The NUMAchine Multiprocessor: Design and Analysis

Robin Grindley

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
Computer Engineering Group
University of Toronto

Abstract

This dissertation considers the design and analysis of NUMAchine: a distributed, shared-memory multiprocessor. The architecture and design process leading to a working 48-processor prototype are described in detail. Analysis of the system is based on a cycle-accurate, execution-driven simulator developed as part of the thesis. An exploration of the design space is also undertaken to provide some intuition as to possible future enhancements to the architecture.

Shared-memory multiprocessors and parallel processing are becoming increasingly common not only in the scientific domain, but also as a replacement for mainframes in the field of large-scale enterprise computing. The shared-memory programming paradigm provides an intuitive view of memory as a globally shared resource among all processors. This is more familiar to programmers of uniprocessors than the alternative, message-passing. The distribution of memory across the system leads to Non-Uniform Memory Access times (NUMA), since processors have fast access to local memory and slower access to remote memories across the system network.The architecture contains features which attempt to hide or reduce the effects of this non-uniformity.

NUMAchine provides cache coherence in hardware, making it an instance of the general class of multiprocessor architectures called CC-NUMA (for cache-coherent NUMA). The system network in NUMAchine consists of a hierarchy of rings. We show how certain properties of rings allow for an efficient cache coherence scheme with reduced overheads in comparison to other CC-NUMA architectures. We use the simulator, which we developed as part of this project, to explore the NUMAchine design space in an attempt to discover how changes in various aspects of the architecture affect overall performance.

Acknowledgments

First and foremost, I would like to thank Zvonko Vranesic and Michael Stumm for their guidance and support, without which this thesis would certainly never have come to fruition. And of course I also have to thank Zvonko for teaching me some of the finer points of squash, even if he did take an inordinate amount of pleasure in thrashing me. To Michael, thanks for the proof that sleep is not actually a biological necessity.

To the Punks, well, what can I say? It's been a long, strange trip, and you guys were along for the whole ride. But now I guess it is time to get off the roller coaster, stagger around a bit and fall down.

Without a close group of friends to provide emotional support, I would have run out of steam long ago. To Dan, Gus, Kate, Andy and Stef, my thanks. And feel free to call the debt whenever you want.

Penultimately, a toast to the NUMAchine team. Who ever thought that by simply banging your head against a piece of hardware for years on end you could get it to work? Don't tell anybody.

And finally I would like to thank my parents. To my mother, whose patience with a son who seemed destined to be in school forever bordered on the beatific, my love and thanks. And to my father, who could not stick around for the end of the party, the best I can do is to promise that I will honour the memory.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

There will never be such a thing as too much computing power. As new levels of computing power become available, either applications grow in size and complexity, or new problems become feasible. The most cost-effective means of increasing computing power over the last 30 years has been to ride the technological wave. Increased density of integrated circuits, lower voltages and increased clock speeds have reduced the pressure on computer designers to apply novel architectural solutions to the performance problem. Keeping pace with Moore's Law [Moore 1975] has been an astounding technical feat for the semiconductor industry. But the basic architecture of a computer system has remained relatively unchanged: a single processor with one or more levels of cache, and a dedicated volatile memory.

The power behind Moore's Law is exponential growth. In the long term, this is also one of its shortcomings. The faster a finite system grows, the faster it will reach its natural limits. In the case of silicon technology, the hard upper limit is imposed by the speed of light. We can get a feel for this limit by supposing that with line sizes less than 0.1 micron, it will be possible to partition a large design into smaller clock domains each occupying at most $1mm^2$. Light propagation time over a distance of 1mm is approximately 3 ps, so after accounting for noise and timing margins, a conservative upper bound is about 100 GHz. Although electronics still has a fair bit of headroom, unless some revolution in device physics comes along, even entirely new technologies such as all-optical switches will be unable to break through this barrier. Given that the current state-of-the-art for processors in 1999 is around 1 GHz, a doubling in speed every 1.5 years would mean that the single processor will have hit its speed limit in about 12 years, and so we can fairly conservatively predict the tailing off of Moore's law by 2010 or 2020.

The Semiconductor Technology Association has a roadmap [Semiconductor 1997] which lays out major technological hurdles and indicates how the industry plans to overcome them. The end of the road for the 1997 version of the plan is at 2012, where they are predicting a 0.05 micron line size, 0.5 V power supplies and speeds of 3 GHz. However, they acknowledge that to achieve these goals a paradigm shift in lithography techniques will be necessary, which implies a large leap in production costs. In addition to this, system integration will become much more complex. The roadmap shows processors using 175 W, and at speeds over 1 GHz traces on a printed-circuit board start behaving like transmission lines, which requires a whole new level of design expertise. All of these factors point to a diminishing rate of return on performance from the lowest architectural levels. Not only will it become technologically infeasible to expect silicon (or whatever other materials win out) to provide future performance gains, but economically it will become more cost-effective to search for higher-level architectural solutions. Taking advantage of parallelism available at various different levels offers a solution. At the lowest levels, microprocessors use parallelism to execute multiple instructions in a single clock cycle. At the system level, parallel processing allows large tasks to be sped up by splitting them into a number of smaller tasks. It is this system-level parallelism which will be the focus of this dissertation.

Parallel processing systems have evolved significantly from the original mainframe, which offered a very rough form of parallelism through batch processing. The search for much higher levels of performance led to supercomputers, which although powerful were also extremely expensive. Parallel computing is just now entering its commoditization phase, following much the same trend as PC technology in the seventies; from an expensive technology available only to the few, it is now becoming a commonplace approach that will be not only available to, but usable by, the multitudes. In broadening its appeal, the field is also expanding into new areas. Until recently, parallel systems were almost exclusively the domain of large scientific applications. Nowadays parallel processing ‘servers’ are becoming increasingly popular for tasks such as database and on-line transaction processing. While commercial versions of these machines do exist, they tend to be much more expensive on a per-processor basis than comparable workstation technology and are more difficult to use. The challenge of making these systems cost-effective, efficient and usable still has to be met.

1.1 Goals of the NUMAchine Project

The primary goal of the NUMAchine project—at least from the hardware perspective— is to show that it is possible to develop a low-cost, scalable shared-memory multiprocessor that has good performance. Cost must be kept low because we feel that multiprocessors need to move into a commoditization phase. As PC technology has shown, commoditization leads to widespread adoption and also innovation, in a kind of feedback loop.

When we speak of scalability, we do *not* consider the goal to be a machine that can scale to a very large number of processors. Multiprocessors with four processors are now becoming commodity parts. In the next decade we expect to see this number move up to a few tens of processors. We thus set our scalability goal at about 100 processors. We consider our goal achieved if the performance and cost of the machine scale well up to this limit, and leave the problem of higher levels of scalability to future architects.

The trade-off between cost and performance is also important. Traditionally, multiprocessor research has focussed on performance as the main goal: much effort has been expended in trying to squeeze out a few more percentage points of parallel efficiency. If there is a lesson to be learned from the last decade's ascendancy of the PC over the workstation, it is that computer users do not care whether a machine is efficient or not. All they care about is whether the computer can do the job they want it to do, and do so cheaply. Thus, our performance goal is really subsidiary to our cost goal. We want to reduce system cost as much as possible without seriously degrading performance. As long as increasing the number of processors produces some gain, the user can decide whether the gain is worth the cost.

NUMAchine is also designed to be used as a platform for research into parallel operating systems, compilers and applications. NUMAchine provides low-level hardware monitoring functionality that can be used by software to analyse system performance. While monitoring is available in commercial machines, it is limited and its functionality is fixed. NUMAchine's monitoring, in contrast, is dynamically configurable. Also, since we designed the hardware, we have a better idea of what to look for (and where to look) when performance does not meet expectations.

To support both software and hardware research it is important that NUMAchine remain flexible. We provide this flexibility in the hardware by use of *field-programmable devices* (FPDs). The logic inside these devices is reconfigurable, making it possible to add or change

functionality at the hardware level, while the chip is still soldered to the board. This also greatly simplifies the debugging process.

These FPDs and the use of *commercial off-the-shelf* (COTS) parts also contribute to NUMAchine's ability to keep costs low.

1.2 Thesis Contributions

NUMAchine is a large, multi-year project. From initial high-level architectural discussions to the current working 48-processor prototype took about five years. The author was one of the principal designers on the project for the entire period. Most of the architectural choices were hashed out in lengthy team design sessions, and so benefited from many viewpoints. The author was solely responsible for creation of the architectural simulator, and the use thereof to validate these design choices. To varying degrees, the author was also involved with all other aspects of the design, including:

- Design specification and schematic capture.
- Design of five of the 36 FPD controllers.
- Design verification of all the individual boards, as well as the general system-level functionality. This also included design and validation of the hardware cache coherence scheme.
- Post-fabrication debugging (the author headed the debugging effort).
- C and Assembler hardware test development. This included the development of processor boot code and low-level device drivers.
- Forward-looking architectural studies, for which the author had sole responsibility. In this area, the set-replacement algorithm for adding associativity to the NC was contributed by the author.

Thus, in part or in whole, the author can lay claim to the following overall contributions from the NUMAchine project:

- A 48-processor working prototype of a distributed, shared-memory multiprocessor with integrated hardware cache coherence. We present evidence that this proof-of-concept machine displays good performance and scalability, while maintaining a very simple and low-cost architecture. The prototype is capable of running a parallel operating system and applications.

- Evidence to support the claim that a hierarchical ring network can be used to implement a multiprocessor with the features of scalability in both performance and cost.
- A remote-access cache, called a Network Cache, which improves the performance of programs in a machine with a NUMA memory system.
- Presentation of a hardware cache coherence scheme which makes use of the ring's ordering properties to achieve good performance. The coherence scheme is also tightly integrated with the Network Cache, and uses a novel lazy approach to directory maintenance which is shown to improve performance.
- Design and implementation of a cycle-accurate event-driven simulator, used for both architectural studies and prototype performance analysis.

1.3 Thesis Overview

This dissertation begins by giving a review of parallel processing architecture and systems in Chapter 2. As a particular instance of this class of machines, Chapter 3 describes the design of the NUMAchine multiprocessor. The chapter also contains a description of the NUMAchine architectural simulator, which is the main tool used for both the initial prototyping and exploratory studies. Chapter 4 makes use of the simulator to analyse NUMAchine's overall performance, in addition to analysing various aspects of the architecture described in Chapter 3. These results are compared to results obtained from the actual hardware. Chapter 5 undertakes an investigation of NUMAchine's design space, with the goal of determining the most effective areas of the architecture to tune in order to increase performance. Finally, Chapter 6 summarizes and presents the major conclusions, along with a description of possible future work and trends in the field.

CHAPTER 2

Background

There have been many approaches to parallel computing over the years. This chapter begins with a brief history and description of parallelism in general, then moves on to descriptions of certain key concepts in parallel systems architecture which are necessary to understand the chapters that follow. It concludes with brief reviews of a number of existing commercial and experimental parallel systems, in order to give a feel for the state of the art.

The field of parallel computing is extremely broad, covering many different aspects of computer architecture, from the lowest levels of a processor's internal architecture to networking and memory system design. The material presented here is only a brief synopsis. In particular, a firm knowledge of uniprocessor architectures will be assumed. (The standard reference in this area is [Patterson 1998]). A good compendium of current knowledge on parallel computer architecture is [Culler 1999], which goes into much greater detail. A list of appropriate references is given in each subsection. (If no reference is given for a term or concept, then it can be found in Culler.)

2.1 An Overview of Parallelism

2.1.1 Early History

The early days of computing actually used a simple version of parallelism by default, as embodied in the *mainframe* systems of the day. These were large batch-processing systems, where many users gained access to multiple processors by submitting their jobs to be processed by whichever processor first became available. The number of jobs was large and the number of processors small, so there was no point in splitting individual jobs into subtasks

and moving away from very coarse-grained parallelism. In addition there was no experience with parallel data structures or algorithms, so there was no support available for the programmer even if such a feature were available in the hardware. The focus in mainframes was on *throughput* and *reliability.* Throughput in this context meant completing as many jobs per unit time as possible. Reliability meant providing enough fault tolerance that the machine would not crash, losing long-running jobs. Mainframes spared no expense to achieve these goals, making them affordable only by large organizations.

As individual processors became faster and cheaper, the advent of the PC became possible. Users of batch-processing systems found it frustrating to have so little deterministic control of their jobs, not knowing how many hours or days later they could expect completion and nobody liked sharing computing resources with others, but the machines were too expensive to support any other model. The mainframes were also very weak at anything involving user interaction with the computer. In the era of the punchcard, the programmer would write and debug code by hand, making completely certain that the program would run as expected before going to the trouble of submitting the job to the actual computer. Processor cycles were too expensive to be wasted on such actions as debugging a program or editing a document. The PC made it economically feasible to have processing power dedicated to one user, not only for running jobs which would have a much more predictable execution time, but also for providing a dedicated single-user interface directly to the machine.

Since programming became an art no longer of the specialist but of the masses using PC technology, it is not surprising that the programming paradigm of one program for one processor became the ‘normal’ way to write code. In addition, since the power of the chips was growing exponentially, there was little need for the average programmer to look further. The advent of Fortran and its eager adoption by the scientific and engineering communities changed this scenario by creating a new class of users and a new class of applications. Scientists found that computers could be used to find approximate numerical solutions to large problems such as systems of partial differential equations which had no tractable analytical solution. The size of the problem was limited solely by the amount of computing available; problem size could be scaled up by increasing the numerical accuracy or the size of the system under investigation.

Parallel computing had by this point found a specialized niche. The machines were typically custom-designed using very exotic architectures and technologies, hence they were

extremely expensive. This limited their use to large research labs and the military. These *supercomputers* used parallelism at a low level through means of vector operations, where a single instruction could perform the same operation simultaneously on arrays of numbers, which is a common feature of scientific programs. Dedicated maintenance staff and programmers were the norm, making the entire field very specialized.

The next wave in parallel research, *massively parallel processing* (MPP), focussed on achieving supercomputer performance with less exotic technologies by means of high-level parallelism. Researchers looked at designing machines with thousands of processors, hoping for performance through shear quantity. It turned out to be a very difficult problem to design good architectures that would work for any arbitrary number of processors. At the same time, the PC market was driving single-processor performance ahead at an exponential rate, often through commoditization of supercomputer architectural techniques. The upshot for MPP research was that it became feasible to achieve supercomputer performance with only a few hundreds of processors. In addition, it was realized that there would always be a handful of users needing supercomputers, but that these users were a special group that could afford to pay a hefty premium. The push to move parallel processing into the mainstream has shifted attention to *multiprocessor* systems containing tens to hundreds of commodity processors.

2.1.2 Low-level Parallelism

Parallelism allows higher levels of performance for a given clock rate, so it is complementary to raw circuit speed. Modern processors internally take advantage of fine-grain parallelism in two ways. The first is *pipelining* which works by breaking a task (in this case a single machine-level instruction) into sub-tasks (or stages), each of which can execute concurrently as long as they do not depend on each other. A typical processor pipeline would include instruction decode, operand fetch, instruction execute and result storage stages[1]. The stages can be completed more quickly since they are smaller and simpler, requiring less complex logic to implement. The second type, termed *superscalar*, utilizes multiple pipelines operating in parallel. Here the ‘task’ is viewed more as groups of individual instructions, and the sub-tasks are the individual instructions which can be launched into different pipelines independently. If there are dependencies, then hardware must provide *interlocks*, which is a form of

1. Intel’s 486DX processor has 5 stages. Newer processors use deeper pipelines (called *superpipelines*). Intel’s PentiumPro, for example, uses 14 stages. See [Burd 1999] for other examples.

synchronization allowing the stages or pipelines to stall until the conditions causing the interlock are clear. Note that VLIW (Very Long Instruction Word) processors are really just a variant on the superscalar design, where the choice of which instructions can be executed in parallel is made statically by the compiler instead of being done on-the-fly in hardware.

2.1.3 Higher-level Parallelism

While previously discussed uses of parallelism have been effective, they have a high cost in terms of hardware design time and complexity. In addition they rely on trying to exploit parallelism dynamically at the lowest level, which means that they have a very myopic view of the task at hand, and can only do a very simple local analysis of the available parallelism. (VLIW compilers can afford to broaden their view somewhat, but the available parallelism is still very fine grain.) Finally, the low level of these solutions means that they can only make use of generic types of parallelism which are common to all programs.

Most computational tasks contain parallelism at various different levels. At the lowest level we have instructional level parallelism, which was discussed above. This is particularly suitable for a processor which uses a reduced instruction set (RISC). The instructions are small and simple with few, if any, side-effects, which makes it feasible for a scheduler to make decisions in hardware at the internal speed of the processor. At a higher level there is thread-level parallelism, in which a large computation is split into smaller computational threads, each of which can run concurrently, either by time-multiplexing on a single processor or using a number of independent processors.

Our work focuses on the use of this higher level of parallelism in a multiprocessing environment. The modern commodity processors used in NUMAchine take advantage of the forms of lower-level parallelism described in the previous section. In most cases the two levels are independent, but occasionally lower-level design choices can have an impact on the high-level design. Such issues will be touched on in Chapters 4 and 5.

2.1.4 A Parallel Taxonomy

One of the first attempts to present a taxonomy for the different classes of parallel architectures was provided by Flynn [Flynn 1972]. This describes the relation between the instruction stream and the data stream and whether or not each of these take advantage of parallelism.

With both instructions and data nonparallel, we have the traditional uniprocessor mode, termed Single-Instruction/Single-Data or SISD.

When a single instruction stream works on multiple data streams it is called SIMD (Single-Instruction/Multiple-Data). An example of this type of machine could involve a single controller coordinating the activity of numerous data-processing engines, which all do the same operation in lock-step but on different data items. (This type of SIMD machine is also called a *systolic array,* drawing upon the analogy of the human heart which pumps blood through the arteries at each step.) The advantage to this approach is that the data engines can be extremely simple and cheap, making large levels of parallelism feasible. In addition, because of the lockstep operation there is no need for synchronization. Other SIMD architectures include vector processors such as CRAY-1 [Russell 1978] and the recent trend to add 'multimedia' instructions to modern processors, an example being SUN's UltraSparc VIS [SUN 1997], which stands for Visual Instruction Set. In image processing, each pixel of the image can generally be processed independently. Since pixels are usually stored as four individual bytes of colour and other information, a single 32-bit load can bring in an entire pixel and a single operation can be performed on all four bytes in parallel.

The final category is Multiple-Instruction/Multiple-Data (MIMD) which applies to shared memory processors. (A MISD architecture does not make physical sense.) In MIMD, processors each fetch their own instruction and data streams independently. In the case that all the instruction streams correspond to the same program, the term Single-Program/Multiple-Data (SPMD) is also used. MIMD machines naturally support thread-level parallelism. The standard approach is to associate one thread with each processor and divide the data set for the program amongst the threads. Note that MIMD and SIMD are not mutually exclusive. A processor in a MIMD system can still take advantage of opportunities for SIMD parallelism in its local portion of the data.

2.1.5 Limits to Parallelism

No system is ever perfect, and parallel computers suffer from various different impediments to efficiency. The most fundamental notion in parallel systems work is that of *concurrency.* Two or more operations are concurrent if they execute simultaneously. If two operations depend on each other in such a way that one must execute before the other then they can not run concurrently, and the available concurrency is reduced. For example, when summing up a list of

numbers, the list can be divided arbitrarily into sublists, and the summation of these sublists can be carried out at the same time. The available concurrency in this case is equal to the number of sublists. The initial division of the list into pieces may allow concurrency if care is taken. Each of the subtasks which calculates a partial sum would need access to the global list. If the same algorithm is used by all the subtasks, *and* it can be proven that the algorithm can never result in two subtasks choosing the same item, then these subtasks can also run concurrently. The alternative would be to have one designated 'master' subtask responsible for the division of the list, and only after the partitioning allow the other subtasks to start summation. In this case, all of the 'slave' subtasks must wait for the master to finish the partitioning, allowing no concurrency at all during the partitioning phase.

The first job in approaching the parallelization of a given problem is to determine what concurrency is theoretically available, which puts an upper limit on the amount of parallelism which can be achieved. For the case of summation given above, for a list containing N numbers the concurrency is $\lfloor N/2 \rfloor$ if we consider addition to be a binary operation that requires at least two operands. (The use of the greatest lower bound is due to the case where N is odd, and one subtask gets three operands instead of two.) If we have enough computing elements available, we can assign one partial sum to each, with each element requiring at most one (or two for N odd) operations to do the sum. In this case we have enough parallelism at our disposal to efficiently use all of the concurrency, and the running time of the partial summation is $O(1)$.

While calculation of the partial sums has been sped up as much as possible, our parallel algorithm now must accumulate all of these partial sums into the global sum. Having a single subtask add up all the partial sums would take $O(N)$ time, and thus would be no better than the single processor case. A more parallel approach is to form the partial sums into a binary tree, and designate a processor at each level of the tree to add the partial sums of the children, which would result in an $O(\log N)$ running time. At each level in the tree, processors must wait for their children's sums to be ready, requiring synchronization. Thus, not only is the parallel algorithm different from the sequential one, but there is overhead associated with the process of parallelization. This overhead can come either from synchronization or extra code needed in the parallel algorithm. Note that for this reason the best sequential algorithm and the best parallel algorithm running on one processor are not the same.

When measuring the performance improvement when running on N processors, the stan-

dard metric used is the *speedup*, which is loosely defined as the total program execution time on a single processor divided by the time for *N* processors. From the foregoing discussion, it is clear that we also have to specify whether the single-processor execution time is for the sequential or parallel version of the program. Both measures have their uses. Comparing to the best sequential algorithm indicates whether parallelization is useful at all. If a parallel algorithm adds substantial overhead and has poor speedup, then the parallel program may never be able to outdo the sequential version by a large enough margin to justify parallelization. Normally though, it is the case that the overhead involved is not significant, and it is usually clear from the outset whether or not there is enough concurrency to justify parallelization. Thus, given that the user knows she will be running the parallel algorithm, she is more interested in how much improvement she can get. The version of speedup using the parallel algorithm to measure both the single-processor and *N*-processor execution times is the more common, and is the one that will be used from here on.

The next question to ask is what fundamental limits exist on the speedup which can be achieved. The basic result here is Amdahl's Law [Amdahl 1967] which states that total speedup achievable by any parallelization of an algorithm is limited by the parts of that algorithm which are inherently sequential. That this is true is obvious by considering a program with execution time T_1 on a uniprocessor, which can be partitioned into two sections: code that is inherently sequential, taking time T_{1S}, and the rest of the program (assumed to be fully concurrent) with execution time T_{1P}. Thus $T_1 = T_{1S} + T_{1P}$. In the best case, we assume that the parallelizable section can make perfect use of the concurrency on *N* processors, so the total execution time for a given *N* is:

$$T(N) = T_{1S} + \left(\frac{T_{1P}}{N}\right) \qquad \textbf{(EQ 2.1)}$$

Thus the limit as *N* gets large is T_{1S}. This means that no amount of parallelism can speed up a given application beyond a certain point, which would seem to indicate that there is a limited usefulness for parallelism, as every program will have some sequential portion. For a while, researchers regarded this conclusion to be true, but we can write the equation for the *speedup* ($T(1) / T(N)$) as:

$$Speedup = \frac{(T_{1S}/T_{1P}) + 1}{(T_{1S}/T_{1P}) + \frac{1}{N}} \qquad \textbf{(EQ 2.2)}$$

What this equation shows is that the knee of the speedup curve is reached once the value of $1/N$ becomes comparable to T_{1S} / T_{1P}. Thus if we assume that the ratio of sequential to parallel code is some non-infinitesimal number, then we cannot reach large speedups. But for many realistic problems, this ratio can actually be made arbitrarily small by increasing the amount of work done in the parallel section. This will usually increase the work in the sequential section as well, but this is not a problem as long as the time complexity (as a function of the amount of work) for the sequential section is lower than that of the parallel section. (For example, if we define some arbitrary parameter, W say, as a measure of the amount of work, then if T_{1S} is $O(W)$ and T_{1P} is $O(W^2)$ we can make the ratio arbitrarily small by choosing W large enough.) If these conditions hold true, then it is possible to achieve large speedups.

2.2 Architectural Aspects of Parallel Systems

2.2.1 The PRAM model

When reasoning about computer architecture, it is important to have in mind a model to act as a framework on which to test out ideas. For uniprocessors, the basic model is from von Neuman, and imagines a processor attached to a memory module, from which it fetches both code and data. The parallel analog to this is called the PRAM (Parallel-RAM) model [Cormen 1989]. As shown in Figure 2.1, the PRAM model assumes some number of processors which share access to the same memory, with each read or write taking one cycle. The basic model allows simultaneous accesses by multiple processors to the same memory location for either reading or writing. It is assumed that concurrent writes store the same value to a given location, or if they do not then at least some mechanism is in place to deterministically choose which value is ultimately visible to future reads. The advantage to such a simplistic model is that it is mathematically tractable. Enhancements to the model disallow concurrent accesses for reads, writes or both to model slightly more realistic systems, with the drawback that these models quickly lose their tractability. None of the PRAM models assign any cost to interprocessor communication, however it *is* assumed that processors have some method of synchronizing their operations, albeit with zero overhead. The typical methods of synchronization include mutual-exclusion locks, which guarantee that at most one processor can own the lock

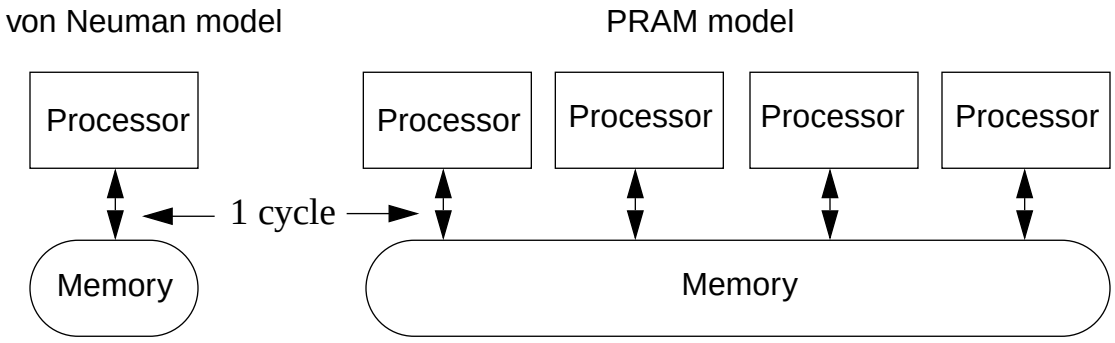

FIGURE 2.1: Von Neuman and PRAM memory models. Both the von Neuman and PRAM models assume zero latency to access memory for either reads or writes. Implicit in the PRAM model is synchronization, which is necessary for program correctness. Both models assume infinitely fast memory and communication, with no contention.

at any time, and barriers, which guarantee that all processors have reached a certain point in their computation before allowing any processor to proceed.

While simple, the PRAM model can still make predictions about parallel performance. The program executed on the processors will contain the overhead due to parallelization, as well as any overhead due to synchronization (e.g. when two processors need to acquire a lock at the same time, the second will be delayed until the first releases it.). If the amount of work is not divided equally between the processors, then some processors may have to wait for others at barrier points, which is termed a *load imbalance*, and contributes to the loss of parallel efficiency. The speedup as measured under the PRAM model is thus the best that any real system could achieve, and is called the *algorithmic speedup*.

Other models which take into account communication costs such as latency, bandwidth and overhead have been proposed. Models such as bulk-synchronous parallel (BSP) [Valiant 1990] or LogP [Culler 1993] extend the PRAM model in an attempt to make it more realistic. While interesting theoretically, these models take no account of real network topologies and assume fixed constant delays.

In real parallel systems, the details of the network have a significant influence on overall system performance. A simple model of communication includes three parameters: latency,

bandwidth and overhead. Latency is the total time between initiating a request and receiving a response. In a uniprocessor, for example, the latency to load a variable consists of the time between the processor executing the instruction, and the result becoming available in the processor's register. For a cache hit typical latencies in a modern 500 MHz microprocessor are on the order of 10 ns, while a cache miss and the resulting read from a local memory might take 200-300 ns. Bandwidth refers to the maximum number of bytes per second that a communication channel can move between a source and a destination. (Note that the source and destination can consist of groups of senders and receivers, in which case the bandwidth is the sum of the bandwidths over all channels which can carry data simultaneously). In a chain of point-to-point connections, bandwidth is usually taken to mean the bandwidth of the slowest link. Overhead represents any fixed delays inherent in communication, and contributes directly to latency. As an example, data may require compression before being transmitted over a channel, which increases overhead (and latency) in direct proportion to the compression time.

The relationships between these parameters is subtle, meaning that simple network models are not very accurate. Latency, for example, usually decreases as bandwidth is increased. If, however, the bandwidth is increased in one link of a communication chain such that some other link becomes the slowest in the chain, overall latency will not decrease as much as expected. Increasing bandwidth could also cause overhead to increase. Take a protocol processing engine which is fed by a communication channel. If channel bandwidth is increased too much, then the processor may reach a point where it cannot handle the incoming packets fast enough, at which point the channel will back up and the network will become congested, leading to an overall *increase* in latency. Bursty traffic patterns are another major source of congestion[2]. Due to its highly nonlinear behaviour congestion is not only difficult to model, but can also have significant effects on performance even if the time-averaged level of congestion is low.

In this thesis we avoid these communication modelling issues by using a detailed cycle-accurate model of the entire system in our simulation environment, including not only latency, bandwidth and overhead, but also network congestion. We will make use of the basic PRAM model to measure the algorithmic speedup for the purpose of comparison with experiments.

2. Note that a system could be designed to handle worst-case bursts, but this would be an overdesign. Bursts are by definition infrequent, so resources designed with bursts in mind would be unused most of the time.

2.2.2 Message Passing vs. Shared Memory

There are two fundamentally different approaches to providing inter-processor communication in a parallel environment. In message-passing, each processor is associated with a local memory which can be used by only that processor. The only way for processors to communicate is by sending messages back and forth, typically using calls such as `Send()` and `Receive()`. Under this paradigm, the programmer must code all communication explicitly, and is responsible for coordinating the senders and receivers. `Send()` and `Receive()` are usually implemented as blocking calls[3]. This means that `Send()` will not return until it can be guaranteed that the message has been consumed by the intended recipient calling `Receive()`, and vice versa. If the programmer makes a mistake, then the program will hang. (This makes programming trickier, but simplifies debugging.)

While message-passing is a technically clean solution to writing parallel programs, it has certain drawbacks. Firstly, experience has shown that message-passing programs are fairly difficult to write. The problem is that the whole notion of communicating processes is unfamiliar to a programmer used to writing sequential code. The most obvious sequential algorithm may have to be changed substantially before it can be implemented efficiently using message-passing. Secondly, since the basic communication mechanism is provided to the programmer by means of the operating system, there is a large overhead for sending a message. This means that sending small messages is very expensive, and the programmer must try to amortize the cost by coalescing as much data as possible into a single message. This approach is not natural to a programmer new to message-passing, and the learning-curve is steep.

Shared memory, in contrast, tries to make the programmer's view of the system look as much like a standard uniprocessor as possible. This paradigm provides the programmer with a single global memory space, which is shared and accessible by any processor. Communication in this case is implicit; in the simplest case one processor writes to a variable that is read by another processor, and the system is responsible for ensuring that modifications are propagated accordingly. In contrast to message-passing, synchronization in the shared memory model must be explicit. An extra complication under shared memory arises if processors are allowed to cache shared variables. In this case a cache coherence scheme (described in the next section) is necessary.

3. Non-blocking, or *asynchronous*, calls can also be used to increase concurrency, although they make the programmer's job more difficult since they require separate synchronization.

Another programming paradigm should be mentioned here, although it operates at a higher level than message-passing or shared memory, and can be implemented on top of either. In *data-parallel programming*, code is written as if it were sequential, with annotations added to indicate where data can be processed in parallel. The standard example here is High Performance Fortran, which looks much like regular Fortran but with extra compiler directives, embedded in comments, which control data partitioning. The goal is to have the compiler and run-time system handle the details of the actual data distribution. Such an environment can run on top of a system that supports either message-passing or shared memory, since the programmer is not supposed to be aware of the lower levels.

There is currently no consensus on which of the message-passing or shared-memory paradigms is better, although the focus in the research and commercial communities has shifted to shared memory. The feeling seems to be that the extra complexity of providing cache coherence is sufficiently offset by the reduced cost of application development (though there is really little hard evidence to back this up). That being said, there are successful commercial message-passing systems, such as IBM's SP2. Indeed, as we will see in section 2.5, there are new approaches to multiprocessor networking that support both schemes, allowing the system and programmer to independently choose whichever is more appropriate.

2.2.3 Cache Coherence

In a shared memory multiprocessor, it is possible for one or more processors to have local copies of a given cache line[4]. (Remember that in a message-passing machine, all memory is private, so this cannot happen.) When all accesses to the cache line are read-only, then there is no problem allowing all processors to use the line concurrently. A processor cannot just simply write to such a shared line if it needs to modify the line contents, however, since this would cause its local in-cache copy to be inconsistent with other copies scattered throughout the system. The purpose of a cache coherence scheme is to provide a mechanism to allow sharing of cache lines in an orderly manner. This is critical from the point of view of both programmers and compilers to insure the correctness of a program.

4. A cache line even in a uniprocessor cache is really a copy of some master block of data which resides at a fixed location in a memory module. Thus the novel feature here is that there can be many such copies.

There are numerous different approaches to providing cache coherence, with the most general classification being into hardware- and software-oriented schemes. The aim of hardware coherence is to push the responsibility for coherence down to the lowest architectural level, making it ‘invisible’ to all levels above. While this is convenient from the point of view of the program- and OS- writer, it greatly increases the complexity and cost of the hardware. Software schemes, on the other hand, leave responsibility for coherence in the programmer’s hands. This makes the hardware simpler, but adds complexity and overhead to the software. Thus, there are advantages and drawbacks to both methods, and it is not clear that any one particular scheme is superior in all cases. One of the advantages of a hardware coherent machine is that software coherence can always be implemented over top, allowing the use of the best scheme for a given situation, whereas if hardware support is not present from the outset, it cannot be designed in after the fact. Indeed, one of the goals of the NUMAchine project is to investigate trade-offs between hardware and software coherence.

The basic idea behind software coherence is that the programmer is responsible for managing writes to shared memory explicitly. This makes shareable memory qualitatively different from private memory and puts more of the burden of correct program execution on the programmer. In work such as Shared Regions [Sandhu 1993], the programmer associates an arbitrarily sized region of shared memory, which could have multiple writers, with a mutual exclusion lock. The programmer is responsible for making sure that any writable shared data is properly associated with a Shared Region, and if not the system makes no guarantees. This approach works well if the regions are large, because the overhead of providing the Shared Region is amortized over a large amount of data. This method can also benefit from block transfers, which can be more efficient than transferring a single cache line at a time.

Hardware coherence, while hiding the details of the protocol from the programmer, costs much more in terms of design time and hardware[5]. In addition, certain design choices such as the size of the basic unit of coherence must be fixed, because supporting flexibility in hardware is too costly. (It should be noted that with modern programmable logic, this could change.) On the other hand, the time overhead of providing coherence in hardware is greatly reduced compared to software. This is not only beneficial but necessary, since hardware coherence usually uses cache lines as the unit of coherence, which typically have sizes of 64

5. Synchronization such as locking must still be provided in a hardware cache coherent system. While concurrent writes to a single cache line are not possible, coherence does not enforce any ordering on writes to multiple cache lines.

or 128 bytes. This fine a granularity requires that the overhead be low for the scheme to remain efficient.

When a cache contains a line in the shared state and wants to modify it, there are two basic choices: update- or invalidation-based schemes. An attempt by a processor to write to a shared line in an invalidate protocol causes a request to be sent to the coherence engine to gain write ownership of the line. Such a request may or may not succeed depending on the cache line's current state. (Another processor may already have requested write access, for example.) When successful, the request causes *invalidations* to be broadcast to all processors sharing the copy. The invalidation command unconditionally kills a specific line if the tags indicate that it is present in the cache. Typically, the coherence engine must wait for acknowledgments (ACKs) to the invalidations to guarantee that all shared copies have been eliminated. At this point an exclusive ownership acknowledgment can be sent back to the original requester, which can proceed with the write. (In MIPS terminology [Heinrich 1994 and MIPS 1996], the original request for exclusive access is called an *upgrade*. They also use the invalidate to serve dual purpose as the upgrade acknowledgment.) Note that if a processor wishes to write to a cache line it does not have in its cache (i.e. the cache misses), then this line could still have shared copies elsewhere. In this case the processor sends a *read exclusive* request. The same invalidation process occurs, but instead of sending an ownership acknowledgment the coherence engine sends back a read exclusive response that includes data. A line that has been modified and written to is said to be in a *dirty* state, meaning that it contains a different value than main memory. If a dirty line must be ejected from the cache, the processor issues a *writeback* of the line to memory, and memory is then considered to be the line's owner. Another option is to have writes always propagate through to memory immediately, so memory is never out of date and is always the owner of data. This is called a *writethrough* scheme. While it simplifies the notion of ownership, it also generates unnecessary traffic for cache lines that are written multiple times by a single processor.

In an update protocol, the processor sends out a request to modify the given line, and includes the new data for the specific target word in the line. If this update request succeeds, the new data is broadcast and merged into all shared copies, and an ACK is sent to the requester, which can then proceed to change its copy. The state of the cache line is then called *dirty shared*, indicating that it is out-of-date with respect to memory, but that there may also be other copies in the system.

There are a number of trade-offs in choosing between an update and writeback-invalidate scheme. If a cache line is used in a producer-consumer fashion, where one processor modifies the line and many processors read it, then an update protocol performs better. An invalidate protocol causes the producer to invalidate the line, then forces all of the consumers to re-read the line on a subsequent load miss. If a line is shared only because many processors needed it in the past, but they will not need it in the future, then an update protocol will update lines which are not needed any more, causing a large amount of unnecessary traffic. In an invalidate protocol, after the invalidation phase, processors that have a real need for the line are forced to fetch it, which keeps the sharing list current. An invalidation protocol also performs better if there are a number of writes to a line before access by another processor, because a write to a dirty line causes no traffic (the processor already has exclusive write permission). On average, studies have shown that most traffic in a shared-memory multiprocessor is better suited to an invalidate protocol [Weber 1989, Culler 1999 and Srbljic 1997].

The most common writeback-invalidate protocol used in bus-based SMP systems is the MESI protocol (the name comes from the possible cache line states Modified/Exclusive/Shared/Invalid), also known as the Illinois protocol from its originators [Papamarcos 1984]. Each possible copy of a line in the processor caches and memory has one of the four states associated with it. The Shared state indicates that one or more caches and memory contain a copy of the line. The Invalid state is functionally equivalent to the cache line not being present in the cache. (An invalidation to a Shared state usually changes the state to Invalid but does not overwrite the cache tags; the Invalid state indicates that the line is not available for use by the processor even if the tag happens to match.) The Modified state is the same as the dirty state above. The protocol ensures that only a single processor cache can have a line in the Modified state, with other caches and memory guaranteed to be Invalid. The Exclusive state indicates that only one processor cache contains a copy, but that the line has not been modified and memory is still up-to-date. An ejected Exclusive line does not require a writeback, since it is not in the Modified state, but writes can proceed immediately since ownership has already been obtained. This state is useful for data that is used only by one processor, also known as *private data*.

A multiprocessor can be used to run multiple sequential programs simultaneously, which is referred to as *multiprogramming*. Here every program’s data is private. Without the Exclusive state such private data would generate a read then an invalidate on a write, but the invalidate in this case just represents wasted bandwidth if it is known *a priori* that no other

processor will be accessing the line.

One of the first hardware cache coherence protocols was called bus-snooping. Early multiprocessors usually consisted of a small number of processors (8 or less) connected by a bus. Busses were a natural choice as they were simple to design and in common usage. For multiprocessing, they also had the advantage of providing a natural *atomic broadcast* mechanism. The one-to-all broadcast is useful for sending updates on cache line information to all processors simultaneously, and being atomic makes it easier to verify the correctness of the protocol since numerous race conditions are avoided. A processor trying to read a line that is dirty in some other processor's cache broadcasts an *intervention* request to all other processors. The bus is held until a data response is provided to the intervention.

One of the problems with snoopy protocols is that the processor caches involved in the snoop also have to handle processor requests. This can cause the snoop to have a high latency while waiting for the cache to finish dealing with its processor, and since the bus is held for the duration of the snoop, this may create or increase bus contention problems[6]. One solution to avoid contention for the critical caching resource is to provide a duplicate set of tags dedicated solely for snooping. This is effective, but the performance must be traded off against the extra cost of the SRAMs needed to provided the second set of tags.

Busses are not a viable solution for connecting more than about 16 processors due to bus saturation. Increasing the bandwidth of the bus by making it wider and faster is possible, but only up to a point. Driving a wide bus (e.g. up to 512 bits) at high speeds with numerous loads requires special (i.e. expensive) drivers. The alternative to busses is to use some other network with better scaling properties. (These scaling properties will be described later in this chapter.) The basic snooping mechanism does not work in the absence of atomic broadcasts, so other schemes are necessary.

A standard approach is to use *directory-based* coherence protocols which keep the state and current location for each cache line in the system in some globally accessible table. The amount of information so stored and the handling of inexact information differentiates directory protocols. A simple directory could store just the fact that more than one processor has a

6. Most busses nowadays use a *split-transaction* protocol, where the bus is released between the request and response, allowing other transactions to use the bus. This helps to alleviate the problem, but makes protocol verification much more difficult because bus transactions are no longer atomic.

copy. A write by any processor would then require a broadcast of invalidation requests to all processors in the system. However, a broadcast invalidate costs not only network resources, but also time in each processor's cache to check if the line is present and kill it. A better scheme is to use some kind of list, specifying exactly which processors have a copy, called a *full directory*. But in this case, the cost of the directory does not scale well with system size. Typically, the total amount of memory in a system scales with the number of processors, N, because a fixed number of processors usually share a memory module. The number of cache lines is thus also $O(N)$. A full directory scheme requires for each cache line a bit-mask containing one bit for each processor in the system, for a total of $O(N^2)$ bits. Since the directory is usually implemented using SRAM, the cost of the system becomes unreasonable for large N.

Two approaches to improving directory scalability are *limited* and *sparse* directories. With limited directories, each cache line entry contains only enough storage for a fixed number of sharers. If this number overflows, then either a broadcast is used or previous sharers must be removed from the list. The efficiency of this approach depends on the amount of sharing being low in the common case, so the overflow handling occurs rarely. A sparse directory makes use of the fact that at any given point in time only some fraction of all the cache lines in the system will be in use, so there is no reason to allocate permanent directory entries for unused lines. The directory is thus created and managed dynamically on an as-needed basis. While this is the most efficient scheme in terms of storage, it also has the largest overhead for processing directory entries. (For a hybrid approach which uses the best features of both models, see the description of the LimitLESS directory in [Chaiken 1991].)

One final consideration for directory-based protocols is their physical distribution. The simplest model is to have the directories collocated with the home memory. For access to a remote memory, this means that the request may travel the entire span of the network before finding that a specific line is contained in some other node which is fairly close to the requester. A solution to this problem is to replicate directory information throughout the system in a bid to reduce the latency for accessing the coherence state. This replication increases the storage requirements for the directory and introduces a new problem of keeping the replicated directory entries coherent, but the limited and sparse directory techniques in the previous paragraph may be used, and the increase in coherence performance may outweigh the cost. Figure 2.2 shows a tree-based network, where the leaves contain the processing elements and memory, with each node above the leaves maintaining directory information for all of the cache lines below, both local and remote. If a cache line has already migrated to a local node,

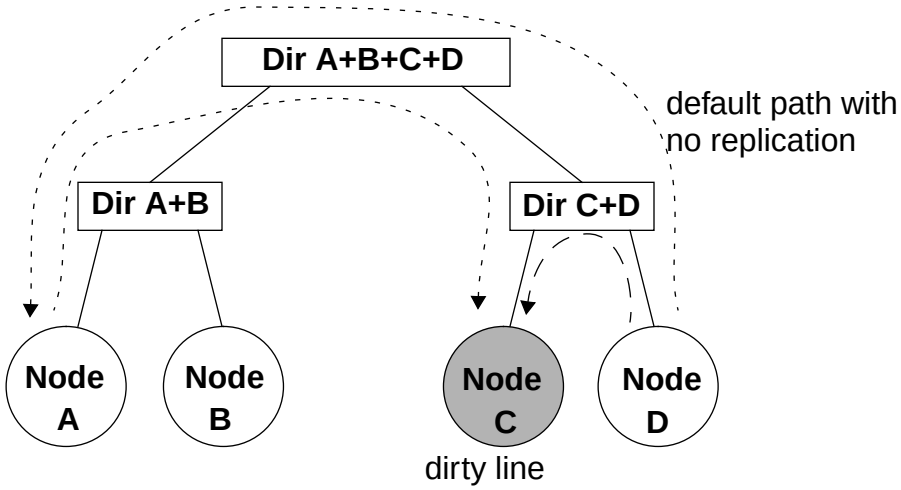

FIGURE 2.2: A hierarchical cache coherence directory. In a tree-based hierarchical network with processing nodes (containing processors and memory) at the leaves, coherence directory information at a given node of the tree is a superset of the information contained in all the node's children. A miss from Node D to a line whose home memory is Node A, but for which a dirty copy is contained in Node C can avoid traversing the entire tree.

then a request can be satisfied in the local directory without having to go the root of the tree (which contains the sum total of all directory information). To maintain the superset property, changes to cache states lower in the tree must be propagated upwards.

There is another approach to coherence called *page-based shared virtual memory* (SVM) [Li 1989]. which is software-based but makes some use of hardware in an attempt to reduce the programming complexity. (Unfortunately it also goes by the name virtual shared memory, or VSM.) SVM supports shared memory by making use of the virtual memory page-mapping mechanism available in all modern microprocessors. In a virtual memory system, a page table is used to map the processor's internal virtual addresses to real physical addresses that correspond to a location in some memory. SVM reduces the cost of hardware by implementing the coherence scheme in the page-fault handlers. The basic unit of coherence is a page, which is typically on the order of 4 KB in size. Because the coherence is done in software, the protocol can be much more complex with little extra cost. On the other hand, being in software means that the overhead for providing coherence is large. For SVM to work well, it must amortize

this cost by achieving a very high hit rate (i.e. a large amount of data re-use).

The first access to a shared page in SVM causes the handler to allocate a page in local memory. The handler must then send a query to the page's home location to ascertain whether the page is clean or dirty. The appropriate remote page is then copied into the local page, at which point all further accesses to this shared page will hit to the replica contained in the local memory. Coherence must still be maintained between the various copies of the page, which poses a problem because of the large block size. A writeback-invalidate scheme would cause the entire page to be invalidated and re-fetched on a write to **any** word on the page, which would generate a huge amount of traffic. If two writes by different processors occur to the same word on the page, then the word really is being shared and the associated coherence overhead is attributed to *true sharing*. On the other hand it is possible that the two processors are writing to entirely different memory ranges which happen to reside on the same page. The coherence overhead in this case is not really necessary, but is solely an artifact of the large coherence grain; such unnecessary overhead is referred to as *false sharing*. Great care must be taken in SVM by the programmer and compiler to lay out data to avoid false sharing. (True sharing is inherent in the algorithm, and is unavoidable.) However, heavy padding of data to adhere to page boundaries can cause memory usage to become very inefficient if the amount of padding is comparable to the amount of data. This will also have the effect of increasing the capacity miss rate in the caches. For these reasons vanilla SVM implementations achieve only mediocre performance.

One way of improving SVM performance would be to somehow allow writes to a page to avoid coherence traffic. One way of doing this is to associate shared data regions with locks (much like Shared Regions) and then observe that while a lock is held, writes to a shared page need not be made visible to other processors. Only when some other processor acquires the lock will it need the data. The idea of relaxing the memory model while making sharing more explicit to the programmer is called *lazy release consistency* [Cox 1992]. This is one instance of what are called relaxed memory consistency models, which is the topic of the next section.

2.2.4 Memory Consistency Models

Another issue that is of concern in the parallel computing domain is memory consistency[7]. As we have seen in the preceding section, cache coherence guarantees that if two or more processors write to a specific address, then some ordering is enforced such that all processors agree

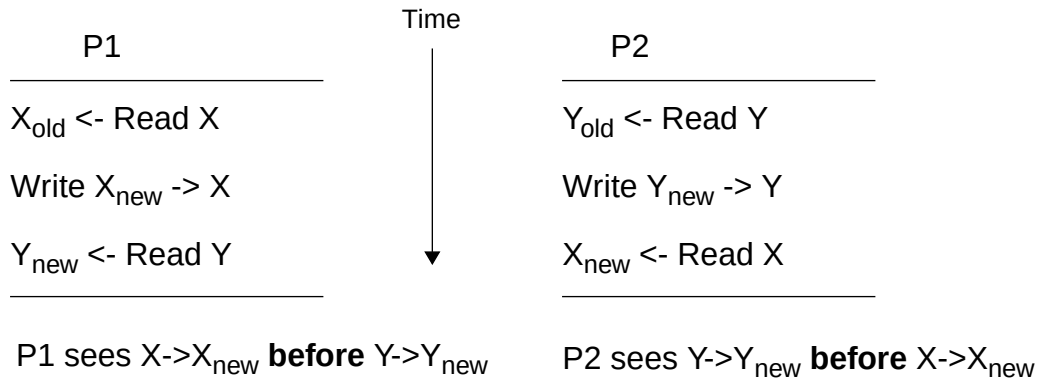

FIGURE 2.3: Memory consistency models. The observed interleaving of reads and writes by different processors defines a memory consistency model. In the figure two shared variables, X and Y, are read and written by different processors, P1 and P2 respectively. In the absence of any consistency, the two processors can disagree on the order in which the writes occurred.

on the current value. What coherence does *not* specify is the order in which reads and writes to different locations by different processors are observed. In Figure 2.3 the possibility of different observed write orderings is shown. If the variables *X* and *Y* happen to be in the same cache line, then coherence will guarantee that the processors see the same order. If P1 gains exclusive access to *X* before P2, then P1 will change the value and the line will be dirty. For P2 to gain either read or write access it must first fetch the line from P1, which means it will see the modification to *X* before it changes *Y,* which is the same order that P1 sees.

When the variables are in different cache lines, ordering constraints must be imposed outside of the coherence protocol. One of the most intuitive consistency models from the point of view of a programmer is *sequential consistency* [Lamport 1979] which is defined as follows:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

7. The nomenclature here is not well standardized, and the terms ‘coherence’ and ‘consistency’ are often used interchangeably. We take ‘coherence’ to mean a mechanism for presenting a coherent view of a *single* cache line to the system. We use ‘consistency’ to refer to the relationship *between different* cache lines.

In this model, for each processor we make an ordered list of all reads and writes in program order. We then form a global list by arbitrarily selecting an entry from each processor's list. If this global ordering is the same seen by all processors then we have sequential consistency. (The particular choice of ordering does not matter, only the fact that all processors see the same ordering.) This is convenient for the programmer because it is the same ordering that would be obtained if the parallel program were actually running as multiple threads on a uniprocessor. Note also that sequential consistency does not obviate the need for synchronization such as mutual exclusion and barriers.

While good for the programmer, sequential consistency imposes constraints on the hardware. A naive implementation would enforce a global order by means of a single sequencing point for all memory accesses, much like a ticket-system in a bakery. This sequencer represents a bottleneck, and would destroy much of the performance advantage of using a parallel system. A more sophisticated implementation can reduce the impact of sequential consistency, although never to zero. (In section 3.1.7 we describe NUMAchine's implementation of sequential consistency, and show that the added overhead is minimal.)

To improve performance, various proposals have been made for weaker consistency models. In many cases sequential consistency is an overly strict regime to impose on the hardware, and is not always necessary from the programmer's point of view. Figure 2.4 shows how certain orderings may not be critical for a program to be semantically correct. The writes to *A* and *B* by P1, for example, can be re-ordered by the processor or the network, as long as they are both visible before the assignment to `flag`.

The weakest possible ordering is no ordering at all, except for the write ordering provided by the coherence mechanism; reads and writes even from the same processor can have their orders swapped. As described in [Goodman 1991], such a system is unusable without an operation such as a *fence*, which guarantees that all accesses before the fence are complete (visible) before any access after the fence[8]. Some other models that lie between the two extremes include *processor consistency* (PC) (also in [Goodman 1991]) which allows reads to bypass writes, *partial-store ordering* (PSO) [SUN 1997a] which allows writes to bypass each other, and *relaxed memory ordering* (RMO) [SUN 1997a] which allows reads and writes to bypass previous reads. Note that when we talk of bypassing, we really mean completion of the access

8. Note that a fence is a more primitive construct than a barrier. Fences enforce ordering on reads and writes from a single processor, whereas barriers provide synchronization between processors.

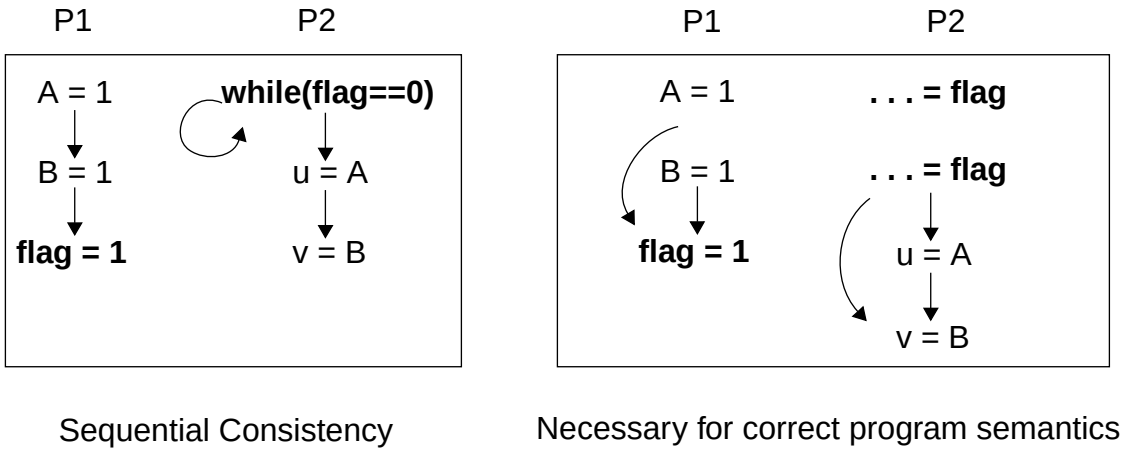

FIGURE 2.4: Relaxed consistency. Arrows indicate the relation 'must occur before'. Sequential consistency maintains the order of every access, but only certain orderings are crucial to the programmer. Accesses to the `flag` variable are the only ones that need to be ordered for synchronization correctness. (Figure taken from [Culler 1999]).

(i.e. the cache is updated and the processor can make the results visible in the register file). Modern processors use techniques such as nonbinding prefetch and speculative execution to increase performance. The memory consistency model has implications for the ability of the processor to retire loads and stores in these instances.

There are numerous other issues pertaining to memory consistency, but they are not pertinent to the work that follows. It turns out, that for certain architectural reasons discussed in Chapter 3, NUMAchine provides fairly natural support for the sequential consistency model. In addition, recent results have indicated that modern processors can make sequential consistency perform almost as well as the relaxed consistency models [Hill 1998 and Gniady 1999]. A brief look at relaxing the memory model will be given in Chapter 4. A very thorough treatment of consistency issues can be found in [Adve 1996].

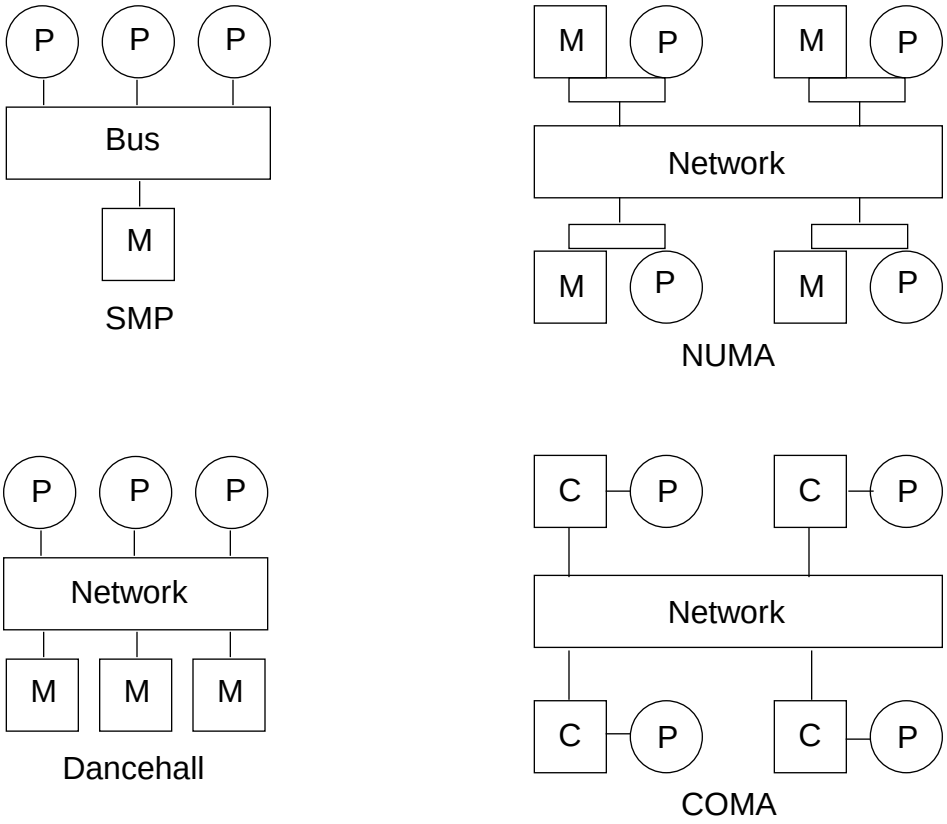

FIGURE 2.5: Various classes of memory subsystems. The three broad categories are UMA, NUMA and COMA**.** In uniform memory access (UMA) architectures such as the dancehall and SMP, all accesses have the same latency. Non-uniform memory access (NUMA) machines allow for both local and remote memories. In a cache-only memory architecture (COMA), the memories are replaced with large caches called attraction memories. (The processors can contain caches in any of these categories.)

2.2.5 Memory Subsystems

There are numerous different options for the layout of memory in a shared-memory multiprocessor. One of the most popular configurations among today's commercial machines is called the *symmetric multiprocessor* (SMP) (see Figure 2.5). This consists of some relatively small number of processors (e.g. 8 or less) along with a memory module sharing a bus. Such an architecture is classified as having *uniform memory access* (UMA), because each processor sees the same latency to memory. With a more general network, there is the notion of accesses

being remote or local, depending on whether they must traverse the network to be serviced or not, respectively. This leads to *non-uniform access* times (NUMA). It is possible to make all memories remote, obviating the need for a processor-memory connection. This is called a *dancehall* architecture [Culler 1999], and is typically also UMA since all memory accesses incur the same remote access penalty.

Any of these systems can support message-passing, software or hardware cache coherence. If a NUMA system uses the shared-memory paradigm, it is called *distributed shared memory* (DSM), and if cache coherence is provided in hardware, then the term CC-NUMA is used.

An entirely different type of memory system is the *cache-only memory architecture* (COMA). To understand the rationale behind this architecture, consider a CC-NUMA system where each processor contains a large dedicated cache to enhance locality. These caches use copies of data from main memory; a cache line not contained in any cache has either never been used, or was used but had all copies ejected. In the best case, all data needed by the processors should be stored in the caches. However, real caches suffer misses. In a cache-coherent multiprocessor, caches can miss for one of four reasons:

- Cold or Compulsory Miss - The very first access to a given line will miss. This is clearly unavoidable, but can be alleviated by prefetching. Since large line sizes tend to have a prefetching effect, they can help to reduce this miss rate.

- Capacity Miss - This is caused by the cache being too small to contain all of the data that is needed. This can only be fixed by increasing the cache size. A cache of infinite size would show no capacity misses. A very small cache with a large line size could increase the capacity miss rate, because large lines have the potential to increase the pollution of the cache with unnecessary data.

- Conflict Miss - This occurs when two different cache lines map to the same entry in the cache, forcing the current occupant to be ejected. This effect can be reduced by increasing the set-associativity or the size of the cache. A fully associative cache would have no conflict misses. A very large line size can also exacerbate the conflict miss rate.

- Coherence Miss - In a cache-coherent machine, lines may need to be invalidated to maintain coherence. This has nothing to do with the cache organization. These misses could possibly be reduced by changing the coherence scheme or reducing the line size

so less data is thrown out for each invalidation, and also by reducing the amount of false sharing.

COMA architectures aim at reducing capacity misses by greatly increasing the size of the caches (to the same size as main memory), and eliminating main memory entirely, since it acts only as secondary storage and a place for coherence to take place. These large memories, used as caches, are called *attraction memories*. They take over the central role in the coherence protocol, and are backed directly by paging to the disk. While COMA architectures do reduce the capacity miss rate, the absence of main memory means that address mapping is not fixed. Given an address that misses in the attraction memory, there is no way of figuring out just from the address itself where to go to fetch the line. (In a CC-NUMA machine the upper address bits typically select a particular memory module in the system.) Thus, the original COMA design requires a global search of the attraction memories' tags until a line is either found, or if not found then paged in. For this reason, COMA machines often use a hierarchical directory scheme to reduce the coherence latency. Also, because the attraction memories are caches, there is a problem when only one attraction memory in the system contains a cache line: if ejected, this line must be saved to disk or moved to another attraction memory, because it is the only existing copy. These are the main reasons why COMA has proven to be extremely expensive both in terms of latency and logic complexity. The trade-off between decreasing capacity misses and increasing coherence miss overhead favours COMA architectures only for those applications that work on very large data sets, and have correspondingly high capacity misses. For an analysis of the relative performance of CC-NUMA and COMA systems see [Stenstrom 1992]**.**

In a *flat COMA* scheme (also described in [Stenstrom 1992]), a dedicated home location is provided for addresses. The home stores only coherence directory information, with data being maintained by the attraction memories as before. While it simplifies directory lookup, flat COMA has difficulties when dealing with the last copy replacement problem. In a normal COMA system with a hierarchical directory, an ejected last copy can move up the tree until it finds a directory which contains a line which can be ejected—either because the line is invalid or because it is shared and there is at least one other copy elsewhere in the system—and can take that line's place. Flat COMA, in contrast, must keep extra information as to which lines are available for ejection, and if no local space is available must start searching through other attraction memories until a suitable destination is found.

While COMA and CC-NUMA schemes both have their pluses and minuses, CC-NUMA seems to be the architecture of choice for large-scale modern commercial multiprocessors. Not only does CC-NUMA's relative simplicity lead to lower overall system costs, but COMA only outperforms CC-NUMA on a small group of applications, and the performance improvement is not enormous. For those cases where COMA's data replication and migration are advantageous, it is possible to achieve a similar effect by performing the same operations at the page level by means of the operating system. (See [Laudon 1997] for a description of the CC-NUMA SGI Origin2000, which uses this approach.)

2.3 Multiprocessor Networks

The most critical design choice in a multiprocessor is the network. It affects performance through its latency, bandwidth and contention-handling characteristics. It also has a major impact on the design complexity of the cache coherence protocol and memory consistency model since, as we have seen in preceding sections, both of these depend heavily on the ordering of transactions, which in turn is impacted by network topology.

One of the most fundamental requirements of a multiprocessor network is *scalability.* This means that certain critical network parameters should increase no worse than linearly as the system size (usually measured by the number of processors, N) is increased. Ideally, we would like a network to be scalable in the following three areas:

- Cost - This should scale linearly with N, so that the marginal cost of adding another processor is constant. It is also desirable for the initial cost (for just a few processors) to be low, so small configurations are feasible;
- Latency - This should be constant, but since logic fan-in cannot be infinite the best that can be expected is $O(\log N)$;
- Bandwidth - This should scale with N.

For bandwidth, it is really the *bisection bandwidth* that should scale linearly. The bisection bandwidth is defined as follows:

Consider an equipartition of a system of communicating nodes, and calculate the aggregate bandwidth between the two partitions. The bisection bandwidth is the lower bound of this aggregate bandwidth over all possible equipartitions.

The linear scaling requirement is necessary if we assume that on average all processors communicate equally with all other processors. (Note that this is not the worst case scenario, which would have *N*-1 processors all accessing the last processor.)

The scalability of a network is determined by its topology. The overall performance of a network is affected by other characteristics such as:

- Direct or Indirect. Direct networks put a switch at each node, meaning that some nodes are closer (in a latency sense) than others. This allows for efficient nearest-neighbour communication. Indirect networks put all nodes on the periphery of the network, making access times uniform between nodes.

- Packet- or circuit-switched. Circuit-switched networks set up a fixed dedicated connection between two nodes. This connection is fast because it is not shared, but there is an overhead cost for the setup and breakdown of the circuit. Once the circuit is set up the routing is fixed, so message routing overhead is low. Packet-switched networks break the communication stream into packets, each of which is routed independently. There is no setup overhead, but routing overhead is increased, because typically more than one routing decision must be made. Packet-switching allows the network resources to be used in parallel for many different streams, which makes it the choice for multiprocessors where many nodes communicate simultaneously.

- Store-and-forward or cut-through routing. Store-and-forward waits for an entire packet to be received at a switching element before routing it to the next. This works well if all packets are of fixed size, but requires buffers large enough to contain some maximal number of full packets. Cut-through (also called wormhole) routing determines the next switch based on the packet header, then passes the rest of the packet through as it comes in. This adds complexity to the flow-control and congestion-handling logic, but uses less buffer space and reduces the latency. Virtual cut-through is a hybrid of these two: it works like wormhole routing, but stores the entire packet if the outgoing channel is blocked.

- Static or dynamic routing. Static routing uses fixed routing tables that do not change in time. Dynamic (or adaptive) routing can change routes to avoid congested areas. It also provides fault-tolerance because a broken channel can be bypassed. Dynamic routing is more complex, and allows for the possibility of different paths between two fixed nodes.

- Error checking, higher level flow-control protocols, and other fault-tolerant features.

In the following sections we will look at some generic multiprocessor interconnects.

2.3.1 Full Crossbars

In a *P*x*Q* crossbar, *P* input ports are fully connected to *Q* output ports through a single switching layer (Figure 2.6 (a)). In most cases, *P* and *Q* are the same. Normally, we consider each input to be paired with an output, providing fully bidirectional links. (A unidirectional crossbar is usually a part of some larger network.) Crossbars provide linear bisection bandwidth scaling, and a constant latency since there is only a single switch. The cost, however, scales as N^2, so the number of ports is typically some small number, say less than 10. Note, however, that crossbars can be hooked together to form other network topologies.

2.3.2 Multistage Interconnection Networks

Multistage Interconnection Networks (MINs) represent a class of networks, some examples of which are the Omega (Figure 2.6 (b)), Banyan and Butterfly networks. Each switch within the MIN is a *p*x*p* crossbar, so the number of switch levels (and hence the latency) scale as $\log_p N$. The cost of the network scales as ($N \log_p N$). All nodes are equally ‘remote’ and suffer maximal latency, which does not allow for local-communication optimizations. The size of a MIN can be grown by adding more layers of switches.

2.3.3 Hypercubes

An *n*-dimensional binary hypercube connects 2^n nodes. A 4-D example is shown in Figure 2.6 (c). If we consider the nodes to reside at the corner of a unit cube in *n*-dimensional space, then each node connects to its *n* nearest neighbours along each dimension. If we label the node position by its *n*-space coordinates (e.g. 0010, 1110), then connections exist between any nodes that differ by one bit. As with MINs, hypercubes offer linear bisection bandwidth growth, and latency that grows as $\log_2 N$. The cost similarly grows as ($N \log_2 N$). One problem with hypercubes is that the degree (number of connections) at each node increases as the system size grows. In practice, this means that the maximum degree of a node (and hence the

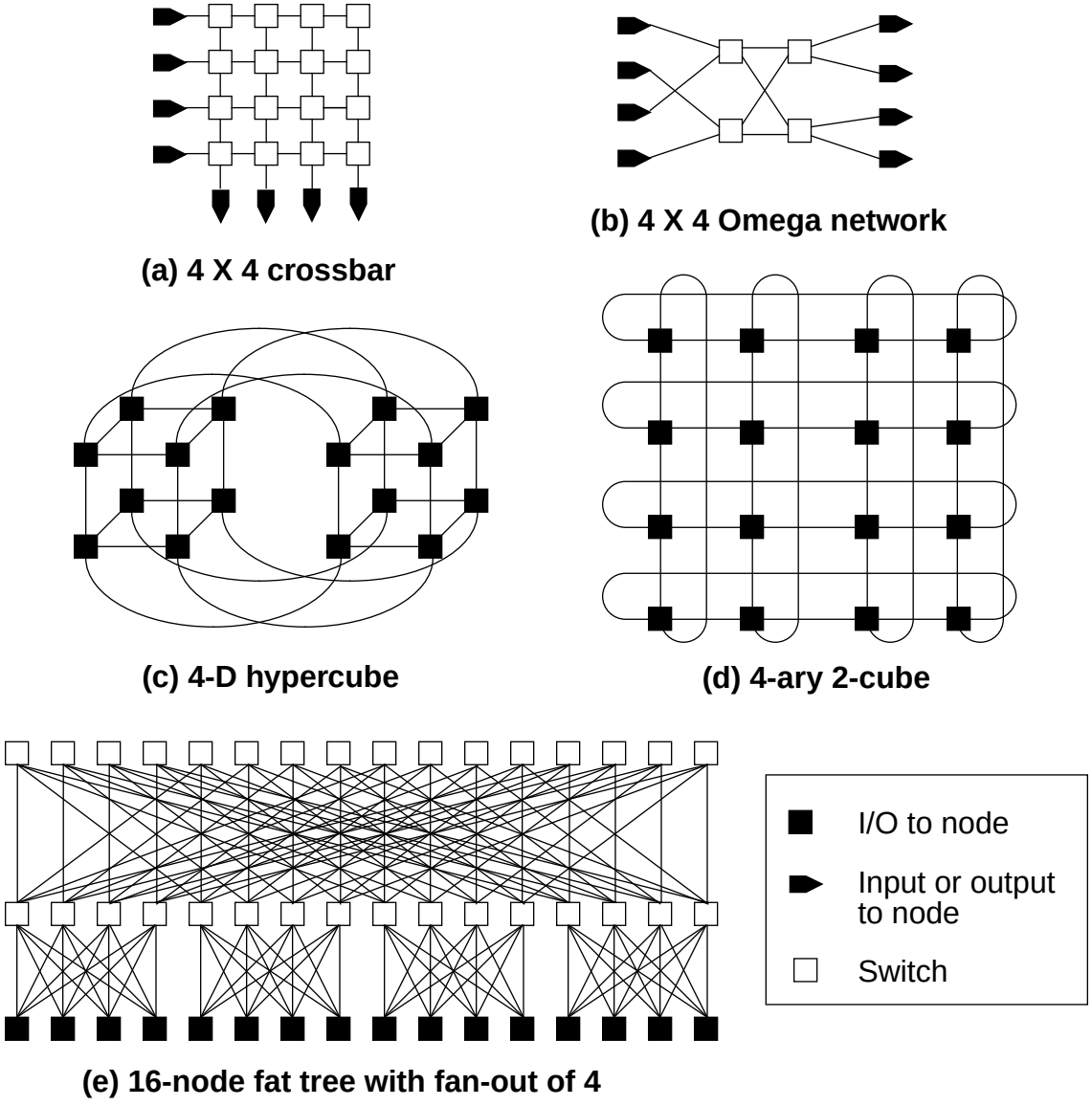

FIGURE 2.6: Scalable interconnection networks. (From [Lenoski 1992])

maximum system size) must be decided upon before implementation. Hypercubes allow for local communication.

2.3.4 *k*-ary *n*-cubes

The *k*-ary *n*-cube is a more general form of the hypercube, with the binary base 2 replaced by an arbitrary base *k*. A 4-ary 2-cube is shown in Figure 2.6 (d). Most systems use a dimensionality of 2 or 3, and grow the system size by increasing *k*. The 2-D and 3-D incarnations are also called *meshes*. Having $k > 2$ means that node symmetry is lost for a mesh; nodes on the boundary of the mesh have lower degrees than nodes in the interior. This is a problem from the hardware standpoint, since either switches with varying port numbers must be produced, or some of the ports must go unused. The symmetry can be regained by adding links between the peripheral nodes along each dimension. (In topological terms this is equivalent to wrapping the edges around and connecting them.) In this case, the network is referred to as a *torus*, and the average latency between nodes is reduced by a factor of two. Routing in a torus is more difficult, as is wiring. For a given dimension, the cost grows linearly. The drawback is that bisection bandwidth only grows as $N^{(n-1)/n}$, and latency increases as $N^{1/n}$.

2.3.5 Fat trees

An *N*-node fat tree with fanout *f* is constructed by superposing *N* individual fanout-*f* trees in such a way that each level of the fat tree has a constant number of links between switches. This provides for linear scaling of bisection bandwidth and log *N* latency increase, while providing multiple root nodes to ensure the root is not a bottleneck. Fat trees are like MINs in that their cost scales as (N log N) and they can be grown by adding layers of switches. They are different from MINs in their ability to provide low-latency shortcut paths by routing only up to the lowest level in the tree necessary to reach the destination.

2.3.6 Busses and Rings

As mentioned before, the most common interconnect topology is the bus due to its simplicity. While the latency and cost are constant, the bandwidth is also constant, which means that a given bus is not scalable beyond a certain point. This saturation point can be moved up by increasing the width and speed of the bus. The drawback is that the cost and complexity increase, because more exotic signalling technologies are required.

Rings also have a fixed bandwidth. However, the latency increases linearly as nodes are

added, as does the cost. One advantage to rings is that they use point-to-point connections, as opposed to busses which have multiple loads on a single wire. Electrically this makes the ring's signalling environment much cleaner, and allows rings to be run at much higher speeds than busses, which can be used to trade off against the latency and bandwidth restrictions.

Busses and rings can both be organized hierarchically to increase the upper limit on their scalability. The higher levels of the hierarchy can be made wider or faster to increase the bisection bandwidth. The hierarchical organization maintains the ordering and broadcast capabilities of busses and rings, which can be used to advantage for cache coherence protocols, as will be seen in Chapter 3.

2.3.7 Network Summary

Table 2.1 summarizes the important network characteristics. The latency is determined on an uncontended network with uniform loads. Switch and wire costs indicate the number of switching elements and interconnections needed, respectively. In a hierarchy, the number of switches or bus/ring interfaces is the same as the number of internal nodes in an *N*-leaf tree with fanout *f*. With $n = \lceil \log_f N \rceil$, we can approximate the sum as

$$Switch(N) \approx N\left(\sum_{i=0}^{n} f^{-i}\right) - N \approx \frac{f(N-1)}{f-1} \qquad \text{(EQ 2.3)}$$

None of the networks is ideal in all respects. Performance has generally been considered more important than cost, which has favoured low-latency networks with good bisection bandwidth, such as tori and meshes. (See, for example, the SGI/Cray T3E and Hewlett-Packard V-Class machines.) However, for small- to medium-sized networks, say less than a few hundred nodes, performance can be increased by the use of custom-designed routers, and scalability is arguably not as critical an issue as cost. This has affected the approach to multiprocessor interconnects, as will be seen in the next section.

2.4 System Area Networks

Up until a few years ago, multiprocessor networks used either LAN-class technology or completely custom-designed solutions. Research in recent years has indicated that multiprocessor

TABLE 2.1: Summary of network characteristics. A system is assumed to contain N nodes**.** For the MIN and fat tree each switch is an *fxf* crossbar. For the hierarchical bus and ring there are f nodes at each lowest level bus/ring. Indirect networks do not allow fast local communication, direct do, except for a bidirectional ring, which does. (Table from [Lenoski 1992].)

Topology	Type	Switch Cost	Wire Cost	Average Latency	Bisec. Band.
Crossbar	*Indirect*	N^2	N	*const.*	N
MIN	*Indirect*	$(N/f)\log_f N$	$N\log_f N$	$\log_f N$	N
Hypercube	*Direct*	$N\log_2 N$	$(N/2)\log_2 N$	$(1/2)\log_2(N/2)$	$N/2$
2-D Torus	*Direct*	N	$2N$	$N^{1/2}/2$	$2N^{1/2}$
3-D Torus	*Direct*	N	$3N$	$3N^{1/3}/4$	$2N^{2/3}$
Fat tree	*Indirect*	$N\log_f N$	$fN\log_f N$	$2(\log_f N\text{-}1) < L < 2\log_f N$	fN
Bus	*Direct*	N	*const.*	*const.*	const.
Ring	*Direct*	N	N	$N/2$	*2 const.*
Hierarchical Bus	*Indirect*	Switch(N)	Switch(N)/N	$2(\log_f N\text{-}1) < L < 2\log_f N$	*const.*
Hierarchical Ring	*Indirect*	Switch(N)	Switch(N)	$2N(\log_f N\text{-}1) < L < 2N\log_f N$	*2 const.*

networks have very specific requirements which differ from those in other networking environments. Latency and bandwidth are particularly critical in the tightly-coupled multiprocessor environment. However just as important is low error rate. In an unreliable network such as a LAN, higher-level protocols (e.g. TCP/IP) provide reliable communications at the cost of much higher protocol overheads. In a latency-sensitive domain, such as multiprocessors, such overheads are unacceptable. System Area Networks (SANs) usually contain fast low-level error-detecting and correcting protocols directly in hardware. In order to avoid contention, they also contain quick flow-control mechanisms. While all of these features would be benefi-

cial in a LAN domain, the added cost is not justifiable, since software protocols provide adequate performance.

The driving force behind SANs was message-passing. Sending a message using standard network interfaces typically involves going through the following steps:

1. Put the message into a buffer in local program memory.
2. Trap into the operating system, which then copies the entire message into a buffer in kernel memory.
3. Send this buffer to the network interface, which normally involves copying it into a buffer local to the interface.
4. Transfer the data across the network, then go through all of these steps in reverse order at the other end.

Clearly the copying done in steps two and three is not absolutely necessary. The goal of SANs is to provide applications with more direct access to the network interface. In fact, it is even possible to do away with the copying of the local program buffer into the network interface by mapping the region of local memory as uncached and making the physical address select the network interface directly. By also providing a direct virtual memory map for the control registers in the network interface, it is possible to have the program initiate the message sending without any operating system involvement whatsoever. Such *zero-copy* schemes have allowed message-passing overheads to come down from tens of microseconds to less than one microsecond, which is particularly beneficial for supporting the small messages that are often necessary in a multiprocessing environment.

The following sections will give brief descriptions of some commercial SANs.

2.4.1 SCI (Scalable Coherent Interface)

SCI [Scott 1992] is an attempt to combine standards for both a physical networking layer and a cache coherence scheme to provide an ‘off-the-shelf’ solution for implementers of CC-NUMA machines. The physical layer specification aims at high-speed bus-like performance, but does not enforce any particular topology. In small configurations, SCI typically uses single or dual rings, for which commercial chipsets are available

The cache coherence protocol is invalidation-based, and relies on a distributed directory using linked lists. Each network interface stores its locally active sharing list entries, which distributes the directory storage across the machine, and also keeps it near the network to

allow quick access. One of the drawbacks to the linked list structure is that invalidations to highly shared blocks must traverse the entire list, which may be scattered across the machine, potentially increasing coherence overhead substantially. For applications where the degree of sharing is low, SCI performs fairly well.

SCI is used in Sequent's NUMA-Q, DataGeneral's AViiON and HP's V-class servers. Unfortunately SCI suffers from early- and over-standardization. Both the physical layer and the linked-list coherence are fairly out-of-date already. SCI is a good example of trying to standardize too much at too low a level while technology is still rapidly changing.

2.4.2 Myrinet

In contrast to SCI, which was targeted specifically at parallel processing networks, Myrinet [Boden 1995] was born from an attempt to increase LAN performance. One of its goals was to allow clusters of workstations to be connected to form a virtual multiprocessor. This attempt to leverage the preponderance of relatively cheap workstations was made popular by Berkeley's NOW (Network of Workstations) project [Anderson 1995].

Myrinet puts an embedded protocol processor, a large amount of SRAM and DMA engines onto its network interface cards to allow most of the network and protocol processing to take place on-board. The Myrinet network consists of an 8-port wormhole-routed full crossbar. While the performance is fairly good, it is not clear whether making a faster LAN interconnect is more fruitful than trying to approach the problem from the other direction and take a custom-designed dedicated multiprocessor network architecture and commoditize it.

2.4.3 Memory Channel II

The Memory Channel II architecture [Fillo 1997] was developed by Digital Equipment Corp. (now owned by Compaq) for use in its Alphaserver product line. (The original Memory Channel had the same basic architecture, but with lower performance.) It uses virtual memory-mapped pages to directly access the network interface, and provides a form of *reflective memory.* Writes to a page are made visible on all other readable pages in the system shared by other nodes. Only one node can map a page for writing, so two-way communication is achieved by pairs of pages. In order to ease integration into SMP nodes, the network interface cards in Memory Channel are implemented on PCI cards, which can simply be plugged into

the ubiquitous PCI I/O bus. Although this increases their latency (because they have to cross the memory-to-PCI bridge) it lowers the cost and implementation complexity considerably. The network is based on an 8x8 full crossbar.

The basic communication mechanism provided is message-passing. Reads to a page only show data that has been written; updates to a page cannot be pulled across, only pushed by the originator. This one-way communication has associated cost penalties for inherent two-way communication patterns such as synchronization. This renders Memory Channel mediocre at best for very fine-grain communication patterns.

2.4.4 Synfinity

The Synfinity interconnect [Weber 1997][9] is specifically designed to support a tightly-coupled multiprocessor. Synfinity utilizes a basic crossbar switching element (in this case 6x6), many of which can be connected in any desired topology. The design aims to provide very high performance, but also very high reliability which is critical for systems in the commercial world.

To attain these goals they split the functionality into three layers. The lowest level is the *fast frame mover* (FFM), which is responsible for the basic data transport. This layer uses source-routed cut-through to provide very low latency (on the order of 40 ns for a single level of switching). The FFM focuses solely on speed, making the logic as simple and fast as possible. The next level up is the *reliable packet mover* (RPM) which uses error-checking and -correcting codes to provide reliable communication. If corrupted packets can not be fixed, then the RPM uses the FFM to re-request the packet. The *interconnect services manager* (ISM) sits at the top of the chain. It is the only part of Synfinity that is dependent on the actual details of the node to which it connects. The two basic services provided by the ISM are a directory-based cache coherence protocol, and a message-passing protocol. Any different or extra functionality only requires a redesign of the ISM. Fujitsu System Technologies has versions of the card for connection either to a PCI bus or directly into the backplane of an Intel SMP. While this is the highest-performing and most integrated SAN of the four presented here, it is not yet shipping in any commercial systems, so it is difficult to judge its overall impact.

9. Note that the paper refers to the Mercury Interconnect. The rights to the technology were bought by Fujitsu, and the name changed to Synfinity for trademark reasons. A modified version of the paper can be obtained from the Fujitsu System Technologies website, http://www.fjst.com.

2.5 Sample Multiprocessors

This section looks at some existing multiprocessors, both commercial and experimental. As evidenced by the varied architectures, there is no one best approach to building these machines. Clustered systems are a common approach, partly because they leverage commodity SMP nodes, and partly for their RAS characteristics. RAS stands for Reliability/Availability/Serviceability, and is turning out to be one of the more important features in commercial acceptance of multiprocessor systems. (Note that in the commercial world, the marketing term for this class of machines is ‘enterprise server’.) A system is reliable if the user can count on jobs being completed in a deterministic and timely fashion, even in the face of heavy loads or system malfunctions. This includes the areas of fault-tolerance and efficient load management. Availability means avoiding downtime due to crashes or maintenance. The standard here is set by mainframes, which can achieve five-9 (99.999%) uptime, or less than 5 minutes of downtime per year. And finally, serviceability indicates a system’s ability to gracefully handle the inevitable failures of processors, memory, disks and networking components. Since multiprocessors, by their nature, have a large number of such components, statistically speaking they will suffer high failure rates. Features for serviceability include redundant power supplies and hot-swappability.

2.5.1 Stanford DASH and FLASH

The DASH [Lenoski 1992a & 1992b] and follow-on FLASH [Kuskin 1994] projects at Stanford are both directory-based CC-NUMA machines using 2-D mesh networks. As shown in Figure 2.7, DASH uses an SMP node consisting of four MIPS R3000 processors sharing a bus with memory and a network controller, which also contains the coherence directory maintenance hardware. The network controller also contains a *remote access cache* (RAC) to reduce the latency of remote accesses for cache lines that are subsequently fetched by another processor in the node, or re-fetched by a processor which has ejected the line. This RAC can also be viewed as a small attraction memory for remote addresses, which offers some of the migration and replication benefits of COMA.

The FLASH design does away with the bus-based SMP and the RAC. Their conclusions from DASH were that the localization of remote references afforded by the SMP/RAC combi-

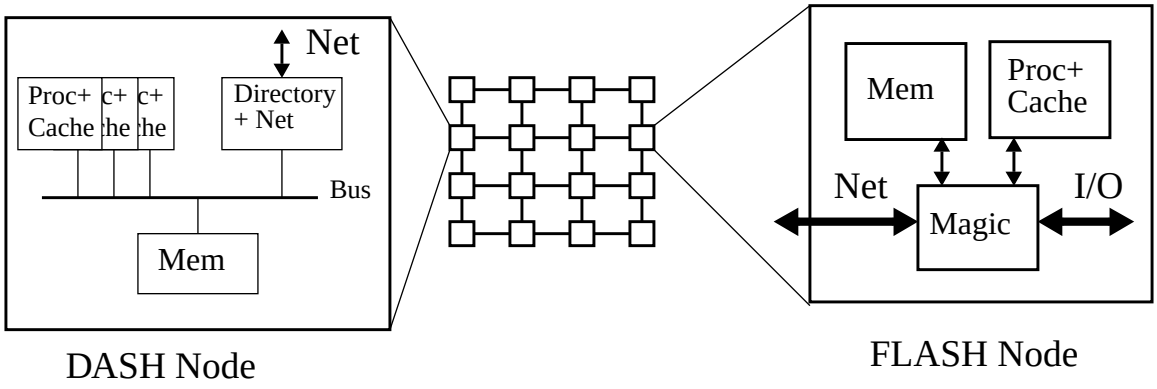

FIGURE 2.7: DASH and FLASH architectures.

nation were not very effective and cost too much in terms of logic. The MAGIC chip performs the functions of a network and memory manager, as well as implementing a microcoded coherence protocol processing engine which enables fast and flexible shared-memory and message-passing. Some of the same benefits of the RAC are obtained by implementing page-based migration and replication policies in software.

2.5.2 Illinois I-ACOMA

Illinois's I-ACOMA [Torellas 1996] uses both a flat COMA coherence protocol and a technique called *simultaneous multithreading* (SMT) [Eggers 1997]. The goal of the COMA research is to investigate techniques for reducing COMA's coherence overhead. An example is the *invalidation cache*, which keeps track of recently invalidated lines (that would miss in the attraction memory) and forwards them directly to the appropriate remote directory. The SMT research is independent and complementary to the COMA work. Realizing that there are limitations to the amount of instruction-level parallelism (ILP) available for superscalar processors, the SMT approach seeks to make use of multiple threads as well as ILP inside a single chip. The compiler is responsible for scheduling multiple threads, each of which feeds a separate superscalar engine inside the processor. The idea is to apply VLIW techniques at the level of threads rather than instructions. Internally the processor can hide latency by choosing other threads to execute when some thread is blocked due to a high-latency access. Since the threads

can be associated with different programs, this approach also supports multiprogrammed workloads. Hardware does not yet exist for the I-ACOMA.

2.5.3 Teracomputer

The Teracomputer [Bokhari 1998] represents a radically different approach to multiprocessing. The memory system uses a basic dancehall architecture, as shown in Figure 2.8. However, instead of trying to reduce or avoid latency, the Tera approach is to look for other work to keep the processor busy while waiting for a high-latency operation to complete. Tera uses a custom-designed heavily multithreaded processor with a very low context-switch overhead to achieve this goal[10]. The current implementation of the Tera processor can support up to 128 instruction *streams*, each of which behaves as a virtual processor with its own registers and context. There are no caches in the Tera architecture, and all memory is equidistant from all processors. On every clock cycle the processor switches unconditionally to the next stream in a 21-deep stream pipeline. Thus each stream can utilize at most 1/21 of a processor's cycles. Since even with optimizations for speed Tera's memory takes about 50 cycles to return a 64-bit word, at least 50 streams must be active in the processor at any one time for it to be fully utilized. While certain applications do display this amount of concurrency, Tera has not presented convincing evidence that their approach is more generally applicable.

2.5.4 SUN E10000

Originally codenamed Starfire, this is SUN's biggest machine in the enterprise server market. This 64-processors SMP is unique in that it uses a globally snooped interconnect in a UMA configuration. As shown in Figure 2.9, nodes consist of up to four UltraSPARC processors, each with memory. A processor cannot access its 'local' memory directly, though. All accesses must go through one of the four globally shared and snooped address busses. (The low 2 bits of the addresses select which address bus is used. The address busses are replicated only for performance reasons.) Data responses use a separate 16x16 crossbar. SUN's approach of throwing hardware at an old design is expensive, and ultimately unscalable. By sticking to

10. This is similar to the SMT concept from the previous section, however the Tera multithreading concept predates SMT by a number of years. The SMT idea really descends from the Tera work.

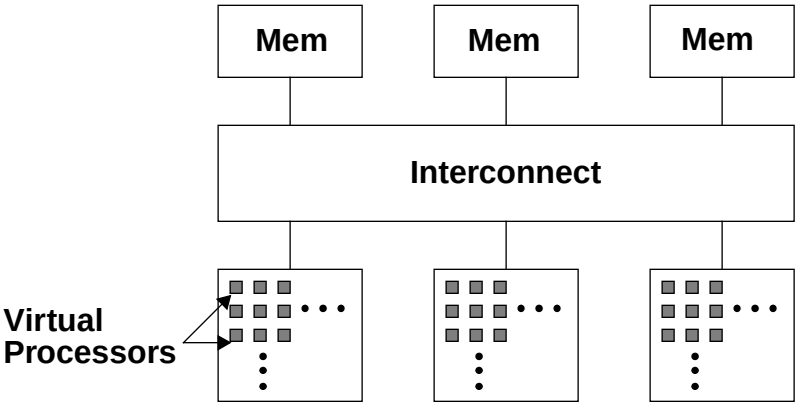

FIGURE 2.8: Teracomputer architecture.

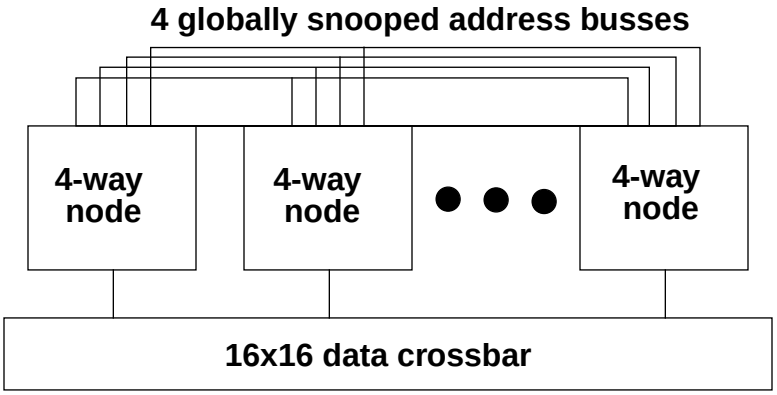

FIGURE 2.9: SUN E10000 architecture. Each node contains 4 UltraSPARC processors and memory. All accesses, local or remote, go across one of the global address busses. Data is transferred separately on the data crossbar.

incremental design improvements, they have managed to keep their costs low enough to compete with other vendors, and the E10000 is still one of the highest performance machines on the market.

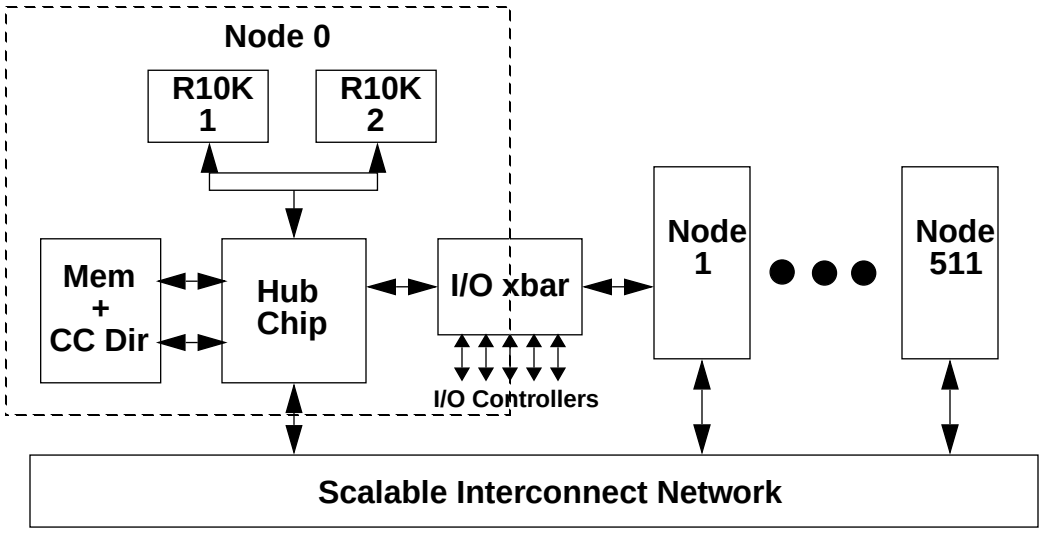

FIGURE 2.10: Layout of the SGI Origin 2000. (Figure from [Laudon 1997].)

2.5.5 SGI Origin

The SGI Origin 2000 is loosely based on Stanford's DASH and FLASH projects. As shown in Figure 2.10, each node contains up to two MIPS R10000 processors sharing a connection to a full crossbar (the Hub chip) and there can be up to 512 nodes (1024 processors). The two processors share a bus into the Hub to save pins, but do not snoop on each other. The node is thus not really an SMP, and there is no clustering. This makes the basic node cheaper than an SMP box, which also provides for better incremental upgrade costs. By eliminating the SMP bus and snooping, the Origin also reduces the latency to remote memory. Pairs of nodes share access to an I/O crossbar, which supports direct memory-to-memory DMA between any two memory modules, and also cache coherent I/O.

The interconnect uses a 6x6 crossbar called a Spider chip (similar in performance and architecture to the Synfinity) to implement a fat hypercube topology. The Origin does not provide any kind of cache for remote accesses, instead relying on page replication and migration. The overall goal of the Origin is to be highly modular and cost efficient, while still providing good performance.

2.5.6 Beowulf

Beowulf [Becker 1995] is not really a parallel machine per se, but an attempt to allow anyone to build their own supercomputer (in their parlance, *parallel workstation*) using COTS (commercial off-the-shelf) parts. Following in the footsteps of projects like NOW and Princeton's SHRIMP [Blumrich 1998], its most common incarnation is as a group of PCs connected together by one or more 100 Mb/s Fast Ethernet networks, and running the Linux operating system with modified network drivers to reduce network overhead. On applications that do not require very fine-grain sharing the performance can be quite good, and the cost is at least an order of magnitude less than any other commercial system. Since it does not address RAS at all, it is unlikely to find favour outside of the research community.

2.6 Conclusion

This chapter has briefly covered some of the major issues involved in design and analysis of parallel computing systems. Given this background and some examples of current multiprocessors, the next chapter will describe details of the NUMAchine architecture, as one particular instance of a CC-NUMA machine.

CHAPTER 3

NUMAchine Architecture, Implementation & Simulator

This chapter will describe the architecture and implementation of the NUMAchine prototype, as well as the architectural simulator, which was used as a design, validation and research tool.

NUMAchine is a distributed shared-memory multiprocessor with hardware cache coherence, which puts it in the CC-NUMA class of machines. This has become a popular architectural choice for commercial multiprocessors (e.g. SGI's Origin 2000, Compaq's Alphaserver and HP's V-class servers). The main reason for this is that it is comparatively simple to implement such a machine by connecting together a number of bus-based SMP nodes using either a LAN or SAN network. The ability to efficiently leverage the years of industry experience in SMPs is causing prices to drop and quality to rise, to the extent that CC-NUMA machines are becoming the de facto standard in 'enterprise computing'. (This term usually implies not only high performance, but RAS features as well. Multiprocessors are encroaching on both the supercomputer and mainframe domains.)

The goal of the NUMAchine project is to develop a simple, low-cost and scalable architecture for distributed shared-memory multiprocessors (DSMs) with up to a few hundred processors. The purpose of building the prototype was to verify the feasibility of the architecture in practical terms, and to provide a hardware platform that can serve as a base for operating system (OS) and compiler research. The simplicity is crucial since NUMAchine must be highly cost-efficient[1]. To achieve our cost goal we used COTS parts, and field-programmable devices (FPDs, in particular FPGAs and CPLDs) for control logic instead of ASICs[2]. Design simulations were used to ensure that the choice of parameters such as datapath widths and speeds for the prototype reached the desired level of scalability. Flexibility was crucial to pro-

1. The funding for NUMAchine was provided by an NSERC grant of around CAN$1.3 million. This money paid for both hardware and designers.

viding support for research. The reprogrammability of FPDs facilitated rapid debugging of the prototype, allowing us to avoid rigorous formal verification. At the same time it permitted us to leave space for future enhancements as results from research are obtained. We also designed NUMAchine to provide plenty of low-level monitoring in hardware, allowing software designers to determine the ultimate cause of performance degradation in the system. (The monitoring functionality will not be described in this dissertation. See [Lemieux 1996] for more information.)

The first section of this chapter describes details of NUMAchine's architecture and implementation. The second section will describe the NUMAchine simulator, which is used for system analysis and validation.

3.1 Architecture and Implementation

NUMAchine consists of a number of *stations*, connected together by a two-level network of hierarchical rings, as shown in Figure 3.1. Each station is basically a bus-based SMP node, composed of four processors, a memory module and a network interface card (NIC). Architecturally speaking, there is nothing special about the choice of a 4-way SMP node. The two main reasons for choosing the number four came from both bus saturation and physical implementation considerations. These factors will be discussed in the next subsection.

A major factor in choosing rings for NUMAchine's interconnect has to do with their inherent ordering properties. As described in Chapter 2, hardware cache coherence and memory consistency models must be aware of possible re-orderings of requests and responses in the network. The normal method for guaranteeing ordering is to use a handshaking protocol where all requests require acknowledgments before they can be considered complete. NUMAchine's cache coherence protocol avoids acknowledgments by taking advantage of the ordering and natural broadcast capabilities of rings. This makes the coherence protocol both simpler to implement, and more efficient. The implementation will be described in section 3.1.5, and the performance will be analysed in Chapters 4 and 5.

2. FPGA and CPLD stand for Field-Programmable Gate Array and Complex Programmable Logic Device respectively. Roughly speaking, they both provide arrays of *logic blocks*, with each block configurable to provide simple logic functions. A complex circuit is decomposed into these smaller functions, and then *mapped* into the device. Application-Specific ICs (ASICs), in contrast, use customized logic specific to the particular circuit, and are not reprogrammable.

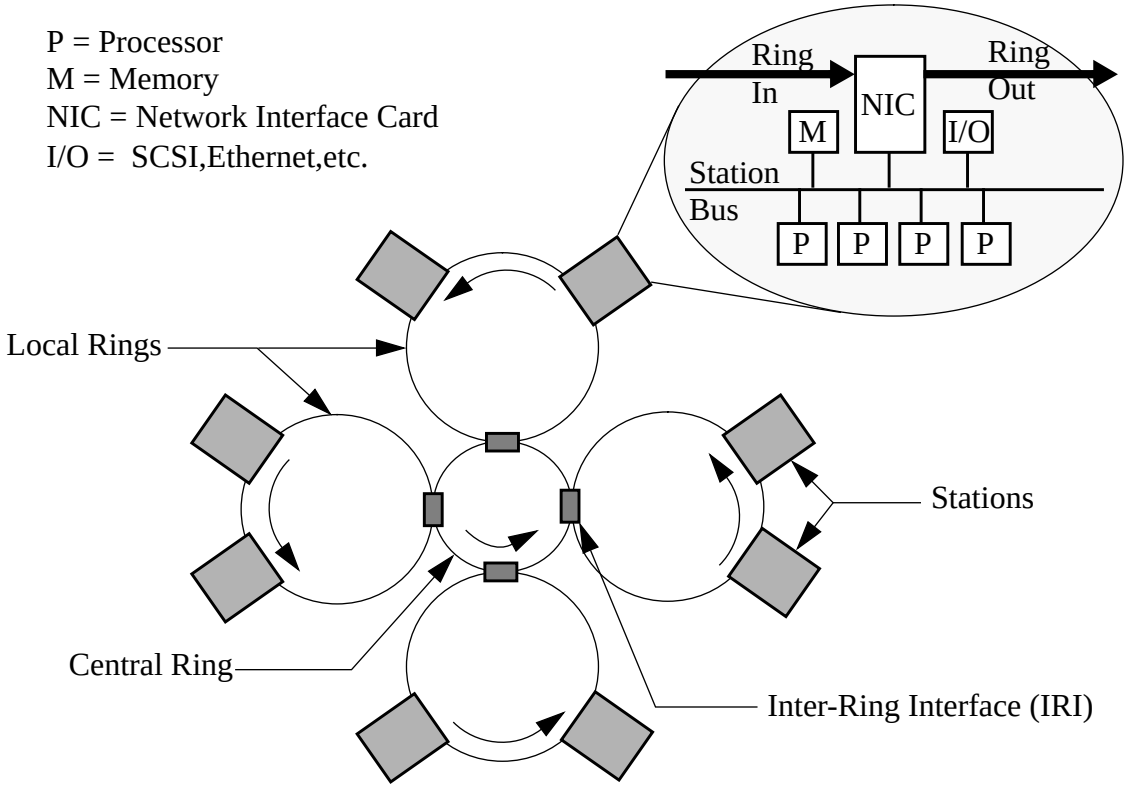

FIGURE 3.1: A high-level view of the NUMAchine architecture. Each Local Ring is shown with two stations, but can contain up to four. With four processors per station, this gives a maximum of 64 processors.

The lower level ring in the hierarchy is called a *Local Ring*, and multiple Local Rings can be connected together by a *Central Ring*. The reason for using a two-level hierarchy instead of a single level has to do with latency and bandwidth considerations. In a single ring, the latency is proportional to the number of nodes in the ring, which does not scale well. In a hierarchical ring, the latency scales as $O(N^{1/L})$ for N processors and L levels of hierarchy, with little increase in the system's cost or complexity since both levels of ring can make use of the same technology. From the bandwidth perspective, a single ring requires $O(N)$ increases in bandwidth to accommodate more processors without degrading performance. Depending on traffic patterns, in the hierarchical case some fraction of requests remain on the Local Ring and do

not need to make use of the Central Ring. In the best case, all traffic would use the Local Ring, resulting in a bandwidth scaling requirement of $O(N^{1/L})$ for the Local Rings (and no scaling requirement for the Central Ring since it carries no traffic). In the worst case of no local traffic, the Local Rings would have no scaling requirement and the Central Ring would require $O(N)$ increases. The actual scaling requirements for real applications lie somewhere between these two limits, but are still an improvement over the single level case.

The use of SMP nodes makes NUMAchine a cluster-based design, meaning that communication is faster if it can take place within a cluster, which also helps with scalability as discussed in the preceding paragraph. Actually, because of the two levels of rings there are three degrees of locality. Accesses are called *Local* if they can be satisfied by memory on the same station as the requester. The latency of a remote (off-station) access depends on whether the Central Ring must be traversed or not. Transactions that use the Central Ring we term *Far Remote*, and those that stay on the Local Ring we call *Near Remote*. The request/response paths on a ring always involve traversing the entire ring (the request uses some of the ring segments, and the response traverses the rest). Thus the Near Remote latency is the same no matter where the requester and responder are situated on the ring. The same is true for Far Remote latencies regardless of which two Local Rings are involved. The difference between the latencies represents the degree of non-uniformity of accesses, or in other words the level of NUMAness of the architecture. A high ratio indicates that the penalty for fetching remote data is high, which makes it worthwhile to spend more effort on trying to reduce the number and latency of remote references. Typical values of the remote-to-local ratio for other CC-NUMA systems are in the range 10:1 to 20:1. As we shall see in the next chapter, NUMAchine's ratio is about 2:1 for near remote, and 4:1 for far remote accesses.

There are numerous techniques for reducing remote access latencies, two such being clustering and caching. We will not consider clustering in this thesis, since it is typically provided at the operating-system level and is complementary to low-level hardware techniques[3]. NUMAchine's hardware supports remote latency reduction by means of the Network Cache (NC). The NC is an integral part of the NIC, and provides on-station storage for remote cache lines which have been fetched across the network. Re-use of the line by other on-station processors incurs only a low local request latency of the same magnitude as a local memory refer-

3. See [Gamsa 1999] for a description of clustering support in NUMAchine.

ence. The design of the NC is described in more detail in section 3.1.2. Its performance is explored in Chapters 4 and 5.

Another factor that affects remote latency is the placement of memory pages amongst the distributed memory modules. The *page-placement policy* directly determines whether memory references are seen as local or remote from a given processor. Techniques such as page migration and replication can help reduce remote latencies [Culler 1999], and are used by systems, such as the SGI Origin 2000, which do not make use of a remote-access cache such as the NC. Since these approaches fall more into the domain of the operating system, we will not consider them in this dissertation. However, we *will* look at different types of page-placement in Chapters 4 and 5 since the placement policy directly impacts the performance of the NC. (It will also be shown that, for simulation purposes, some of the benefits of page migration and replication can be mimicked by an appropriate choice of the paging policy.)

At the architectural level, the final aspect of NUMAchine is the I/O subsystem. Multiprocessors are often used to run very large programs that require substantial I/O bandwidth. NUMAchine provides support for a *parallel file system* which takes advantage of large numbers of smaller disks to increase overall bandwidth and throughput. The implementation of the I/O card will be described in the next subsection, but the details of the parallel I/O support in the operating system will not be covered. The interested reader can refer to [Krieger 1997].

With the architecture more or less fixed, we then moved on to the implementation stage, which consisted of four basic steps:

1. Simulation studies to determine system parameters, such as queue depths and bus widths.
2. Partitioning of controller and datapath logic into COTS chips and/or FPDs.
3. CAD schematic entry and writing of FPD code, using a Hardware Description Language (HDL).
4. Board-level simulations using the Cadence Logic Workbench digital simulation tool to verify functionality.

NUMAchine made use of FPDs exclusively from the Altera family of devices, in order to minimize the number of tools required. While the Altera devices themselves were only marginally better than the nearest competitor, Xilinx, Altera's development environment was easily the best-of-class at the time. Given the complexity of designing such a large and intricate system, good tools turned out to be the most critical aspect of our design effort. We estimate

that the Cadence toolset required on average six months of daily usage before a designer achieved proficiency. Part of this was due to a long list of interoperability problems and bugs in the tools, which is not surprising given the complexity of the tools themselves. However, once mastered, our board-level simulations turned up many design problems that would have taken months to find and fix after the cards had been manufactured.

The next three sections will describe the implementation of the various system components.

3.1.1 Station: Bus, Processors, Memory and I/O

The prototype version of NUMAchine's processor card uses a MIPS R4400 processor operating at 150 MHz with a dedicated 1 MB secondary (L2) direct-mapped unified instruction/data cache implemented using off-the-shelf SRAMs. The MIPS family was chosen because it provided solid support for 64-bit and multiprocessor operations, and already had a number of years of commercial exposure. We also designed NUMAchine to support the next generation of the MIPS family, the R10000, which provides newer architectural features such as a superscalar execution unit and prefetching. However, we have not implemented this design, because the price of the R10000 was over US$5000 per chip.

The processors also have 16 KB separate primary (L1) instruction and data caches on-chip. The R4400 contains cache-control circuitry to manage both the L1 and L2 caches using a MESI protocol[4]. This control logic is responsible for maintaining both inclusion and coherence between the two cache levels[5], as well as dealing with external coherence requests such as interventions. (An intervention is a request to read a dirty cache line from a processor's L2 cache, and there are two types, shared and exclusive. A shared intervention allows the line to stay in the owning processor's cache, but changes its state to shared. An exclusive intervention forces the owning processor to invalidate its copy after returning the data.) The MIPS R4400 can have at most one outstanding load/store at any given time.

4. See [Heinrich 1994] for a description of the R4400 protocol details, and [Culler 1999] for a general description of MESI and other protocols.

5. Inclusion means that a cache line in L1 *must* be contained in L2. In practice this means that a line ejected from L2 must also be ejected from L1 to maintain the subset property. Lines ejected from L1 need only be flushed back to L2 if they contain modified data.

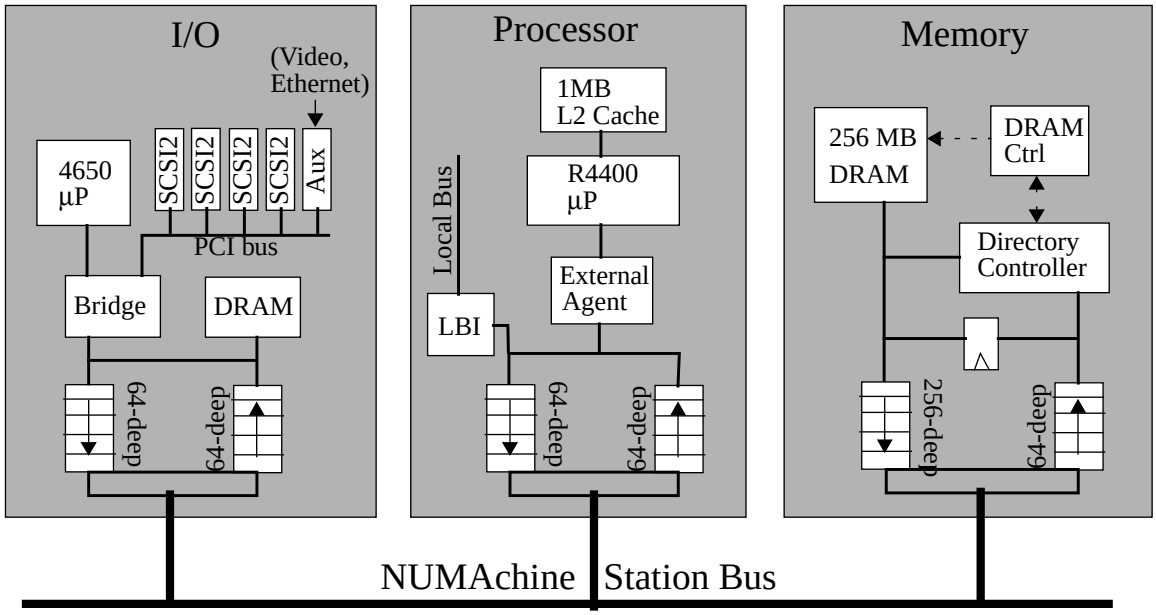

FIGURE 3.2: Cards on the station bus. The I/O card contains a bridge controller, which interfaces between a MIPS 4650 embedded processor, DRAM for I/O staging buffers and a PCI bus. The PCI bus can contain up to four SCSI-2 controllers, and one auxiliary connector for a commodity PCI card (e.g. Ethernet or video). The I/O card also contains the bus arbiter (not shown). The processor card has the MIPS R4400, plus a Local Bus Interface (LBI). The Local Bus provides a DUART, boot EPROM and an off-board debugging connection. On the memory card, the directory controller maintains cache coherence, and interfaces to the DRAM controller, which manages the flow of data into and out of the 256 MB of 4-way interleaved DRAM.

All datapaths in NUMAchine (busses and rings) are 64 bits wide and are clocked at 50 MHz, providing peak bandwidth of 400 MB/s. This choice of width for the datapaths matches the width of the external memory interface of the MIPS processors, and makes the design of the datapaths simpler. During pre-prototype simulations we found that the bus came close to saturating for certain applications, so we looked at changing the width of the bus to 128 bits in the simulator. While this did help, the improvement was only on the order of 5% for overall execution time. The extra cost of bus drivers and complexity in the datapath was not justified given our low-cost target and the minimal performance enhancement.

Figure 3.2 shows the internal layout of the datapaths for the I/O, memory and processor cards. The External Agent on the processor card is a 3-CPLD controller which is responsible

for converting the R4400's system interface bus requests into NUMAchine commands, and also doing some simple encoding/decoding of the NUMAchine address space. The LBI is a low-speed auxiliary bus on which resides a DUART, EPROM containing bootstrap code for the processor, and an off-board connection to allow initial debugging and network access in the absence of an I/O board.

As mentioned above, the station bus uses a Futurebus+ backplane as the physical bus medium. The Futurebus+ specification also includes a full bus protocol, but we decided that this was overly complicated for our needs, so we designed a custom split-transaction protocol. Splitting the transactions means that requests and their associated responses use separate bus transactions. While this allows for more concurrency, it also makes the coherence protocol more complicated.

Addresses and data on the bus are multiplexed onto the same lines. For each valid cycle on the address/data (A/D) bus, a parallel set of 16 lines provide command and control information. Another 8 lines provide data integrity and error detection on the A/D bus. The R4400 can be configured at boot time to use either parity-checking or a more robust error-correcting code (ECC) scheme called *single-error-correcting/double-error-detecting* (SECDED). (See [Heinrich 1994] for details on the ECC.) Parity can detect single-bit errors but cannot fix them. While much weaker than ECC, parity has the advantage that it is much simpler, and most commodity components such as FIFOs and buffers come in versions that include parity-checking circuitry. This allows data integrity to be checked at various points along the datapath, aiding in diagnostics.

The memory card contains up to 256 MB of 4-way interleaved DRAM. The interleaving provides enough pipelining to allow the DRAM to feed the outgoing queue at its maximum rate of one doubleword (8 bytes) every clock cycle. The directory and DRAM controller are also pipelined to allow input request processing to begin while the DRAM is still processing the previous request.

The directory controller maintains the hardware cache coherence state tables. The controller is implemented in 3 CPLDs, which take care of predecoding commands, directory lookup and state transition generation. Placing the entire coherence engine in FPDs was invaluable during the debugging stage. A handful of bugs which managed to slip past the simulation validation runs required only a 30-second reprogramming of the controllers to fix. In fact, at one point we realized that the diagnostic unit on the memory card did not generate parity, which caused the processor to take a parity exception. We added a new command bit

instructing the processor to ignore parity for data only in the specific command. The entire fix to both memory and processor logic took us under half a day.

Not shown in the figure for the memory card is an independent controller called the Special Functions (SF) unit. The SF provides diagnostic information on the memory card, and also provides a block-transfer engine which is tightly integrated with the coherence controller. The SF is basically a coherence-aware DMA engine. The SF is capable of gathering up all cache lines for an arbitrarily large block of memory. Since these cache lines may have dirty data elsewhere in the system, the SF first fetches them and cleans up the directory state, then ships the block out to either an I/O card (for paging) or another memory card (for page replication and migration). At the time of writing, the SF functionality has not been fully integrated into the OS, and so no performance results for the SF will be presented.

And, finally, the I/O card provides disk access, and optionally the ability to plug in a standard PCI Ethernet or video card (for fast frame buffer graphics). The goal of NUMAchine's I/O subsystem is to provide large amounts of parallelism to the file system. The four SCSI-2 controllers can each support numerous disks, although realistically the maximum is around four per controller. As with the SF unit, no performance results will be presented for the I/O card.

3.1.2 Station: Network Interface Card and Network Cache

The NIC provides the connection between the on-station bus and the Local Ring. As shown in Figure 3.3, the NIC consists of two datapaths from the bus to the ring and vice versa. Both datapaths share access to the on-board Network Cache (NC) on a first-come/first-served basis.

Since NUMAchine uses a slotted-ring instead of token-ring protocol, cache lines (of size 64 or 128 bytes) are fragmented into packets for transmission and can be interleaved with incoming packets from other transactions. This necessitates re-assembly of multi-packet transfers back into blocks on the receiving end. The packet re-assembly area shown in the figure consists of 350 KB of SRAM and a CPLD controller. The memory-mapping scheme for the re-assembly area makes use of the fact that each processor can have at most one outstanding request and up to four writebacks active in the system at any one time. Using the sending station's ID in the re-assembly address guarantees that no two data packets can conflict. The

Sinkable and Nonsinkable FIFOs in the diagram are used to implement separate virtual request/reply networks, and will be described in more detail in section 3.1.4.

The synchronizing FIFOs allow the ring to be run at a different clock rate than the stations. Firstly, this allows us to speed up the ring in the case that it proves to be a bottleneck. (It turns out that the Central Ring can saturate and cause problems for certain applications. The Central Ring is designed to handle higher clock speeds as discussed in the next section.) Secondly, clocking in such a large system is difficult to do with low skew. Our clock distribution scheme uses an independent source on each station which generates all on-station clocks. Thus there is no guaranteed phase relationship between stations. The synchronizing FIFOs are

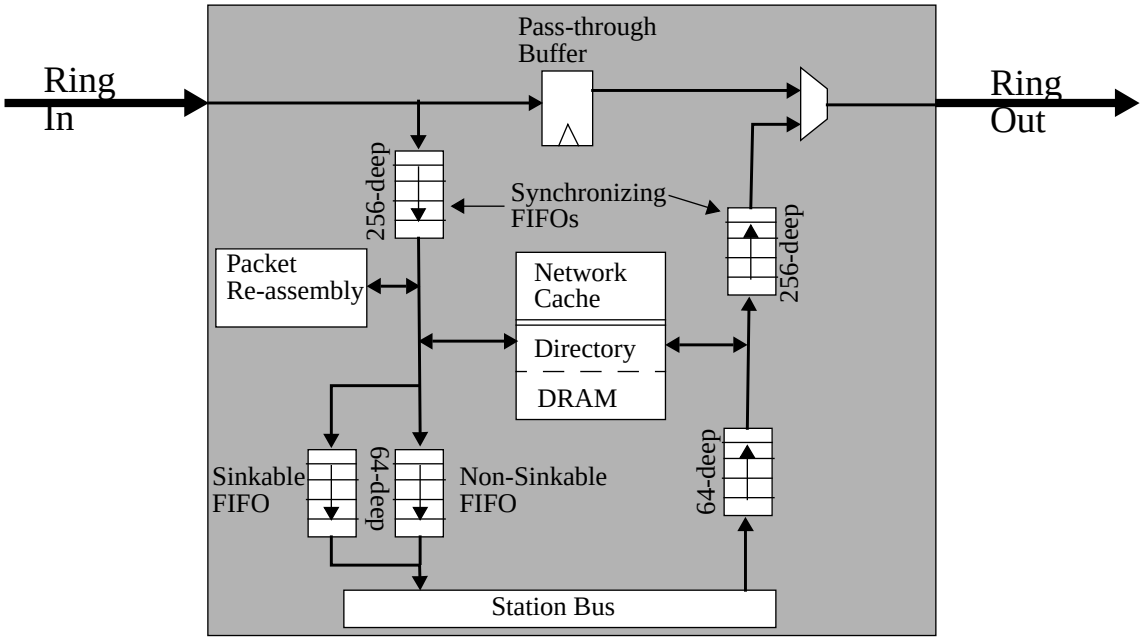

FIGURE 3.3: The NUMAchine Network Interface Card (NIC). This figure gives a simplified overview of the NIC datapaths (no control logic is shown). The synchronizing FIFOs allow the clocks for the ring and the rest of the card to run at different speeds. Cache lines are broken into slot-sized packets on the ring, and so require re-assembly at the receiving end. The Network Cache is time-shared between the incoming and outgoing data paths. The Sinkable and Nonsinkable FIFOs have to do with flow control, as explained in section 3.1.4.

designed to handle different input and output clocks, with internal circuitry to avoid metastability[6].

The NC is implemented using four CPLDs for control, 512 KB of SRAM for directory information, and 8 MB of Synchronous DRAM for cache line storage. Our initial simulation results indicated that the knee of the performance curve for the NC lay at about 4 MB for the prototype. However, the only SDRAM chip geometries available at the time meant that our NC had to be either 2 MB or 8 MB, so we chose the latter.

3.1.3 Rings

The architectural reasons for choosing rings were discussed above. From an implementation standpoint rings are also convenient. Firstly, rings are simple to design and implement as they use only point-to-point connections. Secondly, routing on a ring is simple and fast. In NUMAchine, a structure called a *filtermask* (described in more detail below) is used for routing. At each hop in the ring a single bit in the filtermask indicates whether the packet in the current slot has reached its destination or not. The filtermask also provides support for multicasting at no extra cost. As we shall see in Chapter 5, multicasting is put to good use by the coherence protocol.

The rings are unidirectional and utilize a slotted-ring protocol. This means that for a four-segment ring there are four slots that move one hop forward in each 20 ns clock cycle. A given slot can be either full or empty. If empty, then a NIC with data waiting to be sent out can fill the slot as it goes by. If full, the slot's data is either consumed by the NIC or passed along[7]. On a given clock tick, if a packet is consumed by the NIC then the slot is freed up. NUMachine has an option in the hardware (configurable by the OS) to use the newly freed slot for outgoing packets from the same NIC. In a lightly loaded ring this has the advantage of reducing the average latency for ring access by slightly more than one clock tick. (The alternative is to wait for the next slot to come by. Under light loading this next slot has some small probability of

6. Metastability can occur when the data input to a flipflop does not meet the flipflop's setup- and hold-time requirements. The output of the flipflop hovers at some indeterminate voltage, which is neither a valid logic '0' or '1', for a random period of time.

7. For a multicast/broadcast packet, it is possible to do both. The effect is to split off a copy of the packet for the local NIC while also allowing the original to continue around the ring.

being occupied, thus the savings on average are slightly greater than one.) The use of the just-freed slot also allows insertion into the ring even in the case of a constant flow of upstream packets that are consumed by the NIC, which would otherwise lead to starvation due to the lack of free slots from upstream. On the other hand, the just-freed slot option can cause starvation problems for the downstream NIC if the network is heavily loaded, since in this case each NIC greedily uses any free slots that become available. Simulation results in the next chapter will show that use of the just-freed slot is on average slightly beneficial.

The Inter-Ring interface (IRI), as the name implies, joins the Local to the Central ring, and is shown in Figure 3.4. The architecture is simple, with an up and a down queue, and some control logic (not shown) to perform the ring functions and flow control. (Flow control is discussed in section 3.1.4.) As mentioned above, the Central Ring was designed to allow higher speed operation. To achieve this, the four IRIs necessary to implement the Central Ring for the 64-processor prototype were combined into a single printed-circuit board (PCB). Thus in the figure, the Central Ring connections are actually traces on the PCB, while the Local Ring connection is made using cables (which connect to NIC cards on either side). Our CAD simulations indicate that the Central ring should be able to run at around 75 MHz.

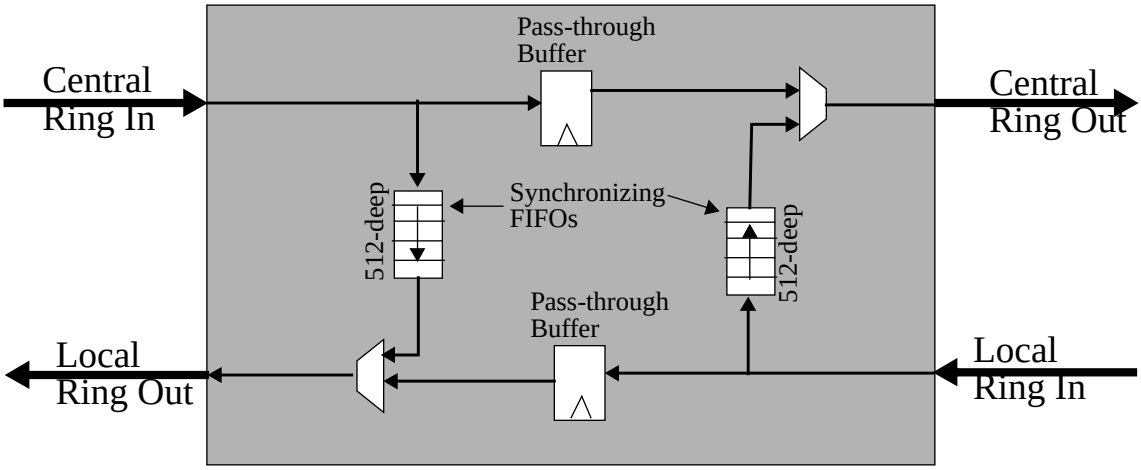

FIGURE 3.4: The Inter-Ring Interface (IRI). This interface does simple buffering and flow control on both the Local and Central Rings. The synchronizing FIFOs allow the upper and lower level rings to run on different clocks.

Much more detailed information on the NIC and ring implementations is available in [Loveless 1996].

3.1.4 Flow Control and Deadlock Avoidance

NUMAchine uses flow control as a means of buffer management to ensure that packets are not lost due to overflow. Deadlock can occur if there is a circular dependence amongst resources stalled by flow control. Consider the example of a memory's input queue, which is stalled due to a large number of incoming requests. Some of these requests require interventions to a local processor, each intervention requiring space in the output queue. The processor, however, is also stalled due to a large number of previous interventions. In the processor's case the stall occurs because the output queue is stuck waiting for memory to become free to accept the intervention response data. Neither the processor nor the memory can make any forward progress in this case. NUMAchine avoids deadlock through use of a throttling mechanism for flow control. In addition, the NIC's sinkable and nonsinkable queues present separate request and response paths, allowing responses to bypass requests in certain cases that would otherwise lead to deadlock. (In all other parts of the system requests and responses use the same paths. This is described in more detail below.)

Even if there is a theoretical upper bound on the number of requests and responses existent in a system at one time, it is not practical to provide enough buffering to accommodate the worst case[8]. There are thus two basic mechanisms for ensuring that data is not lost due to buffer overflows. In the first scheme, send buffers remain allocated (keeping copies of sent data) until an acknowledgment has been received. If one has not been received after some time-out period, the buffer is re-sent. The time-out must be large enough to accommodate the largest possible time for an acknowledgment, so this scheme pays a high performance penalty for buffer overruns, but works well if the probability of such overruns is small. It also allows the buffers to run near capacity. This approach also requires some extra complexity to ensure that livelock does not occur.

8. Each R4400 processor can have at most one outstanding load or store. It is possible, though, for each processor to flush back dirty cache regions, causing a large number of writebacks to flood the system.

A second approach is to use back-pressure or throttling. A receive buffer that is nearing capacity sends a signal to all possible transmitters to stop issuing packets. No timers are needed using this scheme because buffers are never allowed to reach the overflow point. The drawback is that the buffer must leave enough room to handle all possible in-flight packets that might be issued before the throttling can take effect. This can lead to very inefficient average use of buffer space unless the throttling time is fast. Another shortcoming is that all senders are stopped, even if they are not targeting the specific receiver.

NUMAchine utilizes the throttling method because the lack of acknowledgments and timers makes it simpler to implement, and also because commodity buffers are cheap, meaning that inefficient buffer utilization does not carry a heavy premium in dollar terms. Each level in NUMAchine's hierarchy uses flow control with its immediate connections. Before discussing the details of the flow control scheme, we must discuss deadlock avoidance.

A standard approach to handling deadlock in multiprocessor systems is to use separate request and reply networks. To actually physically run two networks in parallel is not practical, so what is usually done is to use separate virtual channels. The same physical data links are used by both, but they have separate buffering and flow control. This is the approach taken in NUMAchine.

Instead of separating the request and response paths, NUMAchine uses *nonsinkable* and *sinkable* transactions, which correspond roughly to requests and responses respectively. A sinkable transaction is one which is known not to generate any kind of further traffic after having been received by a communications endpoint (i.e. a processor, memory or I/O). Write-backs and data responses fall into this category. The key characteristic of sinkable transactions is that their receipt in an input queue will not require any space in an output queue. Nonsinkable transactions such as reads and interventions will usually (unless they are negatively acknowledged) require some means of ensuring that enough space is available in the output queue to handle a cache line's worth of data before they can be accepted.

NUMAchine uses separate flow control mechanisms for sinkable and nonsinkable transactions. Sinkable transactions are stalled only when input queues are nearly full[9]. Since sinkable transactions are guaranteed to be consumed, their processing can continue independent of any output buffering stalls, which means that forward progress is guaranteed even if every

9. The commercial FIFOs used have programmable flags which indicate when more than some specified number of entries in the FIFO have been used.

other part of the system is stopped. In general there are two approaches, both of which are used at various points in the system. The first is to use separate queues for the sinkable and nonsinkable paths. Duplicating the queues is rather expensive, though, so it is only used on the NIC card (as shown in Figure 3.3). This allows sinkable transactions to bypass nonsinkable ones if the latter are stalled. This bypassing feature on the NIC is necessary because it is not a datapath endpoint. For the processor and memory cards, which *are* datapath endpoints, the method used is to have a single queue for both types, but to count the number of nonsinkable transactions that have arrived, and only allow in as many as could possibly cause the output queue to nearly fill up in the worst case.

The processor card uses 64-entry, 8-byte wide queues for both input and output in the prototype. The default system cache line size of 128 bytes uses 17 entries in a queue (16 data + 1 command). The nonsinkable busy counters for processor are thus set to 2, leaving a margin of error of one nonsinkable request in case the bus arbiter gives a grant before the busy signal is recognized. On the processor, having a maximum of two pending interventions is fairly reasonable, since it is unlikely that a single processor is likely to be the target of more than two requests during the time required for a processor to service an intervention. (Note however that the intervention latency is variable, from 8-28 processor cycles. This is due to the fact that the L2 cache controller needed to process the intervention may be busy with internal processor activity.) The memory, since it is likely to be the target of many more simultaneous requests, uses 256-entry queues on the output, and the same 64-entry queues as the processor on the input. The memory's nonsinkable counter is thus set to 8, which will be shown to be more than adequate.

On each ring (both Local and Central) there is a single Stop_Ring signal that is asserted should the queues fill past the 3/4 mark, disallowing insertion of new packets into the ring. No differentiation is made between sinkable and nonsinkable packets, because on the rings we are dealing with single packets, not entire cache blocks. Also, there is enough buffering in the nonsinkable queue on the NIC card, so the only likely cause of a Stop_Ring condition is a flood-of-writes coming through the system. The NIC's ring queues are 256 deep in both the incoming and outgoing directions, while the IRI uses 512-deep queues. This corresponds to 8 and 16 cache lines in the respective queues before ring flow control is triggered. Even under bursty conditions we will show in the next chapter that the ring flow control is only infrequently activated.

3.1.5 Hardware Cache Coherence

This section will give a brief synopsis of NUMAchine's hardware cache coherence scheme. A complete description can be found in [Grbic 1996]. The cache coherence scheme is descended from NUMAchine's predecessor project, Hector, as described in [Farkas 1992].

NUMAchine's cache coherence uses a writeback-invalidate protocol and a full directory which is broken into two levels: home memories and the NCs. The home memory stores the coherence state (described later) and two bitmasks. The first, called the *processor mask* or *Pmask*, indicates which of the four on-station processors potentially has a copy of a cache line. The Pmask may be conservatively inaccurate, because there is no notification of shared cache line ejections, so a processor marked as a sharer may not currently have a copy. This can lead to a case where processors receive unnecessary invalidations, but since the invalidation is sent out in a single atomic multicast on the bus there is no extra bus traffic generated[10]. In addition, invalidations are very low-overhead operations in the L2 caches, so sending too many does not incur much of a penalty as long as it does not happen too frequently.

The second bitmask is called the *filtermask* or *Fmask*, and it stores the same type of information as the Pmask but for remote copies. This is an inexact coarse-grained indication of which rings and stations (not processors) in the system contain copies. From Chapter 2 we know that storing a full bit-vector causes the number of directory bits to scale as O(*N*) for an *N*-processor system. To reduce this cost, we split the Fmask into two pieces, which we refer to as the *ring portion* and *station portion*, corresponding to the two levels of hierarchy as indicated in Figure 3.5. To understand the working of the Fmask, first consider each of the Local Rings to be a single object, ignoring that it is actually composed of stations. If Local Ring *R* (*R*=0,1,2,3) contains *at least* one copy of a given cache line, then we set bit *R* in the ring portion of the Fmask. For the station portion, we set bit *S* if station *S* on any of the Local Rings contains a copy. As shown in Figure 3.5, imprecision arises when two or more rings have different sets of sharing stations. Any pattern of sharing constrained to a single Local Ring is precise, as is any sharing on multiple Local Rings if the set of sharing stations is identical for each ring. All other patterns will include some imprecision. (Note that a dirty copy has a single owning station, so it is always precise.)

10. This is a multicast, not a broadcast, because only those processors with bits set in the Pmask are targeted. If no bits are set (all shared copies are remote), then no invalidation is sent out.

The advantage of this scheme is that it reduces the scaling of the directory storage to $O(N^{1/L})$, where L represents the number of levels of hierarchy. The drawback is that imprecise masks can cause unnecessary invalidations. The worst possible case is where a single different

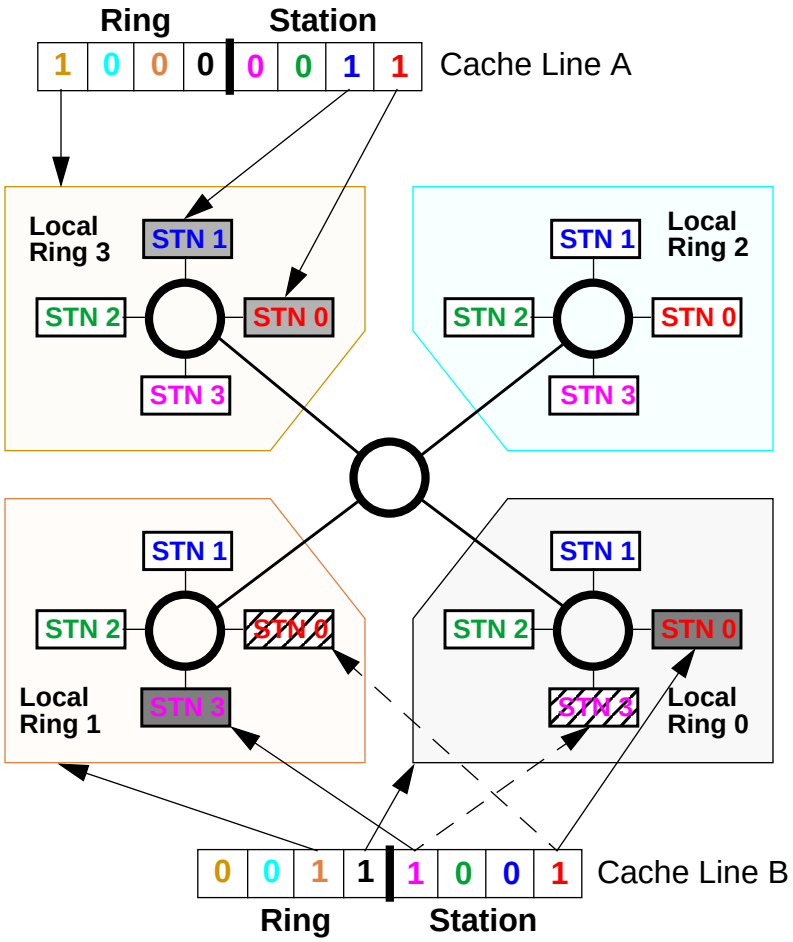

FIGURE 3.5: The NUMAchine filtermask. Colours are used to show the bit correspondences. A bit in the ring half of the Fmask is set if one (or more) stations on the Local Ring of the same colour contains a copy. The station half is similar, with a set bit corresponding to one (or more) of the same-coloured stations The filtermask shown at the top of the figure is precise, since each set station bit corresponds to a true copy of the line. (True copies are shown shaded, false copies are hatched). The filtermask at the bottom is imprecise, since it includes stations which do not actually have copies.

station on each Local Ring contains a copy. This leads to a full Fmask with all bits set, selecting all 16 stations instead of the four that really have copies. There are two reasons why this is not as problematic as it might seem. Firstly, we have already mentioned that invalidations are not very costly. Secondly, typical sharing patterns involve one, two or all processors, and only rarely numbers in between [Culler 1999]. Thus, the frequency with which we send unneeded invalidations to a station is low. The main advantage to the filtermask from an implementation standpoint is that it allows very simple and fast routing and multicasting on the rings. In the following chapter we will look at just how often the filtermask overspecifies stations.

The NC forms the second level of the coherence directory. While the filtermask described above determines which stations in the system have copies, the NC on a station keeps track of all local copyholders using its own Pmask. Because the NC is organized as a cache, it is possible that a line, including the directory contents, may have to be ejected[11]. There are two general approaches to handling this situation: strict and lazy. In the strict approach we must correctly update the home memory directory to reflect the loss of NC information. For a line which shows up as shared in the NC, we would first invalidate all local copies, and then send a message to the home memory requesting that it remove the station from the sharing list. For a dirty copy on-station, the NC would have to make sure that the line was flushed back to memory. In contrast, the lazy scheme simply allows the directory information to be thrown out without informing the home memory[12]. In certain cases, extra states are added to the coherence protocol to handle the situation where directory information for a requested cache line has been discarded. For invalidations to shared lines the extra processing is minimal: the invalidation is simply broadcast to all processors on the station. For dirty lines, the NC must send out intervention requests to all on-station L2 caches and wait for the responses before it can take action, which is costly. However the overall cost may not be excessive if this situation occurs rarely, and the trade-off is that we avoid operations that, while less costly, take place on *every* ejection, and thus occur frequently. NUMAchine uses the lazy scheme, and simulation results in Chapter 4 will show that this choice is a good one.

The coherence state maintained for each line is different in the memory and the NC. The memory state requires two bits for the state, plus eight bits for the Fmask. One bit of the state

11.Note that lines in the NC are ejected only on capacity or conflict misses. Coherence misses only affect the state of the line.

12.If the NC (and not a local processor) is the owner of a dirty line, it must first issue a writeback to the home memory.

information indicates whether the line is *locked* or not. Locking of the line occurs at the beginning of a coherence action that requires multiple stages, and ensures that no other access to the line can take place until the previous transaction completes. The second bit indicates whether the current state of the line is shared or dirty, or in NUMAchine terms Valid or Invalid, respectively. (Valid or invalid here refers to the state of the memory's or NC's data, not the processor's. Thus, Invalid means that the memory or NC does not have valid data, and that the modified data resides in some local processor's cache.) There is a third implicit state bit derived from the Fmask. If the Fmask contains bits set for any stations/rings other than those of the home memory, then there are possibly remote copies, and the line is said to be Global; otherwise the only possible copies are on the station, and the line is called Local. Ignoring the Locked bit which is set independently, there are then four states for a line in home memory:

- Global Valid (GV) - One or more remote stations requested a shared copy of this line at some point in the past. Write access to the line can only be gained by first invalidating all of these potential copies.

- Global Invalid (GI) - A dirty copy of this line is owned by some remote station. Read or write accesses must be redirected to the remote owner.

- Local Valid (LV) - The only potential shared copies of the line are in (home memory) local processors. Write access (local or remote) can be gained by first multicasting an invalidate on the station bus only, since there are no remote copies.

- Local Invalid (LI) - A local processor (as recorded in the memory Pmask) has a dirty copy and ownership of the line. Read or write accesses can be satisfied by means of local bus intervention operations.

Any one of these states can also be locked, which we signify by prepending an 'L_', so for example L_GV is Locked Global Valid. Figure 3.6 shows a simplified sequence of transactions taking place for one particular cache line. In most coherence protocols, invalidates require acknowledgments from all the remote targets because of the possibility of requests bypassing each other in the network. This ensures that all remote nodes are guaranteed to have seen an invalidate before more coherence traffic can target a line. In NUMAchine, however, we can take advantage of the fact that in a hierarchical ring topology there exists only one single path between any source and destination. This means that there is no way for two requests

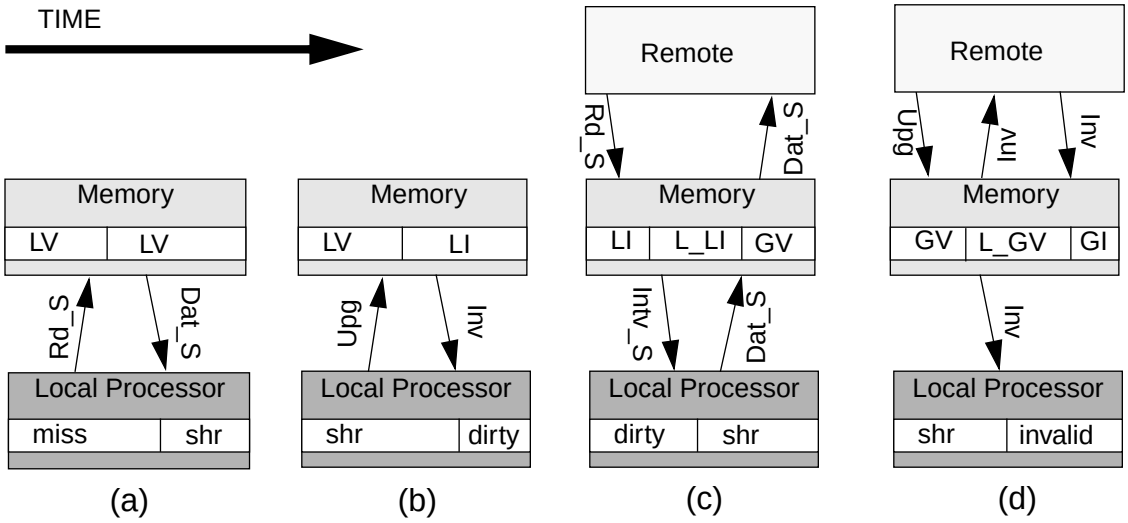

FIGURE 3.6: Coherence actions at the home memory. The memory starts off in state LV, with no copies in the system. (a) A local processor misses and does a shared read (Rd_S) which returns shared data (Dat_S). The state stays LV. (b) The processor writes to the shared line, issuing an upgrade (Upg). Memory responds immediately with an invalidate (Inv). The memory state switches to LI, and the processor's copy becomes dirty. (c) Some remote node does a shared read. Memory locks the line and sends out a shared intervention (Intv_S) to the processor, which responds with shared data and changes its state. Memory forwards the data to the remote node and changes its state to GV. (d) The remote node now writes to the line. The upgrade causes memory to lock the line, and send out one invalidate to the local processor, and another invalidate which traverses the rings, acknowledging the remote node and ultimately returning to memory and switching the line to GI.

from a specific source to arrive in a different order, no matter what the destination. This, in turn, means that an invalidation which has gone around the ring (or rings) and returned to the home location guarantees the ordering, as seen by *any* processor, for future coherence actions to the line. These future accesses must wait for the returning invalidate to unlock the line, and it is not possible for such accesses to generate a request that overtakes a previous invalidate. This ordering property greatly simplifies the design and implementation of the coherence protocol.

State information contained in the NC complements that in the home memory. The set of states is similar to those for main memory, with the following additions and modifications:

- Notin Tag (NT) - This is not a true state, but arises when the request's address does not match the address tag of the current occupant of the line in the NC (i.e. we have a cache miss). This miss can occur either because the request is the first reference to the line (cold miss) or because the information was previously ejected (capacity or conflict miss).

- Notin State (NS) - This state indicates that the NC contains no useful information on the line, although the tag does match. This state is needed only in a rare corner case, where a miss in the NC results in a remote request which eventually gets a negative acknowledgment (NACK). While the tag matches, nothing else is known about the line.

- Local Valid (LV) - This is the same as the main memory state, except that in this case the NC is the owner of the most up-to-date data (stored in the NC DRAM). Remote interventions can be satisfied by the NC directly, without querying the processors. Upon ejection, a line in the LV state must be written back to memory.

When the NC processes a response, there are two locations that require an acknowledgment: the home memory, so that the line can be unlocked, and the original requester. If the original requester happens to also be in the home memory station, then a single response can be multicast on the home memory's bus to satisfy both requirements. Otherwise, there are two approaches. One is to send the response back to the home memory, and let it forward a copy to the requester. This is often referred to as a 3-hop scheme, because the network gets used once for the original request, and twice more for the two stages of the response. NUMAchine uses an optimized 2-hop protocol, whereby the NC is responsible for sending responses to both. It does this by inserting the response into the output queue to the ring twice[13]. In order for coherence to be maintained, it is important that the response to the home memory be the first into the queue. If this were not the case, it would be possible for the requester to see its response while the other response is stuck in the network. The requester could then send another request that could possibly arrive at the home memory before the first response.

13.Note that we cannot use a multicast, because depending on the home memory's and requester's Fmasks, ORing the two may result in an imprecise Fmask, which would erroneously target two extra stations.

3.1.6 Retry Mechanism and Negative Acknowledgments

Since cache lines can be locked by the coherence scheme, a mechanism is needed to handle requests that hit to a locked line in either the memory or NC. The best approach would be to queue up all such requests and service each one in order when the line eventually becomes unlocked. This would guarantee both fairness and correctness properties, but requires more resources and logic in order to implement the pending queues.

Instead, NUMAchine uses a binary exponential backoff, similar to that used for media access control by Ethernet. Such a scheme is known to be stable for low levels of congestion [Goodman 1988].[14] In the NUMAchine version, any request to a locked line is immediately sent a NACK. Upon receipt of the NACK, a processor waits for some period of time before retransmitting the request, giving the line a chance to become unlocked. With each successive NACK, the waiting period is doubled. After some maximum number of retries is reached (64 in the prototype), the request is deemed to have failed, and a bus error is signalled to the processor. The backoff mechanism does not guarantee fairness, because there is no priority given to requests that have been forced to retry a number of times. (This would in any case require breaking the lock on a cache line to allow preemption, making the cache coherence protocol much more complicated.) In the extreme case, lack of fairness can lead to starvation, which in our case is fatal, because the bus error will cause the associated process to be killed even though no fault has occurred in the system.

Early simulation studies and debugging on the prototype showed that the binary exponential backoff approach did indeed suffer from starvation problems. Our initial attempt at a fix involved changing the backoff so that it reached a saturation point, or plateau. After some fixed percentage of the maximum number of retries had been issued, the backoff interval was held constant, giving all processors that had already retried a number of times an equal probability of accessing the line. This did not solve the problem, though, because new requesters could still starve out an old requester. Our solution was to change the plateau value from being the latest (longest) interval to being the initial (shortest) interval. After 32 retries using the binary backoff, the processor then issues the rest of the retries in quick succession to increase the probability of grabbing access to the line as soon as it becomes unlocked. This effectively gives requests with high retry counts a higher priority. One other feature is the use of different

14. Stability in this context means that the number of retries is bounded, i.e. the request will eventually get through.

values for the longest backoff interval for local and remote requests. Remote requests have longer latency for both data and coherence actions, and may require longer retry times. Under our original plateau scheme we allowed the maximum backoff interval to be two or four times as long for remote requests. With our modified scheme, we use the same maximum for both, in order to make the control logic simpler.

While this modified backoff mechanism still does not guarantee fairness or starvation-avoidance, in practice both simulations and programs running on the prototype indicate that the technique works, although the lack of fairness still causes performance degradation. The backoff performance will be investigated in the next chapter.

One possible enhancement to the backoff mechanism is to use a form of request combining, whereby multiple requests to the same address can be merged, and the responses delivered simultaneously. In NUMAchine, combining could take place in the memory and NC for certain types of locked lines. For example, a shared request to the NC that finds the line locked due to a previous, as-yet-unanswered shared request for the same line can be combined. The response (or NACK) can go to both. (Note that combining only works for shared requests.) This guarantees the promptest possible response for the second request, while saving bus bandwidth by reducing the number of retry NACKs. Only small changes are required to the directory maintenance logic to store any subsequent requesters in the Pmask. Combining on the home memory station is somewhat trickier. If the memory module generates the response, then the same approach as the NC will work. However, in the case where the response comes from a remote intervention, the NIC card sends the response directly to the requesting processor and memory. When the second request comes into the memory, there is no way for it to notify the NIC that when the response finally arrives, it must add another target. In this case we could modify the memory coherence controller so that the intervention response both unlocks the line and also forwards the response to the second requester. The only problem with this approach is that a sinkable transaction (the intervention response) could generate a response, which would require modifications to the flow control scheme. We will measure the possible benefits of adding combining in the next chapter.

3.1.7 Memory Consistency

As mentioned in Chapter 2, NUMAchine supports the sequential consistency model, which is

the most intuitive model for writing shared-memory programs. The main reason for choosing this model is that NUMAchine's architecture inherently provides simple and efficient support for sequential consistency, as explained below. Another reason is that the MIPS R4400 processor is not designed to support more aggressive consistency schemes such as release consistency. The R4400 is a non-superscalar microprocessor with blocking caches[15]. (As mentioned before, the original design targeted the MIPS R10000 which *can* support more aggressive schemes, but this newer chip was beyond our budget.) The R4400 *does* have a write buffer, but it is only used for uncached writes, and it is only one word deep. Thus all accesses from a given processor will issue in order. This means that the weakest form of consistency naturally supported by the hardware is processor consistency. (Weaker forms of consistency are possible if the coherence scheme and programming model are changed to allow writes to proceed optimistically, and then providing some means of merging conflicts. This would have led us too far astray from our basic principle to keep the system as simple as possible.)

In analysing the impact of processor versus sequential consistency, we found only minimal differences. The natural sequencing and broadcast capability of the ring are the reasons that the two consistency schemes do not differ greatly. Figure 3.7 shows a case where two variables *X* and *Y* which are in different cache lines are being written by two different processors on different stations (only two of which are shown for clarity). Both stations start off with the same shared values of *X* and *Y.* At the same point in time each station writes to the variable whose home memory location is the other station. This causes upgrades to be sent to the home stations, and in (b) the invalidates are shown already having traversed half the ring and having acknowledged the writes, so the shaded variables contain modified (new) data. If the stations read the other (non-written) variable before the invalidates arrive in (c), it is possible for STN0 to see [*oldX, newY*] and STN1 to see [*newX, oldY*] which violates sequential consistency[16].

In order to provide for sequential consistency, NUMAchine makes use of *sequencing points* on both the Local and Central Rings. (In the figure the sequencing point is shown in the middle of a ring hop for clarity, but in reality the sequencing point is either a pre-assigned station or an inter-ring interface if present. Also note that the figure shows only one ring level,

15.A blocking cache stalls on a miss, disallowing any further loads or stores until the blocking access is finished.

16.Note: we assume here that the invalidates kill the old shared copies *at the end* of their trip around the ring. If, on the other hand, we assume that the invalidates kill all local copies before being injected into the ring, the scenario shown in the figure does not violate sequential consistency. Even with local pre-invalidations it is possible to violate sequential consistency in much the same fashion, but the diagram requires four stations and so is not shown.

but both the Global and Local rings contain sequencing points.) The basic idea is that any

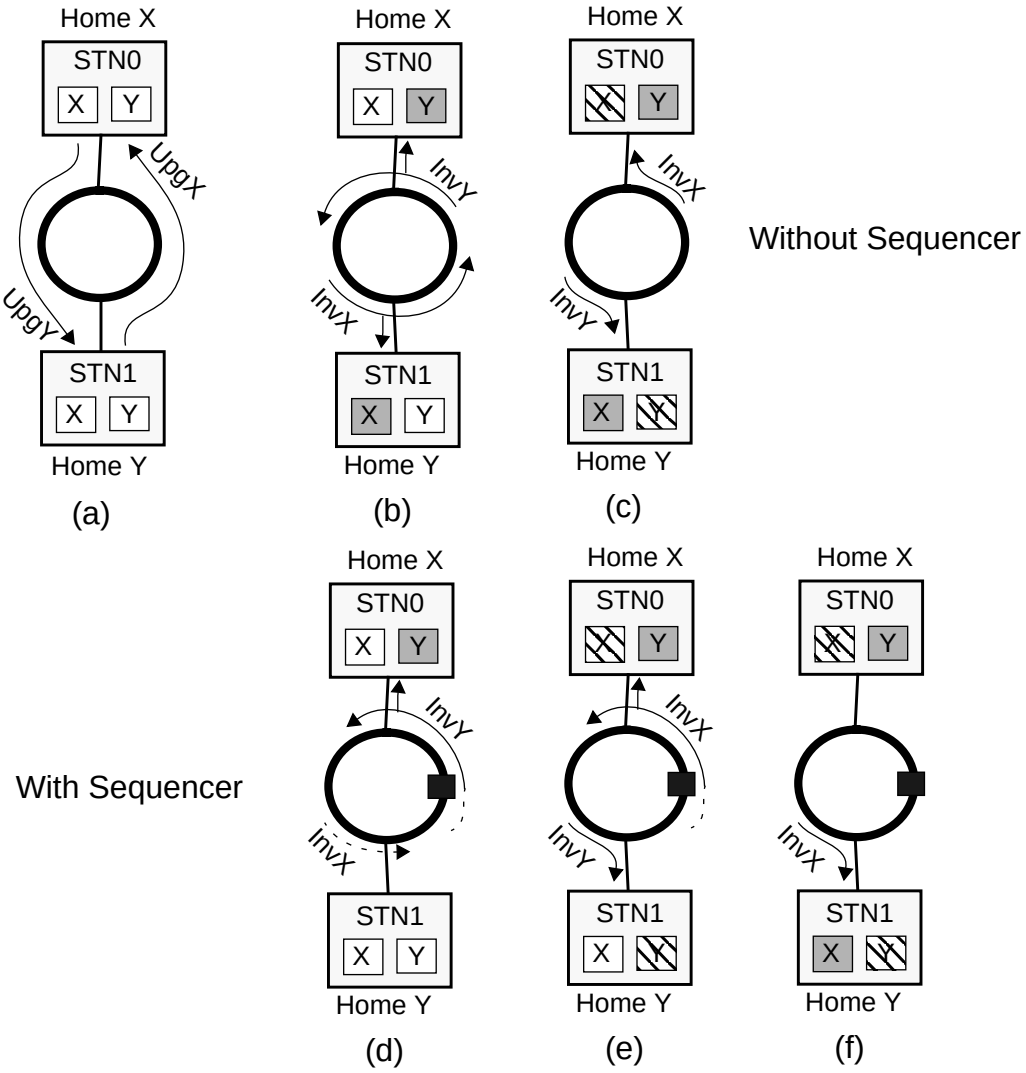

FIGURE 3.7: Sequential consistency in NUMAchine. Only two (of four) stations and one level of ring are shown for clarity. Initially, both stations have shared copies of two variables, *X* and *Y*, which reside in different cache lines. In (a), STN0 writes to *Y*, whose home location is on STN1. STN1 does the same but with *X*. In (b) and (c) shaded boxes indicate modified data, hatched indicate invalid data. In (d), (e) and (f) invalidates are inactive until they pass the sequencer.

broadcast packet is initially inactive, meaning that it is simply passed along without any destination checking. A broadcast becomes active only when it passes the sequencing point on the highest level of ring it must traverse to reach its broadcast targets. This imposes an ordering between any two broadcast packets on a given level of ring, as shown in parts (d), (e) and (f) of the figure. The sole negative impact of providing sequential consistency is to add on average half a ring traversal—two to three hops at 20 ns per hop, or about 50 ns—to the path of broadcast packets.

3.1.8 Architectural Summary

While the use of rings in a multiprocessor is not new, NUMAchine's use of the ordering properties inherent in rings to simplify the design and implementation of the cache coherence protocol, and also to provide sequential consistency, are novel. In the next two chapters we will show that these features of NUMAchine are efficient.

The simplicity of the implementation and the use of COTS parts and FPDs allows NUMAchine to reach its goal of being low cost. In addition, the marginal cost to increase the size of the system stays fairly linear. The cost of moving from an N-station to an N+1-station system is the cost of the station itself, plus one set of ring cables (which cost about $200). There is a jump in cost as the system size crosses 16 processors, since this requires the addition of the Central Ring. However, since the Central Ring is a single card with simple and regular datapaths and FIFOs, it is feasible that the entire card could be integrated into a small number of ASICs (possibly even one). This would allow the cost to scale nearly linearly right up to the maximum system size of 64 processors.

3.2 The NUMAchine Simulator

In modelling a system as complex as a multiprocessor the trade-off between model accuracy and complexity is particularly significant. Simple models cannot possibly capture the details of such a nonlinear network of interacting processes. (By nonlinear we are referring to the fact that a small change in the reference stream could potentially cause a large change in the performance. Features such as caches and congested networks make such systems impossible to model analytically.) On the other hand, the most accurate model would involve describing the system at the gate level, using a Hardware Description Language (HDL) such as VHDL or

Verilog. Not only would such a model take as much time to build as the machine itself, but such a low level of abstraction makes it extremely time-consuming to change architectural features and do forward-looking studies. The approach taken for the NUMAchine simulator is to model at a low enough level to accurately capture the salient details, but no lower. Thus, for example, all queues in the system are modelled accurately, since the average and largest depth of queued entries are good indicators of occupancy and congestion, respectively. For the memory modules, the detailed DRAM and coherence directory interactions are not modelled. Instead, we represent the coherence directory lookup time and DRAM access times as single numbers. For the directory lookup this is a fairly accurate approximation. For the DRAM, features such as refreshing, which can cause an extra delay, are ignored because they happen infrequently or do not have a large impact.

The NUMAchine simulator is *execution-driven* and based on MINT [Veenstra 1993]. Being execution-driven means that the simulator uses a real parallel executable binary as input and runs the program using an interpreter and a virtual model of the processor, in this case the MIPS R3000[17]. MINT forms the front-end of the simulator, and we provide the back-end that models NUMAchine's memory system. The two halves are linked together into a single executable called Mintsim. As a front-end, MINT is responsible for creating as many virtual R3000s as there are parallel threads. (It creates a virtual processor each time a new thread is spawned.) MINT executes the instruction stream until it encounters a load, store, or synchronization operation, at which point the virtual processor blocks (stalls) and sends a request to the NUMAchine architecture back-end (see Figure 3.8). The back-end takes the request and passes it through caches, busses, rings, etc., generating appropriate delays at each step. Eventually a response is scheduled to go back to the appropriate processor, at which point the processor unblocks and continues executing as before until the next load, store or synchronization event occurs. In this way the stream of references maintains correct temporal ordering, due to the feedback path between the back- and front-ends. This temporal ordering is particularly important for modelling the caches and cache coherence. This technique yields more accurate results than *trace-driven* simulation, which uses a pre-generated static listing of event/time

17. The R3000 does not support the MIPS IV Instruction Set Architecture (ISA), in particular it only supports 32-bit words, not 64-bit. This is definitely a drawback considering that all microprocessors are moving towards 64-bit operation, but it would have required too much effort to modify MINT.

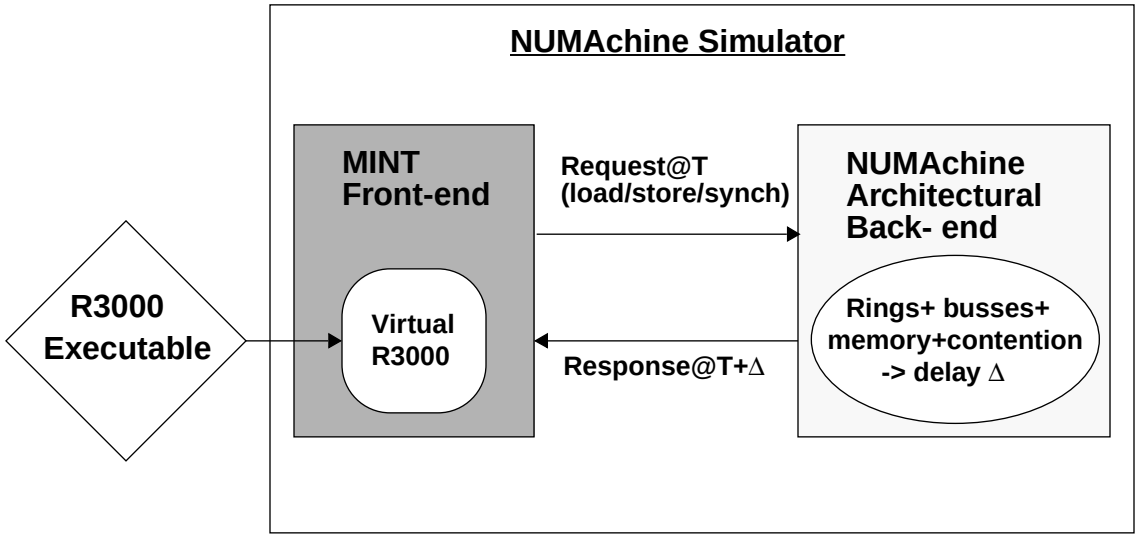

FIGURE 3.8: The NUMAchine simulator structure. The diagram shows a single thread running on one virtual R3000 processor. WIth *N* threads, there are *N* independent virtual R3000s, each communicating with the back-end in parallel.

pairs. (In an execution-driven simulation, the delays are generated by the component models, which is more accurate and flexible than using static timings.) Another benefit of execution-driven simulations is that large trace files (typically many hundreds of megabytes in size) need not be either generated or stored.

When MINT encounters an OS call, it either runs the call natively on the host machine (for file operations) or mimics the behaviour of the function internally. In either case the call takes zero time as seen from the virtual processor. While modelling operating system activity would provide a more accurate picture of real-world performance, it is not possible to do given our use of MINT. Even if it were possible, there are good reasons to stick with the simpler model. Operating system code has very different behaviour patterns than regular programs. Modelling the two together, while more realistic, makes analysis more difficult since the two reference patterns are intertwined, and may even affect each other in ways that are difficult to predict, particularly when caching is included. A much more intricate toolset is required for this type of simulation. Newer simulation environments such as SimOS [Rosen-

blum 1997] do allow this level of complexity, but are not designed to support user-written back-ends, thus they do not allow the modelling of specific architectures.

Another important factor in system-level modelling is page management. We chose not to model page fault overhead in Mintsim. As with OS activity, modelling this behaviour would be more realistic but would also make the analysis much more difficult. Mintsim *does* have the ability to model round-robin, first-hit and fixed page mapping schemes, but in all cases the initial fault takes zero time. We also chose not to add features such as page migration and replication, again for complexity reasons. (We chose to use the first-hit policy. Chapters 4 and 5 will shed more light on this choice.)

Mintsim also has the ability to accurately model the full NUMAchine memory hierarchy behaviour for either the whole program, or for just the parallel section. We define the start of the parallel section by inserting a special dummy routine into the Splash2 source code just before thread creation. The end of the parallel section uses a similar dummy routine call after the main thread has successfully waited for all children to finish[18]. Mintsim's default behaviour is to model only the parallel section.

When skipping over the sequential code, Mintsim correctly executes all instructions with the correct opcode timings, but allows all loads and stores to succeed immediately, without checking the cache and, even more importantly, without doing any page mapping. This has two effects. The first is to underestimate the time spent in the sequential section. The second is to leave the cache in a cold state when the parallel section starts. Underestimating the sequential time makes the performance (as measured by speedup curves) look better, because the same amount is subtracted from both numerator and denominator in the speedup equation. Cold cache effects in the parallel section increase its execution time, which tends to reduce measured speedup, countering the first effect. The net effect is negligible, as will be shown in the next chapter.

We used the simulator as both a validation and design tool. On the validation side, we took great pains to ensure that the cache coherence and ring network models were accurate. We then used the simulator to check for correct operation of the coherence scheme and rings. In one case the simulator found a very rare corner case that exposed a bug in the coherence

18.The dummy routine, `generate_event()`, is a null routine in the source code, but is recognized by MINT and translated into a call to the `sim_user()` routine in the back-end, which takes the appropriate action.

protocol. As an aid to validation we found it useful to include *coverage tables* for the protocol. These tables keep track of how many times each state transition in the protocol is activated. Some transitions never occurred in even the longest and most complicated simulation runs, necessitating either synthetic programs designed specifically to activate the transition, or as a last resort careful verification by hand. The coverage tool also proved useful in providing feedback on the frequency of state transition usage, allowing for possible future optimizations for the most commonly taken transitions.

It should be noted that while the simulator proved extremely useful as a validation tool, it did *not* provide formal verification. Formal verification of the cache coherence protocol, for example, can be achieved by formulating the protocol as a (large) theorem in temporal logic which can be shown by the rules of logic to be true or false. While work in the area of formal verification has been under way for some time (e.g. see [Hailpern 1980]), tools such as Murϕ [Park 1996] lacked the power and to handle problems of this complexity.

As a design tool, the simulator allowed us to answer many ‘what-if’ questions quickly. For example, we originally designed the station bus to be 128 bits wide instead of 64 bits. As we started designing the control logic, we discovered that implementing a 64-to-128 multiplexer in the datapath of the processor card was more complicated and costly than we had anticipated. A week of simulation indicated the performance improvement was only on the order of 10%, and so was not worth the effort.

3.2.1 Simulator Implementation

The simulator back-end consists of about 20,000 lines of C++ code, representing over one man-year of programming. The use of object-oriented design proved to have both benefits and drawbacks. Given the size and complexity of the simulator, abstracting away details into classes helped keep the code manageable. Class inheritance was useful in forcing us to first codify the common features of an entity such as a cache into a base class, then later differentiating into specific instantiations such as direct-mapped or set-associative.

On the down side, linking the back-end into MINT proved problematic because the latter is written in C. The original version of the simulator had both the front- and back-ends using separate event-scheduling engines. This turned out to be quite slow, and it was decided to rework the back-end to use MINT’s highly tuned scheduler. This required some advanced C++ techniques, which will be described shortly.

The basic architecture of the back-end is a set of independent concurrent processes communicating by passing *messages* through *ports*. The messages are of fixed format, and contain all information pertaining to a single event in the back-end. Examples of events are load or store references from a processor, a request for bus access, or a packet on the ring. Ports are meant to be generic connection points for simulator entities. For example a Processor object has a Memory port, which could be connected to a Cache, Bus, or Memory object, depending on the architecture. Ports are bidirectional, and contain two methods, `Send(Message*)` and `Receive(Message*)`. Once a connection is made between two ports, calling a port's `Send()` with a message will activate the `Receive()` method at the other end of the connection. (Note that a given simulator object can have multiple ports. For example a processor could have ports to memory, a cache and a monitoring object.) The `Send()` method sends a message in zero time, which is useful for maintaining status information, but not for general modelling. Another method, `SendAtTime(Time t, Message*)` sends the message at a time `t` units in the future.

The basic operation of the simulator is thus to invoke `Receive()` methods at scheduled times. These `Receive()'s` process their messages and by means of `Send()'s` cause other `Receive()'s` to be scheduled at later times. Ultimately a processor's `Receive()` (from the memory port) will be activated, which will cause a call to MINT which causes the appropriate thread to unblock.

As mentioned above, using the MINT scheduler causes some difficulties when interfacing to the C++ back-end. MINT is designed to schedule its own internal events, which basically contain pointers to function calls in the MINT code. To use this framework for the C++ methods, some means of encapsulating a method call to an object is required. One way of doing this is to use an object called a *functor* [Coplien 1993]. This is an object which behaves as a function. By suitably encapsulating these functors, it is possible to have the MINT scheduler call the appropriate object's method at the correct time.

We found the performance of the simulator to be quite good. For example a typical application[19] run on an SGI Challenge machine with 150 MHz R4400 processors took 17 seconds to execute natively, and 1570 seconds in the simulator (running on a Sun Ultra 4 with 296

19.The application was the Splash2 kernel Cholesky, using the tk18.O input. The Splash2 benchmark suite will be described in the next chapter.

MHz UltraSparc-II processors), giving a slowdown on the order of 1000 times. This allows programs with running times of a few tens of minutes to be simulated in under a day, which allows for rapid feedback and experimentation.

Additional information on the simulator can be found in Appendix A.

3.2.2 Simulator Correctness

The most important consideration in using a simulator to model a complex system such as NUMAchine is the level of belief in the simulator's modelling accuracy, and whether the simulator is itself functionally correct.

Our approach to validating the simulator consisted of two stages. For initial validation, we used some small hand-designed synthetic benchmarks, for which we could predict the results. One example of such a benchmark is Single-Reader (SR). In SR, each processor allocates an array, then simply reads through it. (Note that compiler optimizations typically have to be turned off for these benchmarks, to keep the optimizers from throwing away all code; the results of reads in SR are never used.) The number of iterations is specified as an input parameter. The array size can be chosen to fit into or overflow any given level of cache. In either case the number of cache hits and misses, and the overall latency can be calculated and compared against simulator output. Another useful feature of SR is a parameterizable *array stride*. On each iteration the processor-to-array correspondence can be changed by using some fixed stride length to walk through the different arrays. An array stride of zero is the default, and a stride of four causes processors to use arrays from different stations on each iteration, thus testing the Network Cache. Other synthetic benchmarks included Single-Reader/Single-Writer (SRSW), used to verify write coherence actions, and Multiple-Reader/Single-Writer, which helped test out broadcast invalidates.

The second stage of validation occurs after the hardware prototype has been built, and involves redoing measurements on the hardware and comparing the results against the simulator. Results of this validation will be presented in Chapter 4.

3.3 Conclusion

This chapter presented the architecture and implementation of the NUMAchine multiprocessor and NUMAchine's architectural simulator, Mintsim. We showed how NUMAchine's use

of a two-level ring hierarchy allows for a system that scales well up to 64 processors both in terms of latency, and cost. We considered ordering properties of rings which enabled simple and efficient implementations of a novel hardware cache coherence scheme and the provision of a sequentially consistent programming model.

NUMAchine makes use of cache in the Network Interface Card, with the goal of reducing the latency penalty for remote versus local accesses. This Network Cache (NC) helps to reduce the level of NUMAness of the machine. The coherence directory scheme, of which the NC is an integral part, uses a novel, lazy approach to the maintenance of coherence directory information. This lazy approach does not bother trying to maintain inclusion between cache levels.

We outlined the general procedure used for implementation of the prototype, and stressed the importance of system-level simulations using high-powered CAD tools to verify design functionality.

The design and implementation of the NUMAchine simulator was discussed, as was its use during the prototype design to choose appropriate system parameters and validate system functionality. Mintsim's flexibility also allows it to be used as a research tool, which will be its role in the next two chapters. These chapters will analyse the overall performance of the prototype, as well as providing justification for the design choices, such as the lazy directory protocol, sequential consistency, ring hierarchy and backoff mechanism.

CHAPTER 4

Prototype Performance & Analysis

This chapter investigates the performance of the NUMAchine architecture using Mintsim. We first consider the overall performance, then look in more detail at the Network Cache, rings, backoff mechanism and coherence protocol.

4.1 Simulation Environment

To analyse the performance of NUMAchine we use Mintsim and a subset of the programs from the Splash2 benchmark suite [Woo 1995]. The Raytrace application from the Splash2 suite had a problem linking with libraries, so we could not get it to run. Volrend, Radiosity and FMM all had execution times greater than half a day for a single datapoint, so we decided not to use them. The other programs had execution times ranging from about 5 minutes for FFT, up to over 2 hours for Barnes.

Throughout this chapter, Splash2 programs all use the parameters specified in the Splash2 characterization paper as the defaults for up-to-64 processor configurations. For completeness, the applications used and their parameters are shown in Table 4.1.

The parameters for the simulated hardware are the same as the NUMAchine prototype described in Chapter 3. The rest of this section discusses details of the model of the prototype used in Mintsim.

4.1.1 Station Bus

The station bus is modelled using the hardware's default round-robin scheduling scheme. If the bus is idle, a request succeeds immediately. Transaction duration on the bus includes any cycles required for data, as well as one idle cycle at the end of a bus transaction, which is nec-

TABLE 4.1: Splash2 program parameters for the prototype analysis.

Splash2 Program	Parameters/Description
FFT	*-m16 -l7 -n512*: 64K complex doubles, 128-byte cache line size, 512 cache lines (64KB cache)
Cholesky	*tk18.O*: medium-sized sparse matrix
Barnes	*16K particles*
LU	*-n512*: 512x512 matrix, 16x16 blocks
Ocean	*-n256: 258x258 ocean grid*
Radix	*-r1024 -n1048576 -m2097152*: 1M keys, 2M maxkey, radix 1K
Water	*512 molecules*

essary for turnaround of the bus drivers. If any transaction targets are busy (due to the flow control scheme outlined in Chapter 3), then the transaction is skipped for the current round of arbitration. (Note that for multicasts there can be multiple bus targets. If any one of them is busy, the whole multicast must wait.)

4.1.2 Queue Modelling

All queues in the simulator model depths correctly; that is, a cache line written into a queue uses up the full 17 entries (16 data doublewords[1] + 1 command)[2], in order to accurately gauge average and maximal queue usages. During the design stage, this allowed us to determine optimal queue sizes. For analysis purposes, these numbers provide a measure of burstiness and contention; queues that handle large bursts have maximum queue usage values that differ significantly from the average.

In most cases, the queues are modelled with zero pass-through latency. That is, a write into the queue is immediately available at the output. Typically the queue is one element of a chain in a datapath, and the queue latency is simply lumped in with other overheads to speed up the simulation[3]. One case where queue latency is modelled directly is for the FIFOs that

1. We use the MIPS definition of a ‘word’ as containing 32 bits, and a ‘doubleword’ 64 bits.
2. Logically the simulator treats a cache line as a single message (not 17 separate messages) for simulation efficiency.

inject packets onto the rings (both Local and Central). We use a 30 ns delay in this case, which is a typical number for an IDT72205-15 SyncFIFO running on a 20 ns clock [Integrated 1994].

4.1.3 Memory Card

The memory card uses the lump-sum model since it consists of a single datapath. The default is to have an 80 ns delay for the coherence directory lookup, which takes into account the time to look up the state information, as well as the time to generate a single (non-data) command packet. If a DRAM access is necessary, then a further 200 ns are added to model a cache line access. This is the time for the DRAM controller to get the first doubleword of data into the output queue and ready to go onto the bus. Note that since the DRAM operation is pipelined, the time is *not* 320 ns (16 doublewords x 20 ns clock cycle). This model is somewhat oversimplified, because in reality the directory lookup and DRAM accesses are partially overlapped. This is an oversight which could be fixed in future simulation studies. Since the net effect is to underestimate the performance, we decided to leave this as is. We will ignore the line size issue throughout the remainder of this thesis, and will stick with a fixed 128-byte line.

4.1.4 Processor Card

Accurate modelling of the processor card is crucial for good simulation results, because this is where the critical L2 cache resource resides. With the MIPS R4400, the L2 cache cannot be simultaneously accessed for both internal processor activity and external coherence requests such as interventions. We model this by locking out the L2 cache on a first-come/first-served basis. All cache access latencies are modelled using numbers from [Heinrich 1994][4]. The L1 instruction and data caches have zero latency, while L2 accesses require either 3 or 4 cycles for a read, and up to 32 cycles for an L2 cache line refill.

3. Whenever a sequence of dependent events happen with deterministic timing, they are lumped together in this fashion. The speed of the simulator is directly related to the number of events that need to be scheduled.
4. The intervention response latency given in the manual is 8-28 cycles, with the variability coming from the non-deterministic time to gain access to the L2 cache. Since we model this feature independently, the number we use is the minimum, 8 cycles.

4.1.5 Paging Policy

Mintsim supports three page placement policies: round-robin, first-hit, and a fixed scheme where a file containing page mappings is provided as input to the simulator. The fixed scheme requires a preprocessing pass on the program to determine page usage statistics, and is intended for future work. Of the two others, round-robin is the most common for this type of study. A problem with the round-robin policy is that a page which is used by only one processor can be placed on a remote node. This not only increases average latency, but also needlessly increases capacity pressure on the NC cache lines. A first-hit policy (also called first-touch) does not suffer from this problem, since private pages will always be located in local memory[5]. When more than one processor shares a page, first-hit will place the page so that it is local to at least one of the processors. One of the major drawbacks to using first-hit is that sequential startup code (e.g. initialization of all shared data structures) can touch pages that ultimately will be used exclusively or mostly by other processors. In the worst case, all pages could be located in the master thread's local memory.

An OS that provides page replication and migration can achieve the best of both approaches. An initial round-robin placement can use page-sharing statistics along with page migration and replication to evolve over time to an allocation that is close to what would have resulted from first-hit. Although it is possible to do the same thing in the simulator, it would be quite complicated. Because we are not modelling page fault overhead, it would be difficult to justify modelling migration and replication overhead, although they clearly have a significant effect on performance. For simulation simplicity, we instead use a first-hit policy for memory references during the parallel section of the program. (As mentioned in section 3.2, generating references only for the parallel section is Mintsim's default mode of operation.)

4.1.6 Instruction Fetches and Sequential Code

Mintsim has the capability to model instruction fetches. With this feature turned on, the back-end instantiates an L1 instruction cache, and also an L2 instruction cache if necessary. (The

5. If a program has large phases, where the groups of processors sharing a page change between phases, then first-hit will not mimic an effective migration and copying scheme. The page allocation for the Splash2 programs used are static, and do not suffer from this problem.

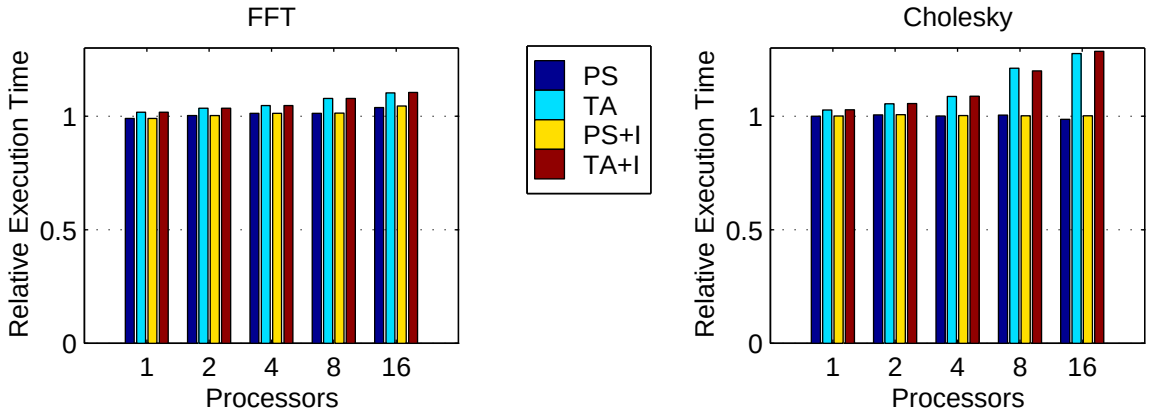

FIGURE 4.1: Modelling of sequential code and instruction fetches. The first two bars show the increase in execution time for the Parallel Section (PS) and the Total Application (TA) when the sequential startup code is modelled accurately. The second two bars show the increase when instruction fetching is turned on (in both the parallel and sequential sections). All execution times are normalized to the respective times for a simulation run without sequential code or instruction fetching. All simulations (including the normalization runs) use round-robin page placement, since first-hit would tend to allocate all pages on the home memory of the processor responsible for the sequential initialization code.Numbers for 32 and 64 processors are not shown because the round-robin scheme caused programs to take bus errors due to NACK time-outs, due to some heavily contended pages being placed remotely from all sharing processors.

default is to use a unified L2 cache, in which case instructions compete with data for L2 cache space.) Simulation time is roughly doubled by turning on instruction fetches. Instruction streams for these types of scientific applications are highly regular and have small footprints, and even a small L1 instruction cache is sufficient to achieve very high hit rates. With 1 MB of L2 cache in our prototype, conflict problems between instructions and data are insignificant. We expect modelling of instruction fetches to have little impact, and present results below indicating that this is the case.

In Figure 4.1 we show the result of running the programs FFT and Cholesky in the default (parallel-only) mode, full-application mode, and full-application plus instruction fetches. Cholesky has the highest ratio of sequential-to-parallel code in our group of applications. All simulation runs used a round-robin page placement, because as mentioned above using first-hit when modelling sequential initialization code can lead to all pages being allocated in one memory. There are two important conclusions to be drawn from the figure. The first is that modelling instruction fetches has almost no overall effect on either full-application or parallel-

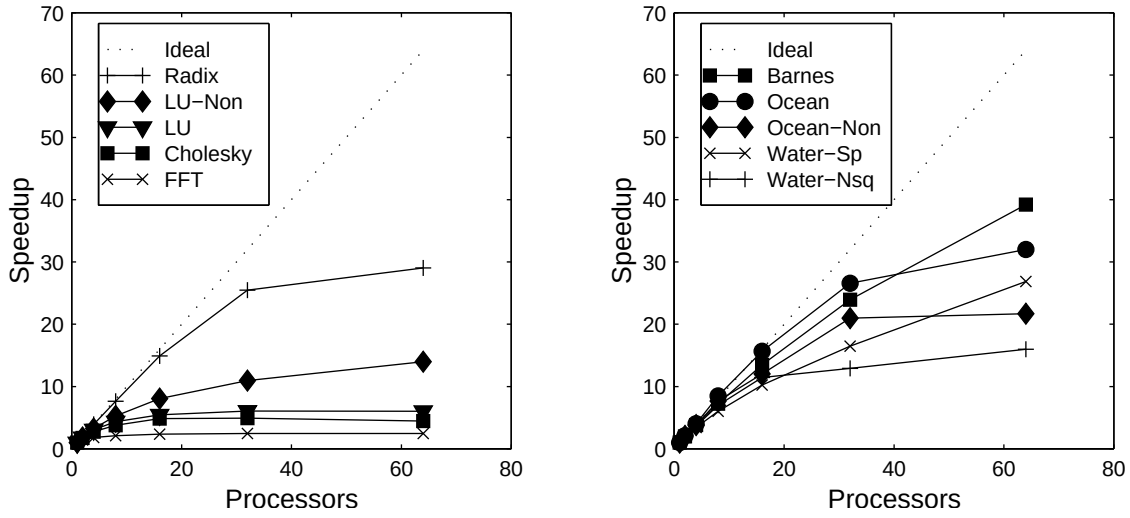

FIGURE 4.2: Simulated prototype speedups for the Splash2 programs. The graph on the left is for the Splash2 kernels, that on the right for the applications.

section execution times, as can be seen by comparing each '+I' bar against its non-'+I' counterpart. This result matches our prediction from the preceding paragraph. The second conclusion is that while accurate modelling of the sequential code definitely has an impact on the application's total execution time (and thus the speedup), it has practically no effect on the execution time of the parallel section. Thus, cache warm-up and page-mapping effects of the sequential startup code have little impact on the performance of the parallel code section, which justifies our choice for Mintsim's default mode of operation. The key point is that we are not so much interested in the performance of the sequential code as in the efficiency of the parallel section.

4.2 Prototype Analysis

To begin our examination of the prototype performance, we show in Figure 4.2 the speedup curves for the Splash2 programs. (Note that all of these curves were generated using the default fast mode for the sequential section, meaning that the curves overestimate the performance.)

We define NUMAchine's performance to be good if increasing the number of processors always results in some gain (i.e. there is never a slowdown compared to some smaller number of processors), and the maximal speedup at 64 processors is roughly over 20. (These criteria may not be acceptable for very high performance machines, but we consider them appropriate for our implementation, which is targeted towards low cost.) These simple speedup curves indicate that the performance of the NUMAchine prototype is good for most of the Splash2 applications except Water-Nsquared and Ocean-Noncontiguous. This is not problematic since both of these are older versions of the programs, and the newer versions do meet our criteria. For the Splash2 kernels the only program that exhibits good performance is Radix. What we are really interested in measuring in this study, though, is not whether the Splash2 programs parallelize well (which some of them such as FFT do not, particularly for the small default problem sizes), but whether our architecture can support efficient parallelism.

If we plot the same graphs but ignore the sequential execution time and focus on the ratio of execution times of the parallel sections of code, which we call the *parallel speedup,* then we get a better picture of NUMAchine's parallel efficiency. In Figure 4.3 we see a large improvement in the poorly performing kernels, with only LU and Cholesky still exhibiting poor performance. (The figure also shows the algorithmic speedups from the Splash2 paper for comparison. Note that Cholesky has a poor algorithmic speedup, indicating that it will never be able to run well in parallel.) The main reason for the difference in true versus parallel speedups lies in the choice of problem sizes. Programs that show a large difference between the two have small ratios of parallel to sequential code, and thus cannot achieve good overall speedups due to Amdahl's Law. While choosing larger problem sizes would alleviate this problem, it would also lead to longer simulation times. More importantly, it makes the results difficult to compare against other studies. In this and in the next chapter we will use parallel speedup as our metric for the efficiency of the architecture.

In the rest of this chapter we will explore various aspects of the design and see how they impact on performance. In the next chapter, we will investigate ways of modifying the architecture and tuning system parameters to achieve better performance.

4.2.1 Comparison of the Simulator and the Prototype

As a check on the simulator, we ported the Splash2 programs to NUMAchine in order to compare results from the real hardware against those from the simulated hardware. The port was

only partially completed as of writing, so the only applications working well enough to generate results were Cholesky and Barnes.

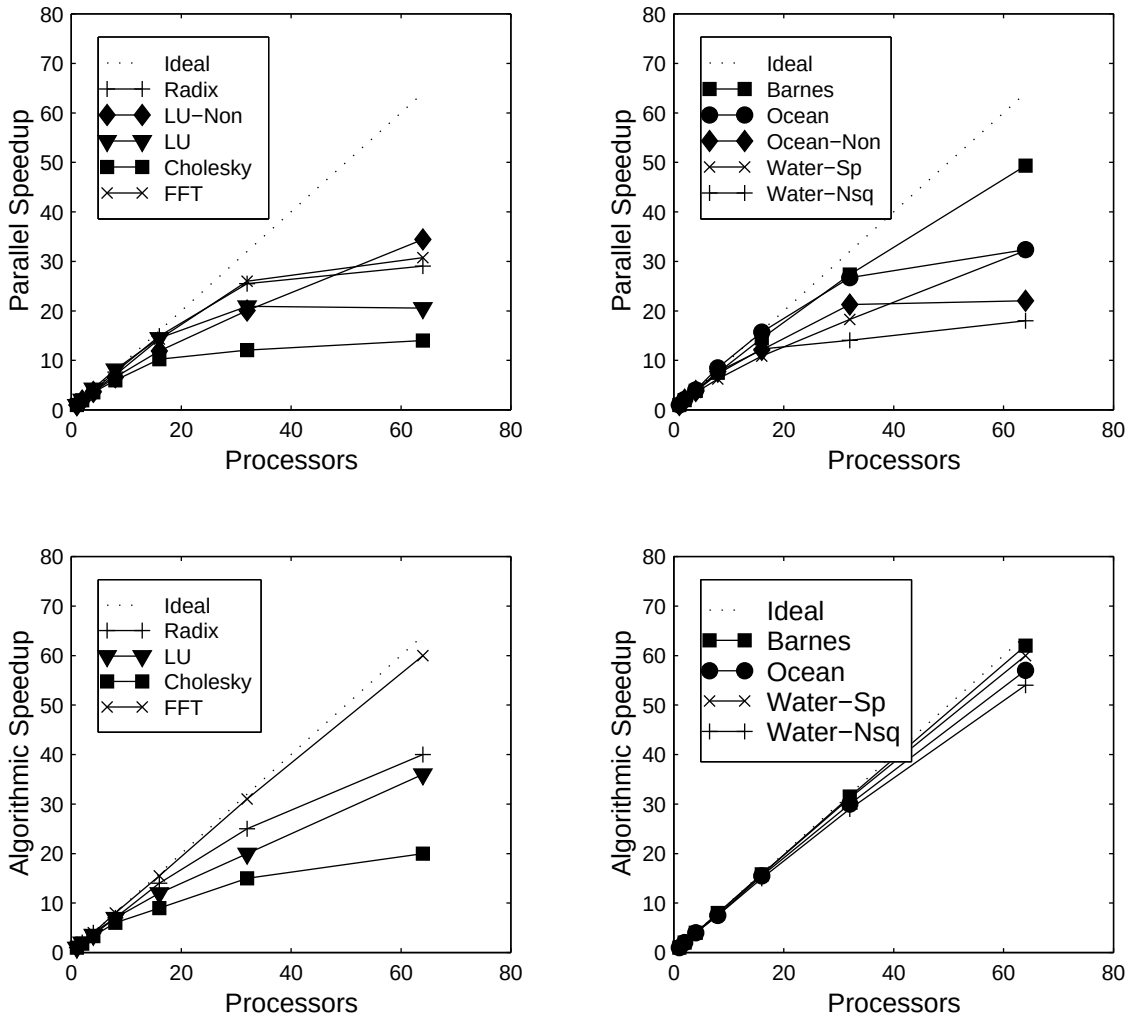

FIGURE 4.3: Parallel versus algorithmic speedups. The significant improvement for certain programs indicates that the problem size is too small in these cases. There is not enough work in the parallel section to allow for large speedups. For comparison purposes, the bottom two graphs show the algorithmic speedups from Figure 1 in [Woo 1995].

The first comparison is between uniprocessor execution times in the simulator and the prototype. This gives an idea of whether the overall (i.e. absolute) timescale of the simulator is accurate. The hardware consisted of a single Local Ring with 16 processors, running at 40 MHz. We ran the two programs under Mintsim at the reduced speed using a single processor, with sequential code included. We expect the simulator numbers to be lower, given that the simulator does not model OS activity or page faults. The results are shown in Table 4.2. The numbers are low by a factor of two to three, which is quite good. (Comparisons of simulation versus real hardware measurements are not typically reported in the literature. Note that the values of the numbers presented in the table are not too significant, only the fact that the ratios between them are within an order of magnitude.).

TABLE 4.2: Uniprocessor simulated versus hardware execution times.

Program	Simulated ExecutionTime (seconds)	Hardware Execution Time (seconds)
Barnes	*72.5*	*216*
Cholesky	*9.9*	*30*

Next we compare the speedups (i.e. relative timing) in hardware versus those presented in Figure 4.2. For the hardware and simulator we measure speedups relative to their respective uniprocessor times. The result is shown in Figure 4.4. For a program such as Barnes, which has a fairly long runtime, and can thus amortize the overhead due to paging (which is not modelled), the agreement is very good. Cholesky's short runtime cannot amortize this overhead, and the agreement is not as good, although there is a correlation between the simulated performance and that of the hardware.

4.2.2 Fmask Performance

As the number of stations in the system increases, the probability of the Fmask overspecifying stations increases. We can measure this imprecision in the Fmask by keeping track of the exact number of sharers in the memory coherence directory. When an invalidation is sent out, we divide the actual number of stations targeted by the real number of sharers to end up with the *overinvalidation rate*. For the simple case of two sharers, the overinvalidation rate can be

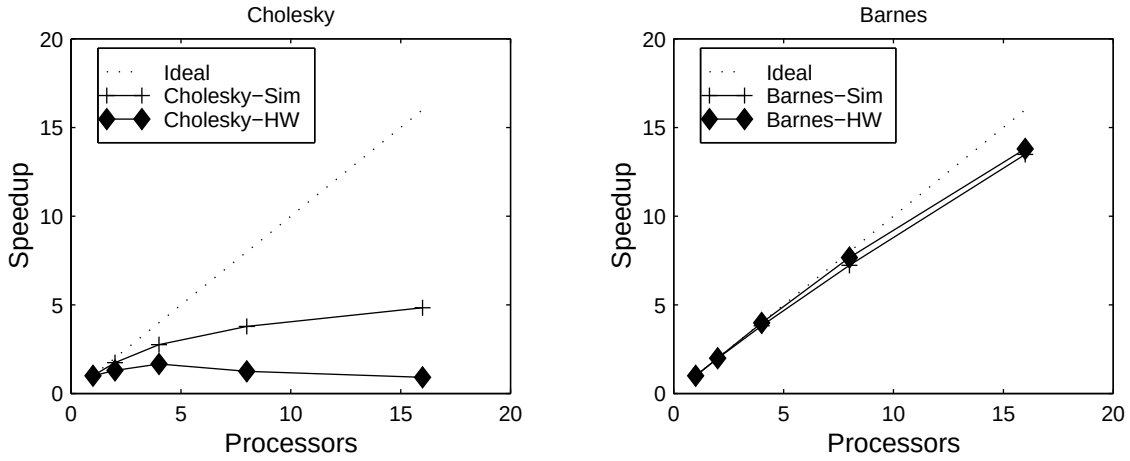

FIGURE 4.4: Simulator versus hardware prototype speedups. Speedups in both cases are total speedups, not parallel speedups. Cholesky has a short run time, thus in the real hardware it cannot amortize the error due to paging overhead, which accounts for the discrepancy in the results.

as high as two if the sharers are on different rings and stations. (Note that if the sharers are on the same station *or* the same Local Ring then the overinvalidation rate is one, i.e. there is no overinvalidation and the Fmask is precise.) We expect the rate to be around two if the number of sharers on average is two, because we did not tune the Splash2 programs to take advantage of locality. Figure 4.5 shows the overinvalidation rate averaged over all invalidations. The rates reach roughly 2.5 for some of the programs at 64 processors, indicating that there are sharing patterns involving three or more processors. However, invalidations incur little overhead in the processors, so the important point is that multicast invalidations do not on average become broadcast invalidations to all stations. Avoiding heavy broadcast traffic is important in maintaining system scalability, as shown in [Farkas 1992].

4.2.3 Ring Performance

The first aspect of the rings we will look at is average utilization. In a single ring clock cycle a ring slot can be used for one of three reasons:

- Send Packet - Inject a packet into an empty slot.

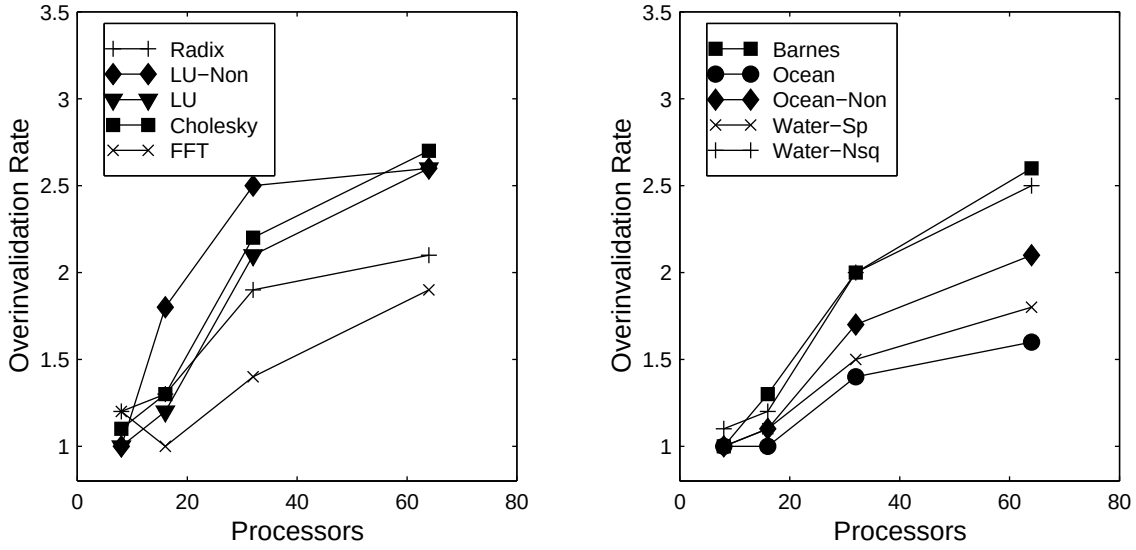

FIGURE 4.5: Overinvalidation rates. This is the ratio of stations that are actually targeted by an invalidate to the number that need to receive the invalidate. Rates greater than one indicate imprecision in the Fmask. Note that there can only be imprecision if there is more than one station, so the number of processors starts at 8.

- Forward Packet - Pass along an upstream packet that has only downstream targets. This includes broadcast packets that do not select the ring interface.
- Split Packet - Write a copy of the packet into the receive queue, and also forward it to the next ring interface. This is for broadcasts that *do* target the receiving ring interface.

Note that we do *not* consider the receipt of a terminal packet[6] to have used the slot. This is because the slot becomes free on the current clock tick; whether we choose to use the just-freed slot or not is independent of the slot's availability for carrying new traffic. We can now define the ring utilization (from the point of view of a single ring interface as):

$$RingUtilization = \frac{PackSent + PackForwarded + PackSplit}{TotalRingSlots} \quad \textbf{(EQ 4.1)}$$

6. A terminal packet is one which is consumed by the ring interface, creating an empty slot. This could be a point-to-point packet, or the final receiver of a multicast.

Note that the total number of ring slots consists of the number of slots in a given amount of execution time.We use the parallel execution time to calculate this number. Using the total application time would unfairly deflate the utilization numbers, since by default we do not model any ring traffic outside of the parallel section. (In addition, we do not expect as much remote traffic during the sequential section, and the nature of the traffic patterns would be different in any case. By keeping statistics only for the parallel section we avoid adding this noise to our measurements.) To arrive at the overall average utilization, we average over all the ring interfaces. For the Local Rings, this means the ring interfaces in the NICs, plus the Local Ring side of the inter-ring interfaces (IRIs). For the Central Ring utilization we just average over the Central Ring portions of the IRIs.

We show the results for the Central Ring in Figure 4.6, and the Local Ring in

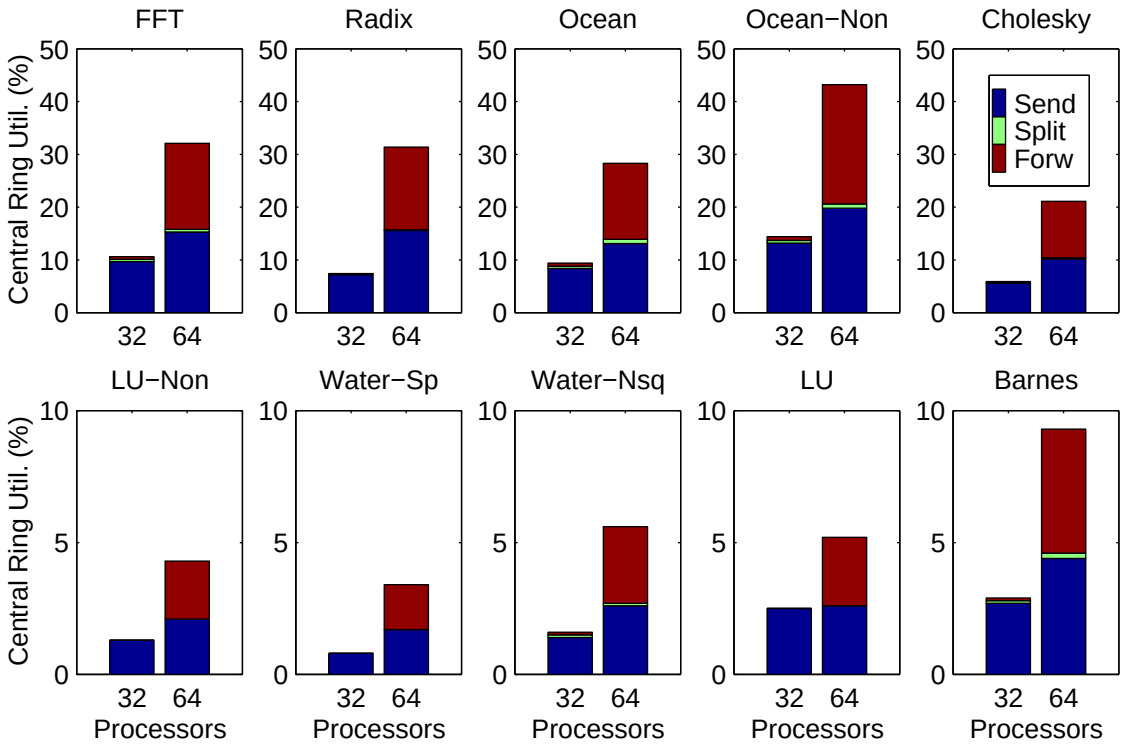

FIGURE 4.6: Central Ring utilizations. The Central Ring only exists in 32- and 64-processor configurations. Note that the top and bottom rows have different vertical scales.

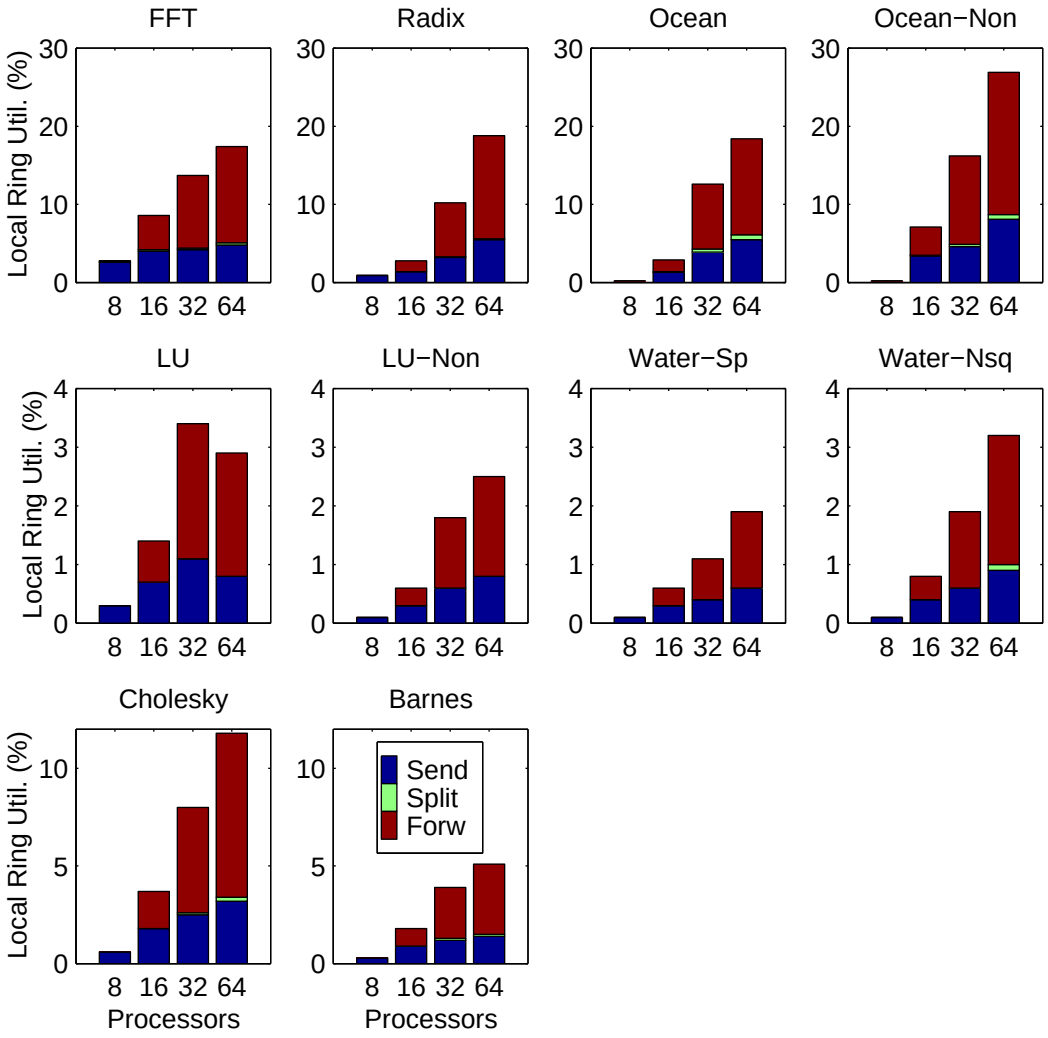

FIGURE 4.7: Local Ring utilizations. No rings exist for configurations of four or fewer processors. Also note that each row has a different vertical scale.

Figure 4.7.The large utilizations for Ocean, FFT and Radix arise from the high communication-to-computation ratio for these programs, as described in the Splash2 paper [Woo 1995]. As expected, the Central Ring utilization is higher than that of the Local Ring, except for the case of 32 processors[7]. This high average utilization indicates that the Central Ring becomes

7. For 32 processors the Central Ring consists of only two hops, which makes it almost equivalent to a full-duplex point-to-point connection. With the exception of invalidations, all traffic produced by one node is consumed by the other; there is no bypass traffic.

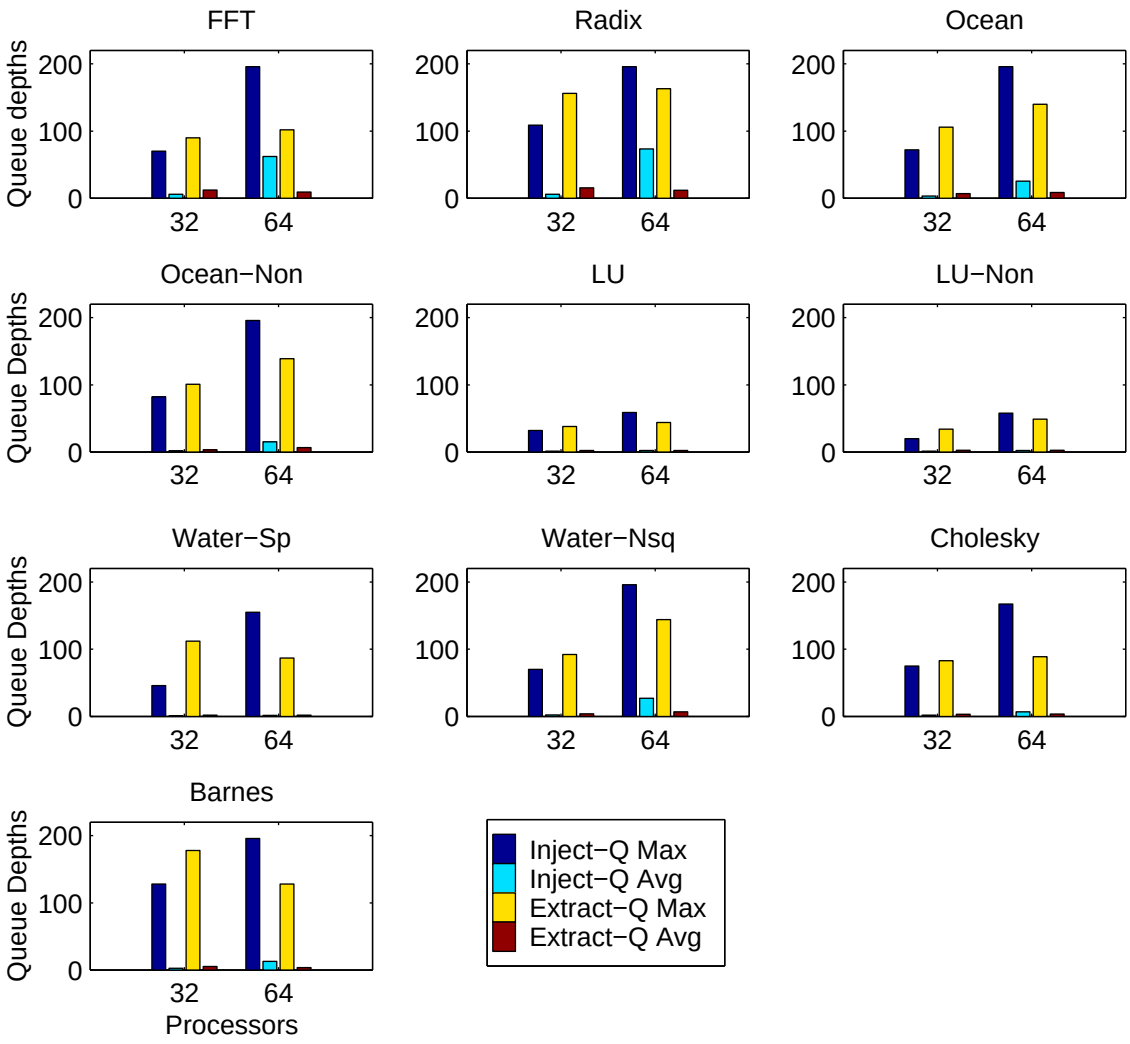

FIGURE 4.8: Central Ring queue utilizations. The numbers shown are the maximum and average depths for the queues that inject into and extract from the Central Ring.

congested, which is the reason for designing the Central Ring to allow for higher clock speeds. Increasing the Central Ring speed will be considered in Chapter 5.

As a measure of congestion, we consider the maximum and average depths of the queues in the network interfaces. A large difference between the maximum and average values indicates bursty traffic and long ring access latencies. Results are shown in Figures 4.8 and 4.9, showing clear evidence of heavy congestion. Programs such as Barnes show large maximum

queue depths, but have low overall utilizations, indicating that the bursts are few and short-lived. FFT, on the other hand, has large average utilization but low maximal queue depths,

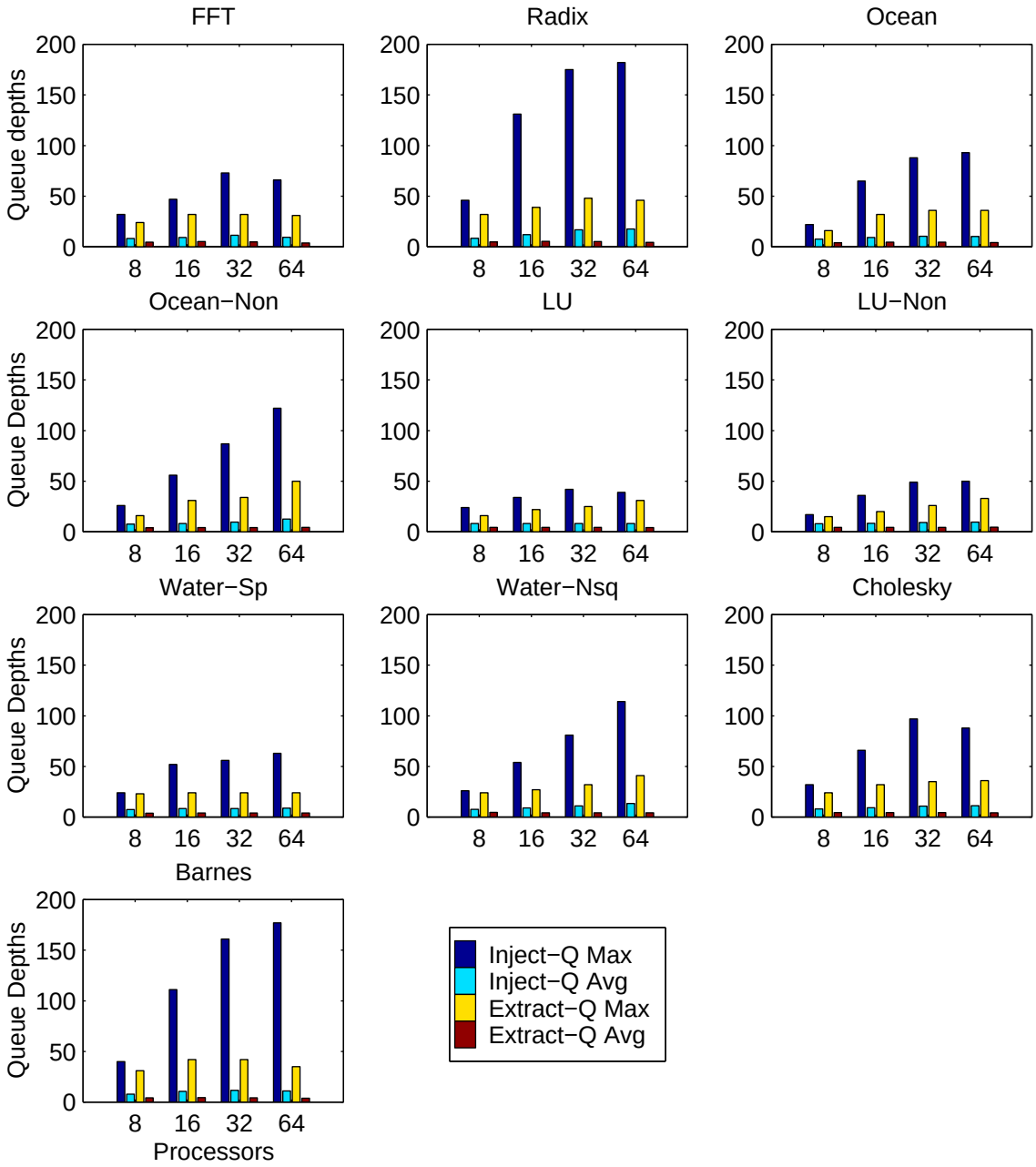

FIGURE 4.9: Local Ring queue utilizations.

leading to the conclusion that its traffic is more evenly distributed in time.

In general, congestion is worse in the Central Ring, except for the two LU programs, which show about equal—and comparatively low— levels of congestion in the Local and Central Rings. Maximum depths in the Central Ring also seem to be fairly symmetric between the injecting and extracting queues. This is not true for the Local Rings, where the most congested programs such as Barnes show a marked asymmetry, with the injection side being worse. This effect has to do with the NC. To see the reason behind this, consider that a queue only fills up if average input and output rates differ. The ring-injection queue on the NIC card has only one output, onto the ring. On the input side, it is fed both from the bus and the NC. Thus we expect one or both of these sources to show unusually high levels of activity. Indeed, if we look in the simulation output files, it turns out that Barnes has one of the highest NC hit rates (53% for 64 processors), with hits generating data (i.e. cache line) responses. Since a good fraction of responses are expected to involve forwarding, the NC has the potential to significantly increase pressure on the ring-injection queue, although this does save bus bandwidth. One possible method for increasing the ring-injection rate is to make use of the just-freed slot, which we consider next.

The measurement involves turning on the switch to use the just-freed slot and re-running the simulations. We show the results for FFT in Figure 4.10. The graphs show the relative improvement by making use of the just-freed slot. For the utilization curves we have split out the separate components of the utilization, so the total increase in utilization is the sum of the three. We see improvements up to about 30%, mostly for the 64-processor configuration. For the queue depths, the ratios are inverted, so that a ratio greater than one represents a *decrease* in the depth. For the Local Ring, we see the most improvement in the maximum depth of the ring-injection queue, which is to be expected. The improvement on the Central Ring is more drastic, and also much more variable. The ring-injection queue's maximum is improved by a factor of 7 (from 70 down to 10), and the average usages also go down in the 32-processor case.

The large change in queue depths has to do with the fact that for 32 processors the Central Ring has only two hops. In this case all data packets injected onto the Central Ring are immediately consumed after one hop, since there is only one possible destination. Each consumed data packet generates a just-freed slot. If the system does not allow the use of these just-freed slots, a steady stream of such incoming packets will prevent the packet consumer from inject-

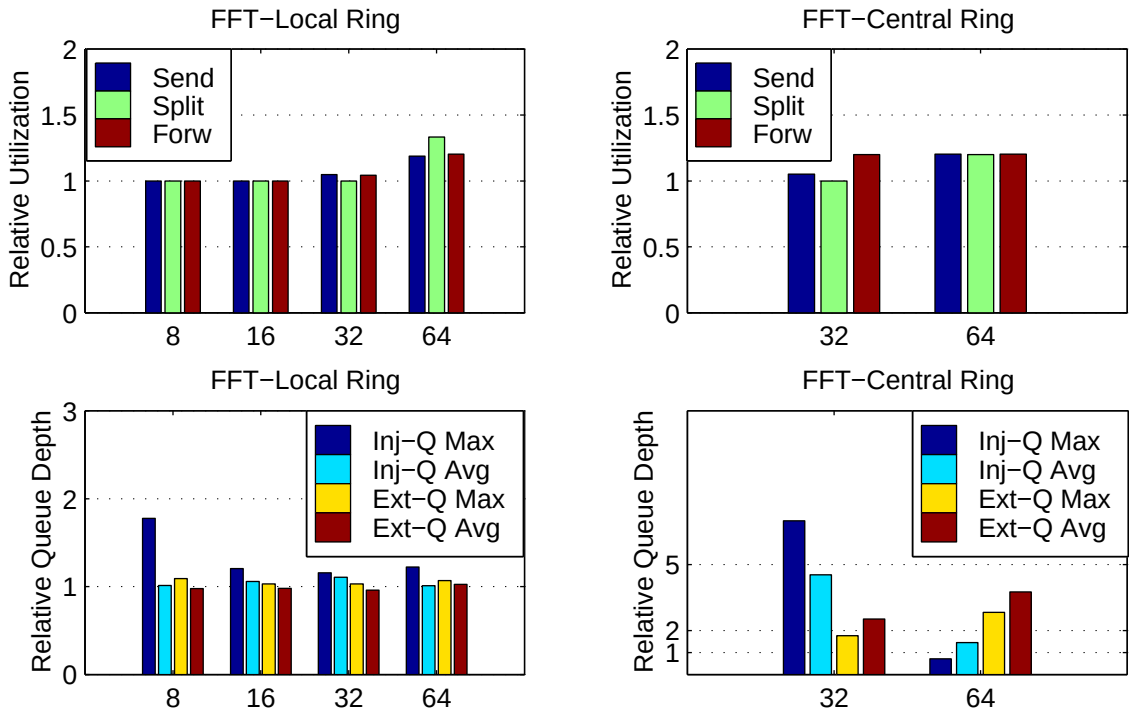

FIGURE 4.10: Use of the just-freed slot. The graphs show the effect of switching to the use of the just-freed slot. In the utilization graphs, a number greater than one represents the relative increase in utilization over the default. In the queue depth graphs, a number greater than one corresponds to a *decrease* (i.e. improvement) in the respective queue depth. (The large improvements in queue depth for the Central Ring at 32 processors are due to the fact that in this configuration data bursts can cause one of the nodes to stall. See the text for an explanation.)

ing any of its own packets onto the ring, effectively stalling it. Note that it is possible for such a scenario to occur with a ring of more than two hops, but due to the larger number of pairs of communicating nodes the probability of all packets being consumed by one node is less likely. (And even if one node *is* stalled, this does not stop other downstream nodes from communicating, nor does it stop broadcast traffic.).

For 64 processors there is actually an *increase* in the value of the maximum for the ring-injection queue, from 196 to 275. The rest of the queue values show improvements. The net effect on performance was to reduce the parallel execution time by 4% at 32 processors, and 15% at 64 processors. The conclusion is that use of the just-freed slot does improve performance, and with only minimal changes to the ring control logic. There are cases where use of

the just-freed slot can cause starvation of a downstream node. For example, if two stations stream data to each other, using up all slots, a third station will not be allowed to inject packets. However, a similar starvation scenario would also be possible in a system that did *not* make use of the just-freed slot.

4.2.4 Network Cache Performance

The most basic metric for Network Cache performance is the hit rate. We consider a request for a remote cache line to be a hit in the NC if it does not end up generating any network traffic. The simplest case is when data fetched for a shared read from one processor can be returned to a subsequent shared read from another processor. It is also possible that the second read could come from the same processor if the line was ejected from the processor's cache. We count this as a hit as well, since the NC is helping to alleviate the processor cache's capacity misses. A more complicated scenario consists of a processor doing an exclusive read for a line that is dirty in another processor's cache. The NC sends out an intervention, with the response going to both the requester and the NC. While more involved, remote accesses are still avoided. There are five possible types of NC hit:

- SHR_LV: A shared read with the NC the owner of the line (Local Valid state). The NC responds with data.
- SHR_GV: A shared read with the line globally shared (Global Valid state). Again the NC can respond with data.
- SHR_LI: A shared read for which the NC mediates the intervention to obtain the dirty copy (Local Invalid state) in a local processor. The line ends up in the LV state.
- EXC_LV: An exclusive read (or upgrade) to an NC-owned line. The NC responds with data or an invalidate, and changes the line's state to LI.
- EXC_LI: An exclusive read (or upgrade), with the NC mediating the exclusive intervention.

Figure 4.11 indicates that while the hit rates can be quite good, there is considerable variability in the behaviour. FFT rarely hits, because its access pattern consists largely of migratory data which has little spatial or temporal locality. Radix has an all-to-all communication phase which is heavily write-dependent. Since Radix's writing pattern is fairly random, the

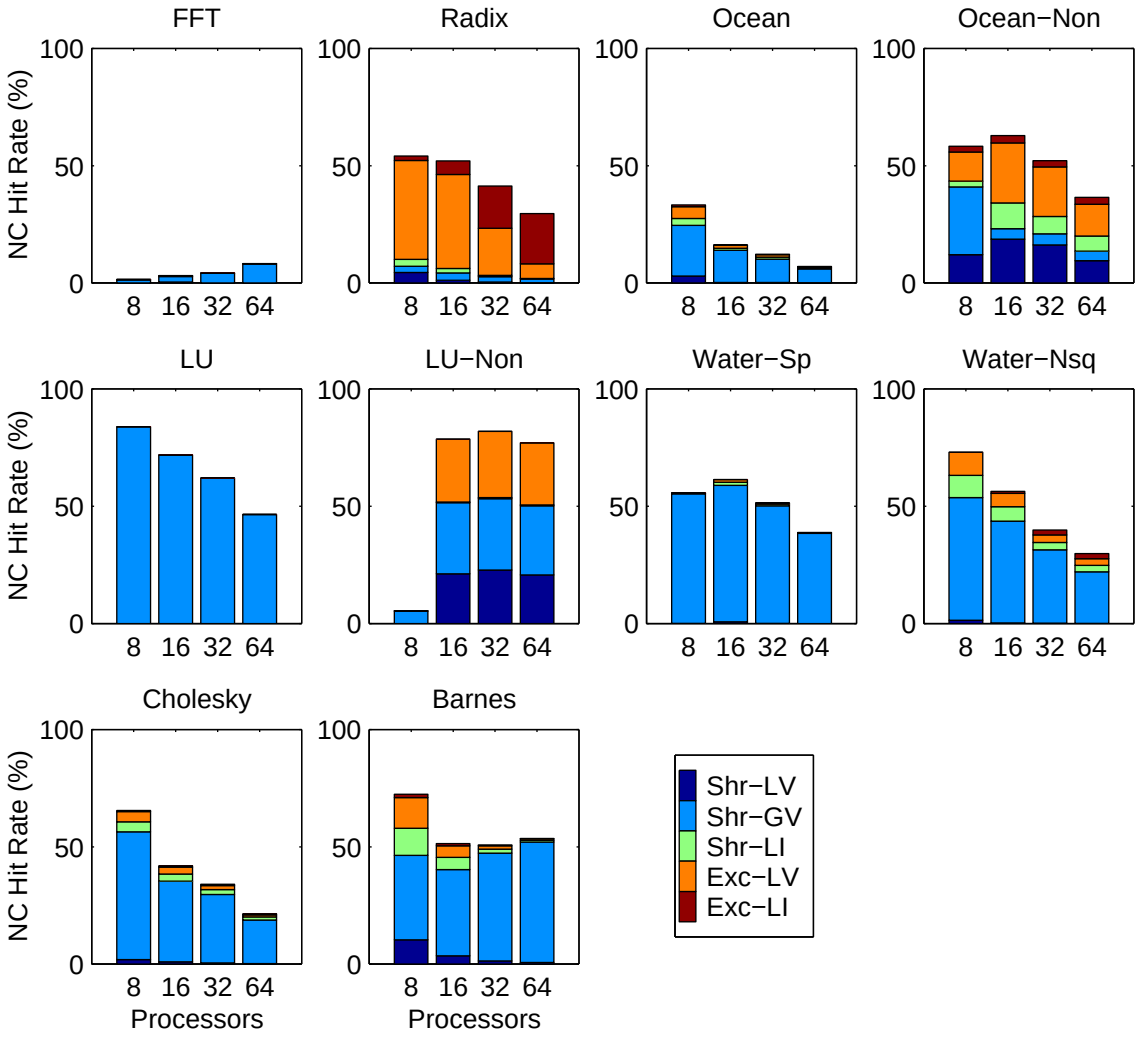

FIGURE 4.11: Network Cache hit rates. The classification of the various types of hit are given in the text. The total number of incoming requests to the NC in most cases is greater than 30,000. The exceptions are FFT (around 3,000) and LU-Non with 32 processors (around 5,000).

probability that a line will be shared on the same station goes down as the number of stations goes up, which accounts for the declining hit rate. (This trend is generally applicable, as can be seen from the figure.) The most prevalent source of hits is from accesses to globally shared read-only data.

It is not possible to draw any specific conclusions about the overall effect of the NC on performance from the hit-rate graphs. While the NC certainly does provide a caching effect (sometimes substantial), the trade-off is increased latency due to the directory lookup for misses that eventually have to go remote. We defer further discussion of net NC performance benefits until the next chapter, where we present results on the effect of NC size, including a system without an NC.

As mentioned in Chapter 3, NUMAchine's NC has a number of architectural features which are novel amongst remote-access caches. The feature with the most potential for deleterious side-effects is the laissez-faire attitude that the NC takes towards directory information. Remember that the NC can in most cases silently throw out old information to make room for new, without notifying local processors or the home memory. The following list enumerates the repercussions of throwing out cache lines with the given NC states:

- LV - This line must be written back to the home memory, which thus gets notification of the event. All local shared copies remain in the processors, but any new accesses must go remote. To get into the LV state, this line had to first exist in the NC in the LI state, then become shared by another processor. This history of local sharing means the line has a high probability of accesses in the near future.
- GV - There is no notification of this ejection. Local shared copies remain. Future accesses have to go remote.
- GI - No information is lost. The meaning of this state is that the NC knows only that there are no copies of the line on the station.
- LI - There is no notification, and this ejection represents the greatest amount of information loss, because this state specifies exactly where the dirty copy is on the station. The dirty copy remains undisturbed. Any remote interventions looking for this line must broadcast the intervention to all processors, and wait for all responses, which is costly. Since there is no place to store this line in the NC, a writeback from the processor is forwarded on to the home memory.

Not all cases of information loss are detrimental. For example, if a line is no longer being actively used, then the loss of information is inconsequential. To gauge how much useful information is actually lost under this scheme, we look at two different statistics. In the first case, we would like to know how many times a remote request (from the home memory) comes into the NC expecting to find information, but does not. The most costly scenario is the one described above where an intervention must be broadcast. A second, less costly, case

occurs when a broadcast invalidate comes in and needs to invalidate any local copies. In the absence of a GV hit (where the Pmask would specify the processors to invalidate), the NC must broadcast the invalidate to all processors. This does not involve much of a performance penalty because invalidates are relatively cheap.

For intervention broadcasts, the result is simple: there are almost none. The majority of the programs showed no broadcast interventions, shared or exclusive, while the number of specific interventions (i.e. those that found directory information) ranged from 20 up to 60,000. Only Cholesky, Barnes and Ocean-Non showed any significant numbers of broadcast interventions. The greatest number was for Ocean-Non, with 122 broadcast interventions versus 56,000 of the specific type (for a 32-processor configuration). The conclusion is that sharing accesses to dirty lines occur with high temporal locality. Information ejected for these lines is usually stale.

The case for invalidates turns out to be much the same. For 8- and 16-processor machines the numbers are identical to those for the interventions: almost all invalidates are specific. For the 32 and 64 processors, the number of broadcast invalidates suddenly jumps up, almost approaching the number of specific invalidates. The reason for this becomes clear if we recall the fact that overinvalidation due to Fmask imprecision occurs only for 32- and 64-processor systems. The broadcast invalidations thus arise not from the fact that directory information was thrown out, but because there was never any information to begin with; stations without any copies are being incorrectly targeted with invalidations. The number of broadcast invalidations recorded in the NC matches the overinvalidation rates from Figure 4.5. Thus we can conclude that GV lines generally get invalidated before their directory information has time to be ejected.

The preceding analysis has only considered the effect of directory information loss on incoming *remote* requests. Local requests can suffer from information loss in two ways. Shared requests to lines that would have stayed in the NC if it had more capacity pay a remote access penalty. We will defer this question until the next chapter, where we use an infinite-sized NC cache to determine the effect of capacity problems. The second case is where dirty directory information is lost. A local request must go to the home memory, which still thinks that the NC has the most up-to-date information on the line. The memory sends the request back to the originating NC, at which point the NC realizes that the line must be locally dirty, but the directory information was lost. The NC must then broadcast the intervention and keep track of responses, even though it will not end up keeping the information. (The space in the

directory is already used by some other line which may be needed by local processors, thus we do not want to eject the current occupant because of a remote request.) In the simulator, these requests are flagged as *false interventions*. Analysis of the output files shows that the story here is the same. In the large majority of cases there are zero false interventions. Barnes at 16 processors shows around 50 false interventions, and Ocean-Non shows around 100 at 16 and 32 processors.

The final conclusion is that the lazy coherence directory scheme used by NUMAchine is a definite win. It avoids sending directory update information wherever possible, and pays almost no price for the lost information.

4.2.5 Request and Backoff Latency

We have seen in some detail in the previous sections how the rings and coherence scheme can affect memory accesses. In this section we will take a step back and look at request latencies. Whatever the ultimate cause, long latencies (for requests or synchronization) are what cause performance loss in a multiprocessor. Though we know that the system does suffer from congestion, the high-level effect on performance is primarily through increases in latency. In this section, we measure the contention-free latency and then compare it to latency measurements for a simulation run that had high levels of congestion.

TABLE 4.3: Base contention-free latency for a local read.

Transaction Step	Latency (in 20 ns clock cycles)
L1 cache miss	*0*
L2 cache miss	*1.33 (4 processor cycles@150 MHz)*
External Agent	*1.5 (30ns FIFO delay)*
Bus (request)	*5 (4-cycle arbitration delay + 1-cycle transfer)*
Memory	*14 (80ns directory lookup + 200 ns DRAM access)*
Bus (response)	*21 (4-cycle arbitration + 17-cycle transfer)*
External Agent	*12 (30ns FIFO delay + 16 data cycles@75 MHz EA speed)*
Total	**55 (1100 ns)**

The basic latency for read requests in an idle system can be calculated by adding up the latencies for each step along the transaction's path. We show the calculation for a read request to local memory in Table 4.3. For comparison, measurement using a logic analyzer connected to the processor card and the bus shows a measured latency to the first word of 1120 ns[8]. The same measurement for a near remote access gave the result 4100 ns. For a far remote, the latency was 4900 ns.

For remote accesses in the simulator, we can do the same type of calculation. A near remote access takes an additional 1160 ns compared to the on-station request. A far remote access adds only 300 ns more since the path is almost the same as for the near remote, with the addition of four IRI FIFO delays and one round-trip traversal of the Central Ring. We thus have the ratios 1100:2260:2560, or 1:2:2.3. The discrepancy between the simulator and hardware for remote latencies is due to overly optimistic assumptions for controller overheads when modelling the hardware. While field-programmable devices are very flexible, their speed is not very high. In order to achieve our 50 MHz clock rate, we had to add many synchronizing flip-flops on inputs (e.g. for FIFO empty flags) to maintain setup times, and break complex decoding logic into multiple stages. The prototype uses nearly a dozen controllers in the NIC's datapath, each contributing 3-4 cycles of latency. This overhead is incurred on both the local and remote stations, and accounts for roughly 2200 ns of extra delay. The balance of the difference, around 700 ns, was found to come from late changes during debugging to the operation of the ring controller, to force an isolated read response to use only every other ring slot. Unfortunately, we discovered these discrepancies too late to re-run the simulations. This causes our performance numbers to be optimistic, particularly for programs with low NC hit rates.

The contention-free numbers will increase in the face of network congestion, and also because of backoffs. In Table 4.4 we show latency measurements for one example of a highly congested system: Ocean with 64 processors. The local and remote memory latencies increase by about 25% and 60% respectively. The next two numbers show the effect of backoff on the latency. The simulator output does not show the average number of retries required, only the average latency. Although the latency increases are very large —380% and 330% for local and

8. To perform this measurement, we looked at an R4400 signal pin called ValidOut*, which indicates that a request is ready to come out of the processor. We measured up until another R4400 signal, ValidIn*, was asserted, meaning that the *first* doubleword of data had been returned. By measuring to the first doubleword instead of the last, we are assuming a critical-word-first arrangement of the cache line.

TABLE 4.4: Congested latencies for a 64-processor Ocean simulation.

Access Type	Number of Requests	Average Latency
Local Mem	*2000*	*1380 ns*
Remote Mem	*5000*	*6420 ns*
Local Memory - Retry	*48*	*4220*
Remote Memory - Retry	*106*	*13500 ns*

remote, respectively—the frequency with which they occur is low. The number of retries is generally low for all the programs. One exception is Cholesky, where over 25% of requests to the NC ended up having to retry, with average latencies of around 9000 ns instead of the typical time for an NC hit of around 1100 ns.

The heavy latency penalty and lack of fairness in the retry mechanism make it one of the weaker points of NUMAchine's architecture. How weak can only really be answered by modifying the simulator to model an ideal system with pending request queues. This is left for future work.

4.2.6 Flow Control

The flow control mechanism can result in ring-stoppage occurring on either the upper or lower level rings, or busy-waiting on the bus. The simulation results show that for the applications we tested the Central Ring never locks up. (The Central Ring locks if the queue in the IRI going from the Central Ring down to the Local Ring fills up.) On the Local Ring, no ring locks are generated by the NIC cards. The only case where locking occurs on the Local Ring is from the IRI. (In this case it is the upward queue from the Local to Central Ring in the IRI that fills up.) Figure 4.12 shows that most programs do not cause the ring to lock, but for those that do, such as Radix, locking occurs frequently. For Radix the all-to-all communication pattern causes the Central Ring to become a bottleneck. The IRIs cannot inject packets onto the Central Ring fast enough, and the resulting buffer overruns cause the Local Ring side to lock up. This could be fixed with a faster Central Ring. Faster rings will be explored in Chapter 5.

On the bus, busy waiting happens if the sinkable- or nonsinkable-busy flags are set for the target(s) of a transaction. The arbiter does allow other non-blocked transactions to proceed,

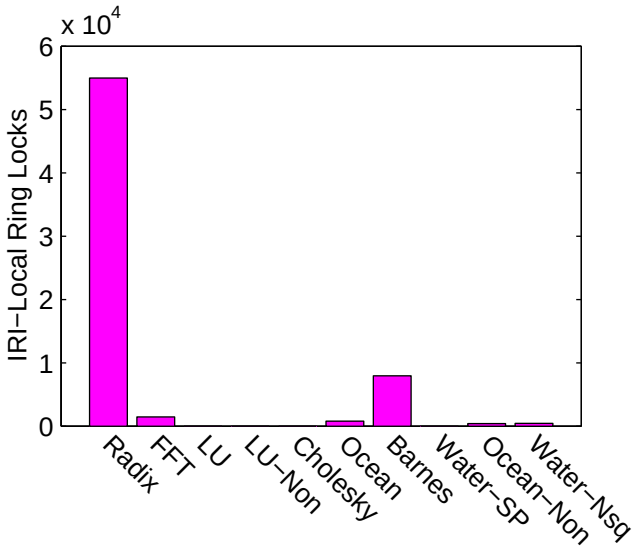

FIGURE 4.12: Local Rings locks caused by the IRI.

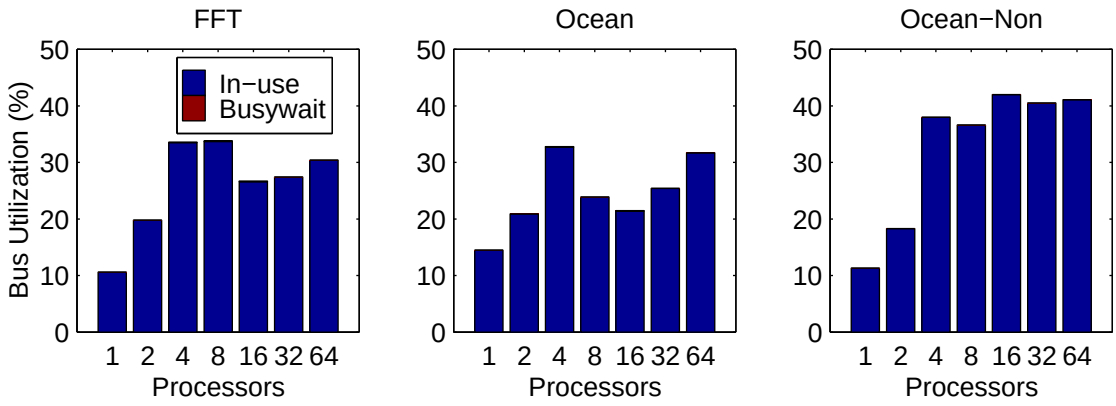

FIGURE 4.13: Average bus utilization. Only the three heaviest bus users are shown. The rest of the programs have utilizations less than 20%. The busywait fractions are negligible in all cases (< 0.1%).

though, so the performance penalty is not as severe as for ring locks. For most programs, the bus utilization is less than 20%. From Figure 4.13 we see that the busywait fractions are minuscule. Only the flow control on the ring plays any role in NUMAchine's performance.

4.3 Conclusion

We began by defining ‘good’ performance in the context of a simple, low-cost multiprocessor. Consistent speedup growth with increasing system size, as well as a lower bound on the speedup of roughly 20 at 64 processors represent our basic criteria. We showed that in terms of parallel speedup, which is speedup with sequential code time set to zero, NUMAchine exhibits good performance. We then went on to analyse certain architectural aspects discussed in Chapter 3.

We found that the rings perform fairly well, although they do suffer from congestion problems. Allowing a ring-injection buffer to use a ring slot that has just been freed up in general helps lessen the buffer’s congestion, has a small overall positive effect on performance, and requires no extra logic.

NUMAchine’s filtermask structure and lazy coherence directory maintenance were shown to be efficient. The overinvalidation rate from the use of the filtermask reached highs of around 2.5 (i.e. on average 2.5 as many stations were sent invalidations as actually needed them). This was acceptable since superfluous invalidates to a processor’s cache incur very little overhead. This result also indicates that the filtermask works to reduce invalidation traffic compared to a scheme where all invalidations cause broadcasts to all stations. This filtering of invalidation traffic was shown to be important for system performance in NUMAchine’s predecessor, Hector [Farkas 1992].

Interventions, on the other hand, are much more expensive. The Network Cache, because of its lazy approach to maintaining directory information, can reach a state where it must broadcast interventions to all processors. We showed that this state almost never occurs, indicating that the NC has a high probability of holding onto coherence directory information that is needed in the future. Hit rates in the NC generally ranged from 50% down to 20%, with the lower rates occurring for higher numbers of processors. The overall magnitude of the performance enhancement from using the NC is investigated in the next chapter.

We took a brief look at request latencies and the effect thereon of the binary-backoff NACK-and-retry mechanism. For requests that did not require a backoff, average latencies for local requests increased by only 20% over the base local latency in a contention-free system. Remote requests showed larger latency increases of around 300% on average due to congestion in the network. For requests that did require backoffs, the increases in local and remote

latencies were much higher: 400% and 600% respectively. This result shows that the backoff mechanism has the potential to be a serious source of performance degradation. An alternative scheme that queues requests and services them in FIFO order was proposed, but comparison of the two was left to future studies.

Finally, we showed that the flow control scheme rarely comes into play, except for injection onto the Central Ring, and even then only for some programs. The queueing provided in the system is sufficient to avoid flow control, even in the face of high average bus utilizations.

In the next chapter we will tune and modify certain aspects of the system to try and gain some insight in to ways of increasing NUMAchine's performance.

CHAPTER 5 *Simulation Studies*

In this chapter we use Mintsim to explore the parallel processor design space. The goal is to determine what changes could be made in the design to most effectively increase overall system performance. Changes will be considered to the Network Cache, rings, coherence protocol and consistency scheme. The general metrics used to measure system performance continue to be parallel execution time and the associated parallel speedup.

The design space we are considering is huge. A parallel processing system contains hundreds of different independent system parameters, making a systematic exploration of the entire space impossible. In selecting certain aspects of the system for modification, it is just as important to determine which others are to be held fixed, and why. Major architectural features that remain unchanged are described in the following section, with justifications for each.

5.1 Fixed Simulation Parameters

The most important ingredient in any multiprocessor system is the network. For the purposes of this work, the basic network topology will stay ring-based. One major reason for doing this is practical and has to do with the simulator. The design and verification of the simulator's ring components were very time-consuming. In addition, the single-path nature of the ring is an essential ingredient in the cache coherence protocol. Switching to a mesh or some other multiply-connected topology would necessitate not only new network component code, but also a completely redesigned (and verified) coherence protocol. Besides this practical limitation, there are other good design reasons for limiting the scope to rings. Previous work has indicated that rings compare favourably to meshes [Ravindran 1997]. Also, as mentioned previ-

ously, one of the goals of this work is to show that respectable parallel system performance can be achieved without resorting to complicated or expensive hardware. We will show how the ring performance from the last chapter can be improved upon with minimal added complexity.

Our basic approach is to use *problem-constrained* (PC) scaling [Culler 1999]. Basically, we carefully choose a fixed problem size as described in the rest of this section, and then run this problem on a varying number of processors. The alternatives are *time-constrained* (TC) or *memory-constrained* (MC) scaling. In TC scaling, the problem size is increased with the number of processors *N*, such that the total execution time (wall clock time) is kept fixed. In the real world, users are often limited by the amount of time that can be spent on a given problem. (Users typically would like a job that runs overnight, and will scale down the job to fit into this timeframe.) In this case it is the total amount of work done which scales with *N*. In TC the dataset size for each cache generally does not decrease, or does so slowly, thus it avoids jumps in performance as data sets start fitting entirely into caches (which can happen in PC). However, because our simulator actually runs serially, TC scaling would cause our simulation times to scale with *N* as well, which would lead to week-long simulation runs for each datapoint for programs like Barnes. The last type of scaling is *memory-constrained*, where the total amount of memory used by each processor is kept fixed. This method is useful when memory is limited, because overflowing the memory can lead to severe thrashing. The drawback is that it is difficult to determine speedups for MC, because it is hard to determine the actual amount of work done for *N* processors for all but the simplest applications. We argue below that by taking care with the choice of problem size, the PC model can avoid any anomalies, and provide good results.

We will not change the basic unit of one processor with some fixed amount of dedicated cache. (We will, however, change the associativity of the cache, as discussed below.) Since the simulator uses MINT as a front-end, it is not feasible to model a different instruction set architecture (ISA) or a superscalar design[1]. Outside of the occasional custom-designed chip (e.g. the Tera [Bokhari 1998]), the most common processors used in parallel systems are RISC-based, thus we would in any case expect very little difference in the memory reference stream by switching to a different ISA.

1. We also avoid newer architectures such as simultaneous multithreading or single-chip multiprocessors.

Because processor speed is increasing faster than memory speed, we increase the speed of the processor in the simulator to model systems that will exist over the next few years. This is done by setting a parameter in the input file to indicate the speed of the processor, as well as loading in a different set of opcode timings. In the NUMAchine prototype, the processors run at 150 MHz, but for the purposes of these experimental studies we increase the speed to 1000 MHz.

We model system sizes of up to 64 processors, since, as we have argued previously, this is the upper limit on expected real systems for the next few years. We want to estimate the performance of running ‘large’ problems on such a system. In this context ‘large’ means that we do not want an application’s entire dataset to be able to fit into the cache, for any number of processors. The latter point is subtle but important. If we are not careful, then it is possible to choose a problem size which does not fit in the cache for small numbers of processors, but does start fitting into the cache before the system size reaches our maximum. In such cases the performance will show an abrupt jump in performance, causing *superlinear speedup*. Clearly this is an artifactual result, due to the crossing of some architectural boundary and should be avoided. If we were to use the default NUMAchine 1-MB L2 cache, then on a 64-processor machine we would need problems with data sets much larger than 64 MB. Such large problems would take on the order of a few hours to run in hardware, which means weeks of simulation time for each datapoint. The standard solution in this case is to scale down both the system cache sizes and the problem sizes. Scaling down is complicated by the fact that complex systems such as a multiprocessor are highly nonlinear, meaning that it is not possible to just reduce all parameters by some constant factor.

In scaling down we have to pay careful attention to the *data set* and *working set* sizes for a specific application. The data set size is the amount of memory required when running on a uniprocessor. Note that this is not the same as the problem size, which depends not only on the data set size, but also on other program parameters (e.g. the number of iterations to converge on a solution). The working set is more nebulously defined as the ‘current’ data being processed at a given point in the program, and as such an application can have numerous working sets over time. It is common for working sets to fit entirely into caches, even small primary caches. (In fact, a well-coded program should try to fit inner loops into the cache, for example by splitting a large inner loop into multiple smaller loops, in order to take better advantage of spatial and temporal locality.) For this reason, working sets tend to scale fairly slowly with

increased problem size. Thus, in scaling down, we would like to have caches that are large enough to accommodate typical working sets, because it is reasonable to expect that this situation would also hold for large problems running on a real machine. But we would like the caches small enough that the full per-processor data set does *not* fit into the cache.

Besides cache size considerations, we also have to be careful when selecting the other cache organization parameters: line size and associativity. Both of these have a direct impact on miss rates. Of particular importance in scaling down are capacity and conflict misses. Increasing the capacity misses is actually our goal, since we want to mimic data sets that are too big for our caches. Cold misses are not considered, other than to ensure that our simulation runs are long enough statistically speaking to make cold-cache effects negligible. Conflict misses represent a hazard. The default NUMAchine configuration uses direct-mapped caches for both primary and secondary caches, this functionality being fixed by the choice of the R4400 processor. However, with a 1 MB secondary cache, the probability of conflict misses is low. As the processor cache size is reduced, the conflict miss rate increases. We can compensate by increasing associativity, but the question is how much is reasonable? Modern RISC processors such as the MIPS R10000 and Alpha 21264 use 2-way associativity, while others such as the PowerPC 604 are already 4-way associative [Burd 1999]. Another consideration is that the shared memory model under which the Splash2 programs are compiled uses four distinct memory regions: shared, stack, heap and data/bss[2]. The natural choice is thus 4-way associativity. (Actually, initial simulation studies were performed with direct-mapped caches, so that the results showed severe anomalies. Extensive checking using simulator debug traces showed that the anomalies were due to conflict misses.) Figure 5.1 shows the performance improvement for 4-way associativity over direct-mapping for one particular pathological case where the Barnes application with 64 processors had a severe conflict miss problem. (It turned out that in one specific inner loop, just one of the 64 processors happened to have a stack area that conflicted with the data/bss region, which was enough to kill the performance.) We also tried using 2-way associativity, which showed some improvement but still suffered from excessive conflict misses.)

Having settled on 4-way associativity, our solution for selecting an appropriate cache size was to first pick appropriate application sizes given simulation time constraints, and then set the size of the processor cache small enough to steer clear of problems.

2. See Appendix A for a description of these regions.

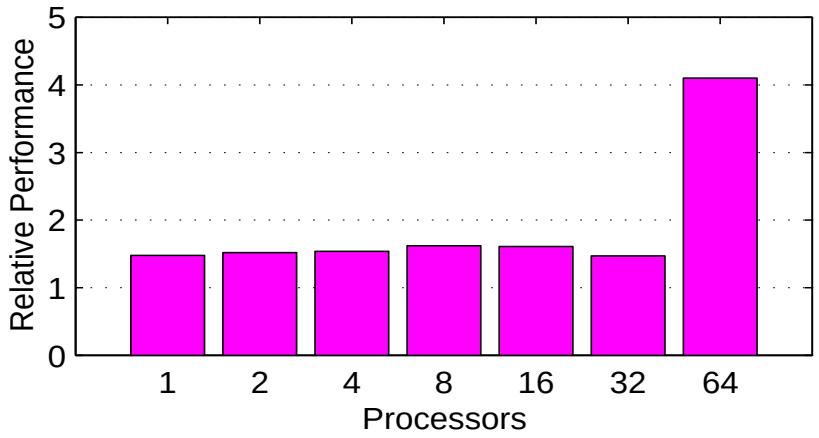

FIGURE 5.1: Direct-mapped versus 4-way associative processor caches. The performance increase shown is the ratio of execution times of the parallel sections of the program for the direct-mapped version over the 4-way associative version for a particular run of the Barnes application.Analysis of simulator debug traces showed that the large anomaly at 64 processors was due to cache conflicts.

Using the curves of cache usage in the Splash2 paper [Woo 1995], an L2 cache size of 64 KB was chosen. For *N*=64 this results in 4 MB of total cache in the system. To be conservative, application parameters were chosen such that the total memory allocated to shared regions was > 16 MB. (These shared regions are where MINT stores the globally shared data structures which are divided amongst the processors.) Since the shared area does not include local (private) data such as the stack and the heap which also use up cache space, this choice should be quite safe. Numerous test runs of each application were conducted with varying problem sizes, using the MINT '-s' switch to specify the maximum allowable size of the shared memory region. The applications were run through the simulator with the switch set to 16 MB with ever increasing problem sizes until MINT died with an error message indicating that the amount of shared memory requested by the application was more than the maximum. This set of parameters was then used as the default for the studies that follow. A complete list of the Splash2 application and kernel problem sizes is given in Tables 5.1 and 5.2.

Another consideration when choosing the problem size is run length. Simulation runs should last long enough that they are statistically significant. The choices above all result in

TABLE 5.1: Problem Sizes for Splash2 Kernels

Application	Our Parameters (If different)	Splash2 Default Parameters	Data Set Description
CHOLESKY	chol_tk29.O -C65536	chol_tk18.O -C16384 -B32	Large sparse matrix blocked to 64KB cache. Splash2 default uses the medium matrix, with 16KB blocking.
RADIX	-n4194304 -m8388608	-n262144 -m524288 -r1024	4M integer keys with maximum value 8M. Default is 1M keys with 2M maximum.
LU	-n512	-n128 -b16	512x512 matrix. Default is 128x128 matrix.
FFT	-m20 -n512 -l7	-m10 -n65536 -l4	1M complex doubles, 512 cache lines of size 128B. Default is 1K points, 64K lines of size 16B.

simulated run times on the order of a few seconds, which for the basic simulator timescale of 1 ns means over one billion cycles. Even with cache miss rates less than 1% this guarantees many millions of memory references into the hierarchy, and should render any cold-cache effects negligible.

With two levels of cache in the processor, it is still possible to have a minor cross-over problem when local data structures fit into one level of cache but not the other. To avoid any such anomalies in the results, only one level of cache is used. (The dual-level cache is actually hardwired into the simulator code, so what was done was to make both the L1 and L2 caches have the same size and line size, thus effectively behaving like a single level of cache.)

5.2 Algorithmic Speedup of the Test Programs

Before using these applications to probe our design space we need to verify that they are the correct tools. As we saw in Figure 4.3, certain combinations of programs and problem sizes do not parallelize efficiently even under the PRAM model. We want to ensure that we use only programs that do parallelize well, in order to avoid drawing false negative conclusions.

TABLE 5.2: Problem Sizes for Splash2 Applications

Application	Our Parameters (If different)	Splash2 Default Parameters	Data Set Description
OCEAN	*(same as default)*	-n258 -e1e-07 -r20000 -t28800	258x258 grid
BARNES	8K	16K	8K particles, instead of the default 16K
WATER	729 molecules	3 timesteps 512 molecules	interaction of 512 water molecules
FMM	*(same as default)*	16K particles	interaction of 16K gravitating particles

Mintsim can model a perfect memory system, and thus measure algorithmic speedups. This feature is turned on by setting a parameter in the input file, after which the processor module considers every access to the L2 cache to be a hit, no matter what the cache line state.

The results for our subset of the Splash2 programs are shown in Figure 5.2. These differ from the curves in Figure 4.3 because we have changed the problem sizes. From the curves it is clear that LU, Cholesky, Water and FMM are not good choices for out test programs. The main reasons for the non-ideal algorithmic speedups are code overhead due to parallelization for Cholesky, and sub-optimal workload partitioning for the rest. (We determined this by examining the processor utilization statistics from the simulation output. Mintsim keeps track of what percentage of time is spent by processors running or waiting for memory accesses and barriers. A large fraction of time spent waiting for barriers in the PRAM model indicates a workload partitioning problem.)

We thus choose FFT, Radix, Ocean and Barnes with which to do our testing. As described in the Splash2 paper [Woo 1995], FFT and Radix have high communication-to-computation ratios, and work well as stress tests. In addition, the nature of the communication for the two is different. FFT shuffles data using a butterfly pattern, thus the sharing is migratory, while Radix has an all-to-all data reshuffling phase which generates heavy coherence traffic. Ocean and Barnes are good examples of typical scientific applications, complementing the two kernels.

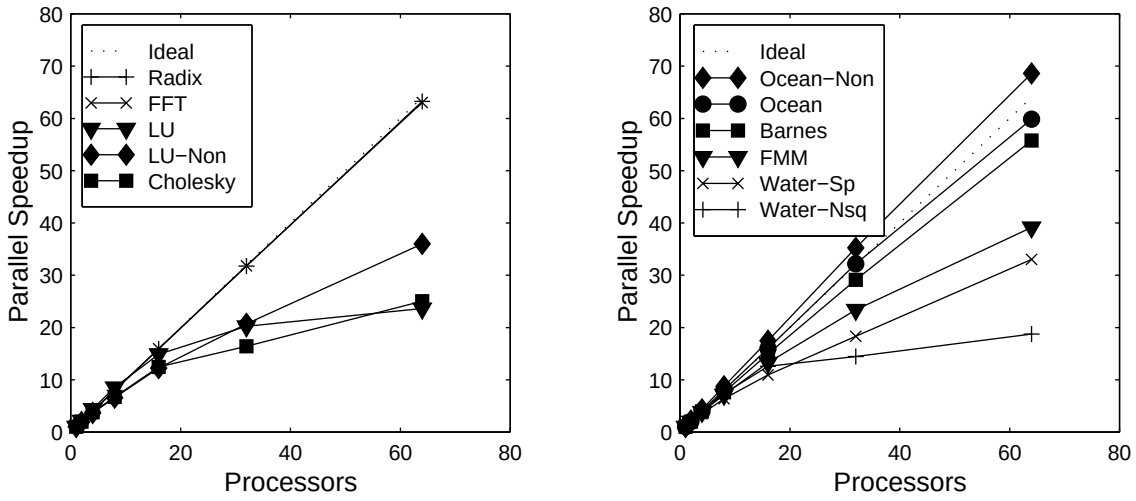

FIGURE 5.2: Algorithmic parallel speedups for the experimental system. These results differ from [Woo 1995] due to the choice of different problem sizes. The Radix and FFT curves lie on top of each other.

5.3 Baseline Performance and Page Placement

With our upper bound defined, we now move to the baseline case, which considers the model above to be running with the real NUMAchine memory and network. Most of the non-fixed simulation parameters are the same as for the simulations presented in Chapter 4, with one exception. Due to the faster processor and smaller cache there will be much more traffic in the system, which necessitates an increase in the queue sizes throughout the model. We make them large enough that they will never overflow, meaning that they are effectively of infinite depth. This is actually beneficial, because queue overflows would cause the NUMAchine flow control mechanism to trigger, which would cloud the results. For these design-space studies we *do* want to model queueing delays caused by contention in the interconnect, but we do *not* want to worry about the effect of finite queues, since this is really an implementation issue and only indirectly related to the architectural questions we wish to study[3]. A good architecture

3. Since the number of requests and writebacks is limited, there are theoretical maxima. With four writebacks from every station going to one memory, we would need space for 60 cache lines of 128 bytes, or 7680 bytes.

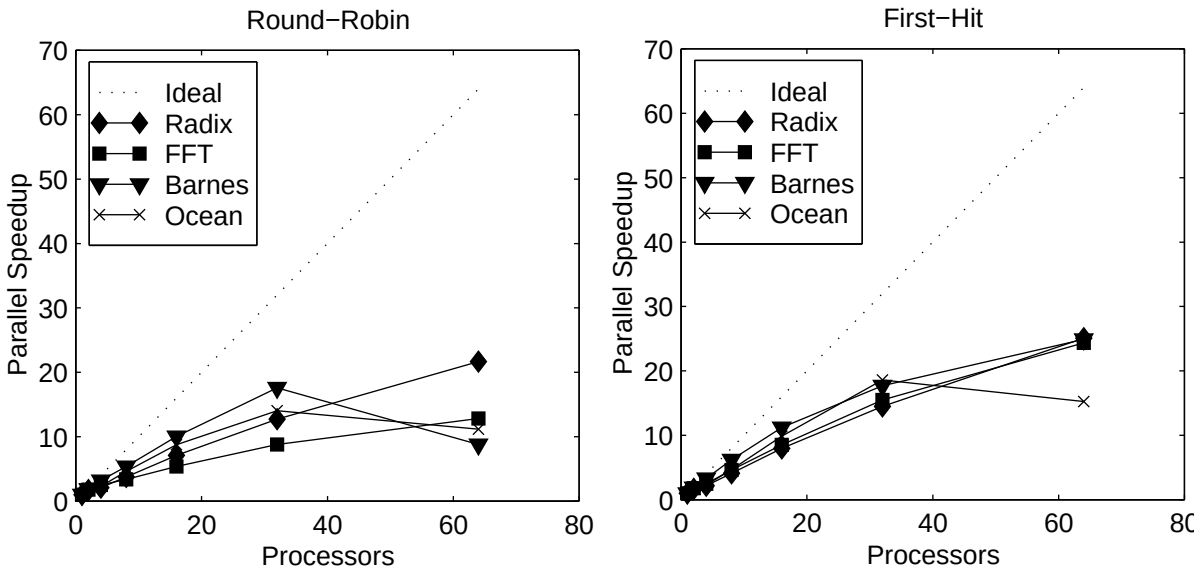

FIGURE 5.3: Parallel speedups of the baseline system with a round-robin and first-hit page-placement policies. Parameters are the same as the NUMAchine defaults, except as noted in the text. The first-hit policy reduces the amount of 'false' remote traffic and coherence overhead.

should minimize the need for large queues, but ultimately one puts in the largest queues one can afford, to reduce the frequency of activation of flow control mechanisms.

In Figure 5.3 we see that the performance is poor for all four programs up until 32 processors using a round-robin paging policy and then becomes a slowdown for Barnes and Ocean. With the faster processors the memory system is being pushed beyond its capacity. When we use a first-hit policy, pages that were unnecessarily placed on remote stations are eliminated, leading to reduced remote traffic and lower coherence overhead.

Though there may be practical implementation problems with a first-hit policy— as mentioned in the previous chapter—they are all at the level of the OS, and do not relate directly to the hardware architecture. The use of a first-hit strategy allows us to more correctly attribute any blame for performance degradation to the hardware. A first-hit policy is assumed for the rest of the results in this chapter.

In Figure 5.4 we break down the performance of the applications by examining the processor utilisations, which show the time spent by the processors doing real work, waiting for

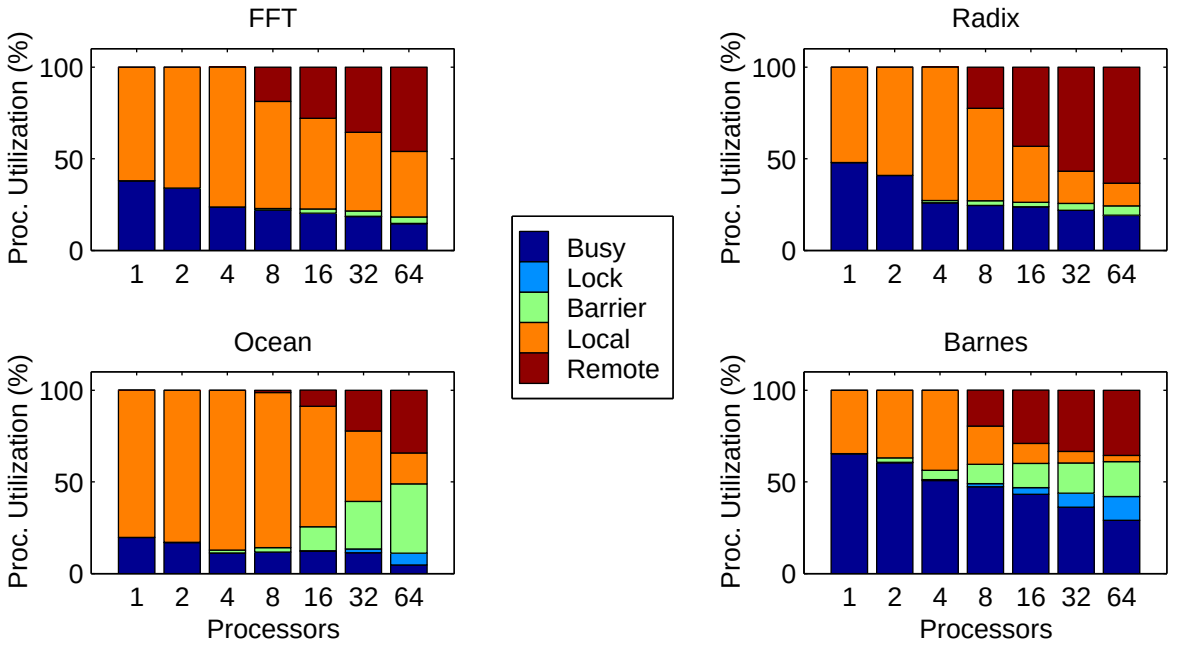

FIGURE 5.4: Processor utilisation graphs corresponding to the first-hit speedup curves in Figure 5.3. Since the algorithmic speedups for all of these programs are nearly ideal, the significant fraction of time spent in synchronization for Ocean and Barnes must be due to overhead created by our architecture.

local or remote memory references, or waiting for synchronization. The poor performance of Ocean (and to a lesser extent Barnes) in Figure 5.3 corresponds to the large fraction of time spent waiting for synchronization in its utilisation graph. Since we know that all four programs have nearly ideal algorithmic speedups, this overhead is being caused by our architecture. An examination of the simulation output files for Ocean reveals that the degradation is due to the backoff mechanism. In the other three programs, the number of accesses that required retries either stayed the same or decreased in moving from 32 to 64 processors, while the average latency for retried requests stayed the same. For Ocean, however, the number of requests requiring a retry *increased* by 50% in the 64- versus the 32-processor case, and the average retry latency for each different category of request doubled. Barnes did not show the same jump in retry overhead, although its total number of retries was considerably larger than Radix and FFT. In section 5.4.1 we propose a more efficient backoff mechanism.

5.4 Comparative Studies

We now begin exploration of the design space, our goal being to improve performance. The areas that we consider are the rings, the consistency model, the coherence protocol, the network cache and the station bus. The NACK-based retry mechanism is closely related to the coherence scheme, both of which are discussed in the next section.

5.4.1 Coherence Overhead

The main question with regards to cache coherence is how much overhead it imposes on the system, and whether it is worth optimizing the protocol. NUMAchine's cache coherence scheme was designed to make use of the efficient ordering and multicasting properties of rings to achieve low overhead.

An essential component of the overall coherence protocol is the NACK-and-retry backoff mechanism, which we introduced to handle access to locked coherence directory lines. If one assumes that contention is usually low, then this approach should work well, but the results of the previous section show that this assumption is false. Our NACK-and-retry scheme suffers from fairness and liveness problems. Because there is no notion of priorities between retries and regular requests, there is no guarantee that a given request will get through promptly or even at all. A much better procedure would be to enqueue and/or merge multiple requests at the memory and Network Cache. This would either mean that every directory entry would require space and logic allocated for a queue to hold the maximum theoretical number of requests (complex and non-scalable) or that a generic queueing pool would need to be allocated and managed to handle any overflows (scalable but still complex). The advantage to using the NACK scheme is that it is very simple and cheap to implement.

To begin our analysis, we run the simulations with coherence 'turned off' to find an upper limit on the amount of performance improvement achievable. In practice this means the following three changes are made to the memory model:

- Stores to the processor cache that find the line present, *no matter what the state (shared or otherwise)*, are treated as hits. No coherence information is passed on to the memory or NC. Of particular note, no upgrades are generated in this scheme, since stores to shared lines are considered hits.

- Read requests (shared or exclusive) to home memory return data unconditionally. The directory is not checked, nor is it updated.
- For the Network Cache, if the line is present in *any state*, it is considered a hit and data is returned. Note that cache misses still have to go remotely to fetch the line, although the remote memory is guaranteed always to hit. Lines that are locked due to other remote accesses in progress still generate NACKs as before. Note that this is the *only* type of NACK that can still occur.

By eliminating all NACKs, except for those to a line locked in the NC by some still-pending remote access, we can also measure the temporal locality of requests for the same cache line to the NC. If processors tend to all access shared lines at the same time, we should see a large number of NACKs. If the accesses are more spread out in time, these NACKs will always be converted to hits under the no-coherence model.

The results are shown in Figure 5.5, and make sense in light of the discussion at the end of the last section. For Ocean the improvement is drastic, indicating that the slowdown between 32 and 64 processors was indeed due to coherence overhead. Barnes shows a slight improvement, with Radix and FFT showing almost no improvement, corroborating the evidence in the utilisation graphs.

The lack of improvement for FFT and Radix points to the conclusion that the coherence overhead is low. The rationale for this statement is that in turning off coherence we eliminate three types of overhead:

- All NACKs, except for local NACKs from the NC.
- Latency to obtain write ownership.
- Latency for remote interventions, because home memory always hits.

FFT and Radix do not have many NACKs, but their simulation output files do show large numbers of write accesses. The negative result for these programs shows that the second two types of overhead must be low. For Radix, the reason for the low overhead is that both with and without coherence the NC hit rate is around 85%, meaning that the coherence protocol requires mostly local transactions. For FFT, the NC hit rates are around 5%, but most of the writes occur to lines which are shared locally on-station, which require only on-station invalidations.

It is not clear how representative these types of reference behaviour are of ‘real world’ applications. This is a generic problem with benchmarks suites, though, and is not specific to our analysis. The strongest conclusion we can draw is that, other than the effects of the back-

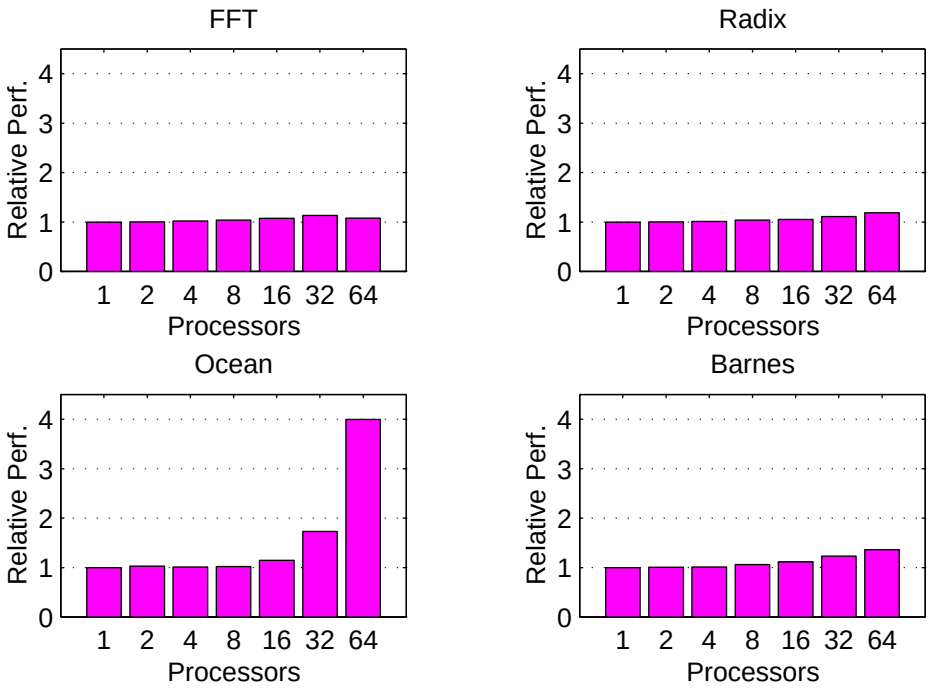

FIGURE 5.5: Turning off cache coherence. The relative performance with cache coherence turned off are shown. The ratios are with respect to the baseline case.

off mechanism, our coherence protocol (including the effect of the NC) performs well for our test programs.

Commercial multiprocessors are frequently being used nowadays for non-scientific applications such as *online transaction processing* (OLTP) and *decision-support systems* (DSS), which make heavy use of parallel databases. This analysis needs to be extended to cover these new application domains, particularly since their access patterns are quite different from those of the Splash2 programs [Barrosso 1998]. The Transaction Processing Council (TPC) provides benchmark suites for both OLTP and DSS, called TPC-C and TPC-D respectively [TPC 1999]. However, both of these suites require a parallel database engine. Porting a database to NUMAchine is a massive task, which is far beyond the scope of this dissertation, and is left to future work.

5.4.2 A Relaxed Consistency Model

As mentioned in Chapter 3, NUMAchine uses a sequential consistency model. Sequential consistency is generally considered in the literature to impose severe restrictions on performance. While this is true when using traditional multiprocessor networks such as meshes and hypercubes, it is not clear that it is as significant for NUMAchine's rings. As mentioned before, the rings already provide a natural sequencing mechanism, so the actual overhead in our case is not as severe. (It could be argued that the choice of rings to begin with is a bad starting point, thus we should not expect much improvement. In our defense we point to other studies that indicate rings and meshes can compare favourably [Ravindran 1996, Hamacher 1997]). The goal of this section is to get a rough estimate of this overhead.

The method presented here does not rigorously model true relaxed consistency. We make no modifications to the applications, and the modifications in the simulator are a minimal set, basically changing the ordering of certain coherence operations and loosening invalidation constraints.

As mentioned in Chapter 3, NUMAchine's rings have sequencing nodes to cause a global ordering of broadcast invalidates. In this section we turn that sequencing off, and consider invalidates to be immediately active on reaching the highest level of ring necessary to reach all targets. We also add an Upgrade Response command. Since NUMAchine uses the Invalidate command as both an ACK to the requester and a kill to other shared copies, the ACK can take longer to arrive than if it were a separate point-to-point command. In this section we separate the functionality of the two, and send the Upgrade Response *before* sending out the Invalidate. If the requester is on the home memory and there are globally shared copies that need invalidating, this can significantly decrease the latency for write permission.

We also make changes to the memory and network cache modules in the case where an exclusive request needs a data response plus an invalidate. Normally in NUMAchine the invalidate must be sent out first, but here we switch the order. They are still sent out back-to-back, so this effect should be fairly minor.

The result is that there is negligible change in performance. The changes are less than 1 %, so no graphs are shown. If we couple this result with recent work showing that modern out-of-order microprocessors can reap many of the performance benefits of relaxed models, while keeping the sequential model [Gniady 1999], there is not a strong case for pursuing relaxed consistency within the NUMAchine framework.

5.4.3 Central Ring Speed

Clearly the raw speed of the network has a tremendous bearing on performance in any multiprocessor system. In a shared memory system with hardware cache coherence it is particularly crucial, because the programmer has much less control over communication than in a message-passing paradigm. The general goal in designing the network is to reduce latency. Congestion in a real network is a highly nonlinear effect: a very small change in the speed of one particular link can cause sudden and drastic performance changes as the contended resource causes knock-on effects throughout the memory hierarchy. Bursty traffic is usually the worst offender in these situations, although multiple processors generating streams of writebacks could also saturate the network. The goal of this section is to investigate the effect of varying the speed of the upper ring to find the point after which increasing the speed will lead to diminishing returns.

The default speed for both levels of ring is 50 MHz. We keep the system size fixed at 64 processors, because this is the worst-case traffic scenario for the Central Ring. From the results of Chapter 4 we know that the ring-injection queues into the top level ring can become very full. Increasing the Central Ring speed should help clear out the queues, but on the other hand will increase the rate of requests being injected into the lower levels of the hierarchy, and could possibly cause the problem to move elsewhere. The key point is that the two levels of hierarchy must be well-balanced. We would like to find this balance point.

If we consider a steady-state pattern of requests flowing throughout the system, with the probability of any two processors communicating being equal, then we can work out theoretically where we think the balance point should be. Let us assume for simplicity that each processor has a total required bandwidth, B, for requests that go to another station (i.e. we ignore traffic on the station bus), whether that station is on a local or remote ring (see Figure 5.6). Assuming the access patterns are the same for the different processors, then some fraction of B is needed for traffic to stations on the same local ring, and the remainder is used for remote stations. Let us call these fractions f_{LR} and f_{RR} respectively, with the condition that $f_{LR} + f_{RR} = 1$. In the absence of contention we can simply add the required bandwidths to arrive at a total. A local ring carries the traffic for both remote and local requests, thus with four stations the necessary local ring bandwidth is $4B$. A fraction Bf_{RR} goes up across the central ring, and since each central ring link sees the traffic from all 16 stations, we have a total

requirement of $16Bf_{RR}$ for the central ring. The ratio of central ring to local ring bandwidth is then $4f_{RR}$. If an application is equally likely to access a station on a remote or local ring, then f_{RR} is 4/5 (there are 12 remote stations versus 3 local ones, or a fraction of 12/15), and we would need just over three times as much bandwidth at the central ring level. We expect most applications to exhibit somewhat better locality, meaning that f_{RR} would be lower. Our estimate then is that central ring bandwidth should be two to three times that of the local ring.

Figure 5.7 shows the results of increasing the central ring speeds in increments of 50 MHz, up to a maximum of 250 MHz. The performance increases by 11% at 100 MHz, and then flattens out. The second graph in the figure shows what is limiting the performance. At 50 MHz the injection queues have large maximum and average depths. Increasing the Central Ring speed reduces this problem, but puts more pressure on the extraction queues, where the *average* queue depths increase by over a factor of two. To truly balance the system, an investigation of the three-dimensional space formed by the speeds of the station bus, Central and Local Rings is required, which is left for future work.

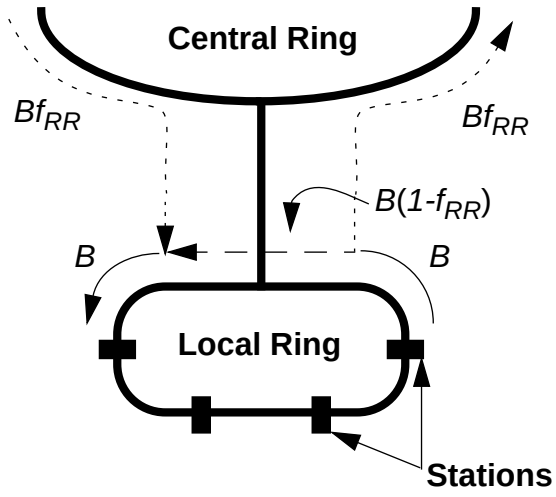

FIGURE 5.6: Bandwidth requirements of the Central Ring. Each station generates traffic with bandwidth *B*. A fraction Bf_{RR} splits off and goes up to the top level ring. In a ring topology, any traffic that returns to the sender must always use **all** of the links on any ring it traverses: what goes up must come down.The asymmetric data sizes of requests and responses is taken into account by the assumption that all stations behave the same: a given station sends requests on its output link, and gets responses on its input link, but it also gets requests from other stations on its input link, and sends responses out.

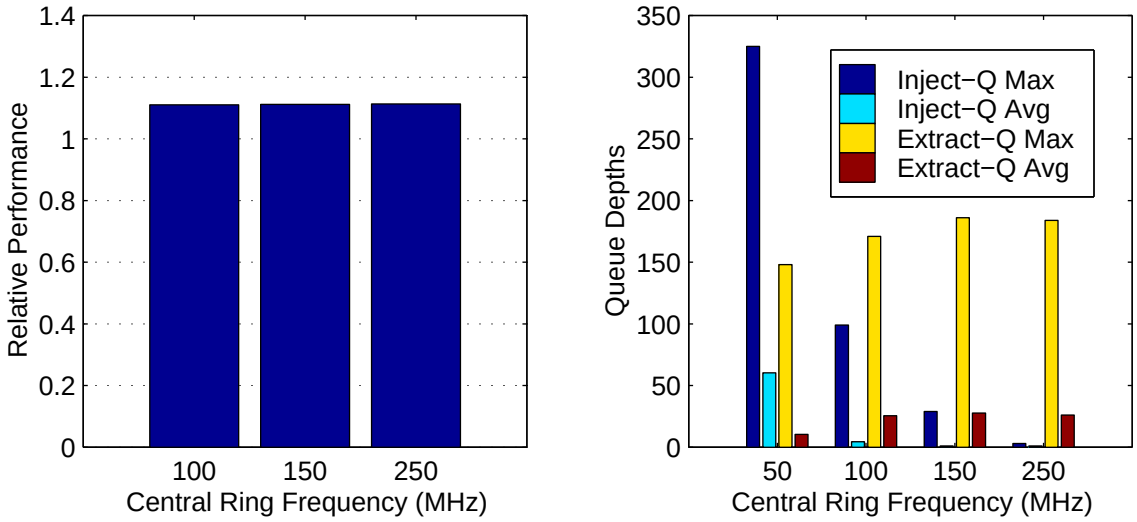

FIGURE 5.7: Effects of increasing Central Ring speed. The left graph shows the relative performance (parallel execution time) compared to the default 50 MHz Central Ring. The figure on the right shows the maximum and average queue depths for the FIFOs that inject onto and extract from the Central Ring.

5.5 Network Cache Performance

As mentioned in Chapter 2, the benefits of network-level caches are not universally agreed upon. The goal of this section is to investigate the performance of NUMAchine's Network Cache. We look at the effect of changing NC sizes, then the NC's level of associativity.

5.5.1 Network Cache Size

The goal of the NC is to reduce remote miss latency. It does this by providing a backup repository for cache lines which are ejected due to conflict or capacity misses in the per-processor caches. In NUMAchine the NC serves a dual role as the local portion of the distributed coherence directory, thus it can also enhance locality for processor coherence misses. So, for example, if a processor obtains write permission for a remote cache line, any further reads or writes

by other processors on the station generate only local traffic. This is only true as long as the line does not get ejected from the NC, and the line does not get accessed by some other processor from another station. Note that it is possible for the compiler and operating system to arrange data and threads to increase this locality. We have *not* modified the Splash2 programs in any way to take advantage of this, so the results presented in this section are conservative.

In order to do the size study, we would also like to have two boundary points for comparison: a zero-sized and infinite-sized NC cache. The zero-sized NC cache is intended to represent a system that has only local bus-snooping[4]. The infinite size is used to study the effect of removing all capacity and conflict misses, which also indicates where increased associativity might be of help (with conflict misses). The implementation of the infinite NC is simple since all that is necessary is for the cache tag lookup function in the NC to use all of the address bits instead of masking off the upper bits. The zero-sized NC is more difficult. To truly model a system with no network caching at all would involve modifying the coherence protocol, because the NC plays an integral role. To avoid this, we find a minimal set of changes which allows the NC to stay in the simulation, but lets it mimic the behaviour of a snooping-only system in most cases. We decided to keep the feature in the NC whereby a remote read is negatively acknowledged if a previous request for the same line is still waiting for a response (the combining case). This functionality would not normally be available in a snooping system, but the extra hardware required in the NIC to perform the same functionality is negligible—one register for each of the four possible outstanding processor requests—and to change this in the simulator would be difficult.

The second change necessary to model a zero-sized cache is to use an infinite directory. This may seem counterintuitive, but we do this to keep a full history of all cache lines that have been brought onto the station, so we can make decisions on whether the line may be available for snooping or not. We must force all received writebacks to continue on to the home memory, since there is no NC cache in which to store them. Another change has to do with shared lines in a global valid (GV) state. Such lines may or may not be in any local processor, since they can be ejected without notification to the NC. In an aggressive bus-snooping scheme, shared read requests can be satisfied by local *shared* in-cache copies from some other processor[5]. Thus we allow the read to a shared line to succeed if the line is GV *and* the line is

4. A system with only local snooping would probably use page replication and migration to enhance locality. Our purpose in this section is only to do self-comparison, not to compare our system against any others.

present in at least one processor's cache on the station. We determine the presence of the line by using a function that physically snoops into the on-station processor caches, but does so in zero time. If the line is not present, we send the request to the home memory and change the state in the NC directory to the not-in state (NS). Lines in the local valid (LV) state should ideally also be snoop hits under our aggressive policy, but this proves too complicated. A line can only be in the LV state if a dirty copy is written back to the NC, or another processor on the station does a shared intervention, so that in our normal coherence scheme both processors and the NC would have shared copies, with the NC being the owner of the line. But the write-back cannot be kept in the NC for the same reason as above.

For the local intervention case, the difficulty arises when deciding on ownership of the line after the intervention has finished. The line must be owned by some object on the station, since it is out-of-date with respect to the home memory, but there is no clear candidate. It turns out that if we try naively to implement our cache-snoop test on the LV line to determine if there is really a copy in one of the processors, then remote interventions can cause a race condition that leads to a coherence error. Our solution is to allow the NC to hit to LV lines, regardless of whether this would be legal under the snooping scheme or not. This gives the no-NC model a slight advantage, but after checking through the results we concluded that this was a minor effect[6].

The final change we made for the no-NC case is to change the directory lookup times to one cycle, to model a zero-overhead bus-snoop. (The NC still sits across the bus from the processors, so bus access time is modelled.)

The NC test sizes are chosen based on the total amount of processor cache which the NC is backing up. In our case, four processors with 64 KB of L2 cache each means that 256 KB of cache is necessary just to back up L2. It is still possible for a cache smaller than this size to have some effect. Firstly, a significant proportion of the lines in the L2 caches are for local or private memory, so the total amount of remote memory cached is less than 256 KB. Secondly,

5. This is normally not done because multiple processors may have a copy, so the bus needs some arbitration method to find out which one gets to respond.

6. The greatest percentage of such hits to LV for the no-NC runs is for BARNES at 8 processors. The *total* number of hits to LV lines (including legal and illegal) is just over 1% of the total requests to the NC. The number of lines that managed to stay in the LV state was greatly reduced in the no-NC case by our policy of forwarding all writebacks to the home memory.

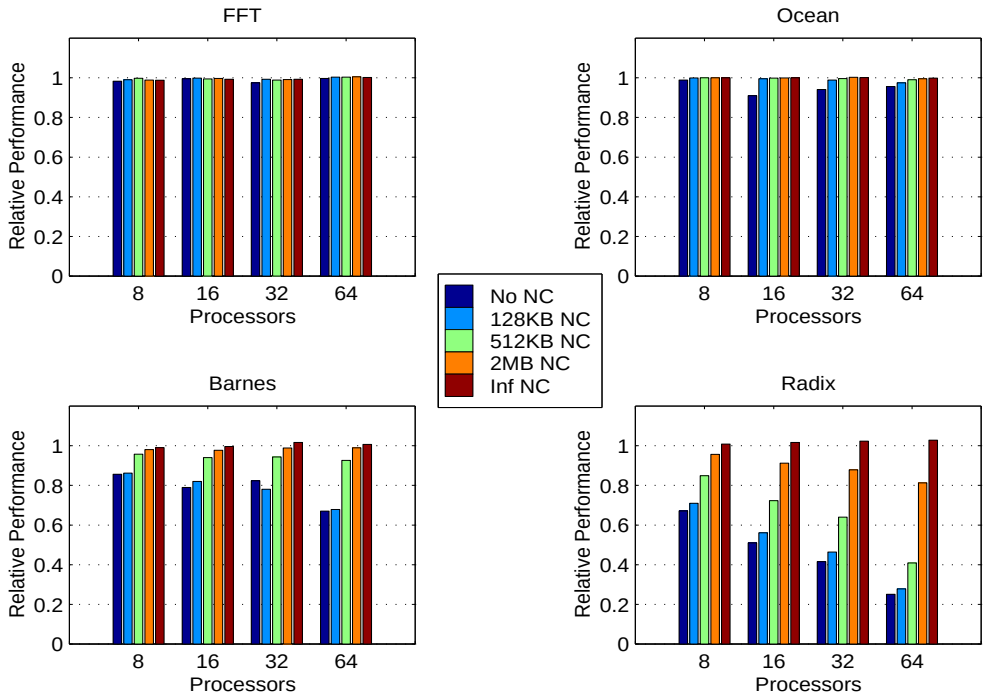

FIGURE 5.8: Effects of Network Cache size on performance. For a given number of processors, the execution time is normalized to that of our baseline case with an 8192 KB cache.

shared lines will have copies in multiple L2 caches, thus the NC cache storage used is less than the direct sum of the L2 set sizes. And finally, as long as the NC size is greater than 64 KB, a line may be kept in the NC which was ejected due to a capacity conflict at the L2 level.

We choose as our smallest size 128 KB, and work up in multiples of four until we reach 2048 KB. These, along with the zero-sized and infinite NC cache results, are normalized to our baseline case of an NC with 8192 KB, which is the size of the NC in the prototype. Results are shown in Figure 5.8. For FFT and Ocean, the size of the NC cache does not matter, but it does not make performance any better or worse than having no NC at all. For Radix and Barnes there is a marked improvement as the NC size is increased beyond the minimum. For Barnes a 512 KB cache is enough to provide nearly the same performance as if the cache were infinite.This is due to capacity requirements, since the output files show that the total footprint of data in the NC is around 700 KB. For Radix, the same footprint is 4.4 MB (per NC), which

explains why it takes the full 8 MB default NC cache to obtain effective infinite-cache performance.

The conclusion is that for certain programs the NC can definitely improve performance by helping with L2 capacity problems. In the next section we examine whether there is a conflict-miss problem by increasing the NC associativity.

5.5.2 Network Cache Associativity

We will use Radix for this comparison, because we have seen in the previous section that exhibits the largest response to changes in the NC size, and with such a large per-NC footprint it also has the best chance of showing conflict problems.

Adding associativity to the NC is not straightforward. The problem is that normally a least-recently used (LRU) way-selection algorithm is used for replacements in an associative cache. The reason LRU works is that good temporal locality normally means that old cache lines contain the least valuable information, so it is safe to eject them. In the case of the NC, the ‘information’ contained in a cache line is not only the data, but the coherence directory information as well. Since the NC is supposed to reduce both cache misses and coherence overhead, we must broaden our definition of information content.

Our algorithm is to assign priorities to certain cache line states that contain the most ‘expensive’ information from a coherence standpoint. Our choices, from highest (we do not want to eject) to lowest (we can afford to toss out) are:

- Locked Any-State - Actually we cannot throw these out. If all ways are locked, the algorithm must NACK the request.
- Local Valid - As described in previous sections, this state is expensive to reach, and indicates recent local sharing so we should keep it around.
- Local Invalid - Almost as useful as LV, but either there is no other processor that wants to share the line, or there has not been enough time for the sharing to occur.
- Global Valid - A globally shared copy, so it should be kept around if no other dirty states need the space.
- Notin State - A copy was requested in the near past, but NACKed. The only information here is that some processor may retry a request soon.

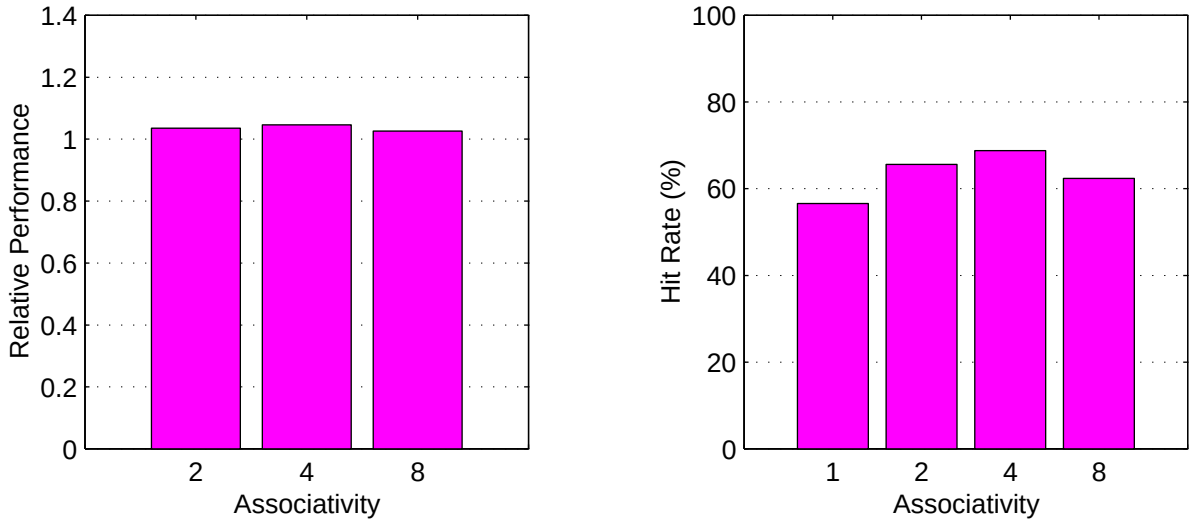

FIGURE 5.9: Effects of adding associativity to the Network Cache. The graph on the left shows the relative improvement in parallel execution time compared to the default 1-way (direct-mapped) cache. The improvement does not go above 5%. The graph on the right shows that there is some small improvement in the hit rate, but clearly not enough to increase performance. WIth 8-way associativity the size of each way becomes small enough that capacity misses start outweighing the benefits of associativity.

- Global Invalid - This contains almost no information at all, indicating only that no copies exist locally or have been requested in the near past. This is the prime candidate for ejection.

This algorithm is simple to realize in the simulator, but would be very expensive in hardware, so it must work very well to be justifiable. Supporting too many ways of associativity is also costly in terms of hardware, since each way needs comparison logic to be able to run in parallel. (The decoding could be done serially, but this destroys the performance benefit of associativity.) For our experiment we use a 512 KB NC size, because from the last section we know that this size does not tap all of the potential of the NC. We model 2-, 4- and 8-way associative NCs. A 4-way associative NC provides room for separate requests from each of the four local processors. More than 8-way associativity is unlikely to gain us much, and would be overly expensive to implement.

In Figure 5.9 we see the results for Radix on 8 processors. (We choose 8 because from Chapter 4 we know that an 8-processor system has the highest hit rates.) The improvement in

performance goes no higher than 5%, and the hit rates improve slightly up to 4-way associativity. This is not enough of a performance gain to justify adding the complexity of associativity to the NC.

5.6 Conclusion

In this chapter we have considered ways of increasing NUMAchine's performance. As a precursor to these investigations we showed that a first-hit page placement policy provides better performance than a round-robin policy. We then used this first-hit policy as the default for the rest of the results.

We first considered the coherence protocol overhead. We modelled a system without coherence overhead, where write accesses were allowed to succeed as long as the data was available, irrespective of its coherence state. We found that the difference in performance between this ideal version and our implemented protocol was minimal for all programs except Ocean, which had 180% and 400% improvements at 32 and 64 processors, respectively. We concluded from this that the coherence scheme is efficient and performs well in most cases, and that the coherence operations with the highest overheads occurred infrequently. More work needs to be done to determine whether Ocean's behaviour is intrinsic, or is due to some artifactual effects and could be fixed by tuning the code.

We considered briefly the relaxation of the sequential consistency model. We only looked at changes to the hardware, and did not consider changing the programming model. Easing the constraints placed upon the hardware implementation by sequential consistency had almost no effect on performance.

In Chapter 4 we saw that the Central Ring caused congestion in the system. We investigated the effect of speeding up the Central Ring. We found that the overall performance could be increased by 11% with an increase of the Central Ring frequency to 100 MHz, after which point the performance did not change. Investigation of the maximum and average queue depths indicated that the reason for the plateau in performance was that the bottleneck had moved from the Central Ring's injection queues to its extraction queues. We proposed that future studies are needed to investigate a full balancing of the station bus, Central and Local Rings to find the 'sweet-spot' that maximizes performance.

With regards to the Network Cache, we looked at the effects of its size and associativity on system performance. We concluded that while some programs are insensitive to NC size, others can show large performance improvements with an NC of even modest size. For an NC supporting P processors with caches of size C, an NC size of around $2PC$ displayed performance that approached the performance of a system with an NC of infinite size.

We also considered adding associativity to the NC. To do this we introduced a novel set-replacement algorithm based on the notion of the ‘importance of information’ represented by various coherence directory states. By reducing the likelihood of a cache conflict, increasing the associativity reduced the probability of finding a line locked by some other access. Our results showed that performance increased by at most 5%, which is not enough to justify the extra cost of adding associativity.

CHAPTER 6 *Conclusion*

6.1 Summary

The goal of this work has been to describe the design, and analyse the performance, of NUMAchine, a distributed shared memory (DSM) parallel processor. NUMAchine is intended as a proof-of-concept machine, showing that a DSM can be implemented at low-cost, while maintaining scalability up to a few hundred processors. It also serves a second role as a research platform for investigations into all aspects of multiprocessing; from the high end of the spectrum, encompassing compilers and operating systems, down to the lowest levels, such as hardware cache coherence protocols and system-area networks.

The hardware component of the NUMAchine project has taken over five years from initial high-level design studies to the completed, functional 48-processor prototype. During this time, the author was responsible for all aspects of the prototyping process, from architectural design and simulation studies, to board manufacture and debugging. The Mintsim architectural simulator, and the simulation studies which make use of it for pre- and post-prototype analysis are both contributions solely of the author.

We gained a wealth of experience from designing and building our own multiprocessor. Perhaps the most important lesson learned is that to be a good architect one needs to have experience with implementing hardware. High level architectural studies are very useful, but they are also a long way from real hardware. Ideas that seem simple at the top level can become a nightmare when it comes to implementation. Our packet re-assembly scheme, for example, was predicated on our choice of a slotted-ring protocol. One reason for choosing slotted rings was to simplify injection of packets onto the ring, the trade-off being added complexity on the receiving (extracting) end. It turned out there were numerous details that made the packet re-assembly controllers the most complicated FPDs to design and hardest to debug.

This added many cycles of latency which we did not take into account in our initial simulation studies, since we thought that this part of the system would be ‘easy’.

Our overall experience with design using FPDs was very positive. They allowed us to reach our performance goals, but also greatly enhanced the ability to quickly debug the system. We designed each FPD with debug pins which were pulled out to headers suitable for connection to a logic analyzer. By reprogramming the devices in-system, while the machine’s power was kept on, we could quickly re-run tests using different trace information for signals internal to the FPD. This enabled us to zero in on bugs in a matter of hours instead of days. Once the bug was found, we could have the fix installed in minutes. Without this facility, we would have had to spin many more revisions of the cards, which would have been impossible with our limited budget. (Our first card, the processor card, needed three spins, while all the other cards needed only two.)

On the other hand, FPDs do have drawbacks, two of the most significant being speed and logic density. Our targeted system-wide design speed of 50 MHz was extremely aggressive given FPD performance in the 1996-1997 time frame in which the chips were purchased. By necessity we developed a very high level of expertise in tuning FPGA and CPLD designs to achieve the desired clock rate. As for logic density, we were not able to fit our most complex controllers, such as the cache coherence directory controllers, into even the largest FPDs available at the time. The extreme case was the controller for the Network Cache (NC) coherence directory, which we managed to fit into four of the largest CPLDs after partitioning the design. This added complexity to both the design and verification. Post-fabrication issues arose when we needed to modify logic on FPDs with internal resource usage over 60%. Because the chips were soldered to the boards, pin placements were fixed. We found that changing logic with these fixed pins made it impossible in many cases for the fitting algorithms in the FPD CAD tools to find a logic assignment that would meet our timing constraints. In these cases we had to either find another way of fixing the logic, or, in the worst case, had to resort to hand-placement of internal logic resources.

Up until a few years ago, a project of this scale with a team this size would have been impossible. Sophisticated CAD (computer-aided design) tools[1] were crucial in allowing us to design and verify our boards all within a unified framework. The verification stage involved

1. We used the Cadence Logic Workbench tool suite, made available to the University of Toronto by a special university licensing agreement between Cadence and the Canadian Microelectronics Corporation (CMC).

running full board-level simulations using hardware-description language (HDL) models. The measure of success for this approach is that the first boards (processor cards) we received back from fabrication were booting up and running code within a week.

6.1.1 Architectural Simulator

The main tool used for architectural validation and analysis, and a major contribution of the thesis, is the Mintsim simulator. Mintsim allows detailed cycle-accurate system-level simulation of NUMAchine's architecture. Most simulators used for comparable architectural studies model at a high level, or else they run slowly and can only use 'toy' applications or trace files, which have unrealistic behaviours or timing. We also have the advantage that our simulated architecture exists as real hardware, allowing us to verify the simulation model, lending credence to the results.

Mintsim is highly flexible, allowing almost any architectural parameter to be varied, and the effect on performance to be easily measured. It is execution-driven, meaning that it uses real parallel applications, presenting the abstraction of 'virtual hardware'. This allows for very accurate modelling of timing and ordering, which is particularly important in the analysis of coherence protocols and networks. Although the simulator is very detailed, it is also fast. Typical slowdowns on the order of 1000 times allow programs with hardware run times of up to one hour to be simulated in a day.

We also described a strategy for increasing confidence in the simulation results. During initial simulator development, synthetic benchmarks with predictable results are used to check simulator output. After the hardware is available, the same measurements performed with the simulator can be duplicated in the hardware.

6.1.2 Architectural Results

We presented results in Chapter 4 which indicate that NUMAchine's ring-based network provides good levels of performance in the role of a DSM system interconnect. The bisection bandwidth of rings is not scalable, and our claims are only valid up to our chosen 64-processor limit. However, we believe that it will be many years before there is much of a market for systems larger than 256 processors, which we consider to be the upper limit for the category of

medium-scale multiprocessors. Indeed, the current ‘sweet-spot’ for commercial multiprocessing is dual- or quad-processor bus-based boxes, containing Intel or Alpha processors. These are becoming commodity items which are available in a typical user’s desktop machine, or as a small-scale server. The ability to scale up to huge numbers of processors is not an important criterion for commercial multiprocessors. In this regard, NUMAchine’s hierarchy of rings, with its simple point-to-point interconnection topology, allows for scalability with low incremental cost. In addition, the simplicity of rings allows them to be run at very high speeds, which both increases bandwidth and reduces latency.

One aspect of NUMAchine’s rings which we did not consider is reliability and fault tolerance. These are crucial given the current market trend towards raising the level of reliability of multiprocessors to that of mainframes. While approaches such as dual rings can address these issues, their cost and performance impact were not considered here.

A significant reason for NUMAchine’s good performance is a novel hardware cache coherence protocol that makes use of the ring topology and its ordering properties to achieve low levels of coherence overhead. This is critical in keeping performance levels high under the shared-memory paradigm. The coherence protocol uses the writeback/invalidate scheme as its basic structure, upon which it builds a dual-level coherence directory which matches the ring hierarchy. No explicit acknowledgments are necessary for invalidations in the protocol, since the rings maintain relative ordering of broadcasts. This ordering property also provides for a natural implementation of a sequentially consistent memory model. Sequential consistency provides the simplest and most intuitive model of memory from a programmer’s perspective, which makes NUMAchine very programmer-friendly. In anticipation of the argument that sequential consistency has inherently lower performance than weaker consistency models, we point to recent results in the literature indicating that the combination of sequential consistency and modern microprocessor architectures can achieve the same levels of performance as weaker models [Gniady 1999].

In Chapter 5 results were presented which showed that a network-level cache can help alleviate the high latency for remote requests in a NUMA architecture. For certain types of programs the Network Cache was shown to contribute to a significant improvement in execution times, while for the remainder of the programs the NC was performance-neutral. The size of the NC plays a role in its performance, and we found that a size about a factor of two larger than the sum of the processors’ caches worked well. Sizes beyond this did not do much to increase performance. In the same chapter we presented a novel set-replacement algorithm

used to implement associativity in the Network Cache, based on a ranking of the information content of various coherence states. We showed that increasing the associativity to 2-way or 4-way increased performance by no more than 5%, and concluded that the added complexity of supporting associativity was not justified.

We expect that the NUMAchine prototype's performance still has considerable room for improvement. As mentioned in Chapter 4, the backoff scheme can be improved upon. Other enhancements to the NC and coherence protocol, such as dynamic protocol selection or the use of multicasting to push data instead of our default pull model, are part of ongoing research. Finally, all of the results presented herein use unmodified Splash2 programs. NUMAchine is a clustered architecture due to its bus-based stations. Three obvious ways of tuning the programs to increase performance would be:

- Allocate data structures and threads to take advantage of low-latency intra-cluster (on-station) access times.
- Make the OS and compilers more NC-aware, in order to make more efficient and intelligent use of the NC cache resource.
- Add support in the operating system for page migration and replication to provide a complementary mechanism to reduce remote latency for access patterns which the NC cannot handle.

NUMAchine is a working prototype. Figure 6.1 shows a partially-assembled prototype with 24 processors. With the addition of the final ring, the full 48-processor system will be operational. We are able to boot up the operating system, and run parallel applications natively on NUMAchine. Now that the hardware is finished, the next phase of research will focus on the OS, support for parallel file systems, and architecture-aware, automatically parallelizing compilers.

6.2 Future Work

The field of parallel systems and programming is roughly 30 years old, but is still in a fairly immature state of development. In the following paragraphs we suggest some ways in which the results of this dissertation could be extended, and possibilities for other related work.

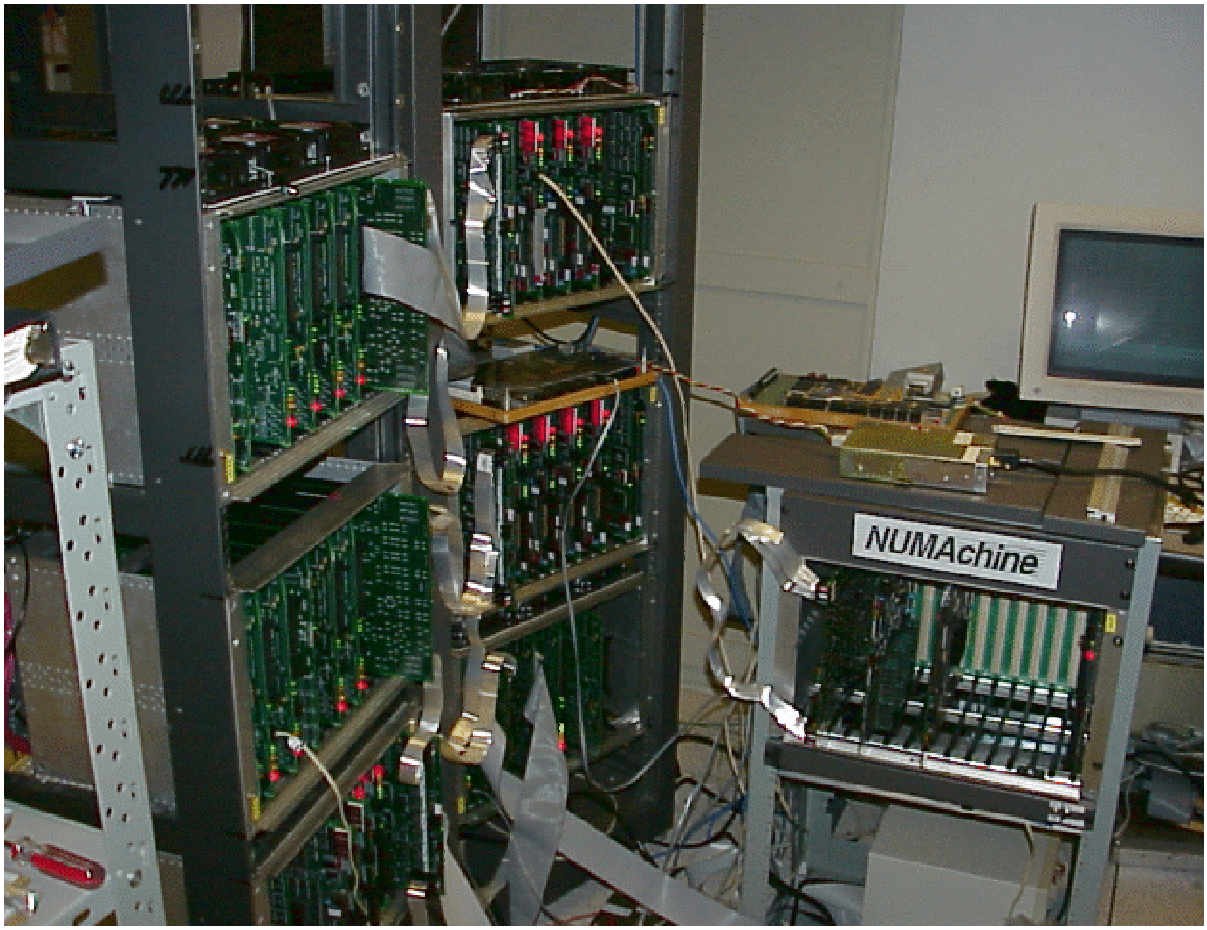

FIGURE 6.1: NUMAchine with 24 processors. The two tall vertical racks contain six stations with four processors in each. The bottom two stations in each rack are connected together by the shiny ring cables to form a 16-processor Local Ring. Another Local Ring sits at the top of the racks. (The two top stations have yet to be installed.) The third Local Ring sits behind the two shown, and faces the other direction. The single-station mini-NUMAchine shown on the right is used for debugging.

In Chapter 4 we found that the exponential backoff approach to handling cache lines locked by coherence actions suffered from liveness and fairness problems, and could lead to very large latencies. A better scheme was proposed, but not analysed. It would be interesting to find out whether the performance of the newer approach was enough to justify the extra cost and complexity.

Advances in FPD architecture present some intriguing possibilities for design of systems such as NUMAchine. In the next few years programmable devices will be available with mixed FPGA/CPLD structures on the same chip. These devices will contain hundreds of millions of gates, as well as megabits of embedded memory. With such resources it would be possible to build an entire coherence engine (including the SRAM directory) inside a single chip, allowing for significant reductions in latency.

As mentioned in Chapter 5, benchmarks are a particularly weak point. Benchmark suites such as Splash2 help to model the performance of scientific applications. However, the huge boom in commercial use of shared-memory multiprocessors means that a majority of compute cycles are now devoted to three major new classes of workloads: *on-line transaction processing* (OLTP), *decision-support systems* (DSS) and Web servers. OLTP typically involves many small read/write accesses to a large database. DSS workloads analyse databases in order to detect business trends, which involves long complex queries accessing the entire database. And Web servers are usually used either for presenting information or searching. All three have very different characteristics, which are quite different from scientific applications. Only recently have researchers started to analyse these workloads in multiprocessing environments [Barroso 1998]. The results indicate that communication-to-computation ratios are much higher for these applications, particularly for OLTP. Sharing patterns are dynamic and non-repetitive, with a comparatively high rate of true sharing, leading to much more time spent servicing coherence misses in the caches. Analyses such as the ones presented in this thesis will need to be redone using more recent benchmark suites such as TPC (Transaction Processing Council) [TPC 1999].

Another area which needs more work is the incorporation of mainframe-class Reliability/Availability/Serviceability (RAS) features as a basic requirement in multiprocessor system modelling. RAS is necessary for such machines to have broad commercial appeal, but it affects both performance and cost. Computer architects must include these aspects in their analyses right from the start; they cannot be treated as independent issues to be designed in later or added on.

This work could benefit greatly from recent advances in the state of the art of simulation. When we started, there existed no commonly accepted framework for doing multiprocessor simulation. We estimate that just providing the infrastructure for our architectural simulator to hook up with MINT took roughly six man-months of work. Tools such as SimOS [Rosenblum

1997], while powerful, are not flexible enough to allow for the kind of detailed back-end architectural simulations discussed in this thesis. Were a simulation environment with the flexibility of MINT and the power of SimOS available, we could have investigated the performance of the architecture including the effects of the operating system and multiprogrammed workloads, which was not feasible with the tools we developed. There are also performance issues regarding cycle-accurate simulation of such large systems. A fitting approach is to parallelize the simulator itself. Research is ongoing in this area but results as yet are poor. (See for example [Carothers 1999]).

And finally, future work on multiprocessing systems will benefit from research into formal verification. Both the cache coherence protocol and the prototype hardware were verified by simulation. While these proved sufficient for our needs, they were only just barely so. With the increasing complexity of modern microprocessors and advanced optimized coherence protocols, formal verification will provide the only means of designing parallel systems in a reasonable amount of time.

APPENDIX A *Notes on the Simulator*

The simulator, as described in Chapter 3, consists of the MINT front-end[Veenstra 1993], and the NUMAchine simulator back-end. The version of MINT used was 2.6, with some minor modifications, which for the sake of completeness will be described below. A brief description of the NUMAchine simulator and its parameter files are also given.

A.1 MINT Modifications

The most fundamental change to MINT was switching the basic time variable from double-precision floating-point variables (`double`)to signed 64-bit longs (`long long`) for both the Sparc and SGI platforms. This allowed the same range of simulation times (since a `double` is stored internally as 64 bits on both machines) but made the simulator considerably faster. Note that compilers which could handle the `long long` type were not universally available when version 2.6 was first distributed. We used the GCC 2.7.2 compiler to compile both MINT and the NUMAchine simulator.

We also modified MINT to pass along to the back end more information as to the source of a given memory reference. The MINT memory space consists of four sections:

- Stack - the standard program stack as used by local variables and for parameter passing,
- Heap - a private per-thread area used for memory obtainedthrough `malloc(),`
- Share - a global memory section, accessible by all threads and allocated using the `us_malloc()` call, containing the shared data structures (e.g. arrays and trees) that are the targets for most of the work done by a parallel application,
- Data/BSS - the initialized (Data) and uninitialized (BSS) static data sections that are allocated at program load time.

This allowed the back end to accumulate statistics for each category, which was useful as the reference patterns were different for each type.

In order to do future studies on prefetching and updating, we added the capability to handle both of these types of references at MINT's lowest level. To get a program to generate such references, the source code would have to be modified by hand to insert a dummy 'sys-

tem call' with a virtual address, which MINT recognizes and passes along to the newly added back-end routines: `sim_update(), sim_prefetch_shr()` and `sim_prefetch_exc()`.

A 'feature' of the original MINT 2.6 version was that the results obtained when running the Sun and SGI versions of the code were very slightly different (less than 1%). While not significant, the very fact that the same binary application code running through supposedly identical virtual machines inside MINT on the two architectures did not match was worrisome. After much debugging, it was found that the section of memory containing the environment variables passed to the simulator were of different sizes on the two platforms. This caused the alignment of MINT's internal virtual memory space for the parallel application to differ. The resulting slight change in cache access and page fault patterns gave rise to the anomaly. The MINT code was modified so that the starting point for its virtual memory space was aligned to a large page size. After this change the results on the two platforms were identical.

There were also a few miscellaneous additions and fixes to the following files:

- `exec.c` - make the `PUNMAP()` macro the exact inverse of `PMAP()`, which was probably a bug,
- `icode.h` - fixed a bug in the address space decoding logic which would incorrectly flag addresses as 'invalid',
- `subst.c` - added the routines `mint_getenv(), mint_putenv()` since as mentioned above the default environment variable handling caused subtle anomalies on different platforms. Also added were the previously unsupported operating system calls `shmemalign()` and `getpagesize()`.

Some of these changes may also be present in later versions of MINT. However later versions did not add any functionality we found necessary for our work, so we stuck with version 2.6. All of the modified code for MINT is available by contacting the author.

By default, MINT's time scale is arbitrary; there is no assumption of any kind of units for MINT's internal clock. Moreover, unless told otherwise it assumes that every machine instruction takes one clock cycle to execute. While this is true for the majority of the RISC R4400 instructions, it is not true for all, and in particular arithmetic operations such as multiplication and division take on the order of tens of cycles for both fixed- and floating-point versions. To present a more realistic model MINT allows the number of cycles per instruction (also known as CPI) for all the assembler opcodes to be specified in a text file using its '-c' option. An example of such an *opcodes* file is shown in Listing A.1. The number in the second column is the number of clock cycles for the instruction. Since the fundamental time unit in the simulator is 1 ns, the sample parameter file corresponds to a 1000 MHz processor. This file can be scaled to a processor running at F MHz by multiplying each value by ROUNDUP(1000/F). These timings were not supplied with MINT, but taken from the R4400 reference manual [Heinrich 1994].

```
regimm 1  jal 1     beq 1     bne1      blez 1    bgtz 1
addi 1    addiu 1   slti 1    sltiu 1   andi 1    ori 1
xori 1    lui 1     cop0 1    cop1 1    cop2 1    cop3 1
beql 1    bnel 1    blezl 1   bgtzl 1   lb 1      lh 1
lwl 1     lw 1      lbu 1     lhu 1     lwr 1     sb 1
sh 1      swl 1     sw 1      swr 1     cache 1   ll 1
lwc1 1    lwc2 1    lwc3 1    ldc1 1    ldc2 1    ldc3 1
sc 1      swc1 1    swc2 1    swc3 1    sdc1 1    sdc2 1
sdc3 1    sll 1     srl 1     sra 1     sllv 1    srlv 1
srav 1    jr 1      jalr 1    syscall 1 break 1   sync 1
mfhi 1    mthi 1    mflo 1    mtlo 1    mult 7    multu 7
div 32    divu 32   add 1     addu 1    sub 1     subu 1
and 1     or 1      xor 1     nor 1     slt 1     sltu 1
tge 1     tgeu 1    tlt 1     tltu 1    teq 1     tne 1
bltz 1    bgez 1    bltzl 1   bgezl 1   tgei 1    tgeiu 1
tlti 1    tltiu 1   teqi 1    tnei 1    bltzal 1  bgezal 1
bltzall 1 bgezall 1 add.s 4   sub.s 4   mul.s 7   div.s 23
sqrt.s 54 abs.s 2   mov.s 1   neg.s 2   round.w.s 4
trunc.w.s 4         ceil.w.s 4          floor.w.s 4
cvt.d.s 2 cvt.w.s 4 c.f.s 1   c.un.s 1  c.eq.s 1  c.ueq.s 1
c.olt.s 1           c.ult.s 1           c.ole.s 1
c.ule.s 1           c.sf.s 1            c.ngle.s 1
c.seq.s 1           c.ngl.s 1           c.lt.s 1
c.nge.s 1           c.le.s 1            c.ngt.s 1
add.d 4   sub.d 4   mul.d 8   div.d 36  sqrt.d 112 abs.d 2
mov.d 1   neg.d 2   round.w.d 4         trunc.w.d 4
ceil.w.d 4          floor.w.d 4         cvt.s.d 4
cvt.w.d 4 c.f.d 1   c.un.d 1  c.eq.d 1  c.ueq.d 1
c.olt.d 1 c.ult.d 1 c.ole.d 1 c.ule.d 1 c.sf.d 1
c.ngle.d 1          c.seq.d 1 c.ngl.d 1 c.lt.d 1
c.nge.d 1 c.le.d 1  c.ngt.d 1 cvt.s.w 6 cvt.d.w 5 mfc0 1
mfc1 3    mfc2 1    mfc3 1    mtc0 1    mtc1 3    mtc2 1
mtc3 1    cfc0 1    cfc1 2    cfc2 1    cfc3 1    ctc0 1
ctc1 3    ctc2 1    ctc3 1    bc0f 1    bc0t 1    bc0fl 1
bc0tl 1   bc1f 1    bc1t 1    bc1fl 1   bc1tl 1   bc2f 1
bc2t 1    bc2fl 1   bc2tl 1   bc3f 1    bc3t 1    bc3fl 1
bc3tl 1   cop_reserved 1      cop_invalid 1
terminate 1         b 1       li 1      move 1    nop 1
```

LISTING A.1: Mint opcodes file for a 1000 MHz processor.

A.2 Notes on the NUMAchine Simulator (Mintsim)

Mintsim is described in Chapter 3. Here we provide an example of the simulator command file, to give an idea as to Mintsim's flexibility.

While the basic CPU model is fixed by the use of MINT, and features such as the ring-based network and its associated cache coherence protocol are fixed due to software development time constraints, just about any other parameter in the model is variable. A simulation run is controlled by a text parameter file, an example of which is shown in Listing A.2. The

values shown are the defaults for a 32-processor implementation of the NUMAchine prototype hardware.

```
#
# NUMAchine simulator command file
# ================================
#
# NOTE: Where options are commented out, the value
# indicates the default
#
# General Simulator
# -----------------
#  Geometry is: Ring0s/Ring1:Stations/Ring0:CPUs/Station,
#  so product is the number of cpus in the system

set sim geometry 2:4:4          # 32-CPU system
# set sim barrier_type ideal
#   Types are: “ideal”, “simple”/”tree” (Hardware)
#   “soft”/”softtree” (Software)
# set sim lock_type spin
#   Types are: “spin”, “ideal”
# set sim coherence 1         # Use cache coherence or not
# set sim global_line_size 0
#   If Mem and NC use different line size to L2
# set sim use_nc numa
#   Whether NC is used or not, and what type
#   Others: none, inni (part of Network Int)
# set sim relaxedconsistency 0
#   Default is sequential consistency
# set sim page_type roundrobin
#   Others: firsthit, fixed
# set sim pagemap_file ??
#   For fixed page_type, gives page mappings
# set sim snoop 1
#   If no NC, then use snooping
# set sim perf_pref 1
#   Turn on perfect prefetching

#
# Network Cache
# -------------
# set netcache enhanced 0       # Turn on NC enhancements
set netcache size 8192          # In KB
# set netcache assoc 1          # N-way associative
set netcache read_time 200      # in ns
set netcache write_time 200     # in ns
set netcache tag_time 80        # in ns
# set netcache fifo_delay 30    # in ns
set netcache fifo_width 8       # in bytes
set netcache inq_size 256
set netcache outq_size 256
# set netcache inq_ovfl 0.75
# set netcache outq_ovfl 0.75

#
# Ring Interface
```

```
# --------------
# set ringint freq 50e6         # In Hz
set ringint width 8             # In bytes
# set ringint parallel_rings 1
#   For counter-rotating rings choose 2
# set ringint use_freed 0
#   Use a slot just freed by removal?
# set ringint inq_size 256
# set ringint inq_ovfl 0.75
# set ringint outq_size 256
# set ringint outq_ovfl 0.75
# set ringint sinkq_size 256
# set ringint sinkq_ovfl 0.75
# set ringint nsinkq_size 256
# set ringint nsinkq_ovfl 0.75
# set ringint max_wb_cnt 256
# set ringint ttl_tickets 256
#   How many outstanding requests in SRAM?
# set ringint fifo_wdelay 30

#
# Ring Ring Interface  see Ringint for descriptions
# -------------------
# set ringring freq 50e6
# set ringring use_freed 0
# set ringring upq_size 256
# set ringring upq_ovfl 0.75
# set ringring downq_size 256
# set ringring downq_ovfl 0.75

#
# Memory
# ------
set memory read_time 200        # in ns
set memory write_time 200       # in ns
set memory tag_time 80          # in ns
# set memory fifo_delay 30      # in ns
set memory fifo_width 8         # in bytes
# set memory inq_size 64        # depth, width given above
# set memory outq_size 64
# set memory inq_ovfl 0.75
# set memory outq_ovfl 0.75

#
# Processor
# ---------
# set proc freq 150e6   # in Hz
# set proc splitL2 0    # use split/unified L2 cache (1/0)
# set proc L1_Icache_size 16      # in Kbytes
# set proc L1_Icache_linesize 32  # in bytes
# set proc L1_Icache_assoc 1
# set proc L1_Dcache_size 16
# set proc L1_Dcache_linesize 32
# set proc L1_Dcache_assoc 1
```

```
# set proc L2_Icache_size 1024
# set proc L2_Icache_linesize 64
# set proc L2_Icache_assoc 1
set proc L2_Dcache_size 1024
set proc L2_Dcache_linesize 128
set proc L2_Dcache_assoc 1

#
# External Agent
# --------------
# set extagent si_freq 75e6
# set extagent max_retry 64
#   How many retries per request before error?
set extagent fifo_width 8
# set extagent fifo_delay 30
# set extagent inq_size 64
# set extagent outq_size 64
# set extagent inq_ovfl 0.75
# set extagent outq_ovfl 0.75

#
# Bus
# ---
# set bus freq 50e6
set bus width 8               # in bytes
set bus arb_latency 4         # in bus clocks (default 0)

# Debugging
# ---------
#
# On=1/Off=0. Can set individually by class, or for class
# 'sim', which turns on all classes. (Beware, the latter
# is A LOT of trace info.)
#
# set memory trace 1
# set bus trace 1
# set netcache trace 1
# set ringint trace 1
# set ringring trace 1
# set cache trace 1
# set extagent trace 1
# set proc trace 1
# set sim trace 1    # This is equivalent to the -t flag
#
# This is the most useful debugging feature. Trace all
# usage across the
# simulator to a particular cache block. Address will be
# rounded to cache block size

# set sim trace_addr 0x12345678

#
# Running the simulator
# ---------------------
#
#  Build instantiates objects and connects them. Run
```

```
#  starts the simulation. Set the type of reporting before
#  doing run.

# report     # Do “fullreport” for a per-instance report.
             # The default “report” is a summary, giving
             # averages and standard deviations
build
run
```

LISTING A.2: Default simulator parameter file for a 32-processor system.

The output from Mintsim is also a text file, containing reports from each of the simulator objects (e.g. busses, caches, etc.) indicating usage, latencies, statistics, etc. The output can be generated in either a detailed per-object form, or a summary form. The summary report shows averages over all of the particular objects in a class. The simulator output file is too large to be shown here.

References

Abandah, G. A. and E. S. Davidson. **1998**. “Characterizing Distributed Shared Memory Performance: A Case Study of the Convex SPP1000”. *IEEE Trans. on Parallel and Distributed Systems* Vol. 9 No.2, February. Pages 206-216.

Adve, S. V., V.S. Adve, M.D. Hill and M.K. Vernon. **1991**. “Comparison of Hardware and Software Coherence Schemes”. *Proc. 18th Annual Int’l Symp. on Computer Architecture (ISCA’91).* Pages 298-308.

Adve, S. V., and K. Gharachorloo. **1996**. “Shared Memory Consistency Models: A Tutorial”. *IEEE Computer* Vol. 29, No. 12, December. Pages 66-76.

Agarwal, A., R. Simoni, J. Hennessy and M. Horowitz. **1988**. “An Evaluation of Directory Schemes for Cache Coherence”. *Proc. 15th Annual Int’l Symp. on Computer Architecture (ISCA’88).* Pages 353-362.

Agerwala, T., J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias and M. Snir. **1995**. “SP2 System Architecture”. *IBM Systems Journal* Vol. 34, No. 2. Pages 152-162.

Aichinger, B. P. **1992**. “Futurebus+ as an I/O Bus: Profile B”. *Proc. 19th Annual Int’l Symp. on Computer Architecture (ISCA’92)*. Pages 300-307.

Amdahl, G. M. **1967**. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. *AFIPS 1967 Spring Joint Computer Conference.* Vol. 40. Pages 483-485.

Anderson, T. E., D. E. Culler and D. A. Patterson. **1995**. “A Case for NOW (Network of Workstations)”. *IEEE Micro* Vol. 15, No. 1, February. Pages 54-64.

Archibald, J., and J-L. Baer. **1986**. “Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation”. *ACM Trans. Comput. Syst.* Vol. 4, No. 4, November. Pages 273-298.

Archibald, J. **1988**. “A Cache Coherence Approach for Large Multiprocessor Systems”. *Proc. of the Int’l Conference on Supercomputing*. Pages 337-345.

Baldwin, R. **1993**. “A Cache Coherence Scheme Suitable for Massively Parallel Processors”. *Proc. of the Int’l Conference on Supercomputing ‘93*. Pages 730-739.

Barroso, L. A., K. Gharachorloo and E. Bugnion. **1998**. “Memory System Characterization of Commercial Workloads”. *Proc. 25th Int’l Symp. on Computer Architecture (ISCA’98).*

Becker, D. J., T. Sterling, D. Savaraese, J. E. Dorband, U. A Ranawak and C. V. Packer. **1995**. "Beowulf: A Parallel Workstation for Scientific Computation". *Proc. Int'l Conference on Parallel Processing (ICPP'95)*.

Bilardi, G., K.T. Herley, A. Pietracaprina, G. Pucci and P. Spirakis. **1996**. "BSP vs LogP". *Proc. 8th Annual ACM Symp. on Parallel Algorithms and Arch.* Pages 25-32.

Blumrich, M. A., R. D. Alpert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi and R. A. Shillner. **1998**. "Design Choices in the SHRIMP System: An Empirical Study". *Proc. 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*.

Boden, N. J., D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W.-K. Su. **1995**. "Myrinet: A Gigabit-per-Second Local Area Network". *IEEE Micro* Vol. 15, No. 1, February. Pages 29-36.

Bokhari, S. H., and D. J. Mavriplis. **1998**. *The Tera Multithreaded Architecture and Unstructured Meshes*. NASA/CR-1998-208953, December.

Brown, S., N. Manjikian, Z. Vranesic, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, Z. Zilic and S. Srbljic. **1996**. "Experience in Designing a Large-scale Multiprocessor using Field-Programmable Devices and Advanced CAD Tools". *Proc. of the 33rd IEEE Design Automation Conference (DAC'96).* Pages 427-432.

Burd, T. **1999**. *General Microprocessor Info*. http://infopad.eecs.berkeley.edu/CIC/summary/local.

Carothers, C. D., K. S. Perumalla and R. S. Fujimoto. **1999**. "Efficient Optimistic Parallel Simulations using Reverse Computation". *Proc. 13th Workshop on Parallel and Distributed Simulation*.

Cox, A. L., and W. Zwaenepoel. **1992**. "Lazy Release Consistency for Software Distributed Shared Memory". *Proc. 19th Int'l Ann'l Symp. on Computer Architecture (ISCA'92)*. Pages 13-21.

Chaiken, D., J. Kubiatowicz and A. Agarwal. **1991**. "LimitLESS Directories: A Scalable Cache Coherence Scheme". *Proc. 4th Int'l Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS-IV).* Pages 224-234.

Charlesworth, A. **1998**. "Starfire: Extending the SMP Envelope". *IEEE Micro* Vol. 18, No. 1, January/February.

Choi, L., and P-C. Yew. **1996**. "Compiler and Hardware Support for Cache Coherence in Large-scale Multiprocessors: Design Considerations and Performance Study". *Proc. 23rd Annual Int'l Symp. on Computer Architecture (ISCA'96)*. Pages 283-294.

Coplien, J. O. **1993**. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA.

Cormen, T. H., C.E. Leiserson and R.L. Rivest. **1989**. *Introduction to Algorithms*. McGraw-Hill, New York, NY.

Culler, D.E., R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken. **1993**. "LogP: Towards a Realistic Model of Parallel Computation". *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.* Pages 1-12.

Culler, D. E., J. P. Singh and A. Gupta. **1999**. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, San Francisco, CA.

Davis, H., S. R. Goldschmidt and J. Hennessy. **1990**. *Tango: A Multiprocessor Simulation and Tracing System*. Tech. report #CSL-TR-90-439, Stanford University.

Eggers, S., J. Emer, H. Levy, J. Lo, R. Stamm and D. Tullsen. **1997**. "Simultaneous Multithreading: A Platform for Next-generation Processors". *IEEE Micro* Vol. 17, No. 5, September/October. Pages 12-19.

Erlichson, A., B.A. Nayfeh, J.P. Singh and Olukotun. **1994**. *The Benefits of Clustering in Shared Address Space Multiprocessors: An Applications-Driven Investigation*. Tech. report #CSL-TR-94-632, Stanford University.

Falsafi, B. and D.A. Wood. **1997**. "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA". *Proc. 24th Int'l Symp. on Computer Architecture (ISCA'97).* Pages 229-240.

Farkas, K., Z. Vranesic and M. Stumm. **1992**. "Cache Consistency in Hierarchical Ring-Based Multiprocessors". *Proc. Supercomputing '92*, November. Pages 348-357.

Fillo, M., and R. B. Gillett. **1997**. "Architecture and Implementation of MEMORY CHANNEL2". *Digital Technical Journal* Vol. 9, No. 1. Pages 27-41.

Flynn, M. J. **1972**. "Some Computer Organizations and Their Effectiveness". *IEEE Transactions on Computing*, Vol. C, No. 21, September. Pages 948-960.

Gamsa, B., O. Krieger, J. Appavoo and M. Stumm. **1999**. "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System". *Proc. 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*. Pages 87-100.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. **1994**. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA.

Gharachorloo, K. **1995**. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, also available as tech. report #CSL-TR-95-685, Stanford University.

Gniady, C., B. Falsafi and T. N. Vijaykumar. **1999**. "Is SC+ILP=RC?" *Proc. 26th Int'l Symp. on Computer Architecture (ISCA'99)*. Pages 162-171.

Goodman, J., A. G. Greenberg, N. Madras and P. March. **1988**. "Stability of Binary Exponential Backoff". *Journal of the ACM* Vol. 35, No. 3. Pages 579-602.

Goodman, J. **1991**. *Cache Consistency and Sequential Consistency.* Tech. report #CS-TR-91-1006, University of Wisconsin-Madison, Comp. Science Dept, February.

Grbic, A. **1996**. *Hierarchical Directory Controllers in the NUMAchine Multiprocessor.* M.A.Sc Thesis, University of Toronto. http://www.eecg.toronto.edu/parallel/theses/grbic.pdf.

Grbic, A., S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic and Z. Zilic. **1998**. "Design and Implementation of the NUMAchine Multiprocessor". *Proc. of the 35th IEEE Design Automation Conference (DAC'98)*, June. Pages 66-69.

Gropp, W., E. Lusk and A. Skjellum. **1994**. *Using MPI*. MIT Press, Cambridge, MA.

Gupta, A., and W.-D. Weber. **1992**. “Cache Invalidation Patterns in Shared-Memory Multiprocessors”. *IEEE Transactions on Computers* Vol. 41, No. 7. Pages 794-810.

Hailpern, B.T., and S. S. Owicki. **1980**. *Verifying Network Procotols Using Temporal Logic*. Tech. report #CSL-TR-80-192, Stanford University.

Hamacher, V. C., and H. Jiang. **1997**. “Performance and Configuration of Hierarchical Ring Networks for Multiprocessors”. *Proc. Int’l Conference on Parallel Processing (ICPP’97)*. Pages 257-265.

Hamacher, V. C., Z. G. Vranesic and S. G. Zaky. **1996**. *Computer Organization*. Fourth edition. McGraw-Hill, New York, NY.

Heinlein, J., K. Gharachorloo, S. Dresser and A. Gupta. **1994**. “Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor”. *Proc. 6th Int’l Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS’94).* Pages 38-50.

Heinrich, J. **1994**. *MIPS R4000 Microprocessor User’s Manual*. Second edition. MIPS Technologies, Mountain View, CA.

Hill, M. D. **1998**. “Multiprocessors Should Support Simple Memory Consistency Models”. *IEEE Computer* Vol. 31, No. 8, August. Pages 28-34.

Integrated Device Technologies. **1994**. *Specialized Memories & Modules Data Book*. IDT, Santa Clara, CA.

Krieger, O. and M. Stumm. **1997**. “HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions”. *ACM Trans. on Computers*. Vol. 15, No. 3, August. Pages 286-321.

Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K.Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy. **1994**. “The Stanford FLASH multiprocessor”. *Proc. of the 21st Int’l Symp. on Computer Architecture (ISCA’94).* Pages 302-313.

Lamport, L. **1979**. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. *IEEE Trans. on Computers*. Vol. C-28, No. 9, September. Pages 690-691

Laudon, J., and D. Lenoski. **1997**. “The SGI Origin: A ccNUMA Highly Scalable Server”. *Proc. 24th Int’l Symp. on Computer Architecture (ISCA’97)*. Pages 241-251.

Lemieux, G. **1996**. *Hardware Performance Monitoring in Multiprocessors*. M.A.Sc. Thesis, University of Toronto. http://www.eecg.toronto.edu/parallel/theses/lemieux.pdf.

Lenoski, D. E. **1992a**. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* Ph.D. Thesis, Stanford University.

Lenoski, D. E., J. Laudon, T. Joe, D. Nakahira, L Stevens, A. Gupta and J. Hennessy. **1992b**. “The DASH Prototype: Implementation and Performance”. *Proc. 19th Int’l Symp. on Computer Architecture (ISCA’92)*. Pages 92-103.

Lenoski, D. E., and W.-D. Weber. **1995**. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufman, San Mateo, CA..

Li, K., and P. Hudak. **1989**. "Memory Coherence in Shared Virtual Memory Systems". *ACM Trans. on Computer Systems* Vol. 7, No. 4. Pages 321-359.

Loveless, K. **1996**. *The Implementation of Flexible Interconnect in the NUMAchine Multiprocessor.* M.A.Sc. Thesis, University of Toronto. http://www.eecg.toronto.edu/parallel/theses/loveless.pdf.

MIPS Technologies. **1996**. *MIPS R10000 Microprocessor User's Manual*. Version 2.0. MIPS Technologies, Mountain View, CA.

Moga, A., and M. Dubois. **1998**. "The Effectiveness of SRAM Network Caches in Clustered DSMs". *Proc. 4th Int'l Symp. on High-Performance Comp. Arch. (HPCA'98).*

Moore, G. E. **1975**. "Progress in Digital Integrated Electronics". *Proc. IEEE Digital Integrated Electronic Device Meeting*. Page 11.

Papamarcos, M., and J. Patel. **1984**. "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories". *Proc. 11th Annual Int'l Symposium on Computer Architecture (ISCA'84).* Pages 348-354.

Park, S. **1996**. *Computer Assisited Analysis of Multiprocessor Memory Systems*. Tech. report #CSL-TR-96-696, Stanford University.

Patterson, D.A., and J.L Hennessy. **1998**. *Computer Organization and Design: The Hardware/Software Interface*. Second edition. Morgan Kaufman, San Mateo, CA.

Przybylski, S. A. **1990**. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufman, San Mateo, CA.

Ranganathan, P., V. S. Pai and S. V. Adve. **1997**. "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Bewteen Memory Consistency Models". *Proc. 9th Annual ACM Symp on Parallel Algorithms and Arch.* Pages 199-210.

Ravindran, G., and M. Stumm. **1997**. "A Performance Comparison of Ring- and Mesh-connected Multiprocessor Networks". *Proc. 3rd Int'l Symp on High Performance Computer Architecture (HPCA'97).*

Rosenblum, M., E. Bugnion, S. Devine and S. A. Herrod. **1997**. "Using the SimOS Machine Simulator to Study Complex Computer Systems". *ACM Trans. on Modeling and Computer Simulation* Vol. 7, No. 1, January. Pages 78-103.

Russel, R. M. **1978**. "The CRAY-1 Computer System". *Comm. of the ACM* Vol. 21, No. 1. Pages 63-72.

Sandhu, H.S., B. Gamsa and S. Zhou. **1993**. "The Shared Regions Approach to Software" Cache Coherence on Multiprocessors. *Proc. 4th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming.* Pages 229-238.

Scott, S. L., J. R. Goodman and M. K. Vernon. **1992**. "Performance of the SCI Ring". *Proc. 19th Ann'l Int'l Symp. on Computer Architecture (ISCA'92).* Pages 403-414.

Semiconductor Industry Association. **1997**. *The National Technology Roadmap for Semiconductors: Technology Needs.* Third edition, SEMATECH. http://notes.sematech.org/ntrs/Rdmpmem.nsf.

Simoni, R., and M. Horowitz. **1991**. “Modeling the Performance of Limited Pointers Directories for Cache Coherence”. *Proc. 18th Annual Int’l Symp. on Computer Architecture (ISCA’91)*. Pages 309-319.

Singh, J. P., W.-D. Weber, and A. Gupta. **1992**. “SPLASH: The Stanford ParalleL Applications for SHared Memory”. *Computer Architecture News* Vol. 20, No. 1. Pages 5-44.

Srbljic, S., Z. G. Vranesic, M. Stumm and L. Budin. **1997**. “Analytical Prediction of Performance for Cache Coherence Protocols”. *IEEE Trans. on Computers* Vol. 46, No. 11, November. Pages 55-73.

Stenstrom, P., T. Joe and A. Gupta. **1992**. “Comparative Performance Evaluation of Cache-coherent NUMA and COMA Architectures”. *Proc. 19th Annual Int’l Symp. on Computer Architecture (ISCA’92)*. Pages 80-91.

Stroustrup, B. **1986**. *The C++ Programming Language*. First edition. Addison-Wesley, Reading, MA.

Stunkel, C. B., D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Price, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao and P.R. Varker. **1995**. “The SP2 High-Performance Switch”. *IBM Systems Journal* Vol. 34, No. 2. Pages 185-204.

SUN Microsystems. **1997a**. *UltraSPARC^{TM}-IIi User’s Manual*. Part #806-0087-01. SUN Microelectronics, Palo Alto, CA.

SUN Microsystems. **1997b**. *VIS^{TM} Instruction Set User’s Manual*. Part #805-1394-01, July. SUN Microsystems, Mountain View, CA.

Sundaram, C. R. M., and D. Eager. **1995**. “Future Applicability of Bus-based Shared Memory Multiprocessors”. *Proc. 7th Annual Symp. on Parallel Algorithms and Arch. (SPAA’95)*. Pages 203-212.

Szymanski, T., and H.S. Hinton. **1995**. “Design of a Terabit Free-Space Photonic Backplane for Parallel Computing”. *Proc. 2nd Int’l Conf. on Massively Parallel Processing Using Optical Interconnections (MPPOI’95)*.

Torellas, J. and D. Padua. **1996**. “The Illinois Aggressive COMA Multiprocessor Project (I-ACOMA)”. *Proc. 6th Symp. on the Frontiers of Massively Parallel Computing*.

TPC. **1999**. Transaction Processing Performance Council. http://www.tpc.org.

Valiant, L.G. **1990**. “A Bridging Model for Parallel Computation”. *Communications of the ACM* Vol. 33, No. 8. Pages 103-111.

Veenstra, J.E., and R. Fowler. **1993**. *Mint Tutorial and User Manual.* Tech. report #452, Comp. Science Dept., U. Rochester.

Vranesic, Z. G., S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb and S. Srbljic. **1995**. *The NUMAchine Multiprocessor.* Tech. report #CSRI-324, Dept. of Comp. Science, University of Toronto.

Weber, W-.D. and A. Gupta. **1989**. “Analysis of Cache Invalidation Patterns in Multiprocessors”. *Proc. 3rd Int’l Conference on Architectural Support for Programming Languagnes and Operating Systems (ASPLOS’89)*. Pages 243-256.

Weber, W.-D., S. Gold, P. Helland, T. Shimizu, T. Wicki and W. Wilcke. **1997**. “The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-performance Servers”. *Proc. 24th Int’l Symp. on Computer Architecture (ISCA’97)*. Pages 98-107.

Woo, S. C., J. P. Singh and J. L. Hennessy. **1993**. *The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors*. Tech. report #CSL-TR-93-593, Stanford University.

Woo, S. C., M. Ohara, E. Torrie, J.P. Singh and A. Gupta. **1995**. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. *Proc. 22nd Int’l Symp. on Computer Architecture (ISCA-22)*. Pages 24-36.

Yeager, K. C. **1996**. “The MIPS R10000 Superscalar Microprocessor”. *IEEE Micro* Vol. 16, No. 2, April.

Zhang, Z. and J. Torrellas. **1995**. “Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA”. *Proc. 3rd Int’l Symp. on High-Performance Computer Architecture (HPCA’97)*.