An OS-Based Alternative to Full Hardware Coherence on Tiled Chip-Multiprocessors

Christian Fensch

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2008

Abstract

The interconnect mechanisms (shared bus or crossbar) used in current chip-multiprocessors (CMPs) are expected to become a bottleneck that prevents these architectures from scaling to a larger number of cores. Tiled CMPs offer better scalability by integrating relatively simple cores with a lightweight point-to-point interconnect. However, such interconnects make snooping impractical and, thus, require alternative solutions to cache coherence.

This thesis proposes a novel, cost-effective hardware mechanism to support shared-memory parallel applications that forgoes hardware maintained cache coherence. The proposed mechanism is based on the key ideas that mapping of lines to physical caches is done at the page level with OS support and that hardware supports remote cache accesses. It allows only some *controlled* migration and replication of data and provides a sufficient degree of flexibility in the mapping through an extra level of indirection between virtual pages and physical tiles.

The proposed tiled CMP architecture is evaluated on the SPLASH-2 scientific benchmarks and ALPBench multimedia benchmarks against one with private caches and a distributed directory cache coherence mechanism. Experimental results show that the performance degradation is as little as 0%, and 16% on average, compared to the cache coherent architecture across all benchmarks for 16 and 32 processors.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Christian Fensch*)

Related Publications

Parts of this thesis have been published in the proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 355–366, February 2008.

Acknowledgements

This thesis is the product of a journey that to some extend started in 2001. The accomplishment of this journey would not have been possible without the continuing support of the many friends and colleagues that I met or made on the way. As such, I feel that part of this success is due to their effort in providing me with much needed support throughout all this time.

I would like to express my thanks to my colleagues at the University of California at Irvine Niall Dalton, Fermin Reig, Jeffrey von Ronne, Christian Stork, Lei Wang, Ning Wang and Efe Yardımcı. In particular, I am grateful to Dr. Wolfram Amme and Prof. Michael Franz who convinced me to pursue postgraduate studies outside Germany.

I would like to thank my dear friends Carla Delgado-Battenfeld, Sytze Dijkstra, Christophe Dubach, Rosa Mendoza and Gaya Nadarajan in Edinburgh for their morale support and friendship. In particular, I am grateful that Ingo Battenfeld, Asuka Ishikawa and John Thomson not only shared the office with me but also provided support and encouragement at critical times.

Finally, I am greatly indebted to my supervisor Dr. Marcelo Cintra for his continuing support into my work and for his advice to always aim to publish at a top conference. After many failed attempts, I understand the insight in this and I am grateful that he was so insisting on this issue.

Contents

1. Introduction

1.1. Chip-Multiprocessors and the Cache Coherence Problem

Chip-multiprocessors (CMP) have now replaced very wide-issue out-of-order super scalar processors as they provide higher aggregate computational power, multiple clock domains, better power efficiency, and simpler design through replicated building blocks.

Current chip-multiprocessors are at the moment commonly built around a relatively small number of cores (2 to 8), each with its own L1, and possibly L2, cache, and are connected through an on-chip interconnect to a lower level shared cache. So far, the choice of on-chip interconnect has followed those of multi-chip symmetric multiprocessor (SMP) systems: shared bus fabrics and crossbars. Supporting shared-memory parallel applications requires cache coherence, which is greatly facilitated by the use of buses and crossbars in current CMPs. Such interconnects allow for straightforward hardware cache coherence mechanisms based on snooping [MMG$^+$06, MB05] and directories [KST04, KAO05].

Unfortunately, as pointed out in [KZT05], future technology scaling will lead to on-chip interconnects having different sets of tradeoffs and design issues than traditional off-chip interconnects. In particular, global wires do not scale down at the same rate as other features shrink, which means that either the delay or the area overheads, or both, of buses and crossbars increase (likely exponentially for the delay and polynomially for the area) as process scales. In fact, the detailed study in [KZT05] clearly shows that the area and delay overheads of buses and crossbars will become prohibitively high in CMPs with more than 8 or 16 cores in $65nm$ and smaller processes. In order to scale the number of cores in a CMP above this barrier a different approach will be necessary. In particular it will be necessary to resort to a scalable interconnect type.

Tiled CMPs [BKM$^+$04, CCC$^+$02, KBH$^+$04, SMSO03, TLM$^+$04] (also known as cellular multiprocessors and grid processors) have been investigated with the goal of how to build a larger processor by combining multiple small "cores" together. These systems were designed with different goals in mind: Some designs tried to provide an alternative to ever more complicated super scalar processors by finding new ways of exploiting ILP in sequential applications. Other designs focused on stream processing, where the data is passed from processing element to processing element and each processing element performs a part of the total computation. Tiled CMPs are built from a relatively large number (≥ 32) of relatively simpler cores plus a tightly integrated and lightweight point-to-point interconnect. Such interconnects are suitable not only because their peak bandwidth naturally scales with the number of cores, but also because, due to the short-length wires and low radix, their area overhead is a fixed, independent, fraction of the number of cores (unlike buses and crossbars where this overhead usually increases polynomially). Unfortunately, while these interconnects are very scalable, they do not lend themselves well to the implementation of snooping cache coherence protocols[1]. The alternative to continue enforcing cache coherence in such systems is to employ distributed directory schemes, which have been used in multi-chip multiprocessors in the past (e.g., [LLG+90, ASL03, LL97]). These have proven fairly scalable, reaching up to hundreds of processors. Snooping protocols are already somewhat difficult to completely debug and verify due to subtle corner cases and state transitions [Hag07][2], and distributed directories, with even more states, races, and corner cases, are notoriously even harder to debug and verify (e.g., [ASL03]).

Most of this complexity stems from the fact that requests cannot always be resolved at the home directory, but must in some cases generate further requests (such as forwarding and invalidation requests), which lead to complex protocols with subtle race conditions and several pending states. Moreover, distributed directory schemes are more involved in a CMP environment than they are in multi-chip systems. One reason for this is that in multi-chip systems the home node always has a directory entry for each and every memory line assigned to it as well as a placeholder for the data, while on a CMP each tile has only entries for the few memory lines that fit in its cache. Because caches are small and replacements are frequent, the situation where the directory information is not available on chip can occur more frequently, leading to longer pending state periods and further race conditions. Obviously, keeping more entries in each directory is possible (e.g., [BW04, CPV05, KBK02, ZA05]), at the expense of more directory storage per tile, but it still does not solve the problem of possibly frequent spills of directory state off-chip. All this complexity is of serious concern, as it is not clear how much design re-use is possible with respect to protocols and directory coherence controllers. If little design re-use is possible, and considering previous experiences with the design of such protocols, then designing and verifying the directory coherence protocol for each new generation of the CMP architecture will likely become an expensive bottleneck.

An alternative to enforce coherence in a distributed memory system is to use the OS' virtual memory (VM) system to handle the copies of virtual pages, as was done on software DSM systems (e.g., [CBZ91, KHS+97, Li88, RLW94]). In this scheme, all caches are private and it is the responsibility of software to maintain coherence. As with distributed directories, such schemes have only been tested on multi-chip systems and must be adapted to operate on a CMP. A major drawback of directly porting software DSM schemes to the CMP environment is that such schemes require moving, comparing (*"diff"*), and copying data in physical memory pages to enforce coherence. This is because creating multiple physical copies of the same virtual page is the only way to cope with false sharing and the inability of the hardware to identify which parts of a cache line have been modified. In this way, at communication points, such as lock transfers and barriers, the individual copies must be compared against the previous stable copy of the page and the modifications must be merged into a single new stable copy of the page. These operations are likely to be extremely costly in a CMP, will consume precious off-chip memory bandwidth, and generate much pollution in the relatively small on-chip caches. In fact, previous work on software DSM systems showed that the costs of managing the multiple copies of pages, generating diffs, and updating pages, correspond to a significant fraction of the costs in these systems [ISL96].

[1]Promising recent research has attempted to implement snooping-like protocols on top of scalable interconnects [MHW03, MH06], but their tradeoffs are still open to investigation.

[2]Further suggestion to the difficulty of complete verification is the recent Core 2 Duo Errata AI39: "Cache Data Access Request from One Core Hitting a Modified Line in the L1 Data Cache of the Other Core May Cause Unpredictable System Behavior" [Int07]. While it is not officially stated as so, this clearly suggests some nagging bug in the coherence protocol implementation, which was only identified after product shipping.

Overall, the potentially complex hardware solution of distributed directories and the potentially high-overhead software-only solution of a VM-based scheme are two extremes in the spectrum of solutions for the cache coherence problem in tile CMPs.

1.2. Contribution of this Thesis

This thesis proposes an alternative cost-effective software/hardware mechanism to support shared-memory parallel applications that forgoes hardware maintained cache coherence. The proposed mechanism is based on the key ideas that mapping of lines to physical caches is done at the page level with OS support and that the hardware efficiently supports remote cache accesses. An extension of the basic scheme only allows some *controlled* migration and replication of data. Data is migrated by refreshing the page mappings at barriers. Read-only sharing is done with the help of the existing write-protection mechanisms in the TLB/OS. Overall, the mechanisms allow a sufficient degree of flexibility in the mapping and sharing. This thesis also addresses in depth some issues that arise from the implementation of the technique, such as the implementation of memory locks.

By moving the key coherence handling and decision making to software (in our case the OS), the proposed scheme, like software-managed coherence mechanisms [CH04, KOH$^+$94], benefits from the possibility to modify the protocol after hardware shipping, which may allow for customising the protocol to application behaviour and for more easily fixing bugs. Like other recent attempts to divide coherence labour between OS/software and hardware [ZH07, ZRKH06], the mechanism is likely to be more cost-effective and easier to verify and validate than distributed directory schemes. Unlike such previous trap-based schemes, however, the small hardware extensions to support an extra level of indirection between virtual pages and tiles as well as to support remote cache accesses minimise the need for OS and trap handler activity. In the proposed scheme, only the processor's first load or store to data in a page requires trap handler intervention and only the system's first load or store to data in a page requires full OS intervention. Also, unlike recent hardware-only schemes for co-operative distributed caching [BW04, CPV05, KBK02, ZA05] the proposed scheme does not rely on broadcasts, centralised tag stores, or large redundant tag stores in order to map, locate, and access data cached remotely.

The proposed tiled CMP architecture is evaluated on benchmarks from two very different domains – the SPLASH-2 scientific benchmarks and the ALPBench multimedia benchmarks. The system is compared against one with a distributed directory cache coherence mechanism. Experimental results show that the proposed scheme performs very close to this system with a performance gap as close as 0% (no gap) and 16% on average, across all benchmarks for 16 and 32 processors.

1.3. Summary of Work

The basic idea of this thesis is that instead of keeping the distributed caches coherent, they are prevented from becoming incoherent by simply disallowing data to be stored in more than one cache. However, unlike previous work, this work does not move the data to another cache in order to enforce the invariant. Instead, other processors are allowed to access the cache, which stores the data, remotely. The allocation of which data is stored in which cache is done at page granularity. This information is stored in an OS managed table that is similar to the page table (used to translate virtual to physical addresses). This basic scheme is then extended to allow some form of migration and read-only sharing.

1.4. Structure of this Thesis

The remaining of this thesis is structured as follows: Chapter 2 to 4 discuss the motivation for this work and present important background information. Chapter 5 discusses software DSM systems, which are somewhat between related work and background information and as such deserve a separate chapter, apart from the other related work presented later in chapter 14. Chapter 6 to 9 present the baseline architecture and in detail the proposed architecture extensions. They also discuss support for synchronisation operations and how these operations are implemented in the proposed architecture. Fictional models, which are used to evaluate the performance of the proposed architecture, are presented in chapter 10. Chapter 11 presents the applications used to evaluate the proposed architecture. Chapter 12 explains the simulation setup, while chapter 13 discusses the performance of the proposed system in detail. Finally, chapter 15 concludes this thesis, and shows future research directions.

Part I.

Background

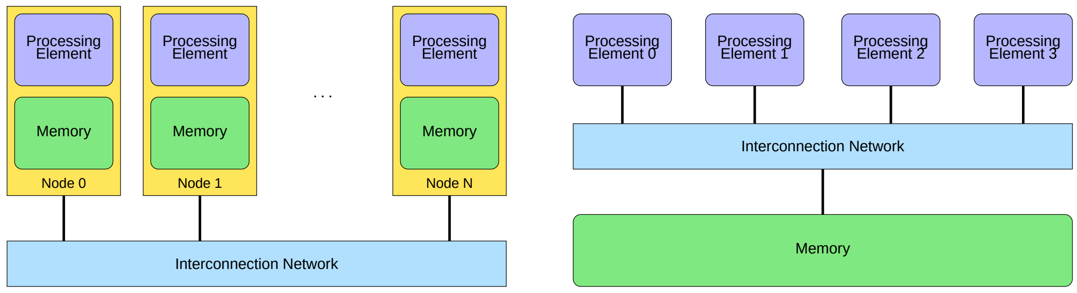
2. Parallel Programming

Parallel programming assumes that a problem can be split into several independent parts that can be executed in parallel. These parts are then run as independent execution threads (the term thread is used here in a broader sense and should not be confused with “thread” as it is used in the context of operating systems). Although there are cases where these threads are completely independent and do not communicate with each other, in most cases some form of communication is necessary. In general there are two different approaches to how this communication is performed. One approach assumes that a shared address space exists that can be used for communication; the other approach assumes a distributed address space. Having a shared address space makes developing parallel applications more similar to developing traditional sequential applications. All programming constructs from sequential programming are still present. In particular the usage of pointers is still possible. For example, one thread might pass a pointer to another thread as part of the communication between them. Similarly a thread might just follow pointers within a linked data structure. However in the distributed memory approach things are not that simple. The destination of a pointer might be different for a different thread. For example a pointer to `0x10000000` might point for one thread to the beginning of a linked list element, while on another one it points to the middle of the data structure. Due to this problem, threads cannot use pointers when communicating with each other. Instead, the whole element has to be sent as a message from one thread to the other. This restriction makes it more complicated to pass information that is stored in pointer based data structures from one thread to another.

On the actual hardware side, these two approaches have been matched with machines that lend themselves particularly well to one of the approaches. Shared memory machines are a good match for the approach that assumes a shared address space, while machines with distributed memory are more suited for the approach that does not assume this shared address space. However, it should be noted that either machine is able to also support the other approach. A shared memory machine can communicate by messages through the shared address space without using pointers. And a distributed memory machine can use an additional software layer that provides the illusion of a shared address space. The following sections describe these programming schemes in more detail and highlight their respective advantages and disadvantages.

2.1. Message Passing

Message passing systems have been developed as a solution to combine the computational power of several smaller systems in order to process larger programs. Usually each system is a self contained computer with its own processor and memory. These nodes are then connected using some kind of interconnect. The basic structure is shown in figure 2.1a. The whole computation is now broken into smaller parts that are assigned to individual nodes. Each node performs a



(a) Message passing system. (b) Shared memory system.

Figure 2.1: Overview of basic parallel systems.

part of the computation and then sends the intermediate results to other nodes that depend on these results.

One advantage of message passing systems is that they scale to thousands of nodes (for example the largest Blue Gene/L installation features 106,496 nodes). Furthermore, since it exposes communication directly to the programmer, he or she can optimise the algorithms to the communication pattern. However, this exposed communication is also the obstacle for the widespread use of message passing systems. Programmers have to handle data placement (including replication) and migration themselves. These tasks are non-trivial and sometimes as demanding as the development of the program itself. Still, it seems that there is today no alternative programming model to write applications for systems with thousands of nodes other than message passing.

2.2. Shared Memory

The Shared Memory model has first been used in early multi chip multiprocessors that connect several CPUs via a shared bus to a memory system (shown in figure 2.1b). This model simplifies it for programmers to write multi processor programs, since it removes the burden of distributing data across different nodes and accessing data that is stored on a remote node. Since CPUs in these early systems did not have local caches or out-of-order execution, the shared bus could act as serialisation point for all memory requests. All write accesses become immediately visible to all processing elements.

One problem has been created with the introduction of hardware caches: processing elements keep local copies of data for fast access under hardware control. This data might no longer be identical to the data stored in main memory: writes by other processors to shared memory might not be reflected in the local cache. Similarly writes to the local cache might not be written back to shared memory that is visible to all.

Another problem is caused by out-of-order execution, since it allows processing elements to reorder instructions differently from the original order specified in the program. While a processing element ensures that by doing so the semantic is not changed with respect to single processor execution, the situation becomes more complicated once out-of-order processing elements are used in a multi processor system. Imagine a program that spins on a memory variable before it reads some other value from memory. An out-of-order processor might not detect a dependency for the second load and issue it before the spinning load.

These problems have prompted research into consistency models, their relations with each other and programming implications. These models will be briefly discussed in sections 2.2.1 to 2.2.3.

In terms of scalability, the largest shared memory system built so far consists of 512 nodes (SGI Altix 3000). While Cray X1E allows up to 1024 nodes with 4 processors each, no such system has been installed so far.

2.2.1. Sequential Consistency

Sequential Consistency (SC) is probably the oldest and most intuitive consistency model that exists [Lam79]. The model makes two requirements: firstly, all memory requests issued by an individual processor appear in the order specified by the program. Secondly, all memory requests from all processors appear to be serviced from a single FIFO queue.

While these requirements make programming fairly straight forward, they limit hardware scalability, instruction level parallelism, etc. For example, a single FIFO queue would soon become a bottleneck. Based on these observations two research areas have started: Ways to support more sophisticated hardware while still maintaining Sequential Consistency and research into other consistency models. The first area includes techniques such as cache coherence protocols (discussed in more detail in chapter 3), memory barriers to prevent reordering of memory instruction across the barrier and speculation based approaches that require a rollback once a modification to an address with an out-of-order load is detected [GFV99, Yea96].

Also, since enforcing Sequential Consistency in hardware can be very costly, research started into other consistency models that are less restrictive than Sequential Consistency and either still allow an intuitive way of programming or implement Sequential Consistency using library functions. Providing a full list of other consistency models is outside the scope of this thesis. Adve and Gharachorloo have compiled a list of several consistency models, their implementation and programming implications [AG96]. Steinke and Nutt have compiled a similar list, but also investigate the relation between different consistency models [SN04]. The next sections present some important consistency models that will be referred to in later parts of this thesis.

2.2.2. Weak Consistency

Even though Sequential Consistency is a very intuitive model, its main problem is the additional cost required to maintain the illusion of serialised accesses to memory. It has been noted that requiring this for every memory operation is mostly unnecessary. Thus, models were developed that do not enforce the requirement of sequential access for all memory accesses, but still offer a sufficient intuitive behaviour to be useful to programmers. Dubois et al. [DSB86] noticed that it was enough to ensure Sequential Consistency for access to synchronisation variables and make a few further restrictions on other memory accesses. The model they proposed was called Weak Consistency and requires that:

- Access to shared synchronisation variables is sequentially consistent.
- No access to a shared synchronisation variable is performed before all previous memory requests to shared data have been completed.
- No access to normal shared data is performed before all access to shared synchronisation variables have been completed.

These conditions ensure that a normal access to shared data will either be before or after a synchronisation operation. All processing nodes must see these normal accesses occur in this order with respect to the synchronisation operation. These simple restrictions match the requirements for locks and barriers very well and as such allow an easy implementation of these synchronisation constructs.

2.2.3. Release Consistency

The above restrictions might be sometimes too strict, since they just assume one kind of synchronisation operation. Instead synchronisation operations are usually used for different purposes: some instructions are used to import updated information (such as the acquire of a lock); other instructions are used to export information (such as the release of a lock). Gharachorloo et al. [GLL+90] used this observation to define a more relaxed consistency model called "Release Consistency" (RC). The model requires that:

- Before a normal load or store operation is allowed to perform[1] with respect to any other processor, all previous acquire accesses must be performed.
- Before a release access is allowed to perform with respect to any other processor, all previous normal load and store operation must be performed.
- Special accesses (such as acquire and release) are sequentially consistent with respect to one another.

An acquire operation guarantees that the process is provided with all required updates. There are two different flavours of release consistency:

Eager Release Consistency requires that all updates be performed when an acquire operation happens.

Lazy Release Consistency tries to delay all updates as much as possible [KCZ92]. The idea is to only make memory updates visible if these are really needed. For example, the arrays `a`, `b` and `c` were modified and thread x is acquiring a lock. Eager Release Consistency would now require that all modifications are made visible to thread x. However, this might be unnecessary, if thread x only accesses array `b`. Thus, with Lazy Release Consistency thread x is only informed that `a`, `b` and `c` are modified. Once an actual access to one of these arrays in performed, then the modifications to that array are made visible to thread x.

Making modifications visible usually involves communication between different threads. By reducing the number of modifications that are made visible, Lazy Release Consistency is able to reduce the amount of communication that is required.

Release consistency is the consistency model that is used by a wide range of languages, such as Java [GJSB05], OpenMP [Ope05] and Unified Parallel C [UPC05].

[1]The expression "to perform" refers to accessing the memory and either reading or storing a value to/from a register. An access is considered "performed" once its outcome can no longer be affected by another access.

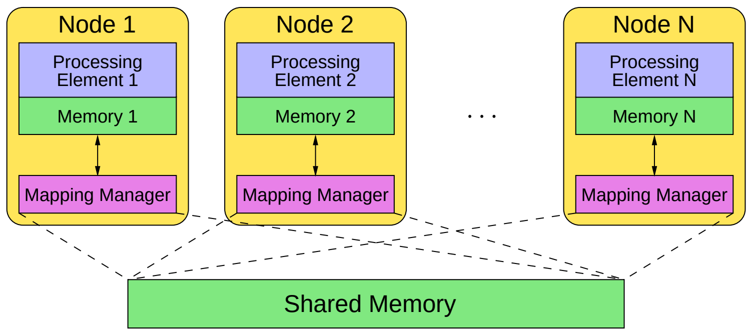


Figure 2.2: Overview of a distributed shared memory.

2.2.4. Processor Consistency

Another consistency model that is rather relevant in practice is "Processor Consistency". It is the model that is adopted by most processor nowadays. It has been defined by Goodman [Goo89] and requires the following: "a multiprocessor system is said to be *processor consistent* if the result of any execution is the same as if the operations of each processor appear in the sequential order specified by its program." Thus while writes by the same processor are always observed in the order specified in the program by all other programs, writes performed by different processors can be observed in different order by different processors. For example, processor X might see the write by processor A first and then B's write. However, from processor Y's point of view, it was B who wrote first and then A.

2.3. Distributed Shared Memory

Distributed shared memory systems try to offer the view of a shared memory system (as shown in figure 2.1b) on a system that in reality looks like a system as shown in figure 2.1a. In order to provide the illusion of a single shared address space a mapping manager is necessary (as shown in figure 2.2). Several solutions exist for implementing this manager; some of these solutions are implemented completely in hardware, others only in software. These systems will be discussed in more detail in chapter 3 and chapter 5.

3. Cache Coherence

The Cache Coherence problem affects shared and distributed shared memory systems that use hardware caches. It is illustrated in figure 3.1. The variable `x` with the original value 10 is loaded from main memory into the cache of processing element 1 (❶). Next, processing element N also loads the same variable (❷) and updates its value to 99 (❸). The questions are now: which value will be loaded into the cache of processing element 2, if it tries to load the variable in its local cache (❹)? What happens if processing element 1 accesses variable `x` again (❺)? For example, if the system is supposed to be sequentially consistent, then the system should return the updated value of `x`, which is 99. Cache coherence is the mechanism that will update the cached value eventually. The exact moment depends on the implemented consistency model.

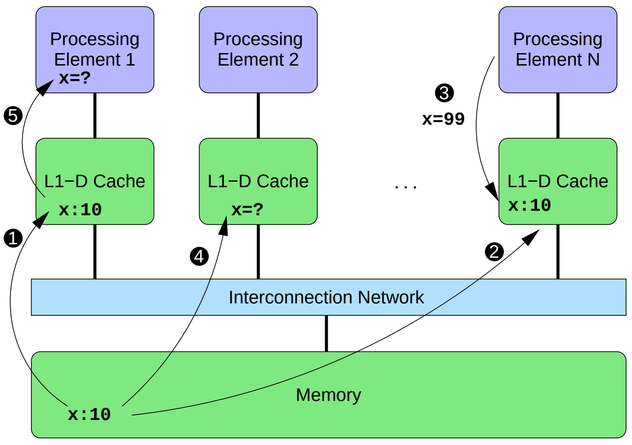


Figure 3.1: Overview of the cache coherence problem in a shared memory system. The variable `x` is loaded into several caches and modified as indicated by the numbers.

A simple solution to the problem is to avoid caching shared data. If all accesses to shared data are directed exclusively to main memory, then the cache coherence problem does not arise. The Cray T3D and T3E were machines that followed this strategy by only caching access to local data [Sco96]. The problem with this approach is that it is very difficult for the compiler to identify non-shared data in order to allow caching of it. Thus, in the end, the compiler often decided to mark almost all data as non-cacheable, which decreased the available system performance significantly.

Hence, other solutions are needed that support caching of data. The following sections will discuss the most common solutions and why they will not be applicable to large scale CMPs. This chapter will then be concluded with a solution that has been developed for multi-node distributed shared memory systems.

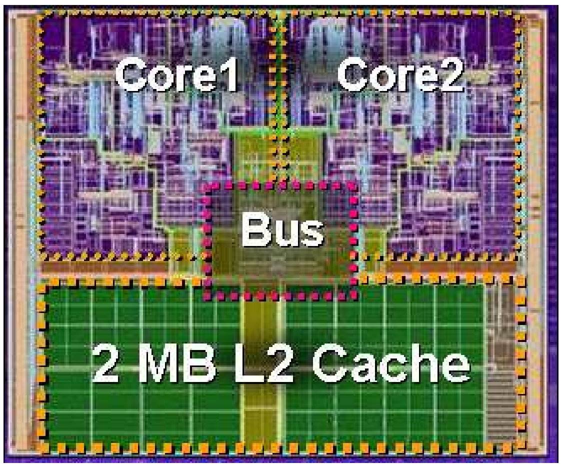
3.1. Snooping

All snoopy based cache coherence protocols have one thing in common: the caches are all connected to the same bus and can observe transactions that happen at the other caches. The two basic strategies of either updating all copies or only allowing a single valid copy of a cache block are described next:

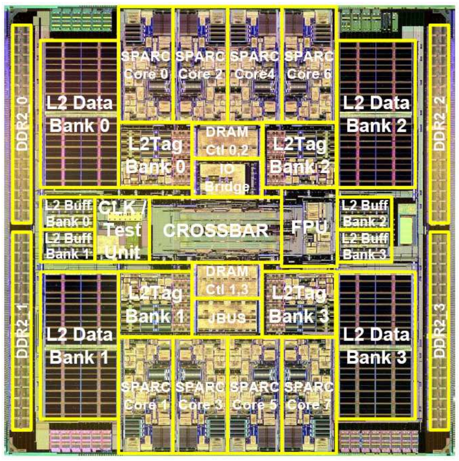
Write Update protocols are the simplest snooping protocols. The basic idea is that a write-through cache is used. Thus all writes are visible to all caches and each cache can update its local data, if it happens to share that cache block. The obvious disadvantage of such a protocol is the increased usage of the shared bus, since all writes go directly to memory. Furthermore, the caches will be quite busy monitoring each message on the bus, which might have an impact on offering low latencies to the local processing element.

Write Invalidate protocols ensure that there is only a single valid copy of a cache block, once it is modified. This is done by sending an invalidation message on the shared bus once a write to a cache block happens. In order to reduce the number of invalidation messages sent, each cache block also stores a state: *M-odified*, *E-xclusive*, *S-hared*, and *I-nvalid*. The states *exclusive* and *modified* indicate that the current cache is the only cache that currently has a copy of the block. Thus no invalidation messages have to be sent in case of write access to this block. Goodman was the first to describe such a scheme in the literature [Goo83]. This scheme is also referred to as MESI protocol, based on the initial letters of the cache block states. While there have been a couple of slight modifications to this basic scheme, it would be beyond the scope of this thesis to discuss them in detail. Further details are available in [AB86].

While snooping protocols have been originally developed for multi chip multiprocessor systems, they are now used in chip-multiprocessors as well. For example, Intel's Core 2 Duo (see figure 3.2a) uses a MESI protocol to keep the two L1 caches coherent. Intel decided to implement a snooping protocol as opposed to a directory based one, due to its reduction in complexity [MMG+06].



(a) Intel Core2Duo [GMNR06].



(b) Sun Niagara [LST+06].

Figure 3.2: Floorplan of two current chip-multiprocessors

3.2. Directory Based

While snooping based protocols lend themselves particularly well to implementation on early bus based shared memory systems, the bus limits the scalability of the system and is destined to become a bottleneck as the number of processors increases. Thus, larger systems usually use multi-staged point-to-point interconnects to allow communication between attached devices. Unfortunately, these networks do not lend themselves to the efficient implementation of broadcasts, which all snooping based protocols rely on.

As an alternative solution, Censier and Feautrier [CF78] proposed a scheme that uses a directory to locate copies of a memory block. Interestingly, this scheme was developed for shared bus systems as a mechanism to reduce the overhead of unnecessary broadcasts. In principle a directory allows only one cache to contain a writeable copy; all other copies must be invalidated before a write can proceed. The basic idea is that for each memory block a directory entry is present. This entry contains a *present* bit for each processor to indicate, if this processor currently caches this memory block. Additionally, each cache line contains a *private* bit to indicate that this cache exclusively holds the only copy of the cache line.

By using this directory, messages no longer have to be broadcasted to all devices but can be sent just to affected ones. The obvious problem with this approach is the extra storage required for all the *present* bits. Several schemes have been proposed to limit the amount of extra storage required, such as the limited directory [ASHH88] or the chained directory [JLGS90]. Another problem with the directory approach is that some transactions require multiple, possibly lengthy sub-transactions, which complicates the directory implementation.

As with snooping protocols, this protocol has been originally used in multi chip multiprocessor systems. However, directory based schemes can be found in current CMPs. For example, Sun's Niagara processor (see figure 3.2b) uses a directory based protocol to keep the private L1 cache of each processing element coherent [KAO05]. The directory information is stored in the shared L2 cache.

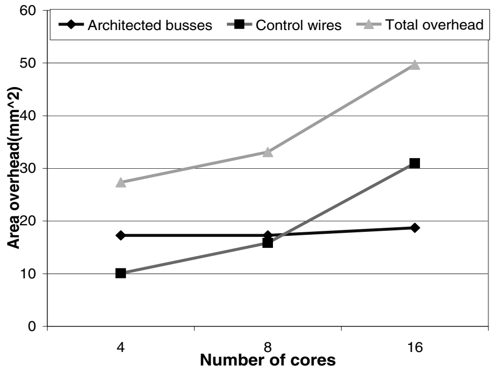
An extension of the directory based protocol for distributed shared memory will be discussed in section 3.4.

3.3. Scalability of Buses and Crossbars

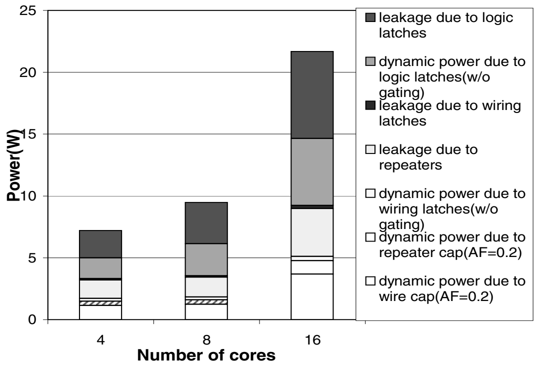
Considering the importance of shared buses and crossbars for current coherence protocols on chip, it is important to see how these interconnects scale with an increasing number of cores. Kumar et al. [KZT05] investigated in detail the effect on area, power and latency in this situation. As a baseline they assumed a $65nm$ process with $400mm^2$ die. These assumptions are mostly in line with current CMPs like Sun's Ultrasparc T2 and IBM's Power6. Both processors are manufactured using a $65nm$ process, but occupy slight less area ($340mm^2$). Each core is considered to be a stripped version of a Power4 core requiring $10mm^2$ of area and $10W$ power (including leakage). All remaining area is assumed to be used as L2 cache.

3.3.1. Shared Busses

The first study in [KZT05] focused on shared buses, with the first results that using a single bus to connect 16 cores is latency wise not feasible. Instead, they suggested to use 2 buses that just connect 8 cores each and are then joined by a peer-to-peer link. The effect on area is



(a) Area overhead for a shared bus.



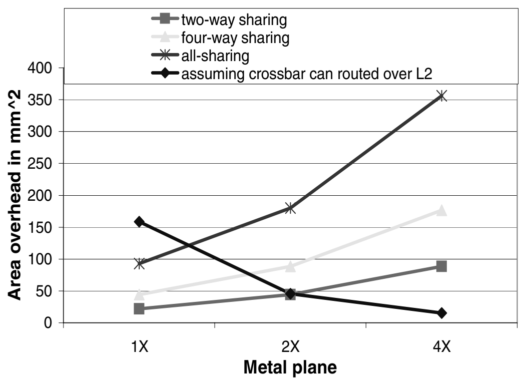
(b) Power requirements for a shared bus.

Figure 3.3: Area overhead and power requirement for a shared bus [KZT05].

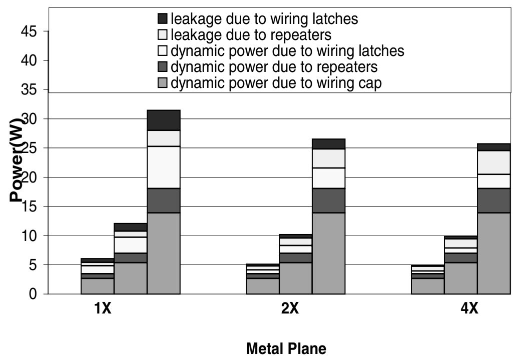
shown in figure 3.3a. Please note that sometimes certain components do not result in a direct area overhead, since the additional wires can be routed in a different metal layer across other components. The area overhead is 7.2% with 4 cores, 8.7% with 8 cores and 13% with 16 cores. This is an equivalent area of 3-5 cores or 4-6MB of cache.

Another interesting thing to notice is the increase in power consumption (see figure 3.3b), with most of the power being consumed in the logic of the shared bus and not its wires. As for accessing data in the L2 cache, over half of the access latency would be due to latencies in the shared bus (assuming no contention on the bus). The performance degradation caused by the shared bus is 10% for 4 cores, 13% for 8 cores and 26% for 16 cores.

3.3.2. Crossbars



(a) Area overhead for a crossbar.



(b) Power requirements for a crossbar. The 3 bars correspond to 2-way, 4-way, and all-way sharing.

Figure 3.4: Area overhead and power requirement for a crossbar connecting 8 cores with 8 banks of cache [KZT05]. The metal plane 1X, 2X and 4X refer to lower, intermediate and upper metal layers that are used during chip manufacturing process. Lower metal layers contain very thin, short running wires, while upper layers contain thick long running wires. As such, wires in the upper layers have lower signal latency than wires in the lower layers. In particular, 1X contains metal layers 1 to 4, 2X contains metal layers 5 and 6, and 4X contains metal layers 7 and 8.

Kumar et al. [KZT05] only investigated crossbars that connect 8 cores to 8 banks of cache. For the crossbar there are two design parameters: first, in order to increase single thread performance, it is important that a single core can access as many cache banks as possible in order to increase the amount of cache available to it. Second, the metal layer the crossbar is routed in has impact on its latency and wire thickness. As a rule of thumb, going to a higher metal plane reduces the latency by almost half, but also doubles the wire thickness. Another design point is the decision of where to place the crossbar. If it is placed next to the core and caches, then wiring will be easy but all components result in an area overhead. If the crossbar is placed above the cache, then wiring becomes much more difficult, but its overhead is greatly reduced.

Figure 3.4a shows the area overheads in this design space. Thus, for a crossbar with acceptable latencies (wiring in the 2X plane) the area overhead is 11.4% for 2-way sharing, 22.8% for 4-way sharing and 46.8% for all-way sharing. A similar picture also displays itself for the power consumption. With an increase in the level of sharing, so does the power consumption increase (figure 3.4b).

Another aspect of these results is: considering that sharing of caches should increase the amount of cache available to a single core, then it might be more useful to use a lower degree of sharing and use the saved area to increase the size of the cache banks.

Considering the area requirement of this crossbar it seems improbable that a crossbar can be used to connect 16 cores due to the expected area overhead.

3.4. Distributed Directory Cache Coherence

The protocols discussed in section 3.1 and 3.2 were designed to provide cache coherence in shared memory systems as shown in figure 2.1b. In order to provide cache coherence in system with distributed shared memory (figure 2.1a), a different solution is needed. One of the main design decisions for these systems was to avoid the bottleneck to the memory system as was present in shared memory systems. Thus while using a centralised directory would solve the coherence problem, it would also prevent scalability of the system.

In order to address these scalability problems, protocols with distributed directories were developed. The first of such systems was the DASH research system [LLG+90] developed at Stanford University. Similar to centralised directories, the main idea is to separate the global sharing state from the caches holding the data. However, in absence of a centralised directory the additional problem of locating the directory for a cache block arises. The solution is that cache blocks are associated, usually at page granularity, with a particular directory. The protocol introduces the following terms:

Home node contains the directory information for a particular page, as well as the actual physical memory for the page.

Remote node is any other node that is not the home node.

Owner node is usually the home node. However, if a cache block is in a dirty state on a remote node, then this node becomes the owner node. Note that there can be only a single owner node for any cache block. Also, only the owner node of a cache block is allowed to update the sharing state regarding this block in the directory on the home node. Such an update then also indicates that the owner node is giving up ownership.

A typical cache miss is resolved as follows: the node determines the home node of the accessed page based on the physical address of the TLB lookup. It then sends a message to the home node (assuming that the accessing node and the home node are different). The action of the home node then depends on the type of the request, and if the home node is also the owner node of that cache block. For example, if the home node is not the current owner, then it has to forward the request to the current owner. Or, if the node requests exclusive access to the cache block, then the home node has to inform all other sharers (for local invalidation purposes) before the request can be granted.

The home node is usually identified by the physical address of a page in the TLB. Thus, the physical address might be in the address range of memory that is really installed in the node. In such a case the home node is the local node. If the physical address is outside the address range of locally installed memory, then the home node is some remote node. Which remote node is the home node can be identified by looking at specific bits in the physical address. The home node itself is usually dynamically assigned using a first-touch policy.

Even though distributed directory protocols can be used to implement sequential consistency, DASH decided for performance reasons to settle with release consistency. While read operations are blocking, writes are performed in a non-blocking way. This might result in a later write being committed to memory first. This situation can arise, if the write operations are handled by different directories. While the first write might still wait for the directory, the second write got already permission to commit. In this case, the order of writes observed by other nodes would be different from the node that issued the writes. In order to provide support for stronger consistency models (such as sequential consistency), DASH offers support for memory fences (a fence ensures that all memory operations have been committed, before allowing the processor to continue). However, it is the responsibility of the programmer or compiler to insert these fence operations at appropriate places into the program. Performance issues, such as this, had the effect that a quite a few distributed shared memory machines used weaker consistency models than sequential consistency by default.

Apart from the originally mentioned DASH research system [LLG$^+$90], a distributed cache coherence protocol has also been used in the following machines. The SGI Origin 2000 [LL97] (and its successor, the Origin 3000) is based on the DASH protocol, but uses several performance improvements. An extra state (*clean-exclusive*) for a cache line has been added. This state is similar to the *exclusive* state in the MESI protocol. Furthermore, the Origin 2000 can upgrade a cache line from *shared* to *exclusive* without certain overheads that exist in the DASH protocol. The final improvement involves a more sophisticated algorithm to deal with deadlock avoidance. Figure 3.5 shows the performance of the SGI Origin 2000 for several SPLASH-2 benchmarks. The Origin has been designed to scale up to 512 nodes with 2 processing elements each. Both protocols seem to have some issues with negative acknowledgements [GSSD00]. While the Origin 2000 resorts to some complicated mechanisms (such as reverting to a strict request-reply protocol) to resolve this, the DASH protocol simply seems to ignore this.

The other commercial machine is the Cray X1 (and its successor, Cray X1E). The machine scales up to 1024 nodes with 4 processing elements in each node. The Cray X1 also uses a relaxed memory model, which does not offer sequential consistency or any other commonly used form of consistency [ASL03].

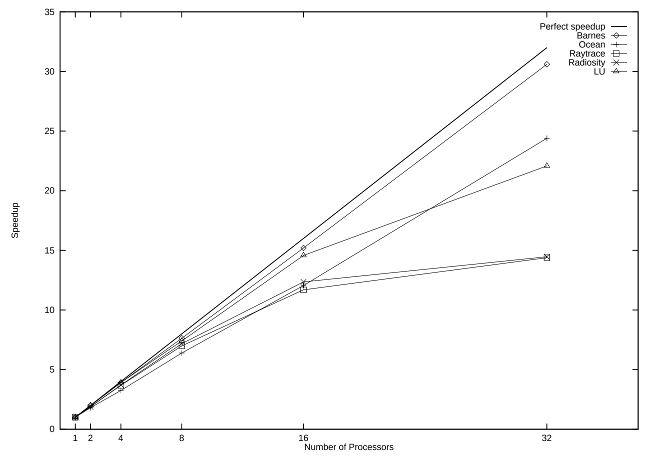


Figure 3.5: Performance of the SGI Origin 2000 on several SPLASH-2 benchmarks [LL97].

3.5. Complexity of Distributed Directory Cache Coherence

Considering the increase in complexity from simple bus based snooping protocols, to directory based protocols and finally distributed directory based protocols, one important question is the verification of such a protocol and its hardware implementation. Due to the complexity of the protocol, it is impossible to enumerate all possible states of the system and verify the correct behaviour in a reasonable amount of time. Instead, other methods are required. The DASH protocol was extensively tested using an FPGA implementation of the protocol controller and the SPLASH benchmarks. While such an approach will probably find most obvious mistakes, it is unlikely that it will find corner cases[1], since it suffers the same drawback as other testing based approaches: exhaustive testing requires an enumeration of all system states, which is not practical for any but the most simplest systems. For example, the FLASH protocol (discussed later in this paragraph) was tested extensively for 7 years. Still, several critical errors were found in the protocol after this time. Eiríksson [Eir96] used a model based approach to verify the SGI Origin 2000 protocol. However, he had to use a simplified model at a higher abstraction layer, ignoring many details to make the verification tractable. Furthermore, he was only able to verify a system consisting of 3 nodes. Abts et al. [ASL03] used also a model based approach to verify the Cray X1 protocol, and also had to simplify this model. However, they were able to record the protocol actions and use this “witness string” to verify the correctness of the Verilog implementation. Still, even with these simplifications the total state space of the Cray X1 would haven been 2^{1664} states. Luckily, only about 200 million states are actually reachable. In the final result this approach was able to find several implementation flaws in one of the cache coherence controllers. Unfortunately, the approach was still too slow to be extended to the other cache coherence controllers. Lie et al. [LCED01] used also a model based approach to verify the correctness of the cache coherence protocol in the FLASH multiprocessor [KOH+94]. Since FLASH uses a programmable protocol controller, it was possible to extract the model semi-automatically from the C source files by extending the compiler. Problems that prevent a fully automated extraction of the model include the weak C type system and bit operations, which are not supported by the model checker and have to be emulated. Using this approach

[1]For example, a cache line not being invalidated correctly. However, this mistake never has any impact, since the application does not access the cache line before it gets evicted from the cache.

it was possible to extract a model for 4 protocols (Unfortunately, the authors do not mention how many protocols are there in total.). The extracted model is then verified using a four node configuration. Lie et al. were able to find 6 critical errors in the 4 protocols, even though these have been tested and used for 7 years. The main obstacle to port this semi-automatic approach to other systems seems to be the need of a protocol implementation in an imperative programming language. The final conclusion is that no one was able to completely verify a distributed cache coherence protocol.

So far no chip multiprocessor uses a distributed directory cache coherence protocol. Thus, one of the open questions is how to adapt such a protocol in order to work in a CMP environment. As pointed out in [KZT05] the tradeoffs for on-chip networks are different from inter-chip networks. Traditionally distributed directory protocols were developed for a scenario in which, first, the network communication speed is relatively slow compared to CPU speed, and second, there is sufficient memory available on each node to store the directory information. However, on CMP things are reversed: the network is clocked at the same speed as the CPU, thus it is fast enough to consume messages produced by the CPU at the same rate as they are produced. On the other hand, fast on-chip memory is relatively limited. Considering the lack of verification of distributed cache coherence protocols for multi-node system, this thesis claims that there is no reason to assume why the situation should be different when moving to a CMP.

Considering one would like to use the unmodified protocol of an existing commercial system, such as the Origin 2000, for a CMP, the first thing to consider is how components in the Origin 2000 relate to components in a CMP. There is no problem to associate the on-chip interconnect on the CMP with the inter-node interconnect in the Origin 2000, the cores on the CMP with the nodes of the Origin 2000, and processing elements in each core with the CPU in each node. A home node of the Origin 2000 stores the sharing information in its main memory. In the CMP environment the cache in each core would be responsible to store this information. This association has the following implications: the first issue arises due to fact that cache entries are frequently evicted from the cache. Since this will also evict the sharing state information, it requires some special attention: either all remote copies of the cache line have to be invalidated before it can be evicted, or the sharing state information has to be saved in main memory and reloaded when necessary. Either solution will add extra states to the cache coherence protocol and, thus, complicate the protocol even further. Note that this issue does not arise in the Origin 2000, since it has enough main memory to store the sharing information for all memory blocks. Thus, no sharing state information is ever evicted. The second issue limits the efficiency of the cache on the home node. Since the home node stores the sharing state information in the cache, it must keep space in the cache for this information, even for data that is only accessed remotely.

4. Tiled Architectures

As seen in the previous chapter, future large scale CMPs are unlikely to rely on shared buses or crossbars in order to scale to the envisioned number of cores. Instead an alternative approach to design scalable many-core processors is needed. Cellular architectures[1] already investigated this problem. However, while they now look like large scale CMP architectures, they were originally a solution to two different problems: first, building ever more complicated super scalar processor will soon hit a complexity wall that makes it impossible to verify these designs. Second, the available ILP that can be exploited by super scalar processors is limited (with recent advances in super scalar designs delivering diminishing returns in performance). Thus, new ways of CPU design might be necessary to put all the available transistors to use.

Cellular architectures try to address these problems by composing a CPU from several cells. Since these cells are rather small in size, wire delays can be ignored within a cell. Also due to the small size, the complexity of each cell is rather limited, making verification of each cell design rather simple. The whole CPU is then designed by replicating many of these cells and connecting them together. The general idea is that a signal can reach all parts within a cell in the same cycle. As for communication to neighbouring tiles, messages usually take one cycle per hop. Additional cycles might be required to setup the communication.

Another characteristic of cellular architectures is the absence of central structures such as result buses or a central control unit. While, of course, some central control is needed, it either has to be provided by the software stack or by a unit that can deal with the fact that it will not be able to broadcast control information across the whole chip within one extra cycle.

What is a Cell? There is no simple answer to this question. Different cellular architectures encapsulate different functionality within a single cell. Some place a whole CPU core with caches within a cell, while others just place a single functional unit. Also some architectures use homogeneous cells while others use heterogeneous ones. Common among all architectures are that cells are relatively small and simple structures. Thus, wire delays can be neglected within a cell and the design can be easily verified. In addition, since every cell is rather small, a large number of them will fit on a single chip. For example, describing a 2-core chip multiprocessor (CMP) like the AMD Athlon 64 X2 as cellular microprocessor with two cells would be inappropriate.

As such, by looking at proposed tiled architectures, it should be possible to derive future large scale CMPs designs. The following sections discuss several such designs that have been proposed and some of them have been implemented. The final section in this chapter then concludes by mentioning some of the characteristics of such large scale CMP design.

[1] Another name for cellular architecture is tiled architecture. Thus, the names “tile” and “cell” are used as a synomym throught this thesis.

4.1. RAW

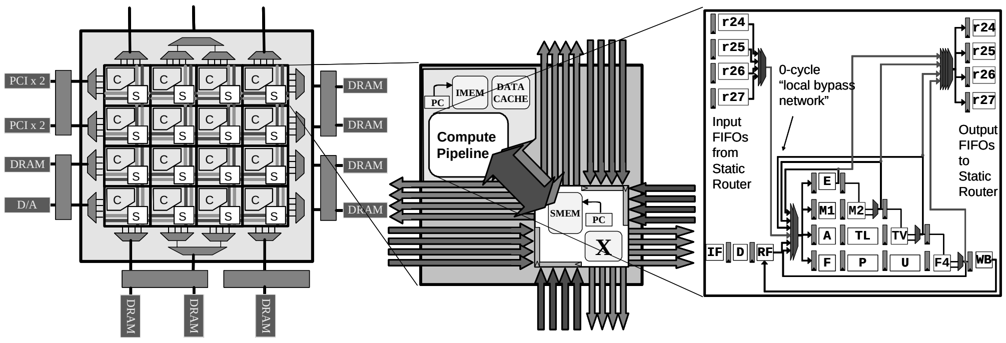


Figure 4.1: The RAW microprocessor [TLM+04].

RAW's development is supervised by Anant Agarwal at MIT [TLM+04]. A survey of the literature reveals that RAW appears to be the first cellular architecture to be implemented in hardware.

RAW is a homogenous cellular architecture with rather large cells. A single cell consists of a single issue MIPS core, 32 Kbytes data cache memory, a programmable static network processor and a dynamic network processor. Sixteen of these cells are arranged in a 4x4 grid to build up a single RAW chip. Figure 4.1 shows an overview of the RAW architecture and how the components are assembled into the RAW chip. A RAW chip has a large number of connector pins (over 1100). This massive number of pins allows the chip to support an I/O bandwidth that is 60 times larger than that of a Pentium III[2]. RAW was designed with so much bandwidth in order to link several RAW chips together (see section 4.1.3). Another option to use this abundance of bandwidth is for streaming applications and to turn RAW into a more DSP like processor.

The following paragraphs describe certain parts of the RAW architecture in detail.

4.1.1. RAW CPU

RAW uses a five stage pipelined, single issue MIPS core with 32Kbytes instruction cache, as it's main CPU. The CPU is a traditional RISC CPU apart from four registers: `r24` to `r27`. These registers connect the CPU to the static and dynamic network processor buffered by a 3-element queue. Any access to them will either read a word from the network processor or write a word to the network processor. Read accesses to the register while the queue is empty will stall the "read register stage" until a value becomes available. A similar effect happens if the "execution stage" tries to write back a value while the queue is full.

It seems that the data cache can be used in two different operations mode. The first mode uses the on-tile memory as a private, fast memory. It also seems that data has to be copied from off-chip memory to on-tile memory under the control of the CPU and the application.

The second mode was added later. It simply uses the on-tile memory as a normal L1 data cache. Currently not much information on this mode is available. A reasonable guess for the introduction of this mode is that, for some classes of applications, programmers prefer not to deal with private data memory and the associated complications.

[2]These are estimated numbers due to lack of detailed information to the public.

4.1.2. On-Chip Networks

As mentioned before, RAW has two different networks: a static network and a dynamic one. The static network is used for messages whose source and destination can be determined at compile time. Thus, the compiler can arrange a predetermined routing, resulting in a static routing behaviour at runtime and messages that do not need headers to inform the router of the destination and message length. The dynamic network on the other hand performs the routing decision at runtime. It is used for messages whose destination cannot be determined statically at compile time, but only at runtime.

Static Network

The static network processor is user programmable. It has four registers, a very limited instruction set, mostly consisting of conditional branches, and ten network ports; two to each neighbour and two connecting to the main CPU. Every instruction consists of two parts: A "normal" CPU instruction and a routing map for all network ports. Thus, the static network processor can route (within certain limits) up to ten words per cycle.

With the static network, it is possible to transmit a value to a direct neighbour tile in just three cycles. Because of these rather low latencies it becomes possible to exploit some ILP with RAW by combining several tiles into a mode that is somewhat similar to VLIW [LPSA02].

Dynamic Network

The dynamic network is designed as a backup network in case the destination of a network access cannot be determined statically. It is also used for memory transfers between the local data caches and off-chip memory. It guarantees that if two messages are sent from tile A to tile B and message *1* is sent before message *2*, then message *1* will arrive first. Apart from this, it does not give any further guarantees. For example, messages sent to different tiles might arrive in different order.

4.1.3. RAW cluster

Another feature that is worth mentioning is the possibility of combining several RAW chips into a RAW cluster. From the applications' point of view the whole system will just look like a, for example, 32x32 RAW chip. Only, the latencies for messages that cross chip boundaries will be higher. There is no information on whether this configuration has been tested in real hardware.

4.1.4. RAW Compiler: Maps

The main RAW philosophy is that all aspects of the underlying hardware should be exposed to the compiler. Only if the compiler is aware of all the delays that certain communication has, it will be able to schedule it in the most efficient way. The underlying idea is that the compiler, which has global knowledge both about the program and the latencies of the underlying hardware, can develop an optimal schedule to achieve peak performance. This approach will also greatly simplify the hardware, since no complicated structures are needed to discover and exploit parallelism at run time.

Maps [LBF+98, BLAA99, Bar00] is the first compiler for RAW. It translates C code into native RAW code and is built on top of SUIF [HAA+96]. The compiler uses a technique called space-time scheduling. It is applied to the instructions within a basic block, assigning them not only a time slot but also a tile. If values that are produced on a tile are needed on another tile, then the scheduler creates instructions that copy the value to that tile by using the network registers and programming the static network processor. The tradeoffs that the scheduler has to take into account are, on the one hand, the communication delays of transmitting a value to another tile, and, on the other hand, utilising the maximum number of tiles to execute independent instructions simultaneously. Space-time scheduling has later been improved by a technique called “Convergent Scheduling” [LPSA02]. Still, this technique does not overcome the restriction that it only applies to a single basic block.

Maps has been developed to support RAW’s private data memory model. It is not clear whether it also supports the cache model, but the lack of information suggests it does not. By using the private memory model, Maps can map data in such a way that it is evenly distributed across all private memories. This distribution works by using certain bits in the address as the tile address. However, there is no hardware based mapping, it is up to the software to identify the tile with these bits and to remove them from the address before performing the real memory access. This scheme allows distributing the data independently from the size of one data element.

In order to get the best on chip memory bandwidth, it is necessary that the data is distributed as much as possible across different tiles. Maps uses two techniques for this distribution: the first one uses the Span package [RR99] to identify pointers that might point to the same memory location. Memory objects that are guaranteed to be different are then allocated to different tiles. The second technique is called *modulo unrolling*, which is very similar to loop unrolling. However, it uses the unrolled loop not only to find independent instructions that can be assigned to different tiles, but also to distribute the matrices that the loop operates on across the local memories.

One of Maps probably biggest shortcomings is that it can perform space-time scheduling only within a single basic block. Hence the possibility of finding independent instructions is rather limited, unless the basic block has been enlarged by, for example, loop unrolling.

4.1.5. RAW Compiler: StreamIt

StreamIt [TKA02] is a high level programming language that has been designed with data streaming applications in mind. In these applications, data enters the chip on one side, is passed from tile to tile, until it leaves the chip on the other side. Based on the description of StreamIt, it is unlikely that it can be used for any other type of application. The RAW group has developed a backend for the StreamIt compiler that supports RAW [GTK+02]. Due to its high I/O bandwidth, RAW is particularly well suited for streaming data processing.

4.1.6. Tilera TILE64

Tilera’s TILE64 is the commercial successor of the RAW processor [ABB+07]. The processor is fairly identical to RAW, with the following changes (see figure 4.2): the simple single-issue, in-order CPU has been replaced with a 3 way VLIW one. The number of tiles has been increased to 64, arranged in an 8x8 layout. The local memory has been changed into a 2 level cache per

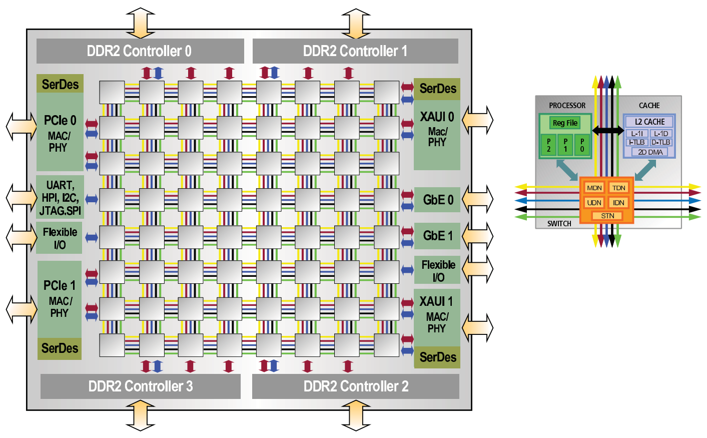


Figure 4.2: The Tilera TILE64 processor [Til].

tile. The number of different dynamic networks has been increased to 4. Other changes include integrated memory controller and different I/O ports. However, the main characteristics of the chip remain the same: TILE64 still uses mesh based on-chip network and seems to be very suited for streaming applications (such as multimedia encoding and decoding; network traffic filtering and routing).

Unfortunately, there is almost no solid information about the technical details available. Especially, no information on cache coherence is available. Tilera only states that each tile is able to run an independent copy of an operating system. Furthermore, several tiles together can run a multiprocessor operating system. Since Tilera has not published any shared memory application results and only focused on stream based application, it can be assumed that the processor has no cache coherence mechanism and most likely implements cache coherence in software (most likely directly in the OS).

4.2. Trips

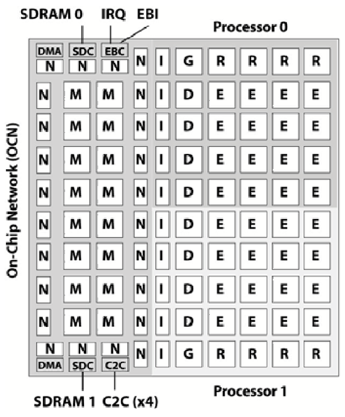


Figure 4.3: The Trips microprocessor [Bur05].

The Trips processor has been developed at the University of Texas, Austin, by a team led by Doug Burger [SNL+03]. A hardware implementation of the Trips processor was finished in the second half of 2006.

The Trips architecture is unlike any architecture of the past. Some parts of it have some similarities with data flow machines, some other aspects bear similarities with VLIW architectures, but most parts are simply novel.

A tile in Trips is not a complete CPU as it is in RAW. Instead, tiles are specialised to only perform a very specific task, like stages in a pipelined CPU. For example, there are execute tiles, instruction cache tiles, data cache tiles, global control tiles, and register file tiles, to just name some of them. Figure 4.3 shows the arrangement of these tiles. The CPU is divided into three areas: on the left of the chip is the data cache. The right side is occupied by two execution cores: one in the upper half and one in the bottom half. An execution core consists of 16 execution tiles, 4 register file tiles, 4 instruction cache tiles, 4 load store queue tiles and 1 global control tile. The other main difference between RAW and Trips is the philosophy followed by these architectures. RAW emphasises a static approach to instruction scheduling and network packet routing, with a single issue, in-order CPU, the static network and a compiler that has to schedule instructions on the CPU and network processor statically. Trips, on the other hand, uses a much more dynamic approach to instruction scheduling and does not have a static network. Instead, all traffic between tiles is routed with one dynamic network out of seven dynamic networks. In addition, instructions that can be executed are (within certain limits) determined dynamically.

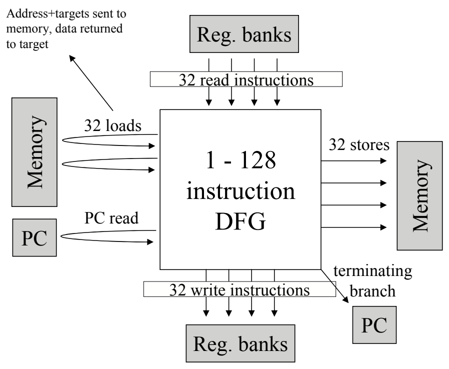


Figure 4.4: A Trips Execution Block [Bur05].

Trips does not execute instructions in the conventional sense; instead a program is divided into blocks, which are similar to HyperBlocks[3] [MLC+92]. A block consists of up to 128 instructions[4] and several support instructions that access the register file or perform memory accesses. Figure 4.4 shows such an execution block. The register file is divided into 4 banks with 32 registers each. A block can read and write up to eight registers in each bank. As for the memory accesses, it is possible to perform up to 32 memory accesses within one block. However, the sum of all stores and loads cannot exceed 32. The most important aspect about a block however is its atomic behaviour: it will only commit completely or not at all. The instructions are statically assigned to a slot in a specific execution tile. If an instruction produces a result, then it also encodes which instructions need the result it produces and sends the result to them. The result is, in contrast to RAW, dynamically routed to the receiving tile. The execution of

[3]A HyperBlock is a collection of basic blocks with a single entry of control. However, unlike a basic block, control can also leave a HyperBlock prematurely without every instruction being executed. A HyperBlock can have an arbitrary number of these premature exists. A final restriction is that the collection of basic blocks within the HyperBlock cannot form a loop.

[4]A block is always filled up to 128 instructions with *noops*.

a block starts with the register file tiles sending values to the appropriate execution tiles. The execution ends once a determined number of results have been produced and are either sent to the register file or data cache. Under optimal circumstances, Trips is able to execute 16 instructions per core at the same time. However, since the results have to be sent across the operand network, they will be delayed by one cycle for every tile they cross. Thus, other independent instructions are needed in order to maintain this performance. Only if all values have been produced, a commit happens.

Since a block could contain several branch instructions (of which only one is really executed), it can have several successors. Just like a normal CPU uses a branch predictor, a block predictor is used in Trips. The quality of this predictor is very important in order to achieve good performance on Trips. Since loading a block into the execution units takes several cycles, it is very important that the next blocks are already being loaded while the current block still executes. In addition, the execution of the next block will be started speculatively based on the results that are already available. This is one source for the 16 instructions per cycle that are needed in order to achieve peak performance.

The next block predictor is part of Trips' global control tile. This approach is different from the RAW approach where every tile has exactly the same capabilities and without a designated controller. The control tile manages the loading of the instruction blocks. It also determines if a block has been completed in order to purge it from the execution tiles. While in the current design with 16 execution tiles the control tile can cope with the delays caused by tile distance, it might become a bottleneck or cause some other problems if one tries to increase the number of execution tiles per core.

As mentioned before every Trips chip contains two cores. These cores share a 1 MB L2 data cache. The cache again is built from tiles. These tiles provide a non-uniform response time to cache accesses [KBK02]. Hence, if data is stored on a tile that is close to an execution core, then the data can be accessed faster than if it is further away. On average this gives better cache hit latencies than a uniform cache, where the hit time is defined as the worst time that it takes until a request is resolved. It also allows the construction of bigger caches that are not limited by wire delays. Another possibility is to migrate data in the cache, such that more frequently used data is closer to the core using it. Trips supports four different modes of operation for this cache memory:

- Normal shared cache for both execution cores.
- Split the cache between each execution core. This strategy minimises the maximum hit latency each core can experience. It also prevents one core from evicting the other core's data from the cache (due to collisions or capacity problems). To my knowledge, Trips does not implement any cache coherence protocols for this mode.
- Use the L2 data cache as scratch pad memory. The memory is mapped to the process as some very fast local memory, however it's up to the application to move data between the off chip main memory and the scratch pad.
- Use one half of the L2 memory as a scratch pad and the other half as a shared L2 cache for both cores.

Trips looks primarily at ILP as its main source for parallelism. While there is little doubt that ILP is available to a certain degree, previous studies [Wal91] have shown that ILP only exists in abundance when completely unrealistic assumptions are made (like perfect branch prediction). Otherwise, the best one can hope to find on average is ILP up to 14 [Wal91]. While Trips team hopes that it will be able to achieve similar performance as current super scalar design with less complicated hardware (that probably is also more suitable for higher frequencies), this thesis advocates that it is beneficial to also look at other forms of parallelism besides ILP. The inability of the design to run “unmodified” shared memory applications might be an inconvenience or a limitation to what kind of parallelism it can exploit.

4.3. Cyclops

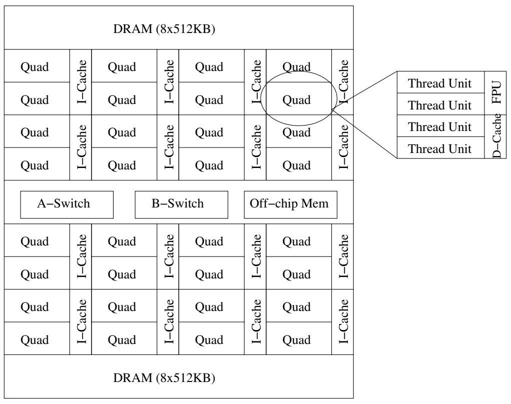


Figure 4.5: Cyclops processor: block diagram [CCC+02].

Cyclops [ACC+03, CCC+02] is part of the Blue Gene Project by IBM. Unlike the commercially available BlueGene/L, Cyclops also defines a new type of CPU. The idea is that an application running on Cyclops consists of hundreds or even thousands of threads.

The basic design of a Cyclops CPU is shown in figure 4.5. It consists of several units called Thread Groups, some I-caches, 8 MB embedded DRAM and some off-chip communication logic. Four Thread Groups share one 32Kbyte I-cache module. A Thread Group is composed of four Thread Units, a data cache and a floating-point unit. While the floating-point unit is shared between the four thread units, the 16 Kbytes data cache is shared with all other Thread Units on the chip. In total, there are 128 Thread Units on the chip and the idea is that each of them executes a single thread. However, compared with RAW it is not possible to combine some of these Thread Units to extract some ILP; thus, if there are not enough threads available, then only a fraction of the chip’s performance can be exploited.

The most interesting part about Cyclops is the memory model. As shown in figure 4.5, Cyclops’ cache is distributed across the whole chip. Figure 4.6 shows the two switches that enable the memory system to load data into any cache and enable every thread unit to access any cache. Cyclops does not provide any hardware cache coherence; it is up to the software to deal with this problem. The data placement is also under the control of the software. Cyclops uses the top eight bits in each physical address to control the placement of data (this also limits the amount of directly addressable memory to 16MB). This placement does not necessarily assign data to exactly one thread group. Several thread groups can form an “interest group”; this group has an interest in the same data. In this case, the data is automatically distributed among the caches that belong to this interest group and a hardware mechanism guarantees that

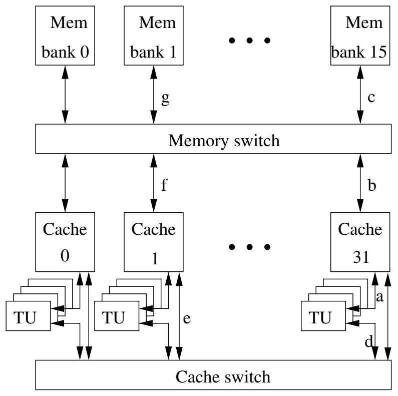


Figure 4.6: Cyclops processor: memory system [CCC+02].

the data is only stored in one cache. Another interest group consists of the whole Cyclops chip, resulting in data being stored in just one location. While in this mode the programmer does not have to deal with the coherence problem, he has sacrificed locality and hence has to deal with longer latencies. However, by using the upper eight bits for cache placement, Cyclops does not really need a cache coherence mechanism as long as the software does not refer to the same physical address via different mapping addresses. For example `0x00ABCFEF` and `0x02ABCDEF` would both refer to the physical address `0xABCDEF`, but it would be accessed via different caches.

Also worth mentioning is that Cyclops uses a special register to enable a fast barrier operation across a subset of the thread units. This barrier register is a special purpose register that can be written individually by each thread; however, a read operation returns the ORed value of all registers. Each thread unit has first to decide, if it wants to participate in the barrier. If it does, it will write a 1 to its register otherwise a 0. Once a participating thread enters the barrier, it will also write a 0 to the register. All threads will have entered the barrier when reading the register returns a 0. Of course, there are some finer details to enable multiple barriers at the same time. Cyclops supports up to four barriers with this register.

4.4. WaveScalar

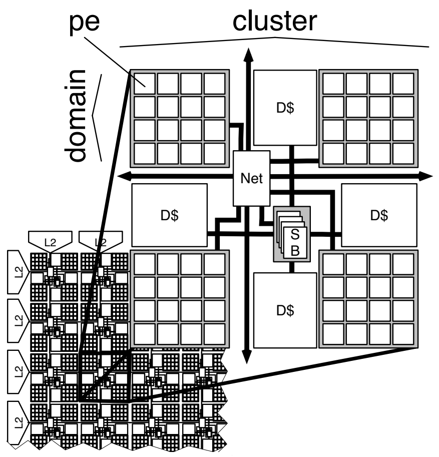


Figure 4.7: Overview of a WaveScalar chip [SSP+04].

The WaveScalar project is supervised at the University of Washington by Mark Oskin et al. [SMSO03]. It is currently still in its early stages of development. The idea is to have thousands of small and mostly identical (how they differ is so far not specified) processing elements. These processing elements execute the program in a way that is very similar to data flow machines. However, unlike Trips, which encodes the dependencies between instructions in the instruction itself, WaveScalar uses a traditional hardware dynamic token match mechanism. Unlike previous data flow architecture, WaveScalar is able to maintain the load-store ordering of imperative programming languages[5] and does not have a centralised control unit. WaveScalar distributes the processing elements across the instruction cache. This combination of logic and memory is called a *Wavecache*. Figure 4.7 shows an overview of the WaveScalar chip. The most important aspect is to move instructions that are linked via data-flow dependencies physically close together to minimise the communication delay.

WaveScalar achieves the load-store-ordering of imperative programming languages by breaking the execution in blocks they call a *Wave*. Only one Wave can be executed at a time. A Wave seems to be similar to a HyperBlock [MLC+92]. Within a Wave, every memory instruction is annotated with its ordering relationship in the original control flow graph. These annotations allow the memory system to execute these operations in the correct order. The current design insists on a partial order of all memory operations. It is not possible to declare two operations as independent. The memory system consists of the usual parts for a two level cache hierarchy. As for the second level cache, it is “conventional, unified, non-intelligent” cache. It is not clear, if this cache is on-chip or not, nor if data is placed in such a way to minimise communication time. As for the later, this work assumes that “non-intelligent” implies that it does not. The first level data cache seems to be distributed across the whole chip. Four clusters of processing elements share a data cache module.

The original architecture was designed to just execute a single thread application. However, the WaveScalar team has identified this problem and modified their design slightly to deal with this issue [SSP+04]. However, most of this work addresses the implementation of synchronisation primitives for a WaveScalar architecture. It does not address the problem of cache coherence of the data caches.

The WaveScalar team developed a hardware prototype of their architecture implemented with FPGAs in August 2005.

4.5. Vector Threads

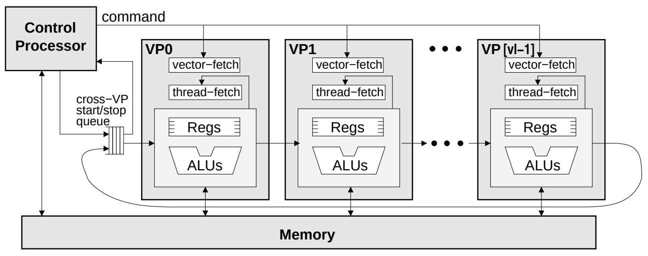


Figure 4.8: Logical View of the Vector Threads Architecture [KBH+04].

[5]This makes programming easier, since the programmer will get the load-store order he/she expects.

The Vector Thread architecture [KBH+04] looks very similar to the Multiscalar architecture proposed by Sohi et al. [SBV95]. However, unlike Sohi's design it does not use speculation (which simplifies the hardware) in order to find additional parallelism. The design idea is to have an architecture that is equally well suited for multi threaded code as for vector/SIMD code. A Vector Thread CPU consists of a main processing element (MPE) and several *vector thread units* (VTU). Work is assigned to the VTUs in form of an *atomic instruction block* (AIB). However, unlike the blocks in Trips these blocks do not have an atomic commit. Instructions within a block commit in the usual way. The VTUs do not have an automatic fetch mechanism for these blocks; the AIBs have to be either assigned to them by the MPE or they have to request them by themselves.

SIMD code is executed by assigning the same code to each of the VTUs and simply changing the offset at which each VTU starts processing the data. Since SIMD code does not contain any branches, all VTUs will execute exactly the same code. For threaded code, different (or the same) AIB will be assigned to each VTU. However, then it is up to each VTU to fetch the next AIB.

Another feature, which has been copied from Multiscalar, is that the VTUs are linked in a one directional ring. This enables them to forward results to the next VTU. One potential application is to execute iterations of a loop partially in parallel that otherwise would not be efficiently parallelisable at all.

As for its memory model, it assumes a standard L1 data cache that is connected via a crossbar to the processing elements. As discussed in section 3.3.2, such a crossbar will not allow the design to scale to a larger number of VTUs. From the published material, it seems that most of the design focused on the VTU and not on the memory system and its scalability.

4.6. Intel's Tera-scale Computing Prototype Polaris

Intel recently presented a prototype of a tiled CMP with 80 cores [VHR+07]. The tiles are arranged in an 8x10 mesh and connected using a packet based point-to-point interconnect. Each tile has 2KB of data and 3KB of instruction memory. There is no indication of whether this memory is used as local stores or as a cache. Nor is any information available, if this processor supports any kind of cache coherence. However, considering the simplicity of each tile and the overall floorplan of the chip, it seems that the memory is local storage not being kept coherent by any hardware mechanism. While this prototype does not support a shared memory programming model, it does give an indication of up to how many tiles future tiled architectures will scale.

4.7. Conclusion

This chapter presented an overview of proposed tiled architectures. These architectures are particular interesting, since they already feature number of cores that have to be expected in future generation of large scale CMPs. One particular area of interest is the interconnect that is used to allow communication between the tiles. A common trend among most tiled architectures (such as RAW, TILE64, Trips, WaveScalar and Polaris) is the use of a mesh based interconnect, combined with packet switching routers. As such it seems reasonable to assume that similar interconnects will be used in future CMPs. In particular, designs such as

RAW, TILE64 and Polaris (which have been implemented in hardware) can be easily envisioned with more complex core, once the total number of transistors supports such designs. As a final note, while this thesis will assume a base line architecture that is very similar to one of the ones mentioned here, it does in no way limit the scheme, which this thesis will proposed, to tiled architectures.

5. Software Distributed Shared Memory

As discussed in section 2.1, Message Passing systems offer the advantage of combining several processing nodes (such as workstations) into a larger system. On the other hand, they have the disadvantage of a more complicated programming model that requires the programmer to deal with data placement.

Software Distributed Shared Memory systems (SW DSM) try to bridge this gap, by combining the advantages of a shared memory system with the scalability and flexibility of loosely coupled systems. By using a software layer, it presents the physically different address spaces as a single shared one. This process is shown in figure 5.1: while each processing element can directly access only its local memory, the Mapping Manager makes it appear that all PEs are using the same address space. Of course, it then has to deal with the cache and memory coherence problem as hardware shared memory systems do. The following sections present several approaches that implement such a system. Please note that providing an in depth description of all available SW DSM systems is beyond the scope of this chapter. Instead, the discussion here focuses on a few significant ones that introduced new ideas to the SW DSM approach. Section 5.1 presents the IVY system [Li88], which is the first SW DSM system. This initial system has prompted additional research into various optimisations. One such area of research is to aid SW DSM system with additional information that is available program language level, but is usually lost at binary level. Such language based system are presented in section 5.2. In particular the SW DSM systems Munin [BCZ90], CRL [JKW95] and Orca [BKT92] are discussed. Munin, as a survey of the literature reveals, was the first system to utilise such additional information in order to characterise the sharing pattern. CRL is noteworthy because it is implemented as a system and compiler independent library. It only requires that before and after shared data is used special library calls are inserted. Orca, on the other hand, was the first system that designed a whole, new programming language to support distributed shared memory programming. The TreadMarks system [KCDZ94], presented in section 5.3, tries to maintain the language agnostics of the IVY system while at the same time trying to optimise the protocol with some insights learnt from language based system. It is also the only commer-

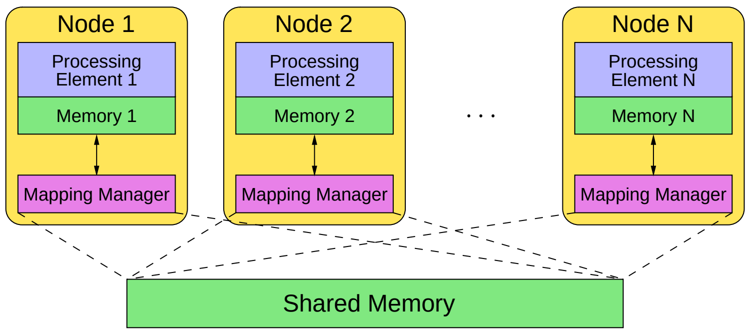


Figure 5.1: Distributed Shared Memory Mapping.

cially available SW DSM system to date. Section 5.4 presents a brief summary of SW DSM system that have been proposed more recently.

The final section in this chapter then relates these works to this thesis and justifies why the TreadMarks systems is a reasonable baseline to compare against.

5.1. IVY

IVY [Li88] is the first system that tried to implement a Software Distributed Shared Memory system on top of a cluster of workstations. It uses the already existing MMU (memory management unit, which normally provides the translation of virtual addresses to physical addresses) in the workstations to provide a distributed shared memory. The scheme roughly works as follows: the multi threaded shared memory application is mapped to the same virtual address space on all processing nodes. A page that is accessed, but does not reside within the physical memory of a node, will cause a page fault. The fault handler will then copy the page over the network from a node that has a copy. In order to ensure data coherence across all nodes, IVY allows a single writer. To ensure this, IVY marks all pages as write-protected within the MMU. If a node tries to write to a page, it will again trigger the page fault handler. IVY will then invalidate all other copies of that page and give write permission to the node performing the write access. By restricting access to a single writer, IVY is able to implement a sequential consistency model for the distributed shared memory.

IVY's implementation consists of a user level part and an operating system part[1], with most services being implemented in the user level part. It can use different algorithms to manage page ownership information: the *centralised manager* runs on a single node and other nodes have to contact this node in case of a page fault. Since this single node might become a bottleneck, the *fixed distributed manager* assigns responsibility for a certain page to different nodes based on some bit in the page start address. If using this manager still results in one node becoming a bottleneck for requests, then IVY can use a *dynamic distributed manager*. A node using this manager will keep track of the owners of all pages itself. This owner might be the true owner, or just the "probable" owner, in case the page already migrated to another node. While this scheme does not suffer from the bottleneck problem, it takes longer to locate the true owner of a page. Another notable feature of IVY is that it supports automatic process migration for load balancing reasons. Also, since the local OS does not know that it is part of a distributed shared memory system, it has no means to create a thread remotely. Instead the thread is created locally and then migrated by the load balancer to an idle node.

Being the first distributed shared memory system; IVY had several shortcomings that limited performance greatly. The main problem is that IVY only allows a single writer to a page and has to migrate pages to nodes that want to write. False sharing of a page causes another problem. False sharing forces a page to be invalidated on another node, even though the nodes accesses different areas on that page. The only solution IVY offers for this problem is to use relatively small pages of size 1KB or 256 bytes.

The following two sections will discuss two different approaches to overcome this issue. The first approach focuses on making additional information available to the SW DSM, by an-

[1] It seems that the operating system part was only necessary due to limited user access to the memory management.

notating the programs with sharing information. The second approach retains the language independent page-based approach, but augments it to address the above problem.

5.2. Language based SW DSMs

As discussed in the previous section the main problem of IVY was that pages would migrate frequently if several nodes would try to write to them. Also, a node that writes would invalidate copies on other nodes, even if these other nodes were reading from a different area on that page. The main reason for this behaviour is false sharing: the SW DSM system does not know that the node that writes will write to a different region on the page than the node that is reading from the page. Similarly, two nodes that write to the same page might write to different addresses that do not interfere with each other. Another reason would be the implementation of user-level synchronisation primitives that rely on sequential consistency.

In order to allow a SW DSM to deal with these different cases efficiently, a SW DSM needs to know the following information: what are the boundaries of a certain object in memory? Using pages as boundaries is a rather arbitrary, practical way that quite likely does not reflect the real object boundaries. The other kind of information that would be beneficial is to know what kind of object this region of memory is. For example, is this object only been read from? Or, does it have multiple writers or just a single one? Using this information the SW DSM can make more sophisticated decisions on how to distribute and keep this region consistent.

Two SW DSM systems that try to address these issues are Munin [BCZ90] and Orca [BKT92]. Both systems identified that there are several different categories of objects that can be found in shared memory program:

Write-Once Objects are written during initialisation but afterwards only read. These objects can be safely replicated across several nodes.

Private Objects are only accessed by the local node. Since changes to these objects are of no concern to other nodes, the SW DSM system does not have to be invoked when accessing these objects.

Result Objects are maybe written by several nodes and will not be read from until a global synchronisation event has occurred. Since the SW DSM system knows that different nodes update different parts of the result object, the writes can be delayed and then later combined at the global synchronisation.

Synchronisation Objects are used to allow a node exclusive access to certain other objects.

Read-Mostly Objects are updated very infrequently. Thus, the SW DSM system can replicate these objects and broadcast changes to all nodes that have a copy.

Migratory Objects should be moved to the node that is currently accessing them. There is usually no need to replicate this object across several nodes.

General Read/Write Objects are objects that cannot be classed as any of the above mentioned objects. The SW DSM system will most likely resort to a single writer policy (maybe at a finer granularity than the size of the object) in order to avoid an incoherent state.

5.2.1. Munin

Several other SW DSMs try to address the problem of pages thrashing by pinning a page to a particular node (and forcing the other node to access it remotely) [RAK89, FP89]. However, a survey of the literature reveals that Munin [BCZ90] appears to be the first system that took advantage of the above mentioned classes of objects by providing different consistency protocols to deal with them more efficiently. It also relaxed the consistency for most parts to release consistency.

Munin is an extension to the Presto parallel programming environment [BLL88]. Presto provides parallelism and synchronisation for C++. As mentioned before, users are expected to insert declarations to provide type-specific information for each object. The compiler then passes this information on to the runtime system, which in turn selects the appropriate consistency protocol.

5.2.2. C Region Library

The C Region Library (CRL) [JKW95] is language and system independent library that provides support for distributed shared memory. The basic idea is that the programmer has to declare all shared data using CRL library function. Before such data can be used, it has to be mapped into the address space of the node that wants to use it. Furthermore, CRL has to be informed of all read and write accesses to the data using special library functions such as `CRL_start_read` and `CRL_end_read`. The user is again responsible for inserting these function calls. Unlike all other so far discussed schemes, it does not need any special compiler or hardware support (like modifying page access permissions) apart from being able to send messages across a network.

5.2.3. Orca

Instead of simply extending an existing programming language with annotations, the approach of Orca [BKT92] goes further. Orca provides a complete system, consisting of a programming language, compiler and runtime system. While Orca can be used from within traditional languages (like ANSI C), it is optimised for programs that are written in the Orca language. The Orca language was designed especially for programming on distributed-memory systems. While this allows one to write new applications in a clean way, it causes problems to reuse existing codes.

5.3. TreadMarks

TreadMarks is an advanced page based SW DSM system [KCDZ94]. Similar to IVY and other page based SW DSM, TreadMarks does not require any changes to the program. Any shared memory program, written for the release consistency model, will run on TreadMarks without modification. TreadMarks is build around the principle of “Lazy Release Consistency” [KCZ92] in order to reduce the additional overhead to keep the system consistent. “Lazy Release Consistency” is similar to release consistency in that it requires special acquire and release events before changes to shared data can be made visible to other nodes. However, instead of making these changes immediately available by sending them to all other nodes that have a copy of that page, the idea is to delay this action until the other node tries to access a page that has been modified. The rationale is that the other node might not access the modified data, and as such, it would be a waste to compute and send the update information. Furthermore as per the RC model, TreadMarks assumes that if different nodes write to the same page in between synchronisation points then these nodes write to different addresses and can be merged in the future.

In order to enable merging of changes, TreadMarks uses a diff based approach: initially all pages are marked as write protected in the TLB. Upon a write access a copy of the page (a so called "twin") and a write notice to remember that this page has been modified are created. The write notice will later also store the diff that encodes the changes such that the diff has only to be created once. The process of creating these diffs has been generally noticed as the most expensive part of such system [ISL96].

As noted before, TreadMarks tries to reduce the number of diff creations by only creating a diff when it is required to ensure correct program execution. In order to be able to associate a write operation with a certain point of the execution time of the application, TreadMarks divides the execution on each node into intervals. An interval begins with an acquire or release operation and ends with an acquire or release operation. Whenever a write notice is created, it is linked to the current interval. In addition to its own interval, the nodes also keep track of the last known interval other nodes are in. This information is held in a data structure called a "vector timestamp". Whenever an acquire event occurs the previous owner of the lock transfers its vector timestamp to the new owner. By comparing its vector timestamp with the new one, the acquirer is able to identify intervals that it does not know about so far. This might include only new intervals on the previous owner, but could also include intervals on other nodes. This ensures that the acquiring node is not only informed about changes that the previous owner made, but also about all changes of previous owners. The acquirer then requests the write notices for each interval it does not know about. For each write notice that is received, the associated page is marked as invalid and linked with the write notice. This allows a quick traversal of write notices in case of an access to the page. Apart from transferring and managing the write notices nothing else is done at this stage. If at a later stage during the execution of the program a memory access to such a page is performed, then a trap to the TreadMarks runtime system will occur. The runtime system will analyse from which nodes it can get the required diffs to validate the local version of that page. Getting the diffs usually means that program execution on these other nodes has to be suspended until the diff has been created.

5.4. Additional SW DSM systems

This section tries to give a brief overview of other SW DSM systems that have been proposed. Quite often similar systems with similar characteristics have been proposed; in such a case only one system is introduced. This selection is guided by which system is best documented and referred to in other publications more often.

The Shasta [SGT96] SW-DSM system used a different approach from all so far mentioned SW-DSM systems. It used binary rewriting technique to convert a parallel shared memory program into a distributed shared memory program. This was done by replacing all load and store instruction with custom code that performs checks to see if the data is shared, exclusively held by another processor, etc. The advantages of this approach are that sharing can be detected at a much finer granularity (multiple of cache line sizes) and that traps to the OS due to page table protection faults are avoided. The disadvantages are that the code size is increased by about 70% and that the instrumentation is always executed as opposed to page based schemes, where only the first access triggers an exception. Shasta employed a couple of techniques to minimise this overhead (e.g. load and stores that use the stack pointer as the base register will access private data, thus no checks need to be inserted).

The schemes discussed up to this point are true software distributed shared memory system. Based on the systems discussed so far, researchers tried to identify inexpensive hardware additions that improve performance, while still the largest amount of work is still performed in software. The Tempest [RLW94] system builds on top of the proposed hardware platform Typhoon. It supports a wide range of data communication: from simple low-overhead message passing to sophisticated, fine-grained memory consistency. In order to support the latter efficiently, it needs hardware support to assign access permissions at a finer granularity than pages. In the proposed implementation, aligned regions with a length of a power of 2 are supported. The actual coherence handler can then either implemented in software or in hardware. The Cashmere [KHS+97] protocol requires the availability of a low-latency mechanism that performs writes directly to main memory of another node. It is in principle another page based SW DSM system, however writes to shared data are doubled: one write updates the local copy, the other write updates the main memory on the home node of that page. The system was later soon after extended into a 2-level system [SDH+97]. Within a node of SMP processors coherence is maintained using the provided hardware mechanism. Coherence across these nodes is maintained by a modified Cashmere protocol.

Currently the main area of interests in SW DSM research are multi-level protocol and heterogeneous systems. One such system is the InterWeave system [CTC+02], which is a 3 level protocol: within an SMP node coherence is enforced with a hardware protocol. Across several such nodes that are linked within a tightly coupled cluster a software protocol similar to Cashmere is used. At the 3rd level the system provides a mechanism to keep objects that are distributed across different servers on the Internet coherent.

5.5. Conclusion

While language based SW DSM systems demonstrate good scalability for some programs, they do distance themselves from the original shared memory goal of providing a simpler programming environment than message passing systems. While the user is not required to modify his/her algorithm to match the message passing paradigm, he/she is required to annotate the used data structures according to their usage. A misclassification will most likely result in a runtime error (for illegal accesses that the SW DSM can detect) or in an incorrect result. A simple classification as "General Read/Write Objects" on the other hand will result in poor performance.

Page based systems on the other hand do not require a change of the application (as long as the consistency model required by the application matches the one provided by the page based system). However, overall these systems were not able to deliver the same performance as their language based counterparts. In some cases [RLW94], it is possible to tune the coherence mechanism of the page based system to the problem, in order to minimise unnecessary communication. But this sacrifices the transparency that the system should provide.

SW DSM system that require the presence of special, simple hardware already left the original idea of SW DSMs to provide coherent shared memory without additional hardware support. However, they do add interesting design points to the general problem of providing shared memory. Furthermore, they highlight the importance to investigate design points that lie between software-only and hardware-only designs.

5.6. Relation to Thesis

SW DSM systems were developed in the context of a loosely coupled workstation environment that does not offer any kind of hardware support for a shared address space. As such they would also lend themselves to the problem of offering cache coherence to a CMP that does not have hardware support for it. However, these systems were developed with a different target and tradeoffs in mind. Inter-node communication latencies were one of the most pressing concerns in these systems, such that performing more computation and utilising sophisticated memory structures to lower communication requirements was the preferred tradeoff. On a CMP the communication latencies are much lower in relation to computation speed than in a multi node system. In addition, the migration and replication of data occurs at main memory for SW DSM systems instead of only at the caches. Furthermore, the limited memory space available on chip limits the usage of sophisticated data structures. Still investigating the suitability of a software only solution is necessary in order to be able to make the case in this thesis for some limited hardware support.

This thesis uses the TreadMarks SW DSM system as a baseline to evaluate the feasibity of software only cache coherent CMP design. TreadMarks was chosen, since it is the only commercially available SW DSM, which indicates that the overall system is very mature. Furthermore, apart from relying on a page protection mechanism in the TLB, it does not have any additional requirements towards the hardware. This simplifies porting it to a different architecture. And as a final remark, Kontothanassis et al. compare the performance of TreadMarks and Cashmere [KHS+97]. They find that even though Cashmere utilises special hardware support, it is only able to outperform TreadMarks for application that exhibit a large amount of false sharing.

The only other true SW DSM for comparison would be Shasta. However, its implementation benefits from the availability of the binary instrumentation tool ATOM [SE94]. Unfortunately, such a tool is not available for the PowerPC ISA, rendering the possibility of reimplementing Shasta beyond the scope of this thesis.

Part II.

Proposed Architecture

6. Baseline Architecture

This thesis is concerned about tiled CMPs consisting of 32 or more processors. At the beginning of this research no such architecture was available. Considering the results of Kumar et al. [KZT05] (discussed in section 3.3) it seems unlikely that these systems will be built using shared buses or crossbars. As such, the usage of snooping based cache coherence approaches is similar unlikely. Since tiled architectures already scaled beyond processor numbers that are possible using buses and crossbars, they presented themselves as a good starting point to look for a baseline architecture. At the beginning of this research, only two tiled architectures seemed mature enough to be considered: RAW and Trips. Trips uses a very different execution model that requires a specialised compiler. As such, it does not really present itself as a typical CMP architecture. RAW on the other hand uses almost standard cores that are connected using a scalable point-to-point mesh interconnect. Such an interconnect would allow the architecture to scale to a larger number of nodes than the current 16 tiles[1]. As extra benefits, RAW already demonstrated strong performance for ILP and DLP and was implemented in real hardware, which emphasises the maturity of the design. In order to base the proposed architecture on a sound foundation, it was decided to use many of the parameters that were used in the RAW processor.

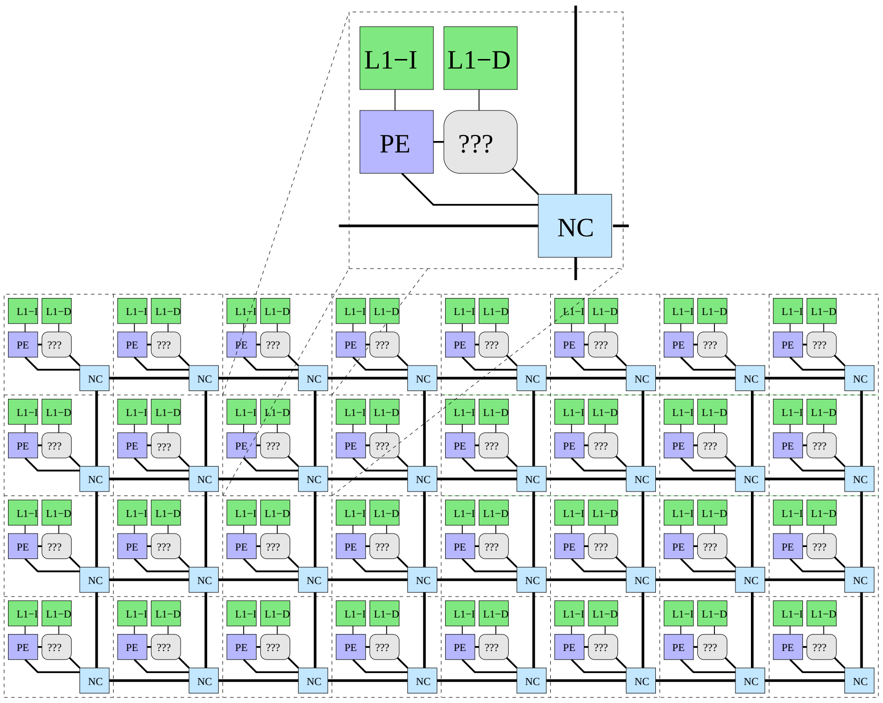


Figure 6.1: Overview of the baseline architecture.

[1]This has been confirmed by its successor TILE64 [ABB+07].

As such, this work assumes a fairly generic tile that consists of a compute processor (PE) that is a simple single-issue RISC processor with separate and private instruction and data caches. These first level caches are virtually indexed and physically tagged[2]. The compute processors are completely independent and there is no global program control in hardware. The on-chip interconnect fabric consists of a point-to-point network with a mesh topology where each tile is connected to its four neighbours. Each tile also contains a very simple network controller (NC) that performs simple dimension-ordered routing. The number of message buffers in the NC is enough to guarantee maximum throughput, which corresponds to four non-conflicting transfers per cycle. Figure 6.1 gives a high-level overview of the architecture.

The shaded gray components labelled with "???" in the figure refer to the additional hardware that is required to support cache coherence. A possible implementation might be a distributed cache coherence controller or the proposed extension that is discussed in chapter 7.

Since this work started from RAW, it left RAW specific architecture extensions in the baseline architecture. These extensions include the static programmable network controller (see section 4.1.2) and registers that are directly linked to the network (see section 4.1.1). These features are not necessary for the proposed architecture. However, in order to be more in line with the polymorphic principle of tiled architectures, which states that tiles can be linked together in different ways to adapt to different workloads, it would be useful to keep them. RAW already showed that it is able to extract some ILP by combining multiple tiles, and performs excellently on data parallel and streaming applications. Thus by keeping these components, these strengths are preserved and a new application domain of shared memory thread level parallelism is added.

[2]Using the virtual address to index the cache is a commonly used technique that allows the cache access to start before the TLB access has been completed. The cache block has to be physically tagged in order to verify that the correct block has been accessed.

7. A Scheme to Avoid Cache-Incoherence

This chapter describes the mechanism this work proposes to avoid incoherence. The basic idea is to treat all L1s as a single logical cache and, thus, avoid replication of data, which can lead to data incoherence. Section 7.1 presents a conceptional overview of this architecture, which is then explained in more detail in section 7.2. This initial architecture is extended later in sections 7.3 and 7.4 to allow some *controlled* migration and replication of data.

7.1. Overview

The basic idea of the scheme is to the fast on-chip network to perform cache accesses on the cache of a remote tile. Thus instead of having several copies of the same data, only one copy exists, which by default is consistent. A tile that accesses data determines the location of data by querying a local table in each tile. This table is similar to a TLB, but instead of translating a virtual address into a physical one, it translates it into a tile number. If the tile number is not the requesting tile, then the memory access is forwarded to the remote tile. Otherwise, it is performed locally.

This scheme is significantly simpler than a directory based scheme. Firstly, the address to tile translation happens at a local hardware structure. Secondly and more importantly, this is all that has to be done. No request has to be forwarded or results into multiple invalidation as in directory based scheme. This reduces complexity and avoids subtle race conditions.

7.2. Detailed Description

As mentioned in the introduction to this chapter, instead of trying to keep the L1 caches coherent, the proposed scheme avoids duplicate copies of a single cache line. To achieve this, every memory line can only reside in one L1 cache (the home cache or tile) and processors in other tiles must perform remote cache reads and writes to access the data. Thus instead of a directory controller, a remote cache access controller (RAC) is added to each tile, in the gray component labelled “???” in figure 6.1. To receive and service remote data requests the RAC is given access to the network and it uses the dedicated port to the cache that is normally used by the snooping or directory controller.

The simplest way to place and locate data in the L1 caches while enforcing a single copy of each line would be to statically map lines to L1 caches based on address. This, however, is too restrictive and takes no account of the data access patterns. At the other end of the spectrum, each line would be dynamically mapped to any one L1 cache and it would be located through broadcasts, centralised tag stores, or redundant tag stores, as has been previously proposed [BW04, CPV05, KBK02, ZA05]. What this thesis proposes is to map whole memory pages to L1 caches through extensions to the OS page table and the hardware TLB mechanisms.

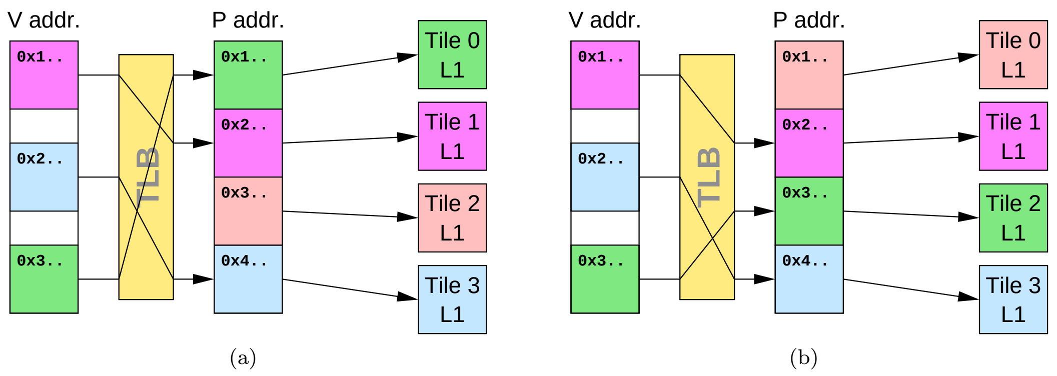


Figure 7.1: Direct mapping between virtual address and tile based on the physical address it is mapped to. For example, Tile 0 is responsible for all addresses that start with `0x1...` .

More specifically, the internal chip structure is exposed to the OS and the traditional page table is extended with a new table that maps virtual pages to architectural tiles. This is matched with a new TLB-like hardware table that caches these translations and allows for fast identification of the home L1 cache where data in the page can be found. Each tile is given one of such hardware structures, which is called MAP. The default policy for the OS to map virtual pages to tiles is first-touch. Note that the proposed mechanism is different from simply mapping memory pages to L1 caches based on physical address and using the virtual-to-physical page translation mechanism to provide the run-time mapping (see figure 7.1). The problem with the latter is that physical addresses are bound to specific L1 caches, which limits the OS flexibility in allocating physical memory and may lead to fragmentation and inefficient use of physical memory. For example, in figure 7.1a the physical address `0x3...` is not used and results in a fragmentation of the address space. Additionally, it makes any changes to the mappings much more involved, as the physical pages have to be moved in memory. For example, in figure 7.1b one might want to change the mapping for virtual address `0x3...` from tile 0 (as in figure 7.1a) to tile 2. Since tile 2 is only responsible for physical addresses starting with `0x3...`, the page content from physical address `0x1...` has to be copied to `0x3...` before the mapping can be changed in the TLB.

One important design decision at this point is where to provide virtual-to-physical address translation. Traditional CMPs keep all the translations of the local processor in the local TLB and ship only physical addresses to access lower level caches. A problem with using physical addresses for the remote cache accesses in the proposed architecture appears when virtually indexed L1 caches are used, which is often the case in order to speed up accesses from the local processor. Thus, performing the virtual-to-physical address translations locally in the case of remote L1 accesses would require some (impractical) inverse translation at the remote tile. This thesis' solution to this problem is to keep the virtual-to-physical address translations only in the TLB next to the home L1 cache and to ship virtual addresses over the network for remote cache accesses.

In this scheme a processor request proceeds as follows (see figure 7.2): firstly, the virtual address is simultaneously used to index the local L1 cache, to perform a local TLB lookup to obtain the physical address, and to perform a local MAP lookup to obtain the identity of the home L1 cache (❶). If the result of the MAP translation points to a remote L1 cache (❷), the local cache access is aborted (❸). In this case, the result of the local TLB lookup is also

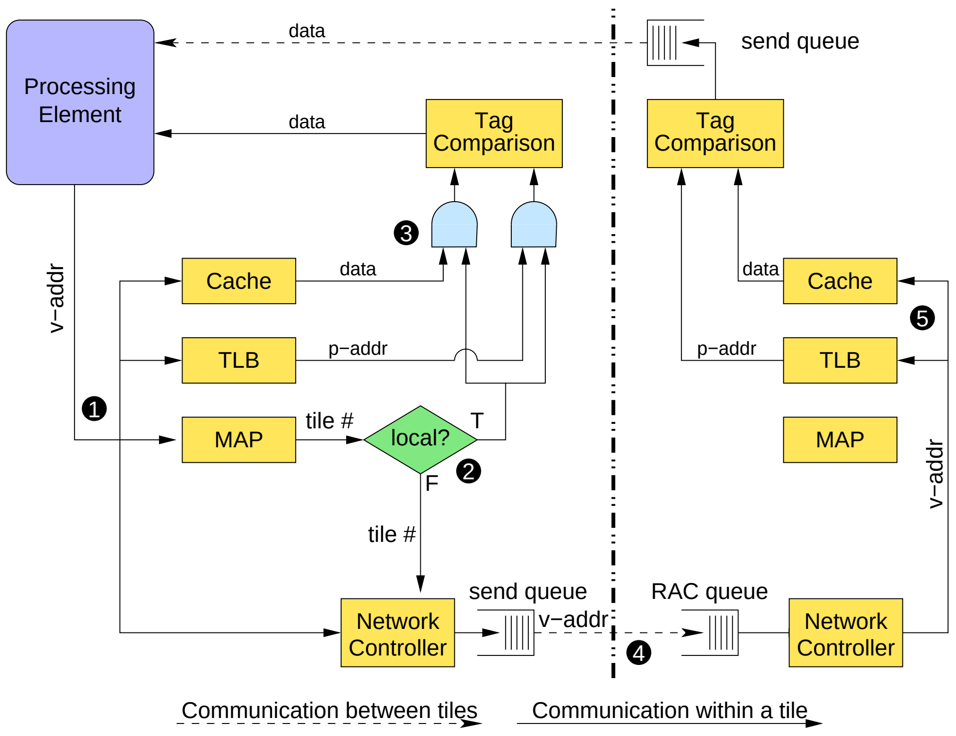


Figure 7.2: Remote Cache Access mechanism overview.

ignored, including a possible TLB miss. The virtual address is then shipped to the RAC in the remote tile over the network (❹). At the remote tile, the virtual address is simultaneously used to index the L1 cache and to perform a TLB lookup (❺). If the TLB lookup succeeds then a tag comparison follows, using the physical tag. A cache or TLB miss is handled as usual.

If the result of the MAP translation points to the local L1 cache then the local cache access proceeds as usual. To avoid delaying local cache requests due to remote cache requests it is assumed that the L1 cache has a dedicated port for incoming remote requests. This is also beneficial for hardware cache coherent systems. Similarly, the TLBs are given two ports, one for local and one for remote requests. Figure 7.3 shows the whole decision process. The chart also shows that the content of TLB and MAP does not have to be identical: if a virtual address A is mapped to remote tile X then there will be an entry in the MAP table for this address. However, since the access to the local TLB will be aborted before a virtual to physical translation can be loaded, the TLB will not contain an entry for virtual address A. Similar, since the MAP table on the remote tile X is not accessed at all during the process, it is possible that the entry for virtual address A has been replaced by some other entry and no longer exists in tile X's MAP table. A similar situation happens when a remote access replaces a TLB entry that is later needed by a local memory access: the access to the MAP table might hit, while the access to the TLB misses. While it would be possible to store MAP and TLB information in the same data hardware structure, this thesis did not investigate this implementation due to fact that the content of both structures can be disjoined. A similar discussion applies to how the operating system stores the OS version of the map and page table. It is possible to store both information within the same data structure. Whether this is possible depends on how misses in the TLB and MAP are being handled. If a hardware mechanism is used to handle such misses in a very fast way, then it might be required that each entry has to be stored in exactly one

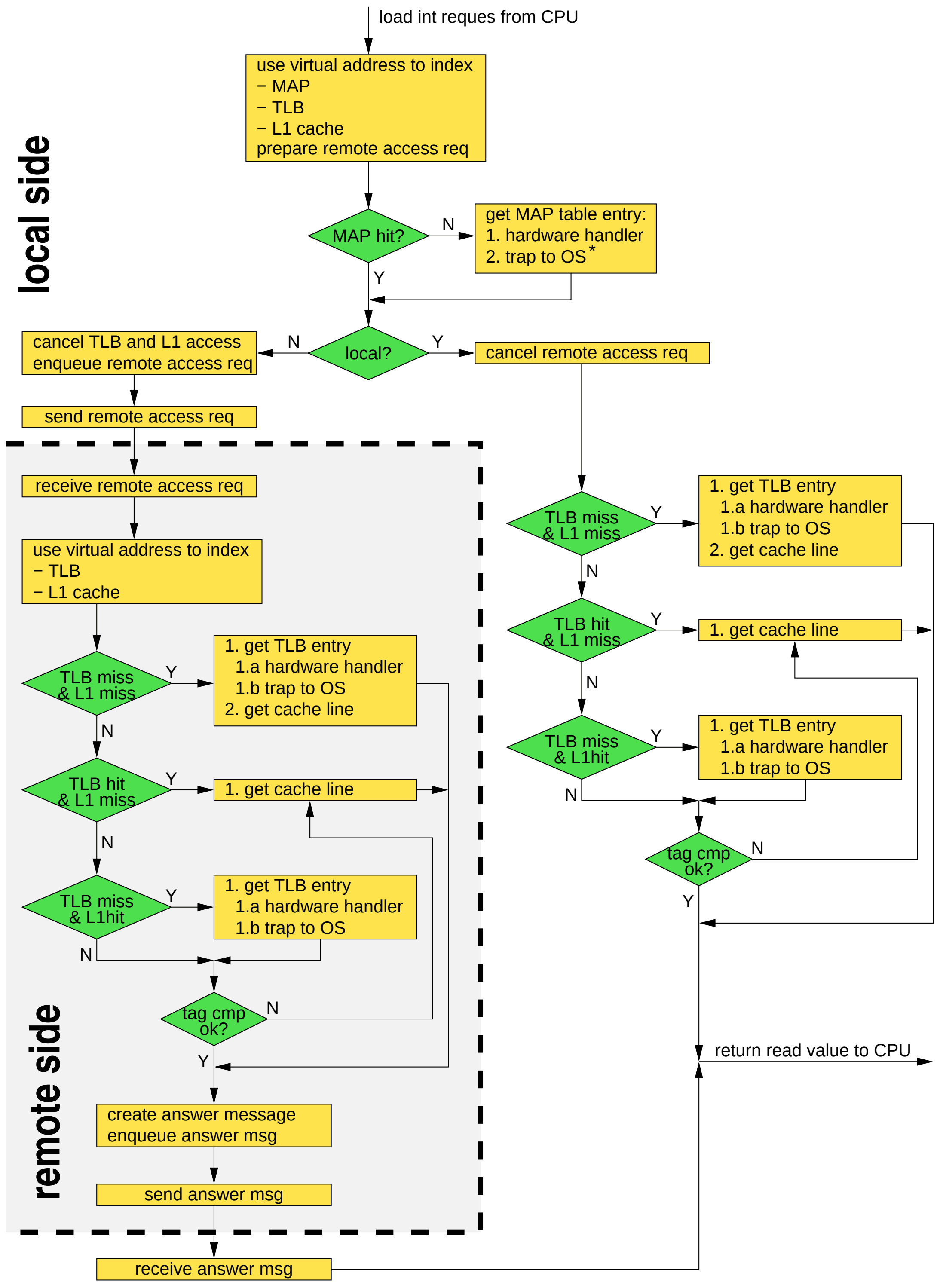


Figure 7.3: Flow chart of remote cache access mechanism, demonstrated with a load request.

word. Otherwise, that mechanism would become more complicated in order to avoid reading inconsistent data.

The above discussion only applies to data caches. Each tile has its own read-only instruction cache.

7.3. Migration Extension

The proposed first-touch allocation strategy combined with the fact that allocation is done at the granularity of pages may lead to poor performance when data migrates across threads. Mechanisms have been proposed to allow migration and replication of memory pages in CC-NUMA machines [VDGR96], but these are tailored to much larger systems with larger latencies and are inappropriate for a tiled CMP like the proposed one.

To alleviate this problem this thesis proposes a simple mechanism that allows for some degree of migration by invalidating the mappings of virtual memory pages to L1 caches. This is done by invalidating the MAP table in all tiles. After an invalidation, a first-touch policy is again used for the new mappings. Note that invalidating the mappings does not in itself migrate pages, but it creates an opportunity for this to happen. The invalidation is more easily implemented at a quiescent state where there are no pending memory requests on chip. A natural point to perform such invalidation is at barriers. In well-designed applications, especially those written for release consistency, barriers are used to signal change in the data access pattern and communication across threads. Thus, barriers are also natural good points for re-mapping and migration. Another important restriction is that all dirty lines in the L1 caches must be written back at a mapping invalidation or, otherwise, the modified data may be unreachable after the re-mapping.

The actual invalidation is done in two phases. Figure 7.4 shows an example of the migration process using two tiles X and Y. Figure 7.4a shows the state of the system just before tile X joins the barrier. Page *p* is owned by tile X and page *q* is owned by tile Y. The value for `b` has been modified in the local cache of tile X and has not been written back to memory yet. The moment tile X joins the barrier, it invalidates its local MAP table (see figure 7.4b). This is done with a new instruction added to the instruction set that is very similar to the existing instructions that invalidate the whole TLB content (for example, `tlbia` in the PowerPC instruction set). At this point, the local cache controller starts writing back dirty cache lines to main memory with the goal of hiding the write-back overhead with the idle synchronisation time (the updated variable `b` is written back to memory). Note that written-back lines may be modified again by remote tiles that have not yet joined the barrier, leading to duplicate write-backs. Experiments show that this occurs very infrequently. Such a situation occurs in figure 7.4c, where the value of `a` has been updated by a remote tile.

Figure 7.4d shows the state of the system just after all tiles have joined the barrier. This is the beginning of the second phase. Just before releasing the barrier one processor invokes a special system call so that the OS can also invalidate its internal virtual-address to tile table (in figure 7.4e this system call is invoked by tile Y). Also at this point (before releasing the barrier) all tiles write back all remaining dirty cache lines. When the writebacks are completed, the contents of the caches are invalidated and the barrier is released (as shown in figure 7.4f). As can be seen, all modified values have been written back to memory. Furthermore, since all MAP tables (the OS one and all hardware ones) have been cleared, the system state is identical to the state at the beginning of execution of the application. Thus, the normal mechanism (as

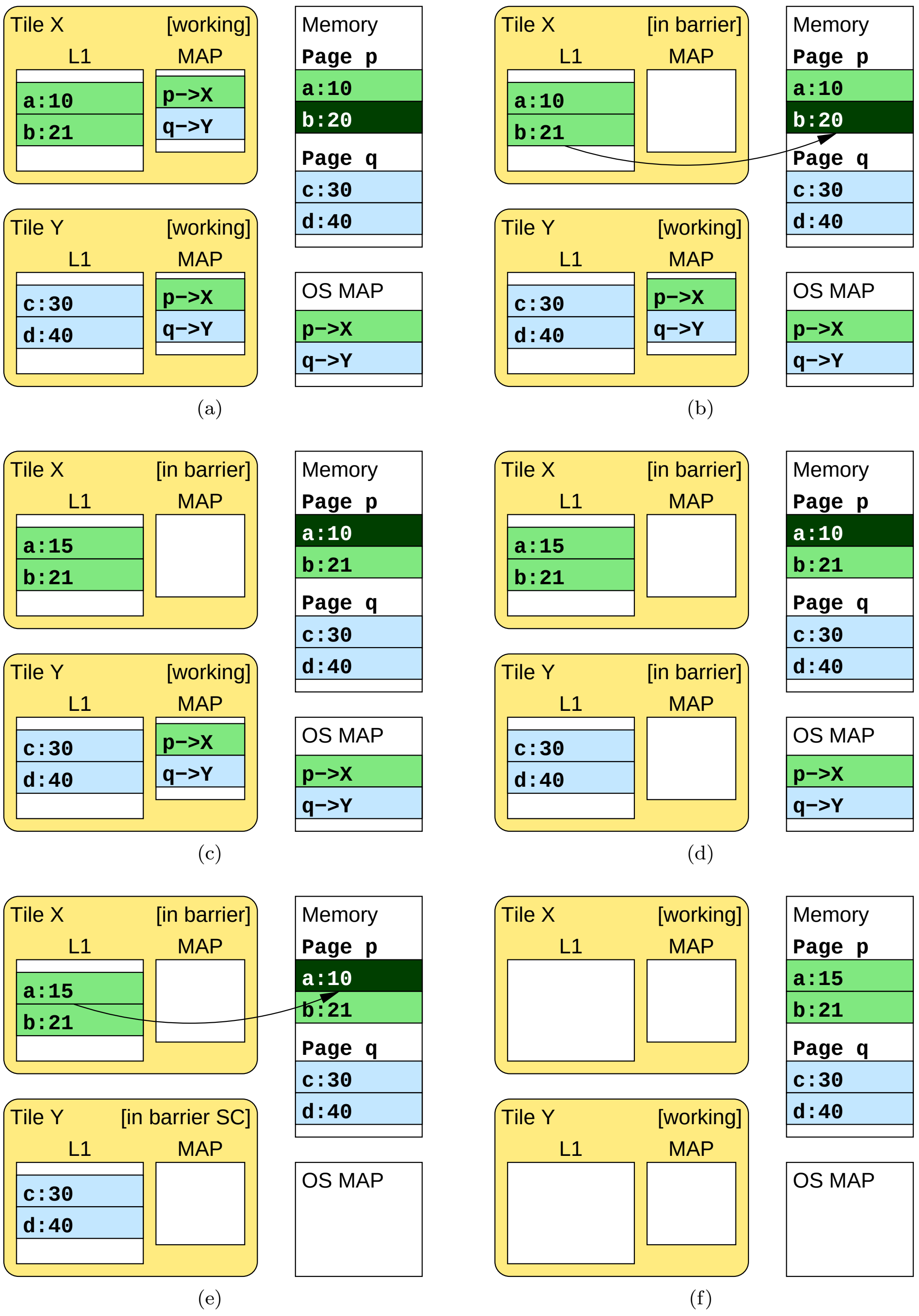


Figure 7.4: Example of the migration extension. The variables `a` and `b` are stored in page p, while `c` and `d` are stored in page q. Memory and cache content printed in inverse indicates stale data.

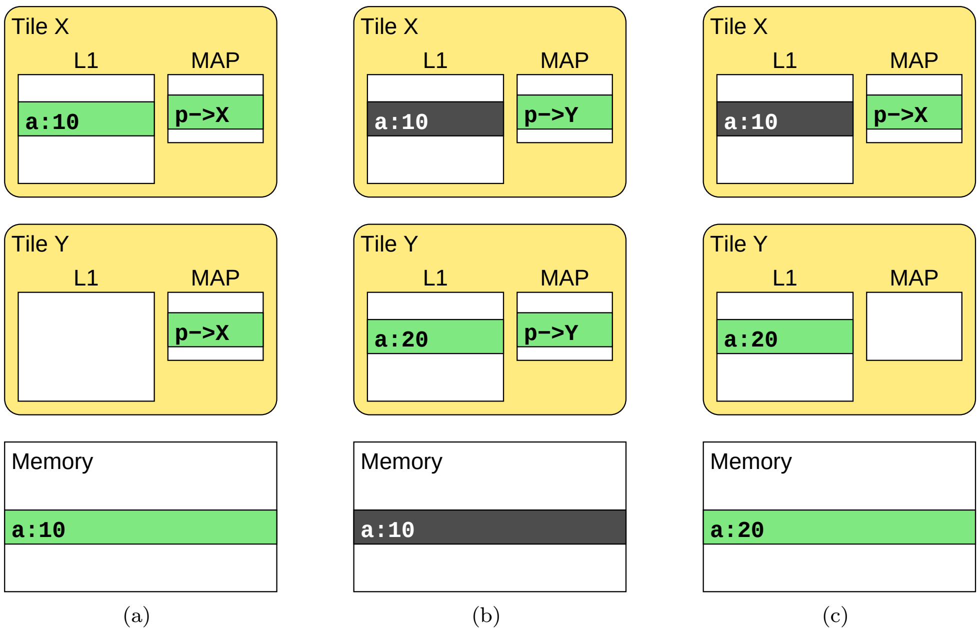


Figure 7.5: Illustration for the reason for invalidating caches at barriers. Part (a) and (b) show the state of the system before and after the first barrier respectively, while part (c) shows the state a little bit after the second barrier. Memory and cache content printed in inverse indicates stale data.

described in section 7) will ensure that the MAP tables will be repopulated again and that no incoherence issues arise.

Strictly speaking it is not necessary to invalidate the content of the caches at every barrier. On a first glance, invalidation is not necessary at all. If the page is migrated to another tile, then the data in the cache will not be accessed and eventually replaced by the LRU (least recently used) policy. If the page is mapped to the same tile, then the data is already in the cache, and does not have to be refetched from memory. Thus, not invalidating is potentially beneficial. While these observations are correct for a single barrier, they do not hold in case of two barriers. Figure 7.5 illustrates the problem. Consider that a variable `a` on page p is mapped to tile X. Tile X caches the variable (either because it accesses the variable itself, or because it serves a remote access) as shown in figure 7.5a and enters the first barrier. If the variable has been modified, then it would be written back to main memory (the figure assumes that there was no need to write it back). However, it does not invalidate the cache line that contains `a`. After the first barrier, the page p is mapped to tile Y and some node updates the value of `a` to 20. Even though tile X still has a cache entry for variable `a` (which is stale), it is not a problem since the current MAP table entry will prevent tile X from accessing the local cache line. Instead, it will perform a remote cache access that will return the current value. The state of the system at this point is shown in figure 7.5b. At this point, Y enters the second barrier. Since variable a has been modified, it has to be written back to main memory. After the second barrier, the page p is mapped again to tile X as shown in figure 7.5c. If now a read access is performed to `a`, then there is a chance that the old value of `10` is returned (in the figure this is the case). If the old or the current value is read depends on whether the cache line containing `a` was evicted from the cache while the page was owned by tile Y. If it was evicted, then it will be reloaded from main memory with the current value; otherwise the stale value

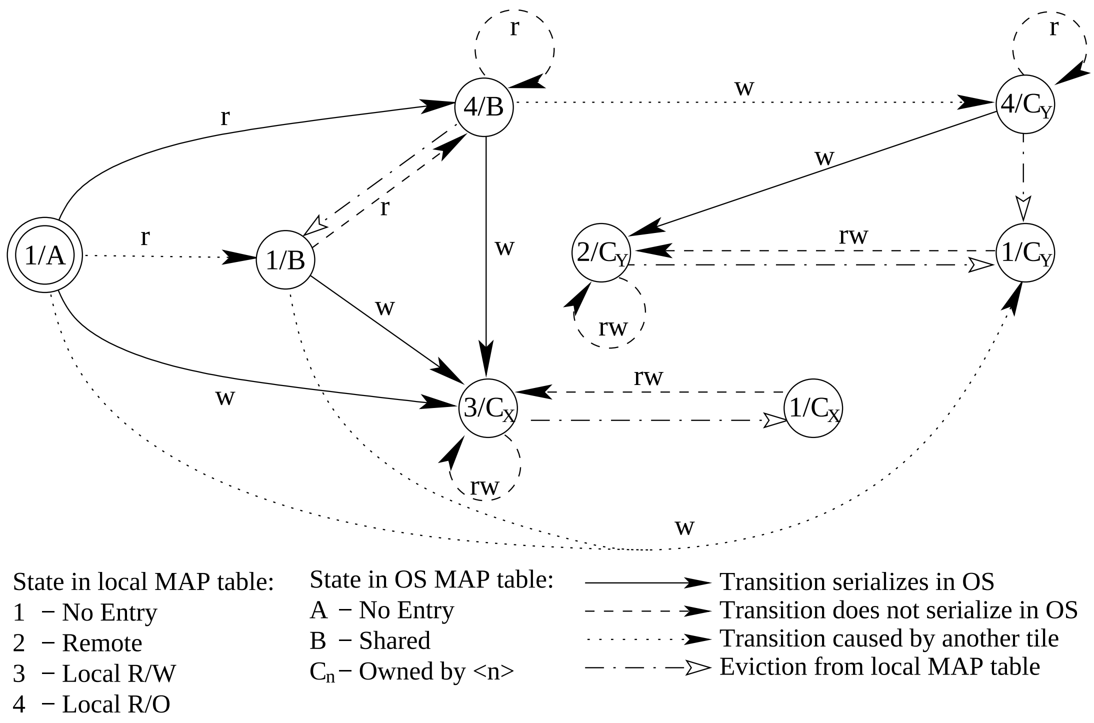


Figure 7.6: Sharing protocol for node X. The state of a page depends on its local (numbers) and OS (letters) state. The OS state might also change, because some other node Y performs a transition that updates the OS MAP table. Such state changes are marked with a dotted line. The operation listed in braces, indicates what kind of operation had to be performed on Y to trigger the state change. A transition serialises in the OS, if it performs an update to the OS MAP table that requires locking the table. Otherwise, the transition can be performed without locking the OS MAP table, assuming that a load word instructions atomically loads the whole word.

will be returned. While the chance of the cache line not being evicted is relatively small[1], it cannot be ruled out. One can also easily see that if the cache had been invalidated at either the first or the second barrier, then this problem would not have occurred. Thus, for correctness it is not necesary to invalidate the cache at every barrier, but at every other barrier.

7.4. Read-Only Sharing Extension

The scheme proposed to map pages to physical caches coupled with the extension to refresh such mappings to allow migration is likely to work well as long as there is not much sharing of data at the granularity of pages. While full-blown sharing requires line-based hardware coherence or complex page based software DSM coherence, some degree of sharing can be easily enforced by the OS with minimal hardware support. What this thesis proposes is a simple mechanism that allows sharing of pages across multiple readers and a single writer at any given time. The full protocol for a tile X is shown in figure 7.6. At the beginning all pages are in state "1/A". This situation is shown in figure 7.7a. Assuming that tile X is the first tile to perform a read access to page p, then this request will trap to the OS. To be precise after the initial miss, the

[1]The cache line will not be accessed at all while the page is owned by Y. Thus, it is a good candidate for replacement according to the LRU strategy.

fast PowerPC hardware miss handler will try to load the entry and also fail. At this point, an exception is raised and control has to be transferred to the OS. The OS then sets up an entry in the OS MAP table, and marks the page as read-only in this table and the tile's local TLB and MAP. The new state for this page is now "4/B" from the point of view of tile X. On all other tiles the new state of the page is now "1/B". In figure 7.7b, such a read access has been performed to variable `a`. Note that these tiles do not have to be notified of this change and other transactions that only affect the OS side state (the letters in figure 7.6). Such "invisible" transitions are marked with a dotted line in figure 7.6. If these other tiles (that are now in state "1/B") now try to read from this page, then they are also allowed to create a local read-only mapping (and change into state "4/B"). Since the entry already exists in the OS MAP table, this miss can be handled successfully by the fast hardware miss handler. In figure 7.7c, tile Y reads variable `b` and causes such a change. Unlike directory based schemes, the OS *does not* need to keep track of which tiles are sharing the page. This would also be difficult, since the OS is not aware which tiles become sharers by means of the fast hardware miss handler. To summarise, two different local MAP table misses are possible. The first type of miss is due to the fact that there is no entry in the OS MAP table. In this case, an exception has to be raised and the OS has to insert such an entry. The second type of miss is normal miss like any cold, capacity or conflict miss in a TLB or cache. This type of miss can be handled by the fast hardware miss handler. Or, if the architecture lacks such a feature, then the OS can obtain the entry by only performing read operations on the OS MAP table (assuming that a load word instruction atomically loads a whole word).

Now, assuming that tile X is the first one to write to page p, then the OS intercepts the write, marks page p as modified and makes processor X the owner of the page "3/C_X". In figure 7.7d, tile X writes to variable `a`. Thus from tile X point of view, page p is now in state "3/C_X". Also note that for tile Y the same page is now in state "4/C_X". If a different tile Y is the first one to write to page p, then it causes an "invisible" transition on tile X. Either into state "4/C_Y" if it had a previous read-only mapping "4/B", or into state "1/C_Y" if it had no previous mapping. In figure 7.7e, tile Y writes to variable `c`. Since tile X does not have an earlier local mapping of page q, page q is now in state "1/C_Y" with respect to tile X. Subsequent reads by processor X with an existing local mapping ("4/C_Y") can continue to use this mapping, and, thus, access local data. However, subsequent writes by processor X to pages, that are either in state "1/C_Y" or "4/C_Y", are then intercepted by the OS and are not allowed to proceed locally. Instead a local MAP entry will be generated (or changed if one already exists) that points to the owner node (the entry is now in state "2/C_Y"). Figure 7.7f shows this situation from the perspective of tile Y. Tile Y performs a write access to b. This write access is intercepted, the local mapping for page p is replaced by a remote mapping, and the request restarted as a remote request. Also note that the cache in tile Y now holds a cache line that contains stale data. This data cannot be accessed as long as there is a remote MAP table entry. The implication of the presence of this stale data will be discussed later. Similarly, reads by processors without a local mapping "1/C_Y" for the page will generate an entry pointing to the owner node "2/C_Y".

The mechanism just described allows processors to continue using local mappings and locally cached data. However, as can also be seen from the state diagram, pages that are in state "4/C_Y" on tile X, are a potential cause for stale data access. Even though tile Y has claimed ownership to such a page and modified its data (otherwise it would not have become the owner), a stale version of that data might be still in the cache of tile X. This situation is

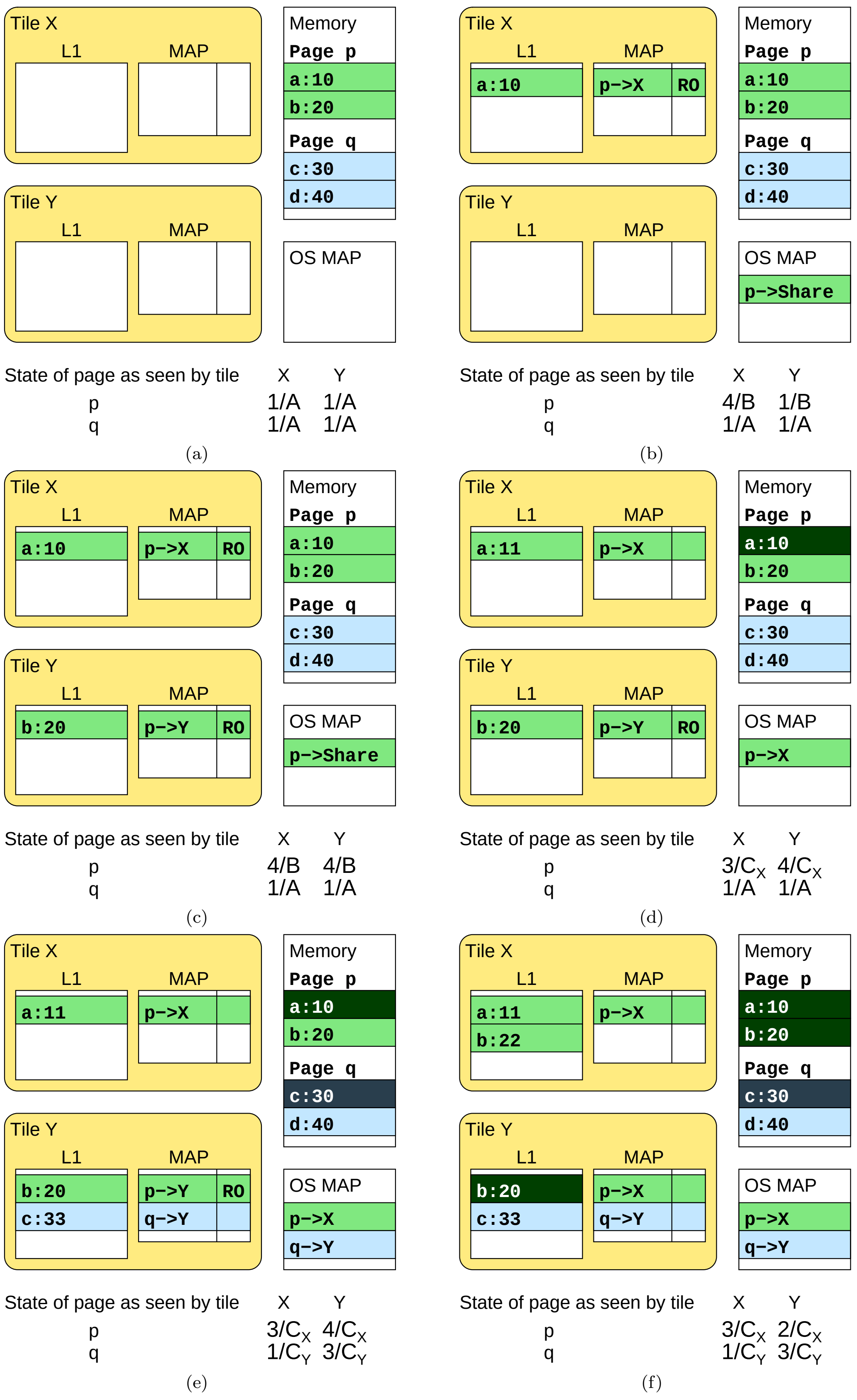


Figure 7.7: Example of the read-only sharing extension. The variables `a` and `b` are stored within page p, while `c` and `d` are stored in page q. Memory and cache content printed in inverse indicates stale data.

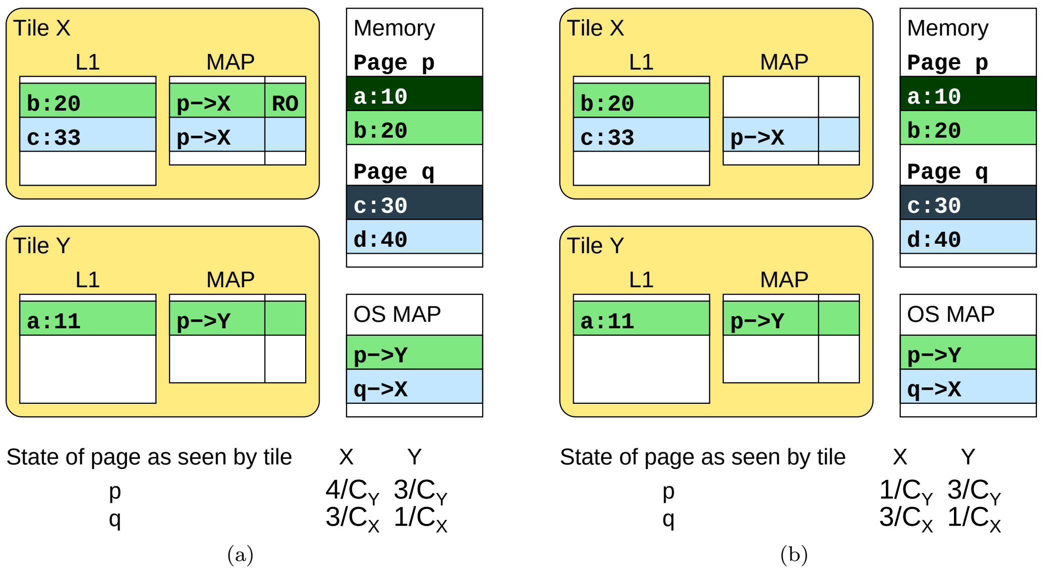


Figure 7.8: Example of the read-only sharing extension. The variables a and b are stored with in page *p*, while c and d are stored in page *q*. Memory and cache content printed in inverse indicates stale data.

shown in figure 7.8a. Tile X has a local read-only mapping of page *p*. If tile X would now perform a read access to variable a, then it would access the out of date version stored in main memory. To prevent use of this stale data, this thesis assumes the release consistency memory model [GLL+90] and invalidates local MAP entries for potentially shared pages (these are pages that are in state “4/B” or “4/C_Y”) on lock acquire operations. This can be done by performing the following logical operation with a bit extra hardware support (2 inverters and 1 NAND gate) on the valid and shared bit[2] of every entry: $valid = valid \ \& \ \neg shared$. This ensures that all pages in state “4/C_Y” are reset into state “1/C_Y”. Unfortunately, also pages that are in state “4/B” are set to state “1/B”, even though there is no need to do so. This will cause some overhead of avoidable misses in the local MAP table. However, the only way to avoid these would be to synchronise the local MAP table with the OS one. Whether this can be done in less time than the overhead caused by the misses, depends on the availability of a mechanism that can synchronise the MAP table issuing fewer memory accesses than which are necessary to refetch the invalidated entries. In particular, not all entries that are invalidated might be accessed afterwards. Thus, synchronising such entries would not be beneficial at all. From the diagram, it can also be seen that pages that are in state “2/C_Y” or “3/C_X” do not have to be invalidated, since these states are final states (except for the possible state change due to eviction from the local MAP table). These states can only be left by resetting both OS and local MAP tables. This only happens at a barrier in order to allow migration of ownership. By performing the selective invalidation, it is guaranteed that all accesses to data modified by other processors will use a new remote mapping and will become remote. Figure 7.8b shows the state of tile X just after the selective MAP table invalidation. An access to variable *a* would now create a remote mapping in tile X’s MAP table and access the current version in tile Y’s cache. Note that cache lines that contain data from page *p* are not invalidated.

[2]Technically speaking: the tile does not know if the page is really being shared with another tile. It only knows that the page can only accesses for reading and might be potentially shared. Thus, the bit should be called “potentially shared”. However for practical reasons, this thesis refers to this bit just as “shared”.

As mentioned before, it is possible that cache lines contain stale data (as shown in figure 7.7f; tile Y caches a stale value of variable `b`). A similar situation can happen in figure 7.8b, if an update to variable `b` is performed. While this data cannot be accessed as long as a remote MAP table entry exists, a barrier could delete such an entry. If now the cache would only be flushed at every other barrier, then it could be possible that such a stale entry is accessed with a new local mapping after the barrier. Thus, it is also necessary to extend the barrier actions used with the migration mechanism of section 7.3 to include a full cache flush in addition to the writebacks and the refresh of the mapping. It is no longer possible to just invalidate caches at every other barrier. Note that no special action is required on lock releases.

The protocol for the read-only extension described so far seems to be asymmetric: if tile X reads a from a page *p*, then it can continue to read using a local copy even after tile Y has written to the same page at a later point. As discussed before, this behaviour is acceptable, if release-consistency is assumed. However, if tile Y writes to the page *p* first, and then tile X tries to read from this page, the tile X has to perform a remote access. This behaviour is not incorrect, but according to release consistency not necessary: it should be possible for tile X to perform read-only accesses to page *p* using its local cache. This observation is indeed correct. However, it only holds as longs as no locks are involved. Consider the following: Tile Y writes to page *p* and becomes its owner. Afterward it releases a lock that is then acquired by tile X. Tile X now wants to read from page *p*. Allowing tile X to access *p* using its local cache, might result in X not being aware of the changes Y has made. Thus, in this case it is not possible that tile X performs local read accesses to page *p*. The only way to avoid this is to keep track of which pages were updated inside a lock-block and which were updated outside. But even without considering locks, the following situation might occur: Again tile Y writes to *p* and becomes its owner. Tile X now also wants to write to variable `a` (stored in a different part of page *p*), resulting in a remote cache access to tile Y. Now tile X performs a couple of other memory accesses, resulting that the entry for *p* is replaced by some other entry. At this point, tile X tries to read the value of variable `a` again. If it now creates a new entry in the MAP that allows to local read-only access, it might be possible that tile Y has not yet written the updated cache line containing `a` to main memory. In this case, tile X would read a stale value. In order to avoid this situation, it would be necessary to record which tiles already performed a write access to which page at some point in the past. Based, on this two examples it can be seen that this asymmetry in the protocol is indeed required to guarantee correct execution. Otherwise, the protocol would have to be extended to deal with these cases correctly.

While this mechanism may seem very similar to previous software cache coherence mechanisms (e.g., [KHS+97]), it differs from these in one crucial way. Namely, it does not allow multiple writers, reverts to a single up-to-date copy of every page upon a write, and enforces remote cache accesses in such cases. The key benefit of this is that in the proposed scheme, no multiple modified copies of physical pages exist at any time and, thus, there is no need to perform expensive diff operations and copying of data in memory.

7.5. Hardware Complexity

Since this thesis proposes to replace a directory controller and its protocol with an RAC, a MAP table and their combined hardware/OS protocol, it is relevant to compare both schemes' area and complexity overheads. In particular, the main goal of the proposed scheme is to provide a less complex alternative. A comprehensive comparison between the two competing approaches would require the full design of the controllers and their circuit implementation. This, unfortunately, is a highly involved task and, instead, this thesis attempts to provide some intuition into why this thesis claims that the proposed scheme is less complex.

Like a directory controller, the RAC has to handle remote read and write requests. Unlike a directory controller, it does not have to deal with forwarded transactions and multiple invalidations, which lead to complex protocols with subtle race conditions and several pending states. The RAC can directly handle requests and generate responses for all transactions in the proposed protocol. Thus, the RAC has fewer states and a much simpler finite state machine, which means that it has simpler logic than a directory controller does. More importantly, this means that the resulting protocol is simpler to verify and validate.

In addition, the MAP table contains information at the coarse granularity of pages and it is only a hardware cache for an OS data structure. A directory, on the other hand, contains information at the finer granularity of cache lines and must maintain information for all cache lines in the chip. In the evaluated systems, the MAP table has a fixed size of 128 entries and each entry has 15 bits for the virtual address tag (for a 32bit address and 4KByte pages), 5 bits to name each of the 32 tiles, and 1 valid bit, leading to a total of 336 bytes per MAP table. For a 32 tile system with 32KByte L1 caches and 32byte lines the directory requires 32 bits for the full sharing vector and 2 bits for the line state for each of the 1024 lines per home directory, leading to a total of about 4KBytes per directory.

On the negative side, the proposed system requires an additional port to the 4-way associative TLB to handle remote accesses independently from the CPU. Also, a few simple, extra hardware structures are needed to support correct synchronisation (these structures are explained in section 8.1). The required structures are a time out mechanism for remote LL requests (including the generation of “reservation cancellation” messages), a bit vector (one bit for each tile in the system) to keep track to which tiles remote stores have been issued, and a modification of `eieio` instruction to ensure that all memory operation have been completed (this the generation of dummy loads to ensure the stores have completed). Note that the cache has a similar number of ports as a cache in a directory scheme. In a directory scheme, the directory controller also needs access to the cache in order to invalidate a cache line or to send the dirty content back to the home node. The only difference is that in a directory scheme the directory controller does not need access to individual words in the cache line. Also, similar to current non-symmetrical CMPs, such as those with clustering of processors and caches and those with multithreading, the proposed architecture requires OS changes because it exposes the internal chip structure to the OS.

7.6. 2-Level Cache Hierarchy

The proposed system so far has only one level of on-chip cache. In order to process larger sets of data, the amount of cache per tile has likely to be increased. The most suitable option for this would be to introduce a multi-level cache hierarchy. This decision offers several design options that are shown in figure 7.9. The probably most intuitive version is shown in figure 7.9a, where the second level cache is simply attached as a backing store to first level cache. The coherence between L1 and L2 can be easily arranged by a write-through L1.

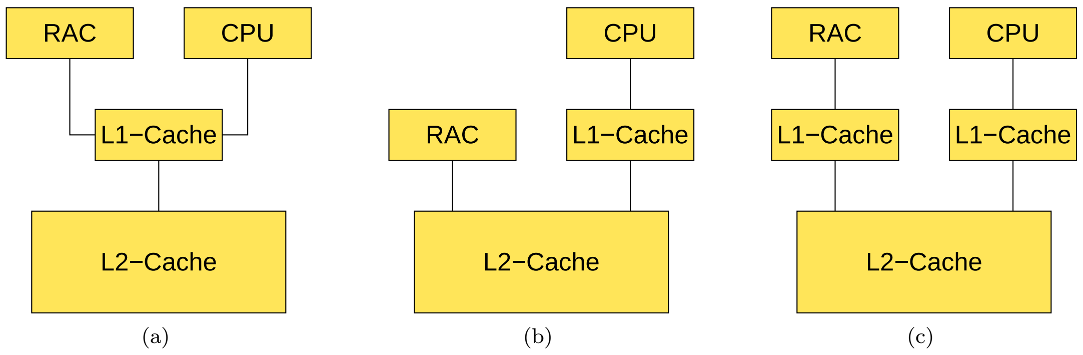


Figure 7.9: Possible options to design a tile with multiple levels of cache.

However, other arrangements are possible. By connecting the remote cache access controller (RAC) directly to the second level cache, one could hope that the first level cache gets less polluted by remotely requested data and, thus, offering a higher hit rate for local CPU requests. On the downside, serving remote accesses would now take longer, since they can only be served at the hit latency of the L2. Furthermore, the design of the L1 becomes more complicated, since it also has to respond to invalidation messages from the L2, in case a cache line has been modified remotely. Such an arrangement is shown in figure 7.9b.

The option shown in figure 7.9c would overcome this problem by adding a dedicated first level cache to the RAC controller. The obvious drawbacks of this approach are the extra costs for another first level cache and the logic required to keep the two private L1s coherent. This configuration was not evaluated, since it seems obvious that it would perform better than option (a) and (b). Furthermore, this performance would be gained by sacrificing the relatively simplicity of a single tile.

7.7. Performance Issues

This section gives a quick overview of potential performance issues. The most important performance issue is the ratio between local and remote accesses. While the latency of the network is already quite low, it still requires 2 cycles (3 or 4 cycles in case of a store) to just send a request using the network. Furthermore, the network might become too congested and can no longer provide the low latencies it is able to offer. Remote accesses especially become an issue if they all target the same tile (for example to access a lock variable). In such a case this tile might not be able to queue up incoming request and had to start replying with nacks.

Another variant of this problem (that too many requests are sent to the same tile) is that cache lines are getting replaced due to conflict misses. Even though all requests are accessing different data, it might be possible that this data happens to map to the same cache line. This could become a problem, once more larger number of tiles than the associativity of the cache try to access data remotely on one tile.

Another common problem in shared memory systems, is false sharing: different words in the same cache line are read and written by different processors. Even though each processor is just accessing its own words and no data is shared, the cache line gets invalidated since consistency is maintained at a larger granularity than words. False sharing that results in invalidation is not possible in the proposed scheme. However, a similar effect is possible if, for example, tile

X only accesses the first half of a page p and tile Y the second half. Even though no data is actually shared, the scheme will assign ownership of p to one tile and forces the other to perform remote accesses.

These performance issues will be addressed during the evaluation of the proposed scheme in chapter 13.

8. Synchronisation

8.1. Locks

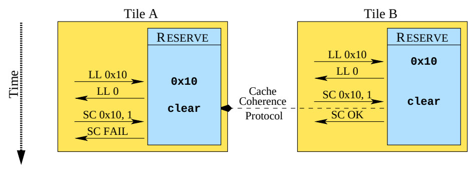
Memory locks have been implemented in the past using either *compare&swap*-style atomic instructions or *load-link store-conditional* (LL-SC) pairs [CS99]. The latter approach has been favoured recently because it allows for the implementation of a variety of higher-level read-modify-write operations in software and because it is easier to implement in hardware with cache coherence.

In the proposed architecture, *compare&swap*-style primitives can be more easily implemented than in current multiprocessors. This is because there is no replication of the lock variable in multiple caches and it is, thus, easier to enforce the atomicity of the primitive. To implement these operations then only requires adding the compare logic to the cache controller and blocking subsequent requests from other processors until the swap is performed.

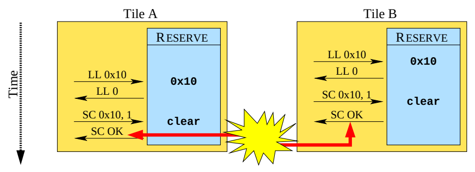
On the other hand, implementing *load-link store-conditional* pairs in the proposed architecture is more difficult than in current multiprocessors with cache coherence. These instructions work by registering the load with a LL and subsequently performing a store that only succeeds if there are no other intervening stores from other processors since the LL. In current CMPs, this is easily achieved by keeping a RESERVE register in the local L1 and relying on the hardware coherence mechanism to detect conflicting stores (figure 8.1a). Keeping the RESERVE in the local L1 of the requesting processor will not work, however, without cache coherence (figure 8.1b). Instead, to implement the LL-SC primitive the RESERVE register is placed in the home L1, and this register is then shared by all processors attempting to obtain any locks that map to this L1 cache.

One problem with this approach is that a livelock is possible when processors attempting to lock *different lock variables* displace each other's LL from the RESERVE register (figure 8.1c). Note that this overwriting of the RESERVE register does not happen in CMPs with private L1 caches and private RESERVE registers, and there is never a conflict between attempts to lock different lock variables. This work's solution to this problem is to change the operation of the RESERVE register such that once set it cannot be overwritten by LL requests to other lock addresses.

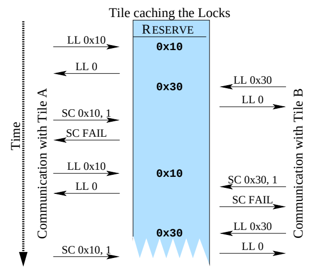
Another problem with this approach is shown in figure 8.1d, where more than one processor obtains the same lock simultaneously. 1) tile A and B both try to obtain the same lock X at address `0x10` by sending LL requests to the tile that is caching the lock; 2) A's LL registers the address of X in the RESERVE register and returns an "open" lock value; 3) B's LL does nothing to the RESERVE register, but also returns an "open" lock value (at this point A and B are correctly competing for the lock); 4) both A and B on observing the lock "open" issue their matching SC; 5) A's SC succeeds and clears the RESERVE register, such that any subsequent SC to X will fail; 6) before B's SC reaches the tile, tile C issues a LL request to X, which returns a "closed" lock value, but registers the value of X in the RESERVE register; 7) finally,



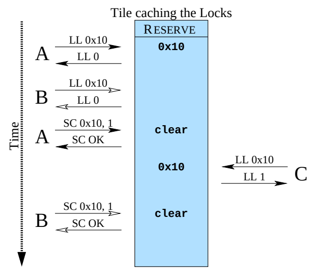
(a) Lock acquire in a system with cache coherence.



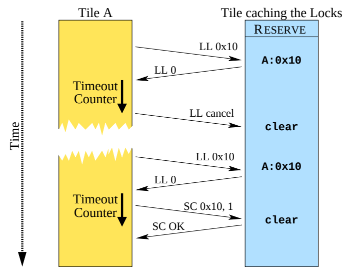
(b) Lock acquire in a system without cache coherence.



(c) Live-Lock.



(d) Same lock acquired twice.



(e) The timeout mechanism to prevent deadlocks.

Figure 8.1: Problems of locks implemented with *load-link* (LL)/*store-conditional* (SC).

B's SC reaches the tile and is matched against C's LL and is considered successful. Thus, in this scenario, both A and B may have been allowed into the same critical section simultaneously.

This thesis' solution to this problem is to extend the RESERVE register with the ID of the tile that successfully sets it, and to only consider successful SCs that match the value in the register and come from the same tile.

The solutions proposed to the two problems above lead to another problem when the thread holding the RESERVE register fails to issue the matching SC, either accidentally or maliciously. To handle this, a timeout mechanism is introduced to clear the RESERVE register. The problem is that this timeout counter cannot be placed in the tile that holds the RESERVE register due to variabilities in latencies in the network. It would be impossible for this tile to decide with absolute certainty that it should have received an answer for the LL. Instead the timeout mechanism is placed in the tile that sent the LL. The timer will start in the cycle the response to the LL request has been received. The processor then has to issue a matching SC request within a given number of cycles, otherwise a special *reservation cancellation* message is sent to the tile holding the RESERVE register. This message is automatically generated by the hardware once the timer runs out. Similarly, since the requesting tile can only keep track of a single LL request answer, issuing a LL request to a different address will also result in the immediate generation of a *reservation cancellation* message sent to the tile that was responsible for the previous request. Note that this timeout mechanism also applies in the case that the tile sending the LL request and holding the RESERVE register are the same tile.

One consequence of just having a single copy of the lock variable is that the *test&test&set* does not result in the same traffic reduction as one might expect. *Test&test&set* is a technique to reduce coherence traffic in the case that the lock is already taken by another thread. The idea is to use a normal load to perform the first test, and only if this first test shows that the lock is still available then issue a LL instruction that requires exclusive access to the lock variable. Since the normal load does not need an exclusive copy, it will just access its local copy without causing further coherence traffic. In the proposed scheme these normal loads will result in remote cache accesses so, they still create network traffic. Still *test&test&set* has also some benefits in the proposed scheme: receiving the answer for a LL request will always start the above mentioned timeout counter. In the case that the lock is taken, no SC will be issued and as such a *reservation cancellation* message will be sent. Using the *test&test&set* approach can at least reduce the number of *reservation cancellation* messages sent.

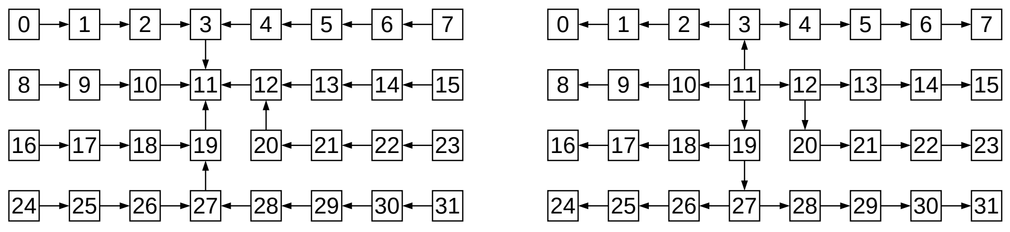
The discussion so far assumed a lock implementation on the baseline architecture without read-only sharing extension (described in section 7.4). Using this extension causes one potential problem: some pages might be mapped as read-only shared in the MAP table of the lock acquiring tile even though they are now owned in the OS MAP table. Accessing such a page could result in potential access to stale data. However, very little change is necessary to adapt the lock to support this extension: after the lock has been successfully acquired the local map table has to be invalidated. One final side effect of using the read-only sharing mechanism is that the LL instruction has to be treated as a write when it comes to replication. Thus, the first processor to issue a LL becomes the owner of the page that contains the lock variable. Subsequent LL instructions from other processors will trigger a change in their local mapping, if there is one, so that all accesses to that page will become remote. This guarantees that any updates to the lock variable will become visible to processors issuing LL instructions even if they previously had a read-only copy of the page.

Appendix C.2 shows some sample code of the lock implementations.

8.2. Barriers

Current CMPs implement barriers in software using locks. In the proposed architecture, barriers can also be implemented using the static network as follows. An attempt to read a value from a network link when the corresponding buffer is empty will stall the network or compute processor until a value becomes available. Thus, this blocking interface can then be used to stall and release processors involved in barriers. In this scheme, the compute processor first writes a value to the static network (i.e., through one of the registers mapped to the static network), informing the network processor that it is joining a barrier, and then reads a value from the static network and blocks. The network processors are then responsible for collecting the information about all compute processors that have already reached the barrier and, once all have reached the barrier, they are responsible for releasing the compute processors.

More specifically, the barrier process can be divided into two phases as shown in figure 8.2. During the first phase, the compute processor informs the static network processor on its tile that it is joining the barrier. The network processors are organised in a tree structure where the longest path from a leaf node to the root has length $\lceil\sqrt{n}\rceil$, with n being the number of tiles. Each network processor is responsible for waiting until all its children have joined the barrier and, once all have joined, is responsible for passing this information to its parent network processor. This process is shown in figure 8.2a. For example, the static network processor in tile 3 waits for a message from its neighbours (tiles 2 and 4) and its own PE informing that they have entered the barrier. Only then it will send a message to tile 11 indicating that all tiles in the first row (tile 0 to 7) have entered the barrier. This signalling is done by simply sending any value over the static network. After signalling the arrival at the barrier to the parent network processor, each network processor reads a value from the network link of the parent network processor and waits for the signal that the barrier has been released. In the above example, the



(a) Entering the barrier.

(b) Release of the barrier.

Figure 8.2: Propagation of a barrier across an 8x4 tile chip.

network processor in tile 3 tries to read a value from the link to tile 11 and stalls until such a value becomes available.

From the PE's point of view, the barrier operation so far works as follows: the PE sends a word via `r24` to the static network processor. This word is the start address for the barrier code in the network processor. Afterwards the PE tries to read a word from `r24`; this operation will stall the PE.

The second phase starts once the root tile (tile number 11 in figure 8.2) has received the signal that all tiles have joined the barrier. At this point, a signal to release the barrier is sent down the tree following the inverse path of the join signal. Upon receiving the release signal, each network processor releases its compute processor and sends the signal further down the tree. This process is shown in figure 8.2b. To continue with the example, once the network processor in tile 3 receives a message from tile 11, it will forward this message to its local PE, which then resumes execution, and to tiles 2 and 4.

Figure 8.2 also shows that in order for this barrier mechanism to be efficient, it is necessary that all participating threads run on tiles that are next to each other. For example, this mechanism could not be employed if only the threads running on tile 0, 7, 24 and 31 wanted to participate in the barrier. In such a case, a fallback to a lock-based barrier implementation would be necessary. However, such situations would be unlikely: in most cases all threads of an application participate in a barrier. Furthermore, the remote cache access mechanism already suggests that it would be beneficial to group these threads spacially together. As such, the OS would most likely try to place the threads of an application to a cluster of tiles. If this is not the case, then again the fallback mechanism of a lock-based barrier has to be used.

This thesis proposes that the operation described above be implemented as a library with the compute and network processor codes. Also, to perform this barrier synchronisation as quickly as possible this thesis proposes to fix the topology of the tree statically for each number of tiles participating in barriers. Then specific code for each topology can be either fixed at compile or link time or can be chosen at run time from a set of pre-compiled codes.

The barrier mechanism discussed so far is the barrier mechanism for the baseline architecture without any extensions. In order to realise the extensions discussed in section 7.3 it is necessary to add the following actions to the scheme described above: first, the tile should write back dirty cache lines. Second, the tile invalidates its local MAP table. After these have been completed, it can join the barrier tree scheme (as described above). However, unlike above, the barrier is not complete once all tiles have joined the barrier. At this stage all caches have to be flushed (including writebacks of cache lines that became dirty again, since the first write back). In addition, one tile has to execute a special system call to reset the global MAP table. In principal, this can be any tile (as long as it is only one tile that is assigned to this task), but it was decided to use the tile that is at the root of the barrier tree. This choice also might improve performance a little bit, since the reverse traversal of the barrier tree can be overlapped with the execution of the system call. To ensure that all these operations have been completed, the tiles join a second barrier tree scheme (identical to the first one). Once this barrier tree scheme signals it has been completed, the barrier can be considered lifted.

Appendix C.3 shows some sample code of the barrier implementations.

8.3. Enforcing Proper Order of Memory Operations at Synchronisation Events

Synchronisation events such as locks and barriers require that no memory access is moved across them. Thus, all outstanding loads and stores have to be completed. PowerPC provides the `eieio` instruction that only allows execution to continue after all memory operations have been completed. The proposed architecture uses the same instruction for this purpose. Ensuring that no load instruction is waiting is relatively simple, since the processing elements knows how many load buffers are currently allocated for pending loads. Remote stores are a more complicated problem. The easiest solution would be to force the remote cache access controller (RAC) to acknowledge every store. However, this would double the amount of network traffic for stores. Instead, the proposed architecture takes advantage of the following network characteristic: while the dynamic network does not guarantee any latencies or in what order messages sent by different tiles to the same destination are received, it does guarantee that message sent between two tiles arrive in the same order they were sent. Moreover, the RAC processes remote requests in first-come first-served order. Thus, if a load is sent to the same tile as a previous store, then the store must have been received and completed before the load could be processed. Hence, the tile sending the load will know that the store also completed. The load indirectly acknowledges the completion of the store.

This idea is used to avoid generating extra traffic by acknowledging every store individually. Instead, a bit in a bit vector is set whenever a remote store is issued indicating to which tile the store has been sent. If the store is later followed by a load, then the corresponding bit is cleared from the bit vector. Now, when a tile encounters an `eieio` instruction, it will check if any load buffers are allocated and if any bits in this remote store bit vector are set. If a bit is set, then the system will send out a "remote store acknowledge" message to the remote tile and allocate a load buffer for the answer. After that it will clear the corresponding bit. Thus, the system now waits for the allocated load buffer to be released.

9. Adapting PARMACS for the Proposed Architecture

The SPLASH-2 benchmarks, which will be presented in section 11.1, do not require any specific implementation of synchronisation and multi-threading primitives (such as locks and fork). Instead they utilise PARMACS [ANMB97] to describe high level constructs such as barriers, locks, thread creation and joining. The PARMACS package uses the m4 macro processor to replace the high level construct with actual code. Such code can be simple library calls that are more or less identical to the original macro (e.g.: `LOCK(l)` is replaced by `pthread_mutex_lock(&l)`) or numerous lines of assembly code. This chapter describes the issues of adapting the PARMACS macros for the proposed architecture.

9.1. CREATE

The CREATE macro is used to create a new thread running a specified procedure. The original PARMACS thread simply requires a function as an argument, and then assumes that it is implemented by some code that runs this function as a new thread. Considering that communication between different tiles is not uniform in the proposed architecture, it would not only be useful to specify which function to run as a thread, but also where to run it. However, neither do the SPLASH-2 benchmarks require any mechanism to assign a thread to a specific processor, nor do they provide any hints as to which threads should be best allocated to which processor if communication latency between processors is not uniform. The only suggestion it makes (as comments in the source code) is where to add some code that binds a thread to a processor, in order to prevent thread migration. Thus, the easiest approach would be to just assign each new thread to the next tile in sequence. Thus the thread number would be identical to the tile number. This is actually very easy, since all threads are created within a single for-loop. The final obstacle is that the mapping of tile number to tile position depends on the total number of tiles in the system. As such, the total number of tiles in the system has to be provided to the CREATE macro as well. An alternative would be to read the total number of tiles from a special purpose register. In addition to enhance compatibility with POSIX threads, support was added to pass a single word argument to the procedure that is being run as a new thread. This could be a pointer to some argument structure or a single integer value. Thus, the format of CREATE is now:

```
CREATE(<procedure>, <thread id>, <total number of tiles>, <arg ptr>)
```

For the thread creation process, it is assumes that the application binary either has some static information about the required number of tiles or performs some system call to the OS to set this information for the rest of the execution. Again, this is a RAW requirement to efficiently schedule one procedure across several tiles in order to exploit ILP. On one of these tiles, the OS starts the main program, while on the other ones a slave procedure is started. This slave procedure starts by trying to read several values from `r25`, the dynamic network register. Since no values can be read there at this moment, it just stalls.

In order to support the joining of threads (see section 9.2), a table is used to keep track of which tiles are currently running a thread and which are waiting to get a thread assigned. Each entry in this table is similar to the condition of the PAUSE primitive (see section 9.5). The condition is set when the tile is waiting for work and unset if it is running a thread.

Once the main thread decides to create a thread, it will allocate some memory for the stack of this thread. Currently 2MB are used for each thread stack, which seems to be enough. Then it will compute the tile number on which the thread is to be run, based on thread number and total number of tiles in the system. With this information, and after updating the running threads table, the main thread creates a message that is sent using the dynamic network and contains the following four words:

1. the stack pointer.
2. a pointer to its entry in the table of running threads.
3. a pointer to the procedure that is to be run.
4. a pointer to an optional argument.

Once the remote tile receives the message, it will initialise the stack pointer `r1` with the first received value. The other 3 values are received in a similar way and stored in registers that are callee saved (according to the Power PC Linux ABI). After that, a minimal initialisation procedure is called to setup thread specific data, like the global offset pointer `r2`. This procedure could not be called at the very start, since it needs a stack pointer. Once it returns, the argument pointer is copied to `r3` and control branches to the provided procedure pointer.

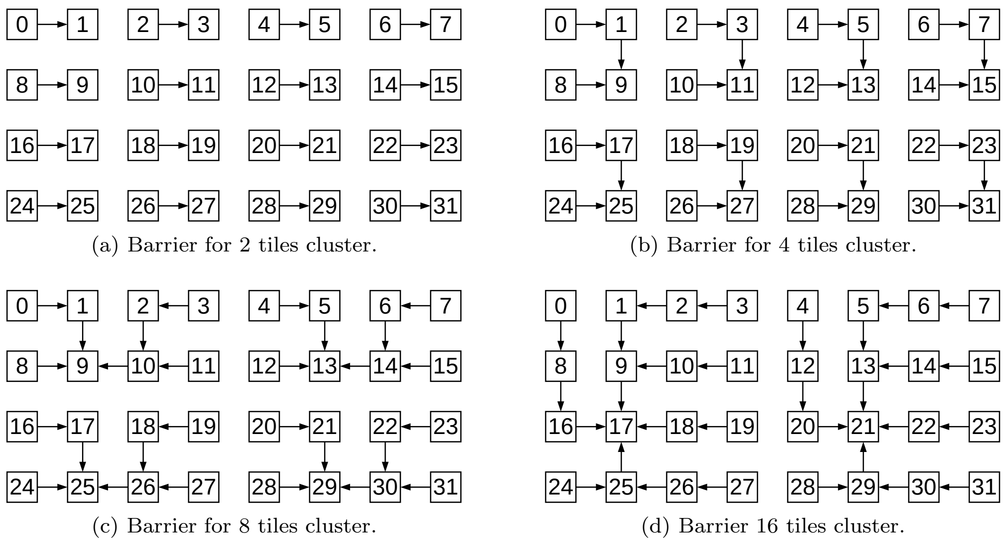
9.2. WAIT_FOR_END

Joining threads describes the process by which a thread waits for a previously created thread to terminate. Execution is then resumed in the thread that waited. If the created thread terminated before the creating thread starts to wait, then it does not wait, but just continues execution. The WAIT_FOR_END macro waits until n previously created threads have stopped execution. These threads can either have stopped execution before the macro or after. In all SPLASH-2 benchmarks, n is always the number of threads that have been created previously within the for-loop with the CREATE macro.

The macro's code checks the status of each thread in the table of running threads. If a thread has already stopped execution, it just continues with checking the next thread. Otherwise, it performs a WAITPAUSE on this entry in the thread table. Once WAITPAUSE finishes, it continues checking the remaining threads.

9.3. LOCK and UNLOCK

A lock is used to ensure that only one thread is able to access a variable that is protected by the lock. The lock implementation is almost identical to the PowerPC reference implementation



(a) Barrier for 2 tiles cluster.

(b) Barrier for 4 tiles cluster.

(c) Barrier for 8 tiles cluster.

(d) Barrier 16 tiles cluster.

Figure 9.1: Clusters of 2, 4, 8 and 16 tiles on an 8x4 tile CMP that can perform a partial barrier.

found in [fre01, Appendix E.2]. Only one modification was made to conserve network bandwidth: if the lock has already been taken by another thread, the thread does not spin on the lock with an `lwarx` instruction, but just with a normal load. Otherwise, the hardware would also generate "Reservation Cancellation" messages, since no matching `stwcx.`[1] is issued within the timeout interval.

9.4. BARRIER

A barrier signals that a certain part of the computation has been completed and threads should synchronise with each other, before starting the next part of the computation. The PARMACS macro BARRIER expects two arguments: a barrier variable and the number of threads that are expected to synchronise in the barrier. While this allows to only synchronise a subset of all running threads, the SPLASH-2 benchmarks never use this feature. Always all running threads are synchronised at a barrier. The barrier implementation takes advantage of this by utilising the static network and the blocking behaviour of the network registers.

The basic barrier macro for an all-to-all barrier follows the description in section 8.2. However, in order to allow the same application binary run on different CMP configuration the following modification was made: the static network controller is programmed to support an all-to-all barrier for the following configurations: 2 (2x1), 4 (2x2), 8 (4x2), 16 (4x4) and 32 (8x4) tiles. The value that is sent from the PE to the static network controller, does not only indicate that the PE wants to join a barrier, but also how many threads are running in total. The static network controller then executes the appropriate barrier code.

In order to implement a barrier that only synchronises a subset of threads two possible solutions exist: first, the barrier can be implemented using a traditional software only approach based on locks or the WAITPAUSE macro. Second, if the threads run in a single cluster on the tiled CMP (for example on tile 0, 1, 8 and 9), then synchronisation with the static network is again possible. This is due to the fact, that all static network controllers are programmed to

[1]The `stwcx.` instruction has a dot at the end. This is not a typo.

support different CMP configurations. Figure 9.1 shows the cluster that can perform a barrier utilising the static network.

9.5. CLEARPAUSE, SETPAUSE and WAITPAUSE

The PAUSE macros allow a thread to suspend execution until a condition is met. This condition is a boolean variable that is unset/set with the CLEARPAUSE and SETPAUSE macros. The WAITPAUSE macro is used to wait until the condition is met.

Implementing these macros is very straight forward: the condition is a boolean variable that is protected by locks to ensure that changes are visible to all threads and no conflicting operations happen simultaneously (e.g., one thread checking the condition and deciding to wait, while another thread sets the condition but does not find the first thread waiting). While most traditional implementations now either let waiting threads spin on the condition variable or suspend the thread with help of the OS, the wait can also implemented utilising the dynamic network and the blocking behaviour of the network register `r25`. A thread that finds a condition not set, and thus has to wait, adds its tile number to a list of waiting threads for this condition. Afterwards it tries to read a value from the dynamic network register `r25`. Since no value is available yet, it stalls.

The thread that sets the condition, checks the lists for waiting threads. For each thread it creates a one-word message that is sent to the tile running this thread via the dynamic network. Once the tile receives the message, it can continue its execution.

Part III.

Setup, Results and Related Work

10. Fictional Models

While it would be preferable to compare the proposed architecture to an existing architecture, this is not easily possible, since no CMP with 32 tiles that supports shared memory applications with a hardware based coherence mechanism has been built so far. Lacking this option one could try to scale down the proposed architecture to compare it with existing small scale CMPs or to scale current designs to 32 tiles using a shared bus or a crossbar. However, scaling down the proposed architecture does not address the area of interest of this research, which is large scale CMPs. Results with small scale CMPs are unlikely to answer if the proposed scheme delivers sufficient performance in a large scale CMP. As for the latter, [KZT05] uses this approach but concludes that buses do not scale beyond 16 nodes and a crossbar not beyond 8 nodes, as discussed in section 3.3. Another option would be to implement a distributed directory coherence protocol on a mesh based interconnect. However, as discussed in section 3.2, implementing and verifying such a protocol, including simulating the timing delays, would require tremendous effort that is way beyond this thesis. The last option would be to assume that there is no hardware support for cache coherence and port one of the software DSM systems discussed in chapter 5. Again, the more sophisticated software DSMs are fairly complicated and would require a significant effort to reimplement the whole system and execute it as part of the simulation on the simulator. Instead, this thesis tries to estimate the performance of the proposed system by comparing it against four fictional models. These models use certain simplifications that reduce the complexity of the implementation and give the model an advantage compared to a realistic implementation. This approach should give some insight into the expected performance degradation in the worst case.

Furthermore, two fictional models based on the proposed system have been developed to identify sources of performance loss. The following sections will describe these models in detail.

10.1. Mystery Cache

This system models a perfect cache coherent system. It is perfect in the sense that it assumes a MESI cache coherence protocol with a shared bus, however:

- There is no arbitration for the bus and it is always available to all tiles.
- Invalidation messages are effective immediately and do not need any time to reach the tiles.
- Cache-to-cache transfers only experience a delay based on the distance and bandwidth of the network. There is no limit on how many different cache lines can be transferred from the same cache at any given time. The system will always choose the tile that offers the lowest latency if multiple tiles have a copy of the same cache line.

This model has been developed to obtain an idea of how well the benchmarks scale in an ideal cache coherent system. Furthermore, it gives an upper bound of how bad the proposed system would fare against such an ideal cache coherent system. This system will be referred to as *Coh*.

10.2. Directory Cache

Directory cache models a system that uses an idealised distributed directory protocol to enforce cache coherence. The model is basically a direct implementation of this protocol from a multi-chip multiprocessor.

For the directory controller, an aggressive hardware implementation is assumed that requires only 5 cycles to process each request. This time is the minimum required to access some fast SRAM, check the current directory state, and decide on the appropriate actions. Finally, it is assumed that the home tile has one directory entry for each cache line in its local cache and, thus, cannot track lines that are not present in the home cache. This last assumption corresponds to existing distributed directory schemes (e.g., [LL97]).

While this protocol is more realistic than Mystery Cache, it still makes the following simplifications:

- Control and invalidation messages take zero time. Thus, the model does not have to deal with the transient states and multiple hop transactions that complicate the implementation of a real distributed directory protocol.
- The latency of a cache-to-cache transfer is determined by just the bandwidth and distance. Sending and receiving cache lines is not serialised.

For fairness of comparison, the directory scheme is augmented with migration of pages at barriers, which minimises the effect of hotspots due to the first-touch home-allocation policy. The cost of migration is the same as in the proposed system: 2000 cycles plus the cost of flushing the caches. This system will be referred to as *Dir-Coh*.

10.3. Software Distributed Shared Memory

This system models a page based software distributed shared memory system. As discussed in chapter 5, several techniques have been used to avoid the “ping-pong” effect and unnecessary generation of diffs. The TreadMarks [KCDZ94] software DSM system seems to be one of the most advanced systems. It allows multiple writers (to avoid the “ping-pong” effect) and supports lazy diffing (to reduce the number of diffs created). Implementing the whole system for a tiled CMP would be beyond the scope of this research. Previous work on software DSM systems showed that the costs of managing the multiple copies of pages, generating diffs, and updating pages, correspond to a significant fraction of the costs in these systems [ISL96]. Thus, instead this thesis focuses on one of these expensive aspects: the creation of diffs. In order to model this overhead accurately, it is necessary to reimplement the book keeping logic of the TreadMarks system. As for the actual memory, the model simply uses a single memory image that is not distributed. Thus, this model simulates the following aspects in detail:

- The system accounts for the creations of twins (copies of a page that will be later used to identify the differences). The creation of a twin involves reading a page completely and then writing it completely to a different memory location. This will obviously have an impact on the data in the cache and is modelled appropriately. Furthermore a trap to the OS is necessary, since a twin is only created in the event of a write access to a page that is write protected in the TLB. All cache accesses that are required to simulate the impact on the cache are issued within the same cycle.
- Diffs are only created when they are actually needed to validate a page that has been marked as invalid. The same lazy strategy as in TreadMarks is used to minimise the number of diff creations.
- Creating a diff involves reading from the modified page and the unmodified twin. Thus, the data of these two pages will be in the cache at the end of diff creation. A simple loop that compares two pages at word (4 bytes) granularity was implemented as a stand alone application and found to complete in a little more than 48,000 cycles. The following assumptions were made for timing of the loop: all data of the first page is in the cache, while all data on the second page is not in the cache. The first page corresponds to the page that is modified by the application, while the second page corresponds to the twin of that page. Since the twin is not being accessed by anyone except the twin and diff creation subroutine, this page would have been evicted from the cache (this is especially likely since diff creation is postponed as much as possible). While there is no intuitive argument for why all data of the first page should be in the cache, it will only give the software DSM system an advantage. The loop did not contain any logic to encode the encountered differences nor to identify the modified bytes within a modified word. It was optimistically assumed that on average these tasks could be done in 2,000 cycles, thus bringing the total costs of the diff operation to 50,000 cycles.
- Since this DSM is running on a CMP with a single physical memory one optimisation is made that would not be possible in a multi node system. During the validation phase at a barrier the system might detect that a page has only been written to by a single tile. So instead of creating a diff and distributing it to all other nodes, it would be enough to update the page mapping in each tile so that the modified page is seen by all threads.

However, on the other side the following aspects are not modelled:

- As mentioned before, the system accounts for the creation of twins. However, this is being limited to the content of the cache. As such the simulation will perform the required cache accesses to create the twin. However, the simulation will not consider the time it takes to create the twin. Allocating the twin page and copying the data, which might include waiting for cache misses, all happens in zero time. While, the simulator still stalls the CPU for the OS trap time, it is unlikely that all these operations can be completed in this time.
- Transferring and applying a diff happens instantly. The receiving tile only has to wait until all diffs required to validate a page are created, after that they are applied instantly.
- Applying the diff does not affect the cache in any way. On the negative side, cache lines that would be prefetched in order to apply the diff, suffer a cache miss later, when they

are really needed. On the other side, the cache is not polluted with the diff data and updated cache lines that will not be accessed.

- The TLB updates that are necessary to implement the CMP specific optimisation for a single writer are assumed to complete in zero time.

This system will be referred to as *SW-DSM*.

10.4. Big Cache

Another way to get a coherent system is to have all tiles share the same L1 cache. In this system the following assumptions have been made:

- Adding more tiles (hence more ports) to the cache does not have any impact on the access times.
- The cache is able to service as many requests simultaneously as there are tiles connected to it.
- The total size of the cache is equal to the combined cache available in a system with private caches.
- Each tile still has its own private instruction L1 and TLB.

One motivation for this model is to investigate the effect of sharing data in a system with private caches. Since there is only a single copy of every data in this system, the total amount of data would be larger than in a system with private caches. This effect might lead to a possible increase in performance. This system will be referred to as *UCA-Shared* (a shared cache with uniform cache access times).

10.5. Fast Network

The models discussed so far describe fictional systems that should give some insight into the expected performance of competitor approaches. Unlike these models, the “Fast Network” and the “Fast Synchronisation” model are simplifications of the proposed system that were designed with the intention to identify sources of performance loss of the proposed approach.

In the proposed architecture remote memory accesses result into messages that have to traverse the on-chip-network. There are several situations where these messages could be delayed due to network congestion:

- A tile might generate memory requests faster than it is able to insert these into the network. For example issuing a sequence of 4 loads (e.g., the 3D position and an attribute of a point) takes 4 cycles. However, inserting a single load message takes at least 2 cycles.
- Several tiles request data from the same tile. Since the remote cache access controller can only start processing one request at a time, the other requests have to wait.
- A third possibility for delays is in the network itself, if a request can not use a certain link because it is already in use by a different request.

In order to evaluate the impact of these delays, a system based on the proposed architecture was implemented that is not affected by the above issues. This system is identical to the complete system with migration and read-only sharing extensions. However, the following modifications were made:

- A tile can inject all waiting messages into the network in a single cycle. A tile can receive an infinite number of messages from the network within a single cycle. Thus, the bandwidth of the network interface is infinite.
- The network can transport any message (regardless of length) to its destination within a single cycle.
- The remote access controller can start processing an infinite number of remote cache accesses in a single cycle. Cache and TLB access are still delayed as before.

10.6. Fast Synchronisation

The migration and read-only sharing extensions (see sections 7.3 and 7.4) of the proposed system require that caches are flushed at a barrier and the MAP table is invalidated on a lock acquire operation in order to prevent stale entries and ensure correct program execution. These operations are potentially very expensive and reason for concern for the scalability of the proposed system. In order to evaluate the impact of these flushes and invalidations, another fictional system based on the proposed architecture was designed. This system is able to synchronise the cache and MAP table with main memory in zero time.

11. Benchmarks

This chapter presents the two benchmark suites that were used to evaluate the performance of the proposed architecture on shared memory applications.

11.1. The SPLASH-2 Benchmarks

The SPLASH-2 benchmarks[WOT$^+$95] are a collection of 4 kernels and 8 applications written for machines with a shared memory architecture. They represent typical scientific and engineering workloads. The benchmark suite is composed of the following programs (a detailed discussion can be found in [WOT$^+$95]):

cholesky is a matrix factorisation kernel that factors a sparse matrix into a lower triangular matrix and its transpose. The algorithm is similar to *lu*, but since it deals with a sparse matrix, it has a larger communication to computation ratio. Also, the algorithm does not use global synchronisation with barriers.

fft is a version of a complex 1-D fft kernel that is optimised to minimise communication. The data is organised as a matrix, which is divided into submatrices in order to partition the work. Each processor in addition to its own submatrix transposes every other processor's submatrix. The transposition occurs in a staggered fashion in order to avoid memory hotspots.

lu is another matrix factorisation kernel that factors a dense matrix into a lower and upper triangular matrix. The matrix is divided into blocks; each owned by a specific processor. Only a processor that owns a block will update it, in order to reduce communication.

radix is a kernel that implements radix sort. Each processor generates a local histogram of its sets of keys. These local histograms are then merged into a global histogram. Afterwards the keys are permutated, resulting in an all-to-all communication.

barnes simulates the interaction of bodies such as galaxies or particles in a 3D space over several timesteps. The application organises its data in octrees[1]. Most of the computation time is spent traversing this octree (one partial traversal for each particle). This makes it very difficult to find a satisfactory distribution of the data in shared memory.

fmm also simulates the interaction of bodies but in 2D space. Similar to *barnes* the data is stored in trees in such a way that makes an intelligent distribution of the data very difficult. However, unlike *barnes* this tree is only traversed twice per timestep.

[1]An octree is a tree in which each internal node can have up to eight children.

ocean simulates large-scale ocean movements, based on eddy and boundary currents. The data is organised as 4D arrays. In order to improve the communication to computation ratio, it now uses a red-black Gauss-Seidel multigrid equation solver (as opposed to successive over-relaxiation). Furthermore, a patch was incorporated into this benchmark to improve the behaviour of one critical lock [HC03].

radiosity computes the equilibrium distribution of light in a scene composed of large input polygons. It uses distributed task queues, quadtrees, list and a BSP tree for data structures. The computational pattern of this application is highly irregular and uses task stealing techniques for load balancing. Thus, a distribution of data to exploit locality is again very difficult.

raytrace renders a 3D scene using ray tracing. The data is stored in octree similar structures. However, since the ray is reflected in unpredictable ways off the objects it strikes, it is again very difficult to find a satisfactory data and workload distribution. Thus distributed task queues and task stealing are used to at least distribute the computation somewhat evenly across all processing elements.

volrend renders a 3D scene using ray casting. This application uses octrees to store information about the scene and similar techniques as mentioned before to achive load balancing. The data access pattern is irregular and input dependent, thus preventing any satisfactory distribution of data.

water-nsquared simulates a system of water molecules and evaluates forces that affect them. It uses an $O(n^2)$ algorithm to perform this computation. Each thread stores results first locally and then merges them into a shared data structure at the end of each iteration.

water-spatial solves the same problem as before, but using an $O(n)$ algorithm. The data is divided into cells using a 3D grid with each thread being responsible for certain cells. Molecules that travel between cells cause communication between processors. However, since cells can only travel to neighbouring cells, communication can also be optimised only to happen between neighbouring processors.

The SPLASH-2 benchmarks scale up to 64 processors, as has been shown in [WOT$^+$95]. Figure 11.1 shows the reported results, for reference purposes. SPLASH-2 is not written for any particular system; the suite only assumes a shared memory system that provides a release consistency or stronger memory consistency model. In order to interact with the host system's multi threading programming model it uses the PARMACS macro package [ANMB97]. This macro package provides simple functions (like create, lock, unlock, and barrier) that the macro preprocessor then replaces with the actual programming constructs used on the host system. While this approach limits access to specialised multithreading extension the host system might offer, it reduces the effort required to port SPLASH-2 to different systems. Apart from using a version of PARMACS (see chapter 9) for the proposed architecture and the mentioned change in *ocean*, no changes to the benchmarks were made in order to make them match better the proposed architecture.

Table 11.1 shows the characteristics of each benchmark. Note that the number of instructions includes the initialisation. While the number of instructions does give an indication of the complexity of the benchmark, it hides the fact that the parallel part of *raytrace* and *volrend*

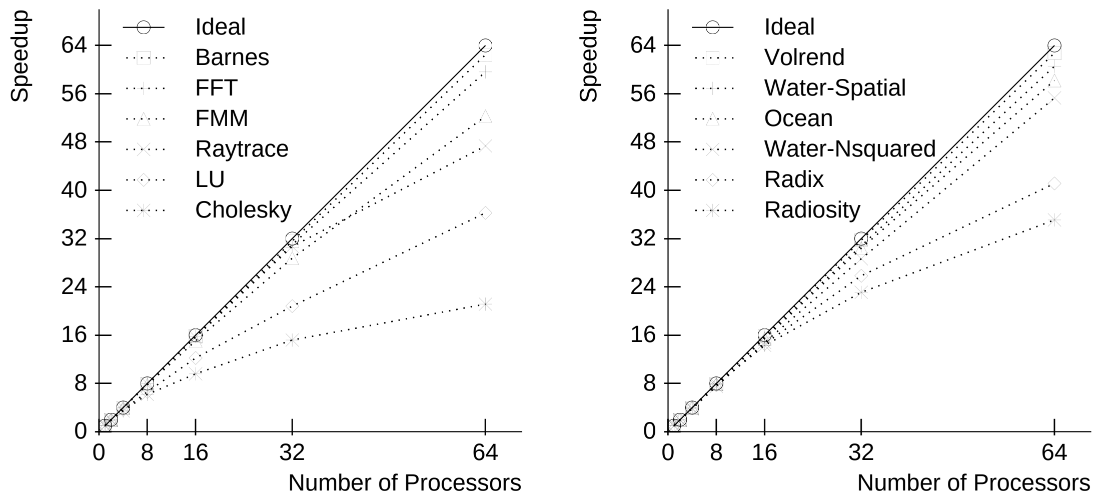


Figure 11.1: Scalability of the SPLASH-2 benchmarks as reported in [WOT+95]

.

	Benchmark	*Input*	*Instr.*	*Pages*	*Locks*		*Barriers*	
Kernels	cholesky	tk29.O	1,234M	11,968	72,075	(19K)	3	(14,558K)
	FFT	65,536 points	58M	975	32	(843K)	7	(120K)
	LU (cont. part.)	512x512 matrix 16x16 block	389M	622	32	(10,923K)	67	(163K)
	radix	262,144 keys	54M	1,057	406	(145K)	12	(153K)
Applications	barnes	16,384 particles	4,361M	924	69,360	(63K)	18	(7,564K)
	fmm	16,384 particles	2,903M	7,470	47,074	(62K)	34	(2,663K)
	ocean (cont. part.)	258x258 grid	412M	4,037	6,656	(67K)	900	(15K)
	radiosity	demo	646M	7,273	281,217	(5K)	19	(1,876K)
	raytrace	car	2,006M	9,291	95,528	(7K)	2	(10,562K)
	volrend	head	1,344M	5,648	38,604	(32K)	20	(1,946K)
	water-nsquared	512 molecules	652M	666	35,360	(18K)	19	(1,063K)
	water-spatial	512 molecules	664M	178	609	(1,067K)	19	(1,069K)

Table 11.1: Characteristics of the SPLASH-2 benchmarks. The number of instructions refers to the total number of instructions for a sequential execution of the benchmark. The number of pages is the number of pages needed for execution on 32 tiles. The number of locks refers to those encountered by all 32 tiles within the application (not library) code. The numbers in parenthesis refer to the average number of instructions per barrier/lock on a 32 tile system.

are about the same size. Similarly, the number of pages includes pages that might only be used during initialisation.

Each benchmark is run until completion with the base input set; the only exception is *radiosity* which is run with a reduced input set in order to keep simulation times manageable.

11.2. The ALPBench Benchmark

The ALPBench benchmark suite has been developed as a set of media applications that try to extract "All Levels of Parallelism" [LSA+05]. These levels are extracted by using both vector instructions and multiple threads. Since the processing element in each tile does not have a vector processing unit (like Altivec or SSE), this thesis only focuses on the multithreading aspects of the benchmarks. Unlike the SPLASH-2 benchmarks, the ALPBench benchmarks are only designed to take advantage of at most 16 threads. While this is somewhat unfortunate, since it does not allow the evaluation of a larger 32 core configuration of the proposed architecture, it does give some insight into the performance of the proposed system for a different class of applications.

The individual benchmarks are (a detailed discussion can be found in [LSA$^+$05]):

facerec tries to find which face in a database matches most closely a given input face. The matching is performed by computing a single number that expresses how similar two faces are. This number is called the distance between two faces and the overall goal is to find the faces with the shortest distance to the given input face. Parallelising this task is trivially done by assigning subsets of the database to each threads. Each thread computes the distance to the input face to each face in its subset and selects the face with the smallest distance. After this parallel phase, the main thread compares the minimum distance computed by each thread and selects the global minimum.

mpegdec is an MPEG-2 decoder. For MPEG-2 compression, each frame is divided into 16x16 pixel macroblocks. Contiguous rows of these macro blocks are called a slice. These macroblocks are then encoded independently. The main thread identifies a slice (contiguous rows of blocks) in the input stream and assigns it to another thread for decoding. The problem here is that the input stream is also variable length encoded. Thus, the main thread has to at least partly decode the input stream, in order to identify slices. This results in a staggered assignment of slices to threads and limits the scalability of extracting parallelism.

mpegenc is an MPEG-2 encoder. The encoder uses in principle the same data structures as the decoder. The encoding process is parallelised by assigning different slices to each thread. However, since these slices can be determined very easily in uncompressed data, the encoding process can be parallelised without much effort by assigning different slices to different threads.

raytrace is another ray tracer. This benchmark was not used, since the SPLASH-2 benchmark suite already includes a ray tracing application.

sphinx3 is a speech recognition program. The benchmark uses a hardware lock (a lock implemented with C library code, that uses atomic instructions) to protect a software lock (a variable that is set with non-atomic instructions). This results in a deadlock situation (both on the proposed architecture and a 2 processor 2-way SMT x86 machine), in which one thread holds the hardware lock and another the software lock. Unfortunately, this problem could not be solved in the available time.

The ALPBench benchmark suite, as mentioned earlier, scales up to 16 processors. Figure 11.2 shows the reported results, for reference purposes. The suite assumes that a pthread library [IEE95] is available in order to take advantage of thread level parallelism. While there is no reason why pthreads cannot be ported to the proposed architecture, a more pragmatic approach was followed to execute the ALPBench benchmarks on the proposed system. Since ALPBench only uses the most basic pthreads constructs, these constructs could also be expressed with the PARMACS macro package [ANMB97]. Thus, instead of writing a pthread implementation for the proposed system, the pthread library calls were replace by their PARMACS equivalent.

Benchmark	*Input*	*Instr.*	*Pages*	*Locks*		*Barriers*	
facerec	ALP Training	2,826M	4,167	30	(321K)	3	(200K)
mpegdec	525_tens_040.m2v	1,049M	456	29	(30,857K)	41	(1,364K)
mpegenc	Output of mpegdec	9,477M	1,339	29	(321,384K)	40	(14,563K)

Table 11.2: Characteristics of the ALPBench benchmarks. The number of instructions refers to the total number of instructions for a sequential execution of the benchmark. The number of pages is the number of pages needed for execution on 16 tiles. The number of locks refers to those encountered by all 16 tiles within the application (not library) code. The numbers in parentheses refer to the average number of instructions per lock/barrier on a 16 tile system.

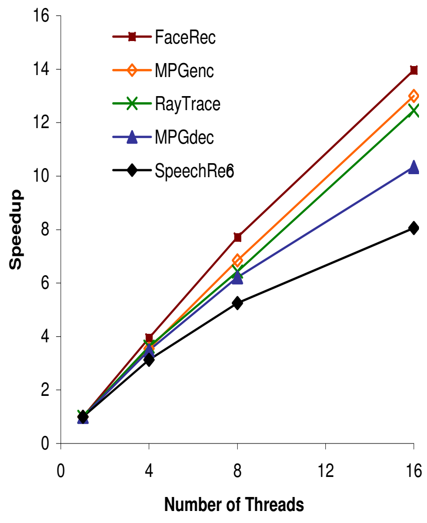


Figure 11.2: Scalability of the ALPBench benchmarks as reported in [LSA+05]
.

Table 11.2 shows the characteristics of each benchmark. Each benchmark is run until completion with the base input set.

12. Simulation Setup

12.1. Simulator

The Liberty Simulation Environment (LSE) [VVA04] was used to implement a cycle accurate simulator. A tile consists of a PowerPC core, a network controller, a data cache module, and a private instruction cache. The single-issue CPU is implemented as an 8-stage pipeline and is simulated in detail. The cache has been implemented with the cache module from SimpleScalar [BAB96]. A more detailed discussion of the Liberty Simulation Environment and the simulator can be found in appendix A and B, respectively.

12.2. System Parameters

The benchmarks were compiled with gcc 3.4.4 and glibc 2.3.5 for PowerPC. The compiler and the libraries were modified such that they do not use the special network registers for normal computation. In addition, the synchronisation primitives were replaced with versions that have been adapted to the proposed architecture.

All benchmarks were run until completion. Since many of the benchmarks have a rather long initialisation phase, a fast-forward mechanism is employed. While the simulator is in fast-forward mode, it will not simulate the network and only executes instructions on one tile. Furthermore, all instructions and system calls complete within one cycle. However, effects on the memory system are still simulated in detail. The fast-forward mode is triggered by special system calls that have been inserted into the benchmarks.

This thesis assumes that the single issue CPU runs at 2GHz; the resulting memory access time is listed in table 12.1. System calls and interrupts to the OS are assumed to take 2000 cycles. Note that PowerPC uses a hashed page table that can be read by a fast hardware handler. The 200 cycles listed in the table are the 200 cycles required by this handler. Only if this handler misses (because there is no entry for that page), then an interrupt to the OS occurs.

Unfortunately, this system configuration was not the one that was used at beginning of this research. A lower clock cycle was assumed and thus less latency to access off-chip memory.

L1 D-cache size	32K
L1 hit latency	3 cycles
L1 miss latency	200+16 cycles
L1 line size	32 bytes
L1 associativity	4-way
L1 writeback buffers	8

TLB/MAP entries	128
TLB/MAP page size	4K
TLB/MAP associativity	4-way
TLB/MAP hit latency	1 cycle
TLB/MAP miss latency	200 cycles
RAC input queue entries	32

Remote cache access latency without any congestion: $2 * (h + w) + t + 1$, where h is the number of hops, w is the number of words in the message (2 or 3), and t is the access time at the remote cache.

Table 12.1: Memory system configuration. For an L1 miss, it takes 200 cycles for the first word to arrive and another 16 cycles to fill the remaining words in the cache line.

	SLOWMEM	FASTMEM
L1 miss latency	200+16 cycles	80+16 cycles
TLB/MAP miss latency	200 cycles	80 cycles
Trap to OS	2000 cycles	1000 cycles

Table 12.2: Differences between the SLOWMEM and FASTMEM memory configuration.

Since simulation time was a serious constraint during this research, it was not possible to rerun all experiments with a single consistent memory configuration. Thus, the results for some of the experiments reported were obtained with this earlier configuration. While the results obtained with this earlier configuration cannot be directly compared against results of the final configuration, they still allow some additional insight into the impact of various extensions that were developed. The memory configuration of this earlier system will be referred to as FASTMEM. The configuration of the final system will be referred to as SLOWMEM. Table 12.2 lists the differences between the two configurations.

12.3. Systems Evaluated

Throughout the evaluation the systems are referred to as: *NUCA-Dist* (*n*on-*u*niform *c*ache *a*ccess with *dist*ributed cache), for a system that implements the proposed architecture, but without the page migration and sharing mechanisms described in sections 7.3 and 7.4. *NUCA-Dist+M* (*M* stands for *m*igration), for a system that implements the proposed architecture and page migration mechanism. *NUCA-Dist+MS* (*S* stands for *s*haring), for a system that implements the proposed architecture with page migration and read-only sharing extension. Only the FASTMEM configuration was used for *NUCA-Dist* and *NUCA-Dist+M* systems. Results for the *NUCA-Dist+MS* system were obtained using both the FASTMEM and SLOWMEM configurations.

For the other systems, the following design decisions were made:

Directory Coherent System This system uses the same configuration as the proposed system (with the exception of not having a MAP table). For the directory controller, this thesis assumes an aggressive hardware implementation that requires only 5 cycles to process each request. This time is the minimum required to access some fast SRAM, check the current directory state, and decide on the appropriate actions. During initial tests with this system it was found that the first touch policy used by the directory system to assign home nodes, results in the creation of hotspots that limit the performance of the directory coherent system in an unexpected way. The measured speedups did not match the reported speedups in [WOT$^+$95] even closely. Thus, the directory scheme was augmented with migration of pages at barriers, which minimises any negative effects from the first-touch home-allocation. The cost of migration is the same as in the proposed system: 2000 cycles[1] plus the cost of flushing the caches. This system will be referred to as *Dir-Coh*.

Software DSM System This system uses the same configuration as the proposed system (with the exception of not having a MAP table). As discussed in section 10.3, the system assumes

[1]Migration requires that the mapping between physical address and the tile that manages the diretory is being reset. Such an operation is most likely not available from user level code, but requires a system call. Since the general assumption for system calls is 2000 cycles, the same 2000 cycles are assumed here.

that it takes 50,000 cycles to create a single diff. As for the trap, whenever a tile tries to write to a write-protected page, the same 2000 cycle latency is assumed, as it is in the proposed system. This system will be referred to as *SW-DSM*.

2-Level Cache Hierarchy Of the several options presented in section 7.6 how to add a second level cache to the system, the option that uses the second level cache simply as a backing store for the first level cache was chosen (see figure 7.9a). Thus, a 128KBytes second level cache was added to every tile. The size of the first level cache remained the same, however a write-through policy is assumed. The total L2 capacity on a 32 tile chip is 4MBytes, which is in line with current CMP offerings. The relatively small capacity per tile is in line with what could be expected from a CMP with 32 tiles. Each L2 has a 20 cycles access time.

The directory cache coherent system *Dir-Coh* also uses a private 128KB second level cache that is connected to the write-through first level cache. The second level cache is now the domain of cache coherence. The proposed system with the second level cache will be referred to as *NUCA2-Dist+MS*, and the directory cache coherent system with second level cache as *Dir-Coh2*.

Mystery Coherent System This system assumes a cache coherence mechanism that is able to keep all caches coherent in virtually no time (as discussed in section 10.1). The system uses the same configuration as the proposed system (with the exception of not having a MAP table). This system will be referred to as *Coh*.

Singe Shared Cache System This system follows the design discussed in section 10.4. The system uses the same configuration as the proposed system (with the exception of not having a MAP table and the L1 cache size). The L1 cache size is the same as the combined size of all L1 caches in the proposed system. For example, a system with 8 tiles will have a single L1 cache of size 256KBytes; a system with 32 tiles will have a single L1 cache of size 1MBytes. Apart from the size, all other cache parameters (including the access time) remain the same as in the proposed system. This system will be referred to as *UCA-Shared* (*u*niform *c*ache *a*ccess with *shared* cache).

13. Performance Results

13.1. Overall Performance

The discussion of the performance of the proposed system starts by looking at the achieved speedup, as shown in figure 13.1 for the *Coh*, *Dir-Coh*, *NUCA-Dist+MS* and *SW-DSM* system. The speedups were measured on a 32 tile system for the SPLASH-2 benchmarks and on a 16 tile system for the ALPBench benchmarks. It comes without surprise that the almost perfect system *Coh* performs best across all benchmarks. On average it achieves an efficiency (speedup divided by number of processors) of 92%. The most notable exceptions are *radiosity* and *mpegdec*, which show efficiencies around 50%, and to a lesser degree *lu* and *radix*. On the other hand, it is able to achieve super-linear speedup for 8 SPLASH-2 benchmarks. In particular, its performance for *ocean* is impressive (almost a 40x speedup). The reason for this is its ability to increase the locally available cache size by migrating cache lines between tiles at almost no cost compared to an off-chip memory access. The reason for including these results is to get an upper bound for the potential speedup available.

Looking at a more realistic system, the performance of the idealised distributed directory coherent system, *Dir-Coh*, comes close to the speedups of *Coh*. Notable exceptions are *fft* (19% gap), *radix* (36% gap), *ocean* (42% gap) and raytrace (21% gap). On average the performance gap and efficiency are 11% and 81%, respectively, for all benchmarks. These results are somewhat better than those found in a real distributed directory machine [LL97] mainly due to the lower communication latencies observed in a single chip and simplifications made in the protocol.

Looking at the performance of the proposed scheme (*NUCA-Dist+MS*) one can see that it performs fairly close to the hardware directory coherence system, with a performance gap ranging from 0% (no gap) to 32% (for *radiosity*), and 16% on average. Moreover, the performance gap is less than 10% for 6 out of 15 benchmarks, which is an impressive result considering that the directory coherence system uses a very aggressive hardware implementation and that the proposed architecture requires only simple hardware support.

13.2. Memory Access Breakdown

To better understand the behaviour of the proposed architecture, the outcome of each processor memory request is tracked for each system. Figure 13.2 shows the breakdown of memory requests for each benchmark and for configurations with 32 (SPLASH-2) and 16 (ALPBench) processors. For each benchmark and configuration, the bar is normalised to the total number of processor memory requests, which does not vary noticeably across the different systems. For *SW-DSM*, these numbers do not include requests that were issued by the software DSM system to manipulate pages. Since for most manipulations the software DSM is performing a linear access to every word in a page, the access pattern is roughly as follows: the first access will

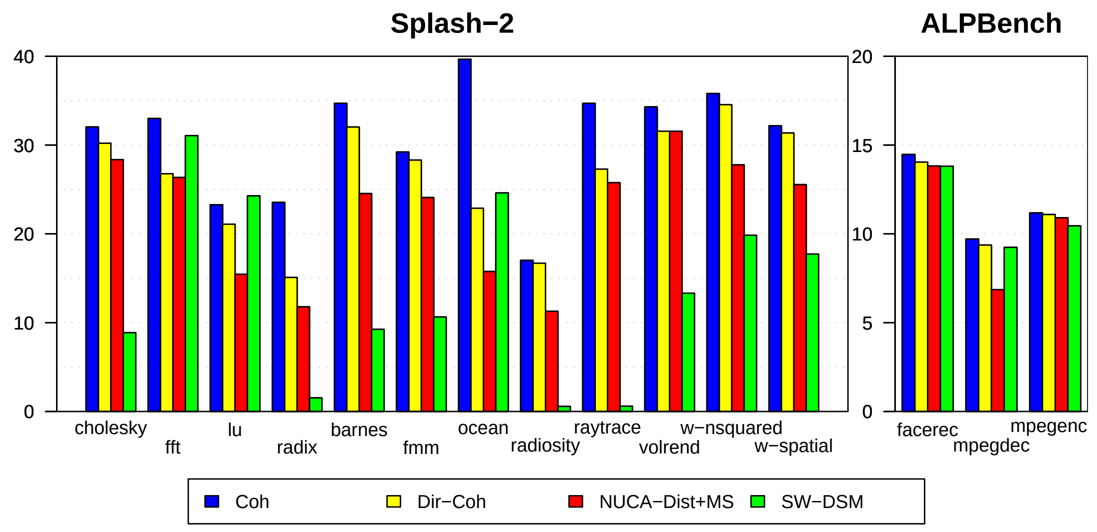


Figure 13.1: Speedups for *Coh*, *Dir-Coh*, *NUCA-Dist+MS* and *SW-DSM*. The y-axis shows the speedup compared to execution time (after initialisation) of a single tile.

be either a miss or a hit, while the next 7 accesses are guaranteed to be hits. This would significantly affect the overall cache access distribution, while at the same time these accesses are simulated to take zero time. Thus, since these accesses do not affect run time directly (they still affect the run time indirectly by replacing data in the cache, resulting either in a prefetch effect or a refetch), they also should not be included in the distribution.

The bars are broken down into the following components: accesses that hit in the local L1 cache (*local hits*); accesses that hit in a remote L1 cache (*remote hits*); accesses that go off-chip following a miss in the local L1 cache (*local miss*); and accesses that go off-chip following a miss in a remote L1 cache (*remote miss*).

The figure shows that the fraction of off-chip accesses is fairly small in most cases, with the exception being *ocean*, where the off-chip accesses for all systems (including the single-tile configuration) account for about 12% of all requests. Another exception is *facerec*, where sequential execution shows only a small number of off-chip accesses, but the parallel systems show a large fraction (22%) of off-chip accesses. Given this generally small number of off-chip accesses, the ratio of local to remote cache accesses is to be expected to be the main differentiating factor.

The results for *NUCA-Dist+MS* show that the fraction of remote cache accesses is fairly small for most benchmarks except *cholesky* (39%) and *mpegdec* (38%), and, to a lesser extent, *lu* (15%) and *radiosity* (18%). Such relative small number of remote cache accesses mostly explains the reasonably good performance of the proposed architecture for many benchmarks. An interesting case is *cholesky* where the fraction of remote cache accesses is high compared to most benchmarks, but its performance with *NUCA-Dist+MS* is good. On the other hand, some benchmarks, such as *ocean* and *barnes* show small fraction of remote cache accesses (0.8% and 0.8%, respectively), but their performance with *NUCA-Dist+MS* is not as good as with some of the other benchmarks. In the case of *barnes* this might be explained by a higher number of off-chip accesses (1.4% for *NUCA-Dist+MS* compared with 0.3% for *Dir-Coh*); for *ocean* on the other hand the off-chip accesses are roughly identical (12.1% for *Dir-Coh* and 11.9% for *NUCA-Dist+MS*). These somewhat unintuitive cases will be discussed in more detail later (for *cholesky* see section 13.9, and for *ocean* see section 13.7). The results for *Dir-Coh*, on the other hand, show that it incurs very few remote accesses (i.e., cache-to-cache transfers), which mainly explains its very good performance. The results for *SW-DSM* will be discussed in section 13.4.

The impact of local, remote, and off-chip accesses can be further seen in figure 13.3, which shows the average load latencies, in cycles, for the different types of loads for *NUCA-Dist+MS* and for the average load for *Dir-Coh*. While the latencies for remote loads in *NUCA-Dist+MS* are significantly larger than those of local loads, the average latencies are fairly close to the local ones and, thus, very close of those in *Dir-Coh*.

While figures 13.2 and 13.3 can be used to explain the behaviour for most benchmarks (that includes all benchmark that haven't been identified as a somewhat unintuitive case in the previous paragraph), it fails to explain the problem with *ocean*. *Ocean* almost does not experience any remote accesses and the average load latency (30.8 cycles) is similar to the load latency for the directory coherent system *Dir-Coh* (31.1 cycles). As section 13.7 will show, part of the degradation is due to the overhead introduced at barriers. Still, this overhead is not enough to explain the whole performance degradation experienced. The problem with the observation so far is that they treat at all memory operations equally. Unfortunately, not all memory operations are equally important. For example, memory operations that are used as part of synchronisation primitives have a much larger potential of slowing down the whole application than other operations. Section 13.3 takes a more detailed look at those memory operations. Another problematic benchmark is either *cholesky* or *lu*. Even though *cholesky* shows a larger fraction of remote accesses than *lu* and experiences a worse average memory latency gap than *lu*, it seems to perform very close to *Dir-Coh*. It turns out that the unintuitive one is indeed *cholesky* and this issue will be discussed in section 13.9.

13.3. Latency of Lock Acquire Operations

As mentioned before, not all memory operations are equally important. Memory operations that are part of synchronisation events are usually on the critical path of execution and, thus their latency cannot be hidden by performing independent operations. Figure 13.4 shows the average time in cycles it takes between the first request for a lock and the time when the lock has been granted. The number of cycles does not include any additional delay that might be caused by invalidating the local MAP for *NUCA-Dist+MS*.

The reason for the overall poor performance is that locks are still implemented using spinning techniques. Since accessing the lock variable will require a remote access (in most cases), latencies for accessing it will be increased. This problem gets worse if several locks are mapped to the same tile. Not only will several independent requests compete for the bandwidth to this tile, but also the tile will only be able to make forward progress on managing one lock while all other lock attempts are stalled (they still receive answers for their LL requests, but will not succeed with a SC).

The large time spent on synchronisation is particularly damaging for *ocean* since it is one of the shorter running benchmarks, with little opportunity to amortise the lost time. For example, for *fmm* the proposed architecture also takes about 4 times longer to acquire a lock than *Dir-Coh*. However, first the absolute time to acquire a lock is one order of magnitude less for *fmm*, and second, *fmm* is the third longest running benchmark (after *mpegenc* and *barnes*) with plenty of time to amortise.

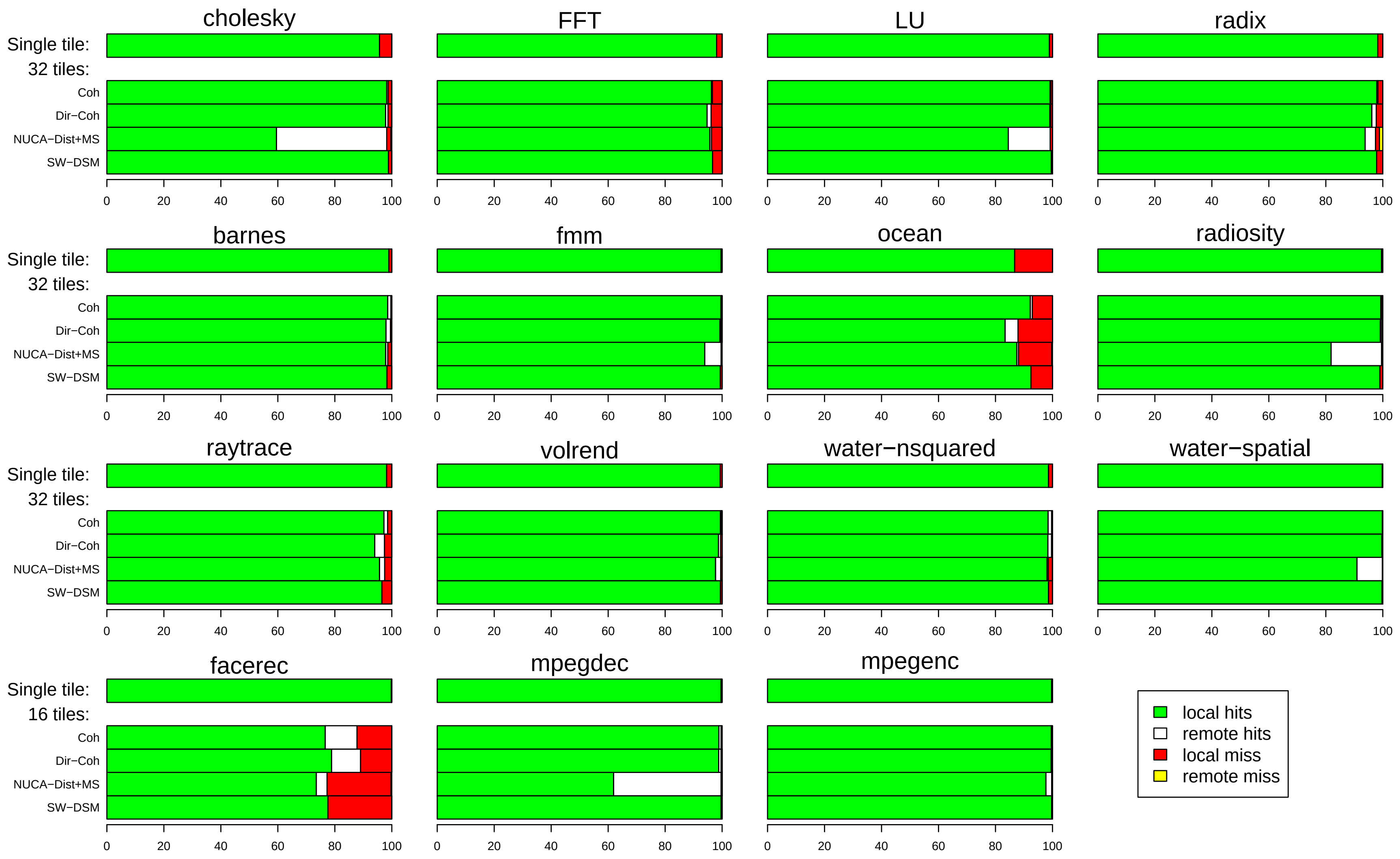


Figure 13.2: Distribution of memory accesses into cache hits and misses for *Coh*, *Dir-Coh*, *NUCA-Dist+MS* and *SW-DSM*. They are then further divided into local and remote accesses for *NUCA-Dist+MS*. For *Coh* and *Dir-Coh*, local hits indicate cache-to-cache transfers. There are no remote misses for *Coh* and *Dir-Coh*. The plot also shows for reference purposes cache hits and misses for a single tile system.

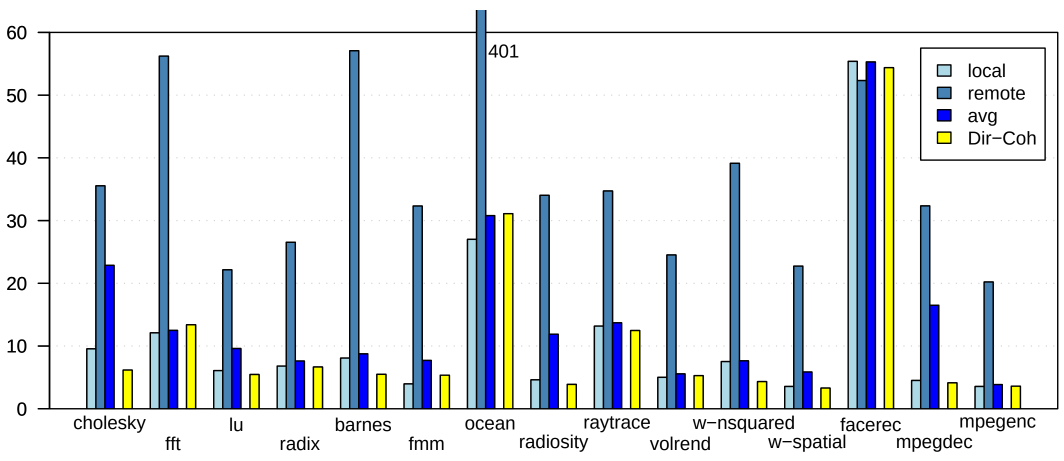


Figure 13.3: Average latencies for local, remote and all loads for *NUCA-Dist+MS*. The average latencies for *Dir-Coh* are shown for reference. The y-axis shows the latency in processor cycles.

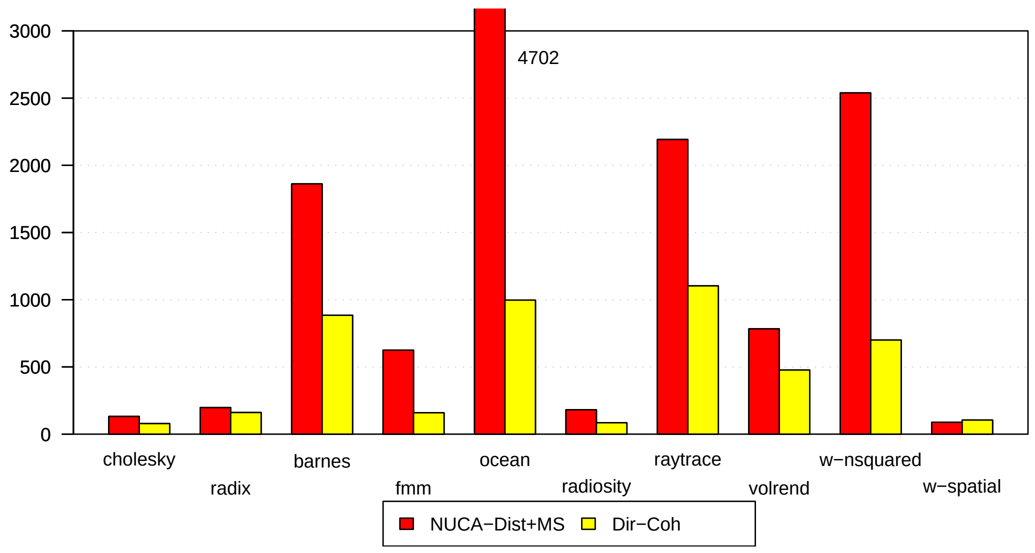


Figure 13.4: Average number of cycles it takes before a lock attempt is granted for *NUCA-Dist+MS* and *Dir-Coh* in a 32 node system. The y-axis shows the latency in processor cycles. Only benchmarks with a significant number of locks were evaluated.

Kuhn [Kuh07] investigates other lock implementations on the proposed architecture and finds that a queue based implementation that also suspends threads by making them spin on a network register is able to achieve far better performance than a simple spin lock based implementation. Unfortunately, he only investigates the impact on small synthetic benchmarks. Still, even using these benchmarks, the potential to reduce network traffic and, thus, reduce runtime is significant. The queue lock has only minimal overhead for a low frequency of locks. However, once the lock frequency increases, the queue lock is able to reduce the runtime to less than 25% of the runtime of the spin lock version.

13.4. Software DSM Results

Figure 13.1 shows the speedup obtained with the simplified software DSM implementation (*SW-DSM*). The numbers are somewhat in line with the speedups reported in [ISL96] (i.e.:

Benchmark	*Speedup*	*Twins*	*Diffs*	*Ratio*
cholesky	8.88	99,230	88,784	89.47
fft	31.06	1,285	154	11.98
lu	24.28	9,825	88	0.90
radix	1.53	17,803	16,488	92.61
barnes	9.25	99,285	70,147	70.65
fmm	10.64	170,118	109,350	64.28
ocean	24.61	128,928	2,317	1.80
radiosity	0.57	131,504	109,462	83.24

Benchmark	*Speedup*	*Twins*	*Diffs*	*Ratio*
raytrace	0.60	99,563	98,235	98.67
volrend	13.32	11,548	9,869	85.46
water-nsq	19.85	10,557	3,573	33.84
water-spa	17.72	5,393	1,551	28.76
facerec	13.81	284	84	29.58
mpegdec	9.14	6,055	3,434	56.71
mpegenc	10.45	18,652	5,832	31.27

Table 13.1: Speedup and diff creation statistics for the software DSM system. The right-most column shows the ratio between created twins and diffs in percent. Results are shown for a 32 tile system in case of the SPLASH-2 benchmarks and for a 16 tile system in case of the ALPBench benchmarks.

Applications that perform well in [ISL96] also perform well on *SW-DSM* and vice versa). However, the simplified system *SW-DSM* obtains in most cases a higher speedup, which comes at no surprise, considering that several operations came for free in this thesis' implementation. As can be seen, the proposed system *NUCA-Dist+MS* is able to outperform the software DSM system in most cases. However, one of the most striking results is that for *fft*, *lu* and *ocean* the software DSM system performs better than even *Dir-Coh*. While the superior performance can be easily dismissed as an artefact that happens because only the diff creation overhead of software DSMs is simulated, it also happens that these three benchmarks have something else in common: they are mostly controlled by barriers instead of locks. It seems that using barriers results in sharing at a coarse enough granularity that only a few number of diffs have to be created. This is confirmed by table 13.1, which lists the number of twins and diffs created. *Fft*, *lu* and *ocean* show the lowest ratio of created diffs compared to any other benchmark. A further reason for *SW-DSM*'s better performance on these benchmarks compared to *Dir-Coh* is that *SW-DSM* does not have to write back dirty cache lines when it arrives at a barrier (as *Dir-Coh* and *NUCA-Dist+MS* do). Strictly speaking, *SW-DSM* has to do the same as part of diff creation process at barriers. However, as pointed out in section 10.3, the creation of diffs is simplified. Modified cache lines will be evicted (thus written back) from the cache if at least 4 diffs are created. However, *SW-DSM* only models the cache content during this phase and not the latencies required to fetch or write back data. Furthermore, if 4 or less diffs are created then some modified cache lines might not be written back at all. This situation is extremely likely to happen if a large percentage of pages are only modified by a single tile and, thus, no diff at all is created for these pages. As the diff to twin ratio in table 13.1 suggests this seems to be the case for *fft*, *lu* and *ocean*.

A final advantage that *SW-DSM* has over the proposed scheme *NUCA-Dist+MS* is that data is always locally shared. Even after diffs had to be created in order to update a page, that page can be then cached locally by any node. In comparison, if a page has been declared owned by a tile in the proposed scheme *NUCA-Dist+MS*, then it will never be read-only shared again, unless all tiles synchronise at a barrier.

Still, it seems surprising that *SW-DSM* does not suffer similar performance degradation for ALPBench. Despite having similar high ratio between twins and diffs created, all ALPBench benchmarks show almost the same performance as *Dir-Coh*. A possible explanation might be that ALPBench also uses mainly barriers for synchronisation. As discussed earlier, *SW-DSM* has an advantage at barriers compared to the other schemes.

Another interesting observation is shown in figure 13.2. As discussed in section 10.3, *SW-DSM* does model the effect of polluting the cache during twin and diff creation. As such, it would be interesting to know if these cache accesses would result in having a prefetching effect or cause more cache misses. The recorded numbers for cache accesses only show accesses made by the application and not by the software DSM system to create twins and diffs. These numbers show that there is no increase in cache misses. In several cases *SW-DSM* has significantly fewer misses than *Dir-Coh* and *NUCA-Dist+MS*. Thus, overall, it seems that *SW-DSM* benefits from a prefetching effect. Unfortunately, it is impossible to say how realistic it is for a real software DSM system to benefit similarly. This is due to the fact, that while *SW-DSM* simulates cache pollution, it does not simulate the precise access times. For example during the creation of a twin or a diff *SW-DSM* might miss several times in the cache, being forced to wait until the data arrives. However, the algorithm that simulates the cache pollution effect issues all requests in the same cycle, thus all misses will be resolved at the same cycle in the future.

On average *NUCA-Dist+MS* performs 28% faster than *SW-DSM*, with the gap ranging from -57% (i.e., *SW-DSM* is faster) and 98%.

Apart from 3 benchmarks (which performance is not to believed since the cost of a barrier is zero in this simplified implementation) the performance of *SW-DSM* is worse than any other system, even though the costs used in the simulation are much lower than to be expected in a realistic implementation. The overall result of these experiments is that the coherence problem within a tiled CMP cannot just be solved by using a software DSM system. Instead it seems that additional hardware support is needed to offer sufficient performance across a wider range of applications.

13.5. Impact of the Migration, Read-Only Sharing and Selective Invalidation

As discussed in sections 7.3 and 7.4, some extensions were added to the base system in order to improve performance. This section discusses in detail the impact of each extension. The analysis starts with looking at the performance of the base system (*NUCA-Dist*). Figure 13.5 shows the performance of this model. The system performs well for all SPLASH-2 benchmarks[1] for up to 4 tiles. However, doubling the number of tiles to 8 results in almost no performance increase for a number of benchmarks (*cholesky*, *lu* and *radix*). This becomes even more obvious, when the number of tiles is doubled again to 16. Now also *barnes*, *radiosity*, *raytrace*, *volrend*, *water-nsquared* and *water-spatial* do not show any further performance gains. Only *fmm* and *ocean* show some signs of scalability.

This behaviour can be explained by looking at the memory access distribution and latencies of remote accesses shown in figures 13.6 and 13.7, respectively. Similar to figure 13.2, figure 13.6 shows the outcome of each memory access. However, unlike figure 13.2 it does so not only for the configuration with the maximum number of supported tiles, but also 2, 4, 8 and 16 tiles. The first thing to notice is that with an increase in the number of tiles the percentage of remote accesses increases. This does not come as a surprise, since shared memory programs usually

[1] Results for the ALPBench benchmarks are not included in this study, since these results were obtained before the ALPBench benchmarks were adapted to run on the system. This study was not repeated with the ALPBench benchmark, since this work did not expect to find any new insights and simulation time is rather long.

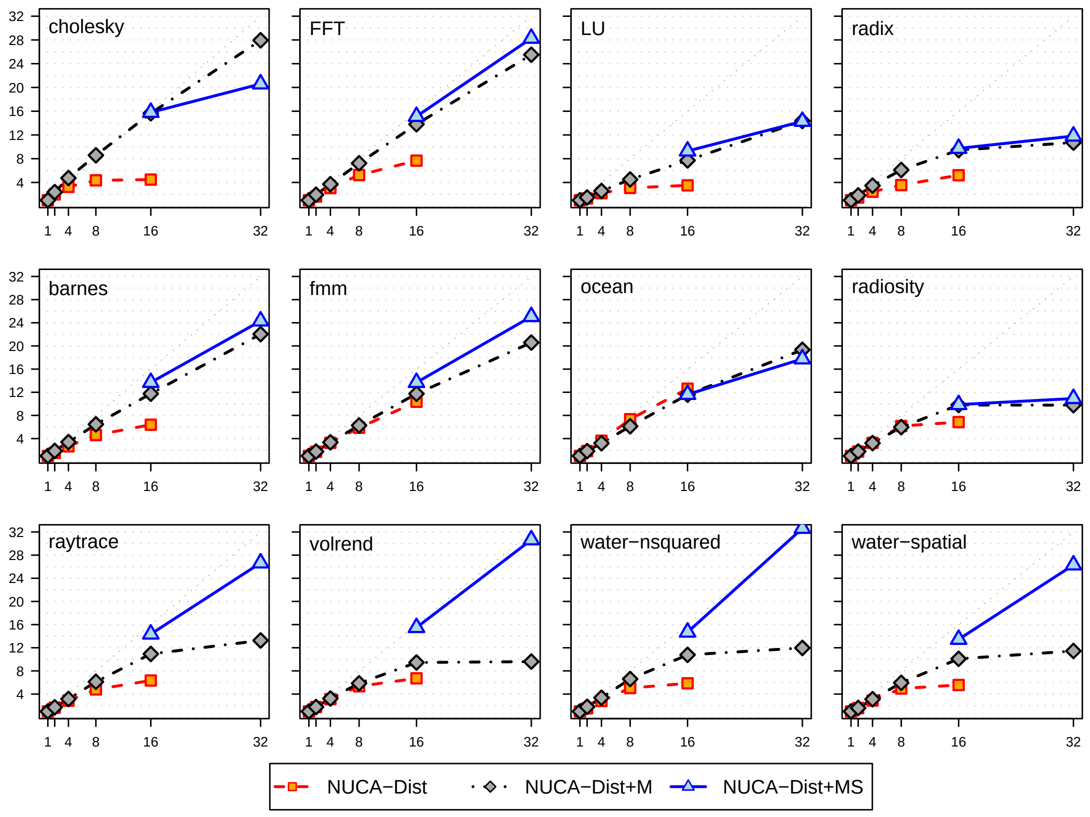


Figure 13.5: Speedups with respect to execution time of a single tile for *NUCA-Dist*, *NUCA-Dist+M* and *NUCA-Dist+MS* for 2 to 32 tiles. The x-axis shows the number of tiles. The y-axis shows the speedup.

use a certain memory area for shared data. The size of this area mostly depends on the size of the problem, and not on the number of tiles. Each tile also uses a certain amount of private data, however, this amount depends on the number of tiles (each tile usually gets a smaller chunk of the work if the number of tiles increases). In the baseline model *NUCA-Dist* due to the first-touch policy, a remote access is required for all tiles to access the shared data (with the exception of the tile that initialised this data). Thus, the percentage of remote accesses increases. *Lu* demonstrates this principle to the extreme. The benchmark only uses shared data, thus, for two tiles one tile can work on local data, the other one has to use remote accesses all the time. Thus, in total about 50% of all accesses are remote. Increasing the number of tiles to 4, means that once again one tile can access all data locally, while the remaining three have to perform remote accesses. This results in the observed 75% remote accesses.

Regardless of whether the remote tile can keep up or not, remote accesses cost more and will be a problem; the bottleneck at the remote node only exacerbates it. Unfortunately, this often happens, as shown in figure 13.7. The latency of remote access (figure 13.7a) starts to increase significantly at a certain point for most benchmarks. At the same point also the average load latency (figure 13.7b) for most benchmarks starts to increase. This point corresponds to the same point when the benchmark stops scaling. The only exception to this is *ocean*, which shows a drastic increase in its remote access latency, while still scaling acceptably. The reason is that *ocean* manages to convert some memory accesses that would otherwise be cache misses

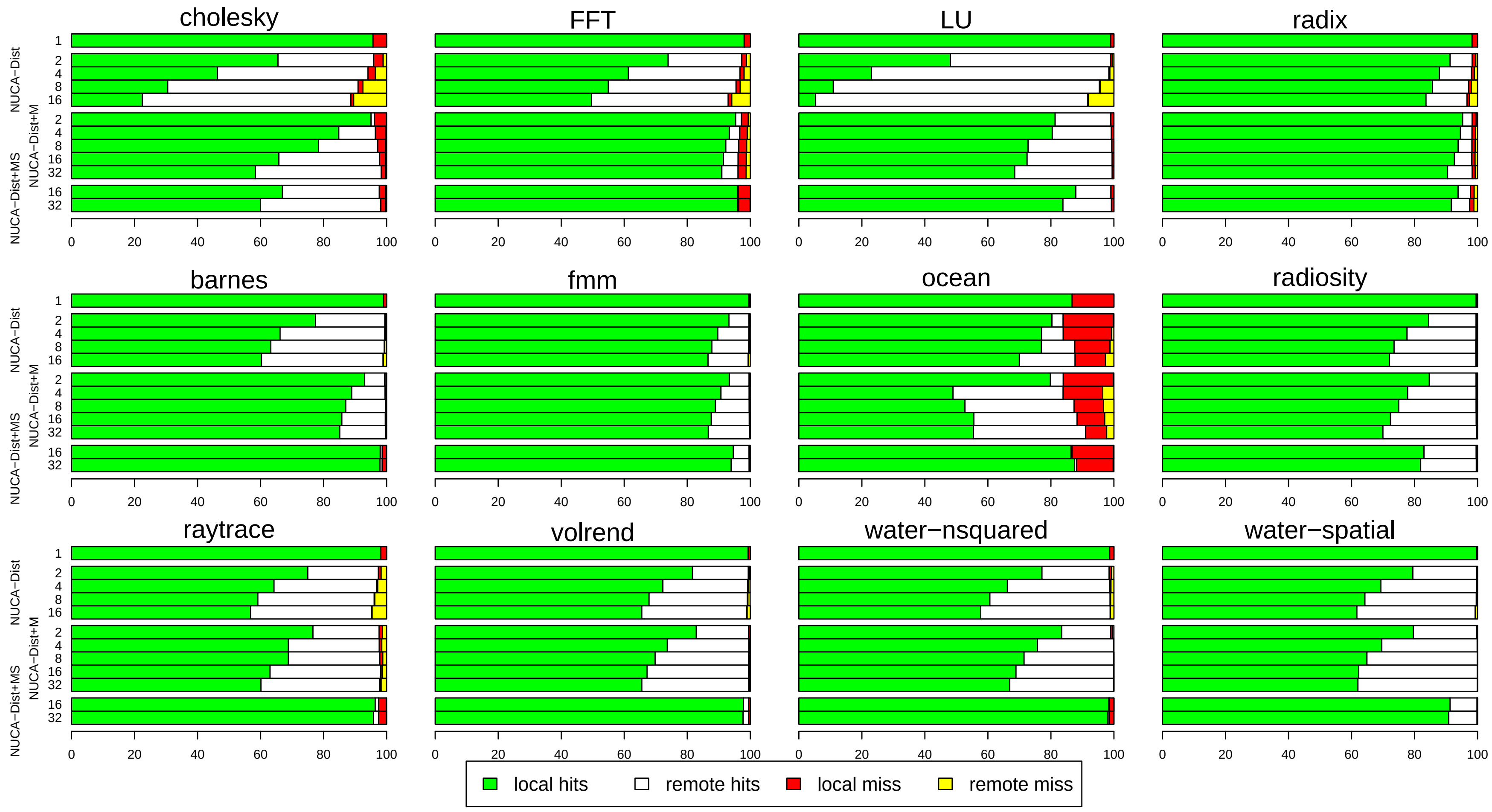
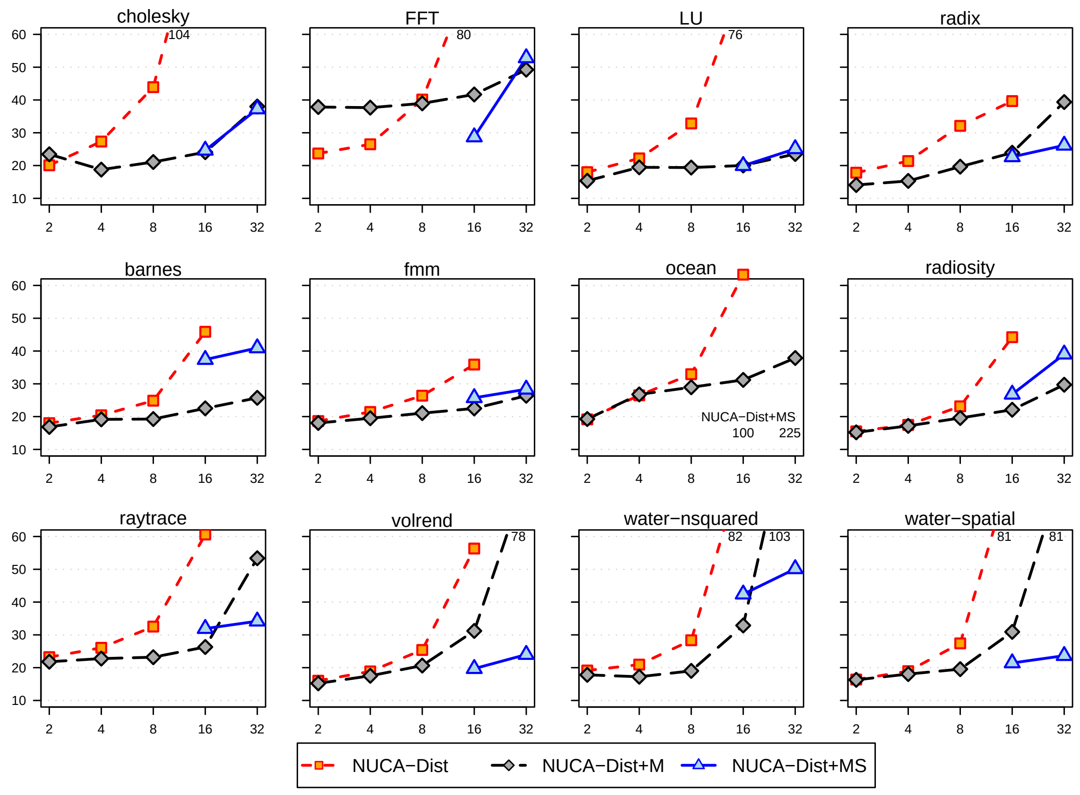
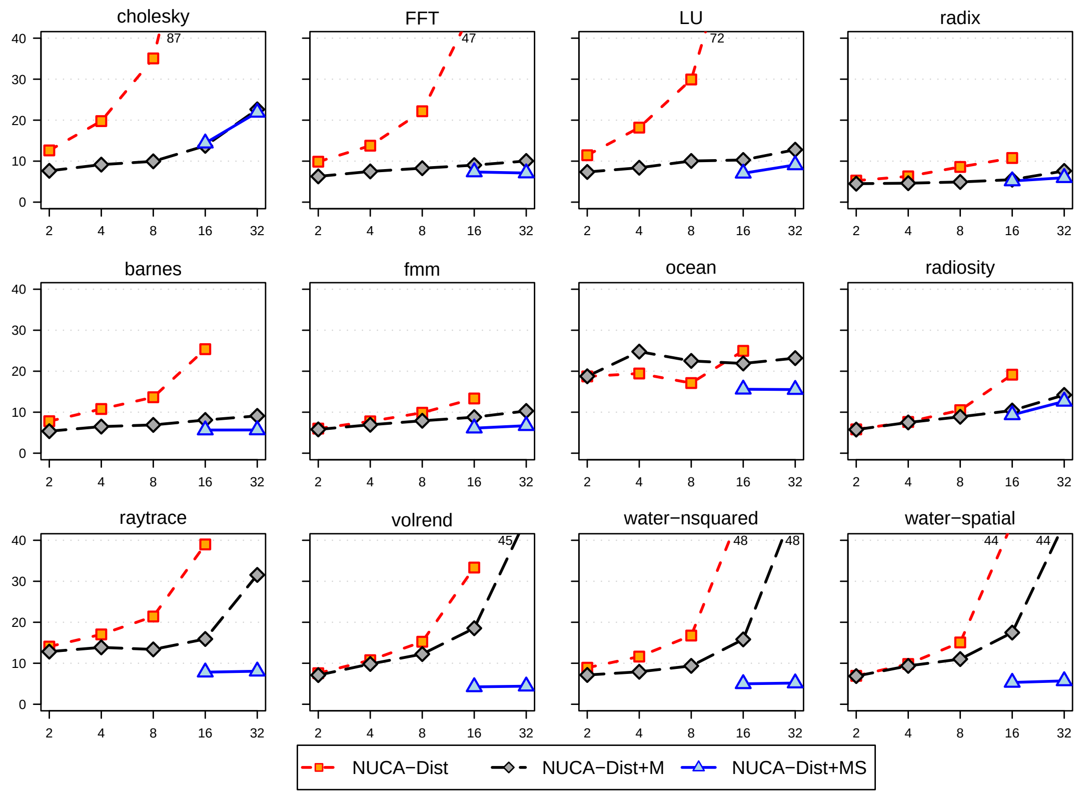


Figure 13.6: Distribution of memory accesses into local and remote accesses. They are then further divided into cache hits and misses.



(a) Average latencies for remote loads.



(b) Average latencies for all loads.

Figure 13.7: Average latencies for remote and all loads. The x-axis shows the number of processing tiles. The y-axis shows the latency in processor cycles.

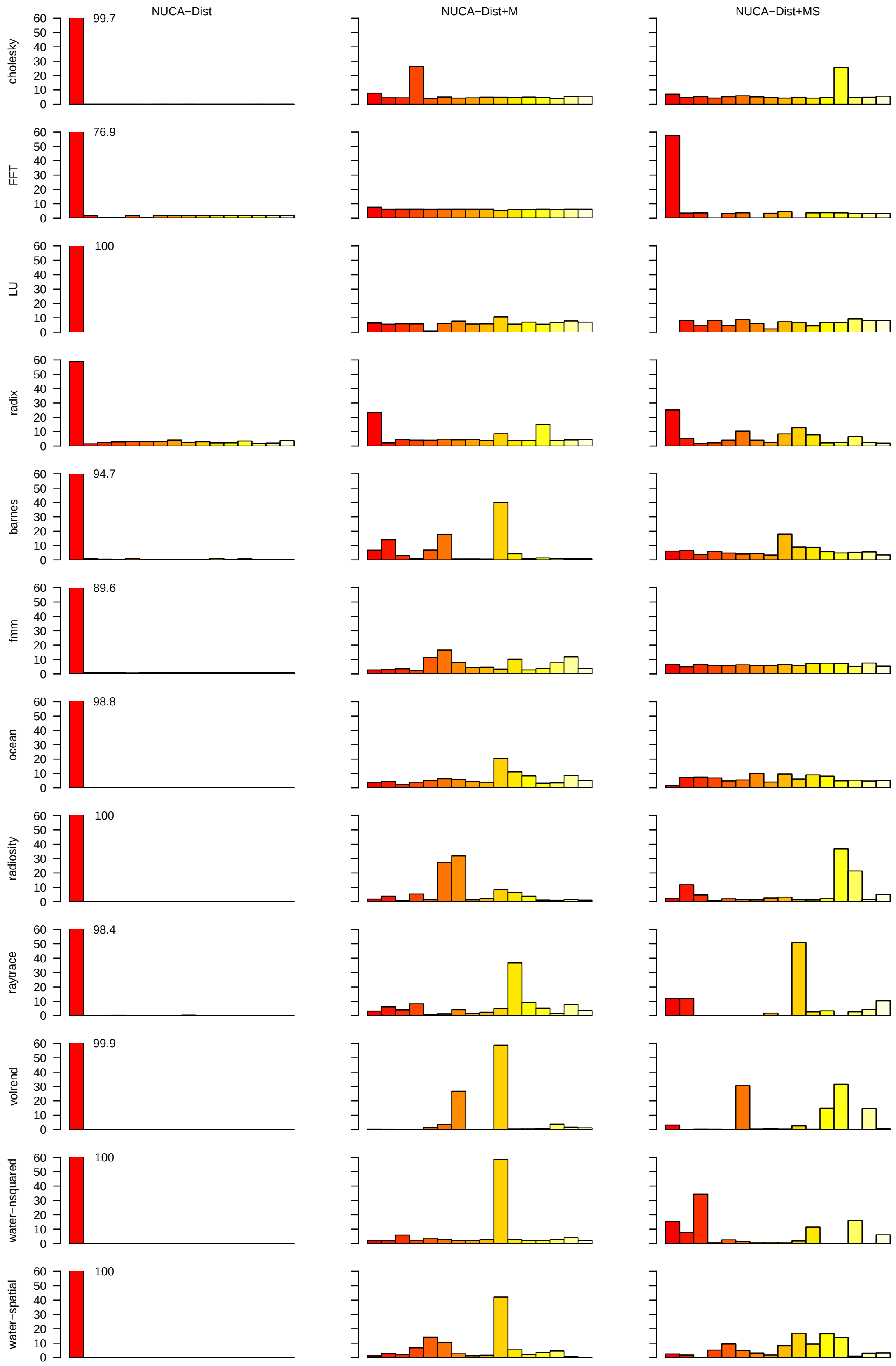


Figure 13.8: Remote access distribution for a 16 tile system. Each bar corresponds to the relative number of remote accesses handled by that tile. The y-axis shows how many remote requests in % were handled by which tile.

into remote accesses. Since the remote access latency is still slightly lower than the cache miss latency, the overall performance is not much affected.

In order to identify the source of this increase in access time, the simulator not only tracks the outcome of each memory access (local vs. remote, hit vs. miss) but also the destination of each remote access. Figure 13.8 shows the distribution of remote accesses across a 16 tile system for the *NUCA-Dist*, *NUCA-Dist+M* and *NUCA-Dist+MS* systems. The distribution is normalised to the total number of remote accesses. Thus, the height of a bar does not reflect the actual number of remote accesses, and as such, individual bars for different configurations or benchmarks cannot be used to compare the actual number of remote accesses handled by that tile. For example, for *fft* with *NUCA-Dist* configuration the plot shows that about 77% of all remote accesses are handled by tile 0. Tile 1, 4, and 6 to 15 handle about 2% and tile 2, 3 and 5 almost nothing. Figure 13.8 identifies the primary reason for the increase in latency: almost all requests are directed to the same tile. This is usually the tile that performed the initialisation of the data sets. With the number of tiles increasing, the number of remote accesses to this tile starts to increase. Unfortunately, this problem gets worse if tiles issue remote cache accesses at about the same time. Furthermore, with all requests being directed to the same tile, the number of conflict misses in the cache also starts to increase. Increasing the latency of remote caches accesses even further. In particular *cholesky*, *fft*, *lu* and *raytrace* suffer from this effect.

The migration extension (*NUCA-Dist+M*) was developed to address these issues. The overall results can be seen in figure 13.5. All benchmarks that had problems scaling to 8 tiles now perform acceptably up to 16 and sometimes even 32 tiles. The main reason is that the percentage of remote accesses has been greatly reduced, as can been seen in figure 13.6. Exceptions are *fmm*, *ocean*, *radiosity* and *water-spatial*. *Fmm* already performs acceptably in the base configuration, since it has one of the lowest remote access percentages of all benchmarks. Migrating pages does not help to improve this ratio even further. *Ocean* shows again a very different behaviour: while the percentage of cache misses is slightly reduced, the percentage of remote accesses increases significantly. The overall performance of *ocean* with migration is now slightly worse than before. The problem with *ocean* is its extreme high number of barriers. The barriers appear so frequently that there is not enough time to amortise the extra time the migration extension requires. Furthermore, the access pattern in *ocean* is of such a nature that the first touch policy does not select pages that are then accessed most by this tile. As for *radiosity* and *water-spatial*, the migration extension is not able to reduce the number of remote accesses. Still, both benchmarks perform better with migration than without. The reason is that the migration extension spreads out the remote accesses across several tiles (as shown in figure 13.8). This in itself results in a reduction of remote access latencies (figure 13.7a) and overall load latencies (figure 13.7b). This observation is also true for all other benchmarks. Overall, the observed performance improvement of *NUCA-Dist+M* over *NUCA-Dist* is as high as 249.3% and is 79.8% on average for all benchmarks and 16 processors.

However, once the number of tiles is increased to 32, the latency problem appears again: the latency of remote accesses for *raytrace*, *volrend*, *water-nsquared* and *water-spatial* increases greatly and prevents these benchmarks from scaling. The reason is that remote accesses are issued at almost the same time and the remote access controller cannot keep up with processing these remote requests. In order to reduce the percentage of remote accesses even further, the read-only sharing extension was developed. The success of this extension can again be seen in figure 13.6 (please note that the results shown do not utilise selective MAP table invalidation at barriers). The number of remote accesses is greatly reduced for all benchmarks except *cholesky*, and for *fft*, *barnes*, *ocean*, *raytrace*, *volrend* and *water-nsquared* almost completely removed. In terms of access distribution little effect is seen, accesses are still as evenly distributed as with *NUCA-Dist+M* (some benchmark a little more even others less even). One interesting thing is that the read-only sharing extension increases the latency for remote accesses for several benchmarks (figure 13.7a). This, however, has to be seen in the context of the reduced number of remote accesses. Thus, actually, the average latencies of all loads are reduced for all benchmarks (figure 13.7b). Furthermore, the read-only sharing extension improves *raytrace*, *volrend*, *water-nsquared* and *water-spatial* significantly and allows them to scale easily to 32 tiles. For the other 8 benchmarks there is only very little improvement in performance. In the case of *cholesky*, there is actually a significant performance degradation. The problem is the additional overhead introduced by the MAP table invalidation whenever a lock is acquired. The selective MAP table invalidation scheme was developed to address this final issue.

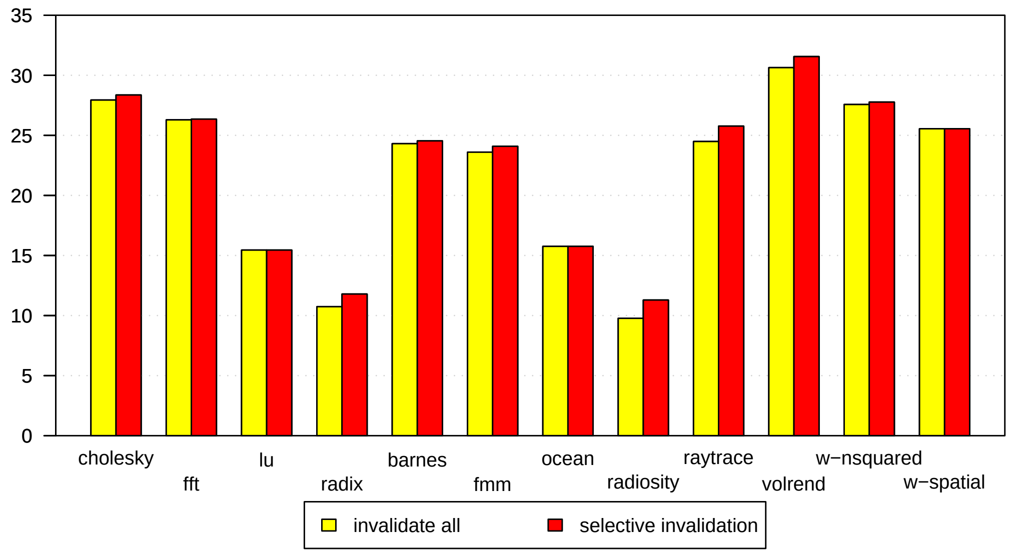


Figure 13.9: Speedups compared to execution of a single tile between a system that invalidates the whole MAP table on lock-acquire and a system that only invalidates selected entries. The y-axis shows the speedup.

Figure 13.9 shows the difference in speedup between a system that utilises this selective invalidation technique and a system that does not[2]. While the overall improvement of this extension is less impressive than for the previous extensions, it is able to improve the performance of some of the worst behaving benchmarks significantly. It is no surprise that selective invalidation does only make a small difference for most benchmarks: Only pages that are assigned to a tile are allowed to stay in the MAP table. However, as we have with the read-only sharing extension most data is shared and as such the corresponding pages still invalidated. The speedup for *radix* improves from 10.7 to 11.8, bringing the gap to the idealised system *Dir-Coh* from 29% down to 22%. *Radiosity* experiences an even larger improvement of the speedup from 9.8 to 11.3. This reduces the gap to *Dir-Coh* from 42% to 32%.

[2]Please note that the results in figure 13.9 were obtained with the SLOWMEM configuration and cannot be compared directly with all other results in section 13.5, which were obtained using the FASTMEM configuration.

Conclusion The last few sections showed that the performance of the baseline system *NUCA-Dist*, is already quite impressive for a few benchmarks. However, in order to allow the architecture to scale to a higher number of tiles and to be more suitable for a wider range of programs, it is necessary to augment it with extensions for migration, sharing and selective MAP table invalidation. The first two extensions come at no extra hardware complexity (assuming that an instruction to clear the local MAP table must have existed anyway for OS). The selective invalidation extension does not require any additional on chip storage (all information is already available in the existing structures), but just a simple circuit (less than 10 transistors per entry) that uses only local information to perform a more intelligent invalidation of some MAP table entries.

13.6. General Notes on Individual Overhead Analysis

Sections 13.7 and 13.9 will discuss the impact of particular negative aspect of the proposed system, namely flushing of caches at barriers, invalidating MAP tables on lock acquire and network latency. Before reading these discussions, a word of warning would be appropriate: while the modifications to the system were made in such a way that they should only eliminate the impact of a single negative aspect at a time, it became apparent that they also might have some other subtle effects on the runtime in general. For example, if a memory access is not delayed in the same way as in the original system, then a pointer can be loaded from memory faster. A write to the target of this pointer might then result that a different tile, as compared to execution on the unmodified system, becomes the owner of that pages. This allocation then might be a better or worse distribution of pages than before. This impact becomes apparent for example in *lu*: while the benchmark only has 32 locks, a modification to the system that eliminates the negative impact of invalidating the MAP table improves the performance by 2%. This is strange since all these locks happen to be in the first part of the benchmark before the first barrier. Since after the barrier the local MAP tables are invalidated and the global MAP table is cleared, there should be no effect afterwards. Furthermore, just the overhead of repopulating the MAP table cannot account for this performance gap. It turns out that by synchronising the MAP table, pages are allocated to tiles in a different way. This distribution is apparently more efficient than the distribution with full invalidation.

Unfortunately, these indirect effects could not be eliminated. Thus, for all the results presented in this section it is difficult to exactly quantify the impact of each overhead. For example, it might be possible that eliminating a certain overhead would give an improvement, which is then cancelled by performance degradation due to a worse page to tile allocation. However, while there are some cases where an overhead elimination resulted in worse performance, in general they were beneficial. Thus, the results in this section give a good idea of the impact of a certain overhead (especially if the effect is similar across several benchmarks), but do not quantify it exactly.

Benchmark	*Barriers*	*Locks*
cholesky	-0.18	0.77
fft	0.23	0.04
lu	8.01	<0.01
radix	8.75	0.17
barnes	-0.79	0.61
fmm	1.67	0.21
ocean	10.56	-0.51
radiosity	3.74	-1.99
raytrace	-1.28	5.57
volrend	0.00	0.00
water-nsq	0.18	0.43
water-spa	0.47	0.04
facerec	<0.01	0.07
mpegdec	-0.88	-0.15
mpegenc	-0.54	-0.54

Table 13.2: Overhead for *NUCA-Dist+MS* with 32/16 processors in % caused by flushing the cache at barriers and invalidating the MAP table on a lock acquire. A negative overhead means that *NUCA-Dist+MS* executes the program faster than the idealised system.

13.7. Analysis of Introduced Overheads by Migration and Read-Only Sharing

The proposed architecture with the migration and read-only sharing extension has two operations that can cause a significant overhead in execution time:

- The migration extension requires that all caches be flushed at a barrier. This operation can be very expensive if the same data is accessed again after the barrier and has to be reloaded from memory. On the other hand, if the data were accessed on a different tile then it would not have been available on that tile anyway and would have to be fetched from memory. The general speed improvement of the remapping scheme suggests that the latter is at least true for some of the data.
- The read-only sharing extension requires a selective invalidation of the local MAP table of the tile that acquires a lock. This means in the best case nothing is invalidated, since all entries refer to non-shared pages. Or in the worst case, all entries are invalidated, since they all refer to shared-pages. Both of these extreme cases are unlikely to happen. The success of the read-only sharing scheme indicates that there must be some shared pages. On the other hand, as discussed in sections 7.4 and 8.1, access to the lock variable requires (even if the lock variable is just read) that this access is treated as a write access. A write access will always result in a MAP table entry that marks the page the lock variable is placed on as owned (either locally or remote). Thus, all pages can only be shared if no lock operations have been performed.

From the above arguments it seems that it is very difficult to just base the overhead on a best and worst case analysis. Instead, specialised versions of the binary and simulator were used to establish how much overhead is caused for each benchmark. In order to establish the overhead of cache flushes, the simulator does not perform a cache flush, but instead updates all cache lines with the current copy from memory within 0 cycles. Please note that the simulator still performs the write back of dirty cache lines, as it would normally happen during a cache flush. In order to establish the overhead of MAP table invalidations at lock acquires, the simulator no longer invalidates, but instead synchronises, the local MAP table with the global OS MAP table within 1 cycle. Table 13.2 lists these overheads. For most benchmarks, these overheads are insignificant. Notable exceptions are: *lu*, *radix* and *ocean* have a significant barrier overhead and *raytrace* has a significant overhead for locks. The barrier overhead does not come as a surprise, since these programs spend the least amount of time between barriers of all benchmarks (see table 11.1). While *fft* spends a similar short amount of time between barriers, it does not experience a similar overhead. The reason is that the data is accessed by a different tile after the barrier anyway. Thus, there is no benefit to keeping the data in a cache of tile that will not access it. In order to establish the impact of the cache flushing at barrier, the simulator was modified to keep track of the cache hit rate during the parallel part of the execution. Figures 13.10 to 13.13 show the cache hit rate during the parallel part of the computation for SPLASH-2 benchmarks running on a 32 tile system. As one would expect, the hit rate drops right after the barrier. However, this is only a short transient impact and hit rates quickly return back to normal. As for *raytrace*, again the 5% slowdown due to MAP table invalidation does not come as a surprise. *Raytrace* has with *radiosity* the highest lock frequency. However, *radiosity* seems not to be affected by it. The difference between these benchmarks is however, that *radiosity* performs about 20% of all cache accesses remotely. Remote cache accesses always indicate that pages are actually owned by a tile, rather than being shared. Since MAP table entries to owned pages are not invalidated at a lock acquire, *radiosity* does not have to repopulate the MAP table as often as *raytrace*. Though, looking at the overall performance, it seems that having to repopulate the local MAP is less performance critical than having to perform 20% remote accesses.

13.8. Network Traffic Analysis

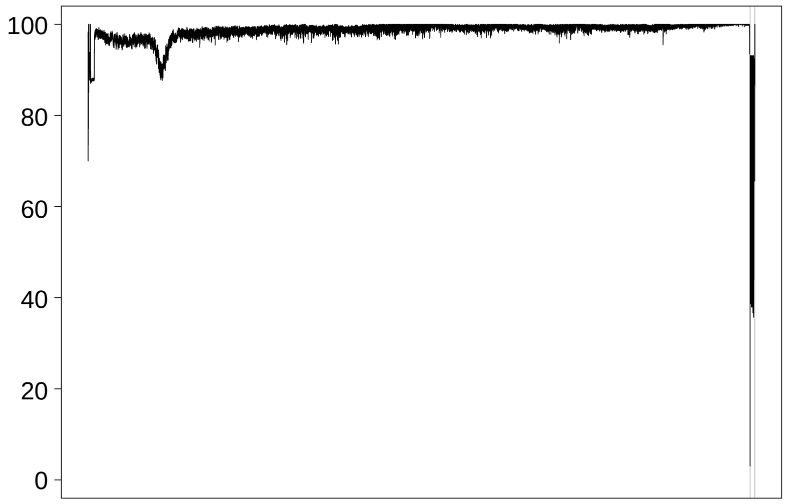
Another important aspect of the proposed architecture is the amount of network traffic that is generated. The amount of traffic allows further insight into potential bottlenecks. In particular it is important to know if the observed network traffic (based on the cache access distribution) happens all at once or is evenly distributed across the whole computation. In order to answer these questions, the simulator counts the number of active messages in each timestep. Active messages also include messages that are waiting in a queue either waiting to be sent to another tile or waiting for a cache miss to be resolved. In order to reduce the amount of collected data, the simulator computes the maximum number of active messages seen so far. Every n cycles this number is recorded in a histogram file and the maximum reset to 0. Thus, the number recorded reflects the peak traffic during the interval.

Benchmark	*Duration*	*Granularity*
cholesky	146,060k	2,000
fft	2,680k	2,000
lu	90,570k	2,000
radix	3,640k	2,000
barnes	425,600k	6,000
fmm	402,400k	6,000

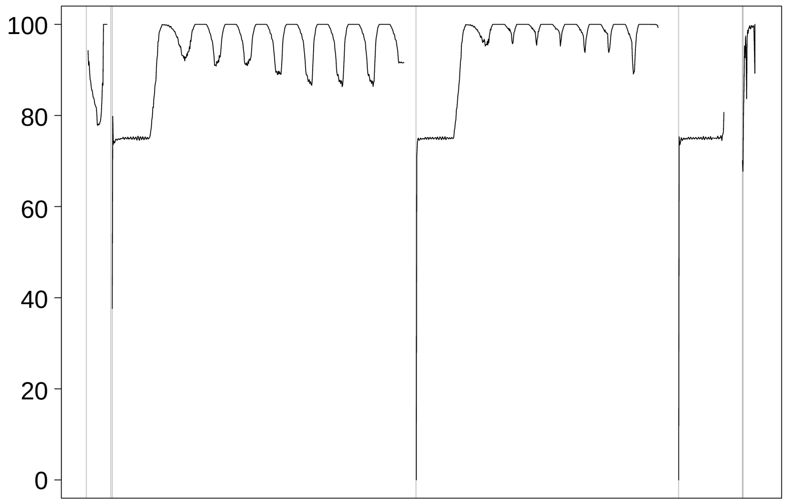
Benchmark	*Duration*	*Granularity*
ocean	151,500k	3,000
radiosity	152,780k	2,000
raytrace	137,890k	2,000
volrend	131,850k	2,000
water-nsq	73,940k	2,000
water-spa	72,780k	2,000

Table 13.3: Duration and granularity used for creating the network activity histograms. Both values are given in processor cycles.

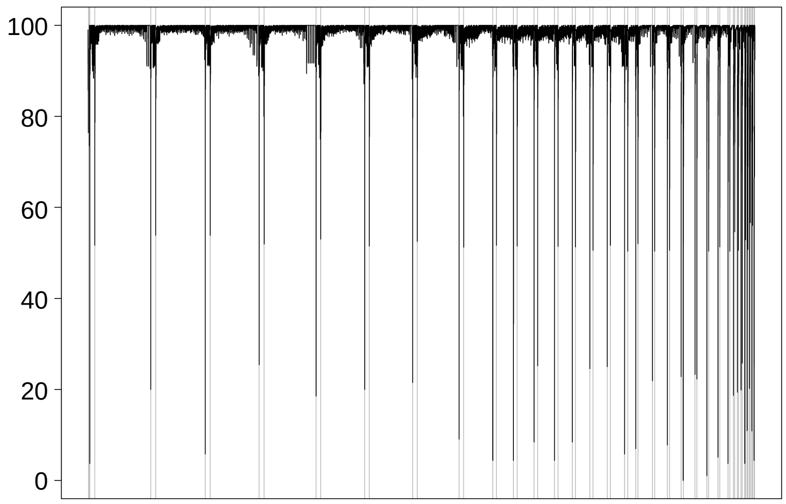
The histograms were collected using a 32 tile system. Table 13.3 lists length of the interval in processor cycles during which the numbers of active messages are collected by the simulator. The interval start and end is determined by the reported begin and end of the computation phase in each SPLASH-2 benchmark. Granularity indicates after how many cycles an entry is written to the histogram file. The resulting histograms are shown in figures 13.14 to 13.17. The vertical gray lines in the plots show the position of barriers. The plots show that for most benchmarks the number of active messages stays below 20 for most of the time. A couple of benchmarks show a spike just before or after a barrier. This spike is caused by the PE executing a large number of remote stores. Since the PE is not stalled when it is creating



(a) cholesky

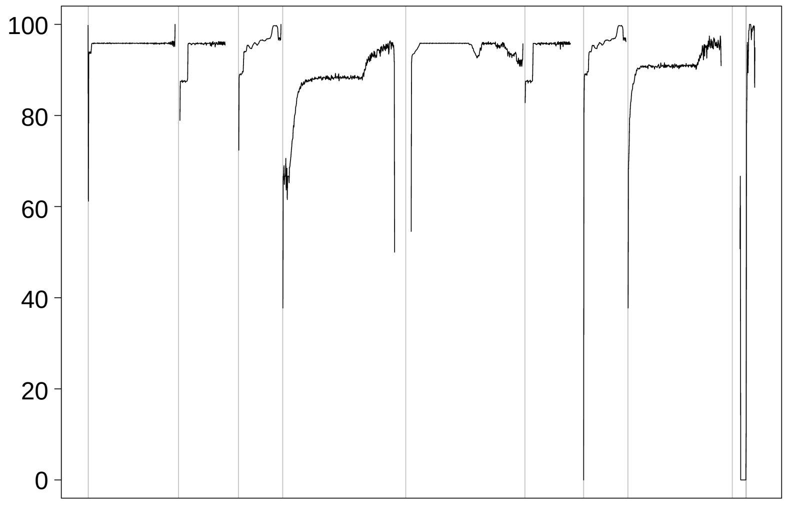


(b) fft

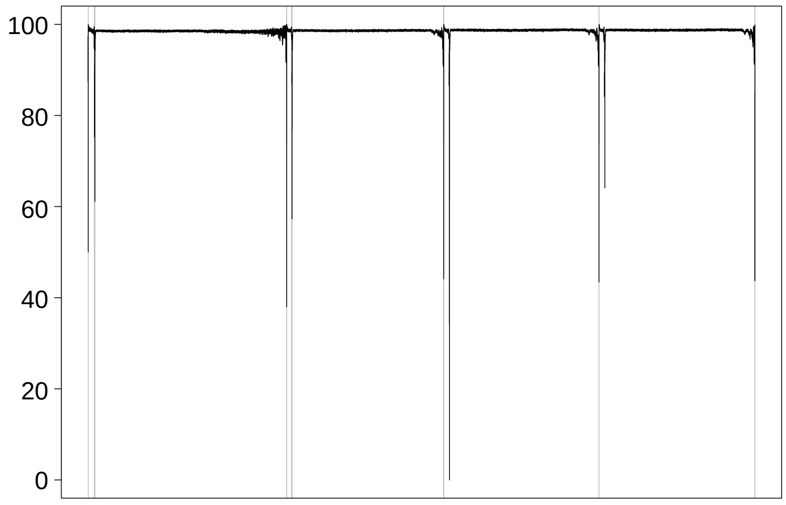


(c) lu

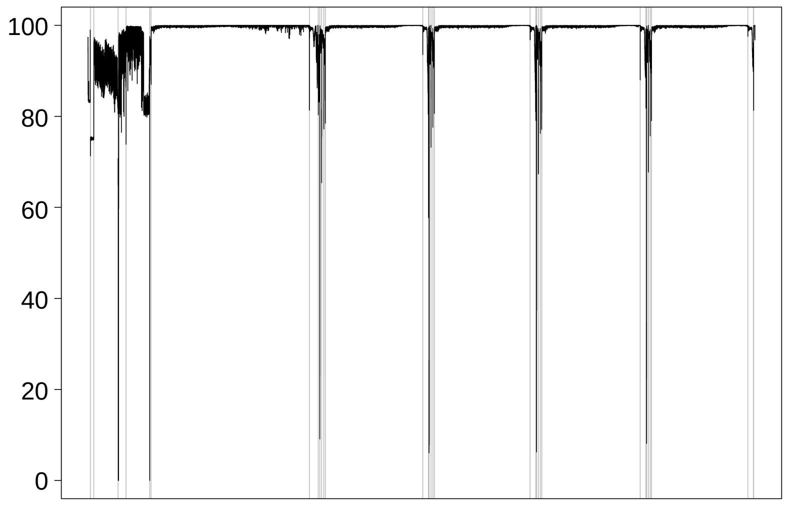
Figure 13.10: Cache hit rate during the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the hit rate in percent.



(a) radix

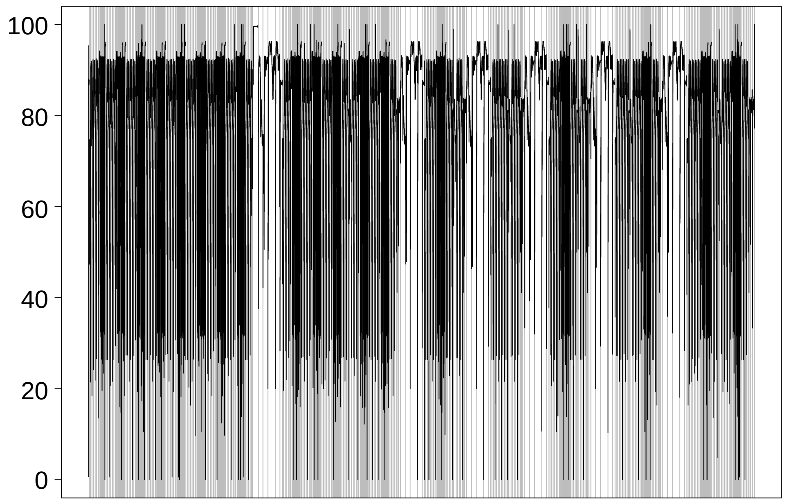


(b) barnes

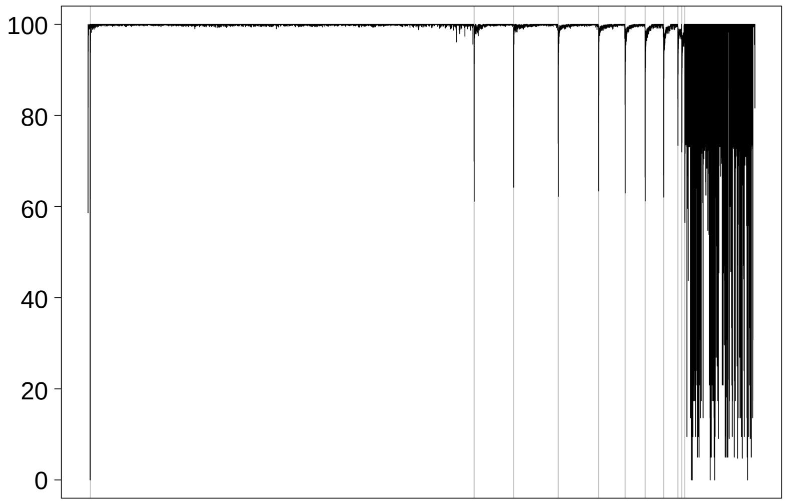


(c) fmm

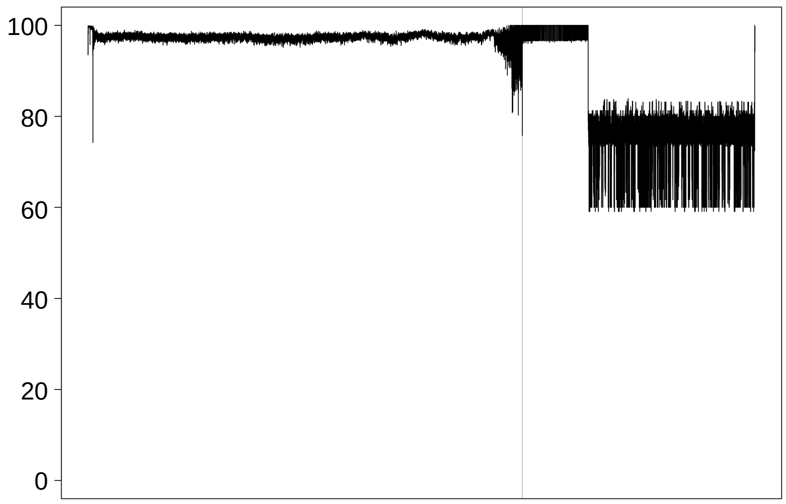
Figure 13.11: Cache hit rate during the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the hit rate in percent.



(a) ocean

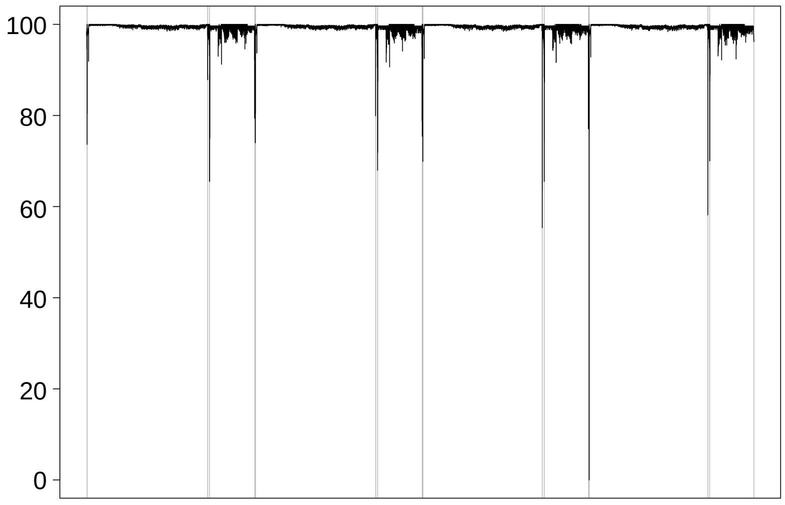


(b) radiosity

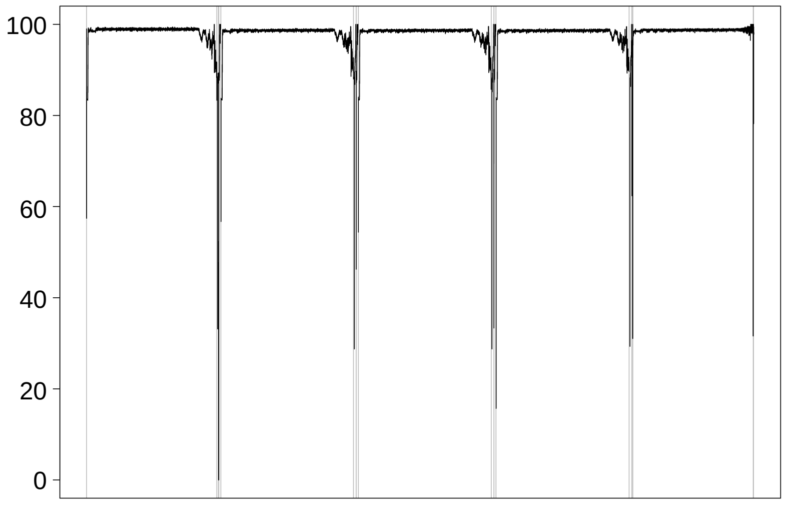


(c) raytrace

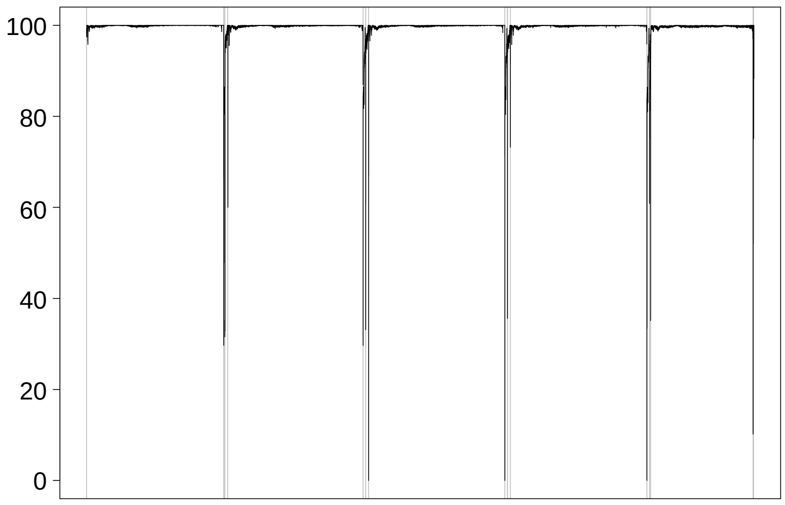
Figure 13.12: Cache hit rate during the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the hit rate in percent.



(a) volrend



(b) water-nsquared



(c) water-spatial

Figure 13.13: Cache hit rate during the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the hit rate in percent.

remote store requests, they just happen to pile up in the send queue. However, these events occur relatively seldom and usually for a very brief period of time, it did not seem necessary to create a throttling mechanism that limits the number of active remote stores. Exceptions to this are *cholesky*, *radix* and *radiosity*.

Cholesky has on average between 20 and 30 active messages. This is not surprising considering it has 39% remote cache accesses. However, it seems that these accesses are rather evenly distributed across the whole execution of *cholesky* and thus cause little congestion (as is confirmed by *cholesky*'s good speedup).

Radix, on the other hand, has two longer periods where there are about 50 active messages in the system. These periods will most certainly result in some messages being delayed and might explain some of the observed gap. In addition, it seems that there is a discrepancy between the shown number of active messages and the relatively low remote cache accesses of about 5%. The explanation is that, unlike other benchmarks, it performs its initialisation in parallel mode. The cache access statistics are collected during the full time the application is in parallel mode. However, the histogram only shows the region that is actually measured as computation time by the benchmark.

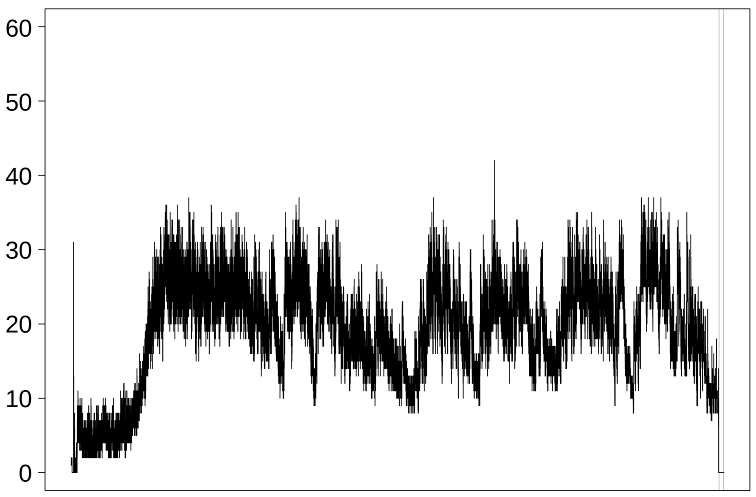
Radiosity has some increased traffic during its second half of execution with active messages between 15 and 35. Again, while this explains some performance degradation, it should not be a problem to handle this amount of traffic.

Benchm.	*Ratio (r/w) abs.*	*Ratio rel.*
cholesky	108,206,273 / 1,432,122	98.69 / 1.31
fft	111,397 / 10,618	91.30 / 8.70
lu	20,313,588 / 1,030	99.99 / 0.01
radix	482,930 / 560,958	46.26 / 53.74
barnes	12,618,927 / 1,259,085	90.93 / 9.07
fmm	69,136,912 / 4,041,156	94.48 / 5.52
ocean	1,288,634 / 23,958	98.17 / 1.83
radiosity	44,334,452 / 1,475,006	96.78 / 3.22
raytrace	4,581,546 / 414,041	91.71 / 8.29
volrend	7,411,302 / 409,394	94.77 / 5.23
water-nsq	618,521 / 372,610	62.41 / 37.59
water-spa	18,842,895 / 673,622	96.55 / 3.45

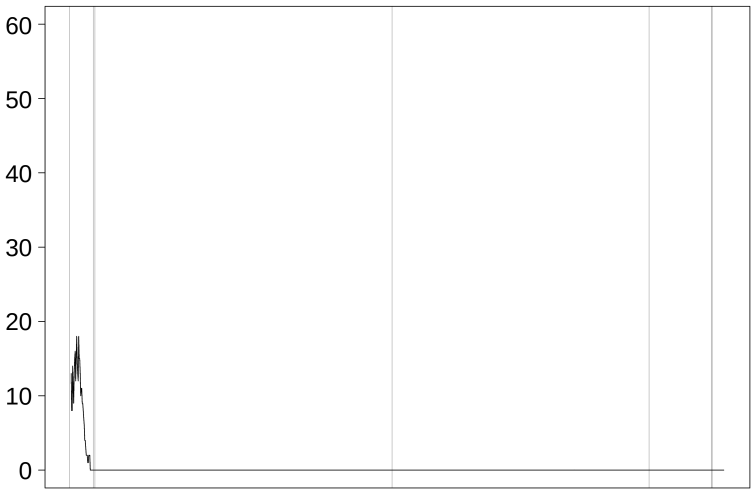
Table 13.4: Ratio between remote read and write accesses during the parallel part of the execution.

Relating the network activity results with the load latency plot (figure 13.3) is somewhat problematic due to two problems: first, the load latency plot shows latencies of all loads during the parallel part of the execution. However, the network activity plots only show the part of the execution that is identified as the computational part. For most benchmarks, these execution parts are identical; for a few (e.g., *radix*) these execution parts are different, since the initialisation happens in parallel. The second problem is that the load latency plot only considers loads, while the network activity plots also show activity caused by stores. For example, it is possible that the network activity plot shows a high network activity, while this activity is only caused by stores and does not affect any loads. Table 13.4 gives a rough answer to how much of the network activity can be attributed to stores. Unfortunately, it does not show how loads and stores are intermixed. Still, if this table is used to group benchmarks into groups with similar load percentage (91%: *fft*, *barnes* and *raytrace*. 94%: *fmm* and *volrend*. 96.5%: *radiosity* and *water-spa*. 99%: *cholesky* and *lu*) one can see that a plot with higher network activity also results in higher load latencies.

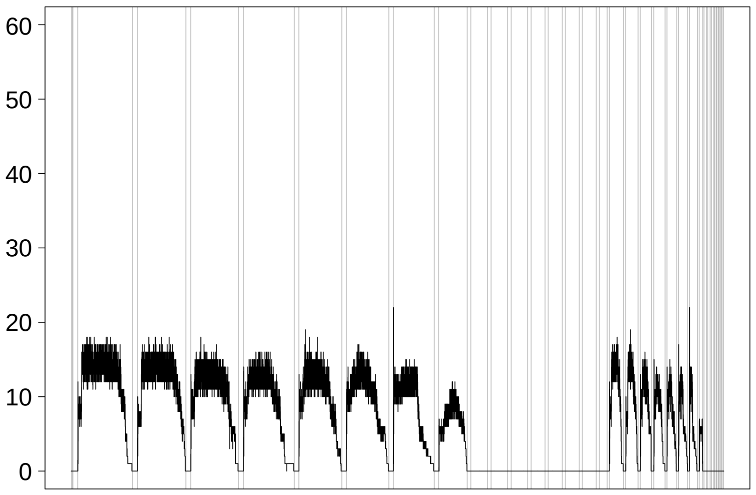
The overall result of the network traffic analysis is that the traffic observed is mostly low and should not have too much impact on most benchmarks. For the three benchmarks with the most traffic (*cholesky*, *lu* and *radiosity*), this has been confirmed by Kuhn [Kuh07], who implemented a detailed network model for the proposed architecture. He found that *cholesky*



(a) cholesky

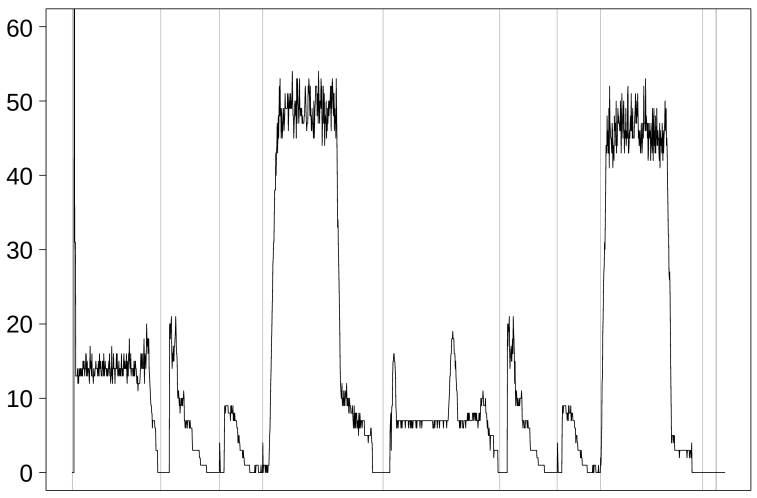


(b) fft

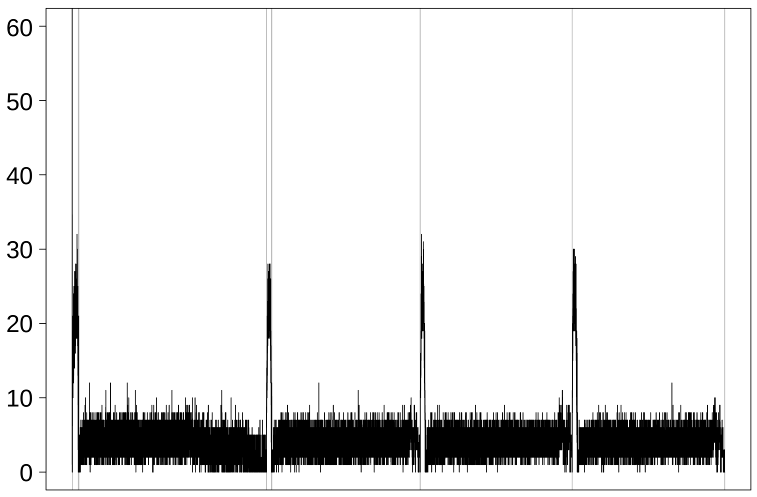


(c) lu

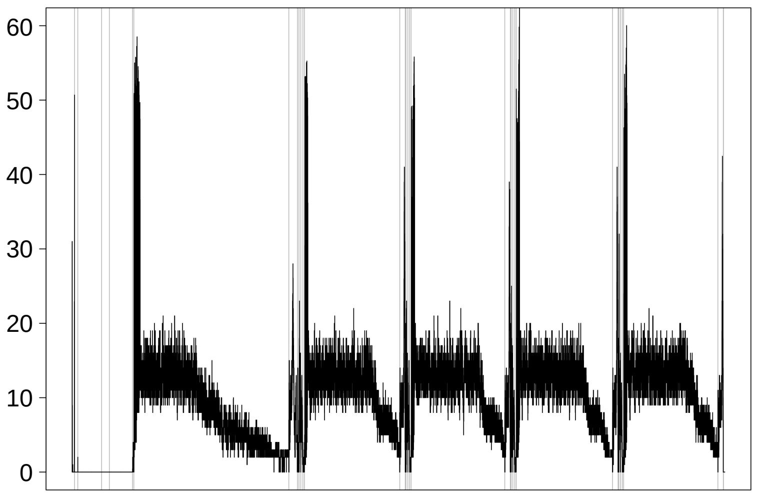
Figure 13.14: Active messages in the network during the execution of the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the maximum number of messages during the interval.



(a) radix

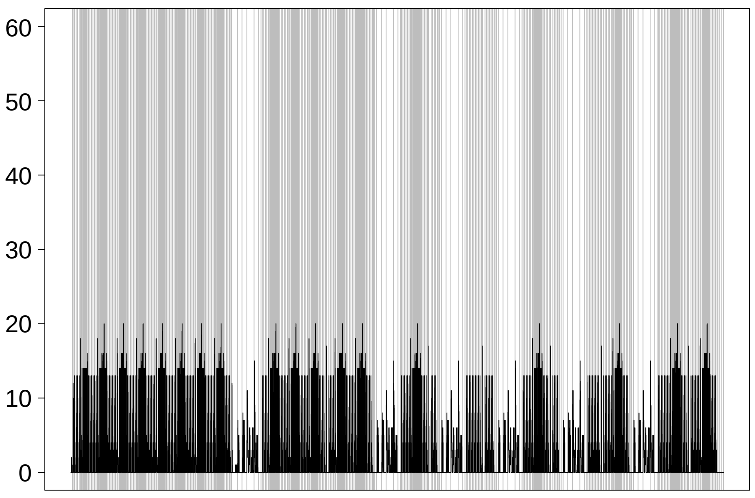


(b) barnes

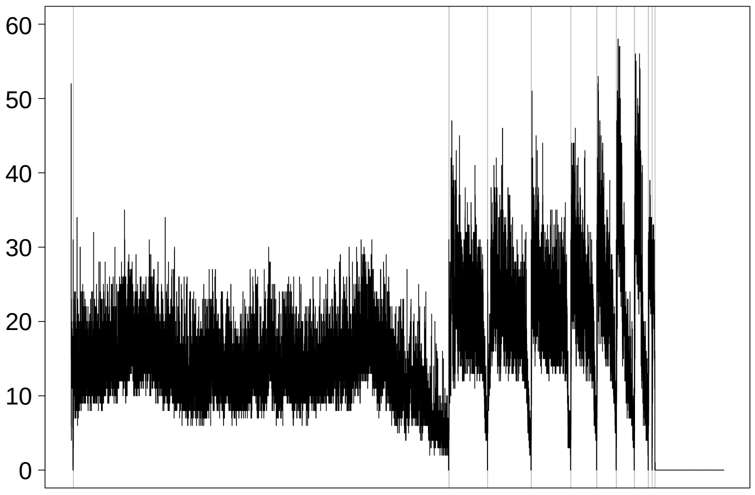


(c) fmm

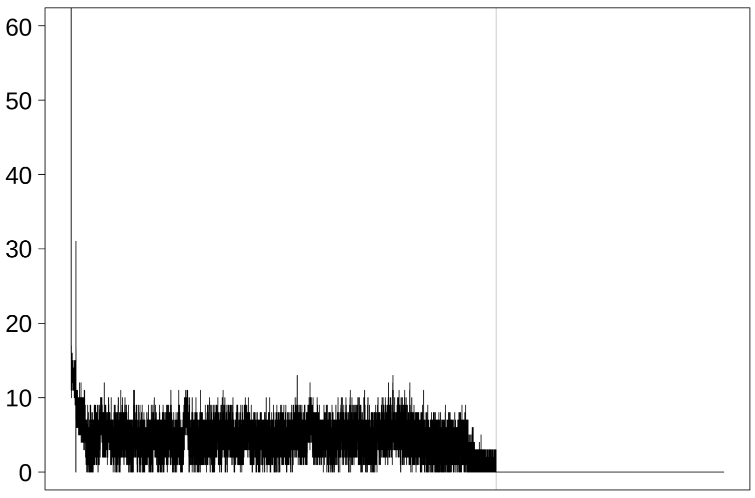
Figure 13.15: Active messages in the network during the execution of the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the maximum number of messages during the interval.



(a) ocean

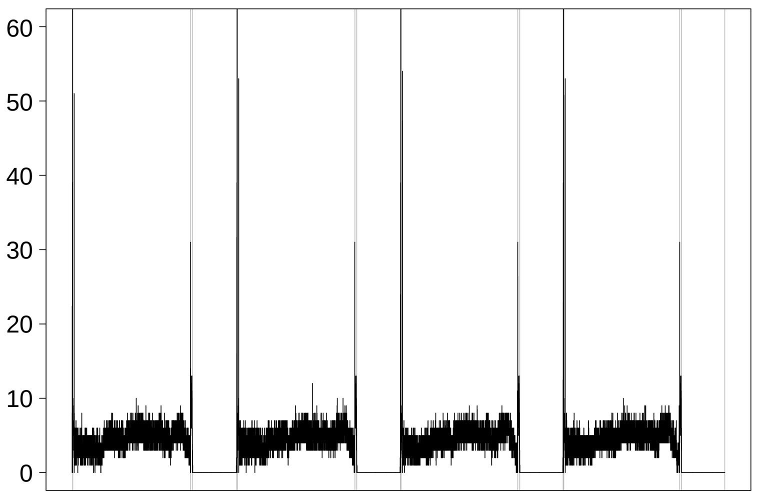


(b) radiosity

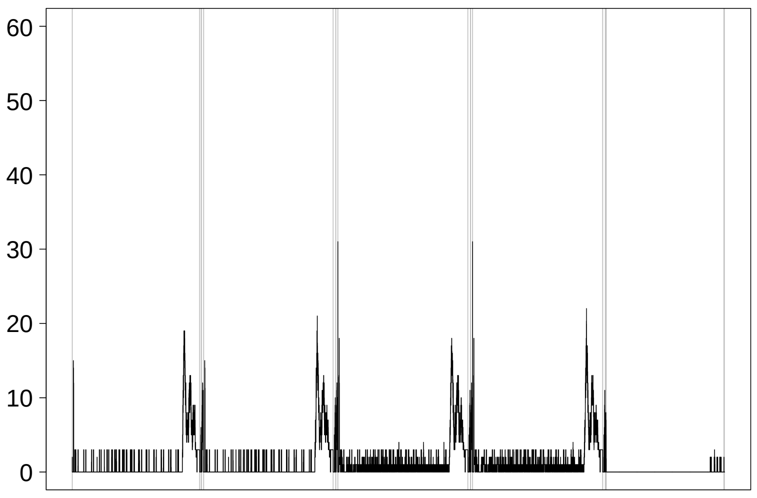


(c) raytrace

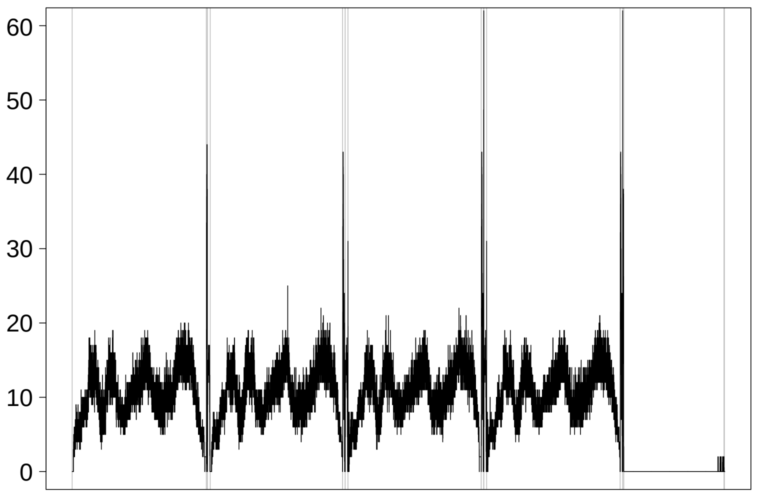
Figure 13.16: Active messages in the network during the execution of the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the maximum number of messages during the interval.



(a) volrend



(b) water-nsquared



(c) water-spatial

Figure 13.17: Active messages in the network during the execution of the parallel part of the application. The gray vertical lines show the position of barriers during the execution. The x-axis shows the elapsed execution time. The y-axis shows the maximum number of messages during the interval.

suffers 0.08% performance degradation, while *lu* and *radiosity* execute about 4% faster. The reason for this increase in performance is most likely a better page to tile distribution. This effect is explained in more detail in section 13.6.

13.9. Network Latencies Analysis

The previous section discussed whether the on-chip network could cope with the amount of generated traffic. This section will have a look at how much performance is lost because a remote access simply takes more time to perform than a local access. This delay is due to two reasons, first, it takes several cycles to route a request across the chip even if the message does not experience any other delay, and second, if several messages are in the network at the same time then messages can get delayed due to limited bandwidth available. The Fast Network simulator (see section 10.5) has been developed to evaluate this aspect. Table 13.5 shows the results of this study. The main observations are that the amount of performance degradation roughly follows the amount of remote memory accesses (see figure 13.2) and compared to the previous overhead study for locks and barriers in section 13.7, it can be clearly seen that network latencies are a serious cause for performance degradation for some benchmarks. Overall, this result explains that a significant reason for the performance gap to *Dir-Coh* is the latency it takes to perform a remote access.

A very interesting case is *cholesky*, since it runs faster on the Fast Network Simulator than the directory coherent version. This result helps to explain the previously observed behaviour of *cholesky* that even though it has about 38% remote accesses it only experiences a 6% slowdown. It seems that the proposed architecture with a fictional network is much more suitable for executing *cholesky* than the directory coherent one. However, this advantage is then negated by about 40% remote accesses and the associated delays. Still the original advantage is enough to limit the performance degradation to just 6%.

Considering the large impact remote access latencies have on the overall performance, it would be interesting to further pin down the reason. The Fast Network simulator makes two simplifications: first, the network latency is assumed to be always 1 (independent of distance and message length). Second, the remote cache access controller (RAC) can start handling all requests that have been received in that cycle. In order to determine the impact of each of these simplifications, the fast RAC of the Fast Network simulator was replaced with the normal RAC of *NUCA-Dist+MS*, which can only start handling a single request per cycle. Table 13.6 shows the results of this analysis.

Benchmark	*Network*	*DC*
cholesky	11.46	6.12
fft	-0.08	1.57
lu	18.51	26.74
radix	3.44	21.87
barnes	0.73	23.38
fmm	5.46	14.91
ocean	-0.45	31.15
radiosity	22.08	32.40
raytrace	2.13	5.57
volrend	1.00	0.00
water-nsq	0.18	19.62
water-spa	12.62	18.53
facerec	1.29	1.57
mpegdec	24.62	26.79
mpegenc	1.80	1.62

Table 13.5: Overhead for *NUCA-Dist+MS* with 32/16 processors in % caused by network delays. *DC* is given as a reference to the performance gap towards the idealised directory coherent system *Dir-Coh*. A negative overhead means that *NUCA-Dist+MS* executes the program faster than the idealised system.

Benchmark	*fast*	*normal*
cholesky	11.46	0.32
fft	-0.08	0.00
lu	18.51	18.51
radix	3.44	-0.60
barnes	0.73	0.28
fmm	5.46	5.46
ocean	-0.45	-0.45
radiosity	22.08	22.51
raytrace	2.13	2.42
volrend	1.00	1.00
water-nsq	0.18	0.18
water-spa	12.62	12.62
facerec	1.29	0.93
mpegdec	24.62	24.20
mpegenc	1.80	1.71

Table 13.6: Comparison of the gaps in % between *NUCA-Dist+MS* and a fast network simulator with and without a fast remote cache access controller. A negative gap means that *NUCA-Dist+MS* executes the program faster than the idealised system.

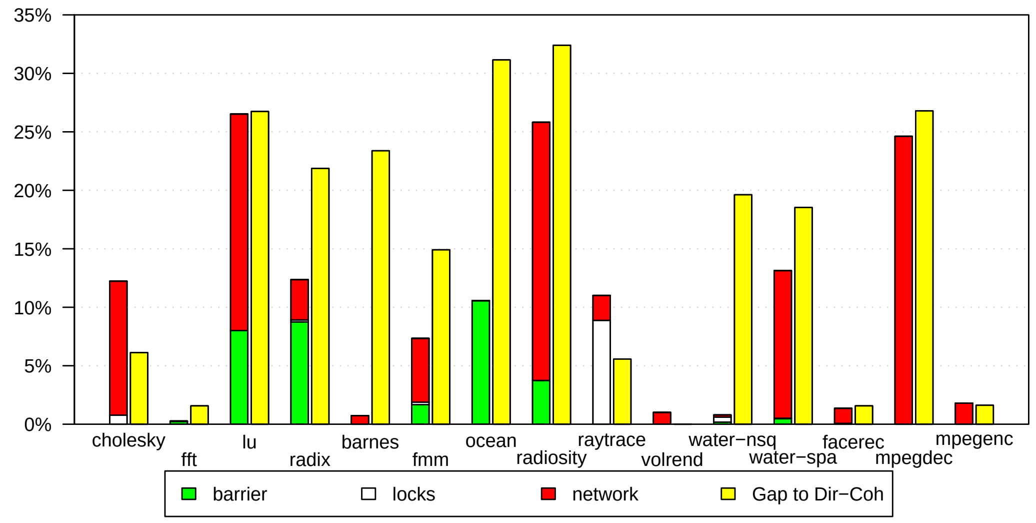


Figure 13.18: Gaps between *NUCA-Dist+MS* and systems that do not suffer from the cache flush overhead at barriers, the MAP table invalidation at lock acquire and network bandwidth and latency restrictions. The gap with *Dir-Coh* is shown as a reference.

The results show that the only reason for the performance increase in *cholesky* and *radix* is indeed the fast RAC. For the rest of the benchmarks, it seems that having an RAC that can start handling multiple requests at the same time makes little difference.

Combination with Lock and Barrier Overhead Analysis Figure 13.18 tries to show the combined impact of the network latency overhead (discussed in this section) and the lock and barrier overhead (discussed in section 13.7) by just adding each individual overhead. However, these results should be taken with a grain of salt. Since all overhead studies resulted in slightly different page to tile mappings, combining them might result in a slightly larger or smaller gap than expected. Still, this plot gives some valuable insight into the behaviour of some benchmarks. As for *fft*, *lu*, *radiosity*, *volrend*, *water-spatial*, *facerec*, *mpegdec* and *mpegenc*, this graphs explains the observed gaps fairly well. To a lesser extend the same can be said for *radix* and *fmm*. However, the observed gap in *barnes*, *ocean* and *water-nsquared* cannot be explained by just studying these three overheads. Some performance degradation for *ocean* and *water-nsquared* can be explained with the longer lock acquiring time (see section 13.3). As for *barnes* and *water-nsquared*, both benchmarks experience somewhat more off-chip accesses with *NUCA-Dist+MS* than *Dir-Coh* (*barnes*: 1.4% vs. 0.3%, *water-nsquared*: 1.4% vs. 0.3%).

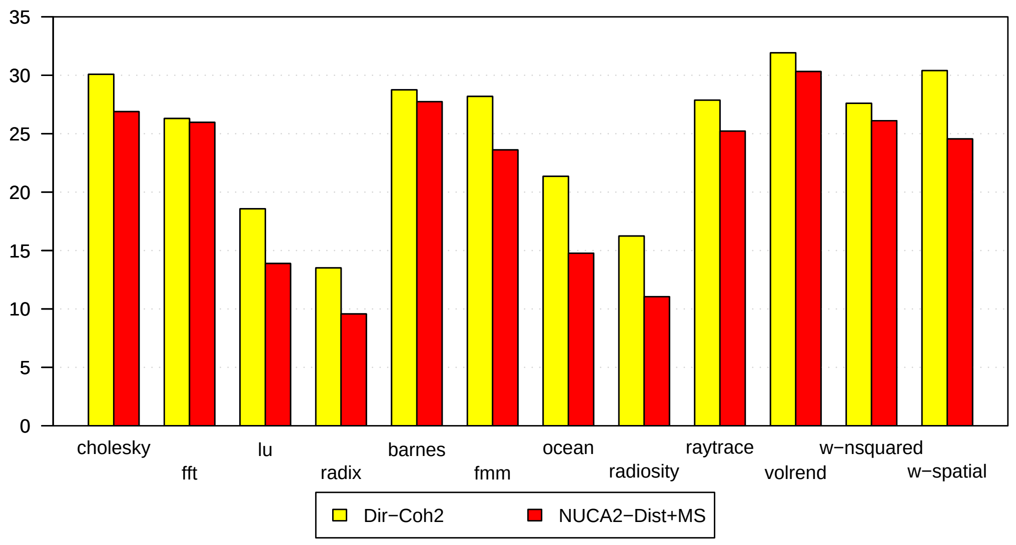


Figure 13.19: Speedups for 32 tiles with 2nd level caches with respect to the execution time (after initialisation) of a single tile also with a 2nd level cache. The y-axis shows the speedup.

13.10. 2-Level Cache Hierarchy

Figure 13.19 shows the speedup results of such a system for both the proposed scheme (*NUCA-Dist+MS*) and the distributed directory scheme (*Dir-Coh*). Note that these speedup numbers are not directly comparable to those in Figure 13.19, because they are normalised to different sequential execution times (with and without the L2 cache, respectively). The figure shows that the performance gap between *NUCA-Dist* and *Dir-Coh* remains mostly the same as for systems without the second-level cache (the gap range is now 1.3%-32% and the average gap is 16%), demonstrating that the proposed scheme also works with the addition of a second level of cache.

13.11. MAP Table Granularity

The configurations evaluated so far use a page table size of 4K. However, considering the very low miss rates in the MAP table, it seems that by decreasing the page size the effects of false sharing within a page can be reduced. This hopefully results in more data being accessed locally. On the negative side, the number of capacity misses in the MAP table is likely to increase.

Table 13.7 shows the results of this analysis. As expected, the number of misses in the MAP table increases and, on average the miss rate is now 3.6 times higher than in a system with 4K MAP pages. However, the impact on the overall performance of the system is less obvious, some benchmarks (like *lu* or *mpegdec*) greatly benefit from smaller MAP pages, while other benchmarks do not benefit at all and actually lose performance (like *fmm*, *ocean* or *mpegenc*). This behaviour can be explained by looking at the cache access distribution of the system with the finer granularity (shown in figure 13.20). For benchmarks that profit from the smaller granularity the number of remote accesses is reduced (in the case *lu* they are almost completely eliminated). For benchmarks that do not profit, the local to remote ratio almost remains unchanged. These benchmarks just have to pay the penalty of missing more often in the local MAP table. In total only 4 benchmarks benefit from the smaller MAP table pages; all other

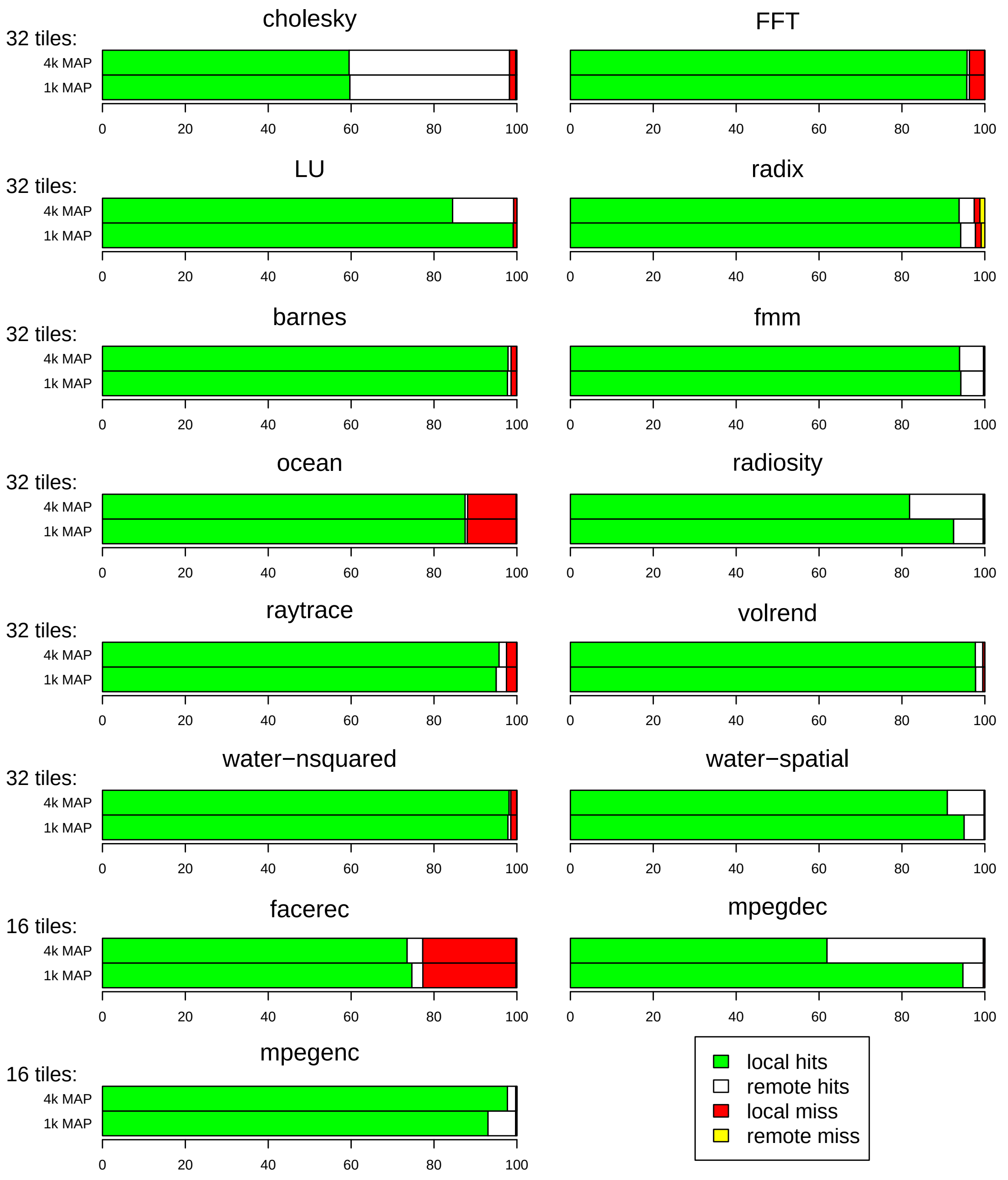


Figure 13.20: Distribution of memory accesses into local and remote accesses for *NUCA-Dist+MS* with a MAP table with either 4k or 1k granularity. The accesses are then further divided into cache hits and misses.

Benchmark	*4k Pages*		*1k Pages*		*Difference*	
	speedup	*MAP misses*	*speedup*	*MAP misses*	*speedup*	*MAP misses*
cholesky	28.36	0.11%	26.65	0.20%	-6.42%	1.81
fft	26.35	0.35%	26.35	0.42%	0.00%	1.20
lu	15.45	0.01%	17.78	0.04%	13.10%	4.00
radix	11.79	1.21%	11.85	1.84%	0.51%	1.52
barnes	24.54	0.08%	23.81	0.32%	-3.07%	4.00
fmm	24.09	0.10%	21.08	0.21%	-14.28%	2.10
ocean	15.76	0.36%	9.03	0.79%	-74.53%	2.19
radiosity	11.29	0.31%	12.16	1.09%	7.15%	3.51
raytrace	25.77	0.90%	24.26	1.36%	-6.22%	1.51
volrend	31.56	0.24%	30.35	0.42%	-3.99%	1.75
water-nsq	27.77	0.03%	25.89	0.28%	-7.26%	9.33
water-spa	25.55	0.01%	25.65	0.03%	0.39%	3.00
facerec	13.82	0.12%	13.81	0.73%	-0.07%	6.08
mpegdec	6.86	0.01%	8.93	0.02%	23.18%	2.00
mpegenc	10.91	0.001%	9.58	0.01%	-13.88%	10.0
Average					-5.69%	3.60

Table 13.7: Difference in speedup and MAP table miss rate between a system that uses 4K and 1K MAP table pages. As before, SPLASH-2 benchmarks were executed on a 32 tile system, while ALPBench benchmarks were executed on a 16 tile system.

benchmarks (with the exception of *fft* whose performance is not affected) lose performance. The average loss is 5.69%. One interesting thing to notice is that there is no correlation between the increase of the MAP miss rate and the impact on the performance. For *fft*, *ocean* and *mpegdec* the MAP miss rate approximately doubles, however the impact on performance varies between a critical slowdown to a significant speedup.

13.12. Effects of Sharing and Replication of Data

The Big Cache Memory model was developed to evaluate the performance gain that can be achieved by not having to sacrifice capacity due to replication. The overall results were, however, somewhat surprising. Figure 13.21 shows the speedups for the SPLASH-2 benchmarks for up to 16 tiles[3]. The figure shows that some benchmarks (*cholesky, fft*, *lu*, *ocean* and *raytrace*) scale very well; with *cholesky* even showing superlinear speedup. However, the performance of some other benchmarks does not scale very well. The performance gain of *radix*, *fmm* and *water-nsquared* starts to flatten out after 4 tiles and in case of *fmm* it even starts to decrease beyond 8 tiles. A similar observation can be made for *volrend*, *water-spatial* and (to a lesser extend) *barnes*, where performance does not increase when going beyond 8 tiles.

The explanation for this slightly surprising result lies in the number of conflict misses in the cache. The problem is that most private data is allocated to the same position within a page on all tiles. Thus there is also a fairly high chance that this data is also mapped to the same cache line. As long as the associativity of the cache is high enough, this mapping does not cause a problem. However, once the associativity is less than the number of tiles, too many addresses may be mapped to the same cache set, causing conflict misses. Thus, data is evicted from the cache before it can be accessed another time, greatly reducing the effectivity of the cache. This would explain the good performance of this model up to 4 tiles.

[3]These results were obtained with the FASTMEM system configuration and should not be compared with results that used the SLOWMEM configuration.

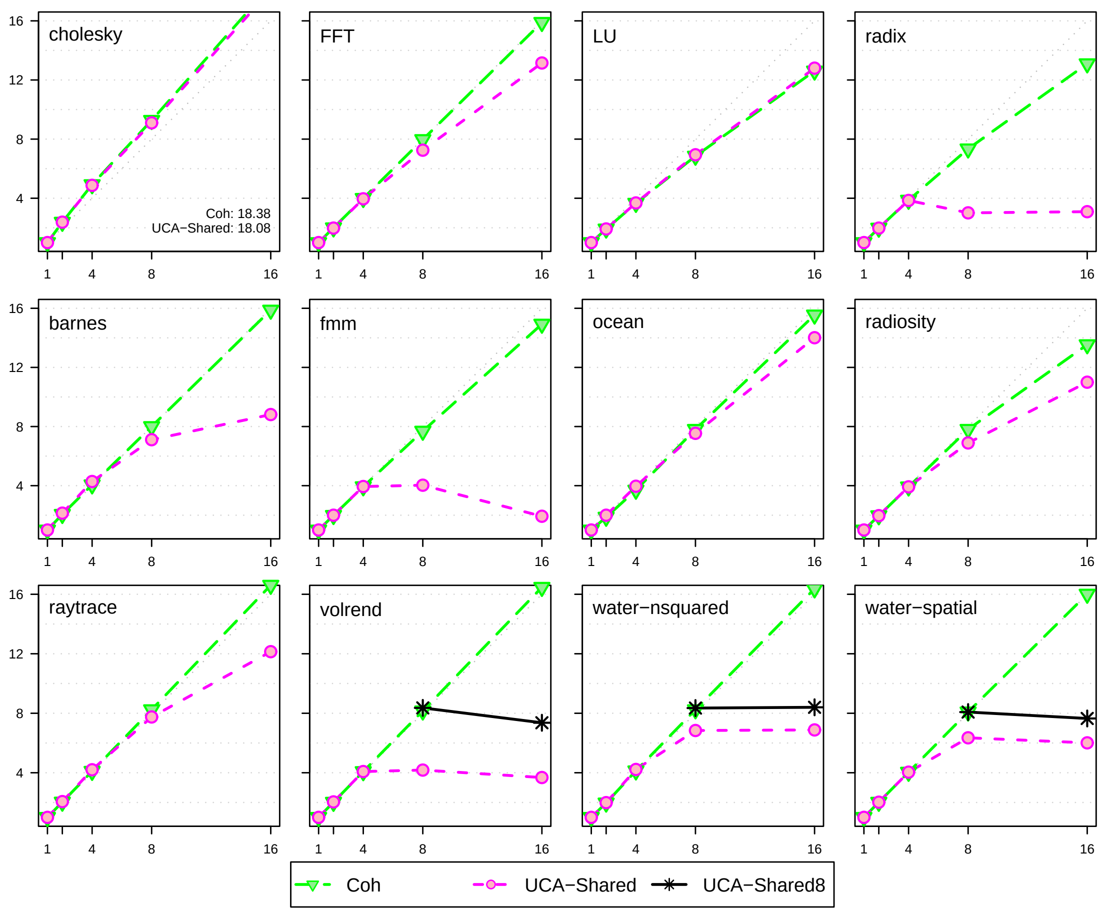


Figure 13.21: Speedups compared to execution of a single tile for *Coh* and *UCA-Shared*. The *UCA-Shared8* system is identical to the *UCA-Shared* system except that it uses an 8 way (instead of 4 way) associative cache. The x-axis shows the number of tiles. The y-axis shows the speedup.

Some experiments[4] were performed to confirm this theory by increasing the associativity for an 8 and 16 tile system to 8. This new system is referred to as *UCA-Shared8*. The effect of this change can be seen in figure 13.21: the three benchmarks that performed poorly on the 8 tile system achieve now a similar speedup as *Coh*. However, this time the scalability was limited to 8. Increasing the number of tiles to 16 did not result in any performance gains. While this shared cache is used as an L1, this thesis claims that these results are also relevant to large shared caches in general. The overall conclusion seems to be that for a shared cache the associtivity of the cache has to be increased as the number of connected tiles is increased. The other alternative would be a more sophisticated data layout that reduces the number of conflict misses.

[4]Due to the long simulation time, these experiments were limited to benchmarks whose scalability from 4 to 8 tiles was significantly reduced and had relative short simulation times. These benchmarks were: *volrend*, *water-nsquared* and *water-spatial*.

14. Related Work

Some related works has already been presented in the background information presented in chapter 4 and 5. As such, this chapter will solely focus on related works that have not been discussed so far.

14.1. Distributed Caches

The works in [CPV03, KBK02] explored how large (multi-megabyte) on-chip L2 caches can be efficiently organised to cope with increasing wire delays. The studies proposed policies for dynamically migrating heavily used lines to banks closer to the processor. However, those works focused on the uniprocessor case. The work in [BW04] explored the extension of the work in [KBK02] for CMPs. Unlike the work presented in this thesis, that work focused on a large shared L2 and assumed that L1 coherence is maintained through directories, but no detailed analysis of the costs of such coherence mechanism are presented. None of these works considers systems where the L2 caches are distributed along with each tile. In addition, the study was limited to only 8 processors. Also, the migration mechanisms proposed differ from ours in that they operate at the granularity of lines and require either broadcast tag lookups or a centralised partial tag store for the "smart search" mechanism.

Closer to this work, [CS06, CPV05, ZA05] considered the tradeoffs in organising the L2 caches in a tiled CMP where L2 is physically distributed along with each tile. Similarly to ours, those works considered the option of organising these distributed L2 caches as a logically single L2 cache. Those works differ from ours in the following ways: firstly, the L1 caches are private to each tile and allow replication of data, such that coherence is always required. No details about the implementation or the costs of such coherence mechanism are presented. Secondly, those works propose techniques, "controlled replication" and "victim replication" respectively, that allow for some degree of replication in the L2 caches that is at the line level and is controlled by the hardware. The work presented in this thesis emphasises simplicity and only allows a very restricted degree of replication that is totally controlled by the OS and, thus, forgoes hardware coherence mechanisms. Thirdly, those works only considered baseline schemes with static mapping of memory lines to L2 caches based on address, which is much more restrictive than the proposed baseline OS managed mapping mechanism. Finally, those works only considered smaller systems with 4 or 8 processors.

14.2. OS Controlled Migration

There have been previous work in OS directed page migration and replication in CC-NUMA (cache-coherent non-uniform memory architecture) environments, such as [VDGR96]. The policies in that work are based on measurements of cache misses and degree of sharing, which are done with hardware counters. The scheme is evaluated for remote to local access time ratios of 2 to 20 times. That work differs from ours in that CC-NUMA machines support fine-grain caching of memory lines, so that the page-level migration and replication is only necessary when the workloads overflow the private caches. In the proposed system, poor page placement always leads to remote cache accesses. Another important difference is that migration and replication of main memory pages may involve actual data movements with high overheads. Finally, there is a difference in the scale of the systems considered, which has a significant impact on the cost tradeoffs.

14.3. Alternative Hardware Cache Coherence Schemes

Apart from the hardware schemes that are discussed in section 3.1, 3.2 and 3.4, a few other hardware based schemes have been recently proposed that take advantage of advances in interconnect mechanisms.

Martin et al. [MHW03] offer an alternative coherence protocol based on token counting for machines that use a low latency unordered interconnect. Instead of assigning specific states (such as *shared*, *exclusive*, etc.) to each cache line, the state is expressed by the number of tokens a processor has for that cache line (e.g. at least one token). The study investigates a tree and torus interconnect. Using a tree network the protocol performs similar to a snooping protocol implemented on the same network. However, performance is greatly improved (about 50%) when the same protocol is run on the torus network. While the protocol was designed for low-latency networks, it still assumes a multiprocessor system. How this protocol behaves in a CMP environment is open to further research. Another constraint is the scalability of the protocol. While it performs well for the studied 16 processors, it still relies on broadcast mechanism that limits its scalability. The authors assume that scaling the protocol beyond 16 processors will require high bandwidth links between tiles and high throughput coherence controllers with low power consumption.

Marty and Hill [MH06] investigate the possibility of providing a cache coherence mechanism for processors that use a ring based on-chip network (like for example IBM/Sony/Toshiba's Cell [PAB$^+$05]). Ring based interconnects use point-to-point connections between two nodes. Thus, they offer a scalable design (unlike buses and crossbars) and simpler logic than packet-switched interconnects with arbitrary topology[1]. Ring based interconnects might offer a preferable compromise between these two network types. The problem with respect to coherence protocols is that ring networks do not provide the same strict ordering that buses do, even though they still offer some ordering of messages. They use the already mentioned token coherence mechanism to provide a suitable protocol for ring networks. By exploiting the ordering available on the ring network, they were able to simplify the token coherence protocol. While they show that their protocol performs better than a protocol that is similar to the one used in AMDs Opteron, it still suffers some stability issues resulting in dead-locks. As with some other studies before, this study limited itself to an 8 tile system. Thus, it is an open question, if a ring based bus or the protocol scales beyond this number of nodes.

Eisley et al. [EPS06] propose to consider the topology of the mesh based on-chip network in order to decrease the latency of memory accesses. In particular a get-shared request that

[1]As opposed to a network controller in a 2D mesh network, a network controller in a ring based network only has to decide if a packet has reached its destination or if it should be forwarded to the next node. It does not have to make a decision to which output port it has to forward the packet, nor has to deal with congestion where several incoming packets want to be routed to the same output port.

is sent to the home-node might be routed across a tile that has a copy of the requested cache line. Thus, the request can be already answered at this intermediate node. In order to keep track of sharers, the protocol constructs virtual trees that use the first requester of a cache line as root. Read requests that touch this virtual tree on their way to the home node are redirected to the root along the tree and can obtain a copy of the data there (or from any node in the tree that happens to possess the requested data). Write requests use a similar technique to speed up the invalidation of sharers. The protocol was tested with the 8 benchmarks from Splash-2 and verified using a model based analysis tool. However for this verification, in order to allow a tractable analysis, the number of concurrent requests was limited to 2. Also, testing was performed using a sequential simulator, which simplifies the protocol implementation significantly due to the absence of truly concurrent events. This is somewhat worrying, since the protocol has several conditions that can easily resolve into races and require deadlock recovery schemes. As for the complexity of the scheme, it seems that the home nodes have still a similar complexity as in directory based schemes. Furthermore, the complexity is increased since the information about the tree is distributed across several nodes. If any of these nodes has to evict this information, then the whole tree has to be torn down. The initial evaluation is performed on a 16 node system and shows an average latency reduction of 27% for reads and 41% for writes. The overall impact on performance is not shown in the paper. The implementation is scaled up to 64 tiles resulting in an average latency reduction of 35% for read and 48% for writes, however assuming the same 2MB private second level cache as in the 16 tiles implementation. This assumption might be mislead, since experiments on the 16 tile system with smaller caches show that performances decreases significantly. No experiments with reduced cache sizes were performed for the 64 tile system. Overall, this protocol offers a very interesting design point, but further investigation is necessary to understands its complexity and performance tradeoffs.

14.4. Hybrid Cache Coherence Schemes

Considering the design complexity and storage that is required to enforce cache coherence in hardware, schemes have been proposed to move some of the complexity and storage into software for various reasons.

The MIT Alewife machine [CKA91] used a hybrid cache coherence mechanism in order to address the storage requirements of a full-map directory. In such a directory an entry for a cache line requires as many bits as there are nodes in the system. This property potentially limits the overall scalability of a full-map directory. The LimitLESS directory protocol used in the Alewife machine, uses a limit vector of sharers: up to 4 sharers can be tracked in hardware. If another node requests shared access to the cache line, then a software handler is invoked. This handler copies the 4 sharers to software managed directory, adds the new sharer to the software directory as well, clears the 4 hardware entries and sets a special bit in the hardware entry to indicate that part of the state is now stored in software. The next 4 sharers can be added by the hardware as usual. If a processor requests exclusive access to a cache line, then the behaviour of the protocol depends on if the “software managed” bit is set. If the bit is not set, then the hardware behaves like in any other hardware directory scheme. If the bit is set, then another software handler is invoked. This handler then sends invalidation messages to all sharers (up to 4 sharers can be stored in the hardware directory and an unbounded number in software). The final action is to inform the hardware directory of the number of expected “invalidation-acknowledgement” messages. Thus, the hardware directory just has to count down until all invalidations have been received. It then can grant exclusive access to the requester. While this hybrid protocols addressed storage issues, it does not simplify the protocol as such: the hardware is still responsible to run a full-blown cache coherence protocol; just parts of the protocol are replicated in software in order to deal with storage constraints.

The Stanford FLASH multiprocessor [KOH$^+$94] is the successor of the DASH multiprocessor. Similar to the DASH multiprocessor the FLASH multiprocessor also has a dedicated protocol processor. However, while the DASH protocol processor is hardwired to a distributed directory cache coherence protocol, the FLASH protocol processor is user programmable. The reason for such a user programmable controller is that the controller can be better tuned to required communication patterns. For example, sequential consistency might not be required, but just release consistency instead. Or the problem is better solved with message passing rather than with shared memory to begin with. To summarise the main design considerations were hardware overhead and flexibility.

Grahn and Stenström [GS95] investigated a class of mostly software coherency protocols. Starting with a full blown distributed, software only directory protocol, they add pieces of hardware to remove the most critical overheads introduced by the software-only protocol. With their best strategy, which is a combination of all individual strategies, they are able to obtain 60%-68% of the performance of a hardware-only protocol. The main design considerations, similar to the one in this thesis, were the increasing complexity of directory based scheme. However, their design considerations differ in 3 points from the proposed scheme: first their starting point is a software-only scheme that is then extended by hardware to eliminate the most critical elements of a software-only scheme. Second, this design assumes a multi node system with associated latencies. And third, they implement a full blown distributed directory scheme.

The work done by Chaudhuri and Heinrich [CH04] also looks into ways of moving part of the coherence mechanism to software. However, their motivation is a different one: they first notice the complexity of implementing cache coherence protocols in hardware. Furthermore, they noticed that the memory controller is also moved into the main processor and modern processors are likely to support simultaneous multi-threading (SMT) [TEL95]. Thus, instead of adding another cache coherence controller to the main processor and increase its complexity, they decided to run the cache coherence protocol on a separate thread in software. The OS scheduler does not see this thread, thus it does not migrate to another processor nor is affected by context switches. On the other hand a new complexity arises: the protocol thread shares execution resources with normal application threads. Thus, there might be cyclic dependencies between these two threads (e.g., the protocol thread waits for a functional unit to become available, the functional unit waits for the result of a load operation that requires attention of the protocol thread). Unlike the proposed scheme, this mechanism runs a full blown distributed directory protocol in a dedicated hardware context.

Zeffer et al. [ZRKH06, ZH07] were concerned with the increasing complexity of coherence protocols and if such complexity is the correct design decision when moving to CMPs. They concluded that instead of adding more hardware support, a better decision would be to add only minimal hardware support and instead move the complexity to the software. The overall benefit would be shorter time to market and a more cost-efficient system. The first system, TMA Lite [ZRKH06], is based on an unmodified, single chip memory system and extends the processor to trap on certain stores and loads. The load trap is activated whenever a “magic-value” is loaded that indicates a coherence miss. As for stores, in order to not trap at every store, the processor is extended with a write permission cache (WPC) that stores addresses to which the processor has write permission. For an 8 node system with 2-way simultaneous multithreaded cores, TMA Lite lags about 25% behind a hardware coherent system.

The second system proposed, CRASH [ZH07], uses a normal processor, but augments the memory system in such a way that it can detect coherence violations/problems. Two bits are added to each cache line: one bit indicates that the node has read permission, the other indicates that it has write permission. Whenever a store or load operation is performed, these bits are checked by the hardware. If the required bit is not set, then an exception is signalled and the handler that resolves these violations is executed in software. For an 8 node system with 2-way simultaneous multithreaded cores, CRASH lags between 5% and 45% behind a hardware coherent system.

While these schemes are close in spirit to the one this thesis proposes, they do require more activity in the OS and trap handler than the proposed scheme. Such activity is greatly reduced in the proposed scheme due to two effects: First, the proposed scheme handles coherence at the coarser granularity of pages (4,096 bytes). TMA Lite supports granularities of 64, 128 and 256 bytes, while CRASH handles coherence at cache line granularity (in this case 64 bytes). Every first access for each processor will trigger an OS trap, in order to get shared or exclusive access. For the proposed scheme only the first access to a page (and if this first access happened to be a read access, then also the first write access) will trigger an OS trap, all subsequent accesses either hit in the MAP table or if they miss there, are handled by the fast PPC hardware handler. Second, the proposed scheme uses a small hardware extensions to support an extra level of indirection between virtual pages and tiles as well as to support remote cache accesses. Thus, only the processor’s first load or store to data in a page requires trap handler intervention and only the system’s first load or store to data in a page requires full OS intervention. Both TMA Lite and CRASH require OS intervention every time a unit of coherence is assigned to a new owner, or additional sharers are added.

Another important difference with all discussed systems in this section is that all those schemes focused on coherence mechanisms for multi-chip systems, while this work focuses on the problem of providing cache coherence for a single, large-scale, CMP.

14.5. Remote Access Mechanisms

Remote memory access schemes have been previously used in DEC’s Memory Channel technology [Gil96]. The system allowed a user level application to perform writes to memory on a remote node. All that is necessary is a normal store instruction to a virtual memory address. The system does not require any kind of OS trap or similar overhead upon this write. In this way it is similar to the proposed mechanism. However, its main limitation is that it can only perform remote write operations. Read operations have to be emulated, by telling the remote node with a remote write to send the data to the requesting node. Other minor differences include that not the whole address space can be used to perform remote writes. All nodes that are connected to the same memory channel network have to share a 512MB window at 8Kbytes granularity. Furthermore, Memory Channel does allow multicast and broadcast writes, which can be used to implement coherence protocols that use an update instead of an invalidation approach.

The M-machine is another architecture that allowed remote cache accesses [FKD$^+$97]. Remote accesses are performed by simply issuing load and store instructions. Unlike the before mentioned Memory Channel, the M-machine also supports remote read access. Unlike the proposed architecture, it uses a software trap based TLB mechanism to generate and handle the remote access. This results in that an uncongested read access to the neighbouring tile is 46 times slower than a local one. Remote write accesses are 37 times slower. Considering the significant higher access latency it seemed to be necessary to also support local caching. The mechanism is similar to the ones previously described in section 14.4 by Zeffer et al. [ZRKH06, ZH07]. The hardware only supports some mechanisms to keep track if a violation occurs; the actual handler is then run in software. However, it seems that it is for the programmer/compiler to decide which access strategy would be best suited in a given situation.

14.6. Fast Synchronisation Mechanisms

Sampson et al. describe a mechanism that also allows fast barrier synchronisation by stalling threads accessing specific resources [SGC$^+$05] instead of using a spin based approach. However, their scheme differs in two ways from the mechanism presented in this thesis: First, they stall the CPU using an L1 instruction cache miss. The 2nd level cache then is modified to not serve the L1 cache miss, until all threads have joined the barrier. Second, they don't use any special network to propagate barrier participation and release information. Instead, the L2 will start serving the instruction cache lines, once all cores have joined the barrier. The potential problem with this approach is that, unless cache lines can be broadcasted to all participating L1 caches, it requires a one by one release of each core. In our approach the release information is broadcasted along a tree, which should be significantly faster than the cache line method.

15. Conclusions and Future Work

15.1. Summary of Contributions

This thesis has been motivated by the current trend of an increasing number of cores on a single chip. Interconnects that are used in today's small scale CMPs will not scale to the number of cores that are assumed in these future CMPs. Instead, the use of a scalable, point-to-point interconnect will be required. However, such an interconnect will make the implementation of cache coherence mechanisms that are currently used in CMPs impossible.

In conclusion, the contributions of this research have been:

- This thesis presents an alternative solution for the cache coherence problem in a large scale, tiled CMP. The proposed solution is less complex than a hardware only solution, since complicated parts of the protocol are executed in software as part of the OS. Considering the reduction in complexity, the solution proposed in this thesis, which performs on average 16% slower than an idealised hardware solution, is an acceptable tradeoff.
- This thesis demonstrates how the basic mechanism of remote cache accesses can be extended to support controlled migration at barriers and read-only sharing.
- This thesis shows how to execute shared memory programs on the proposed architecture and investigate in detail the implications of the design decision on the implementation of locks and barriers.
- This thesis shows that a page based software only scheme is not able to deliver adequate performance on a tiled CMP for shared memory applications.

15.2. Future Work

The following areas are still worth investigating:

- The original RAW processor had an ILP mode in which the instructions of a single basic block are scheduled across several tiles. One interesting direction of research would be to combine this mode with the TLP mode described in this thesis. A previous study [ES03] shows that for small scale super-scalar CMPs, it depends on the application if it is more beneficial to have fewer threads but a wider issue super-scalar core, or more threads and a less wide super-scalar core. Considering the result of this study, it might be that for some applications (e.g., the ALPBench benchmarks) it is more beneficial to use fewer threads but instead combine several tiles that execute one thread together.
- The MAP table uses a selective invalidation protocol when it has to be invalidated upon a lock acquire. This invalidation protocol only uses local information for its decision and

does not access the global OS MAP table. A similar protocol, although more complicated, can be used to perform selective flushing at barriers. The key observation is again that cache lines that belong to owned pages (i.e., pages that the tile has read/write permission for) do not have to be flushed, since they are the most up to date copy there is. Instead, they can be kept in the cache, but have to be downgraded to shared. Such a scheme could be implemented in at least two ways:

1. Before a cache line is flushed, a lookup in the MAP table is performed. If this lookup confirms that the page is owned by the tile, then the cache line is not flushed. If the lookup misses, then it is assumed that the cache line is not on a page that is owned by the tile and no attempt is made to retrieve the page information from the global MAP table.

   While this scheme is relatively simple, it has two problems: first, in order to perform a MAP lookup, the virtual address of a page is required. However, the cache is physically tagged. Second, the cache flush logic requires an access to the MAP table on its critical path.

2. Each cache line also contains an extra *owned bit*. Whenever a cache line is brought into the cache, the owned bit is set according to the sharing state of the page. The owned bit might also be set later, with an action (such as a write to the cache line) that indicates that the tile now owns the page. At a barrier, cache lines with set owned bit will not be flushed, instead their owned bit will just be unset.

   Even though by adding this extra bit to each cache line the whole system looks like a traditional snooping protocol, the main difference is that all decisions can be made locally. The owned bit is never updated because of some remote action.

- Another area of improvement is to utilise knowledge about the application. For example, each thread probably uses some private data. This data obviously does not have to be invalidated, flushed or written back in comparison to the shared data. The question is now how the system can distinguish between shared and private data. Is further hardware support necessary, or can it implemented within the OS component of the proposed scheme? Another option is that the compiler can identify which pages should be best mapped to which tile in order to minimise remote accesses. Such a distribution could be computed by the application and loaded into the OS MAP table as part of the barrier process.

- The system proposed in this thesis simplifies the process of off-chip memory access by simply assuming a certain number of cycles to perform an off-chip memory access. In reality, such access also has to be performed using the mesh network. Thus, memory accesses from tiles that are closer to the centre of the chip would have to traverse several nodes before they can go off-chip. Similarly, requests by tiles that are close to the edge of the chip might experience delays caused by the requests coming from the centre. Another question would be which network these requests would use: the same network that is used for the remote memory accesses proposed in this thesis, or better a dedicated third dynamic network? Or as another option, maybe merge the dynamic networks into one network and use a virtual channel for each message type?

- One aspect that is becoming more and more important with future processor designs and is completely ignored by this thesis is power consumption. For example, while the power consumed by the relative simple cores is most likely acceptable, will the power that the on-chip network consumes negate this favourable power balance? Thus, in general it would be interesting to know if power issues make certain design decision made in this thesis less attractive.
- RAW exploits data level parallelism in loops and during stream based computation. These techniques work at very fine granularity and use RAW's blocking register communication for passing of data. The proposed scheme could be used to provide a similar way to exploit data level parallelism, however at a coarser granularity. For example, instead of using blocking register communication, several tiles could write the results remotely into another tiles cache. In another approach, the application could direct the OS to migrate ownership of a page to another tile at other times than a barrier. However, in this scenario it is the responsibility of the application to invalidate mappings to this page in all tiles that had a mapping. This should be relatively simple, if only one tile has a local mapping. This approach could be used to pass large blocks of data between tiles in a more asynchronous way (the receiving tile can read the information in a different order than the data is produced).

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ABB+07] Anant Agarwal, Liewei Bao, John Brown, Bruce Edwards, Matt Mattina, Chyi-Chang Miao, Carl Ramey, and David Wentzlaff. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Hot Chips - A Symposium on High Performance Chips*, August 2007.

[ACC+03] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. *SIGARCH Computer Architecture News*, 31(1):26–38, 2003.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12), 1996.

[ANMB97] Ernest Artiaga, Jose Ignacio Navarro, Xavier Martorell, and Yolanda Becerra. Implementing PARMACS Macros for Shared-Memory Multiprocessor Environments. Technical Report UPC-DAC-1997-7, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya – Spain, February 1997.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, May 1988.

[ASL03] Dennis Abts, Steve Scott, and David J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[BAB96] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[Bar00] Rajeev Barua. *Maps: A Compiler-Managed memory system for Software-Exposed Architectures*. PhD thesis, MIT Laboratory for Computer Science, January 2000.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.

[BKM+04] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BLAA99] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A Compiler-Managed Memory System for RAW Machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 4–15, May 1999.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software – Practice & Experience*, 18(8):713–732, 1988.

[Bur05] Doug Burger. Tiled Architectures. Course taught at HiPEAC ACACES Summerschool, June 2005.

[BW04] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, December 2004.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CCC+02] Călin Caşcaval, José G. Castaños, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Henry S. Warren, Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 311–322, February 2002.

[CF78] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CH04] Mainak Chaudhuri and Mark Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 124–137, June 2004.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable Cache Coherence Scheme. *ACM SIGPLAN Notices: ASPLOS-IV Proceedings*, 26(4):224–234, April 1991.

[CPV03] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, December 2003.

[CPV05] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 357–368, June 2005.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture - A Hardware/Software Approach*, chapter 5.5.3. Morgan Kaufmann, 1999.

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 264–276, June 2006.

[CTC+02] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Michael L. Scott. Multi-Level Shared State for Distributed Systems. In *Proceedings of the 2002 International Conference on Parallel Processing*, pages 131–140, August 2002.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[Eir96] Ásgeir Th. Eiríksson. Integrating Formal Verification Methods with a Conventional Project Design Flow. In *Proceedings of the 33rd Annual Conference on Design Automation*, pages 666–671, June 1996.

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

[ES03] Magnus Ekman and Per Stenström. Performance and Power Impact of Issue-width in Chip-Multiprocessor Cores. In *Proceedings of the 32nd International Conference on Parallel Processing*, pages 359–368. IEEE Computer Society, October 2003.

[FKD+97] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. *International Journal of Parallel Programming*, 25(3):183–212, June 1997.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[fre01] freescale semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, 2001.

[GFV99] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[Gil96] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM}Language Specification.* Prentice Hall, 3rd edition, June 2005.

[GLL$^+$90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[GMNR06] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to Intel Core Duo Processor Architecture. *Intel Technology Journal*, 10(2):89–97, May 2006.

[Goo83] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[Goo89] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, February 1989.

[GS95] Håkan Grahn and Per Stenström. Efficient Strategies for Software-Only Protocols in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 38–47, June 1995.

[GSSD00] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[GTK$^+$02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, October 2002.

[HAA$^+$96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[Hag07] Erik Hagersten. Personal Communication regarding the verification of the coherence protocol of Sun Microsystems' Enterprise Servers E3000, E4000, E5000 and E6000. July 2007.

[HC03] Mark Heinrich and Mainak Chaudhuri. Ocean Warning: Avoid Drowning. *SIGARCH Computer Architecture News*, 31(3):30–32, 2003.

[IEE95] IEEE. *IEEE Standard 1003.1c-1995 Thread Extension.* IEEE, 1995. Formerly POSIX.4a. Now included in 1003.1-1996. Also known as POSIX.1c.

[Int07] Intel. *Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence Specification Update*, July 2007. Document No: 313279-016.

[ISL96] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Understanding Applications Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 122–133, May 1996.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–226, December 1995.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *Computer*, 23(6):74–77, June 1990.

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

[KBH$^+$04] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 52–64, June 2004.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter 1994 Technical Conference Proceedings*, pages 115–131, January 1994.

[KCZ92] Peter J. Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KHS$^+$97] Leonidas I. Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michał Cierniak, Srinivasan Parthasarathy, Wagner Meira, Jr., Sandhya Dwarkadas, and Michael L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 157–169, June 1997.

[KOH$^+$94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March-April 2004.

[Kuh07] Marius Kuhn. Conception and Implementation of a Dynamic On-Chip 2D Mesh Network Simulator for a Multi-Core Chip. Master's thesis, Universität Duisburg-Essen, 2007.

[KZT05] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419, June 2005.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LBF$^+$98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, October 1998.

[LCED01] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A Simple Method for Extracting Models from Protocol Code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 192–203, June 2001.

[Li88] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 2, pages 94–101. Pennsylvania State University Press, August 1988.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[LLG$^+$90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, June 1990.

[LPSA02] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 111–122, November 2002.

[LSA$^+$05] ManLap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Proceedings of IEEE International Symposium on Workload Characterization*, pages 34–45, October 2005.

[LST$^+$06] Ana Sonia Leon, Jinuk Luke Shin, Kenway W. Tam, William Bryg, Francis Schumacher, Poonacha Kongetira, David Weisner, and Allan Strong. A Power-Efficient High-Throughput 32-Thread SPARC Processor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, volume 2, pages 2–4, February 2006.

[MB05] Cameron McNairy and Rohit Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March-April 2005.

[MH06] Michael R. Marty and Mark D. Hill. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 309–320, December 2006.

[MHW03] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[MLC$^+$92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–54, December 1992.

[MMG$^+$06] Avi Mendelson, Julius Mandelblat, Simcha Gochman, Anat Shemer, Rajshree Chabukswar, Erik Niemeyer, and Arun Kumar. CMP Implementation in Systems Based on the Intel Core Duo Processor. *Intel Technology Journal*, 10(2):99–107, May 2006.

[Ope05] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 2.5, May 2005. URL: http://www.openmp.org/mp-documents/spec25.pdf.

[PAB$^+$05] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, volume 1, pages 184–185, 592, February 2005.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Amin Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 160–169. Pennsylvania State University Press, August 1989.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[RR99] Radu Rugina and Martin C. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 77–90, May 1999.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[Sco96] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[SDH+97] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[SGC+05] Jack Sampson, Rubén González, Jean-Francois Collard, Norman P. Jouppi, and Mike Schlansker. Fast Synchronization for Chip Multiprocessors. In *ACM SIGARCH Computer Architecture News - Special Issue: dasCMP'05*, volume 33(4), pages 64–69, November 2005.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 291–203, December 2003.

[SN04] Robert C. Steinke and Gary J. Nutt. A Unified Theory of Shared Memory Consistency. *Journal of the ACM*, 51(5):800–849, 2004.

[SNKB00] Karthikeyan Sankaralingam, Ramadass Nagarajan, Stephen W. Keckler, and Doug Burger. SimpleScalar Simulation of the PowerPC Instruction Set Architecture. Technical Report TR2000-04, The University of Texas at Austin, Computer Architecture and Technology Laboratory, Department of Computer Sciences, University of Texax, Austin, 2000.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[SSP+04] Steven Swanson, Andrew Schwerin, Andrew Petersen, Mark Oskin, and Susan Eggers. Threads on the Cheap: Multithreaded Execution in a WaveCache Processor. In *Proceedings of the 5th Workshop on Complexity-Effective Design*, June 2004.

[TEL95] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[Til] Tilera Corporation. TILE64 Processor Product Brief. http://www.tilera.com/pdf/ProBrief_Tile64_Web.pdf.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (Part of ETAPS 2002, LNCS 2304)*, pages 179–196, April 2002.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, March 2002.

[TLM+04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Anant Agarwal, and Saman Amarasinghe. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, June 2004.

[UPC05] UPC Consortium. UPC Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[VDGR96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

[VHR+07] Sriram Vangali, Jason Howard, Gregory Ruhi, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyerl, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskotel, and Nitin Borkarl. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 98–99, 589, February 2007.

[VVA04] Manish Vachharajani, Neil Vachharajani, and David I. August. The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 195–206, June 2004.

[Wal91] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System*, pages 176–189, April 1991.

[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 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[Yea96] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[ZA05] Michael Zhang and Krste Asanović. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, June 2005.

[ZH07] Håkan Zeffer and Erik Hagersten. A Case For Low-Complexity MP Architectures. In *Proceedings of the Conference on Supercomputing*, November 2007.

[ZRKH06] Håkan Zeffer, Zoran Radović, Martin Karlsson, and Erik Hagersten. TMA: A Trap-Based Memory Architecture. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 259–268, June 2006.

Appendix

A. Liberty Simulation Environment

The Liberty Simulation Environment (LSE) is a framework written by David August et. al. [VVA04] to develop simulators. The framework is not meant to be a simulator by itself, but offers support for construction of one. Instead of building a single monolithic simulator, the main idea of LSE uses a divide&conquer approach: try to break your simulator into smaller building blocks, implement these and the use LSE to glue them together into a more complex simulator. LSE does not have any restrictions on what such a building block can be: it can be as simple as a single gate or as complicated as a super-scalar out-of-order CPU core. A building block can also be hierarchically composed from other LSE building blocks. For example, figure A.1 shows the components of a single tile and how these components are connected with each other. Using such a tile as a building block it is then very simple to connect several tiles in to a larger system (for example a 4x4 CMP, as shown in figure A.2.

Each module defines a number of ports that it can use to communicate with other modules. The LSE framework provides handlers that get invoked whenever some data becomes available at the port. It is the responsibility of the programmer to implement appropriate actions. The LSE framework also offers a feedback mechanism that allows the module to signal if it is able to consume the data. For example, an adder should only produce a result if both input operands are ready, otherwise it should not consume the already available operand.

This communication mechanism is the strong point of LSE. The main scheduler of LSE keeps track of all generated signals and passes them on to the handler in each module. That way the simulator designer does not have to figure out in what order to process the parts of his simulator in order to obtain correct simulation behaviour.

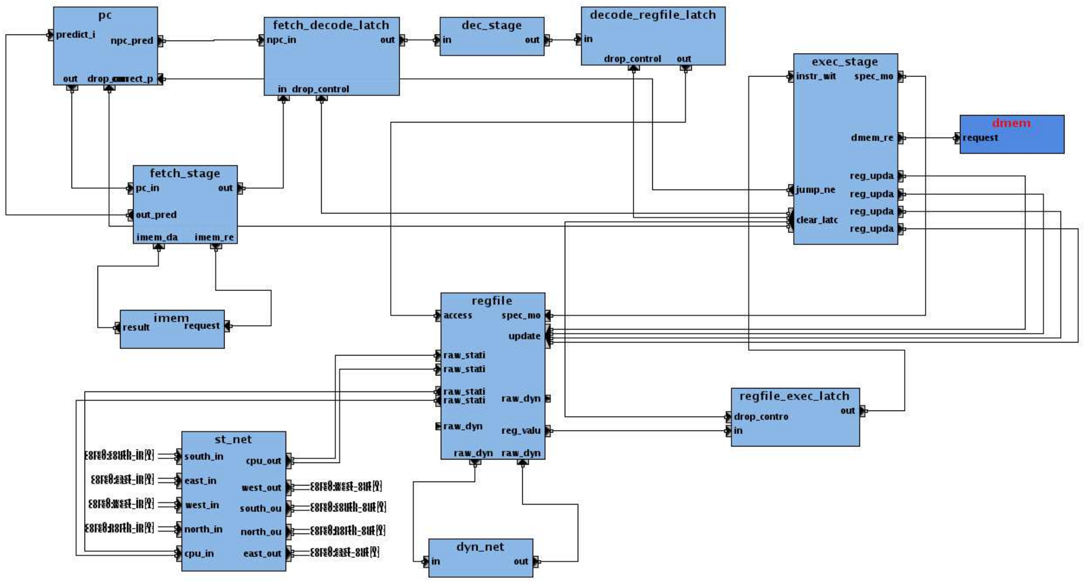


Figure A.1: Visualisation with Liberty of the pipeline of a single tile.

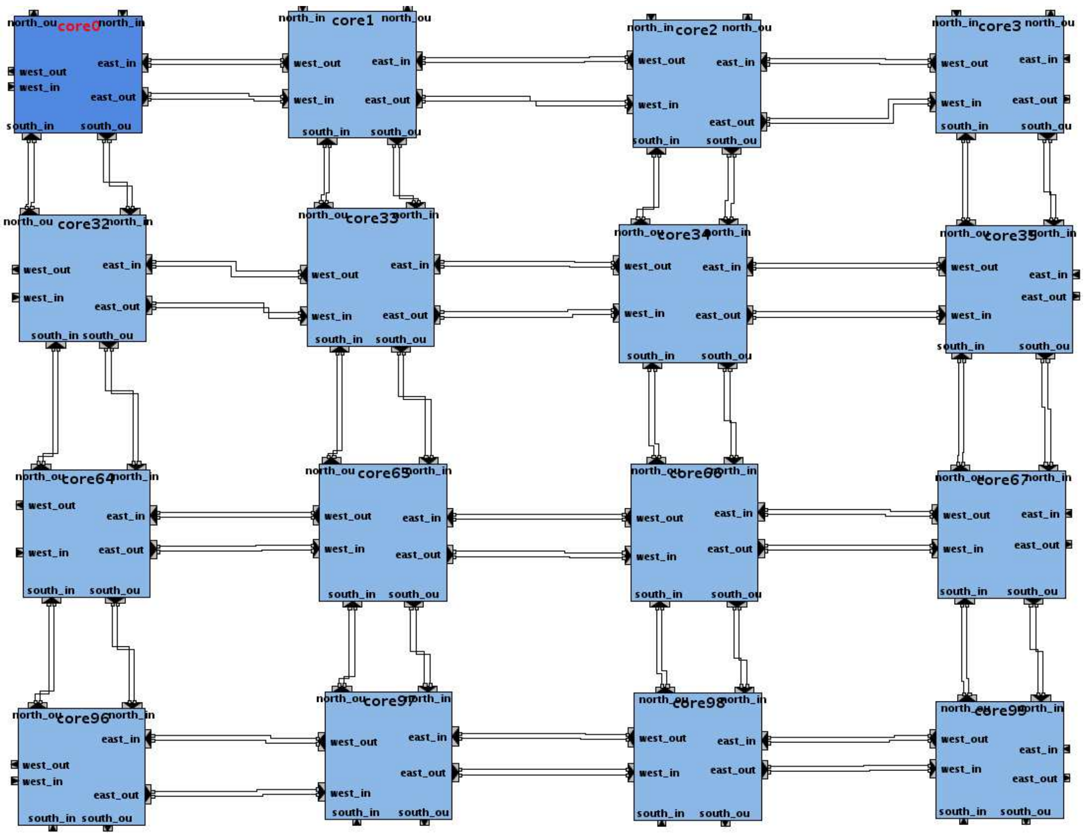


Figure A.2: High level visualisation with Liberty of a 16 tile system.

Despite these advantages LSE has two severe drawbacks that reduce the speed of the resulting simulator significantly:

- LSE replicates code for each instance of a module. This results into a large block of code that has to be executed in order to simulate a single cycle. Starting with 16 tiles, this block of code is so large that it no longer fits into the 2MB L2 cache of the host system. This causes a significant loss in simulation performance.
- The interaction between modules is mapped to function calls. These function calls are sometimes very simple, such as setting a flag. Since each instance of a module is created as an individual C file, most compilers will not be able to optimise this code using inlining. Thus the final simulator will execute a significant number of function calls that could have been avoided. Compilers (such as Intel's ICC), that are able to perform inlining across multiple source files, achieve a reduction in simulation time between 30% and 40%. Unfortunately, Intel's ICC fails to compile simulator configurations for more than 8 tiles.

LSE has been succeeded by UniSim that addresses these issues. Unfortunately, there is no automated process of converting LSE simulator specifications to UniSim specifications. Thus, it was not feasible to change the simulator infrastructure to take advantage of the better runtime behaviour of UniSim.

B. The RAWplus Simulator

The proposed architecture was developed with the goal in mind to have a tiles architecture that is equally well suited to support ILP, DLP and TLP. While the first two levels were already being handled by the original RAW processor, it was not able to support TLP. Considering these design constraints, it was necessary to have a simulator that offered the special features of RAW (like network linked general purpose registers) and the ability to plug in different memory modules. In particular the network linked register file caused a problem finding a simulator. Since most simulator packages do not support to change the register file, only one other option remained: instead of linking general purpose register with the network, use memory mapped registers. This might have been a feasible option, but it did not provide the latencies required to make RAW's ILP strategy feasible.

Thus in the end, I was faced with the task of writing the simulator mostly from scratch. This chapter describes some design decision and details of the implementation.

B.1. The Main CPU

The implementation follows the schematic of the RAW CPU as shown in [TKM+02]. The execution and write-back stages were merged into one stage, in order to keep the implementation simpler in supporting instructions with different latencies. While the original RAW supported the MIPS instruction set, I decided to use the PowerPC one[1]. The instruction latencies and dependencies were taken from PowerPC port of SimpleScalar [SNKB00]. This port also provided the instruction decoding functionality. Each stage of the pipeline is connected with a latch to the next stage. Each latch also supports flushing its contents instead of passing it to the next stage.

The CPU module contains an ELF loader that can handle static linked PPC binaries.

B.1.1. Fetch Stage

The fetch stage contains an instruction memory model, a branch predictor and a program counter. In its current implementation, the memory model assumes a perfect I-cache that will never miss. The branch predictor on the other side is virtually non-existing; it always predicts a branch as not-taken (even unconditional ones). Each instructions that is fetched, is augmented with a field of the predicted address of the next instruction. The branch predictor simply sets this field to current program counter (PC) plus 4. The execution stage will later use this field to handle a misprediction and initiate a rollback. The branch predictor itself is a LSE module and can be replaced without effort with a more sophisticated one.

[1]The main reason for this decission was that I were more familiar with this ISA. In the end I do not think that it made much of a difference, since both are RISC instruction sets.

B.1.2. Decode Stage

The decode stage performs a predecode of the instruction into input and output dependencies. Unfortunately, PowerPC instructions can have up to 5 input dependencies and 5 output dependencies. This made the resulting record of the predecoded instruction rather large. Since LSE uses a pass-by-value mechanism to pass information between different modules, it made the whole simulator rather slow. Hence, the decode stage is also responsible for allocating a decoded instruction record and just to pass a reference to this one between modules. It contained some further optimisation to only decode an instruction once; otherwise the same instruction would be decoded several times, every time the pipeline stalls.

B.1.3. Register File

The register file was the one problematic component that basically prevented the use of any existing simulator due to the four registers that are directly linked to the on-chip network. Also these network registers prevented the use of the common simulation technique that just computes when the results become availability of the input operands and functional units. The problem is that an operand can be delayed in the network due to another message that is generated after the arrival time of the operand has already been computed. The required bookkeeping and update mechanism would be fairly complicated and potentially a very big source of timing mistakes.

The register file is implemented in a very straightforward way. For all non-network registers the register file stores the following information: is the register ready to be read and what is the last instruction to update the register? The last information is necessary to avoid write-after-write hazards that could be possible if a register is written to twice without being read in between. The network registers are implemented as a 3 element queue (identical to the queue used in the static network). Reading from the register removes one element from this queue. Writing to the register involves checking if the network processor has space to receive the data and then sending the data. In order to ensure the correct order of data that is sent the register file has to remember the order of the instruction that generate the data, since the data has to be send in order.

B.1.4. Execution and Write-Back Stage

The execution and write-back stages are combined into a single module. The reason is that once an instruction has been accepted for execution, there is nothing that can prevent it from completing execution. Thus it is possible to compute the time at which the result will be available. This event is then inserted into a queue that is used to determine which instructions will be able to write-back their results to the register file. The queue is necessary to ensure that only one instruction writes back at any given cycle. Still, the write-back does not have to be successful in all cases: a write to a network register might fail, since the network processor has no space available to receive the data. In such a case the whole execution stage is stalled, it cannot accept a new instruction in this cycle. The execution stage also uses this stall mechanism to simulate the time it takes to handle a system call, without actually executing OS binary code.

A similar stall happens when the execution stage is not able to accept an instruction for execution. For example, a system call will only be executed, if the execution unit is currently not busy (i.e. there no functional unit, that is still trying to write a result back to the register file). Similarly, the memory fence instruction `eieio` will only accepted for execution, if no memory operation is still waiting for completion. Load instructions are another example that need a load buffer to support non-blocking memory access.

B.2. Memory System

The memory stage is normally located between the execution and write-back stage in the standard RISC pipeline. This stage is split between the execution system and the memory system. The execution stage performs the address computation (like adding a fixed offset to a base address stored in a register) and allocates a load buffer, if the memory instruction produces a result that is written back to a register. Using this information, it then constructs a memory request for the memory system.

The execution stage is connected with a single one-way connection to the memory module and sends requests via this channel. A request is either a normal memory request (read/write) or some special memory management command (like write confirm, see section 8.3). Memory requests that require a reply from the memory system, such as loads, also contain a pointer to the functional unit in the execution that will later produce the result. While this not the most elegant solution and also violates the encapsulation idea of an LSE module, this decision was necessary to keep simulation time manageable. Overall, the interface between the execution stage and the memory system is very easy and allows plugging in different memory systems.

B.2.1. The Remote Cache Access Module

This is the module that implements the scheme presented in section 7.

B.2.2. The Cache-Coherence Module

This is the module that implements the scheme presented in section 10.2.

B.2.3. The Software Distributed Memory Module

This module implements a simplified TreadMarks distributed shared memory system (which is discussed in more detail in Section 10.3). It manages similar data structures as were used in the original TreadMarks system in order to determine when diffs have to be created. The creation of diffs is simulated by stalling the execution stage for certain number of cycles. It uses the same mechanism to stall the execution that has been described in Section B.1.4 in order to simulate system calls. Locks and barriers are implemented with special 1-cycle system calls that stall the execution stage if the tile has to wait for the lock or at the barrier.

B.3. The Static Network

The static network processor is not implemented as a fully pipelined processor, as specified in the RAW specification. Instead it is assumed to have a perfect branch predictor, so that the fetch stage never misses and causes stalls in the pipeline. Furthermore the instruction memory is loaded as part of application loading process. The application itself does not have any way to modify this instruction memory. Since unlike the compute processors, the static network processors execute different code on each tile, it is necessary that each processor loads a different program. This is done by requiring that along with each executable for the compute processor several executables for the static network processors are present in the same directory. The convention for the filenames for these executables is that it is the same filename as the compute executable appended with `_stnXXX` where `XXX` is the number of the tile. These executables for the static network processors are not stored as a binary file, but instead in assembly language. This approach removes the need to implement an assembler and an instruction decoder for static network processor. Instead the "assembler" translates the code for the network processor directly into the simulator's internal data structures.

B.4. The Dynamic Network

The dynamic network is very seldom used in the evaluated systems. Exceptions are the starting of threads on other tiles (see section 9.1) and the PAUSE macros (see section 9.5), which are only used by *radix*. As such there are almost never more than one message active in the dynamic network. Thus, instead of implementing a detailed simulation of the dynamic network a simpler one was implemented: a model that ignores congestion between tiles (just congestion at the end point is correctly simulated). The arrival of the first word just depends on the distance between the two tiles. The dynamic network module decodes the header of the message and reserves an arrival slot at the destination tile. These arrival slots determine in which order messages sent to the same tile are received: they are received in the order of arrival slot allocation, i.e., a message that is sent later but from a closer tile cannot overtake another message. In addition, if the words of a message are not generated one word per cycle, then similar gaps are inserted on the receiving tile before they are transferred to the register file module.

Due to the very low number of messages in this network (for all benchmarks, except *radix*, 32 or less), the accuracy of the model was not verified with the more precise dynamic network model.

B.5. Collected Statistics

The simulator collects the following statistics for the total execution of the program (please note that some of these statistics do not apply to all memory models):

- Number of cache/TLB/MAP hits and misses.
- Average latency of a load request.
- Number of executed instructions.
- Number of cycles.
- Number of messages that experienced a delay caused by congestion in the send or receive queue.
- Average delay in cycles caused by congestion in the send and receive queues.
- Average distance of remote requests sent and serviced.

- Time spend in cycles to write back or flush the cache.
- For each cache flush I record the number of lines that belong to pages that were owned by the tile, shared with others or stale. Stale lines are the only lines that have to be flushed in order to ensure correct program execution.

In addition to these, it is possible to record the changes of some statistics during the course of the execution. The simulator currently only supports this feature for a single window with a set resolution. The window refers to an interval during the execution defined by a start and cycle number. The resolution defines after how many cycles the simulator generates a data point that describes the current state of the system. For example, it is possible to generate such a data point every 5,000 cycles starting in cycle 100,000,000 and stopping in cycle 200,000,000. The following statistics can be recorded in this way:

- Cache hit rate. This statistic can be sometimes misleading, since the simulator only records the ratio, but not how many cache accesses are performed during the resolution interval.
- Maximal and average number of elements in the send and receive queues.
- The total number of messages in the system. In order to simplify the computation of this value, I simply use the number of allocated messages. However, this number also includes messages that are currently waiting in a queue.

B.6. Fast Forward Mode

During initialisation, the simulator runs in fast forward mode, using just one tile and assuming no pipeline stalls due branch misprediction or not ready registers. Also, system calls complete in just 1 cycle. Memory accesses are still performed as in normal operation mode and cause delays. Thus, the memory system is in the same state after initialisation as if the simulator would have executed the whole time in normal mode. The switch between execution modes is triggered by special system calls within the application.

C. Sample Listings

C.1. Thread Creation

Initial Thread on "Slave Tiles"

```
void slave_spin() asm ("asm_slavespin");
asm (
     "asm_slavespin:\n"    /* initialise r0 and r1 on the network processor */
     "  li    24, 0\n"     /* with required constants */
     "  li    24, 1\n"
     "asm_slavespin_loop:\n"
     "  cmpwi 0, 2, 0\n"   /* check if register 2 has been setup? */
     "  mr    1, 25\n"     /* dynamic1 contains the stackpointer */
     "  mr    16, 25\n"    /* dynamic1 contains ptr -> threads[t_no] */
     "  mr    14, 25\n"    /* dynamic1 contains the method address */
     "  mr    15, 25\n"    /* dynamic1 contains arg */
     "  bne-  r2ok\n"
     "  li    3, 0\n"      /* ok we have todo a minimal pthread init. */
                           /* this init has only to be done once, afterwards */
                           /* register 2 is non-null */
     "  li    4, 0x20\n"
     "  bl    __libc_setup_tls\n"
     "r2ok:\n"
     "  mtctr 14\n"
     "  mr    3, 15\n"
     "  bctrl\n"
     "  nop\n"
     "  li      6,1\n"            /* load 1 into r6 */
     "slavespin_lk1:\n"
     "  lwarx   5,0,16\n"         /* load lock into r5 */
     "  cmpwi   0,5,0\n"          /* check if lock is taken */
     "  bne-    slavespin_lk2\n"
     "  stwcx.  6,0,16\n"
     "  bne-    slavespin_lk1\n"  /* stcwx unsuccessful => try again */
     "  tlbia\n"                  /* invalidate map to prevent access to */
                                  /* stale data */
     "  lwz     5,4(16)\n"        /* load  .count into r5 */
     "  stw     6,8(16)\n"        /* store 1 into .flag   */
     "  addi    3,16,12\n"        /* r3 -> .waiting[0]    */
     "  slwi    5,5,2\n"          /* r5 <<= 2 [get byte offset in .waiting[] */
     "slavespin_rel_wait:\n"
     "  cmpwi   0,5,0\n"          /* check if all waiting have been notified */
     "  beq-    slavespin_fin\n"
     "  subi    5,5,4\n"          /* decrement offset by 4 */
     "  lwzx    6,3,5\n"          /* load .waiting[r5>>2] into r6 */
     "  addis   25, 6, 0x0100\n"  /* add header to address */
     "  li      25, 0x4321\n"     /* write junk to network */
     "  b       slavespin_rel_wait\n"
```

```
    "slavespin_fin:\n"
    "   eieio\n"
    "   stw 5,0(16)\n"              /* release lock, r5 has to be 0 */
    "   b asm_slavespin_loop\n"
    "slavespin_lk2:\n"   /* lock taken, wait */
    "   lwzx     5,0,16\n"
    "   cmpwi    0,5,0\n"
    "   beq-     slavespin_lk1\n" /* lk is available */
    "   b        slavespin_lk2\n" /* still taken */
    "   nop");
```

Create a Thread on "Main Tile"

```
/* We need todo several things:
   - create a private stack for this thread
   - send a msg on the dyn. network to the tile that should run this
     thread.
 */

/* create a new stack, if none exists */
if(threads[thread_number].stack_ptr == 0) {
    void *tile_stack = mmap(NULL, 2097152, PROT_READ | PROT_WRITE,
                            MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if((int)tile_stack == -1) {
        fprintf(stderr,
                "Cannot allocate memory for thread %d's stack! [%d]",
                (thread_number), (int)errno);
        exit(1);
    }

    /* stack grows down, thus stack ptr points to end of allocated area. */
    /* ok, something is weird here.
       + 2MB would point just at the first byte after the allocation.
       + 2MB - 4 would point at the last word allocated
       + 2MB - 8 actully works. return addr is saved at sp + 4 (sp value
                 of stack pointer at method begin).
    */
    threads[thread_number].stack_ptr = ((int) tile_stack) + 2097152 - 8;
}
int tile_addr;

/* translate tile number into tile address */
switch(total_no_of_threads) {
    case 32: asm (
                 /* 8 x 8 and 8 x 4 processor layout */
                 "   rlwinm %[addr], %[no], 2, 21, 26\n"
                 "   rlwimi %[addr], %[no], 0, 29, 31\n"
                 : [addr] "=&r" (tile_addr)
                 : [no] "r" (thread_number) );
             break;

    case 16:
    case 8:  asm (
                 /* 4 x 4 and 4 x 2 processor layout */
                 "   rlwinm %[addr], %[no], 3, 21, 26\n"
                 "   rlwimi %[addr], %[no], 0, 30, 31\n"
                 : [addr] "=&r" (tile_addr)
```

```
                        : [no] "r" (thread_number) );
                    break;

        case 4:
        case 2: asm (
                        /* 2 x 2 and 2 x 1 processor layout */
                        "  rlwinm %[addr], %[no], 4, 21, 26\n"
                        "  rlwimi %[addr], %[no], 0, 31, 31\n"
                        : [addr] "=&r" (tile_addr)
                        : [no] "r" (thread_number) );
                    break;

        default: fprintf(stderr, "ERROR: Invalid number of processors!\n");
                    exit(-1);
    }

    threads[thread_number].flag = 0; /* mark thread as running */

    asm volatile (
         /* add remaining header info and write header to dyn network */
         "  addis 25, %[addr], 0x0400\n"
         /* write stack ptr to dyn network */
         "  mr    25, %[stack_ptr]\n"
         /* write *threads[] to dyn network */
         "  mr    25, %[thread_ptr]\n"
         /* write method address to dyn network */
         "  mr    25, %[func]\n"
         /* write method argument ptr to dyn network */
         "  mr    25, %[arg]\n"
         : /* no outputs */
         : [stack_ptr] "r" (threads[thread_number].stack_ptr),
           [func] "r" (ptr_to_function_to_run_as_thread),
           [thread_ptr] "r" (&threads[thread_number]),
           [arg] "r" (ptr_to_function_argument), [addr] "b" (tile_addr)
    );
```

C.2. Locks

Baseline Locks

```
int __tmp;

asm volatile (
     "1:\n"                  /* lock not taken, but somehow we didn't get it */
     "  lwarx   %[tmp],0,%[lock]\n"
     "  cmpwi   0,%[tmp],0\n"
     "  bne-    2f\n"
     "  stwcx.  %[one],0,%[lock]\n"
     "  bne-    1b\n"
     "  b       3f\n"
     "2:\n"                  /* lock taken by someone else, spin using
                                normal load, to avoid reservation
                                cancellation messages */
     "  lwzx    %[tmp],0,%[lock]\n"
     "  cmpwi   0,%[tmp],0\n"
     "  beq-    1b\n"
```

```
      "  b        2b\n"
      "3:\n"                  /* lock succesful acquired */
    : [tmp] "=&r" (__tmp)
    : [lock] "r" (&<lock_variable>), [one] "r" (1)
    : "cr0", "memory");     /* make sure no memory access is moved across
                               the barrier */
```

Locks Supporting Read-Only Sharing

```
int __tmp;

asm volatile (
      "1:\n"                  /* lock not taken, but somehow we didn't get it */
      "  lwarx    %[tmp],0,%[lock]\n"
      "  cmpwi    0,%[tmp],0\n"
      "  bne-     2f\n"
      "  stwcx.   %[one],0,%[lock]\n"
      "  bne-     1b\n"
      "  mapia\n"             /* invalidate map to prevent access to stale
                                 data */
      "  b        3f\n"
      "2:\n"                  /* lock taken by someone else, spin using
                                 normal load, to avoid reservation
                                 cancellation messages */
      "  lwzx     %[tmp],0,%[lock]\n"
      "  cmpwi    0,%[tmp],0\n"
      "  beq-     1b\n"
      "  b        2b\n"
      "3:\n"                  /* lock succesful acquired */
    : [tmp] "=&r" (__tmp)
    : [lock] "r" (&<lock_variable>), [one] "r" (1)
    : "cr0", "memory");     /* make sure no memory access is moved across
                               the barrier */
```

Unlock

```
asm volatile (
      "eieio\n"               /* make sure all loads/stores are complete
                                 before releasing the lock */
      "stw %[zero],0(%[lock])\n"
    :
    : [lock] "b" (&<lock_variable>), [zero] "r" (0)
    : "memory");            /* make sure no memory access is moved across
                               the barrier */
```

C.3. Barriers

Baseline Barriers

```
if(no_of_threads > 1) {
    int addr;               /* start address of the barrier code on the static
                               network processor */

    /* this assembly code does a very fast translation of the number
       threads in the barrier into an offset into the static network
       processor code. I used assembly since this computation can be done
       very efficiently if you can count the number of leading zeros in a
       binary number.
    */
    asm (
         "cntlzw %[addr], %[P]\n"
         "subi   %[addr], %[addr], 30\n"
         "neg    %[addr], %[addr]\n"
         "mulli  %[addr], %[addr], 24\n"
        : [addr] "=&b" (addr)
        : [P] "r" (no_of_threads)
    );
    addr += 0x10;

    asm volatile (
        "eieio\n"          /* make sure that all memory requests are
                              completed */
        "mr 24, %[addr]\n" /* assumption barrier code is at 0x00000010 in
                              the static net processor */
        "tw 1, 24, 24\n"   /* eat a value (the trap instruction cannot
                              trigger since r24 == r24) */
        "tw 1, 24, 24\n"   /* wait on r24 for the barrier being released */
        :
        : [addr] "r" (addr)
        : "memory"         /* make sure no memory access is moved across
                              the barrier */
    );
}
```

Barriers Supporting Migration

```
if(no_of_threads) > 1) {
    int addr;              /* start address of the barrier code on the static
                              network processor */

    /* this assembly code does a very fast translation of the number
       thrads in the barrier into an offset into the static network
       processor code. I used assembly since this computation can be done
       very efficiently if you can count the number of leading zeros in a
       binary number.
    */
    asm (
         "cntlzw %[addr], %[P]\n"
         "subi   %[addr], %[addr], 30\n"
         "neg    %[addr], %[addr]\n"
         "mulli  %[addr], %[addr], 32\n"
        : [addr] "=&b" (addr)
        : [P] "r" (no_of_threads)
    );
    addr += 0x20;              /* assumption barrier code is at 0x00000020
                                  in the static net processor */

    asm volatile (
        "  eieio\n"            /* make sure that all memory requests are
                                  completed */
        "  dcbst 0, %[null]\n" /* store all modified cache lines */
        "  mapia\n"            /* flush local map */
        "  mr   24, %[addr]\n" /* jump to barrier code */
        "  tw     1, 24, 24\n"  /* eat a value (the trap instruction cannot
                                   trigger since r24 == r24) */
        "  cmpwi 0, 24, 0\n"    /* Should this tile flush the global map? */
        "  beq+  1f\n"          /* branch, if this tile should not flush the
                                   global map */
        "  li     0, 777\n"     /* sytsem call 777: flush global map */
        "  sc\n"
        "1:\n"                  /* branch destination, in case this tile
                                   should not flush the global map */
        "  dcbf   0, %[null]\n" /* flush all cache lines */
        "  mr    24, %[addr]\n" /* second barrier */
        "  tw     1, 24, 24\n"  /* eat a value (the trap instruction cannot
                                   trigger since r24 == r24) */
        "  tw     1, 24, 24\n"  /* wait on r24 for barrier release */
        :
        : [addr] "r" (addr), [null] "r" (0)
        : "cr0", "r0", "memory" /* make sure no memory access is moved
                                   across the barrier */
    );
}
```

Code for Network Processor 3

```
move r0, $Ci1                    /* initialise r0 to 0 */
move r1, $Ci1                    /* initialise r1 to 1 */
jr $Ci1                          /* get method start addr from cpu */
nop                              /* and jump to method */
nop             route r0 -> $Co1, r0 -> $Wo1      /* 32: barrier for 2 tiles */
nop             route $Wi1 -> $Co1
b -5
b -6
nop             route $Si1 -> $Co1, $Si1 -> $Wo1 /* 64: barrier for 4 tiles */
nop             route $Wi1 -> $Co1, $Wi1 -> $So1
b -9
b -10
nop             route r0 -> $Co1, r0 -> $Wo1      /* 96: barrier for 8 tiles */
nop             route $Wi1 -> $Co1
b -13
b -14
nop             route r0 -> $Co1, r0 -> $Wo1      /* 128: barrier for 16 tiles */
nop             route $Wi1 -> $Co1
b -17
b -18
nop             route $Wi1 -> $Co1, $Ei1 -> $So1 /* 160: barrier for 32 tiles */
nop             route $Si1 -> $Co1, $Si1 -> $Wo1, $Si1 -> $Eo1
b -21
b -22
```

Code for Network Processor 11

```
move r0, $Ci1                    /* initialise r0 to 0 */
move r1, $Ci1                    /* initialise r1 to 1 */
jr $Ci1                          /* get method start addr from cpu */
nop                              /* and jump to method */
nop             route r0 -> $Co1, r0 -> $Wo1      /* 32: barrier for 2 tiles */
nop             route $Wi1 -> $Co1
b -5
```

```
b −6
nop                 route r0 -> $Co1, r0 -> $No1      /* 64: barrier for 4 tiles */
nop                 route $Ni1 -> $Co1
b −9
b −10
nop                 route r0 -> $Co1, r0 -> $Wo1      /* 96: barrier for 8 tiles */
nop                 route $Wi1 -> $Co1
b −13
b −14
nop                 route r0 -> $Co1, r0 -> $Wo1      /* 128: barrier for 16 tiles */
nop                 route $Wi1 -> $Co1
b −17
b −18
nop                 route $Wi1 -> $Co1                /* 160: barrier for 32 tiles */
nop                 route r1 -> $Co1, $Ni1 -> $No1, $Ni1 -> $Wo1, $Si1 -> $So1, $Ei1 -> $Eo1
b −21
b −22
```