherenceProtocol(protocol=env['PROTOCOL'])
    elif env['PROTOCOL'] in directory_protocols:
        for cache in root.L1dcaches:
            cache.dirProtocolName = env['PROTOCOL']
            cache.dirProtocolDoTrace = coherenceTrace
            if coherenceTraceStart != 0:
                cache.dirProtocolTraceStart = coherenceTraceStart
            cache.dirProtocolDumpInterval = inDumpInterval

# Connect L1 caches to CPUs
for i in xrange(int(env['NP'])):
    root.simpleCPU[i].dcache = root.L1dcaches[i]
    root.simpleCPU[i].icache = root.L1icaches[i]
    root.detailedCPU[i].dcache = root.L1dcaches[i]
    root.detailedCPU[i].icache = root.L1icaches[i]
    root.L1dcaches[i].cpu_id = i
    root.L1icaches[i].cpu_id = i

if l1mshrsData != -1:
    for l1 in root.L1dcaches:
        l1.mshrs = l1mshrsData
        l1.tgts_per_mshr = l1mshrTargets

if l1mshrsInst != -1:
    for l1 in root.L1icaches:
        l1.mshrs = l1mshrsInst
        l1.tgts_per_mshr = l1mshrTargets

###############################################################################
# Fast-forwarding
###############################################################################


if env['BENCHMARK'] in Splash2.benchmarkNames:
    # Scientific workloads
    root.sampler = Sampler()
    root.sampler.phase0_cpus = Parent.simpleCPU
    root.sampler.phase1_cpus = Parent.detailedCPU
    if 'ISEXPERIMENT' in env and env['PROTOCOL'] in directory_protocols:
        root.sampler.periods = [0, 50000000000] # sampler is not used
        for cpu in root.detailedCPU:
            cpu.max_insts_any_thread = \
                Splash2.instructions[int(env['NP'])][env['BENCHMARK']]
    elif 'SIMINSTS' in env:
        root.sampler.periods = [0, 50000000000] # sampler is not used
        for cpu in root.detailedCPU:
            cpu.max_insts_any_thread = int(env['SIMINSTS'])
    elif 'FASTFORWARDTICKS' not in env:
        fwticks, simticks = Splash2.fastforward[env['BENCHMARK']]
        root.sampler.periods = [fwticks, simticks]
    else:
        root.sampler.periods = [env['FASTFORWARDTICKS'],
                                int(env['SIMULATETICKS'])]
    root.setCPU(root.simpleCPU)
elif not env['BENCHMARK'].isdigit():
    # Simulator test workloads
    root.sampler = Sampler()
    root.sampler.phase0_cpus = Parent.simpleCPU
    root.sampler.phase1_cpus = Parent.detailedCPU
    root.sampler.periods = [int(env['FASTFORWARDTICKS']),
                            int(env['SIMULATETICKS'])]
    root.setCPU(root.simpleCPU)
else:
    # Multi-programmed workload
    root.samplers = [ Sampler() for i in xrange(int(env['NP'])) ]

    fwCycles = \
        workloads.workloads[int(env['NP'])][int(env['BENCHMARK'])][1]
    simulateCycles = int(env['SIMULATETICKS'])
    simulateStart = max(fwCycles)

    for i in xrange(int(env['NP'])):
        root.samplers[i].phase0_cpus = [Parent.simpleCPU[i]]
        root.samplers[i].phase1_cpus = [Parent.detailedCPU[i]]
        root.samplers[i].periods = [fwCycles[i], simulateCycles
                                    + (simulateStart - fwCycles[i])]

    root.setCPU(root.simpleCPU)

###############################################################################
# Interconnect and L2 caches
###############################################################################

if env['BENCHMARK'] in Splash2.benchmarkNames:
    BaseCache.multiprog_workload = False
else:
    BaseCache.multiprog_workload = True

if env['BENCHMARK'] in Splash2.benchmarkNames and 'FASTFORWARDTICKS' not in env:
    if 'PROFILEIC' in env:
        print >>sys.stderr, "warning: Production workload, \
                            ignoring user supplied profile start"
    icProfileStart = 0 #Splash2.fastforward[env['BENCHMARK']][0]

if env['BENCHMARK'].isdigit():
    if 'PROFILEIC' in env:
        print >>sys.stderr, "warning: Production workload, \
                             ignoring user supplied profile start"
    fwCycles = workloads.workloads[int(env['NP'])][int(env['BENCHMARK'])][1]
    icProfileStart = max(fwCycles)

moduloAddr = False
if env['BENCHMARK'] in Splash2.benchmarkNames:
    moduloAddr = True
```

```
Interconnect.cpu_count = int(env['NP'])
root.setInterconnect(env['INTERCONNECT'],
                     L2_BANK_COUNT,
                     icProfileStart,
                     moduloAddr)

root.setL2Banks()
if env['PROTOCOL'] in directory_protocols:
    for bank in root.l2:
        bank.dirProtocolName = env['PROTOCOL']
        bank.dirProtocolDoTrace = coherenceTrace
        if coherenceTraceStart != 0:
            bank.dirProtocolTraceStart = coherenceTraceStart

if l2mshrs != -1:
    for bank in root.l2:
        bank.mshrs = l2mshrs
        bank.tgts_per_mshr = l2mshrTargets

###############################################################################
# Workloads
###############################################################################

# Storage for multiprogrammed workloads
prog = []

###############################################################################
# SPLASH-2
###############################################################################

if env['BENCHMARK'] == 'Cholesky':
    root.workload = Splash2.Cholesky()
elif env['BENCHMARK'] == 'FFT':
    root.workload = Splash2.FFT()
elif env['BENCHMARK'] == 'LUContig':
    root.workload = Splash2.LU_contig()
elif env['BENCHMARK'] == 'LUNoncontig':
    root.workload = Splash2.LU_noncontig()
elif env['BENCHMARK'] == 'Radix':
    root.workload = Splash2.Radix()
elif env['BENCHMARK'] == 'Barnes':
    root.workload = Splash2.Barnes()
elif env['BENCHMARK'] == 'FMM':
    root.workload = Splash2.FMM()
elif env['BENCHMARK'] == 'OceanContig':
    root.workload = Splash2.Ocean_contig()
elif env['BENCHMARK'] == 'OceanNoncontig':
    root.workload = Splash2.Ocean_noncontig()
elif env['BENCHMARK'] == 'Raytrace':
    root.workload = Splash2.Raytrace()
elif env['BENCHMARK'] == 'WaterNSquared':
    root.workload = Splash2.Water_nsquared()
elif env['BENCHMARK'] == 'WaterSpatial':
    root.workload = Splash2.Water_spatial()

###############################################################################
# SPEC 2000
###############################################################################

elif env['BENCHMARK'] == 'gzip':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.GzipSource())
elif env['BENCHMARK'] == 'vpr':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.VprPlace())
elif env['BENCHMARK'] == 'gcc':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Gcc166())
elif env['BENCHMARK'] == 'mcf':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Mcf())
elif env['BENCHMARK'] == 'crafty':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Crafty())
elif env['BENCHMARK'] == 'parser':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Parser())
elif env['BENCHMARK'] == 'eon':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Eon1())
elif env['BENCHMARK'] == 'perlbmk':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Perlbmk1())
elif env['BENCHMARK'] == 'gap':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Gap())
elif env['BENCHMARK'] == 'vortex1':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Vortex1())
elif env['BENCHMARK'] == 'bzip':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Bzip2Source())
elif env['BENCHMARK'] == 'twolf':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Twolf())
elif env['BENCHMARK'] == 'wupwise':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Wupwise())
elif env['BENCHMARK'] == 'swim':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Swim())
elif env['BENCHMARK'] == 'mgrid':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Mgrid())
elif env['BENCHMARK'] == 'applu':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Applu())
elif env['BENCHMARK'] == 'mesa':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Mesa())
elif env['BENCHMARK'] == 'galgel':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Galgel())
elif env['BENCHMARK'] == 'art':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Art1())
elif env['BENCHMARK'] == 'equake':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Equake())
elif env['BENCHMARK'] == 'facerec':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Facerec())
```

```
elif env['BENCHMARK'] == 'ammp':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Ammp())
elif env['BENCHMARK'] == 'lucas':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Lucas())
elif env['BENCHMARK'] == 'fma3d':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Fma3d())
elif env['BENCHMARK'] == 'sixtrack':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Sixtrack())
elif env['BENCHMARK'] == 'apsi':
    for i in range(int(env['NP'])):
        prog.append(Spec2000.Apsi())

################################################################################
# Multi-programmed workloads
################################################################################

elif env['BENCHMARK'].isdigit():
    prog = Spec2000.createWorkload(
              workloads.workloads[int(env['NP'])][int(env['BENCHMARK'])][0])

################################################################################
# Testprograms
################################################################################

elif env['BENCHMARK'] == 'hello':
    root.workload = TestPrograms.HelloWorld()

else:
    panic("The BENCHMARK environment variable was set to something improper\n")

# Create multi-programmed workloads
if prog != []:
    for i in range(int(env['NP'])):
        root.simpleCPU[i].workload = prog[i]
        root.detailedCPU[i].workload = prog[i]

################################################################################
# Statistics
################################################################################

root.stats = Statistics(text_file=env['STATSFILE'])
```

D.2 DetailedConfig.py

```
from m5 import *
from MemConfig import *
from FuncUnitConfig import *

##############################################################################
# Branch Predictor
##############################################################################

class DefaultBranchPred(BranchPred):
    pred_class = 'hybrid'
    local_hist_regs = '2ki'
    local_hist_bits = 11
    local_index_bits = 11
    local_xor = False
    global_hist_bits = 13
    global_index_bits = 13
    global_xor = False
    choice_index_bits = 13
    choice_xor = False
    ras_size = 16
    btb_size = '2ki'
    btb_assoc = 4

##############################################################################
# CPUs
##############################################################################

class DetailedCPU(FullCPU):

    iq = StandardIQ(size = 64, caps = [0, 0, 0, 0])
    iq_comm_latency = 1
    fupools = DefaultFUP()
    lsq_size = 32
    rob_size = 128
    rob_caps = [0, 0, 0, 0]
    storebuffer_size = 32
    width = 8
    issue_bandwidth = [8, 8]
    prioritized_issue = False
    thread_weights = [1, 1, 1, 1]
    dispatch_to_issue = 1
    decode_to_dispatch = 10
    mispred_recover = 3
    fetch_branches = 3
    ifq_size = 32
    num_icache_ports = 1
    branch_pred = DefaultBranchPred()

    def setCache(self, dcache, icache):
        self.dcache = dcache
        self.icache = icache

class CPU(SimpleCPU):

    def setCache(self, dcache, icache):
        self.dcache = dcache
        self.icache = icache
```

```
###############################################################################
# Root
###############################################################################

class DetailedStandAlone(Root):

    #clock = '3Hz'
    clock = '3200MHz'
    toMemBus = ToMemBus()
    ram = SDRAM(in_bus=Parent.toMemBus)
    l2 = []

    def setCPU(self, inCPU):
        self.cpu = inCPU

    #def setNumCPUs(self, numCPUs):
        #self.interconnect.L1CacheCount = (numCPUs*2)

    def setInterconnect(self, optionString, L2BankCount, profileStart, moduloAddr)
        :
        if optionString == 'bus':
            self.interconnect = ToL2Bus()
            self.createL2(True, L2BankCount, moduloAddr)
        elif optionString == 'myBus':
            self.interconnect = InterconnectBus()
            self.createL2(False, L2BankCount, moduloAddr)
        elif optionString == 'crossbar':
            self.interconnect = InterconnectCrossbar()
            self.createL2(False, L2BankCount, moduloAddr)
        elif optionString == 'ideal':
            self.interconnect = InterconnectIdeal()
            self.createL2(False, L2BankCount, moduloAddr)
        elif optionString == 'idealwdelay':
            self.interconnect = InterconnectIdealWithDelay()
            self.createL2(False, L2BankCount, moduloAddr)
        elif optionString == 'pipeBus':
            self.interconnect = PipelinedBus()
            self.createL2(False, L2BankCount, moduloAddr)
        elif optionString == 'butterfly':
            self.interconnect = InterconnectButterfly()
            self.createL2(False, L2BankCount, moduloAddr)
        else:
            panic('Unknown interconnect selected')

        if profileStart != -1 and optionString != 'bus':
            self.interconnectProfiler = InterconnectProfile()
            self.interconnectProfiler.traceSends = True
            self.interconnectProfiler.traceChannelUtil = True
            self.interconnectProfiler.traceStartTick = profileStart
            self.interconnectProfiler.interconnect = self.interconnect

    def createL2(self, bus, L2BankCount, moduloAddr):
        for bankID in range(0, L2BankCount):
            thisBank = None
            if bus:
                thisBank = L2Bank(in_bus=Parent.interconnect, out_bus=Parent.
                    toMemBus)
            else:
                thisBank = L2Bank(in_interconnect=Parent.interconnect, out_bus=
                    Parent.toMemBus)
            if moduloAddr and not bus:
                thisBank.setModuloAddr(bankID, L2BankCount)
            else:
                thisBank.setAddrRange(bankID, L2BankCount)
            self.l2.append(thisBank)

    def setL2Banks(self):
        self.L2Bank0 = self.l2[0]
        self.L2Bank1 = self.l2[1]
        self.L2Bank2 = self.l2[2]
        self.L2Bank3 = self.l2[3]
```

D.3 FuncUnitConfig.py

```python
from m5 import *
class IntALU(FUDesc):
    opList = [ OpDesc(opClass='IntAlu') ]
    count = 4

class IntMultDiv(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=3),
               OpDesc(opClass='IntDiv', opLat=20, issueLat=19) ]
    count = 2

class FP_ALU(FUDesc):
    opList = [ OpDesc(opClass='FloatAdd', opLat=2),
               OpDesc(opClass='FloatCmp', opLat=2),
               OpDesc(opClass='FloatCvt', opLat=2) ]
    count = 4

class FP_MultDiv(FUDesc):
    opList = [ OpDesc(opClass='FloatMult', opLat=4),
               OpDesc(opClass='FloatDiv', opLat=12, issueLat=12),
               OpDesc(opClass='FloatSqrt', opLat=24, issueLat=24) ]
    count = 2

class ReadPort(FUDesc):
    opList = [ OpDesc(opClass='MemRead') ]
    count = 0

class WritePort(FUDesc):
    opList = [ OpDesc(opClass='MemWrite') ]
    count = 0

class RdWrPort(FUDesc):
    opList = [ OpDesc(opClass='MemRead'), OpDesc(opClass='MemWrite') ]
    count = 4

class IprPort(FUDesc):
    opList = [ OpDesc(opClass='IprAccess', opLat = 3, issueLat = 3) ]
    count = 1

class DefaultFUP(FuncUnitPool):
    FUList = [ IntALU(), IntMultDiv(), FP_ALU(), FP_MultDiv(), ReadPort(),
               WritePort(), RdWrPort(), IprPort() ]
```

D.4 MemConfig.py

```
from m5 import *

###############################################################################
# CACHES
###############################################################################

class BaseL1Cache(BaseCache):
    in_bus = NULL
    size = '64kB'
    assoc = 8
    block_size = 64
    mshrs = 4
    tgts_per_mshr = 4
    cpu_count = int(env['NP'])
    is_shared = False

class IL1(BaseL1Cache):
    latency = Parent.clock.period
    is_read_only = True

class DL1(BaseL1Cache):
    latency = 3 * Parent.clock.period
    is_read_only = False

class L2Bank(BaseCache):
    size = '1MB' # 1MB * 4 banks = 4MB total cache size
    assoc = 8
    block_size = 64
    latency = 14 * Parent.clock.period
    mshrs = 8
    tgts_per_mshr = 4
    cpu_count = int(env['NP'])
    is_shared = True
    is_read_only = False

    def setModuloAddr(self, bankID, bank_count):
        self.do_modulo_addr = True
        self.bank_id = bankID
        self.bank_count = bank_count

    def setAddrRange(self, bankID, bank_count):
        offset = MaxAddr / bank_count
        if bankID == 0:
            self.addr_range = AddrRange(0, offset)
        elif bankID == (bank_count-1):
            self.addr_range = AddrRange((bankID*offset)+1, MaxAddr)
        else:
            self.addr_range = AddrRange((bankID*offset)+1, ((bankID+1)*offset))

###############################################################################
# INTERCONNECT
###############################################################################

class ToL2Bus(Bus):
    width = 64
    clock = Parent.clock.period

class InterconnectBus(SplitTransBus):
    width = 64
    clock = 1 * Parent.clock.period
    transferDelay = 4
    arbitrationDelay = 5
    pipelined = False

class PipelinedBus(InterconnectBus):
    pipelined = True

class InterconnectIdeal(IdealInterconnect):
    width = 64 # the cache needs finite width
    clock = 1 * Parent.clock.period
    transferDelay = 0
    arbitrationDelay = 0

class InterconnectIdealWithDelay(IdealInterconnect):
    width = 64 # the cache needs finite width
    clock = 1 * Parent.clock.period
    transferDelay = 4
    arbitrationDelay = 5

class InterconnectCrossbar(Crossbar):
    width = 64
    clock = 1 * Parent.clock.period
    transferDelay = 4
    arbitrationDelay = 5

class InterconnectButterfly(Butterfly):
    width = 64
    clock = 1 * Parent.clock.period
    radix = 2
    banks = 4

    if int(env['NP']) == 2:
        # total delay is 10 clock cycles (2*2+3*2)
        transferDelay = 2 # per link transfer delay
        arbitrationDelay = 0 # arb in switches, no explicit delay
        switch_delay = 2
    elif int(env['NP']) == 4:
        # total delay is 10 clock cycles (2*3+1*4)
        transferDelay = 1 # per link transfer delay
        arbitrationDelay = 0 # arb in switches, no explicit delay
        switch_delay = 2
    else:
        # total delay is 9 clock cycles (1*4+1*5)
        transferDelay = 1 # per link transfer delay
        arbitrationDelay = 0 # arb in switches, no explicit delay
        switch_delay = 1


################################################################################
# MEMORY AND MEMORY BUS
################################################################################

class ToMemBus(Bus):
    width = 8
    #clock = 1.5 * Parent.clock.period
    clock = 4 * Parent.clock.period

class SDRAM(BaseMemory):
    #latency = 200 * Parent.clock.period
    latency = 112 * Parent.clock.period
    uncacheable_latency = 1000 * Parent.clock.period
```

D.5 Spec2000.py

```
from m5 import *
import os
import os.path
from shutil import copy, copytree
import glob

# Originally written by James Srinivasan
# Further modified by Magnus Jahre <jahre @ idi.ntnu.no>

if 'NP' not in env:
    panic("No number of processors was defined.\ne.g. -ENP=4\n")

rootdir = os.getenv("DIPPROOT")
if rootdir == None:
  print "Envirionment variable DIPPROOT not set. Quitting..."
  sys.exit(-1)

# Assumes current working directory is where we ought to run the benchmarks from,
    copy datasets to etc.

# Root of where SPEC2000 install lives

spec_root = rootdir+'/experiments/benchmarks/spec2000/SPEC_2000_REDUCED'

# Location of SPEC binaries
spec_bin  = rootdir+'/experiments/benchmarks/spec2000/'

# String to benchmark mappings

def createWorkload(benchmarkStrings):
    returnArray = []

    for string in benchmarkStrings:
        if string == 'gzip':
            returnArray.append(GzipSource())
        elif string == 'vpr':
            returnArray.append(VprPlace())
        elif string == 'gcc':
            returnArray.append(Gcc166())
        elif string == 'mcf':
            returnArray.append(Mcf())
        elif string == 'crafty':
            returnArray.append(Crafty())
        elif string == 'parser':
            returnArray.append(Parser())
        elif string == 'eon':
            returnArray.append(Eon1())
        elif string == 'perlbmk':
            returnArray.append(Perlbmk1())
        elif string == 'gap':
            returnArray.append(Gap())
        elif string == 'vortex1':
            returnArray.append(Vortex1())
        elif string == 'bzip':
            returnArray.append(Bzip2Source())
        elif string == 'twolf':
            returnArray.append(Twolf())
        elif string == 'wupwise':
            returnArray.append(Wupwise())
        elif string == 'swim':
            returnArray.append(Swim())
        elif string == 'mgrid':
            returnArray.append(Mgrid())
        elif string == 'applu':
            returnArray.append(Applu())
        elif string == 'mesa':
            returnArray.append(Mesa())
        elif string == 'galgel':
            returnArray.append(Galgel())
        elif string == 'art':
            returnArray.append(Art1())
        elif string == 'equake':
            returnArray.append(Equake())
        elif string == 'facerec':
            returnArray.append(Facerec())
        elif string == 'ammp':
            returnArray.append(Ammp())
        elif string == 'lucas':
            returnArray.append(Lucas())
        elif string == 'fma3d':
            returnArray.append(Fma3d())
        elif string == 'sixtrack':
            returnArray.append(Sixtrack())
        elif string == 'apsi':
            returnArray.append(Apsi())
        else:
            panic("Unknown benchmark is part of workload")

    return returnArray

##############################################################################
##############################################################################

class GzipSource(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '164.gzip/input/ref.source') , '.')

    executable = os.path.join(spec_bin, 'gzip00.peak.ev6')
    cmd = 'gzip00.peak.ev6 ref.source 60'

class GzipLog(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '164.gzip/input/ref.log') , '.')

    executable = os.path.join(spec_bin, 'gzip00.peak.ev6')
    cmd = 'gzip00.peak.ev6 ref.log 60'

class GzipGraphic(LiveProcess):
    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '164.gzip/input/ref.graphic') , '.')

    executable = os.path.join(spec_bin, 'gzip00.peak.ev6')
    cmd = 'gzip00.peak.ev6 ref.graphic 60'

class GzipRandom(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '164.gzip/input/ref.random') , '.')

    executable = os.path.join(spec_bin, 'gzip00.peak.ev6')
    cmd = 'gzip00.peak.ev6 ref.random 60'

class GzipProgram(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '164.gzip/input/ref.program') , '.')

    executable = os.path.join(spec_bin, 'gzip00.peak.ev6')
    cmd = 'gzip00.peak.ev6 ref.program 60'

###############################################################################

class VprPlace(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '175.vpr/input/ref.net') ,  'refVpr.net')
    copy(os.path.join(spec_root, '175.vpr/input/ref.arch.in') , 'refVpr.arch.in')

    executable = os.path.join(spec_bin, 'vpr00.peak.ev6')
    cmd = 'vpr00.peak.ev6 ' +                          \
          'refVpr.net refVpr.arch.in place.out dum.out ' +                \
          '-nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num
             2'


###############################################################################

class Gcc166(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor
    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '176.gcc/input/ref.166.i'), ".")

    executable = os.path.join(spec_bin, 'gcc00.peak.ev6')
    cmd = 'gcc00.peak.ev6 ref.166.i -o ref.166.s'

class Gcc200(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '176.gcc/input/ref.200.i'), ".")

    #executable = os.path.join(spec_bin, 'cc100.peak.ev6')
    #cmd = 'cc100.peak.ev6 200.i -o 200.s'
    executable = os.path.join(spec_bin, 'gcc00.peak.ev6')
    cmd = 'gcc00.peak.ev6 ref.200.i -o ref.200.s'


class GccExpr(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '176.gcc/input/ref.expr.i'), ".")

    executable = os.path.join(spec_bin, 'gcc00.peak.ev6')
    cmd = 'gcc00.peak.ev6 ref.expr.i -o ref.expr.s'

class GccIntegrate(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '176.gcc/input/ref.integrate.i'), ".")

    executable = os.path.join(spec_bin, 'gcc00.peak.ev6')
    cmd = 'gcc00.peak.ev6 ref.integrate.i -o ref.integrate.s'

class GccScilab(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '176.gcc/input/ref.scilab.i'), ".")

    executable = os.path.join(spec_bin, 'gcc00.peak.ev6')
    cmd = 'gcc00.peak.ev6 ref.scilab.i -o ref.scilab.s'

##############################################################################

class Mcf(LiveProcess):
    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '181.mcf/input/ref.in'), "refMcf.in")

    executable = os.path.join(spec_bin, 'mcf00.peak.ev6')
    cmd = 'mcf00.peak.ev6 refMcf.in'

##############################################################################

class Crafty(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '186.crafty/input/ref/ref.in'), "./craftyref.in"
        )

    executable = os.path.join(spec_bin, 'crafty00.peak.ev6')
    cmd = 'crafty00.peak.ev6 '
    input = 'craftyref.in'      # source for stderr

##############################################################################

class Parser(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '197.parser/input/ref.in'),   "refParser.in")
    copy(os.path.join(spec_root, '197.parser/input/2.1.dict'), ".")

    # for some reason this constructor gets called twice but if the target already
         exists copytree will fail so check first
    if not os.path.exists("words"):
        copytree(os.path.join(spec_root, '197.parser/input/words'), "words")

    executable = os.path.join(spec_bin, 'parser00.peak.ev6')
    cmd = 'parser00.peak.ev6 2.1.dict -batch'
    input = 'refParser.in'      # source for stderr

##############################################################################

class Eon1(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '252.eon/input/ref/eon.dat'),              ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/materials'),            ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/spectra.dat'),          ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.control.cook'),   ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.camera'),         ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.surfaces') ,      ".")

    executable = os.path.join(spec_bin, 'eon00.peak.ev6')
    cmd = 'eon00.peak.ev6 chair.control.cook chair.camera chair.surfaces chair.
        cook.ppm ppm pixels_out.cook'

class Eon2(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '252.eon/input/ref/eon.dat') ,             ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/materials') ,           ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/spectra.dat') ,         ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.control.rushmeier') ,  "
        .")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.camera') ,        ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.surfaces') ,       ".")

    executable = os.path.join(spec_bin, 'eon00.peak.ev6')
    cmd = 'eon00.peak.ev6 chair.control.rushmeier chair.camera chair.surfaces
        chair.rushmeier.ppm ppm pixels_out.rushmeier'

class Eon3(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '252.eon/input/ref/eon.dat') ,             ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/materials') ,           ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/spectra.dat') ,         ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.control.kajiya') ,  ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.camera') ,        ".")
    copy(os.path.join(spec_root, '252.eon/input/ref/chair.surfaces') ,       ".")

    executable = os.path.join(spec_bin, 'eon00.peak.ev6')
    cmd = 'eon00.peak.ev6 chair.control.kajiya chair.camera chair.surfaces chair.
        kajiya.ppm ppm pixels_out.kajiya'


################################################################################


class Perlbmk1(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums') , ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/diffmail.pl') , ".")

    # for some reason this constructor gets called twice but if the target already
        exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib') , "lib")
    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib diffmail.pl 2 550 15 24 23 100'

class Perlbmk2(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/makerand.pl'), ".")

    # for some reason this constructor gets called twice but if the target already
         exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib makerand.pl'

class Perlbmk3(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/perfect.pl'), ".")

    # for some reason this constructor gets called twice but if the target already
         exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib perfect.pl b 3 m 4'

class Perlbmk4(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/splitmail.pl'), ".")

    # for some reason this constructor gets called twice but if the target already
         exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib splitmail.pl 850 5 19 18 1500'

class Perlbmk5(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/splitmail.pl'), ".")

    # for some reason this constructor gets called twice but if the target already
          exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib splitmail.pl 704 12 26 16 836'

class Perlbmk6(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/splitmail.pl'), ".")

    # for some reason this constructor gets called twice but if the target already
          exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib splitmail.pl 535 13 25 24 1091'

class Perlbmk7(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/lenums'), ".")
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/cpu2000_mhonarc.rc'), "."
        )
    copy(os.path.join(spec_root, '253.perlbmk/input/ref/splitmail.pl'), ".")
    # for some reason this constructor gets called twice but if the target already
        exists copytree will fail so check first
    if not os.path.exists("lib"):
        copytree(os.path.join(spec_root, '253.perlbmk/input/ref/lib'), "lib")


    executable = os.path.join(spec_bin, 'perlbmk00.peak.ev6')
    cmd = 'perlbmk00.peak.ev6 -I./lib splitmail.pl 957 12 23 26 1014'

###############################################################################

class Gap(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '254.gap/input/ref/ref.in'), "./gapref.in")

    # copy input file by file
    for file in glob.glob(os.path.join(spec_root, '254.gap/input/ref/*')):
            if os.path.basename(file) != "ref.in":
                copy(file, ".")

    executable = os.path.join(spec_bin, 'gap00.peak.ev6')
    cmd = 'gap00.peak.ev6 -l ./ -q -m 192M'
    input = 'gapref.in'

###############################################################################

class Vortex1(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
    copy(os.path.join(spec_root, '255.vortex/input/persons.1k'), "./persons.1k")
    copy(os.path.join(spec_root, '255.vortex/input/lendian.rnv'), "./lendian.rnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian.wnv'), "./lendian.wnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian1.raw'), "./lendian1.
        raw")

    executable = os.path.join(spec_bin, 'vortex00.peak.ev6')
    cmd = 'vortex00.peak.ev6 lendian1.raw'

class Vortex2(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '255.vortex/input/persons.1k'), "./persons.1k")
    copy(os.path.join(spec_root, '255.vortex/input/lendian.rnv'), "./lendian.rnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian.wnv'), "./lendian.wnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian2.raw') , "./lendian2.
        raw")


    executable = os.path.join(spec_bin, 'vortex00.peak.ev6')
    cmd = 'vortex00.peak.ev6 lendian2.raw'

class Vortex3(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '255.vortex/input/persons.1k') , "./persons.1k")
    copy(os.path.join(spec_root, '255.vortex/input/lendian.rnv') , "./lendian.rnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian.wnv') , "./lendian.wnv"
        )
    copy(os.path.join(spec_root, '255.vortex/input/lendian3.raw') , "./lendian3.
        raw")


    executable = os.path.join(spec_bin, 'vortex00.peak.ev6')
    cmd = 'vortex00.peak.ev6 lendian3.raw'

################################################################################

class Bzip2Source(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '256.bzip2/input/ref.source') , ".")

    executable = os.path.join(spec_bin, 'bzip200.peak.ev6')
    cmd = 'bzip200.peak.ev6 ref.source 58'

class Bzip2Graphic(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '256.bzip2/input/ref.graphic') , ".")

    executable = os.path.join(spec_bin, 'bzip200.peak.ev6')
    cmd = 'bzip200.peak.ev6 ref.graphic 58'

class Bzip2Program(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '256.bzip2/input/ref.program') , ".")
    executable = os.path.join(spec_bin, 'bzip200.peak.ev6')
    cmd = 'bzip200.peak.ev6 ref.program 58'

###############################################################################

class Twolf(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file by file
    for file in glob.glob(os.path.join(spec_root, '300.twolf/input/ref/*')):
        copy(file, ".")

    executable = os.path.join(spec_bin, 'twolf00.peak.ev6')
    cmd = 'twolf00.peak.ev6 ref'


###############################################################################

class Wupwise(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '168.wupwise/input/ref/wupwise.in'), ".")

    executable = os.path.join(spec_bin, 'wupwise00.peak.ev6')
    cmd = 'wupwise00.peak.ev6'

###############################################################################

class Swim(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '171.swim/input/ref/swim.in'), ".")

    executable = os.path.join(spec_bin, 'swim00.peak.ev6')
    cmd = 'swim00.peak.ev6'
    input = 'swim.in'

###############################################################################

class Mgrid(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '172.mgrid/input/ref/mgrid.in'), ".")

    executable = os.path.join(spec_bin, 'mgrid00.peak.ev6')
    cmd = 'mgrid00.peak.ev6'
    input = 'mgrid.in'

###############################################################################

class Applu(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '173.applu/input/ref/applu.in') , ".")

    executable = os.path.join(spec_bin, 'applu00.peak.ev6')
    cmd = 'applu00.peak.ev6'
    input = 'applu.in'

###############################################################################

class Mesa(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '177.mesa/input/ref.in') , "./refMesa.in")

        # Can't find this file
        #copy(os.path.join(spec_root, 'benchspec/CFP2000/177.mesa/data/ref/input/
            numbers') , ".")

    executable = os.path.join(spec_bin, 'mesa00.peak.ev6')
    cmd = 'mesa00.peak.ev6 -frames 1000 -meshfile refMesa.in -ppmfile mesa.ppm'

###############################################################################

class Galgel(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '178.galgel/input/ref/ref.in') , "refGalgel.in")

    executable = os.path.join(spec_bin, 'galgel00.peak.ev6')
    cmd = 'galgel00.peak.ev6'
    input = 'refGalgel.in'

###############################################################################

class Art1(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
    copy(os.path.join(spec_root, '179.art/input/a10.img')    , ".")
    copy(os.path.join(spec_root, '179.art/input/c756hel.in') , ".")
    copy(os.path.join(spec_root, '179.art/input/hc.img')      , ".")

    executable = os.path.join(spec_bin, 'art00.peak.ev6')
    cmd = 'art00.peak.ev6 -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.
        img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10'

################################################################################

class Art2(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
    copy(os.path.join(spec_root, '179.art/input/a10.img')     , ".")
    copy(os.path.join(spec_root, '179.art/input/c756hel.in')  , ".")
    copy(os.path.join(spec_root, '179.art/input/hc.img')      , ".")

    executable = os.path.join(spec_bin, 'art00.peak.ev6')
    cmd = 'art00.peak.ev6 -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.
        img -stride 2 -startx 470 -starty 140 -endx 520 -endy 180 -objects 10'

################################################################################

class Equake(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
    copy(os.path.join(spec_root, '183.equake/input/ref/inp.in') , "inpEquake.in")

    executable = os.path.join(spec_bin, 'equake00.peak.ev6')
    cmd = 'equake00.peak.ev6'
    input = 'inpEquake.in'

################################################################################

class Facerec(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file by file
        for file in glob.glob(os.path.join(spec_root, '187.facerec/input/ref/*')):
            if os.path.basename(file) == "ref.in":
                copy(file, "./refFacerec.in")
            else:
                copy(file, ".")
        for file in glob.glob(os.path.join(spec_root, '187.facerec/input/all/input
            /*')):
            copy(file, ".")

    executable = os.path.join(spec_bin, 'facerec00.peak.ev6')
    cmd = 'facerec00.peak.ev6'
    input = 'refFacerec.in'

################################################################################
```

```
class Ammp(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file by file
    #for file in glob.glob(os.path.join(spec_root, '188.ammp/input/*')):
    #    copy(file, ".")
        if not os.path.exists("input"):
        copytree(os.path.join(spec_root, '188.ammp/input') , "input")

    executable = os.path.join(spec_bin, 'ammp00.peak.ev6')
    cmd = 'ammp00.peak.ev6'
    input = 'input/ref.in'

##############################################################################

class Lucas(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
    copy(os.path.join(spec_root, '189.lucas/input/ref/ref.in') , "./lucasref.in")

    executable = os.path.join(spec_bin, 'lucas00.peak.ev6')
    cmd = 'lucas00.peak.ev6'
    input = 'lucasref.in'

##############################################################################

class Fma3d(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file(s) to run directory
        copy(os.path.join(spec_root, '191.fma3d/input/ref/fma3d.in') , ".")

    executable = os.path.join(spec_bin, 'fma3d00.peak.ev6')
    cmd = 'fma3d00.peak.ev6'

##############################################################################

class Sixtrack(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

        # copy input file by file
        for file in glob.glob(os.path.join(spec_root, '200.sixtrack/input/all/
            input/*')):
            copy(file, ".")
        for file in glob.glob(os.path.join(spec_root, '200.sixtrack/input/ref/*'))
            :
            copy(file, ".")
    executable = os.path.join(spec_bin, 'sixtrack00.peak.ev6')
    cmd = 'sixtrack00.peak.ev6'
    input = 'inp.in'

###############################################################################

class Apsi(LiveProcess):

    def __init__(self, _value_parent = None, **kwargs):

        LiveProcess.__init__(self)  # call parent constructor

    # copy input file(s) to run directory
    copy(os.path.join(spec_root, '301.apsi/input/ref/apsi.in'), ".")

    executable = os.path.join(spec_bin, 'apsi00.peak.ev6')
    cmd = 'apsi00.peak.ev6'
    input = 'apsi.in'

###############################################################################
```

D.6 workloads.py

```
# Autogenerated workload configuration file

workloads = {

2:{
1:(['sixtrack', 'gcc'],[1092486526, 1089243335]),
2:(['twolf', 'mcf'],[1059202723, 1053008801]),
3:(['twolf', 'twolf'],[1019199715, 1022103917]),
4:(['gcc', 'bzip'],[1002476696, 1071000820]),
5:(['equake', 'vpr'],[1051263392, 1035899636]),
6:(['applu', 'mesa'],[1078310236, 1065791271]),
7:(['vortex1', 'vortex1'],[1097073523, 1075978661]),
8:(['galgel', 'gcc'],[1057204678, 1031439111]),
9:(['art', 'twolf'],[1047406642, 1059060970]),
10:(['crafty', 'gap'],[1064237847, 1017853995]),
11:(['parser', 'gcc'],[1081587350, 1011430120]),
12:(['perlbmk', 'ammp'],[1003893316, 1057282190]),
13:(['vortex1', 'perlbmk'],[1033080485, 1028848432]),
14:(['mgrid', 'gcc'],[1077758250, 1025841852]),
15:(['art', 'eon'],[1078380907, 1082856533]),
16:(['fma3d', 'sixtrack'],[1054547902, 1059269574]),
17:(['apsi', 'bzip'],[1016725975, 1005591980]),
18:(['ammp', 'apsi'],[1002336280, 1068720536]),
19:(['sixtrack', 'apsi'],[1080496374, 1053859612]),
20:(['gap', 'vortex1'],[1074901437, 1097439116]),
21:(['vpr', 'parser'],[1019331389, 1046595780]),
22:(['sixtrack', 'gcc'],[1093357813, 1021595111]),
23:(['crafty', 'ammp'],[1052896999, 1054364834]),
24:(['bzip', 'twolf'],[1025306235, 1090087001]),
25:(['fma3d', 'fma3d'],[1034277622, 1081375595]),
26:(['bzip', 'ammp'],[1068774434, 1069302295]),
27:(['eon', 'bzip'],[1063951215, 1029525317]),
28:(['mgrid', 'ammp'],[1077166566, 1054993749]),
29:(['mesa', 'mgrid'],[1010844627, 1033250230]),
30:(['eon', 'gcc'],[1083534947, 1095151804]),
31:(['mgrid', 'mgrid'],[1056202128, 1023508059]),
32:(['twolf', 'gzip'],[1047072697, 1071838019]),
33:(['facerec', 'lucas'],[1037717684, 1014186334]),
34:(['galgel', 'twolf'],[1020359060, 1017316543]),
35:(['gcc', 'wupwise'],[1042255024, 1069960666]),
36:(['mgrid', 'swim'],[1068900962, 1055587084]),
37:(['fma3d', 'lucas'],[1029779454, 1008639060]),
38:(['wupwise', 'galgel'],[1086346408, 1077694823]),
39:(['perlbmk', 'vortex1'],[1060507355, 1031964886]),
40:(['sixtrack', 'bzip'],[1030217175, 1034127697])
},

4:{
1:(['ammp', 'mgrid', 'perlbmk', 'parser'],[1041955945, 1047775879, 1025548197,
    1008908800]),
2:(['lucas', 'gcc', 'mcf', 'twolf'],[1026388815, 1050246990, 1046272284,
    1056602690]),
3:(['eon', 'eon', 'mesa', 'facerec'],[1085828439, 1085086202, 1098261402,
    1004267962]),
4:(['vortex1', 'ammp', 'equake', 'galgel'],[1027564658, 1014658355, 1009037399,
    1039572509]),
5:(['gcc', 'galgel', 'apsi', 'crafty'],[1029306590, 1091516994, 1016968254,
    1091181775]),
6:(['applu', 'equake', 'art', 'facerec'],[1001909990, 1013915170, 1046887563,
    1056979138]),
7:(['applu', 'gap', 'gcc', 'parser'],[1082162802, 1059139806, 1013409002,
    1085694384]),
8:(['gap', 'swim', 'twolf', 'mesa'],[1042656444, 1061963955, 1085903965,
    1036190567]),
9:(['sixtrack', 'fma3d', 'apsi', 'vortex1'],[1074480257, 1031183064, 1098143364,
    1012919523]),
10:(['ammp', 'bzip', 'equake', 'parser'],[1077398959, 1003951563, 1072415593,
    1053509179]),
11:(['vpr', 'twolf', 'applu', 'eon'],[1040680776, 1031568211, 1082293995,
    1041436570]),
12:(['galgel', 'crafty', 'mgrid', 'swim'],[1031527863, 1044545857, 1082173250,
    1096751917]),
13:(['twolf', 'fma3d', 'galgel', 'vpr'],[1062306790, 1060828350, 1098129008,
    1043023932]),
14:(['bzip', 'vpr', 'bzip', 'equake'],[1084019868, 1038244774, 1003412847,
    1097472955]),
15:(['galgel', 'crafty', 'vpr', 'swim'],[1070880481, 1027287316, 1060235344,
    1058807655]),
16:(['mcf', 'wupwise', 'mesa', 'mesa'],[1054249832, 1006759950, 1014557494,
    1030953598]),
17:(['applu', 'parser', 'apsi', 'perlbmk'],[1075021039, 1053158322, 1034718910,
    1026856922]),
18:(['mgrid', 'perlbmk', 'gzip', 'mgrid'],[1049328406, 1079074439, 1096282781,
    1079036253]),
19:(['mcf', 'sixtrack', 'gcc', 'apsi'],[1090116441, 1068921998, 1066705590,
    1092093538]),
20:(['ammp', 'gcc', 'art', 'mesa'],[1011080402, 1007932868, 1079537464,
    1095718719]),
21:(['perlbmk', 'apsi', 'lucas', 'equake'],[1051169802, 1057285545, 1064666557,
    1019744818]),
22:(['vpr', 'crafty', 'vpr', 'mcf'],[1073177627, 1082019945, 1021734200,
    1066267018]),
23:(['gzip', 'equake', 'mgrid', 'mesa'],[1097569789, 1080949028, 1056929996,
    1079797826]),
24:(['facerec', 'applu', 'fma3d', 'lucas'],[1013937124, 1035387836, 1051243465,
    1041436071]),
25:(['gap', 'applu', 'parser', 'facerec'],[1008180602, 1067057433, 1083231912,
    1080419219]),
26:(['mcf', 'apsi', 'twolf', 'ammp'],[1014292526, 1058328743, 1061373130,
    1050686626]),
27:(['swim', 'sixtrack', 'ammp', 'applu'],[1052228680, 1059328443, 1080039777,
    1026620495]),
28:(['art', 'fma3d', 'swim', 'parser'],[1082308602, 1095181635, 1012762841,
    1035776155]),
29:(['apsi', 'gcc', 'vortex1', 'twolf'],[1050080000, 1076827259, 1024773007,
    1088514951]),
30:(['mgrid', 'gzip', 'apsi', 'equake'],[1015952145, 1024722623, 1059266770,
    1077591627]),
31:(['mgrid', 'equake', 'vpr', 'eon'],[1015263556, 1063692577, 1044670814,
    1092770749]),
32:(['wupwise', 'gap', 'twolf', 'facerec'],[1073842062, 1077919529, 1009246189,
    1048001712]),
33:(['galgel', 'equake', 'lucas', 'gzip'],[1040375492, 1037630973, 1017422599,
    1094439053]),
34:(['facerec', 'gcc', 'facerec', 'apsi'],[1085839746, 1069300438, 1073285869,
    1062627766]),
35:(['mesa', 'mcf', 'swim', 'sixtrack'],[1094695081, 1092502223, 1029829307,
    1052267670]),
36:(['mesa', 'sixtrack', 'equake', 'bzip'],[1063594040, 1062127033, 1040041781,
    1060015597]),
37:(['mcf', 'gap', 'gcc', 'vortex1'],[1033902102, 1001090684, 1030020762,
    1048547872]),
38:(['facerec', 'lucas', 'mcf', 'parser'],[1092600483, 1066508342, 1027466999,
    1060969516]),
39:(['twolf', 'eon', 'mesa', 'eon'],[1098504941, 1088612335, 1009372945,
    1069289808]),
40:(['apsi', 'apsi', 'mcf', 'equake'],[1092680129, 1068726226, 1098316344,
    1073035913])
},

8:{
1:(['gap', 'applu', 'vpr', 'gap', 'mcf', 'mcf', 'twolf', 'vortex1'],[1073187984,
    1054331802, 1084995449, 1037520798, 1079155778, 1006168982, 1011603043,
    1086569852]),
2:(['galgel', 'mgrid', 'twolf', 'mesa', 'equake', 'equake', 'swim', 'facerec'
    ],[1010114816, 1033266768, 1074762683, 1073328214, 1085178419, 1090120043,
    1028221923, 1077227382]),
3:(['ammp', 'mgrid', 'vpr', 'art', 'lucas', 'parser', 'galgel', 'gzip'
    ],[1073035280, 1082456083, 1064051258, 1047745378, 1091351248, 1043930366,
    1088556254, 1019159065]),
4:(['mgrid', 'apsi', 'equake', 'eon', 'crafty', 'twolf', 'mcf', 'bzip'
    ],[1087805348, 1010946963, 1058724149, 1080641094, 1052238876, 1061243098,
    1079000456, 1020553687]),
5:(['bzip', 'lucas', 'ammp', 'eon', 'perlbmk', 'gcc', 'parser', 'vpr'
    ],[1000157189, 1096256620, 1018004216, 1022509362, 1063759077, 1020126159,
    1017677198, 1091187089]),
6:(['parser', 'gzip', 'equake', 'bzip', 'wupwise', 'gcc', 'perlbmk', 'mcf'
    ],[1054015368, 1072027794, 1037141802, 1002590313, 1032598159, 1062865867,
    1086592320, 1077206505]),
7:(['parser', 'eon', 'gcc', 'swim', 'swim', 'vpr', 'galgel', 'swim'],[1009382201,
    1008386991, 1013955362, 1085186502, 1049653866, 1081052784, 1004535245,
    1000701983]),
8:(['lucas', 'bzip', 'applu', 'equake', 'mgrid', 'ammp', 'ammp', 'gcc'
    ],[1062490836, 1000066629, 1031295467, 1003301732, 1095699052, 1066621695,
    1084377485, 1048723512]),
9:(['ammp', 'gap', 'mesa', 'facerec', 'eon', 'vpr', 'bzip', 'galgel'],[1079442762,
     1070093875, 1097495616, 1031240052, 1022413214, 1090524731, 1086977938,
    1057202237]),
10:(['parser', 'swim', 'twolf', 'gcc', 'vpr', 'bzip', 'facerec', 'gzip'
    ],[1090257985, 1009293715, 1022410113, 1011023325, 1056748310, 1026660178,
    1009495664, 1090156263]),
11:(['crafty', 'vpr', 'sixtrack', 'crafty', 'lucas', 'crafty', 'equake', 'apsi'
    ],[1080684386, 1072953538, 1074549432, 1025906445, 1034048076, 1078018402,
    1090184500, 1095714609]),
12:(['art', 'crafty', 'eon', 'vortex1', 'fma3d', 'mgrid', 'crafty', 'equake'
    ],[1015843197, 1002447314, 1053646470, 1021827013, 1045943050, 1058122519,
    1071926742, 1080330548]),
13:(['twolf', 'vpr', 'mesa', 'fma3d', 'equake', 'sixtrack', 'gap', 'gzip'
    ],[1081428568, 1054183010, 1045848482, 1094992822, 1033922911, 1085605808,
    1068053379, 1087356592]),
14:(['twolf', 'mesa', 'crafty', 'equake', 'vortex1', 'mgrid', 'swim', 'gap'
    ],[1056751279, 1088275153, 1020060137, 1003606008, 1078323826, 1022366158,
    1019548656, 1034359417]),
15:(['eon', 'mgrid', 'mcf', 'perlbmk', 'wupwise', 'crafty', 'twolf', 'swim'
    ],[1069111934, 1033143088, 1094184021, 1030649130, 1031499067, 1082028261,
    1071878030, 1057197432]),
16:(['crafty', 'bzip', 'applu', 'apsi', 'gzip', 'galgel', 'equake', 'perlbmk'
    ],[1060295839, 1061605217, 1078408538, 1040793226, 1017904531, 1012317818,
    1009187526, 1008829265]),
17:(['gzip', 'apsi', 'bzip', 'mgrid', 'gap', 'art', 'art', 'bzip'],[1062227060,
    1043287066, 1072789027, 1090842725, 1081090176, 1038163398, 1076091368,
    1015033152]),
18:(['eon', 'equake', 'vortex1', 'art', 'gcc', 'apsi', 'facerec', 'gzip'
    ],[1031823376, 1011627676, 1014152137, 1001876755, 1062533802, 1095016852,
    1075002641, 1070354554]),
19:(['eon', 'mesa', 'vortex1', 'eon', 'gcc', 'lucas', 'equake', 'galgel'
    ],[1027310441, 1002162446, 1059120707, 1074827012, 1028078935, 1023522418,
    1049001044, 1086781145]),
20:(['apsi', 'bzip', 'galgel', 'ammp', 'art', 'galgel', 'ammp', 'sixtrack'
    ],[1036116693, 1008009383, 1088504472, 1035365251, 1075786712, 1026648985,
    1065582559, 1051291310]),
21:(['parser', 'parser', 'gap', 'gap', 'ammp', 'applu', 'vortex1', 'art'
    ],[1072012746, 1057840199, 1061743354, 1099794688, 1078982271, 1037341159,
    1023276131, 1043919426]),
22:(['crafty', 'swim', 'twolf', 'galgel', 'swim', 'twolf', 'twolf', 'parser'
    ],[1050818960, 1064897126, 1052515062, 1086678782, 1066161812, 1040178194,
    1067884105, 1006949019]),
23:(['vpr', 'vortex1', 'parser', 'twolf', 'eon', 'equake', 'gzip', 'fma3d'
    ],[1005348033, 1035925273, 1045642080, 1023304669, 1029043677, 1049160390,
    1006073261, 1044794122]),
24:(['vortex1', 'galgel', 'ammp', 'parser', 'bzip', 'vpr', 'mesa', 'ammp'
    ],[1063194330, 1097275798, 1082541931, 1003883854, 1079087447, 1090030082,
    1093793294, 1041142266]),
25:(['twolf', 'facerec', 'perlbmk', 'gzip', 'vpr', 'vortex1', 'wupwise', 'eon'
    ],[1053550239, 1018430586, 1014624132, 1008809943, 1029707942, 1046921866,
    1001808722, 1028638607]),
26:(['gap', 'sixtrack', 'eon', 'applu', 'swim', 'perlbmk', 'vpr', 'apsi'
    ],[1054115093, 1005445699, 1097166435, 1065930690, 1024858175, 1026537869,
    1045341467, 1023002078]),
27:(['gap', 'gap', 'gap', 'twolf', 'mcf', 'gap', 'lucas', 'bzip'],[1077017001,
    1099668442, 1085122280, 1089412190, 1087185975, 1039440799, 1078171971,
    1013558211]),
28:(['vpr', 'vpr', 'twolf', 'mesa', 'gap', 'bzip', 'gzip', 'sixtrack'
    ],[1017259907, 1013013234, 1030175803, 1090459182, 1076078700, 1032172360,
    1023272393, 1027539069]),
29:(['swim', 'equake', 'swim', 'wupwise', 'fma3d', 'sixtrack', 'lucas', 'vortex1'
    ],[1005106200, 1023192637, 1003810844, 1072497715, 1014349757, 1001914569,
    1081817129, 1086550774]),
30:(['wupwise', 'vortex1', 'gap', 'vpr', 'fma3d', 'vortex1', 'art', 'mgrid'
    ],[1062614948, 1058495568, 1085462700, 1056121083, 1005862000, 1035665738,
    1098565664, 1015543999]),
31:(['applu', 'perlbmk', 'applu', 'galgel', 'crafty', 'wupwise', 'gap', 'ammp'
    ],[1048307705, 1027263945, 1094469129, 1055082711, 1075216695, 1078291964,
    1015047097, 1099078359]),
32:(['swim', 'bzip', 'swim', 'apsi', 'vpr', 'gcc', 'twolf', 'twolf'],[1090335490,
    1063630973, 1073476422, 1035755503, 1050094400, 1081428824, 1027995780,
    1021164635]),
33:(['swim', 'galgel', 'eon', 'gap', 'lucas', 'ammp', 'equake', 'apsi'
    ],[1043985407, 1065211955, 1005038440, 1011662553, 1045691442, 1001412047,
    1015576344, 1084843544]),
34:(['vpr', 'twolf', 'apsi', 'vpr', 'mesa', 'applu', 'mgrid', 'fma3d'
    ],[1057562916, 1028743098, 1067453907, 1087649740, 1012866796, 1020715300,
    1050920626, 1034913162]),
35:(['vortex1', 'perlbmk', 'mesa', 'eon', 'lucas', 'equake', 'mesa', 'equake'
    ],[1029345593, 1016169734, 1013286006, 1002052717, 1080794866, 1033967832,
    1080996824, 1062109113]),
36:(['gap', 'eon', 'mgrid', 'gcc', 'parser', 'mesa', 'swim', 'bzip'],[1031957692,
    1025634021, 1051330127, 1019081762, 1002465854, 1015611877, 1081170265,
    1064397098]),
37:(['equake', 'mcf', 'galgel', 'crafty', 'bzip', 'ammp', 'vortex1', 'crafty'
    ],[1007445950, 1076581005, 1031048765, 1070570356, 1038710655, 1054579808,
    1014641175, 1092263752]),
38:(['facerec', 'wupwise', 'vpr', 'eon', 'sixtrack', 'bzip', 'perlbmk', 'art'
    ],[1017668229, 1062388660, 1045846686, 1064095596, 1044136084, 1049590197,
    1035253692, 1061796023]),
39:(['gzip', 'crafty', 'crafty', 'wupwise', 'gap', 'gap', 'eon', 'art'
    ],[1011530049, 1039498517, 1059870493, 1050430777, 1097320584, 1026210416,
    1065924402, 1024174983]),
40:(['vpr', 'mcf', 'mgrid', 'equake', 'galgel', 'mcf', 'facerec', 'gzip'
    ],[1052954017, 1084642384, 1029631047, 1045662487, 1014823135, 1017907122,
    1069887119, 1010792942])
},

}
```