A Quantitative Performance Evaluation of SCI Memory Hierarchies

Roberto A Hexsel

Ph.D.
University of Edinburgh

1994

Abstract

The Scalable Coherent Interface (SCI) is an IEEE standard that defines a hardware platform for scalable shared-memory multiprocessors. SCI consists of three parts. The first is a set of physical interfaces that defines board sizes, wiring and network clock rates. The second is a communication protocol based on unidirectional point to point links. The third defines a cache coherence protocol based on a full directory that is distributed amongst the cache and memory modules. The cache controllers keep track of the copies of a given datum by maintaining them in a doubly linked list. SCI can scale up to 65520 nodes.

This dissertation contains a quantitative performance evaluation of an SCI-connected multiprocessor that assesses both the communication and cache coherence subsystems. The simulator is driven by reference streams generated as a by-product of the execution of "real" programs. The workload consists of three programs from the SPLASH suite and three parallel loops.

The simplest topology supported by SCI is the ring. It was found that, for the hardware and software simulated, the largest efficient ring size is between eight and sixteen nodes and that raw network bandwidth seen by processing elements is limited at about 80Mbytes/s. This is because the network saturates when link traffic reaches 600–700Mbytes/s. These levels of link traffic only occur for two poorly designed programs. The other four programs generate low traffic and their execution speed is *not* limited by interconnect *nor* cache coherence protocol. An analytical model of the multiprocessor is used to assess the cost of some frequently occurring cache coherence protocol operations. In order to build large systems, networks more sophisticated than rings must be used. The performance of SCI meshes and cubes is evaluated for systems of up to 64 nodes. As with rings, processor throughput is also limited by link traffic for the same two poorly designed programs. Cubes are 10–15% faster than meshes for programs that generate high levels of network traffic. Otherwise, the differences are negligible. No significant relationship between cache size and network dimensionality was found.

Acknowledgements

This dissertation is one of the products of my living in Scotland. It was a period of much discovery, both technically and personally. On the personal side, I lived there long enough to become well acquainted with British culture and politics. Some of the good memories I will keep are from many hours spent with BBC Radio 4, BBC 1 and 2, Channel 4, The Edinburgh Filmhouse, The Cameo Cinema, The Queen's Hall, The Royal Sheakespeare Company. Through these media, I met the Bard and Handel, Inspectors Taggart and Frost, Jeremy Paxman and John Pilger, Marina Warner and Glenys Kinnock, Noam Chomsky and Edward Said, Dennis Skinner and Tony Benn, Prime Minister Question Time and Spitting Image, Peter Greenaway and a wealth of European cinema. Along with many others, these people, their work and the institutions they work for became an important element of my thinking. Being in exile is not easy but it can be an extremely enriching experience. It was for me.

Many people took part, directly or indirectly, in the work that is reported here. Some were related to me in a professional capacity, some helped by just being there. I would like to express my gratitude to them all.

The Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Ministério da Educação, Brazil, awarded the scholarship that made possible my coming to Britain.

I would like to thank the people at the Department of Computer Science for making easy my life and endeavours as a graduate student. In particular, I am grateful to Angela, Chris (`cc`), Eleanor, George, Jenny, John (`jhb`), Murray, Paul, Sam and Todd. I would also like to thank all the people who put up with my simulations hogging the compute servers and/or their workstations.

Dave Gustavson, the chairman of the IEEE-SCI working group, provided useful coments on a paper about SCI rings that evolved into Chapter 4. He kept insisting, thankfully, that *SCI=rings* is not true!

Nigel Topham was my supervisor and I am indebted to him for the opportunity to work on Computer Architecture. I am also indebted to Nigel for his guidance and support. Had I followed his advice more closely on a few occasions, much time and grief would have been saved. Stuart Anderson was my second supervisor and I thank him for being there during the mid-summer crises.

My parents offered much needed support and encouragement. Without their financial support in the later stages of this work, it would not have been completed.

I must mention the hillwalkers, with whom I discovered much beauty and wilderness in Scotland. Fábio, Kees, and Paul got me started. During the addiction stage we endured mist, rain, gales, sleet and snow, and occasionally, breathtaking views from the tops and ridges of many Munros and Corbetts. In this period were also participants Brian, Irene, Jessy, Olivier, Peter, and many others. The time we spent together on the hills and glens will be sorely missed.

I am also grateful to the house- and office-mates with whom I shared the many frustrations and few delights of being a foreign graduate student: Claudio, Davide, Deena, Li Guo and Matthew. Also, the friends who were neither office/house mates nor hillwalkers: Alice & Alberto, Cláudia, Jane & Stephen, Laureance, Mandy and Margareth. I am grateful to Joaquin for the many long discussions we had, which helped me to stay within reasonable levels of sanity.

Many people kept writing to me, in the forlorn hope of a reply. Their letters kept me in a healthy state of homesickness. Bila & Conrado, Dina, Edela, Guga and Lina, kept a constant and most welcome stream of letters flowing during these years.

Most of all, I thank Helena, for her endurance, love and wee Julia.

Declaration

I declare that this Thesis was written by myself and that the work it describes is my own, except where stated in the text.

Roberto André Hexsel
22nd of September 1994.

Table of Contents

1. Introduction **1**

2. Shared Memory Multiprocessors **5**

2.1 Interconnection Networks 6

2.2 Shared Memory Implementations 10

2.2.1 Cache Memories 10

2.2.2 Multiprocessor Cache Coherency 12

2.2.3 Ring Based Shared-Memory Multiprocessors 15

2.3 The Scalable Coherent Interface 16

2.3.1 SCI Communication Protocol 17

2.3.2 SCI Cache Coherence Protocol 18

2.3.3 Related Work 21

3. The Architecture Simulator **24**

3.1 Simulation Methodology 24

3.2 The Simulated Multiprocessor 27

3.2.1 Processors and Memory Hierarchy 27

3.2.2 The Simulation Model of SCI Rings 30

3.3 The Workload 32

3.3.1 SPLASH Programs 33

3.3.2 Parallel Loops 34

3.3.3 Data Set Sizes 36

3.4 Accuracy of the Simulation Results 38

4. The Performance of SCI Rings **40**

4.1 Performance Metrics 40

4.2 Node and Ring Design 42

4.2.1 Design Space 42

4.2.2 Characterising the Workload 43

4.2.3 Cache Size and Cache Access Latency 48

4.2.4 Processor Clock Speed 53

4.3 Throughput and Latency 54

4.4 Other Ring-based Systems 59

4.4.1 Comparing DASH and SCI 59

5. A Model of the SCI-connected Multiprocessor **67**

5.1 The Analytical Model 67

5.2 Costing Sharing-lists and Conflict Misses 73

6. The Performance of Meshes and Cubes **78**

6.1 The Simulated Multiprocessor 78

6.1.1 Routing 79

6.1.2 SCI Switches 80

6.2 SCI Meshes 81

6.2.1 Machine and Cache Size – SPLASH Programs 81

6.2.2 Machine and Cache Size – Parallel Loops 83

6.2.3 Throughput and Latency 85

6.3 SCI Cubes 89

6.3.1 Machine and Cache Size 89

6.3.2 Throughput and Latency 92

6.4 A Comparison of Rings, Meshes and Cubes 95

6.4.1 Throughput and Latency 97

6.4.2 Cache Size and Network Dimensionality 99

7. Conclusion **101**

A. Performance Data **114**

A.1 SCI Rings 115

A.1.1 chol() – DASH Parameters 115

A.1.2 mp3d() – DASH Parameters 116

A.1.3 water() – DASH Parameters 117

A.1.4 chol() 118

A.1.5 mp3d() 121

A.1.6 water() 124

A.1.7 ge() 127

A.1.8 mmult() 130

A.1.9 paths() 133

A.2 SCI Meshes 136

A.2.1 chol() 136

A.2.2 mp3d() 137

A.2.3 water() 138

A.2.4 ge() 139

A.2.5 mmult() 140

A.2.6 paths() 141

A.3 SCI Cubes 143

A.3.1 chol() 143

A.3.2 mp3d() 143

A.3.3 water() 144

A.3.4 ge() 145

A.3.5 mmult() 146

A.3.6 paths() 147

List of Figures

2.1 Interconnection networks: network size *versus* cost. 6

2.2 SCI link interface. 18

2.3 Sharing-list setup. 19

2.4 Sharing-list purge sequence. 20

3.1 Simulation environment. 26

3.2 Architecture of the processing nodes. 27

3.3 SCI link interface. 30

4.1 Execution time breakdown for `chol()`, `mp3d()` and `water()`. 45

4.2 Execution time breakdown for `ge()`, `mmult()` and `paths()`. 46

4.3 Shared-data read hit ratios for 64K, 128K, 256K and 512Kbytes coherent caches. 48

4.4 Execution time as a function of cache size, for `chol()`, `mp3d()` and `water()`. 51

4.5 Execution time as a function of cache size, for `ge()`, `mmult()` and `paths()`. 52

4.6 Speedup achieved by doubling processor clock frequency, with cache sizes of 64K and 256Kbytes. 53

4.7 Throughput per node, coherent cache sizes of 64K and 256Kbytes. . 54

4.8 Average round-trip delay, with cache sizes of 64K and 256Kbytes. . 55

4.9 Latency *versus* throughput on 2-, 4-, 8- and 16-node rings. 56

4.10 Traffic per link, for cache sizes of 64KK and 256Kbytes. 57

4.11 Distribution of packet sizes for 256Kbytes caches. 58

4.12 Average packet size for 64K. 128K, 256K and 512Kbytes caches. . . 58

4.13 Speedup plots for `chol()`, `mp3d()` and `water()`. 62

4.14 Execution time breakdown for `chol()`, `mp3d()` and `water()`. 63

4.15 Normalised execution time breakdown for `chol()` `mp3d()` and `water()`, for shared data references. 64

4.16 Execution time breakdown for `chol()`, `mp3d()` and `water()` – 100MHz clock. 65

4.17 Normalised execution time breakdown for `chol()`, `mp3d()` and `water()`, for shared-data references – 100MHz clock. 65

5.1 Distribution of error in the model predictions when compared to the simulation result. 73

5.2 Effect of long sharing lists on the performance of `paths()` with 256K and 512Kbytes caches. 74

5.3 Effect of conflict misses on the performance of `mmult()` with 128K and 512Kbytes caches. 75

5.4 Effect of conflict misses on the performance of `paths()` with 256K and 512Kbytes caches. 75

5.5 Effects on performance of conflict misses and sharing-list length for `mp3d()` on 256Kbytes coherent caches. 77

6.1 A four-by-four SCI mesh. 79

6.2 Data paths of a two-dimensional SCI switch. 80

6.3 Execution time plots for `chol()`, `mp3d()` and `water()`. 82

6.4 Coherent cache shared-data read hit ratio plots for `chol()`, `mp3d()` and `water()`. 82

6.5 Execution time breakdown for `chol()`, `mp3d()` and `water()`. 83

6.6 Execution time plots for `ge()`, `mmult()` and `paths()`. 84

6.7 Coherent cache shared-data read hit ratio plots for `ge()`, `mmult()` and `paths()`. 85

6.8 Execution time breakdown for `ge()`, `mmult()` and `paths()`. 85

6.9 Throughput per node for 256Kbytes caches. 86

6.10 Round-trip delays for 256Kbytes caches. 86

6.11 Link traffic per dimension, output buffer traffic per dimension and bypass buffer traffic per dimension, for 256Kbytes caches. 88

6.12 A four-by-four-by-four SCI cube. 89

6.13 Execution time plots for `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`. 90

6.14 Coherent cache shared-data read hit ratio plots for `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`. 90

6.15 Execution time breakdown for `chol()`, `mp3d()` and `water()`. 91

6.16 Execution time breakdown for `ge()`, `mmult()` and `paths()`. 92

6.17 Throughput per node and round-trip delay, for 256Kbytes caches. . 93

6.18 Link traffic per dimension, output buffer traffic and bypass buffer traffic, all for 256Kbytes caches. 94

6.19 Performance of 4-node and 8-node multiprocessors, with 256Kbytes caches. 95

6.20 Performance of 16-node and 64-node multiprocessors, with 256Kbytes caches. 96

6.21 Processor throughput and round-trip delay for `mp3d()` and `paths()`, 256Kbytes caches. 97

6.22 Execution time and processor throughput for `paths()`, processor clock of 100 and 200MHz, 256Kbytes caches. 99

6.23 Performance of `mp3d()` and `paths()` with 128K, 256K and 512Kbytes caches, on 4-, 8-, 16- and 64-node multiprocessors. 100

Chapter 1

Introduction

As soon as the newest, biggest and fastest computer is delivered to its users they become emboldened by their new computational prowess and attempt to solve larger and more complex problems. This in turn creates the need for an even bigger and faster computer since the problems they are trying to solve grow at a faster rate than computer architects can design new machines. In the last few years, the high-performance computing community has been turning its attention to massively parallel machines, that is, several hundred processors cooperating in the solution of ever larger and more complex problems.

One of the challenges facing multiprocessor designers is the machinery to allow a large number of processors to share data. One school of design advocates as little sharing as possible; all cooperation and synchronisation is achieved by the exchange of messages. The machines based on message passing are called *multicomputers* since each processing element in these machines contain processors and memory and is a computer in its own right. The other school of design advocates as much sharing as possible. In these *multiprocessors*, large sections of the address space is shared by all processors. The crucial difference between the two designs is the cost of processor cooperation. In a multicomputer, any communication involves the assembly and dispatch of a message over the communication channel. In a multiprocessor, processors simply write or read memory locations when they have to cooperate. Unfortunately, sharing memory is more complicated than "reading and writing to memory". With current technology, the cooperating processors are likely to be a couple of feet from each other and from the memory modules. The communication between processors and memory is concealed by hardware mechanisms which make it invisible to the programmer and are, in theory, faster than the software mechanisms involved in message-passing systems.

Communication entails transmission delays. While two cooperating processors are accessing a shared variable, a third processor might become interested in that variable as well. If the distances between the three processors are different, it is very likely that they will perceive changes in memory state at different times. Which version of the datum is "the good one" then? The problem is made more complicated by the use of cache memories. A cache is a block of very fast memory that sits near the processor in order to reduce delays in the path between processor and main memory. Each processor keeps in its cache copies of memory words it has referenced in the recent past, in the hope they will be referenced again in the near future. If one of the processors updates the contents of its cache, all other copies of the data must be updated or, at least, invalidated.

In a shared memory multiprocessor, keeping the shared portion of the addressing space in a consistent state is the task of the memory subsystem. This subsystem consists of the memory itself (caches and the slower main memory) and the mechanisms that allow processors to cooperate while keeping memory in a consistent state. These physically-distributed logically-shared memory systems have received much attention from the Computer Architecture community recently. Chapter 2 surveys some of the research in this area.

One of the proposals for implementing shared memory is the Scalable Coherent Interface (SCI), an IEEE standard for providing a coherent memory interface to as many as 65520 processing nodes [IEE92]. SCI consists of three parts. The first defines a set of physical interfaces such as connectors, cabling, board sizes and clock rates. The second part defines a communication subsystem that allows processors to exchange information efficiently and correctly. The communication subsystem is based on unidirectional point to point links and data are transferred between processing nodes by packets containing commands and (sometimes) data. The third part defines a scalable cache-coherence protocol. Memory is kept consistent by the invalidation of out-of-date copies. The protocol keeps track of the number and location of the copies of each shared data block by maintaining a linked list of the copies. SCI is described in more detail in Section 2.3. Other efforts at evaluating the performance of SCI as a communication medium are also surveyed in Section 2.3.

This dissertation contains a performance evaluation study of a complete SCI-based multiprocessor where the influence of both interconnect and memory hierarchy are investigated in detail. Such a study is important because it can reveal deficiencies and bottlenecks that might be overlooked when only parts of a complex system are exercised. Other researchers have investigated the performance of some aspects of SCI and their work has been focussed mainly on the communication subsystem. The performance of a shared-memory multiprocessor depends on the communication medium between processors and memory as well as on its implementation of shared-memory. For example, studies on SCI's communication subsystem rely on assumptions about traffic patterns that do not seem to occur in practice.

While every simulation environment is based on certain assumptions about the simulated system, the quality of the results it produces depends crucially on how well the simulator "implements" reality. Thus, a set of architectural parameters that is representative of current designs was selected and a multiprocessor based on these parameters was used to execute scientific programs that are also representative of current practice. The results produced by the simulated system are therefore a fair indication of the performance of an actual system.

The simulation experiments described here are, as far as the author is aware, the first to examine in depth the behaviour of a complete multiprocessor, consisting of processors, a memory hierarchy and an SCI interconnection. The relationship between cache coherency and interconnect is crucial to the performance of a multiprocessor since these two must operate in synergy. In particular, a network must have enough capacity to transmit the cache coherency commands without introducing unreasonable delays and, the coherency protocol must generate low network traffic for the most common patterns of data sharing.

The simulator used in the research reported here is driven by reference streams produced by the execution of real programs, rather than from a synthetic workload. The workload consists of three programs from the SPLASH suite [SWG91] – Cholesky, MP3D and WATER – and three parallel loops – Gaussian elimination, matrix multiplication and all-to-all minimum cost paths. The simulation environment, the simulated multiprocessor and the programs used to drive the simulator are described in Chapter 3.

The simplest pattern of interconnecting processing nodes with SCI is in a ring. In this topology, the output interface of a node is connected to the input interface of the next "downstream" node. Chapter 4 contains the results of experiments designed to assess the performance of SCI rings and the relationship between performance, the number of processors in the ring and the design parameters of the memory subsystem. Rings were simulated with one, two, four, eight and sixteen 100MIPS processors. The memory system parameters investigated were cache size, cache access latency and processor clock speed. These experiments provide data on the behaviour of the communication system and cache coherence protocol and expose performance bottlenecks.

Simulation is an inherently slow, albeit accurate, method for evaluating the performance of computing systems. The simulation experiments reported here take a very long time to run; over 100 CPU hours in some cases. A much less time consuming alternative is to use an analytical model of the system under scrutiny. One such model of the behaviour of the SCI-connected multiprocessor is presented in Chapter 5.

Earlier on it was said that there is a need for machines with large numbers of processors. Chapter 4 presents evidence that the number of processors (of the type simulated) that can be efficiently interconnected in a single SCI ring is rather small. In order to increase the number of processors in a system, richer interconnection patterns must be used. Chapter 6 explores the performance of systems with up to 64 processors, interconnected with SCI links in two different topologies, namely the mesh and the cube. The behaviour of the memory subsystem and of the communication networks are investigated in detail. The extensions to the simulation environment presented in Chapter 3, such as routing and switches, are described in Section 6.1.

The findings from Chapters 4, 5 and 6 are summarised in Chapter 7. The Appendix contains tables with the numeric data produced in the experiments.

Chapter 2

Shared Memory Multiprocessors

From the programmer's point of view, an ideal shared memory multiprocessor can be programmed in the same way as multiprogrammed uniprocessors. This means that a shared memory multiprocessor must provide two abstractions. First, it has a large addressing space accessible by all processors. Second, the cost of references to *any* location in the address space must be roughly the same. Computer architects have to design machines that implement these abstractions. The processors and memory modules might be built on several printed circuit boards yet the whole system has to behave as if implemented on a small silicon monolith, in order to fulfil the second abstraction. With current technology, a possible implementation is to have a processor and a portion of the address space on each module, and to interconnect the modules so that processors have access to all the memory in the system. Section 2.1 presents some of the possibilities for interconnecting the processors and memory. Section 2.2 discusses the implementation of the shared-memory abstraction. Section 2.3 describes in some detail the Scalable Coherent Interface (SCI). SCI defines an interconnect and a shared-memory implementation. Logically-distributed memory systems, or message passing multicomputers, are not discussed here. Good surveys on these systems can be found in [IT89,Sto90,Hwa93].

2.1 Interconnection Networks

There are many ways of interconnecting the components of a shared-memory multiprocessor and this section contains a non-exhaustive survey of such interconnects. The size of the system that can be built with each of the networks depends on inherent characteristics of each network, such as electrical (capacitive loading) or topological (too large a distance between two nodes). The balance between cost and performance as well as the intended application determine the suitability of a given design. Figure 2.1 depicts the design space for interconnection networks. In the figure, network types are roughly ordered by hardware costs and number of nodes they can support.

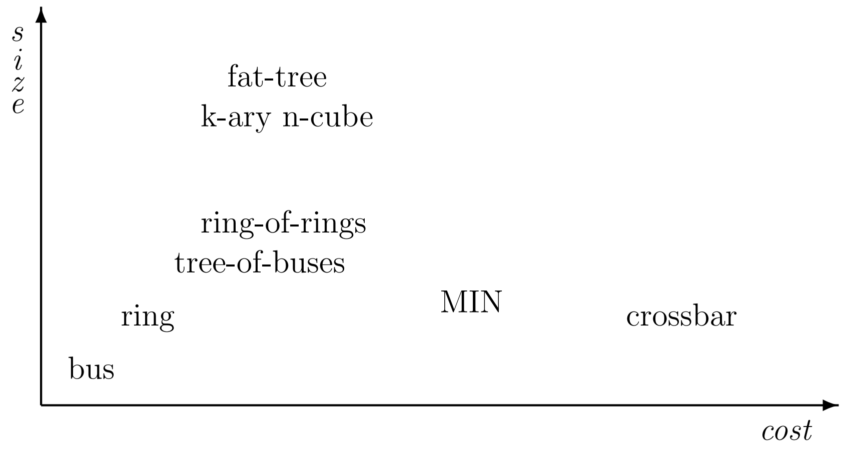


Figure 2.1: Interconnection networks: network size *versus* cost.

Two very important characteristics of a network are its bandwidth and latency. *Bandwidth* is the rate of information transfer. The term can be used to refer to the bandwidth of a link, in which case it is determined by the “width” of the link (number of signal-carrying wires) and the signalling rate (network clock frequency). Network bandwidth also measures the amount of information transfer that a whole network can support per time unit. Network bandwidth depends on link bandwidth and topology. The *topology* of a network defines what nodes are connected to what nodes, and through how many intermediary nodes. Network *latency* is a measure of the time it takes for a unit of information to be moved from its source to its destination. Latency also depends on link width, network clock rate and topology. Processor *throughput* is the rate of data transfer achieved by processors. The throughput is a fraction of the available bandwidth and it is limited by network bandwidth and latency. *Packets* or *messages* carry data or command/control information. A packet consists of a number of *flits* or *symbols*. A symbol is the smallest unit of information transferred along a link in a single network clock cycle. The packet header normally contains the destination address, source address, command or control information, and sometimes data.

Bus, tree-of-buses. A *bus* consists of a set of parallel signal and data lines used by devices to communicate. A bus is normally used in a master-slave fashion, where a master module temporarily assumes control of the bus and slave modules act on its commands. This network has some serious limitations. One is electrical loading which limits the number of devices that can be connected to a bus. The other limitation arises from the competition by devices for access to the bus. The conflicts must be resolved by an arbiter and that increases the time it takes for an access cycle to complete. Buses are in widespread use as they are an inexpensive yet effective way of connecting small numbers of processors – up to about 32 devices. The VMEbus [VIT90] and Futurebus$^+$ [IEE91] are two examples of high performance buses. A tree, or hierarchy, of buses can be used to connect a larger number of devices. When an access request is destined to a device on a distant branch, the request is propagated up towards the root and then down towards the destination. This architecture allows for concurrent activities in disjunct branches of the tree. The buses near the root can become bottlenecks if traffic levels are high. Examples of such machines are the Wisconsin Multicube [GW88] and the Data Diffusion Machine [HALH91,Hag92].

Ring, ring-of-rings. The processing nodes can be connected by unidirectional point-to-point links into a *ring* where the output of one node is connected to input of the next. A ring is topologically and functionally equivalent to a bus but provides higher performance and can connect up to about 64 devices. Since links connect only two nodes, the transmission rates can be much higher than in buses because of the smaller and more controllable electrical loading. The low-level transmission control mechanism between two adjacent nodes can be synchronous or asynchronous. In the former, each datum transmitted is acknowledged by the receiver. In an asynchronous ring, it is assumed that all data transmitted is accepted by the receiver and synchronisation actions between output and input take place only at certain intervals.

In a bus, a centralised arbiter decides when a device can transmit onto the bus. In a ring, the *access mechanism* determines when a given device can start transmitting a message [Tan89]. There are three access mechanisms. In a 'token ring' a token circulates around the ring and when a node is in possession of the token it transmits its data and then passes the token along. In a 'slotted ring', the ring is logically divided in slots and when a device sees an empty slot, it can insert a message in that slot. Thus, in a slotted ring, there can be several messages in flight at any moment. The third access mechanism is 'register insertion'. The SCI ring uses this mechanism and it is described in detail in Section 2.3 (page 17). Besides the higher transmission rates, the other advantage of slotted and register insertion rings over buses is the possibility of several devices transmitting concurrently. The Express Ring [BD91] is an example of a slotted ring. The Cambridge Ring is a high-speed token-ring local area network [HN88]. As is the case with buses, rings can also be assembled in a hierarchy, in what is called a *ring-of-rings.* Hector [VSLW91] and the KSR1 [BFKR92] are examples of multiprocessors built as a hierarchy of slotted rings.

Multistage interconnection network. A *multistage interconnection network* (MIN) consists of a sequence of permutation circuits followed by multi-way switches. Machines built with MINs normally have one such network between processors and memory and another between memory and processors in what is called a 'dance hall' architecture. Some interconnection patterns might require multiple round-trips over the network since, depending on the topology, not all pairs <processor, memory> are reachable by just steering the switches appropriately. MINs are thus called *indirect networks.* Several topologies for the permutation circuits have been proposed and used in high performance vector processors. Examples of permutation circuits are the Delta, Butterfly and Shuffle networks [IT89, Sto90,JG91,Hwa93].

Crossbar switch. A *crossbar switch* is a grid of buses and at each crossing there is a switch that can connect the vertical and horizontal buses. Processors are attached to, say, the vertical buses and memory modules to the horizontal buses. These networks are expensive for large machines but have very high bandwidth between processors and memories and allow for high levels of concurrency [Hwa93].

k-ary n-cube. The nodes of a *k-ary n-cube* contain n links to neighbouring nodes, one for each of the n dimensions [Sei85,Dal90]. Along a given dimension, there are k nodes. For example, a binary 3-cube is a cube with 2 nodes on each of three dimensions. Networks with arity $k = 2$ are called *hypercubes.* The links can be uni- or bi-directional. With bi-directional links, buses can be used to connect the nodes along each dimension. With uni-directional links, the nodes along each dimension must be connected in rings. In a k-ary n-cube, dimensionality n, ring size k and number of nodes N are related by

$$N = k^n, \qquad k = \sqrt[n]{N}, \qquad n = \log_k N \tag{2.1}$$

If uni-directional links are used, the number of link interfaces, or fanout, is $2n$. Network capacity is proportional to the fanout and thus to n. For a given system size and link width, higher dimensional networks are more expensive since they use more link interfaces. However, the maximum distance between two nodes is smaller [JG92]:

$$d_{max} = n(k-1) \tag{2.2}$$

since, in the worst case, a packet must visit all nodes in a ring passing through all links minus one before switching on to the next dimension. This must be done on all dimensions. The number of "ring hops" is n. If the cost to switch rings is c, the maximum static latency is proportional to

$$l_{max} = nk + (c-2)n. \tag{2.3}$$

An important issue is the *routing* strategy used to deliver messages in k-ary n-cubes. Since there is more than one possible route from source to destination, routing must be deterministic and deadlock-free. In a *store-and-forward network*, complete incoming packets are buffered before a routing decision is made. If buffers become full, new packets can either be dropped or sent on alternative routes. In a *wormhole routed network*, routing decisions are made as soon as the first packet symbol arrives at a node. This means that the first symbol of a packet might reach its destination before the last symbol is transmitted by its source. In either mechanism, there is a possibility of paths being blocked by packets in flight and thus of deadlock. One routing mechanism that is deadlock-free is described in Section 6.1.1 (page 79).

Fat-tree. In a *fat-tree*, leaves are normally processing nodes (processor + memory) and internal nodes are normally switches, with nodes connected via bi-directional links. The main characteristic of this network is that bandwidth increases towards the root. This compensates for the higher levels of traffic suffered at nodes far from the leaves. An example of a fat-tree network is that in the Connection Machine CM-5 [Lei85,LAD+92]. A fat-tree can also be implemented as a ring-of-rings, as in the KSR1 [BFKR92], or as a hierarchy of buses, as in the Data Diffusion Machine [HALH91,Hag92].

2.2 Shared Memory Implementations

This section discusses some of the proposed or existing implementations of the shared-memory abstraction. The basic problem to be solved by any implementation is to provide all processors with a coherent view of memory. Because memory modules are physically distributed in space, two processors could perceive the update of a given memory word at different instants. This would be a most undesirable feature since the majority of programs would not exhibit deterministic behaviour or, the programming of such machines would be an exceedingly difficult task. The solution is to ensure that all processors in a system perceive updates at the same instant and thus do not operate on stale data.

2.2.1 Cache Memories

Matters are further complicated by the use of cache memories [HP90,Sto90, Hwa93]. The rate at which DRAMs can service access requests is much lower than the rate at which processors issue requests – in the range of about 2 to 100 times. One solution is to add a small but very fast block of memory between processor and main memory DRAMs. This small and fast memory is called a *cache* memory since, in principle, it is hidden from the programmer. The success of cache memories depends on the Principles of Locality. *Temporal locality* means that if a memory word was referenced in the recent past, it is very likely that it will be referenced again in the near future. *Spacial locality* means that if a memory word was referenced, other words in its neighbourhood are very likely to be referenced as well.

When the processor issues a request for a word, if that word is in the cache (a *hit*), it is returned with low latency. If the word is not in the cache (a *miss*), it is fetched from main memory before being returned to the processor. Caches are normally designed to service a read request in one processor clock cycle, and a write request in two or more cycles. The *hit ratio* is the ratio between the number of hits and the number of processor requests. The closer the hit ratio is to unity, the shorter it takes to run a given program.

A cache is divided into a number of *frames*, with a fixed number of words per frame. Memory is divided into *lines* that have the same size as cache frames. Before any lines are copied from memory to the cache, all cache frames are invalid. A frame may contain a copy of a memory line, in which case the contents of the frame are said to be valid. A valid frame might be dirty if it has been written to and its contents are different from those of the memory line. If a valid frame is not dirty, it is clean.

Since caches are smaller than physical memory, a mapping of lines to frames must be used. In a *fully associative* cache, any memory line can be loaded into any frame. The matching between an address issued by the processor and the contents of the cache frames is done by a fully associative search. When a newly referenced line needs to be loaded into a full cache, some policy must be used to decide which line is to be evicted to make space. The policies are usually some form of least recently used or random replacement. In a *N-way set associative* cache, each *set* consists of *N* frames and a mapping function on the address bits relates lines to sets. A 1-way set associative cache is called a *direct mapped* cache.

There are four types of cache misses. *Compulsory* misses occur when a line is first referenced and must be brought into the cache (also called 'cold-start misses'). *Capacity* misses occur when the data set of a program is larger than the cache. Blocks that are flushed for lack of space must be retrieved later on. *Conflict* misses occur when two lines map to the same cache frame. *Coherency* misses are caused by actions of other processors, *i.e.* a shared line is updated by another processor.

A cache can be *write-through*, that is, writes are passed along directly to the memory. In a *write-back* cache, the copy of a line is only updated in the cache. When the updated frame must be flushed to make space for a newly referenced line, the contents of the frame are copied back to memory. Another issue related to writes is line allocation. *Write-allocate* caches allocate a frame on a write. *No-write-allocate* caches do not allocate frames on writes. Normally, write-through caches do not allocate frames on writes.

Some systems have a *write-buffer* between cache and memory. It consists of a queue of words to be written to memory. Writing to the write-buffer takes one or two processor clock cycles whereas writing to memory can take many cycles, *e.g.* flush the (possibly unused) dirty line then write the new line to the cache/memory. When the word at the head of the queue is written to memory, that word is retired from the buffer and a new write cycle is started. Write-buffers can greatly reduce the time a processor stalls while waiting for writes to complete.

2.2.2 Multiprocessor Cache Coherency

On a multiprocessor, the use of caches creates a serious problem. Since many copies of a given line may exist at any time, when one of the copies is updated, all other copies must be either updated or marked invalid. One solution is to rely on software techniques to separate out read-shared, write-shared and not-shared (local) data. Read-only variables can be read and copied by all processors. Writable variables can have no copies at all. The variables belonging to each of these classes are grouped and allocated to read-only/read-write pages and the operating system manages the updates of writable pages. Examples of software controlled shared-memory systems are Ivy [LH89], Mach [FBYR89] and Clouds [RAK89]. The disadvantage of this technique is the low level of parallelism that it yields on some applications [CDK$^+$94]. A hybrid approach can be used where some of the most common coherency operations are implemented in hardware as in the Galactica Net [WLI$^+$94] and Alewife [CA94].

Another solution is to implement in hardware a cache coherence protocol that keeps memory in a coherent state [Ste90,CFKA90,CKA91,Hwa93]. Such a protocol defines the sequence of actions needed on changes of memory state, *e.g.* when a cached line is updated. A cache coherence mechanism consists normally of the cache controllers and of a directory. The cache controllers contain state-machines that control the protocol actions. The *directory* holds the state, location and number of copies of each memory line.

The actions of a cache coherence protocol depend on the state of each memory line. Besides the three states needed for uniprocessor caches (invalid, clean, dirty), a cache coherence protocol has to determine whether a line is shared, since this case must be treated differently by the protocol actions. The directory entry for a line consists of its state and one or more pointers to the copies of the line. A *full directory* is normally implemented with a bit vector per cache or memory line where each bit points to one of the processors in the system. If a processor has a copy of the line, the corresponding bit is set. A *partial directory* only holds a few pointers to processors with copies. When the number of copies exceeds the number of pointers, either further copying is disallowed or all existing copies are marked invalid. Weber and Gupta, in [WG89], contend that for well-designed programs, 3 or 4 pointers are sufficient to keep invalidation of copies to a minimum. A *chained directory* keeps the list of copies as a linked-list or linked-tree, rooted at memory. The directory is normally split in two sections, part in memory and part in the caches. Full directories can be implemented with a presence flag vector in memory. The caches only keep the state of each line they hold. Partial and chained directories keep the pointers and line state partly in memory and partly in the caches.

Snooping protocols. Most of the early cache coherence protocols proposed and implemented were *snooping* protocols. These are well suited for small and medium size machines built around a bus. In these machines, broadcast operations can be implemented efficiently since all devices attached to the bus can monitor bus activity. Whenever a write to a shared line occurs, all cache controllers attached to the bus update any copies of the line they might hold, that is, all controllers "snoop" on bus activity. Normally, the state of each cached line is kept at the cache itself and no pointers to copies are needed. Since rings are topologically equivalent to buses and allow for inexpensive broadcasts, ring-based multiprocessors tend to use snooping protocols [BD91,BD93,VSLW91,BFKR92].

Invalidation protocols. One of the disadvantages of buses and rings is their poor *scalability.* A system is said to be scalable when more processors can be added without any major degradation in performance or cost [Hil90,Bel92,Sco92]. The main advantage of more general networks is their scalability to large number of processors. However, broadcasts are inefficient on such networks. More processors mean more network traffic when messages signalling changes in data have to be broadcast. The solution is to avoid broadcasts by using directory-based cache coherence schemes that send cache coherency commands to just those caches that hold copies of shared lines. When a line is written to, messages are sent to the caches that have copies of the line. Normally, the copies are invalidated since this uses less bandwidth than updating large numbers of copies [ASHH88,CFKA90, Ste90,Hwa93].

Examples of *invalidation protocols* are the one in Alewife [CKA91,CA94], that uses a partial directory with five pointers and software support when the number of copies exceeds five. Thapar and Delagi, in [TD91], propose a distributed linked-list invalidation protocol. SCI [IEE92], also uses a distributed linked-list directory. The Scalable Tree Protocol has its directory organised as a distributed linked-tree [NS92]. DASH is a full directory cache-coherent shared-memory multiprocessor [LLG$^+$90,LLJ$^+$92]. The memory coherence is maintained by a distributed invalidation directory-based protocol. Section 4.4.1 (page 59) compares the performance of an SCI-based multiprocessor to that of DASH.

DASH, like SCI-based machines, is called a Cache Coherent Non-Uniform Memory Access Machine (CC-NUMA) because of the difference in access times for local and remote references. The KSR1 is called a Cache Only Memory Hierarchy (COMA) because its memory hierarchy consists of only primary and secondary caches, with no main memory. The Data Diffusion Machine is also a COMA type multiprocessor [Hag92]. Its interconnect is a hierarchy of buses and coherency is maintained by a write-invalidate snooping protocol. The HORN DDM [MSW94] uses a Banyan MIN network [IT89] and cache coherency is maintained by an invalidation linked-list based protocol. Stenström *et al.* present a comparative performance study of COMA and CC-NUMA architectures in [SJG92]. Hagersten disagrees with those results and presents a more refined model in [Hag92]. This new model compares NUMA and COMA architectures with similar levels of complexity and design optimisation. The latency equations more closely reflect COMA's behaviour and are based on more realistic design parameters. For the architectures investigated, the new model yields results that give COMA a clear performance advantage over NUMA. The difference in performance stems from the better tolerance to network latency exhibited by the COMAs.

Memory consistency models. The use of caches in shared memory multiprocessors improves their performance by reducing the number of cycles during which processors are stalled waiting for memory requests to complete. A memory system is said to be *sequentially consistent* if the programming model it provides is the same as a multiprogrammed uniprocessor [Lam79]. To implement sequentially consistent memory, the processors must stall until every memory request completes. "Completion" is determined by the absolute ordering of actions as perceived by all processors in the system: a memory request completes when its effects have propagated throughout the whole system. Sequential consistency is a very restrictive model since it might disallow the use of write-buffers, for example [DS90].

A more relaxed consistency model is *processor consistency* [Goo91]. In this model, the writes by each processor are always completed in the order they appear in the program text, *i.e.* in program order, but writes from different processors might complete in any order. The order of reads is not restricted as long as the reads do not involve other processors. Write-buffers are allowed in processor consistent memory systems and their performance is much better than sequentially consistent systems, mostly because expensive invalidations can be overlapped with other memory requests [GGH91].

An even less strict model is defined by a *weak ordering* of events [DS90]. Under this model, the synchronisation operations are sequentially consistent. The order of successive writes by the same processor must be respected. Buffering of ordinary references is allowed but not of references to synchronisation variables. A processor acquires locks and releases them in program order, by using the commands `lock()` and `unlock()`, respectively. All ordinary memory requests by that processor must be completed before each subsequent `lock()` and `unlock()` is issued. A further weakening is possible by only imposing an ordering on lock releases – this model is called *release consistency* [GLL$^+$90]. The *weakest memory-access order* model is proposed by Bitar in [Bit92], where a semantics for asynchronous multiprocess computation is defined.

Weaker models yield better performance at increased hardware and programming costs. In weaker models, the cache controllers must keep track of each outstanding request. Also, the programmer must label the synchronisation points where consistency must be enforced. Of the five models, processor consistency seems to be the best cost *versus* performance compromise [GGH91].

2.2.3 Ring Based Shared-Memory Multiprocessors

Uni-directional rings have some very interesting topological properties. Ordering is easily enforced and broadcasts can be efficiently implemented. Snooping protocols can be designed to take advantage of these features. Also, rings allow for the pipelining of messages and thus can be designed as very low-latency, high-bandwidth networks. Rings of rings can be used for scaling up to larger numbers of processors. The problem with this topology is the latency of cross-ring transactions. These latencies are not prohibitive however because broadcasting and snooping can hide some of the delays.

To date, the KSR1 [BFKR92] is the only commercially available ring-based shared-memory multiprocessor. It consists of a hierarchy of rings and cache coherence is maintained by a snooping write-invalidate protocol. An important feature of the KSR1 is its memory hierarchy, composed only of primary and secondary caches. The KSR1 can grow up to 1088 processors in a two-level hierarchy of rings. The *ring:0* can accommodate 32 processors; the *ring:1* supports up to 34 *ring:0*'s. The remote access latency on a 32-node ring is 6μs and, to reduce its effects, the KSR1 supports the software mechanisms prefetch and poststore.

Farkas *et al.*, in [FVS92], present an invalidation-based cache coherency scheme for a hierarchy of rings that takes advantage of the natural broadcasting and ordering properties of rings. In the Hector multiprocessor [VSLW91], processors are grouped in clusters that also contain memory modules and a communication controller. The coherency protocol is based on snooping both at the intra-cluster buses and on the rings. The amount of traffic caused by broadcasting consistency commands can be reduced by filtering out incoming and/or outgoing messages at inter-ring interfaces. The performance results presented are somewhat unreliable because of the simulation method (asynchronous event generation) and the traces employed (from a different architecture). The performance of the coherence scheme, when compared to that of no caching, yields speedups in the range of 30–195% on three applications from the SPLASH suite [SWG91] – SA-TSP, LocusRoute and PTHOR.

Barroso and Dubois describe the Express Ring in [BD91,BD93]. It is based on a slotted ring where each of the ring interfaces is a pipeline that can hold three symbols. The pipeline is divided in slots of different sizes for control and data messages. When a node sees a message directed to itself, the message is removed from the ring and the empty slot can be used by the next downstream node. Their simulation results indicate that remote access latency is rather small. They investigate two cache coherence protocols, one based on snooping and the other on a full-map directory. Their results indicate that the snooping protocol yields better performance. Also, the performance of the slotted ring is significantly better than that of a split transaction bus. The maximum number of nodes that can be assembled on an Express Ring is limited to between 32 and 64.

2.3 The Scalable Coherent Interface

The project that became SCI started as the design of a very high performance bus. Early on it became obvious that the bandwidth of a bus would always place a hard limit on performance. The solution was to employ point to point connections since these allow for much higher clock rates and richer interconnection patterns. The description that follows concentrates on those features of SCI that are of relevance here. For full details, please see [IEE92].

SCI defines three subsystems, namely the physical-level interfaces, the packet-based logical communication protocol, and the distributed cache coherence protocol. The physical interfaces are high-speed uni-directional point-to-point links. One of the versions prescribes links 16 data-bits wide which can transfer data at peak speed of 1 Gbyte/s. The standard supports a general interconnect, providing a coherent shared-memory model, scalable up to 64K nodes. An SCI node can be a memory module, a processor-cache pair, an I/O module or any combination of these. The number of nodes on a ring can range from two to a few tens. For most applications, a multiprocessor will consist of several rings, connected together by switches, *i.e.* nodes with more than one pair of link interfaces.

2.3.1 SCI Communication Protocol

The *communication protocol* comprises the specification of the sizes and types of packets and of the actions involved in the transference of information between nodes. A packet consists of an unbroken sequence of 16-bit symbols. It contains address, command/control and status information plus optional data and a check symbol. A command/control packet can be 8 or 16 symbols long, a data packet can be 40 or 48 symbols long and an echo packet is 4 symbols in length. A data packet carries 64 bytes of data. 16- and 48-symbol packets carry an additional pointer (node address) used in some cache coherence operations, *e.g.* sharing-list purges (see below).

The protocol supports two types of action: *requests* and *responses*. A complete transaction, for example, a remote memory data-read, starts with the requester sending a request-send packet to the responder. The acceptance of the packet by the responder is acknowledged with a request-echo. When the responder has executed the command, it generates a response-send packet containing status information and possibly data. Upon receiving the response-send packet, the requester completes the transaction by returning a response-echo packet. The communication protocol ensures forward progress and contains deadlock and livelock avoidance mechanisms.

The network access mechanism used by SCI is the register insertion ring. Figure 2.2 shows a block diagram of the link interface. A node retains packets addressed to itself and forwards the other packets to the output link. A request transaction starts with the sender node placing a request-send packet, addressed to the receiver node, in the output buffer. Transmission can start if there are no packets at the bypass buffer and no packet is being forwarded from the stripper to the multiplexor. At the receiver, the stripper parses the incoming packet and diverts it to the input buffer. On recognising a packet addressed to itself, the stripper generates an echo packet addressed to the sender and inserts it in place of the 'stripped' packet. If there is space at the input buffer, the echo carries an ack (positive acknowledge) status. Otherwise, the packet is dropped and a nack (negative acknowledge) is returned to the sender who will then retransmit the request-send packet.

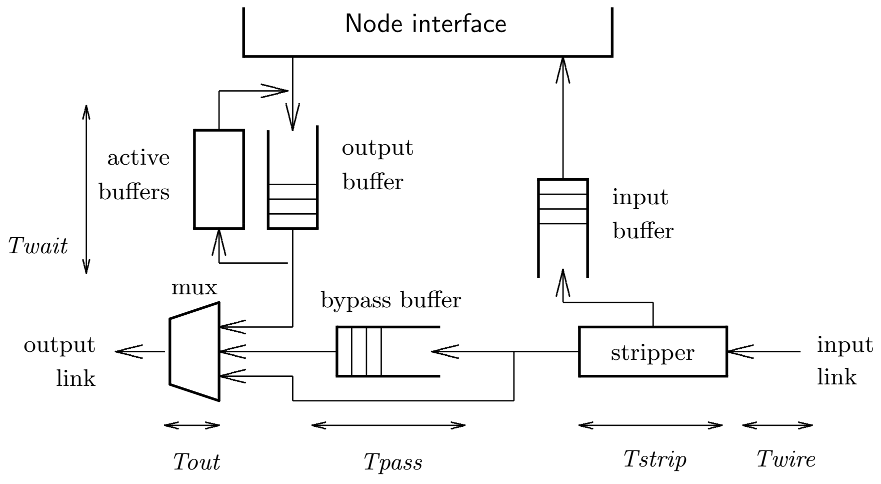


Figure 2.2: SCI link interface.

It is likely that during the transmission of a packet, the bypass buffer might fill up with packets not addressed to the node. Once transmission stops, the node enters the *recovery phase* during which no packets can be inserted by the node. Each packet stripped creates spaces in the symbol stream. These spaces, called idle symbols, eventually allow the bypass buffer to drain, when new transmissions are then possible. The protocol also ensures that the downstream nodes cannot insert new packets until the recovery phase is complete. This will cause a reduction in overall traffic and create enough idles to drain the bypass buffer.

When a packet is output, a copy of it is kept in an *active buffer.* If the status of a packet's echo is ack, the original packet is dropped from the active buffer and the node can transmit another packet. If the echo carries a nack, the packet is retransmitted. This allows for one or more packets to be active simultaneously, *e.g.* one transaction initiated by the processor and other(s) initiated by the cache or memory controller(s). The number of active buffers depends on the type of the "pass transmission protocol" implemented. The options are: only one outstanding packet, one request-send and one response-send outstanding or, several outstanding request- or response-send packets.

2.3.2 SCI Cache Coherence Protocol

The SCI cache coherence protocol is a write-invalidate chained directory scheme. Each cache line tag contains pointers to the next and previous nodes in the doubly-linked sharing-list. A line's address consists of a 16-bit node-id and 48-bit address offset. In protocol actions, the cache and memory controllers can determine the node address of any memory line by the 16-bit node-id. The storage overhead for the memory directory and the cache tags is a fixed percentage of the total storage capacity. For a 64-byte cache block, the overhead at memory is 4% and at the cache tags 7%. Note that these are *capacity* overheads and do not translate directly into cost.

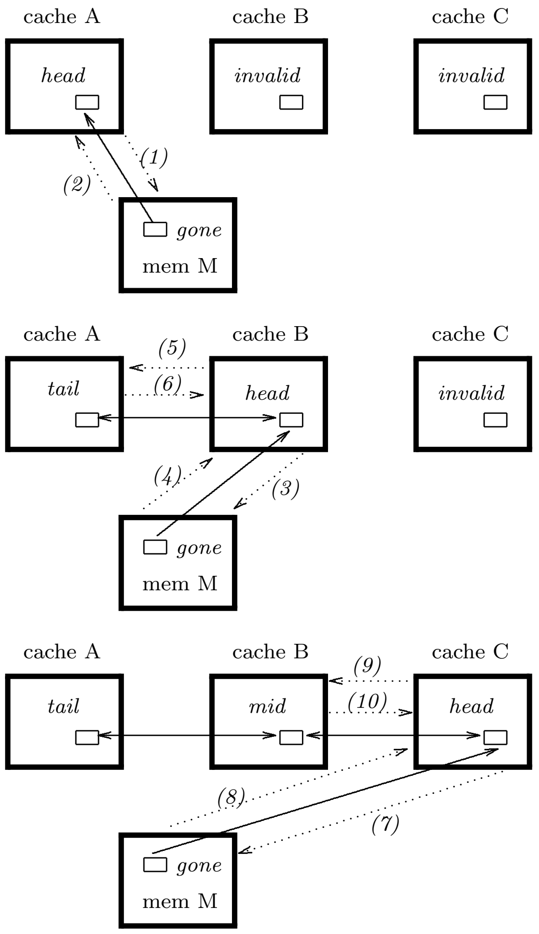


Figure 2.3: Sharing-list setup. Solid lines represent sharing-list links, dotted lines represent messages.

Consider processors A, B and C, read-sharing a memory line L that resides at node M – see Figure 2.3. Initially, the state of the memory lines is home and the cache blocks are invalid. A read-cached transaction is directed from processor A to the memory controller M (1). The state of line L changes from home to gone and the requested line is returned (2). The requester's cache block state changes to the head state, i.e. head of the sharing-list. When processor B requests a copy of line L (3), it receives a pointer to A from M (4). A cache-to-cache transaction, called prepend, is directed from B to A (5). On receiving the request, A sets its backward pointer to B and returns the requested line (6). Node C then requests a copy of L from M (7) and receives a pointer to node B (8). Node C requests a copy from B (9). The state of the line at B changes from head to mid and B sends a copy of L to C (10), which becomes the new head. In SCI, rather than having several request transactions blocked at the memory controller, all requests are immediately prepended to the respective sharing-lists. When a block has to be replaced, the processor detaches itself from the sharing-list before flushing the line from the cache.

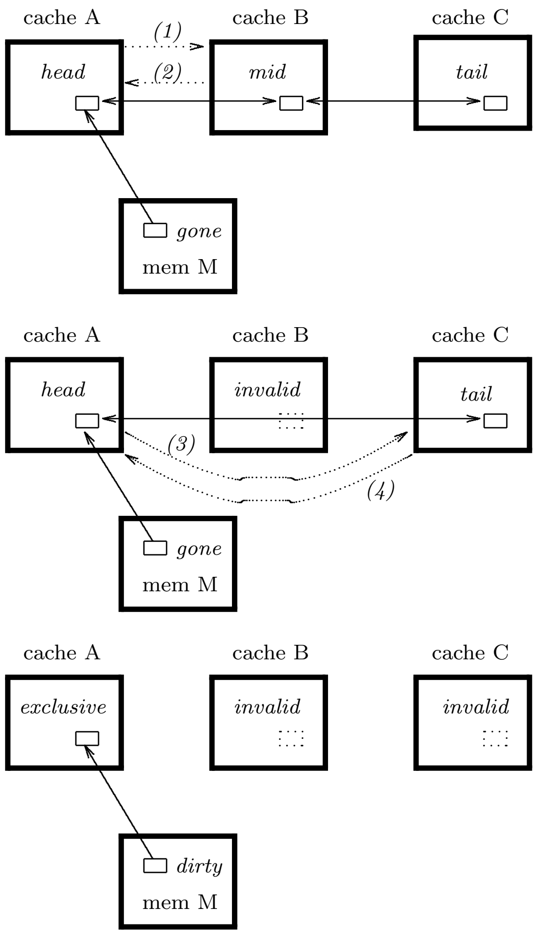


Figure 2.4: Sharing-list purge sequence. Solid lines represent sharing-list links, dotted lines represent messages.

Before writing to a shared line, the processor at the head of the sharing-list must purge the other entries in the list to obtain exclusive ownership of the line – see Figure 2.4. Node A, in the head state, sends an invalidate command to node B (1). Node B invalidates its copy of L and returns its forward pointer (pointing to C) to A (2). Node A sends an invalidate command to C (3) which responds with a null pointer, indicating it is the tail node of the sharing-list (4). The state of line L, at node A, changes to exclusive and the write completes. When a node other then the head needs to write to a shared line, that node has to interrogate the memory directory for the head of the list, acquire head status and then purge the other entries. The node that holds the tag in main memory of a line (*i.e.* its home node) can be determined from the most significant 16 bits of the line's address. If the writer is at the middle or tail, it first has to detach itself from the sharing-list before attempting to become the new head.

2.3.3 Related Work

Scalability up to 64K nodes comes at the price of added complexity at the communication and cache coherency protocols. For instance, a write to a shared datum needs a larger number of packets for its completion than that needed by DASH's protocol [LLG$^+$90]. Johnson, in [Joh93], proposes extensions to SCI's cache coherence protocol to alleviate this problem on larger systems. Additional links can be added to the sharing-lists thus transforming them into sharing-trees. The proposed schemes improve significantly the performance of invalidations even for low degrees of sharing.

Nilsson and Stenström, in [NS92], describe the Scalable Tree Protocol (STP), a cache coherence protocol based on sharing-trees. The advantage of trees over lists is that, to invalidate n copies of a line, only $O(\log n)$ messages are needed on a tree whereas $O(n)$ messages are needed on a list. [NS93] compares the performance of three types of directories, namely a full-map, an SCI-like linear-list, and a tree-based, in an ideal architecture. The full-map has better performance because it minimises invalidation traffic. The list-based is worse than STP if the degree of data sharing is high or if memory is sequentially consistent. If data is migratory [WG89], *i.e.* shared by at most two processors, then the linked-list performs better than STP because of the lower latency involved in invalidating just a few copies.

Aboulenein *et al.*, in [AGGW94], examine SCI's hardware synchronisation primitive, Queue On Lock Bit (QOLB). Its potential efficiency comes from it fitting neatly with the linked-lists since waiting processes are naturally enqueued when they join the sharing-list for the lock.

Bugge *et al.*, in [BKB90], compare the performance of three uniprocessor memory architectures, two of which are based on a 32-bit and on a 64-bit wide Futurebus$^+$. The third employs SCI links between secondary cache and memory.

The emphasis is on memory hierarchy design and thus coherence related issues are not investigated. Their trace-driven simulation results indicate that the SCI-based system outperforms the other two when the secondary cache hit ratio is higher than 90%. Also, with a time-shared multiprogramming workload, secondary cache *size* has the largest impact on the performance of the memory hierarchy whereas the influence of tag access latency is small. The simulated caches are much larger than the ones investigated here. The primary caches are 128K bytes, and the secondary range from 1M bytes to 8M bytes. Their results do not agree with those presented here because their simulations ignore coherence traffic and the increase in latency with ring size and, the workloads are very different both in nature and size.

Bogaerts *et al.*, in [BDMR92], present simulation results for 10-node rings and a multi-ring system with 1083 nodes for data acquisition applications in particle physics. They concentrate on the bandwidth consumed by SCI moveXX transactions for DMA and ignore coherence related events. Their experiments with 10-node rings show that certain traffic patterns can severely limit the bandwidth available to each node. For DMA move256 transactions (move 256 bytes with no acknowledgement), the effective bandwidth is about 175 Mbytes/s per node. When fair bandwidth allocation is employed, effective bandwidth drops to 125 Mbytes/s. For this type of transaction, the data portion of the packet is proportionally larger than for 64-byte transactions thus incurring in smaller transmission overheads. The experiments with 64-byte data packets investigate pathological cases and the results are not indicative of more normal conditions.

Scott *et al.*, in [SGV92], present an analytical model of the SCI logical communication protocol. Scott's dissertation [Sco92] presents a more detailed discussion on the model and results. The model is based on M/G/1 queues and the ring is modelled as an open system. Both uniform and non-uniform workloads are investigated. The model is validated against simulation results. The flow control mechanism, used to enforce fairness in bandwidth sharing (simulations only), is effective in preventing starvation and in reducing the effects of a hot transmitter on the ring. This mechanism is not as effective for non-uniform routing distributions. Fairness comes at a cost however. The maximum ring throughput is reduced by up to 30%, larger rings being more adversely affected. Read-request/read-response data-only aggregate ring throughput, for 64 byte data blocks, is around 600Mbytes/s, fairly distributed among the nodes. They show that an SCI ring compares favourably to a conventional bus.

The scalability of k-ary n-cubes has been investigated under different sets of constraints. In a *synchronous network*, each flit (symbol) is acknowledged by the receiving node, that is the handshake is on a flit by flit basis. This makes the network clock frequency dependent on the distance between sender and receiver. Dally found that, in the context of networks built in a single VLSI chip, low-dimensional networks ($n = 2$) yield the lowest latencies [Dal90]. This conclusion holds true under constant bisection width, which is the case in an area limited two-dimensional VLSI circuit. Agarwal investigated these networks under different constraints, namely constant channel width and constant node size and concluded for the need of higher dimensionality than in Dally's study ($3 \leq n \leq 5$) [Aga91]. SCI is an *asynchronous network* because clock synchronisation between adjacent nodes is not based on flit transmission between nodes, rather the clocks of two adjacent nodes are assumed to be running at the same frequency[1]. This makes the network clock cycle independent of wire length and allows for the pipelining of flits onto wires. Unlike synchronous networks, the clock frequency depends mainly on the technology employed in the interfaces and to a much smaller degree on the topology of the network. Scott and Goodman investigated the scalability of asynchronous k-ary n-cube networks [SG91]. Their results point to even higher optimal dimensionalities than in synchronous networks ($4 \leq n \leq 12$). Thus, asynchronous k-ary n-cubes should be grown by increasing dimensionality n while keeping ring size k constant.

[1]The link interface circuits provide a few cycles of *elasticity* to accomodate small differences in frequency.

Chapter 3

The Architecture Simulator

This chapter describes the simulation methodology and justifies the choices and compromises made in the implementation of the methodology. Section 3.1 presents the simulation methodology and the simulator design. Section 3.2 describes the simulated architecture and how this architecture is "implemented" by the simulator. This section also contains the model for the behaviour of the SCI rings. The memory reference stream generator and the programs that comprise the workload are presented in Section 3.3. Finally, Section 3.4 discusses the accuracy of the simulation results.

3.1 Simulation Methodology

There are various methodologies for investigating the performance of computing systems and a choice of method entails making tradeoffs between computational cost and accuracy of the predictions. Jain, in [Jai91], discusses the tradeoffs between analytical modelling, simulation and direct measurement. Przybylski, in [Prz90], discusses trace-driven simulation and analytical modelling of cache memories and memory hierarchies. Both authors argue that while analytical models provide answers at a much lower computational cost (*i.e.* in less time), their accuracy is limited by the level of detail and complexity of the models. On the other hand, simulation studies provide more accurate results but take longer to perform.

One approach to architecture simulation is to drive the simulator with traces of execution of a number of programs. Trace-driven simulation has two inherent problems. The first is the dilation introduced by instrumenting the program whose trace will be used to drive the simulator. This dilation makes the instrumented program from two to 2000 times slower than the original program, depending on the technique used for tracing [BKW90,GNWZ91,KEL91]. Because of the dilation, the relative timing of events in the simulated system may, in the worst case, bear no relationship with event ordering on an actual system.

The second problem stems from changes in the run-time environment of a parallel program on consecutive program runs. Potentially, each time a program runs, the interleaving of memory references can be significantly different from previous runs because of differing allocation of threads to processors and/or other system-dependent factors. Fortunately, Koldinger *et al.*, in [KEL91], conclude that dilation induced effects on miss ratio and bus utilisation measurements are negligible, and that multiple-run effects are insignificant unless one is interested in absolute values for a given metric.

Taking the above into consideration, as well as availability of tools, the methodology chosen for the investigation described in this dissertation is on-the-fly reference stream generation. The simulator comprises about 3000 lines of C code, six application programs that were ported and adapted plus several shell scripts for post-processing the simulation results. The simulation environment that was employed is described below.

The simulator consists of a memory reference stream generator and an architecture simulator. They both execute concurrently as Unix processes and communicate through Unix sockets. Figure 3.1 depicts the simulation environment. The stream of memory references is generated as a by-product of the execution of the simulated parallel program. The reference stream is piped to the architecture simulator which computes the latency of each (simulated processor) reference to memory. This latency is used by the reference stream generator to choose the next simulated thread to run. In this way, the latency of each individual memory reference is accounted for thus reproducing with good accuracy the interleaving of memory references on a real machine.

The architecture simulator consists of an approximate model of the SCI link interfaces and of a detailed model of the distributed cache coherence protocol. The model of the ring interfaces is similar to those in [SG91,SGV92] but rather than using statistical analysis, traffic-related values are measured and directly influence the behaviour of the simulated system. The model of the cache coherence protocol mimics the “typical set coherence protocol” as defined in [IEE92].

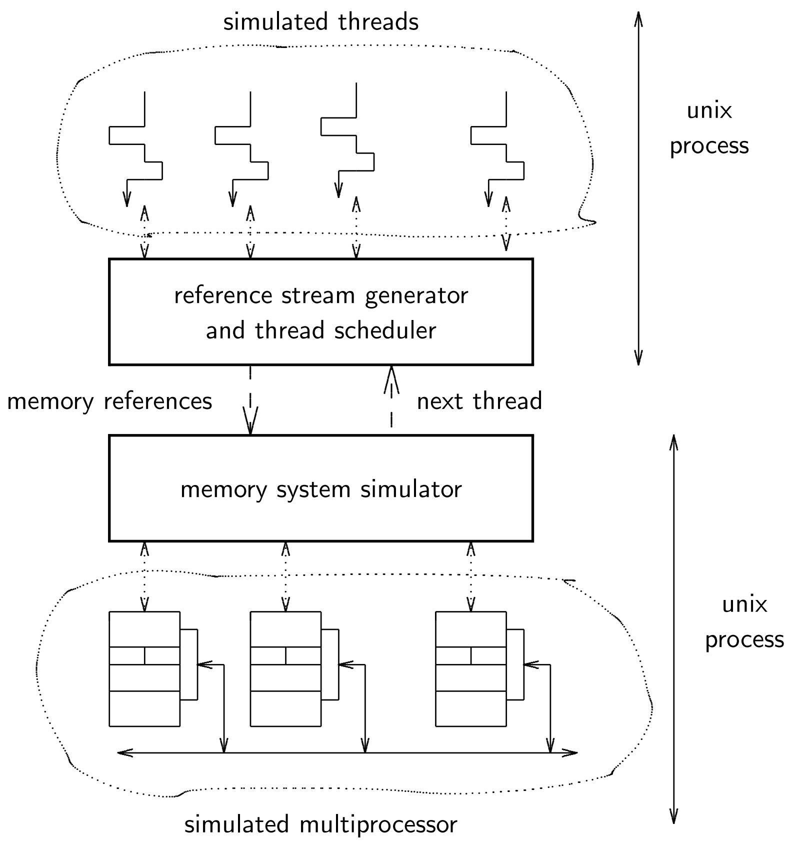


Figure 3.1: Simulation environment.

The address sequences used to drive the simulator are generated by instrumenting the parallel programs described in Section 3.3 with Symbolic Parallel Abstract Execution (SPAE) [GNWZ91]. SPAE is based on the GNU `gcc` compiler and allows for tracing parallel programs at any desired level of detail. The resolution of the reference stream generator is at instruction/data reference level. The cost of each memory reference is computed from the state of the system – level of network traffic and coherence actions performed – and those values are used to schedule the execution of the simulated processors. The simulated parallel program is split into lightweight threads, one for each simulated processor. Each data reference by the simulated processors causes a context switch; the thread that will next run is chosen by the architecture simulator. Likewise for instructions, except that the context switches occur only at "basic block" borders, as defined by `gcc`. Thus, the global interleaving of memory references is simulated with better accuracy than is possible with straightforward trace simulation [BKB90,BKW90] or with the method proposed in [MB92]. However, the computational cost is much higher. Typically, a simulation run takes from 1 to 100 CPU hours on a lightly loaded Sparcstation2, depending on the data-set size.

3.2 The Simulated Multiprocessor

The multiprocessor consists of a number of processing nodes interconnected in one or more rings by SCI links. Each node contains a processor, a split primary cache, a coherent secondary cache, memory and one or more SCI interfaces – see Figure 3.2. The individual components are described below.

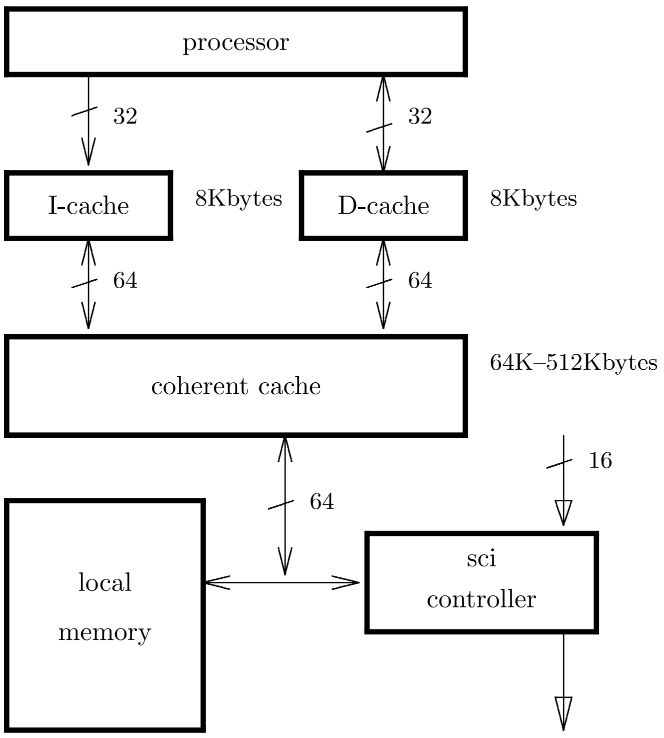


Figure 3.2: Architecture of the processing nodes.

3.2.1 Processors and Memory Hierarchy

The CPU is a 32-bit scalar Harvard processor that performs an instruction fetch and possibly a data read/write access on every clock cycle. The processor clock frequency is a simulation parameter and the values investigated are 100 and 200MHz. The instruction set is that of a SPARCstation processor since that is the processor the simulated code is compiled for and executed on. The simulated processors always stall on memory references (both reads and writes), thus the memory model is sequential consistency.

The memory hierarchy comprises three levels: small split primary caches, large secondary caches and main memory. Primary caches are virtually addressed while secondary caches are indexed by physical (real) addresses. Note that in SCI systems, the SCI-coherent caches must be physically addressed. The size of the instruction cache (i-cache) and data cache (d-cache) is 8 Kbytes each (one page), both being direct mapped. The data cache is write-through with no block allocation on write misses. The secondary cache is direct mapped and, for private data references it is copy-back with no block allocation. The mapping of virtual to physical addresses is performed in parallel with primary cache tag-matching, as proposed in [OMB91,WHL93].

The secondary cache size is a simulation parameter. Sizes investigated are 64, 128, 256 and, 512Kbytes. Local main memory is simulated as if implemented with DRAMs, with a degree of interleaving of 8. On all three levels of the memory hierarchy, cache frames and memory lines are 64 bytes wide, which is the size of the unit of coherency in the SCI cache coherence protocol. The memory hierarchy satisfies the multilevel inclusion property [BW88], and the SCI coherency protocol actions affect only the secondary caches, thus called *coherent caches*. Coherency between primary and secondary caches is maintained by the cache controllers.

The internal buses are 64 bits wide, except the processor-primary caches which are 32 bits wide. The access latency for the secondary caches is 3 processor cycles. Loading a line from the secondary cache into the primary caches or SCI controller costs 3 processor cycles plus 2ns per 64 bit word (16ns). Loading a line from/to memory costs 120ns of access latency plus 10ns per 64 bit word (80ns). Thus, a cache-to-memory read-line transaction costs 246ns for a 100MHz processor. To that, the network latency must be added if one end of the transaction, cache or memory, is at another node. Table 3.1 gives the cost, in processor cycles, of the various types of cache operations.

Cache operation	latency (cycles)
Read from primary cache	1
Fill from secondary cache	5
Fill from local memory	20
Fill from remote node	37–54
Fill from dirty-remote, remote home	44–78
Write owned by secondary cache	3
Write owned by local node	12
Write owned by remote node	47–436

Table 3.1: Cache and memory operation latencies in processor clock cycles.

The *home* of a given line is the node to which the memory page that contains that line is mapped. References to pages mapped to memory on other nodes are called *remote* references. In the early stages of this work, as reported in [HT94a], the mapping of virtual memory pages to nodes was done naively: the first node that references a given page becomes its home. This is inadequate since nodes which are home to large numbers of pages are very likely to become hot spots. Furthermore, this method also causes severe "distance imbalances" since most shared data will be at a short distance from only one processor, which will run relatively quickly, but far away from all the others.

The mapping policy employed here is still simplistic but is fairer. On the first eight page faults, the faulting node becomes the home of the page. Subsequent faults are mapped to the next "upstream" node. If the neighbour's quota has been exceeded, the page is mapped to the next upstream node with free page table entries. If no free entries exist, all nodes are awarded another eight-page quota. This quota policy spreads shared data more evenly than is possible with the naive policy, yielding better performance and more reliable results. With quotas, queue utilisation on the SCI interfaces never exceeds 20% whereas it can be higher than 50% with the naive policy. The lower the queue utilisation the more reliable the simulation results since queue overruns are much less likely to occur at low utilisations.

In order to simplify the simulator, it is assumed that on data accesses the concurrent instruction fetch hits in the primary cache and accesses to local data and instructions do not cause any traffic on the ring. The simulator ignores intra-node contention, that is, the processor of a hot spot node does not see any contention for the internal buses and its local cache or memory. It is also assumed that page faults have zero cost.

Design Choices. There are many architectural features that could be incorporated in the design outlined above. The improvements in performance achieved by the addition of devices such as write buffers, prefetching and weaker memory models are well documented in [HP90,Prz90,Sto90,Hwa93] and references therein.

These devices were not incorporated in the design of the multiprocessor because the main focus of the research described here is the evaluation of SCI as a memory system backbone rather than optimising the performance of a given base architecture. While the author is well aware of the potential improvements in performance by designing a more sophisticated machine, there is real danger of these performance gains masking out and obscuring the inherent behaviour of an SCI-based system. The design space was thus limited to a few parameters and values. To have done otherwise would have increased the complexity of the task, to a combinatorial explosion in the worst case, without necessarily extending the scope of the results.

3.2.2 The Simulation Model of SCI Rings

For the description of the model of an SCI ring that follows, please refer to Figure 2.2, repeated here for convenience in Figure 3.3. A more detailed description of the SCI communication protocol can be found in Section 2.3.

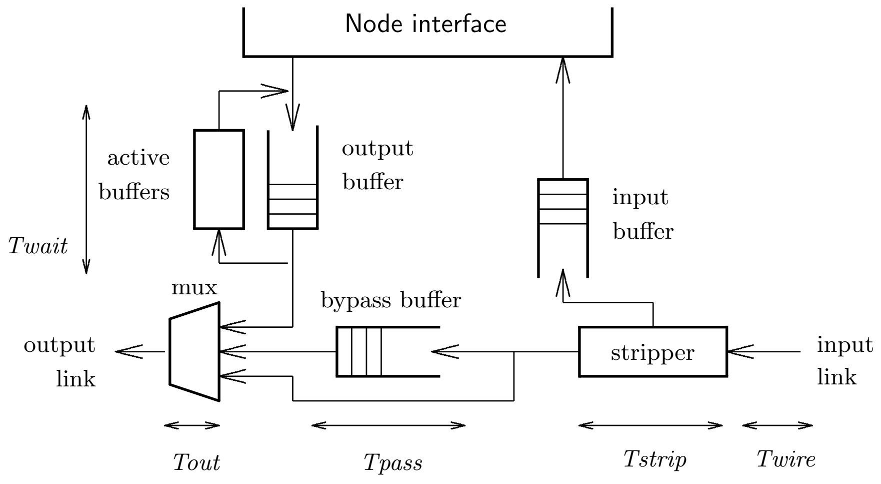


Figure 3.3: SCI link interface.

In accordance with the SCI standard, the network clock cycle is 2ns (500MHz) and the physical links are 16 bits wide. The delay faced by a packet waiting to be transmitted (*Twait*) depends on the number and size of packets passing through the node. Likewise, the delay faced by packets at the bypass buffer (*Tpass*) depends on the frequency and size of packets inserted by the node. Wire propagation delay (*Twire*) is 2ns. The time to parse an incoming packet (*Tstrip*) and the time to gate an outgoing symbol onto the output link (*Tout*) are also 2ns each. Thus, the latency L_{AB}, in network clock cycles, involved in sending a packet from Node$_A$ to Node$_B$ and waiting for its echo can be calculated as follows. To simplify the expressions, the modulus operations on summation indexes were omitted.

$$\begin{aligned}
L_{AB,type} &= \mathit{Twait}_A + \mathit{Tout} + 2\,\mathit{size(type)} && (3.1)\\
&+ \sum_{i=A+1}^{B-1} (\mathit{Twire} + \mathit{Tstrip} + \mathit{Tpass}_i + \mathit{Tout})\\
&+ \mathit{Twire} + \mathit{Tstrip} + \mathit{Tpass}_B + \mathit{Tout}\\
&+ \sum_{i=B+1}^{A-1} (\mathit{Twire} + \mathit{Tstrip} + \mathit{Tpass}_i + \mathit{Tout})\\
&+ \mathit{Twire} + \mathit{Tstrip}
\end{aligned}$$

Where *type* can be one of *Pcmd8*, *Pcmd16*, *Pdata*, *PdataX* and, their sizes are 8, 16, 40 and 48 symbols, respectively (1 symbol = 2 bytes). An *idle symbol* must precede each packet thus making the sizes 9, 17, 41 and 49 in the throughput calculations. The term ($2\,\mathit{size(type)}$) is the time, in nanoseconds, needed to insert a packet into the ring. The peak bandwidth of a link or buffer is the maximum number of symbols that can pass through it per time unit. In the absence of traffic, peak bandwidth of the output or bypass buffer is 500 Msymbols/s (1Gbyte/s).

The average packet size through a link or buffer is

$$\mathit{Pavg} = \sum_p f_p size(p) \,/ \sum_p f_p \qquad (3.2)$$

where $p \in \{\mathit{Pcmd8, Pcmd16, Pdata, PdataX, Pecho}\}$ and f_p is the frequency of packet type p. The throughput S of a buffer is the number of symbols that pass through it per time unit:

$$\mathit{Sbuffer} = \sum_p f_p size(p). \qquad (3.3)$$

The utilisation of a link or buffer is given by the throughput divided by the bandwidth available, times the average packet size. Thus, the number of network clock cycles spent waiting for the transmission of a packet at the output buffer, *Twait*, can be written as

$$\mathit{Twait} = \mathit{Pavg}_{tx}\,\mathit{Stx} \,/ (BWmax - \mathit{Spass}) \qquad (3.4)$$

Similarly for the bypass buffer, *Tpass* is

$$\mathit{Tpass} = \mathit{Pavg}_{pass}\,\mathit{Spass} \,/ (BWmax - \mathit{Stx}) \qquad (3.5)$$

where *Spass* and *Stx* are the throughputs of the bypass and output buffers respectively, ($BWmax - \mathit{Spass}$) is the bandwidth available at the output buffer, and ($BWmax - \mathit{Stx}$) is the bandwidth available at the bypass buffer.

In the equation for the latency L_{AB} (Eq. 3.1), by making *Tpass* and *Twait* zero, the resulting equation yields the static latency of the ring, that is, it depends solely on propagation delays and is, in nanoseconds,

$$6N + 2size(p) \tag{3.6}$$

for N processors and packet p. Conversely, the dynamic component of the latency is obtained by considering only *Tpass* and *Twait*. In the simulations, dynamic latency is estimated from the measured traffic. Buffer utilisation and average packet size are measured at $10\mu s$ intervals (simulated time). Values from interval i are used to compute latencies during interval $i + 1$.

The ring interface model assumes infinite input queues and does not account for the retransmission of packets dropped at their destinations because of full queues. Since the memory is sequentially consistent, processors stall on remote references. However, cache or memory controllers may attempt to transmit response packets to complete outstanding transactions. The effect of more than one source of packets on a node is easily minimised by implementing at least two active buffers [SGV92].

3.3 The Workload

The workload used to study the behaviour of SCI consists of three parallel loops and three "real" programs. The parallel loops are small – tens of source code lines – and exhibit a well defined pattern of memory references, being based on `doall` loops [Sto90]. The real programs are much larger – two to three thousand lines of source code – and are part of Stanford's SPLASH suite [SWG91]. These programs are thread-based and the parallel programming constructs used in them are a subset of Mach C-threads [GNWZ91,SPG91]. The arrays and variables that hold shared data are allocated to a specific range of addresses. The architecture simulator treats references to these addresses as references to shared data.

The use of code optimisation has two major effects on the code produced by a compiler. It makes the programs run faster by the reordering of some groups of instructions and by a reduction on the number of load and store instructions through better utilisation of processor registers [HP90]. The aim of the experiments reported here is the understanding of the behaviour of a certain type of machine. Hence, absolute performance figures for particular pieces of code are of no relevance in this context. The object code used in the simulations was produced by `gcc v2.2.2` without any optimisation flags. The code produced by the compiler is an *input* to the simulator and no assumptions are made about the density of loads and stores.

3.3.1 SPLASH Programs

The SPLASH suite [SWG91] consists of a set of parallel scientific applications that are representative of common usage and practices in the early 1990's. These programs have been used by several other researchers to test their ideas and designs. A brief description of the three programs chosen follows.

Cholesky factorisation. `chol()` performs parallel Cholesky factorisation of a sparse matrix using supernodal elimination. The scheduling of parallel work is done by a task queue and granularity of work is large. The main data structure is the representation of the sparse-matrix itself. Cache size is one of the parameters used by the scheduler to allocate work to processors. The input matrix used is `bcsstk14` which contains 1806 equations and 30824 non-zeroes in the matrix and 110461 in the factor. The matrix `bcsstk14` occupies 420Kbytes unfactored and 1.4Mbytes factored.

Brorsson and Stenström, in [BS92], examine the patterns of reference to shared data in Cholesky. Using a highly optimised compiler and an ideal unit-delay access latency architecture, they report a large fraction of references to exclusive and read-only blocks. With a sampling interval of 32000 processor cycles and 64-byte blocks, under 60% of the references to shared-data are read-only, with under 40% being read-only exclusive. 17% of the references are read-write exclusive and the remainder is read-write shared.

MP3D. `mp3d()` is a rarefied fluid flow simulator based on Monte Carlo methods. The scheduling of tasks is static, synchronisation is based on barriers and granularity of work is large. Molecules are attached to processors rather than to spacial coordinates. Thus, as the simulation evolves, the position of molecules changes significantly but their speed and positions are always computed by the same processors. However, molecule data migrates from cache to cache as molecules collide during the simulation steps. The main data structures are the space array, describing the 3-D space and what molecules may undergo a collision. The other array holds the state of each molecule, namely its position and velocities. The data set is scaled as $1.5 \times \mathit{nodes}$. The simulation lasts 50 time steps.

Weber and Gupta, in [WG89], analyse the cache invalidation patterns in multiprocessors. Using traces for the VAX-32000 with half a million references per processor (most of one time step), 16 processors and infinite caches, they find that there are 1.03 invalidations per shared-write. Since MP3D uses arrays heavily, a typical line of source code requires 15 i-fetches, 5 reads and 1 write. Most of the shared data falls into the "migratory" category – the datum can be shared by all processors but at any one period it is only referenced by one processor.

Gharachorloo *et al.*, in [GGH91], evaluate the performance of memory consistency models. They simulate 10000 molecules for 5 time steps. On an ideal unit-delay access latency architecture, they find that there are 2.3 reads per write to shared data and 1.4 read-miss per write-miss. Also, for over 40% of the read misses in the application, there is a write miss within 30 cycles before those read misses. [BS92] also reports that for a 2000 cycles sampling interval and blocks of 64 bytes, over 50% of shared-data references are read-write exclusive, 10% read-write dominant (one processor has $> 50\%$ of references), 15% are read-write shared-by-few (1 to 4 processors out of 32) and the remainder are read-write shared by many (> 4). There are virtually no read-only shared-data references.

Water. `water()` is an n-body molecular dynamics program that evaluates forces and potentials in a system of water molecules in the liquid state. The scheduling of tasks is static, synchronisation is based on barriers and granularity of work is large. The set of molecules is split evenly amongst the processors and molecules remain attached to processors as they move in tri-dimensional space. The main data structure is an array that holds the state of each molecule – position, velocities and accelerations. The computation describing molecular motion involves a large number of array and floating point operations. The data set is scaled as $1.45 \times$ *nodes* . The system of molecules is simulated for 4 time steps.

Lenoski *et al.*, in [LLJ$^+$92], present performance data for the DASH multiprocessor. They find that water achieves a good speedup (13.3 on 16 processors), has good memory locality and does not place a heavy burden on the memory system and interconnect.

3.3.2 Parallel Loops

Gaussian Elimination. `ge()` solves a system of linear equations by Gaussian elimination and backwards substitution. In this implementation, it is assumed that the system of equations has some property that makes Gaussian elimination without pivoting numerically stable (*e.g.* diagonal dominance). The algorithm consists of several elimination stages. Each stage consists of a vector scale operation of the form $(x_{k+1} = cx_k)$ followed by a 'rank$-$1' update of the matrix

$(A_{k+1} = A_k + dxy)$ where x and y are vectors, c and d are scalars. At the k-th stage, matrix A has dimension $((n-k) \times (n-k+1))$. Input data set size grows as $1.26 \times \textit{nodes}$.

The serial version spends over 97% of the time on the 'rank−1' update. Thus, the parallelisation effort was concentrated there. No attempt has been made to optimise the serial portion of the code. The 'rank−1' update is partitioned by columns and the vertical slices are made as large as possible to minimise overheads. Node_0 does all the serial processing. When the slices of the matrix are of different sizes, the nodes with the lowest indexes compute on the largest slices. Thus, load balancing deteriorates as the computation progresses since the nodes near to Node_0 do more work than those far from it (Node_i for i large). Input data set size grows as $1.26 \times \textit{nodes}$. The source code for `ge()` was provided by Graham Riley, from the Centre for Novel Computing, Manchester University. The parallel version was compiled and optimised for the KSR1 at CNC.

Matrix multiplication. `mmult()` computes $C = A \times B$ for square matrices A and B. The algorithm consists of three nested loops and each processor computes a slice of the result matrix. All of the shared data is read-only and the little write-sharing that occurs is caused by false sharing. A local variable accumulates the partial sums for each of the result matrix's elements. This algorithm is also $O(n^3)$ and the input data set is scaled up as $1.26 \times \textit{nodes}$.

All-to-all paths. `paths()` is a member of the class of transitive closure algorithms. For a graph with N nodes, `paths()` finds the lowest cost path from each node to every other node [DPL80]. The vertices are labelled with the distance between the nodes they join and are stored in the matrix `D`. Thus, `D[i,j]` is the distance between nodes `i` and `j` and absence of a vertex is represented by infinite cost. The simulated graph is a random graph with outdegree 6. Input data set size is scaled as $1.26 \times \textit{nodes}$. The code fragment below is the parallel loop where all of the work is done.

```
doall (t = 0; t < numProc; t++)          /* All-to-all paths */
   for (k = start(t); k < end(t); k++)
      for (j = 0; j < rows; j++)
         for (i = 0; i < rows; i++)
            if (D[i,j] > (D[i,k] + D[k,j]))
               D[i,j] = D[i,k] + D[k,j];
```

3.3.3 Data Set Sizes

There are two choices for the sizes of the data sets. By fixing the data set sizes, one can measure the scalability of a (*program* + *architecture*) by looking at speedup and processor efficiency. The disadvantage is that individual processors do less and less work as the system is grown. Alternatively, by scaling up the data, the work per processor can be kept roughly constant. The performance metric is then execution time, which should remain constant as both machine and data size are grown. The difficulty here is finding the factor by which data is to be scaled up. Since the focus of this dissertation is on performance of memory hierarchies with coherent caches, an adequate way of ensuring a uniform distribution of work across processors is by keeping the number of references to shared data (roughly) constant. By choosing a large enough number of references, the caches can be fully and equally exercised, thus minimising distortion caused by cold starts. Sizes were chosen so that there are at least 1.0×10^6 references to shared data. Either way, data-set size fixed or varying, shared data miss ratios, and indeed sharing behaviour, do change as the machine size is grown. Miss ratios tend to increase with machine size because coherency misses are closely related to the level of data sharing. By scaling up the data sets, both compulsory and capacity misses tend to increase as well because of the larger cache footprints [HP90,Sto90].

Ring size		1	2	4	8	16	64	factor
`chol()`		fixed size input: bcsstk14					–	1.00
`mp3d()`	molecules	3000	4500	6750	10125	15187	34172	1.50
`water()`	molecules	54	78	113	163	237	512	1.45
`ge()`	rows	136	171	216	272	343	545	1.26
`mmult()`	rows	100	126	159	200	252	400	1.26
`paths()`	vertices	70	88	111	140	176	280	1.26

Table 3.2: Input data-set sizes and scaling factors.

Table 3.2 shows the data-set sizes and scaling factors for each of the six programs. No experiments were performed using `chol()` with more than 16 nodes since the amount of work per node would be too small on a 64-node multiprocessor. `ge()` is the program whose simulation runs take the longest – on 16 nodes, it takes about 10 CPU hours longer than any of the other programs. Thus, because of practical limitations, only a few of experiments were performed with 64 nodes. The data set size for `water()` with 64 nodes should have been 497 rather than 512. The choice of 512 is due to the input data set provided with the source code, which contains spatial distributions for up to 343 molecules. The program has an option for internally generating a spatial distribution provided the number of molecules is a cube. Hence the choice of 512.

Ideally, a workload should consist of as many representative programs as possible or practical to ensure that the system under investigation is fully exercised. The author believes that the six programs chosen are representative of the range of behaviours displayed by scientific applications. `ge()` has good locality and the amount of computation on the data is high. `mmult()` also has good locality and the small amount of write-sharing that exists is caused by false sharing. `paths()` does not have good locality. On the contrary, each processor sweeps the entire array and, potentially, each array position can be written to by every processor.

Ring size	1	2	4	8	16	64
			Cholesky – `chol()`			
shared (% wr)	10.4 (18)	12.6 (23)	8.6 (23)	5.2 (23)	2.9 (19)	—
private (% wr)	31.0 (27)	8.5 (26)	2.7 (23)	1.0 (18)	0.9 (17)	—
instructions	71.7	37.0	20.3	11.6	8.1	—
			MP3D – `mp3d()`			
shared (% wr)	5.4 (39)	5.5 (29)	4.5 (27)	5.0 (18)	6.0 (11)	6.7 (6)
private (% wr)	12.2 (18)	9.0 (18)	6.8 (18)	5.0 (18)	3.7 (18)	2.1 (18)
instructions	32.8	27.0	21.1	19.0	18.6	17.0
			Water – `water()`			
shared (% wr)	1.4 (18)	1.5 (17)	2.2 (12)	2.9 (9)	2.9 (9)	5.0 (4)
private (% wr)	14.3 (19)	15.4 (19)	16.2 (19)	16.5 (19)	17.0 (19)	16.0 (19)
instructions	30.0	30.5	33.0	34.7	35.5	39.0
			Gaussian elimination – `ge()`			
shared (% wr)	2.6 (33)	2.6 (33)	2.6 (33)	2.5 (33)	2.5 (33)	2.5 (33)
private (% wr)	13.0 (7)	12.8 (7)	12.8 (7)	12.8 (7)	12.8 (7)	12.8 (7)
instructions	33.6	33.3	33.4	33.2	33.2	33.1
			Matrix multiplication – `mmult()`			
shared (% wr)	2.0(0.5)	2.0(0.4)	2.0(0.3)	2.0(0.2)	2.0(0.2)	2.0(0.2)
private (% wr)	14.2 (14)	14.1 (14)	14.2 (14)	14.1 (14)	14.1 (14)	14.1 (14)
instructions	33.2	33.2	33.3	33.1	33.1	33.2
			All-to-all minimum cost paths – `paths()`			
shared (% wr)	1.0(0.8)	1.0(0.6)	1.0(0.4)	1.0(0.3)	1.0(0.2)	1.0(0.1)
private (% wr)	5.6 (6)	5.6 (6)	5.5 (6)	5.5 (6)	5.5 (6)	5.5 (6)
instructions	15.0	14.9	14.9	14.9	14.8	14.8

Table 3.3: Per processor reference count for the workload, in millions. 256Kbytes secondary caches.

Table 3.3 shows the reference counts per class of reference for the six programs in the workload. See the Appendix for the variations in the reference counts and hit ratios in the five cache sizes simulated. `chol()` behaves differently for different cache sizes since the processing of supernodes takes cache size into account to increase data locality – see the tables with reference counts and hit ratios on page 118. Even so, there is a fair amount of data migration while a given supernode is being eliminated. `mp3d()` exhibits poor locality since the molecules' positions change considerably during the simulation interval. As molecules collide, the data describing their position and velocity is shared by the processors to which the molecules were initially assigned. `water()` has good locality, partly because of the fair amount of computation involved in evaluating positions, speed and energy levels, partly because of the nature of the physical system itself: the molecules are "heavy" and do not move much.

In summary, the workload contains two programs that are compute-intensive (`ge()` and `water()`), two with poor locality and a fair degree of data sharing (`paths()` and `mp3d()`), one with a somewhat erratic behaviour (`chol()`) and one with little sharing (`mmult()`). These four types of behaviour encompass various levels of traffic on the interconnect and differing patterns of data sharing. Thus, both components of SCI, namely the interconnect and the cache coherence protocol, can be examined under a significant range of loading and stress.

3.4 Accuracy of the Simulation Results

The conclusions one can formulate from experimental data are only as good as the accuracy of the data on which they are based. The underlying assumptions and idealisations embedded in the architecture simulator described in this chapter are a compromise between accuracy and the computational cost of attaining such accuracy. While much effort was spent in ensuring the correctness of the "implementation" of the cache coherence protocol, it is nevertheless a model for the actual protocol that has to handle all the complexities of a distributed implementation. The simulator does not model intra-node contention for access to buses and to cache and memory arrays. This is a reasonable assumption since the secondary caches have high hit ratios. If however a node becomes a hot spot, the extra traffic and delays are not accounted for.

The behavioural model of the interconnect is based on average traffic at the interconnect rather than instantaneous values. This is a good approximation since the bandwidth available is very high. Only where the traffic is high, the predictions of the model may loose accuracy due to localised traffic fluctuations. The

SCI link input-queues are modelled as infinite buffers and the possible retransmissions of busied packets are ignored. The simulations show that queue utilisation is low, below 20% in all cases, which indicates that under normal load only a very small number of busied packets would be produced. Thus, the accuracy of the network simulator lies between that of detailed simulation of the SCI communication protocol, where the simulator keeps track of every symbol travelling on the ring [BDMR92,SGV92,Sco92] and, that of trace post-processing [MB92,BD93] or statistical analysis, where the network simulator is driven by random access patterns [BGY87,SGV92,Sco92].

Chapter 4

The Performance of SCI Rings

This chapter contains a detailed investigation of the performance of SCI-based multiprocessors interconnected in a ring. Section 4.1 begins by defining the metrics used to quantify performance. Section 4.2 explores the design space for high-performance processing nodes by assessing the performance of different cache hierarchies. Section 4.3 examines the bandwidth and latency characteristics of SCI rings. Finally, Section 4.4 discusses the performance of SCI rings in relation to that of some of the existing or proposed ring-based multiprocessors. Section 4.4.1 compares the performance of DASH and an SCI-based multiprocessor with similar architectural parameters. Parts of Sections 4.2 and 4.3 were previously published in [HT94a,HT94b].

4.1 Performance Metrics

Within a given price range, the most important characteristic of a computing system is its speed. The question most often asked is "how long does it take to run this or that program?" Since the focus here is on scientific computation, the speed metric is defined as *the time the machine takes to execute a program.* The choice of design parameters that will minimise execution time and cost is a tradeoff between the cost of each subsystem or component and the improvement in speed achievable by incorporating that component, subsystem or policy, into the architecture.

Other metrics are useful in the evaluation of an architecture. Cache hit ratios are very important in assessing memory hierarchies, along with the timing associated with the levels. The locality of reference of different types of memory objects gives rise to three different hit ratios. Instruction references have good spatial and temporal locality and the *instruction hit ratios* are normally high. The locality of shared-data depends on the type of data, *e.g.* arrays or barriers and locks. Thus, *shared-data hit ratio* measures the hit ratio of shared-data references only. Lastly, the *private-data hit ratio* indirectly measures the locality of references for non-shared data *i.e.* stack and heap areas.

In an SCI-based shared memory multiprocessor, data that is actively shared by processors is kept in linked lists, rooted at the data's home memory. When the data is to be updated, the list collapses, the data is updated and the sharing-list is eventually re-established. The collapsing of sharing-lists involves message exchanges between the processor at the head of list and each of the other nodes in the list. *Sharing-list length* is defined as the number of copies that have to be purged when a line is updated. The sharing-list length reflects the level of interference between processors on each other's computation. Because of the serialisation imposed by the coherence protocol, the cost of purging grows linearly with the length of the sharing-list.

The transport mechanism of SCI is based on unidirectional point to point links. The simplest topology that can be implemented with these links is the asynchronous insertion ring. The transmission of a packet is completed when its echo is received by the transmitter. The time lapse between the insertion of a packet into the output buffer and the receipt of its echo is the *round-trip delay* of the ring. The number of packets a node can transmit per time unit depends on the traffic on the ring. The *traffic* seen by a node at its ring interface is defined as the number of symbols per time unit that is output by the ring interface. It consists of all the symbols passing through plus those inserted by the node itself. *Throughput* is the number of symbols per time unit inserted by the node and measures the amount of coherence-related traffic generated by the processor and cache/memory controllers.

A program is said to be *processor bound* if the largest proportion of the execution time is spent performing instructions. Conversely, a program is *memory bound* when the largest fraction of the time is spent on data references. The proportion of references to shared data is only a small fraction of all memory accesses performed by the processor yet they sometimes account for a large fraction of the execution time. The "boundedness" of a program is relevant because it indicates the level of demand placed by the program on the distributed memory system.

4.2 Node and Ring Design

In this section the design space for high-performance processing nodes is explored and the scalability of SCI rings is investigated. Section 4.2.1 presents the simulation parameters considered and discusses the influence these have on system performance. Section 4.2.2 describes the behaviour of the applications with respect to execution time breakdown and coherent cache hit ratios. The effects of cache size and latency are investigated in Section 4.2.3. The generation scalability of SCI rings is considered in Section 4.2.4.

4.2.1 Design Space

The design of a memory system consists of selecting a set of architectural parameters, within price and performance constraints, that will yield a low-latency high-bandwidth path between processor and memory. The combination of parameters has to be tested with what is considered to be a "typical" workload. The design parameters investigated here are secondary cache size and latency, memory access latency, ring size and processor clock speed. The experiments relate ring size to changes in one of the other parameters. The idea is to assess the effect of each individual simulation parameter on the overall performance while relating these changes to ring size.

The cache sizes investigated are 64, 128, 256 and 512 Kbytes. The size of caches should be chosen to minimise the miss ratios, that is, as large as possible, *and* to reduce the number of cycles the processor stalls waiting for memory references to be satisfied. The cache latency depends to a large extent on the memory technology and on the sophistication of the cache policies such as replacement, write-buffers, write-through/back. Within this range of sizes (64Kbytes to 512Kbytes) the latency is independent of the size of the memory array because drivers for data and address lines can be designed to handle the slightly larger loads with relative ease. The latency of static RAMs used in cache design is of the order of a few processor clock cycles. Given the complexity of the cache controller, the latency of the coherent caches was estimated to be three processor clock cycles. The influence of tag access latency is investigated for latencies of two and four processor clock cycles.

The access latency of DRAMs is of the order of tens of nanoseconds – 60 to 180 (in 1994), depending on size and organization of the memory array. When considering the overhead imposed by the coherency protocol, the latency of the memory was set to 120ns. The influence of memory access latency is investigated for latencies of 80 and 160ns.

The number of processors in a ring imposes an inherent limit on the performance of the machine because each packet must travel, on average, one half of the ring while its echo must complete the round. Thus, in the absence of traffic, the static latency imposed by the ring interfaces already imposes a limit on performance. More processors on a ring imply more traffic and consequently longer round-trip latencies. The ring sizes investigated are 2, 4, 8, and 16. The results for uniprocessors are also included to provide a basis for comparison and to assess the effects of interference amongst cooperating processors.

The clock speed of processors doubles roughly every two years. The generation scalability of SCI-based systems is investigated for one clock frequency doubling from 100MHz to 200MHz. The higher rate of memory references, caused by the faster clock, places an extra burden on the interconnection network, potentially doubling the traffic through it.

There is a complex relationship between these design parameters and their effect on performance. Larger caches yield better hit ratios and a faster rate of execution because of the reduction in stalled cycles. The faster rate of execution also means a faster rate of memory requests which might, in turn, increase the number of compulsory and consistency misses. An increase in the consistency misses implies higher network traffic and latency. Faster caches increase the execution rate without reducing capacity and conflict misses. In the following sections some of these trade-offs are explored.

4.2.2 Characterising the Workload

The execution time breakdown for each of the programs in the workload is presented next. The plots show the contributions by instruction fetch and execution and, private and shared-data references. The graphs show the breakdown by activity for systems with 64, 128, 256 and 512Kbytes secondary caches. The graphs also show the breakdown by activity for infinite caches since, with very large caches, capacity misses do not occur and consistency misses are kept to a minimum. The shared-data hit ratios for the four smaller cache sizes are also shown. These data combined provide a good picture of the applications' behaviours.

Execution Time Breakdown

When assessing the performance of a given architecture, the amount of time programs spend on each type of activity must be quantified. In order to do this for SCI systems, execution time was split into six activities. A program spends time (1) performing instructions, (2) on references to private data, (3) waiting at synchronisation points (barriers and locks), (4) on references to shared-data that is local to the processor, (5) on references to shared-data at another node and, (6) in delays caused by network latency. The fraction corresponding to references to shared-data at another node include all the delays involved in remote protocol actions such as cache/memory tag access latencies, as well as loading/storing data from/to caches and main memory. In the simulations performed, time spent performing instructions and references to private data is independent of network traffic. The fraction of time spent on remote references is given by the sum of the network delays and memory/cache latencies at the remote nodes.

Figure 4.1 shows that on all ring sizes, `chol()` spends over 50% of the time executing instructions and, for ring sizes 2–8, over 20% of the time accessing shared data at the local cache and memory. For the 16-node ring, that falls to about 10%. Thus, `chol()` is memory bound. In Figure 4.1, it can also be seen that `water()` spends over 50% of the time performing instructions and over 25% referencing private data. Although its shared-data hit ratios are not very high, less than 15% of the time is spent on shared data references. Thus, `water()` is processor bound.

Figure 4.1 also shows, for `mp3d()`, that the uniprocessor spends 50% of the time on instructions, 30% on data that would be shared on a multiprocessor, and 23% of the time on private data. These values, on a 16-node ring, fall to 10%, 5% and 5%, respectively. The percentage of time spent on network latency climbs steadily from 0% to just over 45%. Thus, `mp3d()` is memory bound. A puzzling feature of this application is that performance *worsens* with larger caches. This is caused by cache pollution: larger caches hold more data from molecules that "moved away" to another node in previous time steps. For instance, on 8-node rings, the number of sharing lists purged per shared-data miss at the coherent cache increases with cache size: 0.927, 0.962, 0.979, 0.991, 0.999, for 64, 128, 256, 512Kbytes and infinite caches, respectively. The average number of copies purged is 1.1 and is independent of cache size. It seems that the simulator models the multiprocessor system at an adequate level of detail since such a counter-intuitive and complex behaviour has been exposed and can be explained from simulation data.

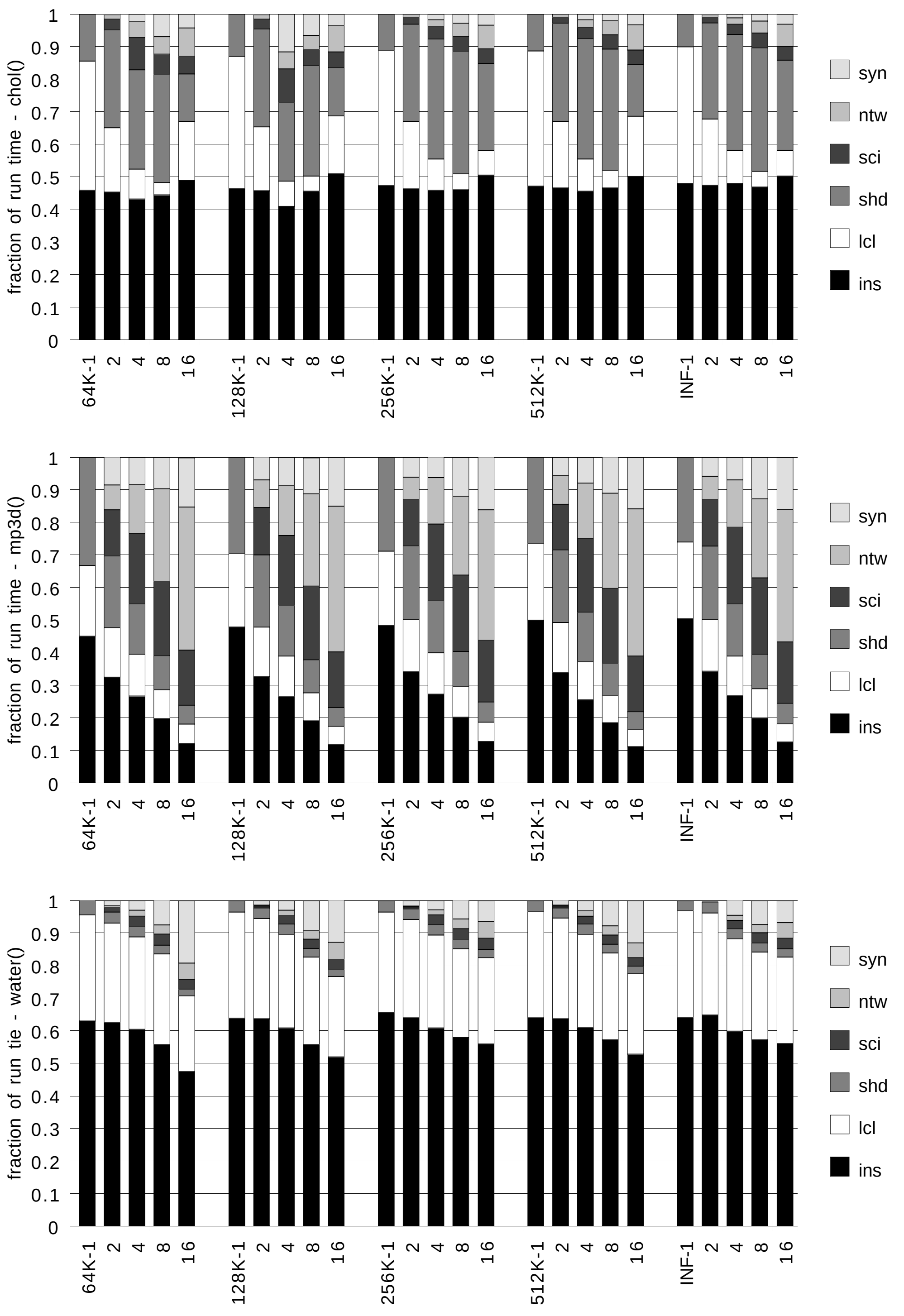


Figure 4.1: Execution time breakdown for `chol()` (top), `mp3d()` (center) and `water()` (bottom). ins stands for instructions, lcl for private data references, syn for synchronisation, shd for local shared-data references, sci to cache and memory latencies on remote references and, ntw for network latency.

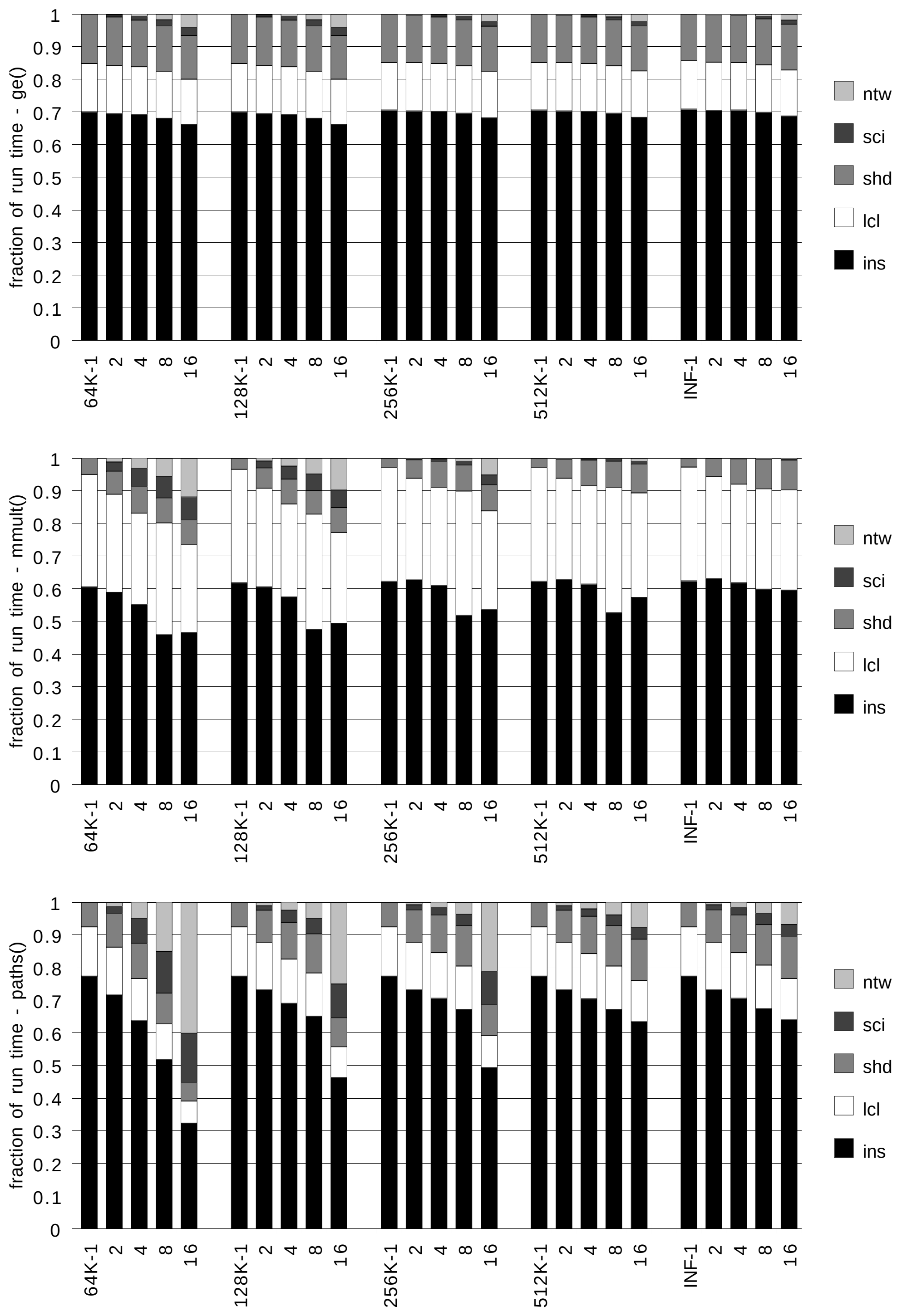


Figure 4.2: Execution time breakdown for `ge()` (top), `mmult()` (center) and `paths()` (bottom). ins stands for instructions, lcl for private data references, shd for local shared-data references, sci to cache and memory latencies on remote references and, ntw for network latency.

`ge()` spends over 67% of the time executing instructions, and 15% on shared data references – see Figure 4.2. Thus, `ge()` is processor bound. `mmult()`, on almost all cases, spends over 50% of the time performing instructions. Since this application has little write sharing, the three larger cache sizes (256, 512Kbytes and infinite) spend little time on remote references. The 8-node rings suffer higher instruction miss ratios then the other system sizes because of conflict misses. `paths()`, on rings of up to 8 nodes, spends over 75% of execution time performing instructions, and about 10% on each of private and shared data references. For the data sets used here and the associated shared-data hit ratios (see below), `paths()` is processor bound. However, if the shared-data hit ratio falls below 90%, `paths()` becomes memory bound. Except from `mp3d()` and `paths()`, the programs are all processor bound. `paths()` is a borderline case: a decrease in the shared-data hit ratio can make it memory bound.

Coherent Cache Hit Ratios

The shared-data read hit ratios of the six programs are shown in Figure 4.3, page 48. See the Appendix for hit ratios of all five types of memory reference. For `chol()`, shared-data hit ratios are always above 90%. `mp3d()` has the worst hit ratios of the workload and the ratios deteriorate with the larger data sets but do not vary significantly with cache size. Even though `water()`'s hit ratios are not very high, it spends less than 15% of the time on shared-data references.

`ge()` has, for all ring (1-16) and cache (64-512Kbytes) sizes, secondary cache hit ratios above 97%, for data and instructions. The shared-data hit ratio of `mmult()` improves with increasing cache size from about 87% (64K) to over 97% (512Kbytes). `paths()`, on the 16-node ring and 256Kbytes cache, has a shared data hit ratio about 7 percentage points lower than on smaller rings and this in turn causes the time spend on network latency to jump from under 5% to 28%. For a 64Kbytes cache, this last value is 47%.

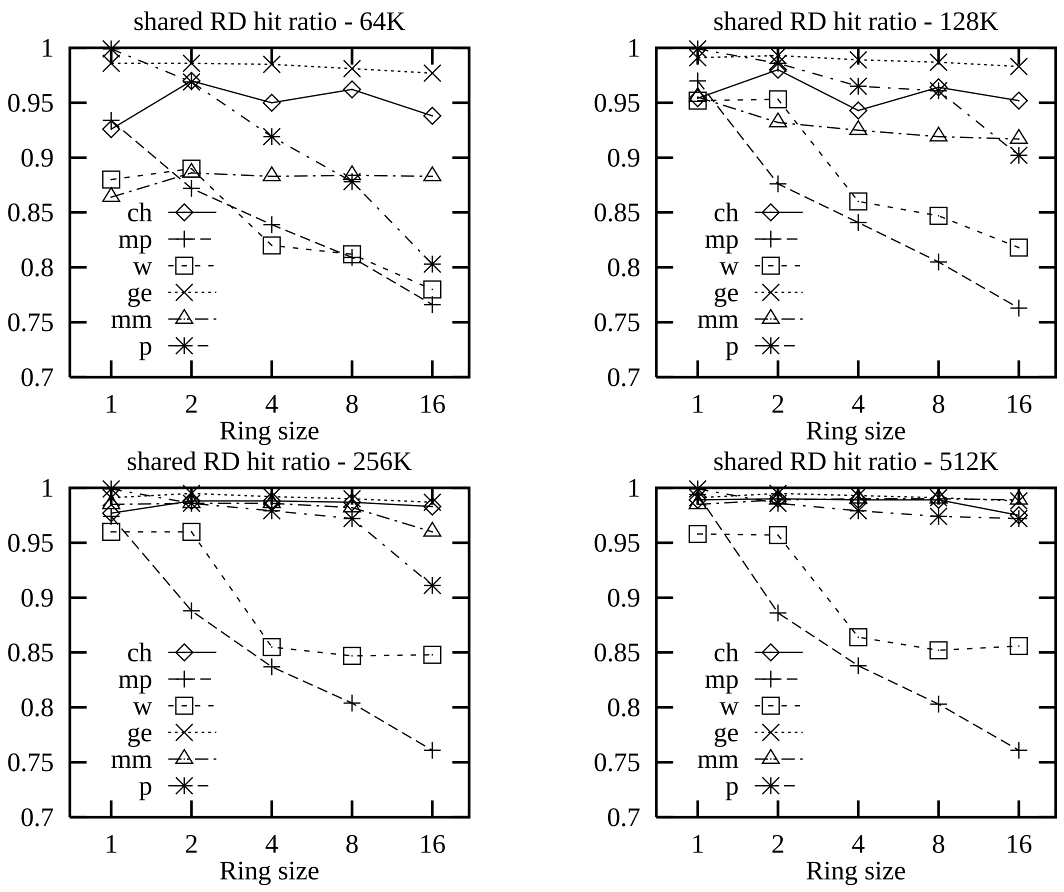


Figure 4.3: Shared-data read hit ratios for 64K (top left), 128K (top right), 256K (bottom left) and 512Kbytes coherent caches (bottom right). 'ch' stands for `chol()`, 'mp' for `mp3d()`, 'w' for `water()`, 'ge' for `ge()`, 'mm' for `mmult()`, 'p' for `paths()`.

4.2.3 Cache Size and Cache Access Latency

Coherent cache size and tag access latency are two of the factors that have most impact on the performance of memory hierarchies. The effect of cache size is examined here. Figure 4.4 (page 51) displays the execution time as a function of ring and cache size for the three SPLASH programs. Recall that the data-set sizes are scaled up to keep the work each processor does constant – see Table 3.2 (page 36). For `chol()`, on a 4-node ring, the 128Kbytes cache is about 35% slower than the two larger sizes. The difference is not as pronounced for the other ring sizes. The 64Kbytes cache being faster than the 128Kbytes is due to an optimisation in `chol()`, by which the supernodes are chosen to fit the coherent caches. For all cache sizes (64-512Kbytes) and ring sizes 2–16, `mp3d()` has shared data hit ratios that are within one percentage point of one another. The same is true of the fraction of run time due to network latency, except that the interval is under 4%. On a 16-node ring, `water()`'s shared data-set does not fit in the 64Kbytes caches. Hence the difference in execution time between the 64K and 128-512Kbytes coherent caches.

Figure 4.5 (page 52) shows the relationship between cache and ring size and speed for the three parallel loops. Recall that the data-set sizes are scaled up with machine size – see Table 3.2 (page 36). For `ge()`, the differences in run time are below 4% and this agrees with the rather small changes in shared data hit ratio with cache size. The performance of the system, when executing `mmult()`, improves with larger cache sizes. The improvement comes from a reduction in conflict misses and network delays. The 8-node machines endure higher instruction miss rates because of conflict misses. As discussed above, `paths()` is a borderline program: if the caches cannot accommodate the working set, the program speed is bound by the speed of the memory and ultimately by the network latency. For the 64Kbytes cache, the impact of the network latency increases dramatically with ring and data set sizes because of the poorer hit ratios.

Sharing-list length. `paths()` has an average sharing-list length that grows roughly as $P/2$, for P processors. The other five programs have sharing-list lengths of one or less for ring sizes 2–8 and under 1.2 for 16-node rings. Sharing-list length is fairly independent of cache size. This is in agreement with [WG89] in that most of the shared-data in `chol()`, `mp3d()` and `water()` is migratory in nature. The same can be said of `ge()`, given its algorithm and simulation statistics. See the Appendix for the sharing-list lengths of all six programs.

Cache Size, Cache and Memory Tag Access Latency

Table 4.1 shows the effects on performance of changing one of the major design parameters while keeping the other two constant. The basis for comparison is a system with 256Kbytes coherent caches with 3 processor cycles of tag access latency, and memory access latency of 120ns. Systems with four and eight nodes were simulated with (1) 128 and 512Kbytes secondary caches, (2) 256Kbytes secondary caches with tag access of two (20ns) and four processor clock cycles (40ns), and (3), memory tag access of 80 and 160ns. The table shows that the factor which has the most influence is the cache tag access latency (between -13% and $+14\%$) while memory access latency has the least influence (between -6% and $+6\%$).

For the workload studied here, caches with 2 processor cycles of access latency yield an average 8.5% speed improvement while, on average, the speed loss can be 8.4% (8.6%) on the 4-node (8-node) ring with a 4-cycle latency coherent cache. The system designers have to weigh the cost increase against the speed gains when specifying memory technology. The plots in Figures 4.4 and 4.5 provide evidence against the use of 64Kbytes secondary caches. The more conservative cache latency of 3 cycles was adhered to for the experiments reported here. Note that values in Table 4.1 follow the pattern of concave curves relating performance to changes in cache design parameters, as discussed by Przybylski in [Prz90].

Nodes	4						8					
change:	c size		c latency		m latency		c size		c latency		m latency	
	128	512	2 cy	4 cy	80	160	128	512	2 cy	4 cy	80	160
`chol()`	1.15	1.01	0.88	1.14	0.98	1.00	1.06	0.99	0.87	1.13	0.97	1.04
`mp3d()`	1.02	0.98	0.92	1.08	0.94	1.06	1.00	1.02	0.95	1.08	0.96	1.05
`water()`	1.00	1.00	0.91	1.10	0.99	1.01	1.03	1.00	0.90	1.09	0.99	1.01
`ge()`	1.01	1.00	0.93	1.07	1.00	1.00	1.01	1.00	0.93	1.07	1.00	1.00
`mmult()`	1.06	0.99	0.93	1.04	0.98	1.02	1.09	0.99	0.93	1.02	0.99	1.01
`paths()`	1.03	1.00	0.94	1.07	0.99	1.01	1.04	1.00	0.94	1.06	1.00	1.01
average	1.04	0.99	0.92	1.08	0.98	1.02	1.04	1.00	0.92	1.08	0.98	1.02

Table 4.1: Sensitivity of execution time to variations in cache size, cache latency and memory latency. The basis is 256Kbytes cache with 3 processor cycles access latency and 120ns memory access latency, respectively. See text for details.

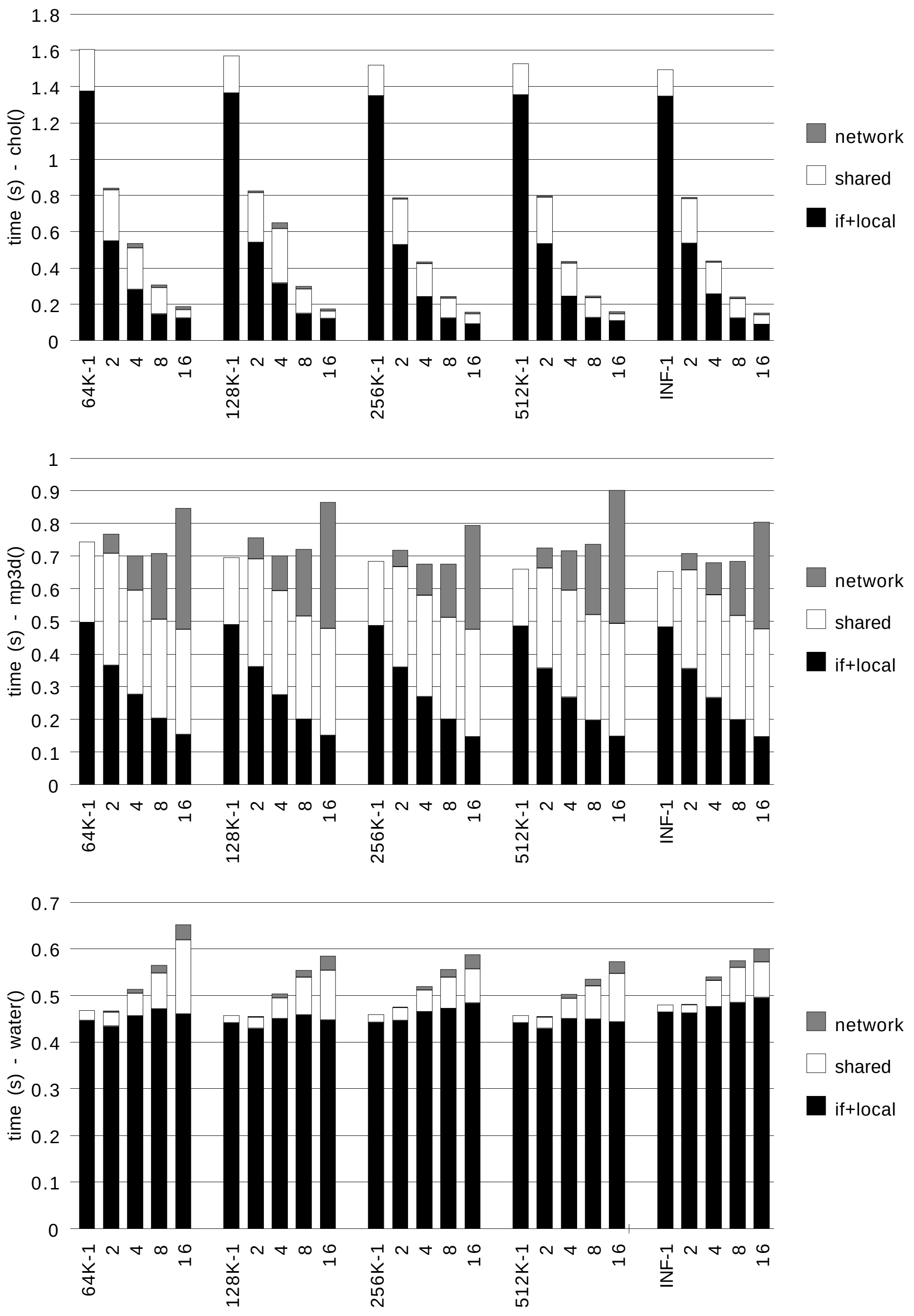


Figure 4.4: Execution time as a function of cache size, for `chol()` (top), `mp3d()` (center) and `water()` (bottom). Time is broken down into network latency, references to shared-data and references to local data and instructions. Data sets are scaled up with machine size.

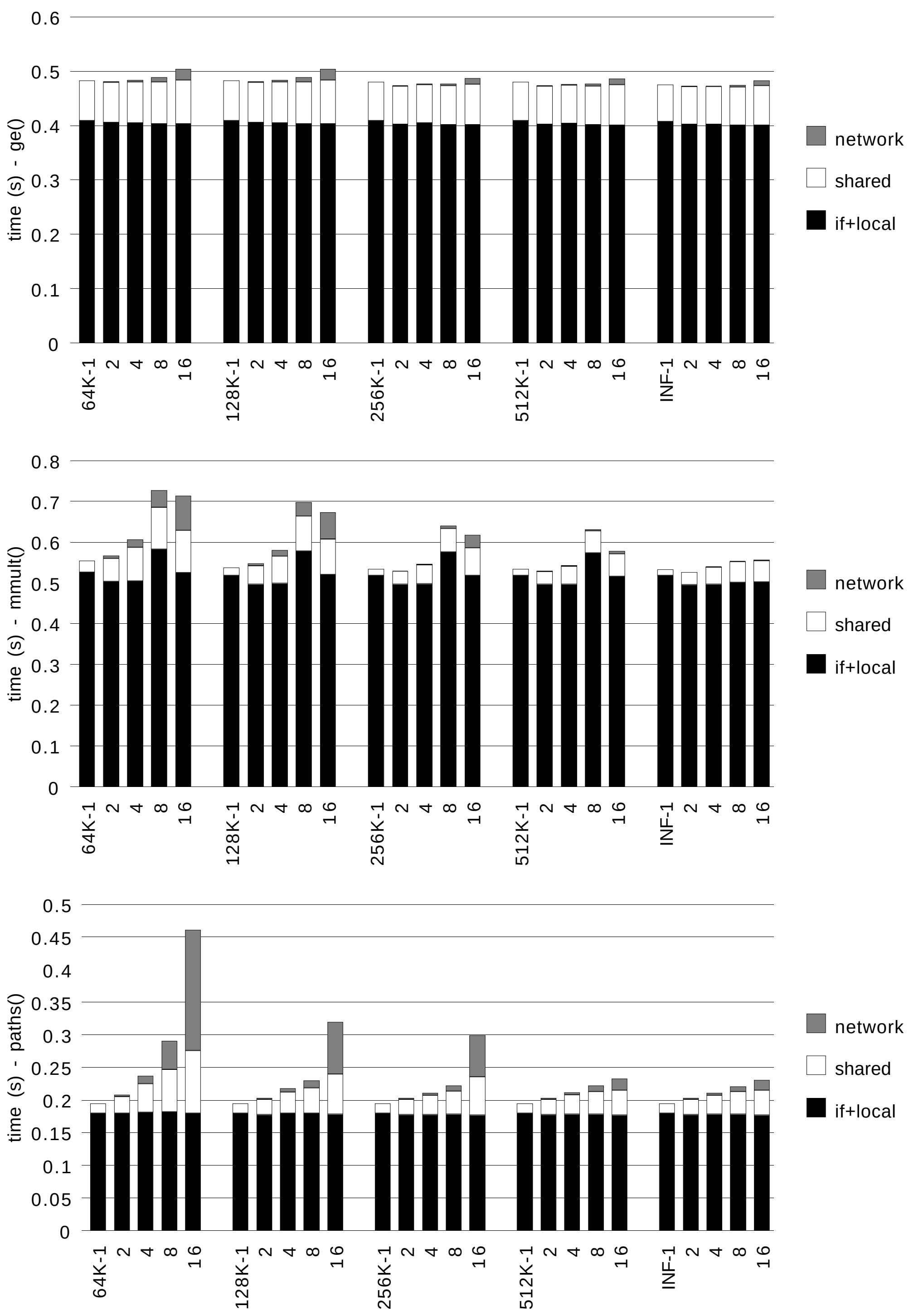


Figure 4.5: Execution time as a function of cache size, for `ge()` (top), `mmult()` (center) and `paths()` (bottom). Time is broken down into network latency, references to shared-data and references to local data and instructions. Data sets are scaled up with machine size.

4.2.4 Processor Clock Speed

Microprocessor technology is evolving at such a pace that the speed of processors, and indeed of workstations, doubles roughly every two or three years. What can be said about the performance of SCI, when the next generation of processors is introduced? Figure 4.6 shows the speedup attained by doubling the processor clock speed while keeping the other parameters unchanged. Note that coherent cache access latency is 3 processor clock cycles in both cases.

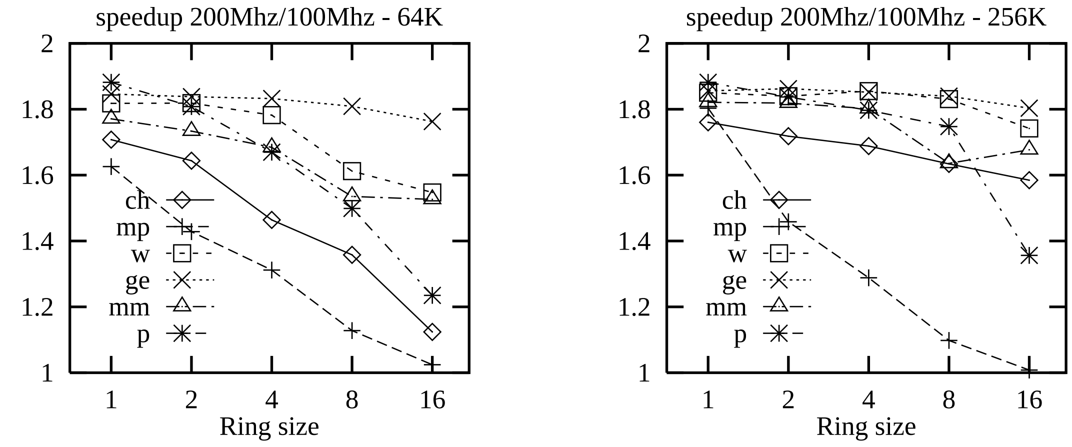


Figure 4.6: Speedup achieved by doubling processor clock frequency, with cache sizes of 64K (left) and 256Kbytes (right). 'ch' stands for `chol()`, 'mp' for `mp3d()`, 'w' for `water()`, 'ge' for `ge()`, 'mm' for `mmult()`, 'p' for `paths()`.

Some of the loss in speedup can be attributed to the relatively slower memory hierarchy, the influence of which can be gauged from the values for the uniprocessor – between about 10 to 37% loss in speedup. As discussed earlier, for a 100MHz clock, an increase of 30% in memory latency slows execution down by up to 6%, `chol()` and `mp3d()` being the worst affected. Most of the loss in speedup for `chol()`, `mp3d()` and `paths()` is caused by network saturation. Plots of the ratio of link traffic for 100 and 200MHz processors are almost identical to those in Figure 4.6. Programs that generate low levels of network traffic can use a lot more bandwidth whereas programs that nearly saturate the ring suffer even higher round-trip delays with a faster rate of network requests.

4.3 Throughput and Latency

The factors that most influence the performance of a network are throughput and latency. Throughput is the amount of data that each processor can inject into the network per time unit. Network latency is the time it takes for a packet to be delivered and acknowledged. The level of traffic on the network is also important because latency increases with traffic since bandwidth is limited.

Node throughput. Figure 4.7 shows the throughput per node, that is, the number of bytes inserted per time unit in the output buffer by the processor and cache/memory controller. Note that the measured throughput includes packet header overhead. Data throughput would be somewhat lower. The reason for including header overheads in the throughput measurement is that cache coherency commands are embedded in the packet headers and these comprise a large fraction of the information transferred by the cache coherence protocol.

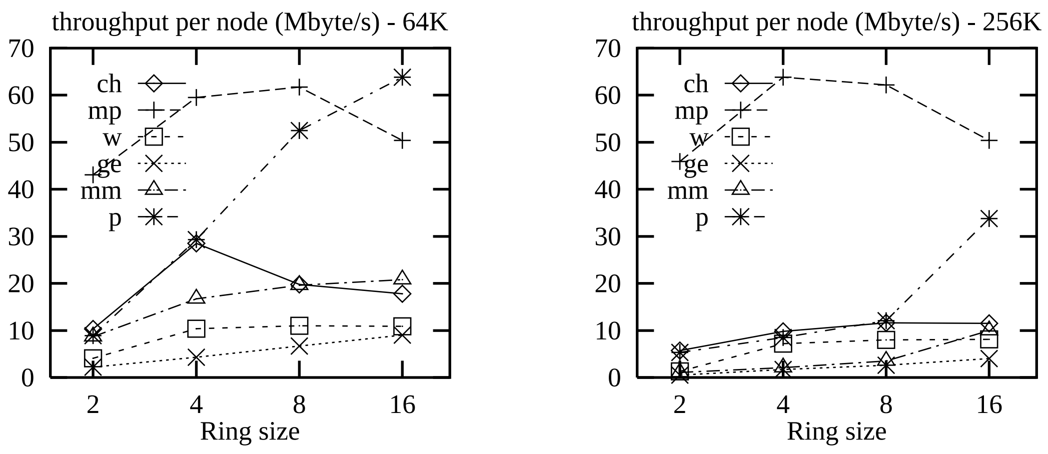


Figure 4.7: Throughput per node, coherent cache sizes of 64K and 256Kbytes. ‘ch’ stands for `chol()`, ‘mp’ for `mp3d()`, ‘w’ for `water()`, ‘ge’ for `ge()`, ‘mm’ for `mmult()`, ‘p’ for `paths()`.

Round trip delay. Figure 4.8 shows the average round trip delay as a function of ring size. This delay is the time elapsed from inserting a packet in the output buffer until its echo is stripped by the sender. Note that latencies experienced accessing memory and caches are not included. The static latency for a 16-node ring is 116ns, for an average packet size of 11 symbols. `chol()`, `water()`, `ge()` and `mmult()` generate low network traffic and enjoy low latencies. `mp3d()` and `paths()` endure much higher latencies because of their higher throughputs and increased network congestion.

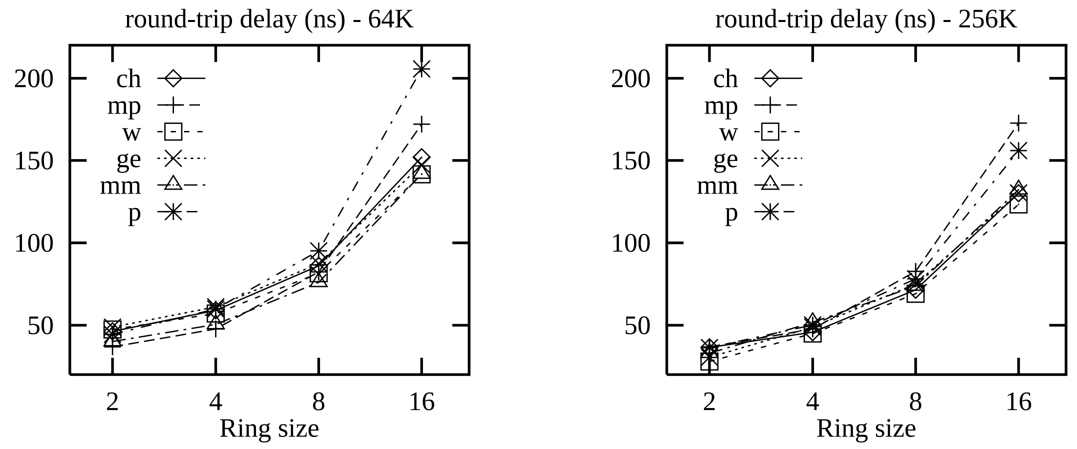


Figure 4.8: Average round-trip delay, with cache sizes of 64K (left) and 256Kbytes (right). 'ch' stands for `chol()`, 'mp' for `mp3d()`, 'w' for `water()`, 'ge' for `ge()`, 'mm' for `mmult()`, 'p' for `paths()`.

Throughput versus latency. It is normally easier to increase bandwidth than to reduce latency, given today's technological constraints. The relationship between node throughput and round-trip delay indicates how well a network design balances latency and bandwidth. The simulation data recorded in this dissertation provide enough points to plot throughput *versus* latency on SCI rings and this is shown in Figure 4.9. From the left, the data points are from `water()`, `chol()`, `paths()`, `mp3d()`, and again, `paths()` and `mp3d()` with 200MHz processor clock frequency.

The plots show a linear relationship between latency l and throughput s for 2-, 4- and 8-node rings. For 16-node rings, that relationship is a parabola with a small quadratic coefficient. The equations that describe ring behaviour, obtained by the least squares method, are given below. The lines defined by the equations are superimposed to the data points in Figure 4.9. Note that these equations are valid for throughputs in the interval $[2, 95]$. Equation 4.5 is the least square fit parabola computed from 16-node rings.

$$
\begin{aligned}
l_2 &= 0.16s + 39.32 && (4.1)\\
l_4 &= 0.14s + 50.82 && (4.2)\\
l_8 &= 0.38s + 73.53 && (4.3)\\
l_{16} &= 1.40s + 119.81 && (4.4)\\
l_{16_2} &= 0.011s^2 + 0.368s + 133.95 && (4.5)
\end{aligned}
$$

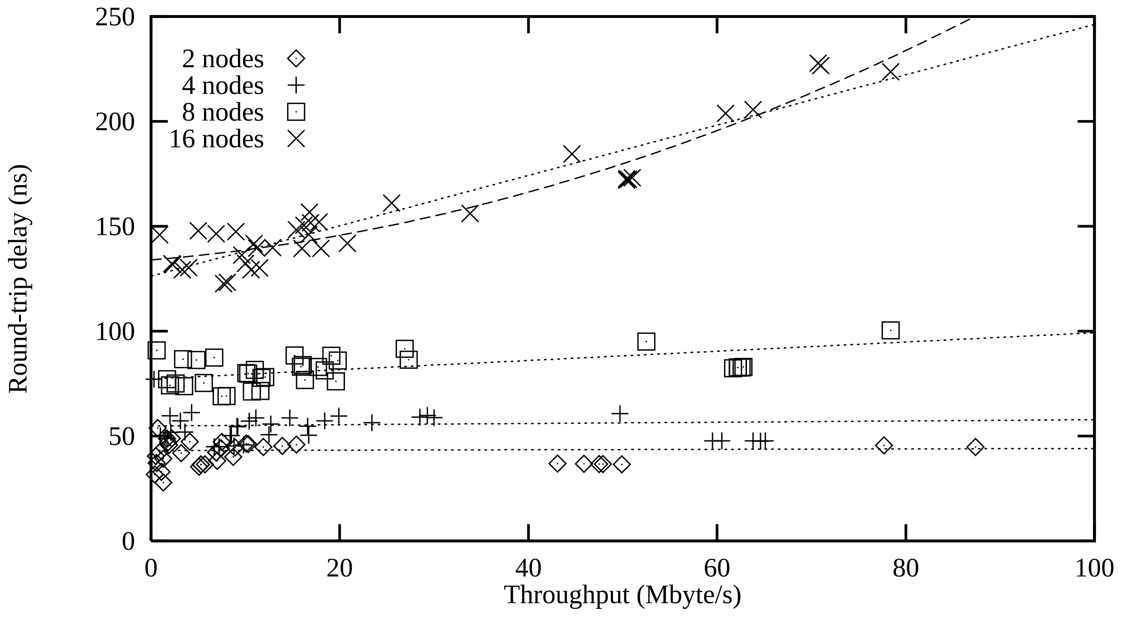


Figure 4.9: Latency *versus* throughput on 2-, 4-, 8- and 16-node rings.

The plots in Figure 4.9 do not show the same behaviour as those obtained with an analytical model by Scott *et.al*, in [SGV92]. There are some differences in the underlying models and assumptions between their model and the simulations discussed in this dissertation. Their model is based on M/G/1 queues and the ring is modelled as an open system: increasing network delays do not decrease the rate of processor requests. Here, the simulator uses 5 packet sizes rather than 3, resulting in average packet sizes of 9–11.5 symbols rather than 12.4. Furthermore, the frequencies of each type of packet are also different: here, data carrying packets account for less than 14% of all injected packets – they assume that 40% of all injected packets are 40-symbol packets. Cache coherence related traffic is also much higher in the simulations here. The ring, as simulated here, does not behave like an open system since processors stall on remote references, and that decreases their rate of network requests. This behaviour is akin to negative feedback. That is why Figure 4.9 does not show the pronounced saturation produced by Scott's model. However, network saturation does occur and its effects can be clearly seen in Figures 4.7, 4.8 and 4.9, as well as in the coefficients of Equations 4.1–4.5. If the rate of network requests were increased by the use of multithreading or weaker memory consistency, saturation effects would tend to be more pronounced because of the potential increases in traffic.

Link traffic. Figure 4.10 shows the traffic per link as a function of ring size. The traffic consists of the packets inserted by a node plus the packets passing through that node towards downstream nodes. `mp3d()` and `paths()` produce high levels of traffic and suffer higher latencies. Traffic levels of 600 to 700Mbytes/s are a limiting factor in the performance of SCI-connected systems since, at these levels, round-trip delays are holding down the rate of network requests by processors. Bypass buffers endure utilisations of over 50% and that leaves few opportunities for injecting packets into the ring. Figures 4.10 and the discussion in Section 4.2.4 (page 53) are clear evidence of the effects of network saturation: doubling processor clock rate does little to improve the performance of programs that are already driving the network into saturation.

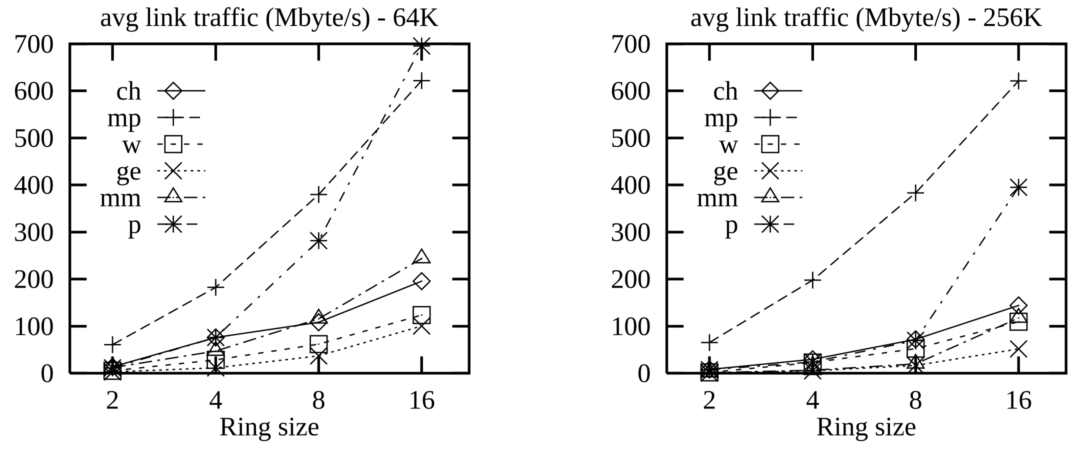


Figure 4.10: Traffic per link, for cache sizes of 64KK (left) and 256Kbytes (right). 'ch' stands for `chol()`, 'mp' for `mp3d()`, 'w' for `water()`, 'ge' for `ge()`, 'mm' for `mmult()`, 'p' for `paths()`.

Average packet size and distribution. The average packet size varies from 18.0 to 22.4 bytes, smaller rings carrying larger packets. Also, smaller caches generate more of the smaller packets that carry the cache coherency commands. Figure 4.11 shows the distribution of packet sizes for `chol()`, `mp3d()` and `water()` with 256Kbytes coherent caches. Data carrying packets account for less than 7% of all packets in 8- and 16-node rings. Thus, data throughput accounts for 20 to

30% of raw throughput. With the exception of echoes, all packets carry cache coherency information and any network delays faced by these packets slow down all operations that involve shared data. Figure 4.12 plots average packet size as a function of cache size. Notice that `pkt48` are 'cache-write' packets that carry the current copy of lines between sharing caches.

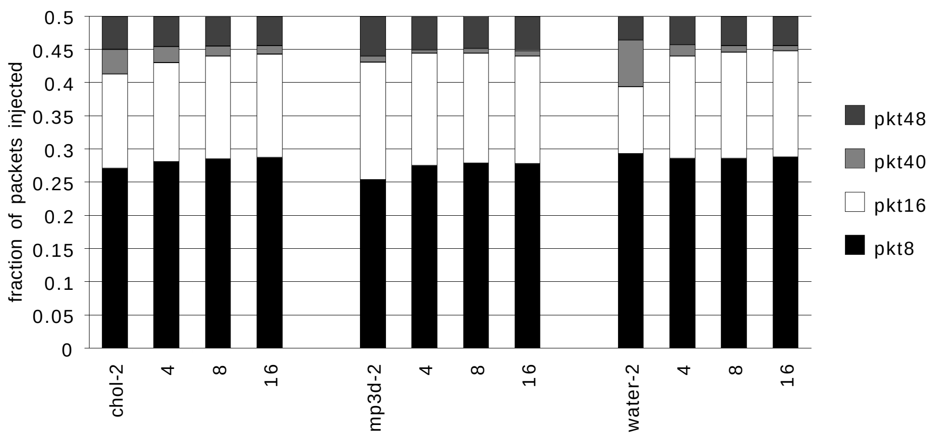


Figure 4.11: Distribution of packet sizes for 256Kbytes caches.

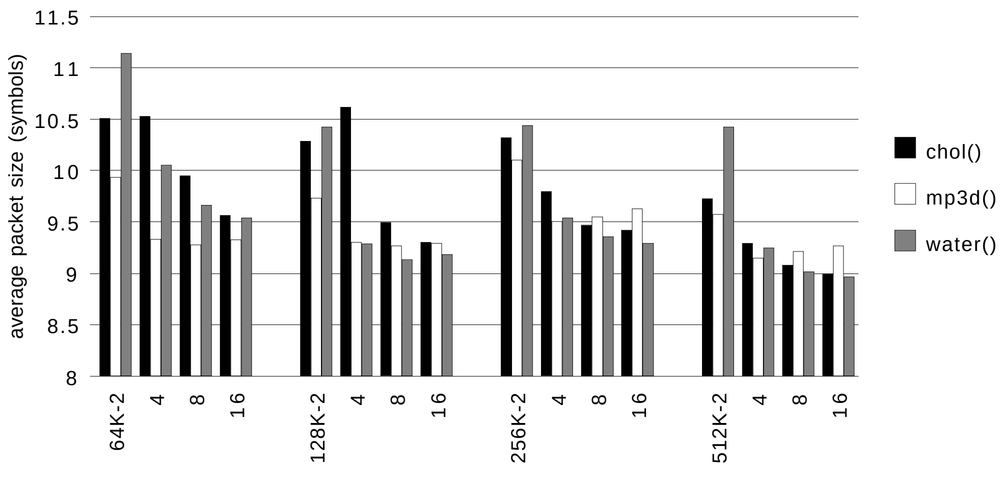


Figure 4.12: Average packet size for 64K (left), 128K, 256K and 512Kbytes (right) caches.

4.4 Other Ring-based Systems

In order to compute the cost of a remote transaction, memory and cache tag access latencies must be added to the round-trip delay. For the simulations reported here, the worst case is a cache-to-memory transaction: *ring latency*$+246ns$ ($30ns+16ns$ plus $120ns + 80ns$). The best case is a cache-to-cache transaction, such as an invalidate transaction, costing *ring latency*$+60ns$ ($2\times 30ns$). Barroso and Dubois, in [BD93], present simulation results for the Express Ring. The multiprocessor's interconnect is based on a slotted ring and cache coherence is maintained by a snooping protocol [BD91]. On a ring with 8 nodes, the shared-data miss latency for `chol()`, `mp3d()` and `water()` is between 280 and 320ns. On a 16-node ring, between 320 and 380ns and, on a 32-node ring, between 390 and 440ns. On 8-node rings, the shared-data miss latencies of an SCI ring are comparable to those of a slotted ring. On 16- and 32-node rings, the SCI ring would have higher latencies.

A comparison with the KSR1 is difficult to make for lack of performance data on the applications employed here. It is likely the results would show the same broad tendencies as those of DASH since the two machines are built from similar technologies – SCI's faster network would provide a performance advantage. The Hector multiprocessor [VSLW91], using a hierarchical snooping protocol [FVS92, HS94] should have a performance comparable to that of the Express Ring. Holliday and Stumm report in [HS94], that Hector's hierarchical protocol scales well to a large number of processors (1024) if the applications possess good locality characteristics.

4.4.1 Comparing DASH and SCI

The Directory Architecture for SHared memory (DASH) multiprocessor was conceived at Stanford University as a workbench for exploring the design of logically-shared physically-distributed memory multiprocessors [LLG$^+$90,GGH91, GHG$^+$91]. A DASH prototype was built and its implementation and performance is discussed by Lenoski *et al.* in [LLJ$^+$92]. The availability of performance data makes possible a comparison between DASH and an SCI-based parallel machine with similar architectural parameters. However, because of intrinsic differences in architecture and run time environments, *i.e.* simulation compared to an actual machine, strict quantitative comparisons would be misleading. A qualitative comparison can nevertheless be informative. The two architectures are described below, followed by the performance comparison.

The DASH architecture. DASH consists of clusters of processing nodes interconnected by twin meshes. The clusters are bus-based multiprocessors within which cache consistency is maintained by a snooping protocol. Inter-cluster consistency is maintained by a full-directory invalidation protocol [LLG$^+$90]. Each cluster contains four processors; each processor is connected to a 128Kbytes split primary cache (64Kbytes for instructions and 64Kbytes for data, write-through) and a 256Kbytes write-back secondary cache. Both caches are direct mapped and support 16-byte lines. The interface between primary and secondary caches consists of a 4-word deep write-buffer and a one-word read-buffer. The intra-cluster cache coherence protocol allows for cache-to-cache transfers. This makes the four secondary caches appear as a single (cluster) cache to remote nodes. Processor clock speed is 33MHz. The clusters also contain memory, the memory directory and inter-cluster communication interfaces. The interconnection network consists of two wormhole routed networks, one for requests and one for responses. The latency through each node (network hop) is about 50ns. The peak bandwidth is 120Mbytes/s in and out of each cluster.

The SCI ring. The SCI ring was simulated with only one processor per node, with the same clock speed and cache hierarchy as DASH. The primary cache is split (64K + 64K), direct mapped, write-through. Secondary caches are 256Kbytes, direct mapped, write-back. Lines are 64 bytes wide. There is *no* write-buffer between the caches. Cache and memory latencies per word are the same as those in DASH – see Table 4.2. Notice that on the SCI ring, the latencies vary with ring size and, in the case of writes, the number of copies of the line. The processing nodes are interconnected by an SCI ring.

Cache operation	DASH	SCI ring
Read from primary cache	1	1
Fill from secondary cache	14	14
Fill from local memory	26	26
Fill from remote node	72	31–37
Fill from dirty-remote, remote home	90	49–60
Write owned by secondary cache	2	14
Write owned by local node	18	14
Write owned by remote node	64	63–207
Write to dirty remote, remote home	82	63–207

Table 4.2: Cache and memory operation latencies, in processor clock cycles.

Some of the architectural differences can make a direct comparison difficult. The clustering of processors in DASH has the potential to significantly reduce the latencies of operations that involve only local processors. DASH's write-buffers, although memory is sequentially consistent, do hide some of the write latencies. The SCI ring, on the other hand, provides about four times higher bandwidth than DASH's twin meshes. Static network hop latency in the SCI ring is much smaller as well. It's not unlikely that these differences might cancel each other out, depending on workload characteristics. Recall that page faults are assumed to have a cost of zero for the SCI ring and that the simulated processors never stall because of internal data dependencies and pipeline bubbles, for example. These would tend to give the SCI multiprocessor an unfair advantage.

The workload for the comparison consists of `chol()`, `mp3d()` and `water()`. Data sets are, for `chol()` `bcsstk14`, for `mp3d()` 40000 molecules simulated over 5 time steps, and for `water()` 343 molecules simulated for two steps. In [LLJ$^+$92], Water is run with 512 molecules; the simulation time for that many molecules is so long as to be prohibitive. Thus, `water()` was simulated with a smaller data-set. The comparison should still be valid since, in the 16-node case, `water()` makes 2.6 million references per processor to shared data and that is enough to fill up the secondary caches at least a few times.

Speedup. Figure 4.13 shows the speedup plots for both DASH and the SCI ring. Speedup data for DASH was taken from Figure 6 and Table 5 in [LLJ$^+$92]. The plot shows that `chol()` has a very similar speedup in both architectures. The differences for 2 and 4 nodes are most likely related to better data mapping in the SCI ring. The differences on `mp3d()` are more pronounced. Network traffic is much higher than with the other two programs and, given SCI's higher bandwidth and lower latencies, it is not surprising that `mp3d()` scales up better in the SCI ring. For `water()`, SCI's advantage comes partly from the better network, partly from the smaller number of molecules and the resulting improvement in hit ratios. In DASH with 16 processors, `water()` uses 4.6% and 5.3% of available bandwidth for the request and response networks respectively. In the SCI ring, it uses less than 1% of the bandwidth. Also, the smaller latencies, coupled with the lower hit ratios on larger machines, give SCI a definite advantage. In these programs, data is mostly migratory. SCI's linear latency when purging long sharing-lists is only felt in synchronisation actions. Those however are infrequent when compared to data references.

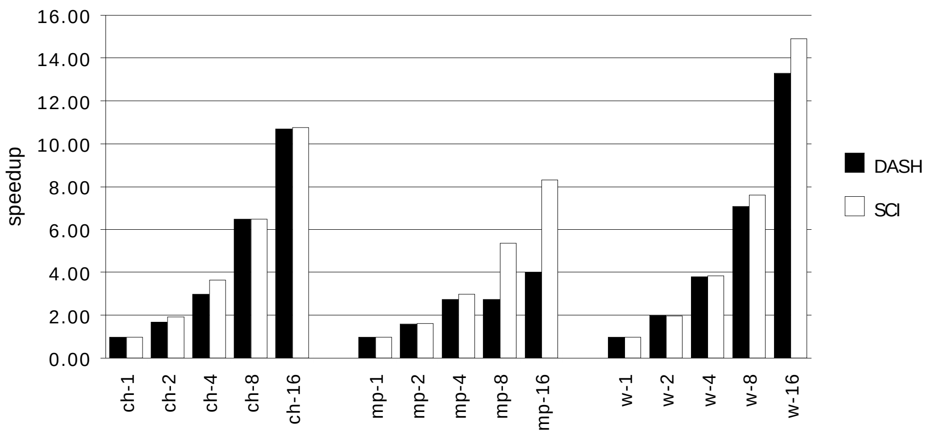


Figure 4.13: Speedup plots for `chol()` (left), `mp3d()` (mid) and `water()` (right).

Ring size	1	2	4	8	16
		Cholesky – `chol()`			
read refs. (10^3)	8444	9883	6875	4638	2904
write refs. (10^3)	1862	2939	2075	1191	556
RD hit ratio	0.97	0.98	0.98	0.98	0.97
WR hit ratio	0.98	0.98	0.98	0.99	0.98
run time (s)	10.26	5.32	2.80	1.58	0.95
throughput (Mbytes/s)	0	1.18	2.06	2.54	2.69
		MP3D – `mp3d()`			
read refs. (10^3)	3619	3755	1594	1041	1098
write refs. (10^3)	2343	1171	585	293	147
RD hit ratio	0.95	0.89	0.85	0.82	0.77
WR hit ratio	0.95	0.89	0.85	0.81	0.74
run time (s)	4.83	2.96	1.61	0.90	0.58
throughput (Mbytes/s)	0	9.5	15.8	19.9	20.6
		Water – `water()`			
read refs. (10^3)	20172	12619	8163	4093	2414
write refs. (10^3)	3071	1536	768	384	192
RD hit ratio	0.96	0.96	0.88	0.84	0.83
WR hit ratio	0.99	0.99	0.95	0.93	0.92
run time (s)	41.96	21.20	10.90	5.50	2.81
throughput (Mbytes/s)	0	0.13	1.02	1.63	1.80

Table 4.3: Per node shared-data reference counts, secondary cache hit ratios, execution time and node throughput on the SCI ring.

In order to evaluate the effect of the network latencies in the performance of the 16-node SCI rings, the three programs were simulated on a 4x4 SCI mesh. This resulted in an improvement of only 2% for `mp3d()` and virtually no change for the other two. Were the traffic higher, the effects on performance would have been much more pronounced. The performance of SCI meshes is further investigated in Section 6.2.

Execution time breakdown. Figure 4.14 shows the execution time breakdown for the workload on the SCI ring. Time is split into: (1) busy time when the processor is performing instructions and operations on data; (2) the time wasted on read misses (RDmiss); (3) the time wasted on write misses (WRmiss); (4) the time spent on synchronisation actions (synchr). The plot also shows the total time spent on references to shared data (shared data) and the time lost because of network latencies (network). The plots indicate that most of the stall cycles come from writes. This is partly because of the high latency of a write (14 cycles) and partly a "normal" feature of cache-coherent shared-memory multiprocessors [GGH91, GHG+91]. The fraction of the time used up by `chol()` in references to shared data increases with ring size because of the relatively higher costs of remote references while the fraction due to instruction fetches becomes proportionally more important as the work per processor decreases. `mp3d()` spends 57–43% of the time on private references, a large fraction of which is on write misses. The fraction of time `water()` spends on shared data references is very small and most of the write-stalls are caused by private data misses.

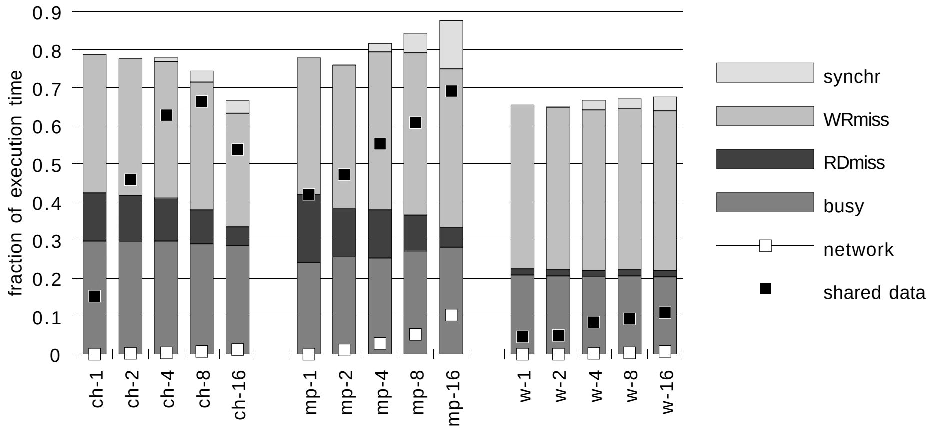


Figure 4.14: Execution time breakdown for `chol()` (left), `mp3d()` (center) and `water()` (right). The lines show the fraction of shared data references and network latency.

A more detailed view of the execution time breakdown is given in Figure 4.15 which shows the normalised breakdown of time spent on references to shared-data only. Over 40% of the time is spent waiting for writes to complete. Synchronisation is a significant activity for `mp3d()` on a 16-node ring. For `water()`, synchronisation takes up about 30% of the time spend on shared-data references but the overall impact is negligible because of the relative cost of each class of reference. The contribution of network latency to `mp3d()`'s execution time is 1, 3, 5 and 10% for 2, 4, 8 and 16 nodes respectively. For both `chol()` and `water()`, network latency takes up less than 1% of the time.

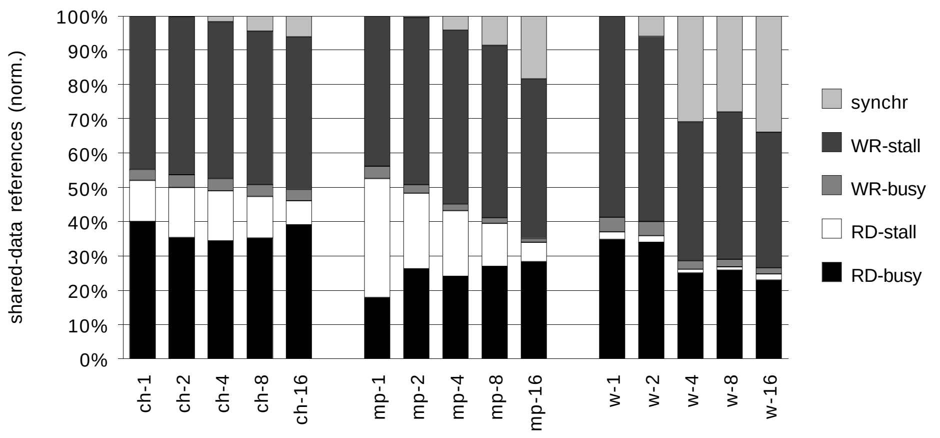


Figure 4.15: Normalised execution time breakdown for `chol()` (left), `mp3d()` (mid) and `water()` (right), for shared data references.

Figures 4.16 and 4.17 show the execution time broken down by type of memory operation for `chol()`, `mp3d()` and `water()` with the architectural parameters used in the rest of this chapter. Compare these to Figures 4.14 and 4.15. Because of the smaller and varying data sets sizes, the instruction-related fraction is relatively larger. Note that both cache and memory access latencies are shorter in Figures 4.16 and 4.17. Also, the primary caches are 8 times smaller.

The most striking differences are the increases in the rate of memory requests and network latency and the consequent increase in the cost of shared-data references. Figure 4.16 highlights, for `mp3d()` in particular, the relationship between network traffic and overall performance. The line for the relative cost of shared data references is roughly parallel to the line for network latency. That is, the relative increase in the cost of shared references, as ring size grows, is caused mostly by higher network traffic and longer latencies. This is not the case, however, for `chol()` because of its fixed-size data set.

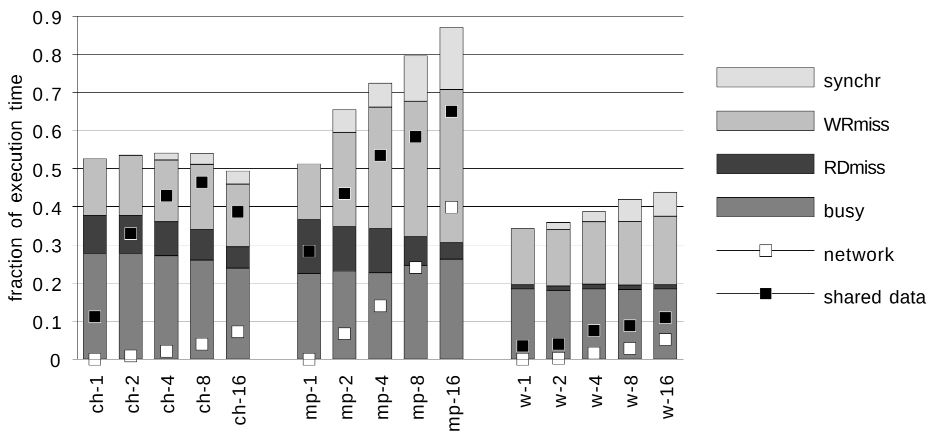


Figure 4.16: Execution time breakdown for `chol()` (left), `mp3d()` (mid) and `water()` (right) – 100MHz clock. The lines show the fraction of shared data references and network latency.

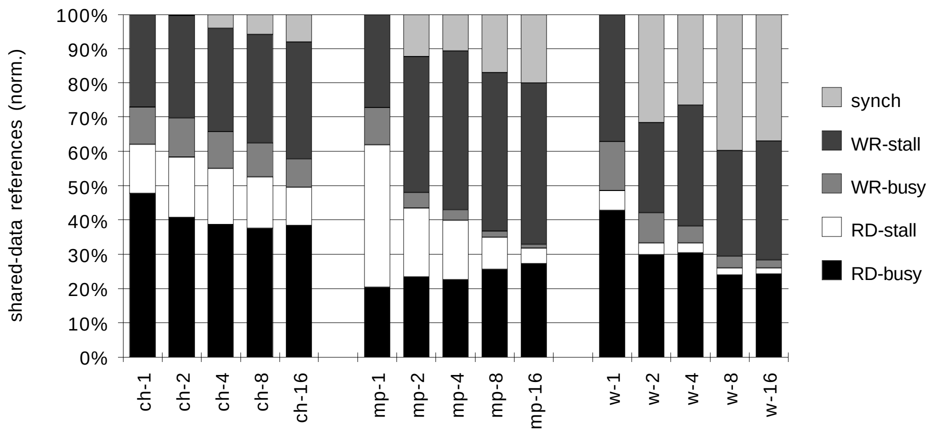


Figure 4.17: Normalised execution time breakdown for `chol()` (left), `mp3d()` (mid) and `water()` (right), for shared-data references – 100MHz clock.

Within the limits imposed by the workload employed, the comparison of the two architectures – coherence protocols and interconnects – indicates that SCI's higher network bandwidth and lower latencies compensate for any advantage that DASH's coherence protocol may offer. It is likely that this conclusion would hold for the less restrictive memory consistency models as well. However, the small sharing sets in the applications do not fully expose SCI's potential bottleneck of purging long sharing-lists serially.

Coda

The detailed investigation of the behaviour of SCI rings in this chapter studied the effects of the inherent limitations of the network, namely the linear static network latencies and decreasing throughput with high levels of traffic. These, combined with the cache coherence protocol place a hard limit on ring size, for a given processor and memory hierarchy. In order to scale up machine size, higher dimensional networks must be employed. The performance of SCI-meshes and SCI-cubes is the subject of Chapter 6. Performance evaluation studies based on simulation provide detailed and accurate results but at high computational costs. When extreme precision is not the main constraint, analytical models can provide reasonably accurate predictions very quickly. An analytical model of the simulated machine is presented next.

Chapter 5

A Model of the SCI-connected Multiprocessor

This chapter presents an analytical model for the performance of the SCI-based multiprocessor. The model is based on the iterative method proposed by Menascé and Barroso in [MB92]. Inputs to the model are the number of processors, reference counts, primary and secondary cache miss ratios, line flush ratio, sharing-list purge ratio and sharing-list length. The model yields the execution time of the program described by the model inputs. The architectural parameters embedded in the model are the same as in the simulation model employed in Chapter 4. The cost of the cache coherence operations is computed from the model inputs and a simplified cost model that estimates the number and size of packets injected into the ring. The throughput *versus* delay model of the SCI ring is described by Equations 4.1 to 4.4.

5.1 The Analytical Model

This section defines the model equations and relates them to the simulation model described in Chapter 3. The execution time is computed as follows. For each type of memory reference (instruction fetch *if*, local data read *lrd*, local data write *lwr*, shared data read *srd* and shared data write *swr*), the number of references *XXcnt* to each level of the memory hierarchy is computed from the individual reference counts and the miss ratios up to that level. Thus, ignoring for the moment references to remote locations, the time taken by each type of reference, for primary cache miss ratio *pcm*, coherent cache miss ratio *ccm*, is

$$\begin{aligned} \mathit{ifCost} &= \mathit{ifcnt}(\mathit{pClk} + \mathit{pcm}_{\mathit{if}} \cdot \mathit{fillpC} + \mathit{pcm}_{\mathit{if}} \cdot \mathit{ccm}_{\mathit{if}} \cdot \mathit{fillcC}) && (5.1) \\ \mathit{lrdCost} &= \mathit{lrdcnt}(\mathit{pcm}_{\mathit{lrd}} \cdot \mathit{fillpC} + \mathit{pcm}_{\mathit{lrd}} \cdot \mathit{ccm}_{\mathit{lrd}} \cdot \mathit{fillcC}) && (5.2) \\ \mathit{lwrCost} &= \mathit{lwrcnt}(\mathit{pcm}_{\mathit{lwr}} \cdot \mathit{wrTrupC} + \mathit{pcm}_{\mathit{lwr}} \cdot \mathit{ccm}_{\mathit{lwr}} \cdot \mathit{wrTrucC}) && (5.3) \\ \mathit{srdCost} &= \mathit{srdcnt}(\mathit{pcm}_{\mathit{srd}} \cdot \mathit{fillpC} + \mathit{pcm}_{\mathit{srd}} \cdot \mathit{ccm}_{\mathit{srd}} \cdot \mathit{fillcC}) && (5.4) \\ \mathit{swrCost} &= \mathit{swrcnt}(\mathit{pcm}_{\mathit{swr}} \cdot \mathit{wrTrupC} + \mathit{pcm}_{\mathit{swr}} \cdot \mathit{ccm}_{\mathit{swr}} \cdot \mathit{wrTrucC}) && (5.5) \end{aligned}$$

Where *pClk* is the processor clock period, *fillpC* is the cost of bringing a line from the coherent cache into the primary cache, *fillcC* is the cost of bringing a line from local memory to the coherent cache and, *wrTrupC* and *wrTrucC* are the costs of writing through the primary and coherent caches respectively.

To each of the above, the time to perform remote memory references must be added. For local data accesses, remote references occur on conflict misses when a line with shared data must be flushed from the coherent cache. For shared data, remote references occur on conflict misses when other shared data must be flushed, on capacity and compulsory misses when new data must be brought into the cache, and on shared data updates when copies must be invalidated before the write to memory can take place. Since instruction references attain very high hit ratios, the effects of remote references caused by instruction fetch misses can be safely ignored.

The cost of remote references can be split into two components. The first is the time taken by cache and memory tag-access latencies. The second component is the network latency which depends on the level of traffic and on the number of packets exchanged in remote cache/memory transactions.

In order to simplify the model, it is assumed that shared data is uniformly distributed amongst the nodes and that the location of the head of a sharing list is independent of the location of its home memory. The probability that the head of a sharing list is at a given node (*hdHere*) is assumed to be inversely proportional to the length of the sharing-list. For N nodes, the probability that the home memory of a given line is the same node from whose cache the line is being flushed/purged is *homeHere*. However, on some cache operations (*e.g* a read to a remote line), the head of the sharing-list cannot be at the node where the operation is taking place (normally a requester). In these cases, the probability that the line is at its home node is $(1/(N-1))$. Since the line is not at the requester's cache, there are $(N-1)$ other places where it could be.

$$\begin{aligned} \mathit{hdHere} &= 1/(\mathit{shLstL}+1) && (5.6)\\ \mathit{hdAway} &= 1-\mathit{hdHere} && (5.7)\\ \mathit{homeHere} &= 1/N && (5.8)\\ \mathit{homeElsw} &= 1-\mathit{homeHere} && (5.9)\\ \mathit{hdAtHome} &= 1/(N-1) && (5.10)\\ \mathit{hdNotAtHome} &= 1-\mathit{hdAtHome} && (5.11) \end{aligned}$$

On a conflict miss caused by local data references, the shared line has to be flushed from the cache. This is accomplished by sending control messages to the nodes towards the head and tail of the sharing list. It is assumed that one message is always sent, thus ignoring the cases where the line is not being shared. If the line is the head of a sharing list (*hdHere*), the line's contents have to be sent to the next node in the list or written back if the line is dirty. A control packet is sent out only if the home of the line is some other node.

$$\begin{aligned} \mathit{xTagAccs} &= \mathit{flushRt}\cdot\mathit{hdHere}\cdot\mathit{fillcC} && (5.12)\\ \mathit{xTrips} &= \mathit{flushRt}(1+\mathit{homeElsw}(\mathit{hdHere}+\mathit{hdAway})) && (5.13)\\ \mathit{xSymb} &= \mathit{flushRt}(16+\mathit{homeElsw}(\mathit{hdHere}\cdot 48+\mathit{hdAway}\cdot 16)) && (5.14)\\ \mathit{xRem} &= \mathit{rtDelay}\cdot\mathit{xTrips} && (5.15)\\ x &\in \{\,\mathit{lrd},\mathit{lwr}\} \end{aligned}$$

where *flushRt* is the number of lines flushed per coherent cache miss, *rtDelay* is the round-trip delay, *xTrips* is the number of round trips needed for that transaction and, *xSymb* is the number of symbols injected into the ring. *xSymb* will be used later to estimate the level of traffic on the ring. *xRem* is the cost of the remote operations for reference type x and *xTagAccs* is the latency of cache and memory tag/data accesses.

The cost of a shared-data read-miss is computed as above but the cache coherence protocol actions are different. For a more detailed description of the protocol actions, see Section 2.3.2 (page 18). If there is a conflict miss, flush the line as above, otherwise fetch the missing line: (1) send a request to the home memory of the line, with probability *hdHere* of it being the at same node. (2) If the line is at its home (*hdAtHome*), the remote SCI controller reads the line from memory and (3a) sends it over the network; (4a) the local SCI controller writes the data to the coherent cache. If the line is not at the home node, the home SCI controller sends back the address of the head of the sharing list (3b); the requester then asks its owner for a copy from the line (4b); the owner's SCI controller reads the line from its coherent cache (5b) and sends it over to the requester (6b); the local SCI controller writes the line to its cache (7b). The cost of looking up a line at the memory directory is *mTagAccs*. The cost of a remote shared data read is given by *srdRem*.

$$\begin{aligned}
srdTagAccs &= homeElsw \cdot hdNotAtHome \cdot mTagAccs + \\
&\quad flushRt \cdot hdHere \cdot fillcC && (5.16)\\
srdTrips &= homeElsw \cdot 2 + homeElsw \cdot hdNotAtHome \cdot 2 + \\
&\quad flushRt(1 + homeElsw(hdHere + hdAway)) && (5.17)\\
srdSymb &= homeElsw(8+40) + homeElsw \cdot hdNotAtHome(16+16) + \\
&\quad flushRt(16 + homeElsw(hdHere \cdot 48 + hdAway \cdot 16)) && (5.18)\\
srdRem &= rtDelay \cdot srdTrips && (5.19)
\end{aligned}$$

In SCI, a write to a shared datum entails the invalidation of all its copies. The SCI controller at the writing processor sends an invalidation message to each owner of a copy thus purging the copies from other caches. After the sharing-list has been purged, the write can proceed. For a detailed description of the protocol actions, see Section 2.3.2 (page 18). The sequence of actions is: (1) the writer requests exclusive ownership of the line to its home memory; (2a) if the line is clean and not shared, the home controller acknowledges the ownership request with a control packet (3a). If there are copies of the line, the home memory responds with the address of the current owner (2b). The writer sends a write-exclusive request to the owner (3b) who reads the line from its cache (4b) and (5b) returns both the valid copy and the address of the next node in the sharing-list (if any). The writer sends an invalidation packet (6b) to the next node in the sharing list, who (7b) reads from its cache the address of the next node in the list and (8b) returns that address to the writer. Actions (6b), (7b) and (8b) are repeated until all copies have been purged. The writer then updates its exclusive copy (9b). The number of sharing-lists purged per shared write miss is *purgeRt*. The average number of copies purged is *shLstL*. The cost of a remote write is then

$$\begin{aligned}
swrTagAccs &= mTagAccs + hdNotAtHome \cdot fillcC + \\
&\quad purgeRt \cdot shLstL \cdot cTagAccs + \\
&\quad flushRt \cdot hdHere \cdot fillcC && (5.20)
\end{aligned}$$

$$
\begin{aligned}
\mathit{swrTrips} &= \mathit{homeElsw}(2 + \mathit{hdNotAtHome} \cdot 2) + \\
&\quad \mathit{purgeRt} \cdot \mathit{shLstL} \cdot 2 + \\
&\quad \mathit{flushRt}(1 + \mathit{homeElsw}(\mathit{hdHere} + \mathit{hdAway})) && (5.21) \\
\mathit{swrSymb} &= \mathit{homeElsw}((8+16) + \mathit{hdNotAtHome}(16+48)) \\
&\quad \mathit{purgeRt} \cdot \mathit{shLstL}(16+16) + \\
&\quad \mathit{flushRt}(16 + \mathit{homeElsw}(\mathit{hdHere} \cdot 48 + \mathit{hdAway} \cdot 16)) && (5.22) \\
\mathit{swrRem} &= \mathit{rtDelay} \cdot \mathit{swrTrips} && (5.23)
\end{aligned}
$$

The execution time is obtained from the above equations, by adding up the time taken by each of the reference types. Round-trip delay is initially estimated from the network traffic, that is, from the number of symbols injected into the ring. Its final value is obtained by an iterative method in which the cost equations are re-computed each time a better estimate for the round-trip delay has been obtained. As a first approximation, the throughput S_0 is set to the number of bytes injected divided by the cost of the references. That underestimates the execution time since network latency is ignored. The round-trip latency $\mathit{rtDelay}_1$ is then estimated with Equations 4.1 to 4.4 (function *ring()* in Equation 5.28). Time t_1 is obtained from equations 5.15, 5.19 and 5.23, but using $\mathit{rtDelay}_1$ for *rtDelay*. This process is repeated until the difference between two iterations is less than 0.1%. Convergence is achieved after 5 to 20 iterations. *xRaccs* is the number of remote references for operation *x*.

$$
\begin{aligned}
\mathit{xRaccs} &= \mathit{xcnt} \cdot \mathit{pcm}_x \cdot \mathit{ccm}_x && (5.24) \\
&\quad x \in \{\, \mathit{lrd,lwr,srd,swr} \}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{allSymb} &= \mathit{lrdSymb} + \mathit{lwrSymb} + \mathit{srdSymb} + \mathit{swrSymb} && (5.25) \\
\mathit{allCost} &= \mathit{ifCost} + \mathit{lrdCost} + \mathit{lwrCost} + \mathit{srdCost} + \mathit{swrCost} && (5.26) \\
S_0 &= 2 \cdot \mathit{allSymb} \;/\; \mathit{allCost} && (5.27) \\
\mathit{rtDelay}_i &= \mathit{ring}(\mathit{numNodes}, S_{i-1}) && (5.28) \\
\mathit{localT}_i &= \mathit{lrdCost} + \mathit{lrdRaccs} \cdot \mathit{rtDelay}_i \cdot \mathit{lrdTrips} + \\
&\quad \mathit{lwrCost} + \mathit{lwrRaccs} \cdot \mathit{rtDelay}_i \cdot \mathit{lwrTrips} && (5.29) \\
\mathit{sharedT}_i &= \mathit{srdCost} + \mathit{srdRaccs} \cdot \mathit{rtDelay}_i \cdot \mathit{srdTrips} + \\
&\quad \mathit{swrCost} + \mathit{swrRaccs} \cdot \mathit{rtDelay}_i \cdot \mathit{swrTrips} && (5.30) \\
t_i &= \mathit{ifCost} + \mathit{localT}_i + \mathit{sharedT}_i && (5.31) \\
S_i &= 2 \cdot \mathit{allSymb} / t_i && (5.32)
\end{aligned}
$$

Model accuracy. The main advantage of analytical models is that results can be obtained much faster than is possible with simulation. However, the accuracy of the results is only as good as the description of the system under study that is embedded in the model. The accuracy of the model predictions can be assessed by comparing them to the values produced by the simulator. Feeding the reference counts and ratios from the workload to the model, the results obtained are in good agreement with the simulation results. Table 5.1 shows the range of variation for each of the programs and Figure 5.1 plots the percentage of error *versus* frequency of occurrence for 120 simulation runs – six programs on five cache sizes on four ring sizes. The worst errors are `mp3d()` (+11%) for the two largest cache sizes and, `paths()` for a 64Kbytes cache (−13%). In both Figure 5.1 and Table 5.1, a negative error means that the model is underestimating the execution time.

Ring size	2	4	8	16
`chol()`	−1, 0	−4, 0	0 +4	0, +4
`mp3d()`	−10, −8	−7, −3	−3, +3	+2, +11
`water()`	0	−1, 0	−3, −1	−6, −1
`ge()`	0	0	0	0
`mmult()`	0	0, +1	−4, 0	−1, +6
`paths()`	−1, 0	−3, 0	−7, 0	−13, 0

Table 5.1: Range of percentage error in the model prediction when compared to the simulation results. A negative error means that the model underestimates the execution time.

The model output for `mp3d()` overestimates network latency by 30% for the two largest cache sizes, and by 12–14% for the three smaller cache sizes. The time spent on shared-data references is also overestimated by about 10–15%. The model does not explicitly account for synchronisation operations but assumes the synchronisation-related memory references to be normal shared-data references. The model underestimates network latencies for `paths()` on the 64Kbytes caches (19%) and shared-data references (33%). The equations that describe the protocol actions do not reproduce faithfully the behaviour of the system under very high levels of coherency activity such as that produced by the high miss ratios caused by `paths()` on small coherent caches. For more “normal” behaviour, model outputs are within 5% of the simulation results.

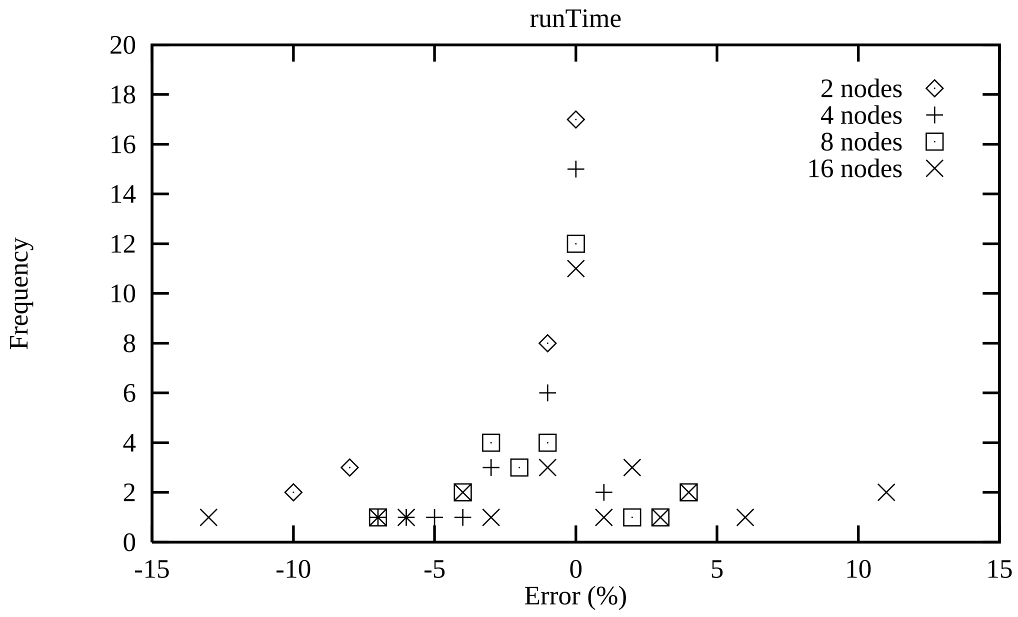


Figure 5.1: Distribution of error in the model predictions when compared to the simulation result. A negative error means that the model underestimates the execution time.

5.2 Costing Sharing-lists and Conflict Misses

Impact of long sharing-lists on performance. One of the bottlenecks in SCI is the potentially high cost of purging long sharing-lists since that cost grows linearly with the number of copies that have to be purged. This can make synchronisation operations expensive unless measures are taken to avoid a large degree of sharing of barriers and locks – see, for example, [MS91,AGGW94]. Some algorithms have large degrees of sharing for some or all of their shared variables, of which `paths()` is a good example. The length of the its sharing-lists grows as 2, 3.4, 5.4 and 7.1 for 2, 4, 8 and 16 nodes, respectively (256Kbytes cache). What would be the impact on performance if the degree of sharing were even higher? Using the parameters from the 256K and 512Kbytes caches, the model produced the results shown in Figure 5.2.

The performance degradation is more easily seen on the 512Kbytes cache. The case that is closest to the simulation is the 5-copies list. Taking that as basis, the longest list (15 copies, 16-node ring) is 4% slower on both cache sizes. The difference between the 1-copy list to the 16-copies list is just over 6%. The model underestimates network latency by 7% for the 256K cache and 3% for the 512Kbytes cache. Taking that into account and assuming that network latency will be underestimated further for the longer lists, the effect of long sharing-lists, for `paths()`, is still acceptably small for systems with up to 16 nodes. Notice however that the number of writes to shared-data is relatively small when compared to the number of shared-data reads – see Table 3.3 (page 37).

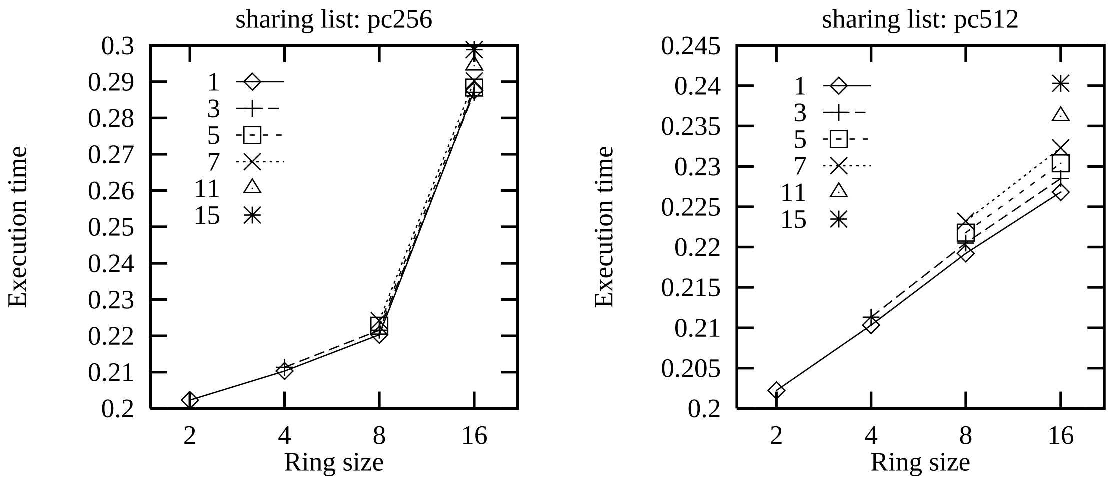


Figure 5.2: Effect of long sharing lists on the performance of `paths()` with 256K and 512Kbytes caches. The number of copies purged varies from 1 to 15.

The cost of conflict misses. Another interesting question concerns the effects of flushing lines from the coherent caches because of conflict misses. `mmult()` has no write-shared data and therefore no sharing-lists are purged, except on a handful of false-sharing cases. Thus, besides compulsory and capacity misses, all other coherency activity is caused by conflict misses. These are responsible for the poor performance of the 8-nodes ring when compared to the other ring sizes. Figure 5.3 (page 75) shows the effects on performance when the number of lines flushed per coherent cache miss varies from 0 to 1.

On the 8-node ring with 128Kbytes (512K) cache, the difference in speed between never flushing to always flushing is 21% (17%). The level of traffic is higher than on the other ring sizes because of the conflict misses between instructions/local-data and shared-data. On the 16-node ring, the difference is 15% (6%). These findings were somewhat surprising since purging sharing-lists is normally identified as the major bottleneck in SCI. Thus, the experiment above was repeated using data from `paths()`, keeping the sharing-list related values as per the simulations but changing the flush ratios. The results are shown in Figure 5.4. On a 16-node ring, 256Kbytes cache (512K), always flushing is 10% (4%) slower than never flushing. Keeping in mind the caveats about model accuracy, for `paths()`, the impact on performance of conflict misses is more pronounced than that of long sharing-lists.

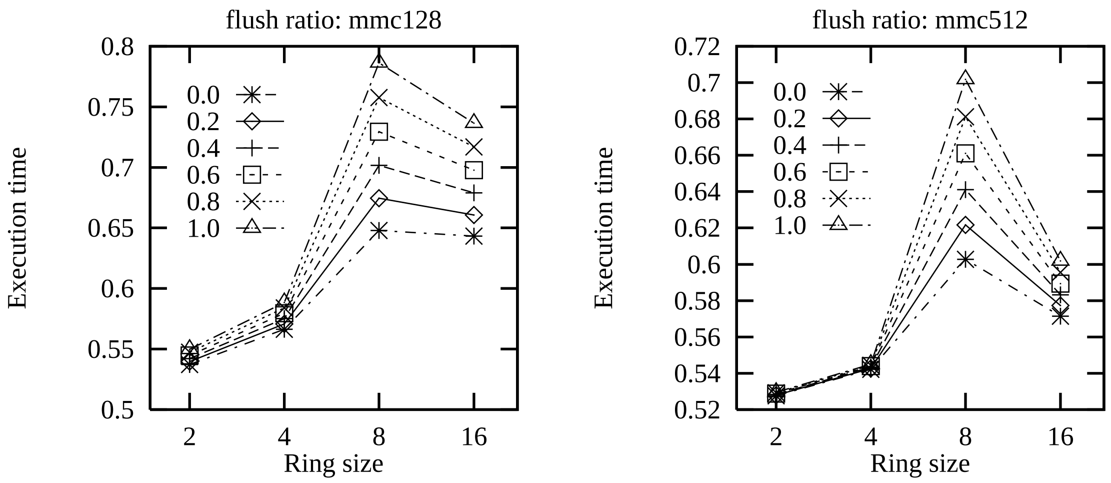


Figure 5.3: Effect of conflict misses on the performance of `mmult()` with 128K and 512Kbytes caches. The flush ratio varies from 0 to 1.

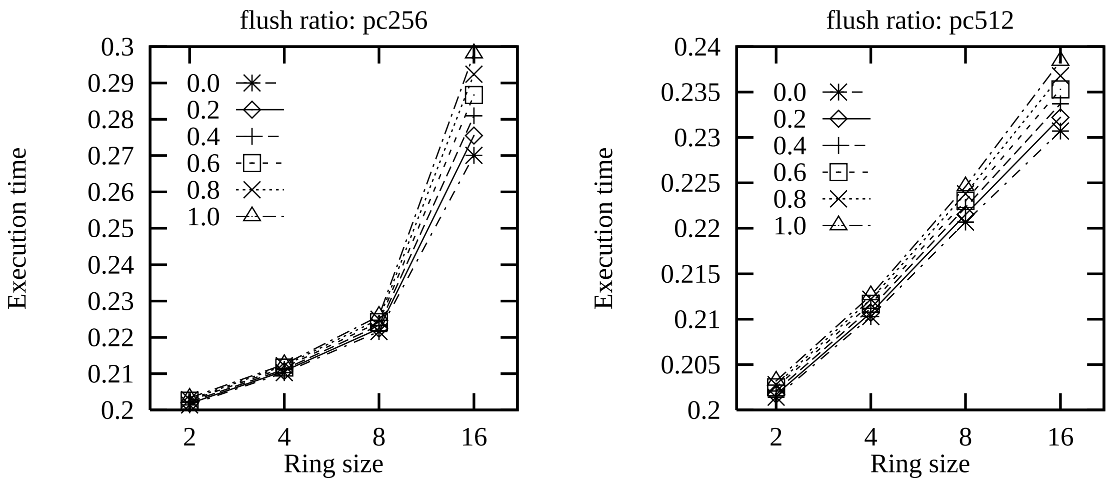


Figure 5.4: Effect of conflict misses on the performance of `paths()` with 256K (left) and 512Kbytes caches (right). The flush ratio varies from 0 to 1.

The effects of long sharing lists and conflict misses on the workload is shown in Table 5.2. The parameters are those of 256Kbytes coherent caches. First, the length of sharing-lists was varied from 1 copy to 15 copies and the table shows the slowdown for the 15-copies (7-copies) lists when compared to the 1-copy lists on 16-node (8-node) rings. Notice that, except from `paths()`, all of the other programs have sharing lists with an average length close to 1. The table also shows the slowdown of a very high level of conflict misses (always flushes: flush ratio = 1.0) in relation to no conflict misses (never flushes: flush ratio = 0.0).

Table 5.2 shows that there is a marked difference in behaviour between `mmult()` and `paths()` and the other programs. The former suffer much more from long sharing-lists than from high levels of conflict misses whereas the opposite is true of the latter programs. The difference is caused by the higher rate of writes to reads on the latter and the number of writes itself. `mmult()` and `paths()` perform only a few thousand writes while the others do hundreds of thousands of writes. Note that five of the six programs have sharing-list lengths much closer to the basis of the comparison (one copy) whereas they all exhibit levels of conflict misses that do not fall at either extreme of the comparison range.

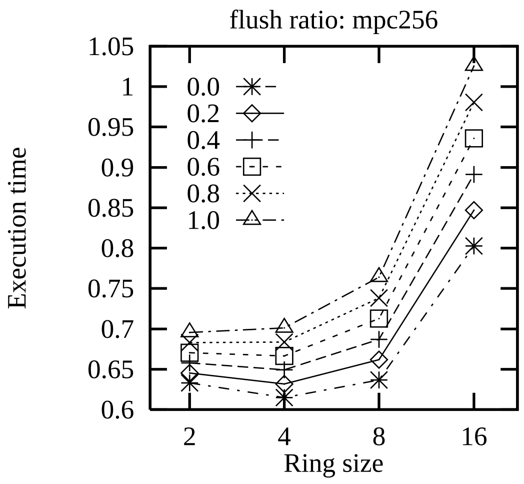
	sharing-lists		conflict misses	
Relative change	7 : 1	15 : 1	1 : 0	1 : 0
Ring size	8	16	8	16
`chol()`	5	25	3	5
`mp3d()`	43	153	20	28
`water()`	4	18	2	3
`ge()`	0	4	1	2
`mmult()`	0	0	17	9
`paths()`	2	4	2	10

Table 5.2: Change is speed (%) for long sharing-lists and high level of conflict misses. Values for 256Kbytes caches. See text for details.

`mp3d()` has the most extreme behaviour of the workload as far as network latency is concerned. Thus, the experiments with varying sharing-list length and flush rates were repeated for that program. Figure 5.5 (page 77) shows the results for `mp3d()`. When comparing 3-copies lists to 1-copy lists, the differences is over 20% for `mp3d()` on a 16-node ring. Small increases in the levels of write-sharing would cause serious performance degradation.

Coda

The analytical model presented in this chapter estimates the performance of programs executing in an SCI ring-based multiprocessor with reasonable accuracy but at a small fraction of the corresponding simulation time. The model predictions are quantitatively good for programs that exhibit a "normal" behaviour and qualitatively good for more extreme behaviours. The next chapter investigates medium-sized multiprocessors in which the interconnects are higher dimensional networks composed of several SCI-rings.



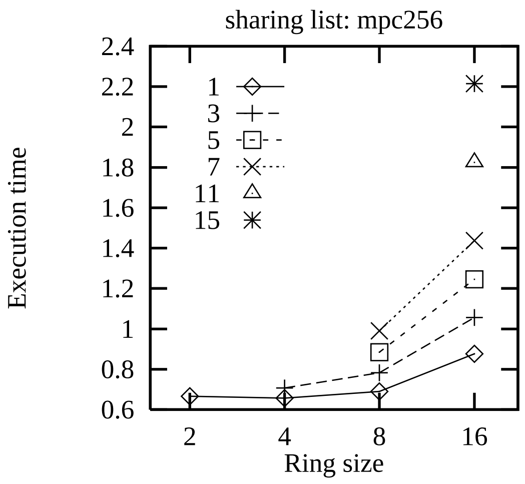


Figure 5.5: Effects on performance of conflict misses (left) and sharing-list length (right) for `mp3d()` on 256Kbytes coherent caches.

Chapter 6

The Performance of Meshes and Cubes

This chapter investigates the performance of SCI-based multiprocessors with up to 64 nodes. Chapter 4 provides evidence against the use of one-dimensional networks for systems with more than 8 to 16 nodes with the type of processor and memory hierarchy simulated. Here, two- and three-dimensional networks are examined. In this chapter, two members of the family of k-ary n-cubes [Sei85,Dal90,SG91] are investigated, namely the mesh ($n = 2$) and the cube ($n = 3$). These networks are implemented by having two or more pairs of SCI links on each node, each pair belonging to a different ring. Thus, an SCI-mesh is actually a mesh-of-rings (a torus) where each node belongs to a "vertical" and a "horizontal" ring.

The chapter is organised as follows. Section 6.1 discusses the simulation environment used to conduct the experiments. Sections 6.2 and 6.3 contain the simulation results for SCI meshes and cubes. Finally, Section 6.4 compares the performance of the workload on SCI rings, meshes and cubes.

6.1 The Simulated Multiprocessor

The simulation environment is basically the same as described in Chapter 3. The network simulator was modified to incorporate the packet router described in Section 6.1.1 and the statistics module was changed to record the extra traffic-related information. Each ring in the network is modelled independently by Equation 3.1. The cost of switching dimension is five extra network cycles (10nS). The cost of a transaction is computed by adding up the memory and network delays on all rings in the path from requester to responder.

6.1.1 Routing

An important component of an k-ary n-cube network is the routing function. An SCI-based network has characteristics of both wormhole routing and store-and-forward [Tan89]. If the network is idle and a given node is not inserting a packet, as soon as the destination address of an incoming packet is decoded by the parser, it can be sent down the link leading to its destination, as is the case in wormhole routing. If however a given node is inserting a packet, incoming traffic is buffered until that packet is fully transmitted. Because of the buffering and the possibility of blocked paths, the routing algorithm must be deadlock-free.

The router "implemented" by the simulator is based on the *e-router* [DS87]. The e-router is shown to be deadlock-free on SCI-based k-ary n-cubes by Johnson and Goodman in [JG91]. The path from source to destination is always chosen by inserting the packet at the highest dimension, where it travels as far as possible before being switched onto the next lower dimension. Deadlock avoidance is ensured by the partitioning of network queues into a set of ordered classes, with the queues in each dimension comprising one of the classes. Suppose, for example, that in the torus depicted in Figure 6.1, node 3 sends a packet to node 8. Node 3 injects the packet into its Y (vertical) link. The packet travels on that ring until it reaches node 11 where it is re-routed onto the X link and reaches node 8, its destination. Notice that two echo packets are created, one from node 11 to node 3, on the Y ring and, one from node 8 to node 11, on the X ring. This algorithm can be easily extended for routing on higher dimensionality networks.

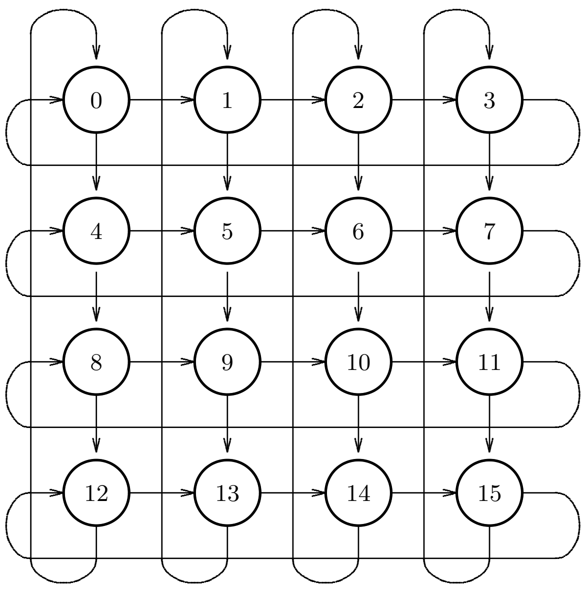


Figure 6.1: A four-by-four SCI mesh.

SCI's communication protocol ensures delivery within a single ring. Point-to-point delivery must be ensured by higher level protocols implemented by inter-ring switches. SCI allows for pipelined packet transmission along the route between source and destination. The switch strips a packet from a higher dimension ring and inserts that packet into a lower dimension ring, while at the same time the packet's echo is being returned on the higher dimension ring. This frees up buffer space at the transmitter stage of the requester (the 'active buffers' in Figure 3.3) and intermediary switches.

6.1.2 SCI Switches

An SCI switch contains a pair of links for each dimension, in the case of k-ary n-cube networks. Configurations for other topologies are described in [JG91]. Figure 6.2 depicts the data path of a two-dimensional switch. The incoming packets at X- and Y-dimension link inputs are (1) passed along if the node is not the destination of the packet and its destination is in the same dimension; (2) steered to the node interface if the packed is addressed to the node; (3) placed at the other dimension's output buffer if the destination node is in that dimension's ring.

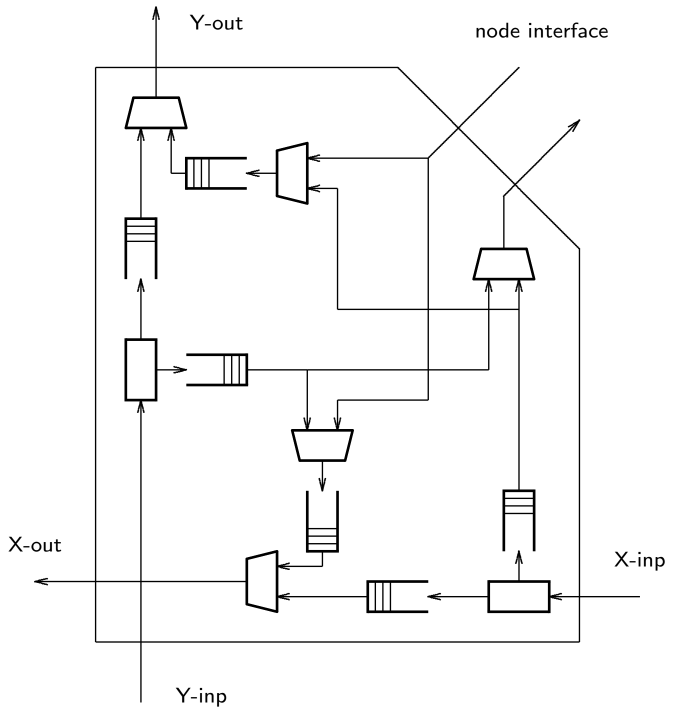


Figure 6.2: Data paths of a two-dimensional SCI switch.

In SCI, memory is physically addressed. Thus, the node address is an integral part of the address of a coherent line. The mechanism that steers packets within a switch can be implemented by simple and fast combinatorial circuits. The decision to accept or re-route a packet can be taken by masking certain address bits [Sei85, Hwa93]. The extension to higher dimensional networks can be achieved in two ways. For low dimensionality networks (3-D or 4-D), a simple extension of the circuit depicted in Figure 6.2 would be feasible. For higher dimensionality, the link interfaces within the switch could themselves be interconnected in a ring. Johnson and Goodman propose such a scheme in [JG91]. Contention for the internal ring would be small since all the changes in dimension occur between adjacent dimensions, that is, dimensions i and $((i + 1) \bmod n)$. Notice that this assumes a deadlock-free routing similar to that discussed in Section 6.1.1.

6.2 SCI Meshes

The performance of SCI-connected meshes is investigated in this section. Figure 6.1 depicts a four-by-four mesh. Note that the rows and columns can be staggered so that all links have the same length [Dal90]. Because of high computational costs, the systems simulated were restricted to three sizes, namely four- (two-by-two), sixteen- (four-by-four) and 64-node tori (eight-by-eight). The effect of secondary cache size on performance is assessed. Three cache sizes were simulated, namely 128, 256 and 512Kbytes. Processor throughput, packet delivery delays, queue throughput and link traffic are investigated as well. Queue throughput and link traffic are useful metrics because they can expose bottlenecks caused by uneven traffic patterns.

6.2.1 Machine and Cache Size – SPLASH Programs

Figure 6.3 shows comparative simulation results of the SPLASH applications on the machine and cache sizes simulated. Recall that the data-set sizes are scaled up with machine size – see Table 3.2 (page 36). `chol()` is 3.2 times faster on a 16 nodes, 128Kbytes system than on a 4-node machine. The speedup is 2.8 with the two other cache sizes – the performance is better but the speedup is worse. `mp3d()` behaves the same way for the three cache sizes. `water()` shows a 12–14% decrease in speed for a quadrupling in system size. This is partly caused by the scaling factor used, where processors in 64-node systems do proportionally more work. `mp3d()` and `water()` show negligible performance gains with increases in cache size.

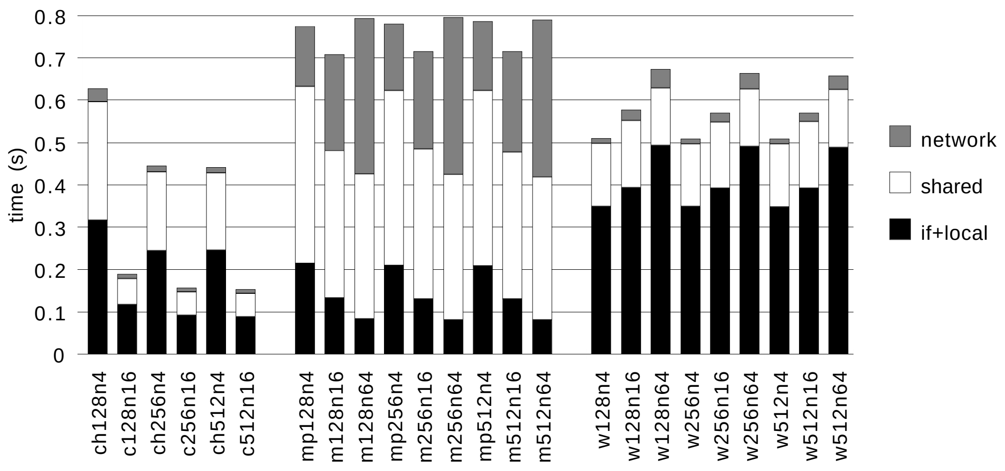


Figure 6.3: Execution time plots for `chol()` (left), `mp3d()` and `water()` (right). The segments show the time spent on shared-data references and network latency. Data sets are scaled up with machine size.

Coherent cache hit ratios. The shared-data read hit ratios of the SPLASH applications are shown in Figure 6.4. `chol()` has a slightly lower hit ratio on the 128Kbytes caches when compared to the two larger cache sizes. The hit ratios of `mp3d()` show a strong dependence on machine size but negligible changes with cache size. `water()` shows a small decrease in hit ratios when going from a 4- to a 16-node machine. The low hit ratios on 64-node machines are caused by the data set not fitting in the caches – the rates improve about 5 percentage points for each cache size doubling.

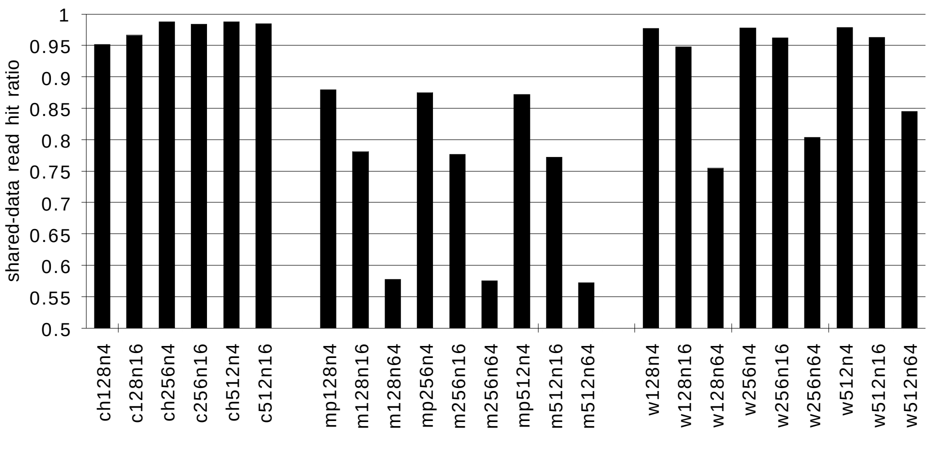


Figure 6.4: Coherent cache shared-data read hit ratio plots for `chol()` (left), `mp3d()` (center) and `water()` (right).

Execution time breakdown. Figure 6.5 shows the execution time breakdown for the three SPLASH applications, for the machine and coherent cache sizes simulated. Time was divided into (1) fetching and execution of instructions, (2) references to private data, (3) references to shared data local to the processor, (4) references to shared data in another node, (5) network delays and (6) synchronisation. The fraction of time spent on references to shared data at another node includes all delays caused by protocol actions such as cache/memory tag accesses as well as loading and storing data in caches and main memory. Both `chol()` and `water()` spend about half the time executing instructions and under 20% of the time referencing remote shared-data. `mp3d()` spends most of the time in references to remote shared data and synchronisation. As is the case with the shared-data hit ratios, there is little change with cache and machine sizes.

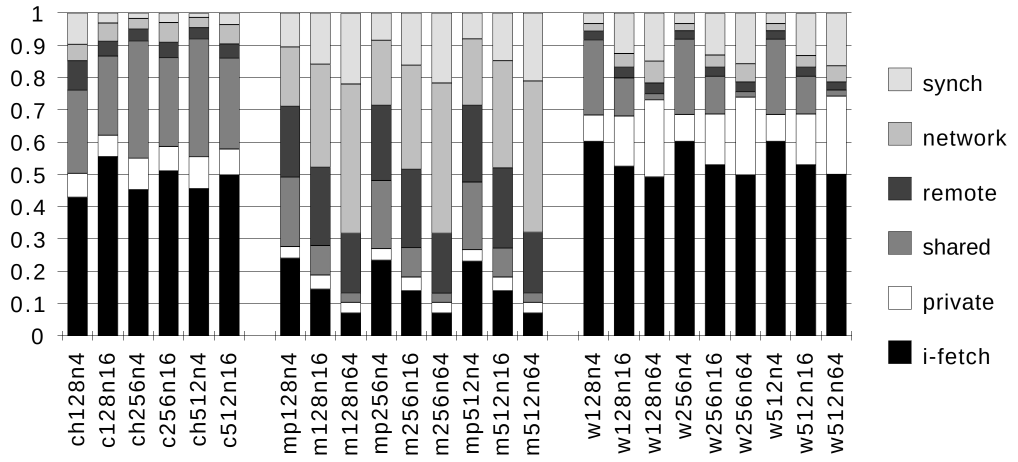


Figure 6.5: Execution time breakdown for `chol()` (left), `mp3d()` (center) and `water()` (right).

6.2.2 Machine and Cache Size – Parallel Loops

Figure 6.6 compares the performance of the parallel loops for the machine and cache sizes simulated. Recall that the data-set sizes are scaled up with machine size – see Table 3.2 (page 36). `ge()` scales up well, with just a small performance loss for a quadrupling of system size. On the 64-node machine, `mmult()` displays an improvement in speed of about 10% for each cache size doubling, because of the improvement in the shared-data hit ratios. The same applies to the smaller systems. `paths()`'s performance depends on cache size. When the shared-data hit-ratios are low, performance degrades badly because of the high cost of fetching needed data and purging sharing-lists. The plots indicate that the network is saturated since a very substantial portion of the execution time is spent on network delays – see Section 6.4.1, page 97.

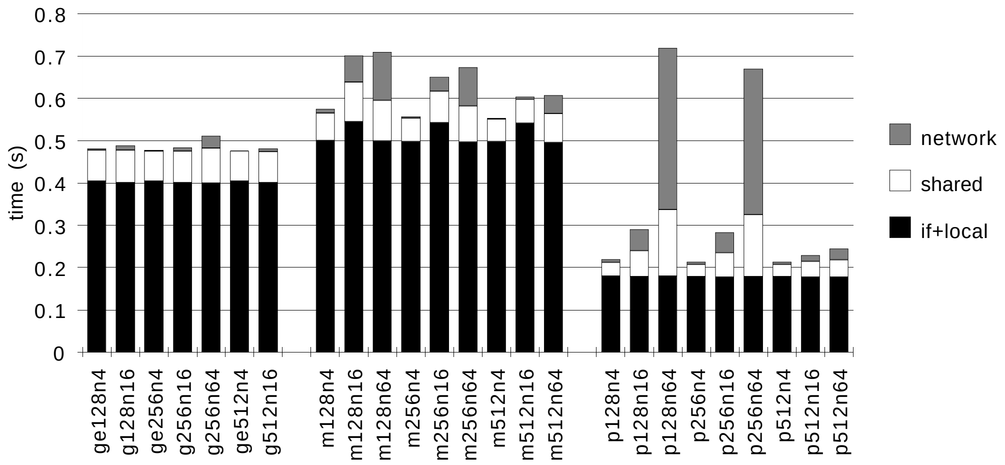


Figure 6.6: Execution time plots for `ge()` (left), `mmult()` (center) and `paths()` (right). The segments show the time spent on shared-data references and network latency. Data sets are scaled up with machine size.

Coherent cache hit ratios. Figure 6.7 (page 85) shows the shared-data-read hit ratios for the parallel loops. `ge()` has high hit ratios and these vary little with cache and machine size. `mmult()` has slightly lower hit ratios and these depend on both cache and machine size. `paths()` on the other hand, shows large changes in hit ratios with cache size and, to a much lesser extent on machine size. The 512Kbytes cache is large enough to contain most of the data set. In the three programs, the decrease in the hit ratio with system size is caused by an increase in interference amongst the processors, that is, larger systems have higher degrees of write-sharing and longer sharing-lists.

Execution time breakdown. Figure 6.8 shows the execution time breakdown for the parallel loops for all the machine and coherent cache sizes simulated. Time was divided into (1) fetching and execution of instructions, (2) references to private data, (3) references to shared data local to the processor, (4) references to shared data in another node and (5) network delays. `ge()` spends a large fraction of its time executing instructions and just a little time on references to remote shared-data. The fraction of execution time in which `mmult()` is stalled waiting for network delays increases with machine size and decreases with cache size. `paths()`, when the caches are large enough to contain the data sets, spends most of the time performing instructions. If the caches are too small, network traffic then takes a very substantial fraction of the execution time.

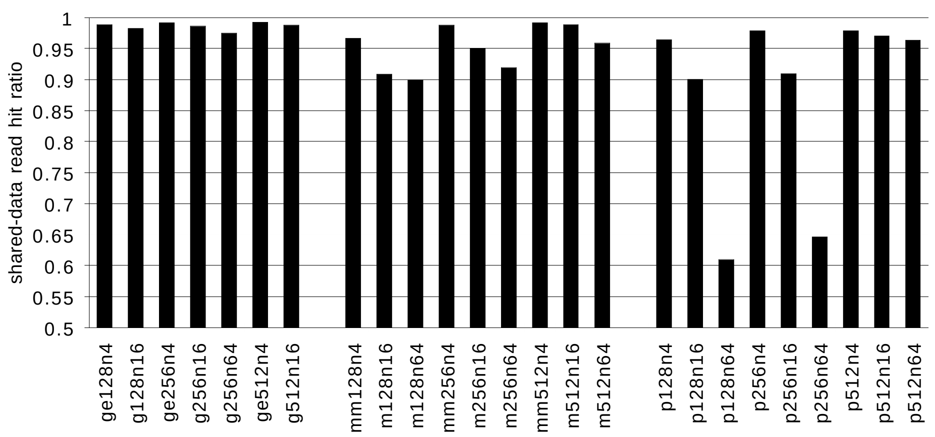


Figure 6.7: Coherent cache shared-data read hit ratio plots for `ge()` (left), `mmult()` (center) and `paths()` (right).

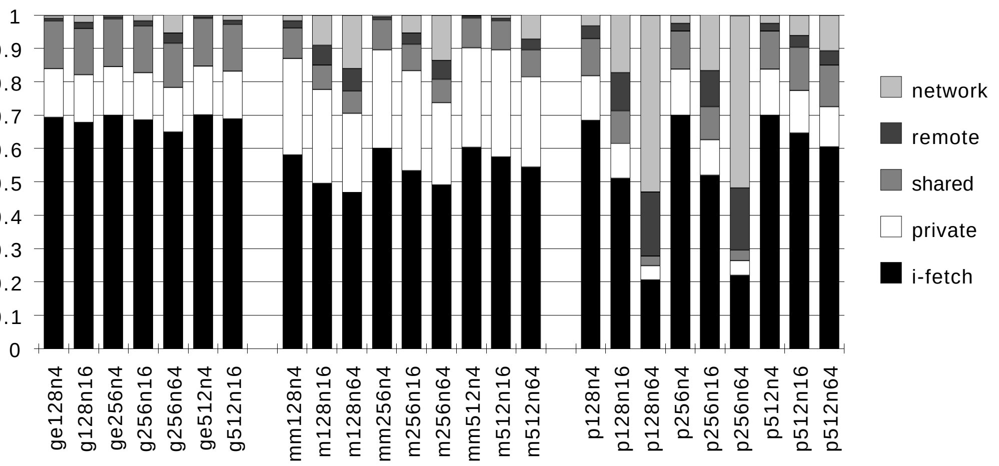


Figure 6.8: Execution time breakdown for `ge()` (left), `mmult()` (center) and `paths()` (right).

6.2.3 Throughput and Latency

The network characteristics of the SCI-meshes is examined next. Unlike the ring, a single processor request can generate up to two packets: one for each network dimension. The first packet is injected by the processor and the second by the SCI interface of the node where the change of dimension occurs. Thus, processor throughput is here computed by taking the traffic generated by onboard processor and cache/memory controller only and dividing it by the execution time.

Figure 6.9 shows the throughput per node for systems with 256Kbytes coherent caches. The applications with the lower shared-data hit ratios are the ones that cause most traffic on the network, namely `mp3d()` and `paths()`.

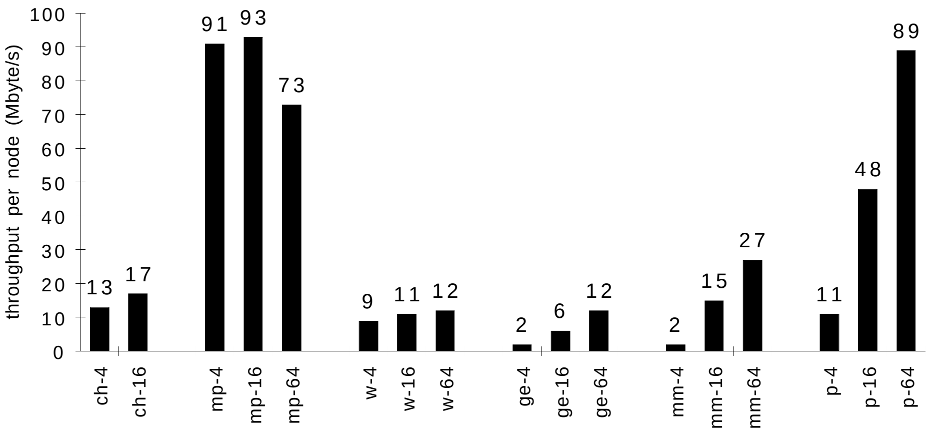


Figure 6.9: Throughput per node for 256Kbytes caches. From left to right: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

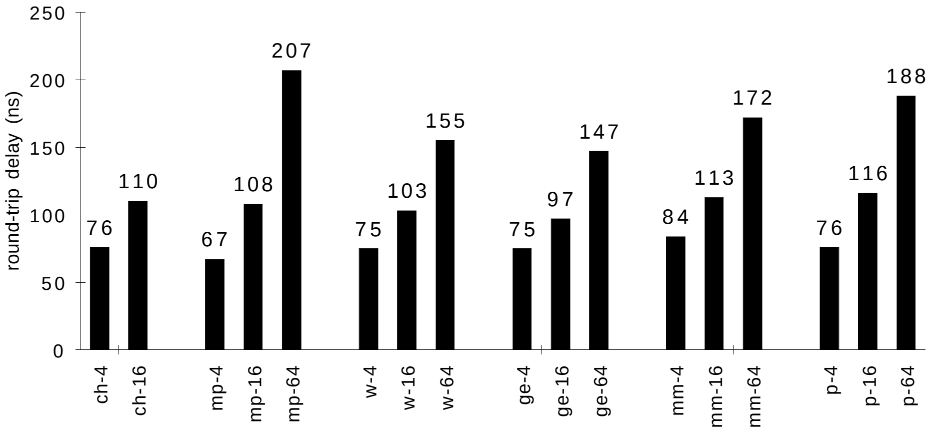


Figure 6.10: Round-trip delays for 256Kbytes caches. From left to right: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

Figure 6.10 shows the round-trip delays incurred by the applications. Those which generate higher traffic endure longer delays, as is the case with single rings. The round-trip delay is computed in a similar fashion to the throughput. The “round-trip” considered is that of packets inserted by the processors. The delay is computed by dividing the time spent on network latencies by the number of packets inserted by each processor. On a 2x2 torus, the average round-trip delay is 75ns; on a 4x4, is 108ns and, on a 8x8 torus, is 174ns. These values are in broad agreement with Equation 2.3 (page 9) , with c set to 5 (five extra network cycles on a change of dimension). Note however that the delays estimated by Equation 2.3 are maximum delays whereas those measured are closer to average or best-case on the smaller systems.

Higher dimensionality networks offer high bandwidth and, in theory, suffer less from network congestion. However, the interaction between allocation of processes to processors and data to nodes can cause non-uniform traffic patterns and hot spots. The quality of a network design depends on how these are tolerated. The following paragraphs discuss the measurements of traffic through links and queues in the network.

In meshes, unlike rings, the routing algorithm and allocation of data to nodes can produce different traffic patterns on individual rings or on the rings belonging to a given dimension. The average value for link traffic tends to hide irregular behaviours. Figure 6.11 (page 88) shows both the average and the peak traffic per link for the two dimensions. Peak traffic varies considerably between the two dimensions whereas average traffic is roughly the same. As is the case with throughput, `mp3d()` and `paths()` cause the highest levels of traffic.

The output buffer holds packets inserted by the processor and local cache as well as packets that changed dimension and are entering the second leg of their trip. The routing has a more noticeable effect here, with the Y-dimension buffers being busier than their X-dimension counterparts – see Figure 6.11. As before, `mp3d()` and `paths()` have the busiest output queues but, except for `paths()`, average traffic is much closer to peak. `paths()` has a hot spot node that handles nearly twice the average traffic.

The plots for the traffic through the bypass buffers are similar to those of link traffic, with X-dimension queues being busier – see Figure 6.11. Because of the routing algorithm, the majority of the packets are injected into Y-rings whereas most of the non-local traffic occurs on the X-rings. This is a consequence of the mapping of data to nodes: when the quota of pages is exceeded, overspill pages are allocated to the neighbour on the X-dimension.

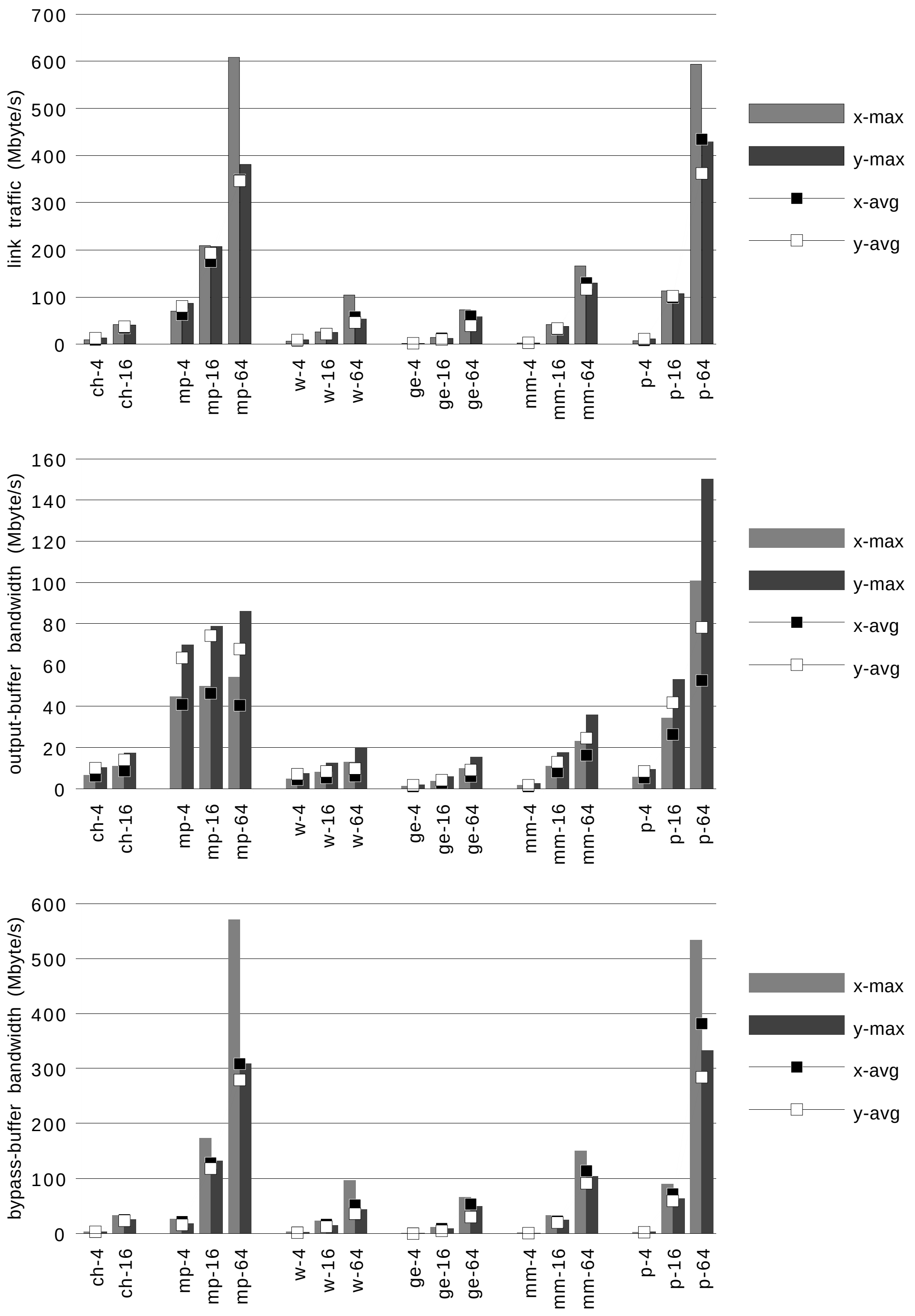


Figure 6.11: Link traffic per dimension (top), output buffer traffic per dimension (mid) and bypass buffer traffic per dimension (bottom), for 256Kbytes caches. From left to right: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

6.3 SCI Cubes

This section discusses the performance of SCI-connected cubes. Figure 6.12 depicts a four-by-four-by-four cube. Because of high computational costs, the systems simulated were restricted to two sizes, namely eight- (2x2x2) and 64-node cubes (4x4x4). The effect of secondary cache size on performance is assessed. The cache sizes simulated were 128, 256 and 512Kbytes. Processor throughput, packet delivery delays, queue throughput and link traffic are investigated as well.

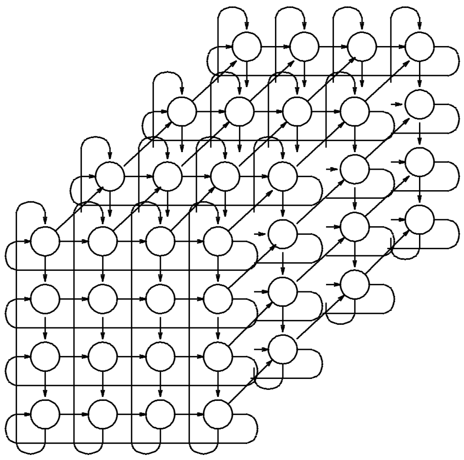


Figure 6.12: A four-by-four-by-four SCI cube. Wrap-around connections in the Z dimension not shown.

6.3.1 Machine and Cache Size

Figure 6.13 shows the execution time for the SPLASH applications on machines with 8 and 64 nodes and 128, 256 and 512Kbytes caches. Recall that the data-set sizes are scaled up with machine size – see Table 3.2 (page 36). `chol()` performs better on the two larger cache sizes and the poorer performance with 128Kbytes stems from lower hit ratios. `mp3d()`'s performance is roughly independent of cache size. Of the 8-node systems, the 128Kbytes is faster because of less cache pollution, as was the case on the single ring (Section 4.2.2). `water()` is 38–40% slower on the larger machine because of the scaling factor used. For `ge()`, the slight loss in speed on the 64-node, when compared to the 8-node, is caused by higher network latency. `mmult()`'s performance depends on both cache and machine sizes. Although the differences are small, they are a consequence of better hit ratios (see below). `paths()` has a very poor performance with the two smaller cache sizes, on the 64-node machines. This is a consequence of the low hit ratios it endures with the smaller caches.

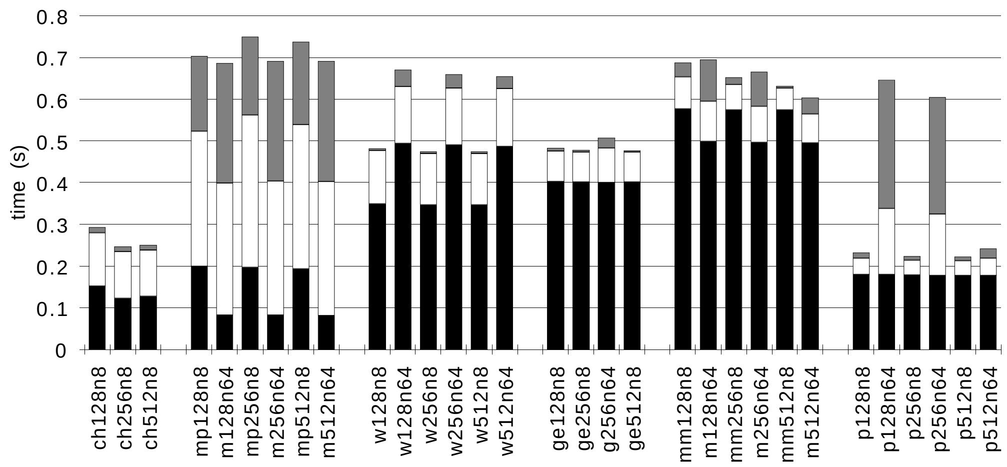


Figure 6.13: Execution time plots for `chol()` (left), `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()` (right). The bottom section of the columns corresponds to time spent fetching instructions and referencing local data, the middle section to references to shared-data and synchronisation, and the top, to network latency. Data sets are scaled up with machine size.

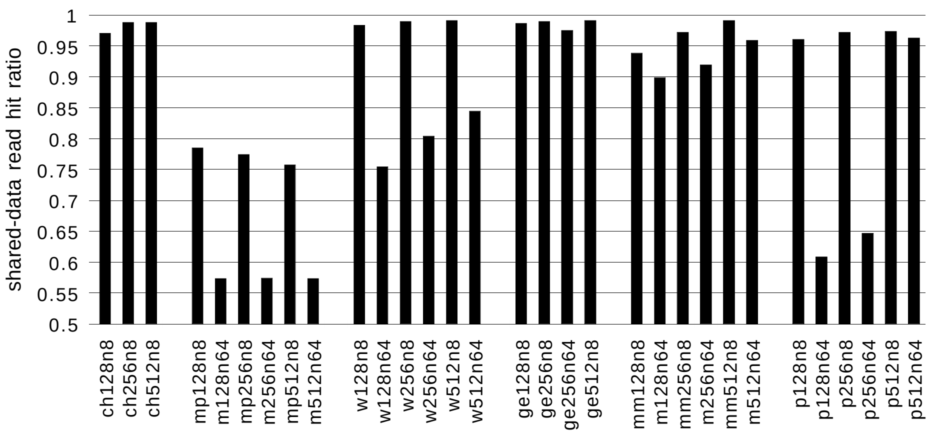


Figure 6.14: Coherent cache shared-data read hit ratio plots for `chol()` (left), `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()` (right).

Coherent cache hit ratios. Figure 6.14 plots coherent cache shared-data read hit ratios for the cache and machine sizes investigated. These hit ratios explain the results presented in Figure 6.13. The situation here is similar to Figures 6.4 and 6.7. `chol()` shows a small improvement with the larger caches. On the 8-node system, `mp3d()` has worse hit ratios on the larger caches. This is caused by higher levels of cache pollution on the larger caches. `water()`, on the 64-nodes system shows an improvement of five percentage points for each cache size doubling. `ge()` has a slightly lower hit ratio on the 64-node, as compared to the smaller system. `mmult()`, on 64-node machines, shows a 5% improvement in the hit ratios of the 512Kbytes cache. Finally, `paths()` needs the 512Kbytes of cache to accommodate its data set and that is reflected in its hit ratios and performance.

Execution time breakdown. Figure 6.15 shows the execution time breakdown for the three SPLASH applications, for all the machine and coherent cache sizes simulated. Time is split into (1) fetching and execution of instructions, (2) references to private data, (3) references to shared data local to the processor, (4) references to shared data in another node, (5) network delays and (6) synchronisation. Recall that the fraction of time spent on references to shared data at another node includes all delays caused by protocol actions as well as loading and storing data in caches and main memory. Both `chol()` and `water()` spend about half the time executing instructions while they spend little time referencing remote data. `mp3d()` spends most of the time in references to remote shared data and synchronisation operations.

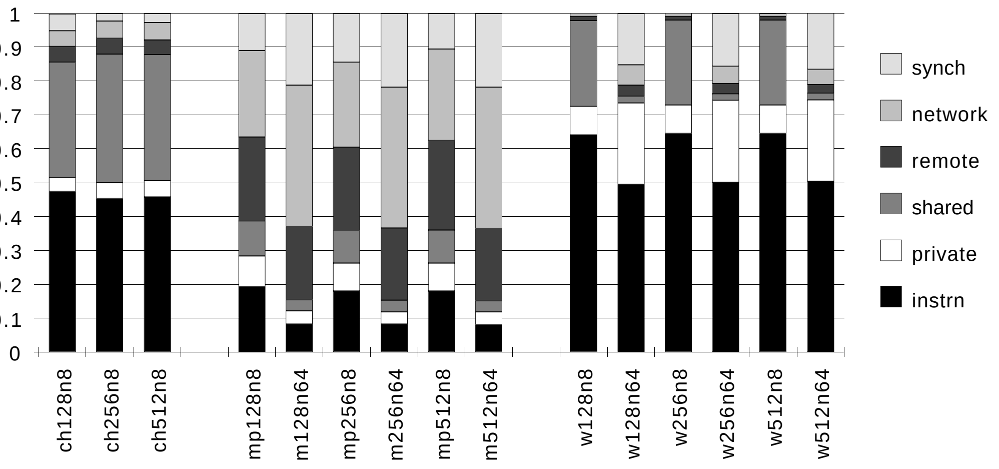


Figure 6.15: Execution time breakdown for `chol()` (left), `mp3d()` and `water()` (right).

Figure 6.16 shows the execution time breakdown for the parallel loops. Time is split into (1) fetching and execution of instructions, (2) references to private data, (3) references to shared data local to the processor, (4) references to shared data in another node and (5) network delays. As with the other two topologies, `ge()` spends a large fraction of its time executing instructions and just a little time on references to remote shared data. The time `mmult()` spends on network latency is inversely proportional to the hit ratios, with the smaller caches spending over 12% of the time on network delays (64-node systems). `paths()`, when the caches are large enough to contain the data sets, spends over half the time performing instructions. If the caches are not big enough, network traffic then accounts for a very large fraction of the time.

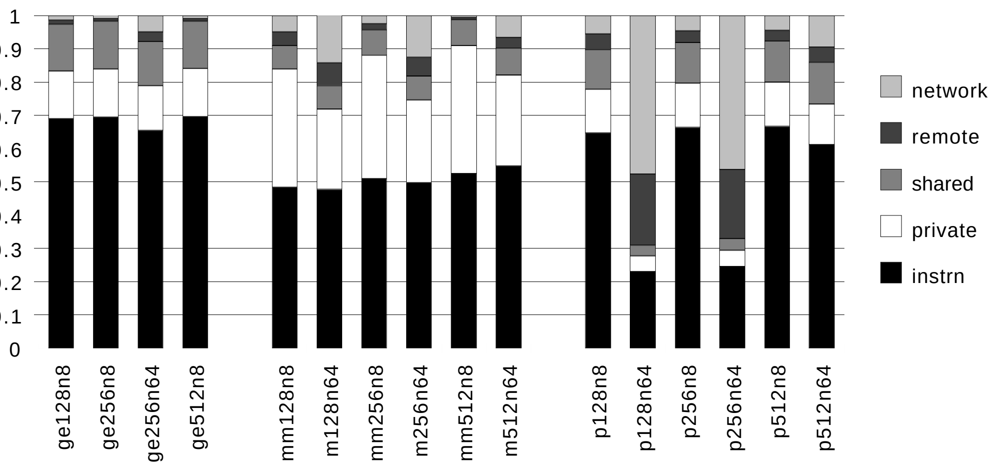


Figure 6.16: Execution time breakdown for `ge()` (left), `mmult()` and `paths()` (right).

6.3.2 Throughput and Latency

Figure 6.17 shows processor throughput and round-trip delay for the six programs simulated with 256Kbytes coherent caches. As was the case with meshes, throughput and latency are computed only from processor generated traffic. The round-trip values are in agreement with Equation 2.3, with c set to 5 (five extra network cycles on a change of dimension). Again, note that the delays estimated by Equation 2.3 are maximum delays whereas those in Figure 6.17 are closer to average or best case. Since the rings are smaller and there are, relatively speaking, more links between nodes, the average distance between nodes is smaller and so are the network latencies, when compared to 64-node meshes.

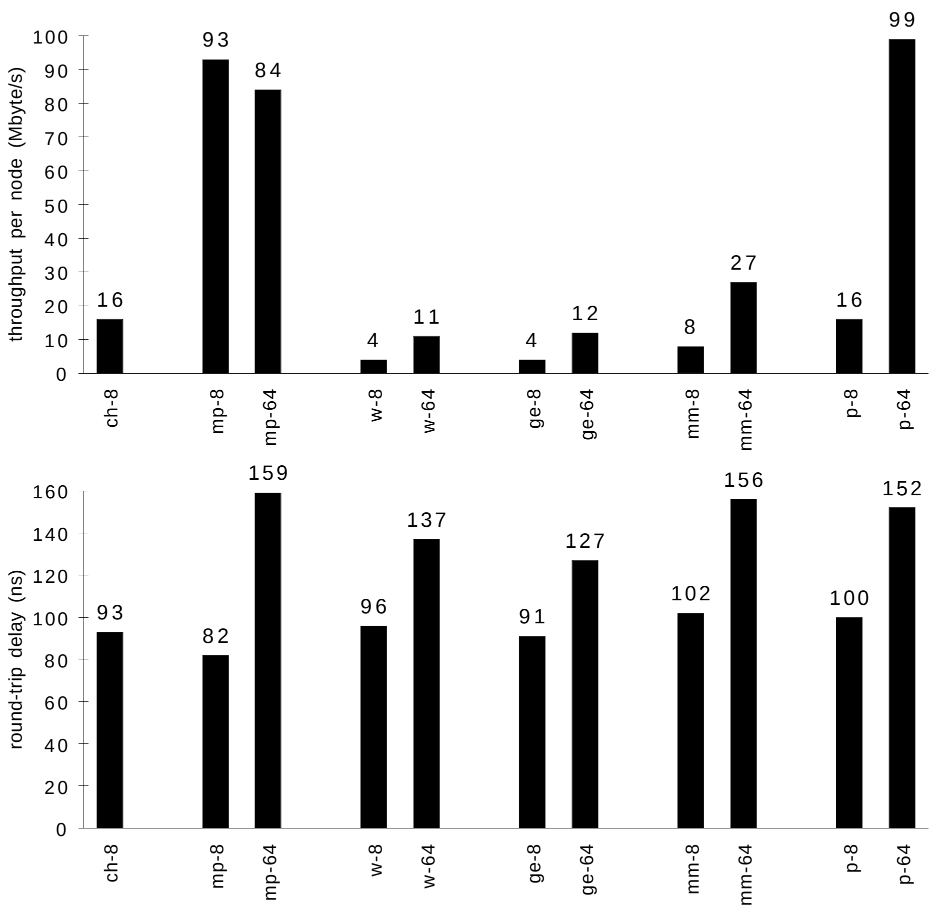
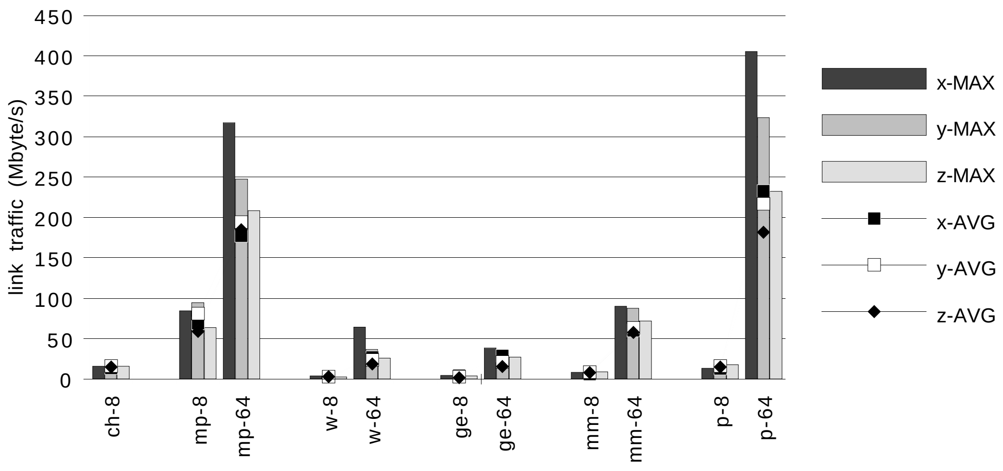
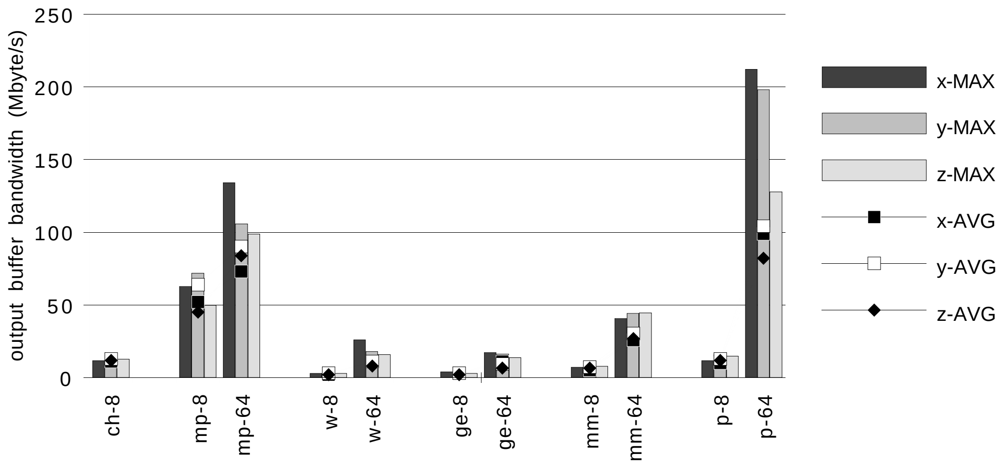


Figure 6.17: Throughput per node (top) and round-trip delay (bottom), for 256Kbytes caches. From left to right: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

The plots for link, output- and bypass-buffer traffic, are shown in Figure 6.18. When comparing to Figure 6.11, the levels of traffic on individual links and queues are lower. This is a direct consequence of increased network capacity. There is another contributing factor which is the proportionally smaller number of nodes on each individual ring, for instance, 4-node rings instead of 8-node rings on the 64-node machines. This decreases the rate of network requests on each ring while the traffic created by packet delivery to nodes in remote rings is divided amongst the links in each dimension.





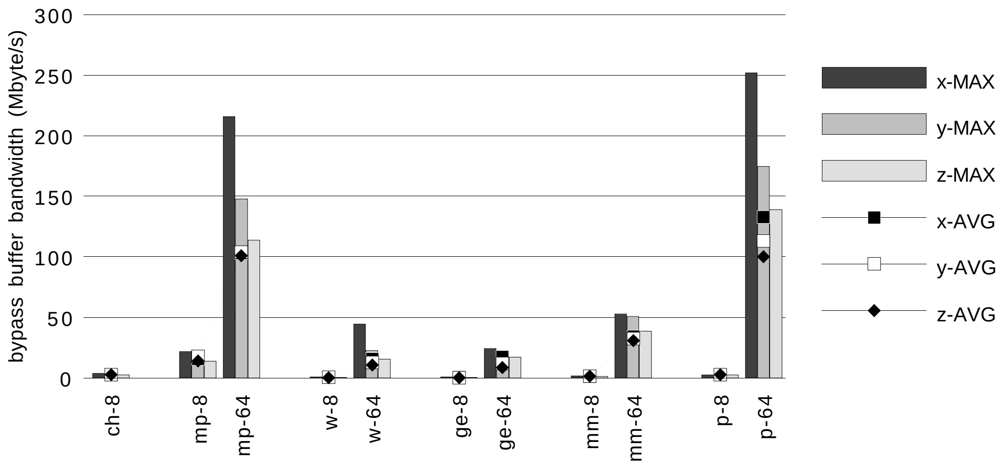
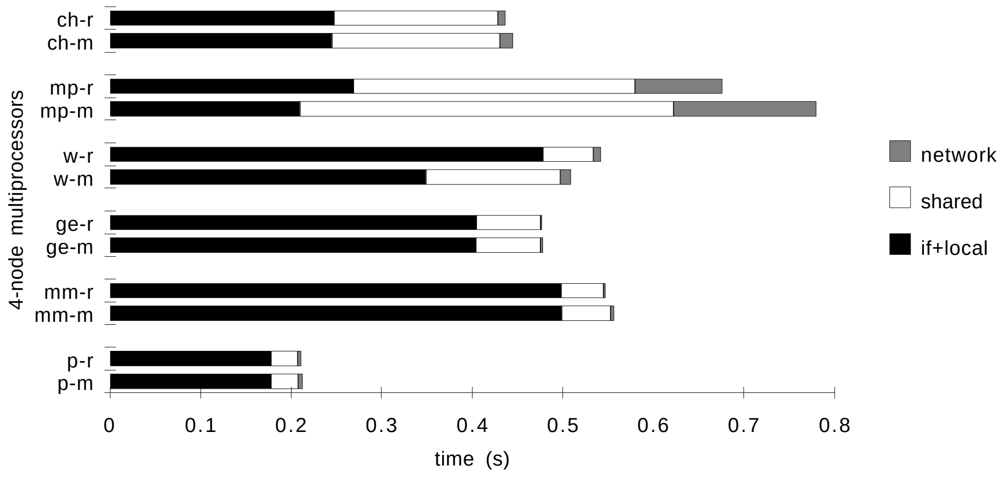


Figure 6.18: Link traffic per dimension (top), output buffer traffic (center) and bypass buffer traffic (bottom), all for 256Kbytes caches. Notice that the vertical scales are all different. From left to right: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

6.4 A Comparison of Rings, Meshes and Cubes

This section compares the performance of the three topologies investigated here, namely rings, meshes and cubes. Since the simulated machine sizes are not all the same on the three topologies, comparisons are drawn for same-sized multiprocessors. The significant metric is execution time. Processor throughput and round-trip latency are examined since these help to explain the relative advantages of one topology over another. Figure 6.19 plots the execution time of 4-node rings and 2x2 meshes, and of 8-node rings and 2x2x2 cubes. The execution time of 16-node rings and 4x4 meshes is shown in Figure 6.20 as well as 8x8 meshes and 4x4x4 cubes.



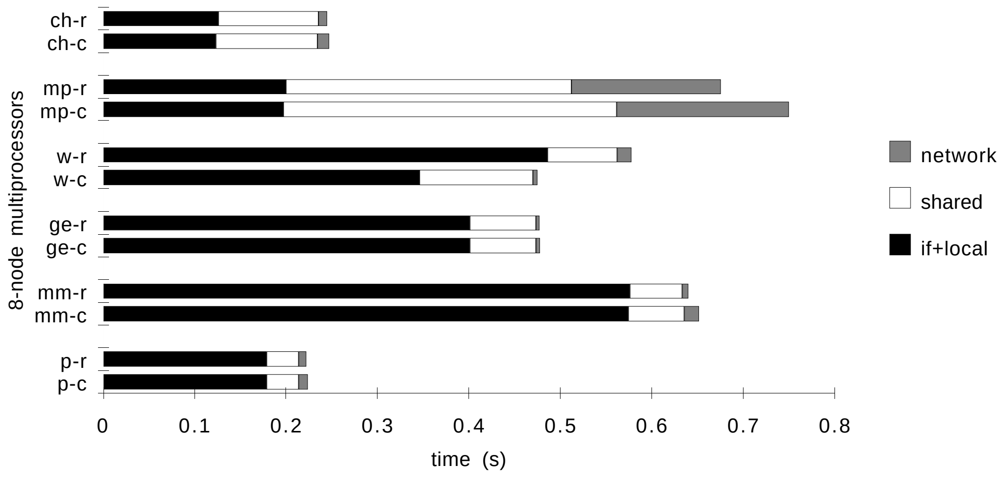


Figure 6.19: Performance of 4-node (top) and 8-node (bottom) multiprocessors, with 256Kbytes caches. The suffixes `-r`, `-m` and `-c` stand for ring, mesh and cube, respectively. From top to bottom: `chol()`, `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

The performance of higher dimensionality networks with 4 and 8 nodes is worse than that in their low-dimensional counterparts – see Figure 6.19. This is caused by the higher cost of changing dimensions, that is, five extra network cycles on each packet delivered to nodes in remote rings. For example, on 4-node systems, on average, two thirds of the packets need to pass through a network switch, thus incurring the extra delays. For small systems, the ring is clearly the best choice, both in terms of performance and cost.

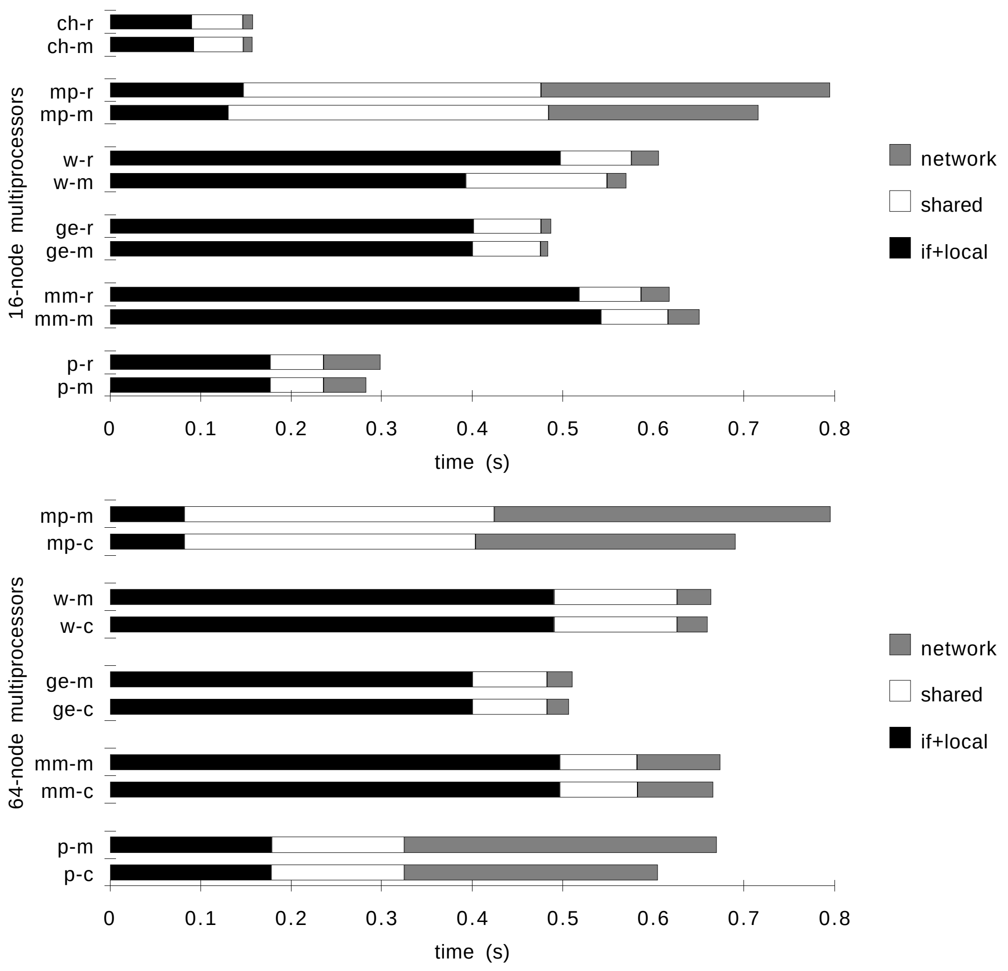


Figure 6.20: Performance of 16-node (top) and 64-node (bottom) multiprocessors, with 256Kbytes caches. The suffixes `-r`, `-m` and `-c` stand for ring, mesh and cube, respectively. From top to bottom: `chol()` (16-node only), `mp3d()`, `water()`, `ge()`, `mmult()` and `paths()`.

On the 16- and 64-node multiprocessors, the higher dimensionality networks are clearly better for programs that cause high levels of network traffic, namely `mp3d()` and `paths()`. For these programs, the performance gains are 15 and 10% respectively. For the rest of the workload, the performance gains are smaller. Figure 6.20 shows that speed improvements stem mostly from a decrease in network latency. Variations in hit ratios and mapping of pages to nodes are the cause of the small variations in the time spent on instruction fetching and execution and shared-data references.

6.4.1 Throughput and Latency

Figure 6.21 plots throughput for `mp3d()` and `paths()`. The speed of these two programs is clearly limited by network latency as can be seen in Figure 6.20. On a ring, throughput can be pushed to 78Mbytes/s by doubling the processor clock speed, as discussed in Section 4.2.4 (page 53). `mp3d()` attains 93Mbytes/s on the 16-node mesh and 73Mbytes/s (84Mbytes/s) on the 64-node mesh (cube). `paths()` achieves 89Mbytes/s (98Mbytes/s) on the 64-node mesh (cube).

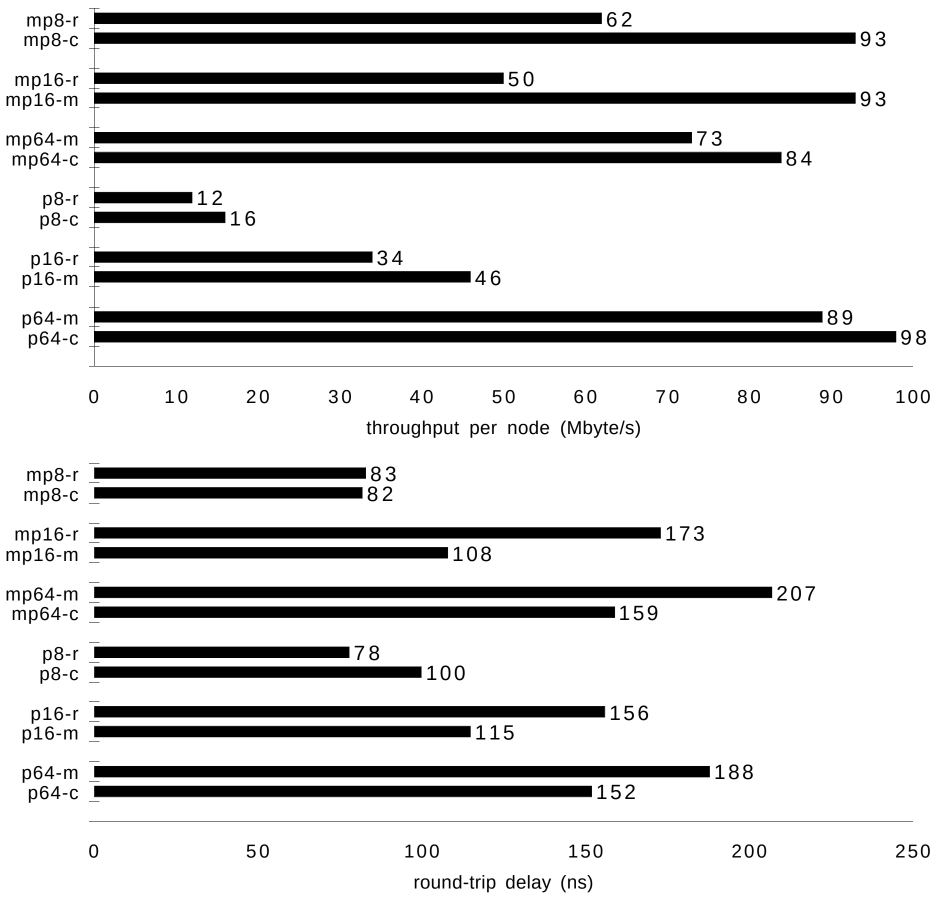


Figure 6.21: Processor throughput (top) and round-trip delay (bottom) for `mp3d()` (`mp`) and `paths()` (`p`), 256Kbytes caches. The suffixes `-r`, `-m` and `-c` stand for ring, mesh and cube, respectively.

The question begging an answer concerns the apparent limit of 100Mbytes/s throughput. Given the evidence provided in Section 4.2.4 and Figure 6.20, it is fair to say that the network places a limit on performance, this limit being node throughput at about 100Mbytes/s. Figure 6.21 also shows the round-trip delay for `mp3d()` and `paths()`. Figure 6.21 explains the performance improvement achieved by higher dimensional networks. Two effects cooperate towards better performance. To a decrease in latency, and increase in network capacity, corresponds an increase in throughput and hence better performance.

The above conclusion indicates that processor throughput is limited at about 100Mbytes/s. To confirm that, more simulations were done with a 200MHz processor clock to increase the rate of network requests. `paths()` was run on 16-node ring and mesh, and 64-node mesh and cube. The results are shown in Figure 6.22. Throughput does go beyond 100Mbytes/s on the 64-node cube. Table 6.1 shows traffic levels, throughput and round-trip delays for `paths()` with 100 and 200MHz processor clock frequencies. Section 4.3 (page 57) discusses the effects of network saturation in SCI-rings. Saturation occurs for link traffic levels at about 600–700Mbytes/s and that places a limit on system performance. The traffic levels recorded in Table 6.1 for the 200MHz *mesh* are near those where network saturation occurs. Traffic levels for the 200MHz *cube* are lower than saturation and throughput reaches 121Mbytes/s. Thus, throughput can be higher than 100Mbytes/s provided network traffic is kept below saturation levels.

		tput.	dly.	peak	avg.	peak	avg.	peak	avg.
dimension		—		X		Y		Z	
mesh	100MHz	89	188	594	434	430	362	—	—
	200MHz	106	194	687	512	483	429	—	—
cube	100MHz	98	152	406	233	324	217	233	182
	200Mhz	121	153	466	283	370	265	289	224

Table 6.1: Processor throughput, round-trip delay and link traffic, for `paths()`, with 100 and 200MHz processor clock.

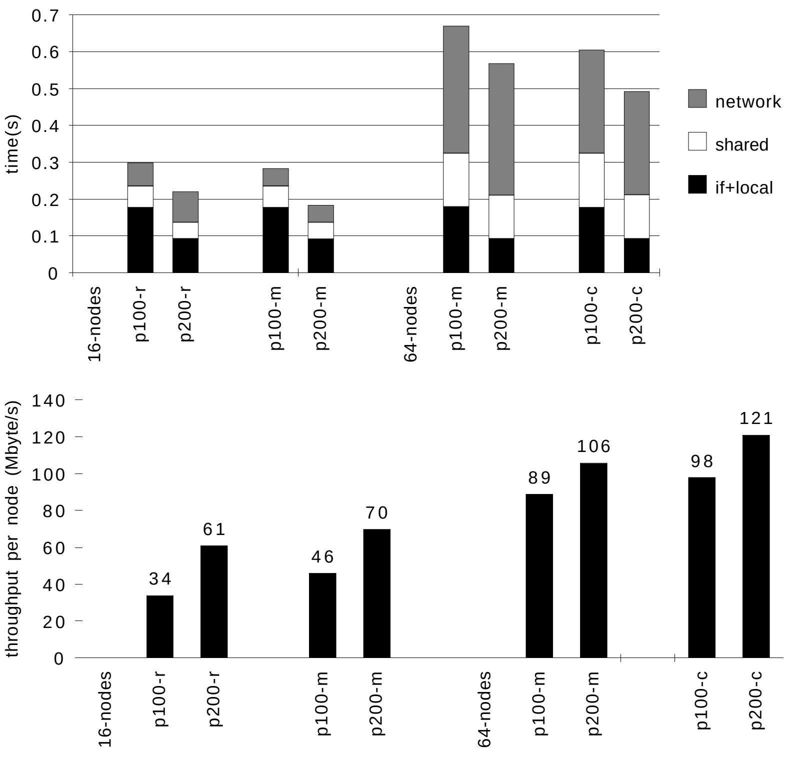


Figure 6.22: Execution time and processor throughput for `paths()`, processor clock of 100 and 200MHz, 256Kbytes caches. The suffixes `-r`, `-m` and `-c` stand for ring, mesh and cube, respectively.

6.4.2 Cache Size and Network Dimensionality

An interesting question concerns the relationship between cache size and network dimensionality. Larger caches cause less network traffic because of their higher hit ratios. Lower traffic means decreases in network latency and that in turn tends to increase the rate of memory and network requests. Higher dimensionality networks have inherently lower latencies and higher bandwidth. These effects also tend to increase the rate of network requests. However, the higher traffic levels are supported better because of the network's higher capacities.

Given a limited budget, the architect has to weigh two options: either use the largest possible cache size or increase the network dimensionality. Both options have an associated cost and both improve performance. In order to gauge the effects of both cache size and network dimensionality, the execution times of `mp3d()` and `paths()` are shown in Figure 6.23 for 128, 256 and 512Kbytes caches and 4-, 8-, 16- and 64-node multiprocessors. These two programs generate the highest traffic levels in the workload. `mp3d()`'s performance benefits more from a higher-dimensional network than from bigger caches. For instance, the 64-node cube is 14 to 16% faster than the mesh while the differences in performance for the three cache sizes is less than 1%. The hit ratios are roughly the same for the three cache sizes on 64-node rings. The behaviour of the 16-node machines is similar. The two smaller multiprocessors suffer from the higher costs of switching dimension. `paths()` behaves differently. The systems with 512Kbytes caches show little performance difference (within 2%) in all sizes and topologies considered. The performance of the two larger machines with 128 and 256Kbytes caches improves by about 10% on the higher dimensionality networks.

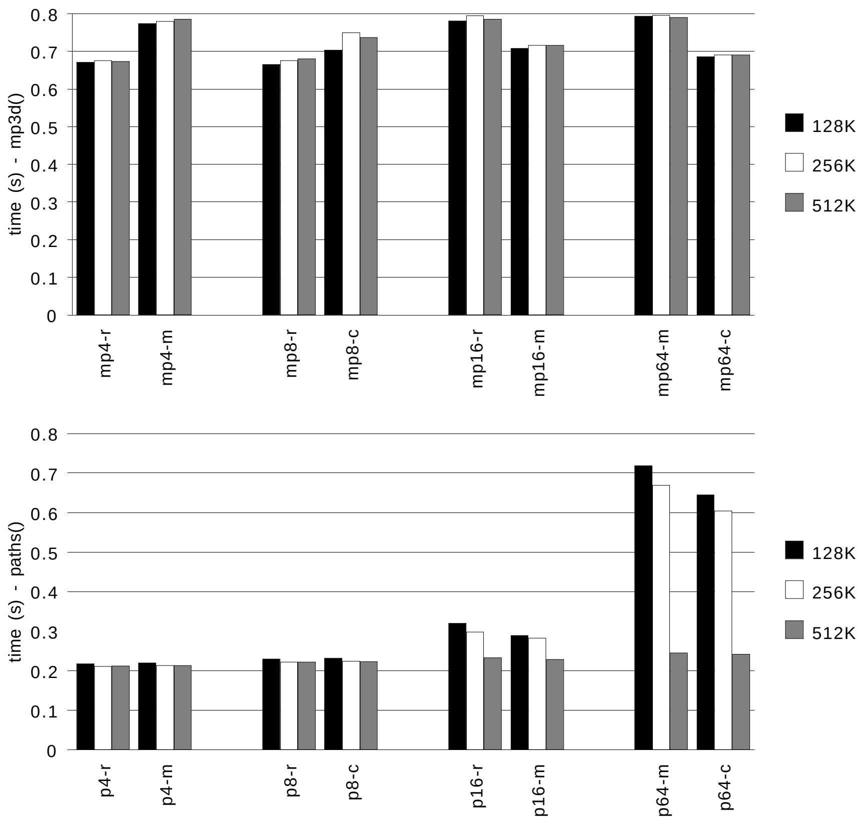


Figure 6.23: Performance of `mp3d()` (top) and `paths()` (bottom) with 128K, 256K and 512Kbytes caches, on 4-, 8-, 16- and 64-node multiprocessors.

Coda

Higher hardware costs should be balanced against the potential performance gains. The performance improvements due higher dimensionality, for the experiments reported here, are in the range of 10% to 15%, *for applications that generate high levels of network traffic.* For programs that make better use of shared-data, the performance gains are negligible.

Chapter 7

Conclusion

This dissertation contains a detailed performance evaluation study of an SCI-based shared memory multiprocessor. Previous studies of SCI based systems have concentrated on network performance and to a large extent ignored the influence of the cache coherence protocol. Here, the interactions between interconnection network and cache coherence protocol were investigated. The results of the detailed simulations are summarised below.

A multiprocessor system was "implemented" in the simulator with components compatible with the current levels of performance. Several architectural parameters were investigated, namely machine size, secondary cache size, processor clock speed and interconnection topology. Machines were simulated with one, two, four, eight, sixteen and sixty-four 100MIPS processors. In order to reproduce accurately the interleaving of the memory references in a NUMA architecture, the architecture simulator is driven by reference streams generated as a by-product of the execution of real programs. The simulated threads are scheduled for execution according to the state of the simulated multiprocessor and the actual delays incurred by references to remote memory and cache coherency actions.

Summary of Results

The workload used in the experiments consists of three programs from the SPLASH suite (`chol()`, `mp3d()` and `water()`) and three parallel loops, namely Gaussian elimination (`ge()`), matrix multiplication (`mmult()`) and all-to-all minimum cost paths (`paths()`). Two of the programs are ill suited for execution on physically distributed memory. `mp3d()` has low hit ratios and its data is highly migratory, causing high levels of cache coherence activity and network traffic. This program exhibits poor performance in every published experiment seen by the author. It is however very useful to expose architectural bottlenecks, as is the case with network saturation (see below). The data used by `paths()` has a high degree of read-sharing and writes to shared data often cause the purging of long sharing-lists. This also causes high levels of network traffic and, for the smaller cache sizes, high levels of coherence activity. These two programs do drive the network into saturation and their performance is, in most of the experiments, limited by network bandwidth and delays.

The other four programs have more regular behaviour and better coherent cache hit ratios. The performance penalties imposed by the cache coherence protocol and interconnect are rather small. The results of the simulation with 64Kbytes caches are a little worse than those with the larger cache sizes (128K–512Kbytes). With the larger caches, the overheads imposed by the cache coherence protocol are always smaller than 5% of the execution time. The losses caused by network latencies are under 10% of the execution time, with higher losses occurring on 16-node rings.

An important figure of merit of an interconnect is node throughput, defined as the network bandwidth available to processing elements. For rings with processors and memory hierarchy as simulated, the experiments revealed that raw processor throughput is limited at about 80Mbytes/s because of network saturation. Data-only throughput is about 20 to 30% of raw throughput. Given that under 14% of all packets injected into the ring carry 64 bytes of data while all except echo packets carry cache coherency information, raw throughput is a better measure of overall system performance.

High levels of network traffic cause queue backlogs in the link interfaces with round-trip delays increasing by as much as 25% as a consequence. For `mp3d()` and `paths()`, network saturation occurs for link traffic at 600 to 700Mbytes/s, for 8- and 16-node rings, and this in turn limits node throughput at 80Mbytes/s. Unlike analytical results produced by others [SGV92], the relationship between throughput and latency was found to be linear for 2-, 4- and 8-node rings. For 16-node rings, the relationship is a parabolic curve with a small quadratic coefficient. The difference between the two sets of results stems from the feedback effects of memory latencies increasing with traffic levels and holding down the rate of network requests by processors. The simulation results for SCI-rings indicate that, for hardware and software with characteristics similar to those investigated here, the maximum efficient ring size is between eight and sixteen. The scalability in these small systems (1, 2, 4, 8, and 16 processors) was found to be fairly good.

A comparison between DASH [LLJ$^+$92] and an SCI-based multiprocessor with similar architectural parameters reveals that SCI's higher bandwidth and lower latencies yield better performance on programs that cause higher network traffic. On programs that generate little traffic the systems exhibit similar behaviour. The programs in the workload simulated do not have high degrees of write-sharing and thus the cache coherence protocols show similar performance. A comparison of medium size machines built with an SCI-based interconnect to contemporary machines, such as DASH, the Express Ring and the KSR1, suggest that SCI's low latency and high bandwidth make it a suitable and efficient interconnect.

Systems of 64-nodes were also investigated. The topologies simulated were a mesh (2x2, 4x4, 8x8) and a cube (2x2x2, 4x4x4). The programs in the workload exhibit similar patterns of behaviour on SCI-meshes and SCI-cubes as those of SCI-rings. `mp3d()` and `paths()` tend to generate very high levels of cache coherence activity and network traffic. The other four programs show good scalability and suffer small losses in performance because of the cache coherence protocol and interconnection network. Because of increases in the data-set sizes, larger cache sizes produce better performances.

No significant relationship between cache size and network dimensionality was found. Meshes and cubes can sustain somewhat higher processor throughput than is the case for rings, mostly because of the increases in network capacity. The programs that produced node throughputs of 60–80Mbytes/s on rings produce 90–100Mbytes/s on meshes and cubes, with cubes supporting 11–16% higher throughputs than meshes. This is because of the inherently higher capacities and lower latencies of cubes. In terms of overall performance, cubes are 10–15% faster than meshes with programs that generate high levels of network traffic, that is, drive the network near to saturation. For programs that produce low levels of traffic, the differences between meshes and cubes are negligible.

An analytical model of the ring-based multiprocessor was described and used to assess the cost of flushing shared-data lines from the caches, and of purging sharing-lists. The performance of the multiprocessor degrades but is acceptable for high levels of conflict misses. The performance degrades badly for high levels of write-sharing – over 300% in one case. The analytical model provides reasonably accurate performance predictions for "well-behaved" programs and qualitatively good predictions for programs with more extreme behaviour.

The results presented above indicate that the Scalable Coherent Interface is a good implementation of the shared-memory abstraction on a machine with physically distributed memory. For the architectural parameters and workload investigated, the cache coherence protocol proved to be efficient and the interconnect provided a high-bandwidth low-latency path between processors and memory.

Further Work

There is still work to be done in the performance evaluation of SCI based multiprocessors. That work can be pursued along two directions. First, further investigation of small systems is needed. Second, the evaluation of systems with hundreds of processors is necessary in order to assess the scalability of SCI-based multiprocessors. Some of the issues that deserve investigation concern both small and large systems. For simulations of large systems, the programs in the workload will need to be adapted, rewritten or replaced because they were designed and coded for medium size machines (32–64 processors). These codes are unlikely to scale up well to hundreds of processors without extensive rewriting.

Some architectural devices can be added to the simulator in order to improve the performance of both SCI meshes and cubes. A better mapping of data to nodes is an important optimisation because it can further reduce the distance between requesters and responders. This entails changes to the programs to reflect different mapping strategies. The efficiency of the synchronisation mechanisms employed here can be improved as well, either by software methods or by adding SCI's QOLB primitive (Queue On Locked Bit) to the simulator [AGGW94].

The simulation of SCI cubes has shown that higher dimensionality networks can sustain higher processor throughput because of lower network latency and higher capacity. The simulated multiprocessor can be modified to take advantage of the unused capacity by the addition of write-buffers between primary and secondary caches. Another alternative is to use multi-threading to hide the latency of remote memory requests. Improvements in processor performance by the addition of these devices would cause increases in network congestion. The question then is how much room for improvement there is before the network becomes the bottleneck.

One of SCI's major advantages is the scalability that is built into the communication and cache coherence protocols. Because of the high computational cost of the simulation runs, the simulation of very large systems will need a different approach. The simulation technique used here produces accurate results but its computational cost is very high. Two alternatives seem attractive. One is the direct simulation of the processors, thus avoiding interactions with the operating system [BDNS93]. The other is to use a customisable synthetic workload. While not strictly realistic, proper tuning of parameters can produce insightful results [HS94].

With 512 or 1024 nodes, a more thorough study of the relationship between cache size and dimensionality is possible since there will be more data points on which to draw comparisons. Another problem relates to synchronisation actions on large machines. Different mechanisms can be simulated and compared. To perform these experiments, either a synthetic workload will be used, or new programs suitable for large scale shared-memory will have to be employed.

Bibliography

[Aga91] Anant Agarwal. Limits on interconnection network performance. *IEEE Trans on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[AGGW94] N M Aboulenein, J R Goodman, S Gjessing, and P J Woest. Hardware support for synchronisation in the Scalable Coherent Interface (SCI). In *Proc of the 8th Intl Parallel Processing Symposium*, pages 141–150, Cancún, 1994. IEEE Comp Soc Press.

[ASHH88] A Agarwal, R Simoni, J Hennessy, and M Horowitz. An evaluation of directory schemes for cache coherence. In *Proc 15th Intl Symp on Computer Arch*, pages 280–289, May 1988.

[BD91] Luiz A Barroso and Michel Dubois. Cache coherence on a slotted ring. In *Proc 1991 Intl Conf Parallel Processing*, volume 1, pages 230–237, August 1991.

[BD93] Luiz A Barroso and Michel Dubois. The performance of cache-coherent ring-based multiprocessors. In *Proc 20th Intl Symp on Computer Arch*, pages 268–277. ACM Comp Arch News 21(2), May 1993.

[BDMR92] J A C Bogaerts, R Divià, H Müller, and J F Renardy. SCI based data acquisition architectures. *IEEE Trans on Nuclear Sciences*, 39(2), April 1992.

[BDNS93] M Brorsson, F Dahlgren, H Nilsson, and P Stenström. The CacheMire test bench – a flexible and effective approach for simulation of multiprocessors. Tech Report Dt-159, Dept of Computer Engineering, Lund Univ, March 1993. In Proc of the 26th Annual Simulation Symposium, Arlington, USA, March 1993.

[Bel92] Gordon Bell. Ultracomputers: A teraflop before its time. *Comm of the ACM*, 35(8):27–47, August 1992.

[BFKR92] H Burkhardt, S Frank, B Knobe, and J Rothnie. Overview of the KSR1 computer system. Tech Report KSR-TR-9202001, Kendall Square Research, Boston, 1992.

[BGY87] L N Bhuyan, D Ghosal, and Q Yang. Approximate analysis of single and multiple ring networks. *IEEE Trans on Computers*, C-38(7):1027–1040, July 1987.

[Bit92] Philip Bitar. The weakest memory-access order. *Journal of Parallel and Distributed Computing*, 2(15):305–331, March 1992.

[BKB90] H O Bugge, E H Kristiansen, and B O Bakka. Trace driven simulations for a two-level cache design in open bus systems. In *Proc 17th Intl Symp on Computer Arch*, pages 250–259. ACM Comp Arch News 18(2), May 1990.

[BKW90] A Borg, R E Kessler, and D W Wall. Generation and analysis of very long address traces. In *Proc 17th Intl Symp on Computer Arch*, pages 270–279. ACM Comp Arch News 18(2), May 1990.

[BS92] Mats Brorsson and Per Stenström. Visualising sharing behaviour in relation to shared memory management. Tech Report Dt-150, Dept of Computer Engineering, Lund Univ, December 1992. In Proc of 1992 Intl Conf on Parallel and Distributed Systems, Hsinchu, Taiwan, December 1992, pages 528–536.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc 15th Intl Symp on Computer Arch*, pages 73–80, May 1988.

[CA94] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proc 21st Intl Symp on Computer Arch*, pages 314–324. ACM Comp Arch News 22(2), April 1994.

[CDK$^+$94] A L Cox, S Dwarkadas, P Kaleher, H Lu, R Rajamony, and W Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc 21st Intl Symp on Computer Arch*, pages 106–117. ACM Comp Arch News 22(2), April 1994.

[CFKA90] D Chaiken, C Fields, K Kwihara, and A Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–59, June 1990.

[CKA91] D Chaiken, J Kubiatowicz, and A Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Intl Conf on Arch'l Support for Progr*

Lang and Oper Sys, pages 224–234. ACM Comp Arch News 19(2), April 1991.

[Dal90] William J Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans on Computers*, C-39(6):775–785, June 1990.

[DPL80] N Deo, C Y Pang, and R E Lord. Two parallel algorithms for shortest path problems. Tech Report CS-80-059, Washington State Univ, March 1980.

[DS87] William J Dally and Charles L Seitz. Deadlock-Free message routing in multiprocessor interconnection networks. *IEEE Trans on Computers*, C-36(5):547–553, May 1987.

[DS90] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans on Software Engineering*, 16(6):660–673, June 1990.

[FBYR89] A Forin, J Barrera, M Young, and R Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for MACH. In *Proc of the Winter USENIX Conf*, January 1989.

[FVS92] K Farkas, Z Vranesic, and M Stumm. Cache consistency in hierarchical ring-based multiprocessors. Tech Report EECG TR-92-09-01, Univ of Toronto, 1992. Also in Proc of Supercomputing '92.

[GGH91] K Gharachorloo, A Gupta, and J Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *4th Intl Conf on Arch'l Support for Progr Lang and Oper Sys*, pages 245–257. ACM Comp Arch News 19(2), April 1991.

[GHG$^+$91] A Gupta, J L Hennessy, K Gharachorloo, T Mowry, and W-D Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proc 18th Intl Symp on Computer Arch*, pages 254–263. ACM Comp Arch News 19(3), May 1991.

[GLL$^+$90] K Gharachorloo, D Lenoski, J Laudon, P Gibbons, and J L Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc 17th Intl Symp on Computer Arch*, pages 15–26. ACM Comp Arch News 18(2), May 1990.

[GNWZ91] D Grunwald, G J Nutt, D Wagner, and B Zorn. A parallel execution evaluation testbed. Tech Report CU-CS-560-91, Dept of Computer Science, Univ of Colorado, November 1991.

[Goo91] James R Goodman. Cache consistency and sequential consistency. Tech Report 1006, Computer Sciences Dept, Univ of Wisconsin–Madison, February 1991. Also published as SCI Committee Report 61, March 1989.

[GW88] J R Goodman and P Woest. The Wisconsin multicube: A new large-scale cache coherent multiprocessor. In *Proc 15th Intl Symp on Computer Arch*, pages 422–431, May 1988.

[Hag92] Erik Hagersten. *Toward Scalable Cache Only Memory Architectures.* PhD dissertation, Swedish Institute of Computer Science, October 1992. ISBN 91-7170-103-6.

[HALH91] E Hagersten, P Andersson, A Landin, and S Haridi. A performance study of the DDM – a cache-only architecture. Tech Report R91:17, Swedish Inst of Computer Science, November 1991.

[Hil90] Mark D Hill. What is scalability? *ACM SIGARCH Comp Arch News*, 18(4):18–21, December 1990.

[HN88] A Hooper and R Needham. The cambridge fast ring networking system. *IEEE Trans on Computers*, C-37(10):1214–1224, October 1988.

[HP90] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1st edition, 1990. ISBN 1-55860-069-8.

[HS94] Mark Holliday and Michael Stumm. Performance evaluation of hierarchical ring-based shared memory multiprocessors. *IEEE Trans on Computers*, C-43(1):52–67, January 1994.

[HT94a] Roberto A Hexsel and Nigel P Topham. The performance of SCI memory hierarchies. In *Proc of the Intl Workshop on Support for Large Scale Shared Memory Architectures*, pages 1–17, Cancún, April 1994. With 8th IPPS.

[HT94b] Roberto A Hexsel and Nigel P Topham. The performance of SCI memory hierarchies. Tech Report CSR-30-94, Dept of Computer Science, Univ of Edinburgh, February 1994.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993. ISBN 0-07-031622-8.

[IEE91] IEEE. *Futurebus$^+$: Logical Layer Specification, 896.1-1991.* IEEE Microprocessor Standards Subcommittee, 1991.

[IEE92] IEEE. *IEEE Std 1596-1992 – Standard for Scalable Coherent Interface.* IEEE, 1992.

[IT89] Roland N Ibbett and Nigel P Topham. *Architecture of High Performance Computers*, volume 2. Macmillan, 1989. ISBN 0-333-48988-8.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley, 1991. ISBN 0-471-50336-3.

[JG91] Ross E Johnson and James R Goodman. Interconnect topologies with point-to-point rings. Tech Report 1058, Computer Sciences Dept, Univ of Wisconsin–Madison, December 1991.

[JG92] Ross E Johnson and James R Goodman. Synthesising general topologies from rings. In *Proc of the Intl Conf on Parallel Processing (ICPP92)*, volume I – Architecture, pages 86–95, 1992.

[Joh93] Ross E Johnson. *Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors.* PhD dissertation, Computer Sciences Dept, Univ of Wisconsin–Madison, February 1993. Also Tech Report 1136.

[KEL91] E J Koldinger, S J Eggers, and H M Levy. On the validity of trace driven simulations for multiprocessors. In *Proc 18th Intl Symp on Computer Arch*, pages 244–253. ACM Comp Arch News 19(3), May 1991.

[LAD$^+$92] C E Leiserson, Z S Abuhamdeh, D C Douglas, C R Feynman, M N Ganmukhi, J V Hill, W D Hillis, B C Kuszmaul, M A St Pierre, D S Wells, M C Wong, S Yang, and R Zak. The network architecture of the connection machine CM-5. In *Proc of the 4th Annual ACM Symp on Parallel Algorithms and Architecture*, San Diego, USA, 1992. ACM SIGACT, SIGARCH.

[Lam79] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans on Computers*, C-28(9):690–691, September 1979.

[Lei85] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans on Computers*, C-34(10):892–901, oct 1985.

[LH89] Kay Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans on Computer Systems*, 7(4):229–359, November 1989.

[LLG$^+$90] D Lenoski, J Laudon, K Gharachorloo, A Gupta, and J L Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In

Proc 17th Intl Symp on Computer Arch, pages 148–159. ACM Comp Arch News 18(2), May 1990.

[LLJ+92] D Lenoski, J Laudon, T Joe, D Nakahira, L Stevens, A Gupta, and J Hennessy. The DASH prototype: Implementation and performance. In *Proc 19th Intl Symp on Computer Arch*, pages 92–103. ACM Comp Arch News 20(2), May 1992.

[MB92] Daniel Menascé and Luiz A Barroso. A methodology for performance evaluation of parallel applications on multiprocessors. *Journal of Parallel and Distributed Computing*, 2(14):1–14, January 1992.

[MS91] J M Mellor-Crummey and M L Scott. Synchronisation without contention. In *4th Intl Conf on Arch'l Support for Progr Lang and Oper Sys*, pages 269–278. ACM Comp Arch News 19(2), April 1991.

[MSW94] H L Muller, P W A Stallard, and D H D Warren. An evaluation study of a link-based data diffusion machine. In *Proc of the Intl Workshop on Support for Large Scale Shared Memory Architectures*, pages 115–128, Cancún, April 1994. With 8th IPPS.

[NS92] Hakan Nilsson and Per Stenström. The scalable tree protocol – a cache coherence approach for large-scale multiprocessors. Tech Report Dt-149, Dept of Computer Engineering, Lund Univ, December 1992. In Proc of 4th IEEE Symp on Parallel and Distributed Processing, December 1992, pages 498–506.

[NS93] Hakan Nilsson and Per Stenström. Performance evaluation of link-based cache coherence schemes. Tech Report Dt-151, Dept of Computer Engineering, Lund Univ, January 1993. In Proc of the 26th Hawaii Intl Conf on System Sciences, January 1993, pages I-486–495.

[OMB91] O A Olukotun, T N Mudge, and R B Brown. Implementing a cache for a high-performance GaAs microprocessor. In *Proc 18th Intl Symp on Computer Arch*, pages 138–147. ACM Comp Arch News 19(3), May 1991.

[Prz90] Steven A Przybylski. *Cache and Memory Hierarchy Design: a Performance-Directed Approach.* Morgan Kaufmann, 1990. ISBN 1-55860-136-8.

[RAK89] U Ramachandran, M Ahamad, and M Y A Khalidi. Coherence of distributed shared memory: Unifying synchronisation and data transfer. In *Proc of the Intl Conf on Parallel Processing (ICPP89)*, volume II, pages 160–169, 1989.

[Sco92] Steven L Scott. *Toward the Design of Large-Scale, Shared-Memory Multiprocessors.* PhD dissertation, Computer Sciences Dept, Univ of Wisconsin–Madison, July 1992. Also Tech Report 1100.

[Sei85] Charles L Seitz. The cosmic cube. *Comm of the ACM*, 28(1), January 1985.

[SG91] Steven L Scott and James R Goodman. Performance of pipelined K-ary N-cube networks. Tech Report 1010, Computer Sciences Dept, Univ of Wisconsin–Madison, February 1991.

[SGV92] S L Scott, J R Goodman, and M K Vernon. Performance of the SCI ring. In *Proc 19th Intl Symp on Computer Arch*, pages 403–414. ACM Comp Arch News 20(2), May 1992.

[SJG92] P Stenström, T Joe, and A Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc 19th Intl Symp on Computer Arch*, pages 80–91. ACM Comp Arch News 20(2), May 1992.

[SPG91] A Silberschatz, J L Peterson, and P B Galvin. *Operating Systems Concepts.* Addison-Wesley, 3rd edition, 1991. ISBN 0-201-54873-9.

[Ste90] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[Sto90] Harold S Stone. *High-Performance Computer Architecture.* Addison-Wesley, second edition, 1990. ISBN 0-201-51377-3.

[SWG91] J P Singh, W-D Weber, and A Gupta. SPLASH: Stanford ParalleL Applications for SHared-memory. Technical Report CSL-TR-91-469, Computer Science Dept, Stanford Univ, April 1991. Also in ACM SIGARCH Comp Arch News 20(1).

[Tan89] Andrew S Tanenbaum. *Computer Networks.* Prentice-Hall, 2nd edition, 1989. ISBN 013166836-6.

[TD91] Manu Thapar and Bruce Delagi. Cache coherence for large scale shared memory multiprocessors. *ACM SIGARCH Comp Arch News*, 19(1):114–191, March 1991. Reprinted from Proc of SPAA 1990.

[VIT90] VITA. *64-Bit VMEbus Specification – Edition D.* VME Bus International Trade Association and IEEE P1014 Working Group, January 1990.

[VSLW91] Z Vranesic, M Stumm, D Lewis, and R White. Hector: a hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–78, January 1991.

[WG89] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *3rd Intl Conf on Arch'l Support for Progr Lang and Oper Sys*, pages 243–256. ACM Comp Arch News 17(2), April 1989.

[WHL93] C E Wu, Y Hsu, and Y-H Liu. A quantitative evaluation of cache types for high-performance computer systems. *IEEE Trans on Computers*, C-42(10):1154–1162, October 1993.

[WLI$^+$94] A W Wilson Jr, R P LaRowe Jr, R J Ionta, R P Valentino, B Hu, P R Breton, and P Lau. Update propagation in the Galactica Net distributed shared memory architecture. In *Proc of the Intl Workshop on Support for Large Scale Shared Memory Architectures*, pages 18–31, Cancún, April 1994. With 8th IPPS.

Appendix A

Performance Data

This appendix contains the statistics for the experiments described in Chapters 4 and 6. The tables are grouped by topology, and within each topology, by program. The the statistics collected in the tables are defined in Sections 4.1 and 6.2.3. Table A.1 defines the meanings of the columns of the reference count tables. Table A.2 defines the meanings for the hit ratio tables. Table A.3 defines the meanings for the traffic and timing tables.

tag	meaning
cSz	cache size
N	machine/ring size
shdRD	shared-data read
shdWR	shared-data write
lclRD	local-data read
lclWR	local-data write
i-fetch	instruction fetch

Table A.1: Per node reference count tables.

tag	meaning
cSz	cache size
N	machine/ring size
lrpch	local read in primary cache
lrcch	local read in coherent cache
lwcch	local write in coherent cache
srpch	shared-data read in primary cache
srcch	shared-data read in coherent cache
swcch	shared-data write in coherent cache
ifpch	i-fetch in primary cache
ifcch	i-fetch in coherent cache
flush	lines flushed per coherent cache reference
purge	sharing-lists purged per coherent cache write
shl	sharing-list length

Table A.2: Hit ratio tables.

tag	meaning
cSz	cache size
N	machine/ring size
n	dimension (meshes and cubes)
lkav	average link traffic
lkmx	maximum link traffic
txav	average output buffer traffic
txmx	maximum output buffer traffic
psav	average bypass buffer traffic
psmx	maximum bypass buffer traffic
rtdly	round-trip delay
rtime	execution time
ntwF	fraction of the time due to network latency
shdF	fraction of the time due to shared-data references (RD+WR+synch)
lclF	fraction of the time due to local references (RD+WR+ifetch)

Table A.3: Network traffic and timing tables.

A.1 SCI Rings

A.1.1 chol() – DASH Parameters

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
256K	1	8444	1863	22763	8303	71750
	2	9883	2939	6641	2168	38930
	4	6875	2075	2049	543	21027
	8	4639	1191	1111	166	14313
	16	2904	556	1172	157	11106

Table A.4: Per node reference counts for `chol()` ($\times$1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
256K	1	0.654	0.999	1.000	0.775	0.973	0.983	1.000	0.000	0.797	0.000	0.0
	2	0.692	0.999	1.000	0.711	0.987	0.990	1.000	0.000	0.662	0.553	1.0
	4	0.754	0.999	0.999	0.708	0.987	0.990	1.000	0.000	0.621	0.694	1.0
	8	0.863	0.999	1.000	0.753	0.987	0.989	1.000	0.000	0.568	0.810	1.0
	16	0.875	0.999	0.998	0.816	0.983	0.986	1.000	0.000	0.516	0.875	1.0

Table A.5: Hit ratios for `chol()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.260	0.000	0.152	0.848
	2	1.5	1.5	1.2	1.2	0.4	0.4	42.0	5.315	0.001	0.456	0.541
	4	5.6	5.7	2.1	2.3	3.5	3.7	52.7	2.801	0.003	0.625	0.371
	8	14.1	14.1	2.5	2.7	11.5	11.8	79.5	1.581	0.007	0.658	0.336
	16	30.2	30.3	2.7	4.0	27.5	28.2	130.2	0.953	0.012	0.528	0.460

Table A.6: Traffic and timing for `chol()`.

A.1.2 mp3d() – DASH Parameters

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
256K	1	3620	2343	10754	2339	35059
	2	3755	1171	5382	1172	21439
	4	1594	585	2694	587	10164
	8	1041	293	1352	295	5585
	16	1098	147	677	148	3951

Table A.7: Per node reference counts for `mp3d()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
256K	1	0.818	0.997	0.999	0.338	0.951	0.950	1.000	0.000	0.911	0.000	0.0
	2	0.826	0.995	0.998	0.681	0.892	0.890	1.000	0.000	0.608	0.710	1.0
	4	0.830	0.993	0.997	0.624	0.853	0.848	1.000	0.000	0.513	0.910	1.0
	8	0.832	0.992	0.997	0.709	0.824	0.812	1.000	0.000	0.472	0.963	1.0
	16	0.833	0.992	0.997	0.860	0.770	0.744	1.000	0.000	0.430	0.985	1.1

Table A.8: Hit ratios for `mp3d()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.833	0.000	0.421	0.580
	2	12.5	12.6	9.5	9.7	2.9	3.0	44.2	2.957	0.013	0.519	0.469
	4	43.6	44.0	15.8	16.9	27.7	29.2	54.0	1.608	0.028	0.541	0.429
	8	105.2	105.7	18.9	20.9	86.3	89.8	82.2	0.900	0.052	0.563	0.384
	16	228.5	229.1	20.6	26.2	207.9	211.3	152.9	0.580	0.104	0.597	0.299

Table A.9: Traffic and timing for `mp3d()`.

A.1.3 water() – DASH Parameters

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
256K	1	20173	3071	200076	46251	475688
	2	12619	1536	100038	23125	242909
	4	8163	768	50019	11563	125162
	8	4093	384	25010	5781	62604
	16	2414	192	12505	2891	32038

Table A.10: Per node reference counts for `water()` (×1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
256K	1	0.912	0.999	1.000	0.917	0.955	0.988	1.000	0.000	0.640	0.000	0.0
	2	0.913	1.000	1.000	0.934	0.956	0.988	1.000	0.000	0.637	0.119	1.0
	4	0.913	0.999	1.000	0.949	0.884	0.948	1.000	0.000	0.539	0.765	1.0
	8	0.913	1.000	1.000	0.949	0.839	0.926	1.000	0.000	0.507	0.907	1.0
	16	0.913	0.997	0.999	0.957	0.826	0.922	1.000	0.000	0.369	0.925	1.0

Table A.11: Hit ratios for `water()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	41.960	0.000	0.047	0.953
	2	0.2	0.2	0.1	0.1	0.1	0.1	44.3	21.200	0.000	0.058	0.941
	4	2.8	2.8	1.0	1.2	1.8	1.9	52.9	10.900	0.002	0.083	0.915
	8	9.2	9.3	1.6	2.2	7.6	7.8	76.5	5.504	0.004	0.090	0.905
	16	20.9	21.1	1.8	3.3	19.1	19.4	127.7	2.812	0.008	0.102	0.890

Table A.12: Traffic and timing for `water()`.

A.1.4 chol()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	8444	1863	22763	8303	71750
	2	9596	2933	6451	2175	37254
	4	6963	1990	2343	628	22257
	8	4805	1205	1018	152	14287
	16	1931	315	1554	398	9160
128K	1	8444	1863	22763	8303	71750
	2	9818	2984	6230	2124	37265
	4	9348	1954	2813	664	29136
	8	4671	1174	1100	184	14209
	16	1904	308	1571	405	9216
256K	1	8444	1863	22763	8303	71750
	2	9609	2942	6333	2165	36918
	4	6561	2002	2071	616	20253
	8	3988	1177	845	180	11576
	16	2387	558	774	155	8132
512K	1	8444	1863	22763	8303	71750
	2	9738	2962	6215	2145	36952
	4	6689	2052	1989	566	20327
	8	3999	1186	851	172	11713
	16	1785	324	1426	389	8336
8M	1	8444	1863	22763	8303	71750
	2	9743	2988	6230	2119	37033
	4	6645	2041	1991	577	20258
	8	3919	1186	831	172	11394
	16	2344	555	771	157	8028

Table A.13: Per node reference counts for `chol()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.651	0.998	0.999	0.732	0.926	0.962	0.998	0.450	0.660	0.000	0.0
	2	0.678	0.999	0.999	0.682	0.970	0.985	0.998	0.441	0.645	0.252	1.0
	4	0.743	0.998	0.998	0.707	0.950	0.974	0.995	0.181	0.571	0.203	1.0
	8	0.860	0.994	0.996	0.746	0.962	0.979	0.996	0.219	0.558	0.343	1.0
	16	0.752	0.978	0.990	0.829	0.938	0.965	0.997	0.183	0.365	0.521	1.0
128K	1	0.651	0.999	0.999	0.733	0.954	0.973	0.998	0.606	0.694	0.000	0.0
	2	0.675	0.999	0.999	0.686	0.980	0.988	0.998	0.618	0.646	0.387	1.0
	4	0.773	0.999	0.999	0.784	0.943	0.970	0.995	0.145	0.533	0.221	1.0
	8	0.843	0.999	0.999	0.745	0.964	0.980	0.997	0.231	0.538	0.415	1.0
	16	0.753	0.998	0.998	0.831	0.952	0.968	0.998	0.403	0.523	0.635	1.0
256K	1	0.650	0.999	1.000	0.735	0.977	0.983	0.999	0.912	0.797	0.000	0.0
	2	0.673	0.999	0.999	0.684	0.988	0.990	0.999	0.913	0.655	0.553	1.0
	4	0.713	0.999	0.999	0.693	0.988	0.990	0.999	0.916	0.600	0.729	1.0
	8	0.797	0.999	1.000	0.706	0.987	0.989	0.999	0.920	0.557	0.816	1.0
	16	0.809	0.999	0.998	0.766	0.983	0.986	0.999	0.894	0.527	0.860	1.0
512K	1	0.651	1.000	1.000	0.736	0.989	0.992	0.999	0.954	0.645	0.000	0.0
	2	0.671	1.000	1.000	0.687	0.990	0.991	0.999	0.953	0.556	0.740	1.0
	4	0.728	0.999	0.999	0.692	0.989	0.990	0.999	0.953	0.552	0.812	1.0
	8	0.810	1.000	1.000	0.704	0.989	0.989	0.999	0.957	0.512	0.879	1.0
	16	0.740	1.000	0.999	0.814	0.975	0.977	0.999	0.925	0.486	0.913	1.0
8M	1	0.650	1.000	1.000	0.735	0.996	0.996	0.999	0.998	0.436	0.000	0.0
	2	0.675	1.000	1.000	0.684	0.992	0.992	0.999	0.996	0.478	0.836	1.0
	4	0.724	1.000	1.000	0.689	0.990	0.991	0.999	0.993	0.473	0.895	1.0
	8	0.804	1.000	1.000	0.698	0.989	0.990	0.999	0.988	0.466	0.913	1.0
	16	0.804	1.000	0.999	0.764	0.986	0.987	0.999	0.978	0.454	0.936	1.0

Table A.14: Hit ratios for `chol()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.607	0.000	0.144	0.856
	2	13.4	13.6	10.3	10.6	3.0	3.1	46.2	0.842	0.014	0.335	0.652
	4	76.0	82.4	28.5	51.4	47.4	68.1	59.0	0.537	0.050	0.427	0.523
	8	108.5	111.4	19.8	28.3	88.7	93.5	86.0	0.307	0.053	0.464	0.483
	16	195.7	200.7	17.8	23.4	177.9	188.3	152.0	0.187	0.087	0.244	0.669
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.571	0.000	0.131	0.869
	2	13.1	13.2	10.1	10.2	3.0	3.0	46.5	0.826	0.014	0.332	0.655
	4	79.6	89.0	30.0	58.0	49.6	78.8	58.8	0.651	0.052	0.461	0.487
	8	88.4	89.7	16.1	19.0	72.2	75.6	83.9	0.299	0.044	0.453	0.503
	16	178.9	182.0	16.2	24.0	162.7	167.3	150.5	0.176	0.081	0.232	0.688
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.541	0.000	0.120	0.881
	2	8.1	8.2	5.7	5.9	2.3	2.3	36.5	0.801	0.008	0.322	0.670
	4	29.6	29.9	9.8	10.1	19.8	20.3	45.8	0.437	0.020	0.414	0.566
	8	71.8	72.3	11.6	13.1	60.2	61.6	71.5	0.245	0.040	0.446	0.513
	16	144.2	146.7	11.5	16.3	132.7	136.6	130.1	0.158	0.072	0.360	0.568
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.527	0.000	0.113	0.887
	2	9.7	9.7	7.4	7.5	2.3	2.3	44.1	0.798	0.010	0.320	0.670
	4	34.5	35.1	12.7	14.2	21.8	23.0	55.7	0.437	0.023	0.422	0.555
	8	88.4	88.9	15.9	17.4	72.5	74.6	83.1	0.246	0.044	0.439	0.517
	16	172.8	173.9	15.4	24.3	157.4	162.6	148.3	0.160	0.078	0.236	0.686
8M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.517	0.000	0.109	0.891
	2	7.2	7.2	5.1	5.1	2.1	2.1	35.4	0.793	0.007	0.320	0.671
	4	26.7	27.4	8.8	9.2	17.9	18.5	45.3	0.433	0.019	0.419	0.562
	8	66.9	67.1	10.7	12.4	56.1	57.0	71.3	0.240	0.037	0.444	0.519
	16	134.6	136.0	10.6	15.4	124.0	127.1	129.2	0.154	0.068	0.353	0.579

Table A.15: Traffic and timing for `chol()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.651	0.998	0.999	0.732	0.926	0.962	0.998	0.450	0.660	0.000	0.0
	2	0.679	0.997	0.998	0.701	0.968	0.984	0.998	0.441	0.638	0.253	1.0
	4	0.767	0.995	0.997	0.739	0.945	0.972	0.995	0.167	0.561	0.185	1.0
	8	0.887	0.996	0.998	0.815	0.959	0.978	0.997	0.204	0.555	0.327	1.0
	16	0.860	0.977	0.988	0.904	0.936	0.965	0.998	0.184	0.361	0.523	1.0
256K	1	0.651	0.999	1.000	0.736	0.977	0.983	0.999	0.912	0.798	0.000	0.0
	2	0.677	0.999	1.000	0.687	0.987	0.989	0.999	0.913	0.668	0.552	1.0
	4	0.728	1.000	1.000	0.695	0.987	0.989	0.999	0.915	0.630	0.686	1.0
	8	0.825	0.999	1.000	0.712	0.987	0.989	0.999	0.921	0.561	0.817	1.0
	16	0.754	0.999	0.999	0.820	0.974	0.977	0.999	0.894	0.509	0.881	1.0

Table A.16: Hit ratios for `chol()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.941	0.000	0.180	0.820
	2	23.6	25.8	18.2	20.5	5.4	5.4	45.6	0.512	0.024	0.376	0.599
	4	113.8	123.4	43.0	73.1	70.8	102.0	59.8	0.367	0.076	0.447	0.478
	8	144.3	146.1	26.8	37.3	117.5	125.4	89.2	0.226	0.075	0.490	0.435
	16	226.6	233.9	20.6	33.2	206.1	219.3	161.2	0.166	0.107	0.208	0.686
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.875	0.000	0.140	0.860
	2	18.1	18.5	13.9	14.3	4.2	4.2	45.3	0.466	0.019	0.357	0.624
	4	63.6	64.1	23.4	24.9	40.2	41.4	56.3	0.259	0.042	0.460	0.497
	8	151.2	151.8	27.3	30.6	123.9	128.4	86.4	0.150	0.079	0.475	0.446
	16	284.6	286.7	25.5	38.4	259.2	266.0	161.1	0.100	0.139	0.248	0.614

Table A.17: Traffic and timing for `chol()`, 200Mhz CPU clock.

A.1.5 mp3d()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	3301	2116	10031	2123	32770
	2	4456	1577	7427	1591	28256
	4	3464	1198	5580	1201	21429
	8	3432	898	4166	900	17675
	16	4720	667	3093	672	17465
128K	1	3301	2116	10031	2123	32770
	2	4016	1578	7426	1590	27368
	4	3461	1196	5579	1201	21423
	8	3768	901	4179	903	18382
	16	5030	665	3083	669	18057
256K	1	3301	2116	10031	2123	32770
	2	3922	1577	7426	1590	27183
	4	3318	1193	5566	1199	21103
	8	4102	895	4151	897	18975
	16	5341	662	3071	667	18647
512K	1	3301	2116	10031	2123	32770
	2	3661	1576	7422	1590	26653
	4	3246	1195	5570	1199	20971
	8	4120	893	4145	896	19000
	16	4972	664	3079	668	17932
8M	1	3301	2116	10031	2123	32770
	2	3836	1578	7427	1590	27014
	4	3446	1193	5563	1198	21351
	8	4281	895	4152	897	19339
	16	5321	662	3071	667	18607

Table A.18: Per node reference counts for `mp3d()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.814	0.991	0.996	0.331	0.934	0.933	0.998	0.439	0.833	0.000	0.0
	2	0.821	0.992	0.997	0.630	0.872	0.868	0.998	0.404	0.607	0.653	1.0
	4	0.825	0.988	0.994	0.639	0.839	0.831	0.998	0.358	0.511	0.850	1.0
	8	0.827	0.990	0.995	0.726	0.809	0.797	0.998	0.421	0.463	0.927	1.1
	16	0.827	0.986	0.994	0.850	0.766	0.748	0.999	0.439	0.411	0.959	1.2
128K	1	0.814	0.996	0.998	0.332	0.970	0.970	0.999	0.596	0.804	0.000	0.0
	2	0.821	0.995	0.998	0.590	0.876	0.871	0.999	0.616	0.560	0.815	1.0
	4	0.825	0.992	0.996	0.639	0.841	0.832	0.998	0.446	0.500	0.899	1.0
	8	0.827	0.994	0.997	0.749	0.805	0.790	0.999	0.576	0.453	0.962	1.1
	16	0.828	0.992	0.997	0.860	0.763	0.741	0.999	0.705	0.411	0.980	1.2
256K	1	0.814	0.996	0.999	0.332	0.974	0.974	0.999	0.596	0.788	0.000	0.0
	2	0.821	0.996	0.998	0.580	0.888	0.883	0.999	0.614	0.528	0.874	1.0
	4	0.825	0.996	0.998	0.624	0.837	0.825	0.999	0.600	0.485	0.961	1.0
	8	0.827	0.995	0.998	0.771	0.804	0.787	0.999	0.810	0.453	0.979	1.1
	16	0.828	0.995	0.998	0.868	0.761	0.737	0.999	0.847	0.408	0.991	1.2
512K	1	0.814	0.999	0.999	0.332	0.994	0.994	0.999	0.646	0.520	0.000	0.0
	2	0.821	0.999	0.999	0.550	0.886	0.880	0.999	0.691	0.492	0.970	1.0
	4	0.825	0.998	0.999	0.615	0.838	0.825	0.999	0.699	0.480	0.984	1.0
	8	0.827	0.998	0.999	0.772	0.803	0.785	0.999	0.978	0.450	0.991	1.1
	16	0.828	0.997	0.999	0.858	0.761	0.738	0.999	0.973	0.411	0.996	1.2
8M	1	0.814	1.000	1.000	0.332	0.999	0.999	0.999	0.992	0.433	0.000	0.0
	2	0.821	1.000	1.000	0.571	0.894	0.887	0.999	0.991	0.495	0.994	1.0
	4	0.825	1.000	1.000	0.638	0.837	0.822	0.999	0.988	0.482	0.998	1.0
	8	0.827	1.000	1.000	0.780	0.804	0.786	0.999	0.985	0.450	0.999	1.1
	16	0.828	1.000	1.000	0.868	0.756	0.731	0.999	0.981	0.410	0.999	1.2

Table A.19: Hit ratios for `mp3d()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.738	0.000	0.331	0.669
	2	60.7	61.5	43.1	43.8	17.7	17.7	36.9	0.750	0.065	0.450	0.485
	4	182.6	186.3	59.5	62.4	123.1	128.9	47.7	0.672	0.131	0.459	0.410
	8	379.9	384.9	61.7	67.4	318.3	326.8	82.3	0.644	0.239	0.445	0.316
	16	622.0	626.7	50.4	53.4	571.5	578.4	172.0	0.764	0.400	0.400	0.200
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.691	0.000	0.294	0.706
	2	67.2	67.6	47.5	48.0	19.6	19.7	36.6	0.737	0.071	0.440	0.489
	4	186.1	189.8	60.5	63.3	125.6	131.4	47.7	0.671	0.134	0.459	0.407
	8	385.0	389.4	62.6	67.6	322.4	331.5	82.7	0.666	0.243	0.453	0.304
	16	624.3	628.9	50.6	54.2	573.7	579.5	172.4	0.781	0.402	0.406	0.191
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.685	0.000	0.289	0.711
	2	64.9	65.0	45.9	46.0	19.0	19.0	36.8	0.718	0.069	0.430	0.502
	4	198.0	201.5	63.8	66.1	134.2	136.6	47.7	0.676	0.142	0.460	0.398
	8	383.0	387.9	62.2	65.2	320.8	325.2	82.7	0.676	0.242	0.462	0.296
	16	621.4	622.6	50.4	54.7	571.0	574.3	172.7	0.795	0.401	0.413	0.186
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.661	0.000	0.266	0.734
	2	70.5	70.5	49.9	49.9	20.7	20.7	36.5	0.717	0.075	0.426	0.499
	4	200.7	203.1	64.6	66.2	136.1	139.9	47.6	0.674	0.144	0.458	0.397
	8	387.5	393.1	62.8	67.8	324.7	329.0	83.0	0.680	0.245	0.463	0.292
	16	631.4	633.0	51.1	54.0	580.3	582.8	172.9	0.786	0.408	0.404	0.188
8M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.653	0.000	0.260	0.740
	2	67.7	67.8	47.9	47.9	19.9	19.9	36.7	0.709	0.072	0.427	0.502
	4	202.9	206.3	65.1	68.8	137.8	140.7	47.7	0.681	0.145	0.465	0.390
	8	384.7	391.2	62.2	66.1	322.5	333.2	82.7	0.684	0.243	0.467	0.289
	16	629.3	632.0	51.0	54.0	578.3	584.0	173.1	0.805	0.407	0.411	0.182

Table A.20: Traffic and timing for `mp3d()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.813	0.990	0.995	0.331	0.932	0.931	0.998	0.374	0.809	0.000	0.0
	2	0.820	0.991	0.996	0.741	0.877	0.873	0.998	0.367	0.616	0.613	1.0
	4	0.824	0.987	0.994	0.758	0.843	0.836	0.998	0.318	0.517	0.822	1.0
	8	0.829	0.993	0.997	0.839	0.805	0.794	0.999	0.389	0.455	0.922	1.1
	16	0.827	0.986	0.994	0.917	0.772	0.755	0.999	0.383	0.413	0.951	1.2
256K	1	0.813	0.997	0.999	0.331	0.988	0.988	0.999	0.630	0.662	0.000	0.0
	2	0.820	0.997	0.999	0.682	0.892	0.887	0.999	0.664	0.519	0.903	1.0
	4	0.825	0.998	0.999	0.770	0.840	0.828	0.999	0.654	0.491	0.958	1.0
	8	0.829	0.998	1.000	0.880	0.800	0.784	0.999	0.867	0.443	0.983	1.1
	16	0.828	0.996	0.998	0.928	0.764	0.743	1.000	0.869	0.414	0.990	1.2

Table A.21: Hit ratios for `mp3d()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.454	0.000	0.401	0.599
	2	101.4	102.4	77.7	78.7	23.6	23.7	45.5	0.525	0.107	0.513	0.380
	4	295.8	298.2	106.3	112.6	189.5	199.3	59.4	0.512	0.206	0.496	0.298
	8	568.0	569.5	102.9	119.9	465.2	476.7	107.0	0.571	0.361	0.447	0.192
	16	786.1	788.4	71.0	87.7	715.1	723.9	226.6	0.747	0.526	0.360	0.114
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.380	0.000	0.309	0.691
	2	114.6	114.8	87.4	87.6	27.2	27.2	44.8	0.492	0.121	0.485	0.393
	4	318.6	320.3	114.0	121.3	204.7	213.6	59.4	0.524	0.224	0.501	0.274
	8	560.7	562.3	101.0	115.7	459.8	468.3	107.2	0.616	0.357	0.471	0.172
	16	783.6	786.8	70.7	86.6	712.9	719.7	227.7	0.789	0.528	0.370	0.102

Table A.22: Traffic and timing for `mp3d()`, 200Mhz CPU clock.

A.1.6 water()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	1150	223	11709	2639	28466
	2	1389	221	11593	2665	28617
	4	1705	236	12384	2849	30772
	8	2635	224	12436	2863	32575
	16	5400	202	12311	2841	37704
128K	1	1150	223	11709	2639	28466
	2	1395	221	11593	2665	28628
	4	1704	236	12384	2849	30769
	8	2892	224	12436	2863	33087
	16	3756	202	12311	2841	34416
256K	1	1177	251	12339	2800	29972
	2	1269	258	12540	2887	30476
	4	1969	265	13141	3027	32974
	8	2594	267	13397	3102	34684
	16	2684	264	13770	3187	35547
512K	1	1150	223	11709	2639	28466
	2	1392	221	11593	2665	28621
	4	1709	236	12384	2849	30779
	8	2603	224	12436	2863	32509
	16	3736	202	12311	2841	34375
8M	1	1177	251	12339	2800	29972
	2	1247	258	12540	2887	30433
	4	2074	265	13141	3027	33183
	8	2653	267	13397	3102	34800
	16	2662	264	13770	3187	35502

Table A.23: Per node reference counts for `water()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.879	0.995	0.999	0.874	0.880	0.964	0.993	0.922	0.478	0.000	0.0
	2	0.897	0.994	0.999	0.896	0.890	0.956	0.992	0.920	0.463	0.116	1.0
	4	0.907	0.993	1.000	0.912	0.820	0.907	0.992	0.896	0.469	0.423	1.0
	8	0.909	0.981	0.993	0.945	0.812	0.905	0.992	0.812	0.253	0.564	1.0
	16	0.910	0.985	0.995	0.975	0.780	0.903	0.993	0.849	0.307	0.616	1.1
128K	1	0.880	0.999	1.000	0.877	0.952	0.986	0.993	0.977	0.617	0.000	0.0
	2	0.897	0.999	1.000	0.898	0.953	0.983	0.993	0.984	0.609	0.338	1.0
	4	0.908	0.999	1.000	0.912	0.860	0.932	0.992	0.963	0.510	0.764	1.0
	8	0.910	0.994	0.997	0.950	0.847	0.922	0.993	0.917	0.320	0.827	1.0
	16	0.911	0.996	0.998	0.964	0.818	0.918	0.993	0.952	0.425	0.848	1.1
256K	1	0.882	1.000	1.000	0.867	0.960	0.988	0.993	0.979	0.630	0.000	0.0
	2	0.899	1.000	1.000	0.873	0.960	0.985	0.992	0.986	0.623	0.358	1.0
	4	0.908	1.000	1.000	0.916	0.855	0.928	0.992	0.968	0.526	0.806	1.0
	8	0.909	1.000	1.000	0.936	0.847	0.922	0.992	0.976	0.508	0.878	1.0
	16	0.910	1.000	1.000	0.938	0.848	0.920	0.992	0.982	0.497	0.918	1.0
512K	1	0.880	1.000	1.000	0.877	0.958	0.987	0.993	0.977	0.623	0.000	0.0
	2	0.897	1.000	1.000	0.898	0.957	0.984	0.993	0.984	0.614	0.359	1.0
	4	0.908	1.000	1.000	0.913	0.864	0.933	0.992	0.966	0.525	0.791	1.0
	8	0.910	1.000	1.000	0.945	0.852	0.924	0.993	0.974	0.506	0.871	1.0
	16	0.911	1.000	1.000	0.964	0.856	0.921	0.994	0.980	0.491	0.917	1.1
8M	1	0.882	1.000	1.000	0.869	0.997	0.997	0.993	0.998	0.323	0.000	0.0
	2	0.899	1.000	1.000	0.872	0.988	0.992	0.992	0.999	0.439	0.778	1.0
	4	0.908	1.000	1.000	0.921	0.893	0.935	0.992	0.999	0.488	0.981	1.0
	8	0.909	1.000	1.000	0.938	0.877	0.926	0.993	0.999	0.485	0.988	1.0
	16	0.910	1.000	1.000	0.938	0.871	0.923	0.993	0.999	0.480	0.992	1.0

Table A.24: Hit ratios for `water()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.468	0.000	0.045	0.956
	2	5.3	5.6	4.1	4.4	1.2	1.2	47.3	0.467	0.005	0.063	0.931
	4	28.2	28.9	10.4	12.7	17.9	19.2	57.1	0.514	0.018	0.094	0.888
	8	61.8	63.2	11.0	16.7	50.8	53.3	81.5	0.565	0.029	0.135	0.836
	16	123.4	125.7	10.9	17.5	112.5	116.5	141.7	0.652	0.050	0.243	0.707
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.458	0.000	0.036	0.964
	2	2.3	2.3	1.8	1.8	0.5	0.5	47.1	0.455	0.003	0.053	0.945
	4	25.4	25.6	9.2	9.8	16.2	17.0	54.5	0.504	0.017	0.088	0.895
	8	57.2	57.6	10.1	13.6	47.1	48.9	80.0	0.554	0.027	0.146	0.826
	16	127.2	127.9	11.2	15.2	116.0	118.5	139.6	0.585	0.052	0.182	0.766
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.483	0.000	0.037	0.963
	2	1.9	1.9	1.3	1.4	0.5	0.5	27.9	0.486	0.002	0.045	0.953
	4	22.4	22.8	7.2	7.4	15.2	15.5	45.0	0.542	0.015	0.103	0.881
	8	52.2	52.9	8.0	9.8	44.3	45.2	69.1	0.578	0.027	0.132	0.841
	16	109.5	111.7	8.1	10.7	101.4	103.9	123.3	0.606	0.050	0.129	0.821
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.458	0.000	0.035	0.965
	2	2.3	2.3	1.7	1.8	0.5	0.5	48.7	0.455	0.003	0.053	0.945
	4	25.2	25.4	9.1	9.6	16.1	16.8	54.7	0.503	0.017	0.088	0.895
	8	58.3	58.6	10.3	13.1	48.1	49.7	79.8	0.536	0.028	0.134	0.838
	16	110.3	111.0	9.6	13.4	100.7	103.2	136.2	0.573	0.045	0.180	0.775
8M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.480	0.000	0.032	0.968
	2	0.9	0.9	0.6	0.7	0.3	0.3	37.2	0.481	0.001	0.038	0.961
	4	20.9	21.3	6.7	6.9	14.2	14.6	44.9	0.541	0.015	0.104	0.881
	8	49.2	49.9	7.5	9.4	41.7	42.1	68.9	0.575	0.026	0.132	0.842
	16	103.7	106.0	7.7	10.5	96.0	98.4	122.6	0.601	0.048	0.126	0.826

Table A.25: Traffic and timing for `water()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.879	0.995	0.999	0.874	0.880	0.964	0.993	0.922	0.478	0.000	0.0
	2	0.897	0.994	0.999	0.896	0.890	0.956	0.992	0.920	0.463	0.116	1.0
	4	0.907	0.993	1.000	0.912	0.820	0.908	0.992	0.896	0.468	0.423	1.0
	8	0.909	0.981	0.993	0.966	0.813	0.905	0.992	0.812	0.252	0.563	1.0
	16	0.910	0.985	0.995	0.984	0.780	0.903	0.994	0.849	0.307	0.617	1.1
256K	1	0.880	1.000	1.000	0.877	0.958	0.987	0.993	0.977	0.622	0.000	0.0
	2	0.897	1.000	1.000	0.898	0.957	0.984	0.993	0.984	0.612	0.361	1.0
	4	0.908	1.000	1.000	0.914	0.863	0.933	0.992	0.966	0.525	0.787	1.0
	8	0.910	1.000	1.000	0.948	0.851	0.924	0.993	0.974	0.506	0.866	1.0
	16	0.911	1.000	1.000	0.969	0.853	0.921	0.994	0.980	0.491	0.913	1.1

Table A.26: Hit ratios for `water()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.262	0.000	0.181	0.820
	2	5.2	5.3	4.1	4.1	1.1	1.1	48.8	0.262	0.006	0.183	0.812
	4	20.9	21.3	7.8	9.0	13.0	14.6	60.0	0.264	0.014	0.184	0.803
	8	66.9	67.4	12.3	14.7	54.6	59.1	89.3	0.271	0.033	0.187	0.781
	16	177.7	178.5	16.0	21.7	161.8	169.7	155.3	0.286	0.076	0.186	0.738
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.259	0.000	0.174	0.826
	2	1.7	1.7	1.3	1.3	0.4	0.4	41.1	0.254	0.002	0.172	0.826
	4	11.4	11.4	4.2	4.7	7.1	8.1	58.4	0.257	0.007	0.174	0.818
	8	36.3	36.8	6.6	7.7	29.7	31.5	87.9	0.259	0.018	0.177	0.806
	16	115.2	116.1	10.1	12.7	105.0	109.8	152.9	0.270	0.049	0.178	0.773

Table A.27: Traffic and timing for `water()`, 200Mhz CPU clock.

A.1.7 ge()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	1724	857	12080	901	33660
	2	1704	848	11935	882	33254
	4	1709	852	11971	878	33351
	8	1700	848	11908	869	33177
	16	1700	848	11906	865	33169
128K	1	1724	857	12080	901	33660
	2	1704	848	11935	882	33254
	4	1709	852	11971	878	33351
	8	1700	848	11908	869	33177
	16	1700	848	11906	865	33169
256K	1	1724	857	12080	901	33660
	2	1704	848	11935	882	33254
	4	1709	852	11971	878	33351
	8	1700	848	11908	869	33177
	16	1700	848	11906	865	33169
512K	1	1724	857	12080	901	33660
	2	1704	848	11935	882	33254
	4	1709	852	11971	878	33351
	8	1700	848	11908	869	33177
	16	1700	848	11906	865	33169
4M	1	1724	857	12080	901	33660
	2	1704	848	11935	882	33254
	4	1709	852	11971	878	33351
	8	1700	848	11908	869	33177
	16	1700	848	11906	865	33169

Table A.28: Per node reference counts for `ge()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.925	0.998	0.998	0.493	0.986	0.990	1.000	0.092	0.593	0.000	0.0
	2	0.924	0.995	0.998	0.482	0.986	0.990	1.000	0.083	0.581	0.032	1.0
	4	0.925	0.996	0.998	0.487	0.985	0.992	1.000	0.062	0.575	0.171	1.0
	8	0.925	0.997	0.998	0.484	0.981	0.990	1.000	0.034	0.580	0.309	1.0
	16	0.925	0.996	0.998	0.483	0.977	0.987	1.000	0.024	0.591	0.418	1.0
128K	1	0.925	1.000	1.000	0.493	0.991	0.993	1.000	0.092	0.539	0.000	0.0
	2	0.924	0.997	0.999	0.482	0.993	0.996	1.000	0.227	0.492	0.078	1.0
	4	0.925	0.998	0.999	0.487	0.989	0.994	1.000	0.093	0.533	0.253	1.0
	8	0.925	0.998	0.999	0.485	0.987	0.993	1.000	0.062	0.538	0.472	1.0
	16	0.925	0.998	0.999	0.483	0.983	0.990	1.000	0.050	0.562	0.588	1.0
256K	1	0.925	1.000	1.000	0.493	0.991	0.993	1.000	0.092	0.538	0.000	0.0
	2	0.924	0.999	1.000	0.482	0.995	0.996	1.000	0.227	0.504	0.087	1.0
	4	0.925	0.999	1.000	0.487	0.992	0.996	1.000	0.121	0.460	0.413	1.0
	8	0.925	0.999	1.000	0.485	0.990	0.995	1.000	0.133	0.454	0.664	1.0
	16	0.925	0.999	1.000	0.484	0.987	0.993	1.000	0.148	0.503	0.767	1.0
512K	1	0.925	1.000	1.000	0.493	0.991	0.993	1.000	0.092	0.538	0.000	0.0
	2	0.924	0.999	1.000	0.482	0.995	0.996	1.000	0.227	0.501	0.089	1.0
	4	0.925	1.000	1.000	0.487	0.993	0.997	1.000	0.121	0.455	0.451	1.0
	8	0.925	1.000	1.000	0.485	0.991	0.996	1.000	0.134	0.419	0.729	1.0
	16	0.925	0.999	1.000	0.484	0.988	0.993	1.000	0.202	0.392	0.863	1.0
4M	1	0.925	1.000	1.000	0.495	0.999	0.999	1.000	0.942	0.477	0.000	0.0
	2	0.924	0.999	1.000	0.482	0.997	0.998	1.000	0.949	0.470	0.141	1.0
	4	0.925	1.000	1.000	0.488	0.996	0.998	1.000	0.952	0.369	0.618	1.0
	8	0.925	1.000	1.000	0.486	0.993	0.996	1.000	0.949	0.375	0.799	1.0
	16	0.925	1.000	1.000	0.484	0.989	0.994	1.000	0.940	0.372	0.898	1.0

Table A.29: Hit ratios for `ge()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.483	0.000	0.152	0.848
	2	2.8	2.9	2.2	2.3	0.6	0.6	48.8	0.481	0.003	0.153	0.843
	4	11.3	11.5	4.3	4.9	7.1	7.9	61.2	0.484	0.008	0.155	0.838
	8	36.8	37.2	6.7	8.0	30.1	32.4	87.4	0.489	0.018	0.158	0.825
	16	100.5	100.8	9.0	12.2	91.5	95.8	147.5	0.504	0.041	0.159	0.800
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.480	0.000	0.148	0.852
	2	1.7	1.8	1.3	1.4	0.4	0.4	39.2	0.475	0.001	0.149	0.849
	4	8.2	8.4	3.1	3.4	5.1	5.3	57.3	0.480	0.005	0.151	0.843
	8	26.5	26.8	4.8	6.5	21.7	23.0	86.3	0.482	0.013	0.153	0.834
	16	77.7	78.1	6.9	9.3	70.8	73.9	146.4	0.494	0.032	0.155	0.813
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.480	0.000	0.148	0.852
	2	0.7	0.7	0.5	0.5	0.2	0.2	50.5	0.473	0.001	0.148	0.851
	4	5.0	5.1	1.7	1.9	3.3	3.6	48.8	0.477	0.003	0.149	0.848
	8	16.2	16.5	2.6	3.0	13.5	14.3	75.1	0.477	0.008	0.151	0.841
	16	51.8	52.4	4.0	5.0	47.8	49.9	130.3	0.487	0.023	0.153	0.823
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.480	0.000	0.148	0.852
	2	0.8	0.9	0.7	0.7	0.2	0.2	53.8	0.473	0.001	0.147	0.852
	4	5.3	5.5	2.0	2.3	3.4	3.6	59.7	0.476	0.004	0.149	0.848
	8	18.8	19.0	3.4	4.1	15.4	16.2	86.6	0.477	0.009	0.150	0.841
	16	56.2	56.9	5.0	6.1	51.3	53.7	147.9	0.486	0.023	0.152	0.825
4M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.475	0.000	0.143	0.857
	2	0.6	0.6	0.4	0.4	0.2	0.2	31.7	0.472	0.001	0.146	0.853
	4	3.0	3.1	1.0	1.3	2.0	2.3	50.1	0.473	0.002	0.146	0.851
	8	12.2	12.6	2.0	2.4	10.2	11.2	74.2	0.474	0.006	0.149	0.845
	16	42.0	43.1	3.3	4.0	38.8	40.9	129.2	0.483	0.019	0.152	0.829

Table A.30: Traffic and timing for `ge()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.925	0.998	0.998	0.493	0.986	0.990	1.000	0.092	0.593	0.000	0.0
	2	0.924	0.995	0.998	0.482	0.986	0.990	1.000	0.083	0.581	0.032	1.0
	4	0.925	0.996	0.998	0.487	0.985	0.992	1.000	0.062	0.575	0.171	1.0
	8	0.925	0.997	0.998	0.484	0.981	0.990	1.000	0.034	0.580	0.309	1.0
	16	0.925	0.996	0.998	0.483	0.977	0.987	1.000	0.024	0.591	0.418	1.0
256K	1	0.925	1.000	1.000	0.493	0.991	0.993	1.000	0.092	0.538	0.000	0.0
	2	0.924	0.999	1.000	0.482	0.995	0.996	1.000	0.227	0.504	0.087	1.0
	4	0.925	0.999	1.000	0.487	0.992	0.996	1.000	0.121	0.460	0.413	1.0
	8	0.925	0.999	1.000	0.485	0.990	0.995	1.000	0.133	0.454	0.664	1.0
	16	0.925	0.999	1.000	0.484	0.987	0.993	1.000	0.148	0.503	0.767	1.0

Table A.31: Hit ratios for `ge()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.313	0.000	0.074	0.926
	2	20.7	24.1	15.3	18.7	5.4	5.4	40.5	0.327	0.022	0.137	0.841
	4	76.9	91.0	26.5	51.4	50.4	63.8	50.7	0.361	0.053	0.184	0.764
	8	168.9	195.7	28.3	65.8	140.6	172.2	78.2	0.476	0.091	0.171	0.738
	16	366.8	406.5	31.9	85.3	334.9	378.8	151.9	0.468	0.195	0.177	0.628
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.293	0.000	0.037	0.963
	2	2.5	2.5	1.9	2.0	0.5	0.5	45.7	0.291	0.003	0.077	0.921
	4	10.2	11.5	3.6	5.3	6.6	7.4	51.9	0.303	0.007	0.107	0.886
	8	32.7	35.6	5.6	9.2	27.1	30.3	75.3	0.391	0.016	0.105	0.878
	16	187.2	195.9	16.0	23.0	171.2	179.2	139.4	0.369	0.087	0.135	0.778

Table A.32: Traffic and timing for `ge()`, 200Mhz CPU clock.

A.1.8 mmult()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	2000	10	12112	2030	33295
	2	2000	8	12091	2024	33240
	4	2010	6	12129	2029	33348
	8	2000	5	12055	2015	33146
	16	2000	4	12046	2012	33122
128K	1	2000	10	12112	2030	33295
	2	2000	8	12091	2024	33240
	4	2010	6	12129	2029	33348
	8	2000	5	12055	2015	33146
	16	2000	4	12046	2012	33122
256K	1	2000	10	12112	2030	33295
	2	2000	8	12091	2024	33240
	4	2010	6	12129	2029	33348
	8	2000	5	12055	2015	33146
	16	2000	4	12046	2012	33122
512K	1	2000	10	12112	2030	33295
	2	2000	8	12091	2024	33240
	4	2010	6	12129	2029	33348
	8	2000	5	12055	2015	33146
	16	2000	4	12046	2012	33122
4M	1	2000	10	12112	2030	33295
	2	2000	8	12091	2024	33240
	4	2010	6	12129	2029	33348
	8	2000	5	12055	2015	33146
	16	2000	4	12046	2012	33122

Table A.33: Per node reference counts for `mmult()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.783	0.993	0.999	0.845	0.864	0.915	1.000	0.000	0.556	0.000	0.0
	2	0.821	0.991	0.998	0.685	0.886	0.909	1.000	0.000	0.677	0.000	0.0
	4	0.823	0.989	0.998	0.555	0.883	0.909	1.000	0.000	0.744	0.000	0.0
	8	0.809	0.914	0.955	0.461	0.884	0.911	1.000	0.000	0.294	0.000	0.0
	16	0.807	0.974	0.990	0.449	0.883	0.907	1.000	0.000	0.593	0.000	0.0
128K	1	0.783	1.000	1.000	0.849	0.955	0.919	1.000	0.000	0.887	0.000	0.0
	2	0.821	0.998	1.000	0.687	0.932	0.914	1.000	0.000	0.878	0.000	0.0
	4	0.823	0.998	0.999	0.556	0.925	0.914	1.000	0.000	0.879	0.000	0.0
	8	0.809	0.922	0.957	0.462	0.919	0.919	1.000	0.000	0.241	0.000	0.0
	16	0.807	0.982	0.991	0.450	0.917	0.918	1.000	0.000	0.576	0.000	0.0
256K	1	0.783	1.000	1.000	0.849	0.985	0.929	1.000	0.000	0.633	0.000	0.0
	2	0.821	0.999	1.000	0.687	0.986	0.926	1.000	0.000	0.616	0.000	0.0
	4	0.823	0.998	1.000	0.556	0.986	0.922	1.000	0.000	0.624	0.000	0.0
	8	0.809	0.924	0.957	0.462	0.982	0.925	1.000	0.000	0.062	0.000	0.0
	16	0.807	0.984	0.992	0.450	0.960	0.928	1.000	0.000	0.407	0.000	0.0
512K	1	0.783	1.000	1.000	0.849	0.985	0.929	1.000	0.000	0.633	0.000	0.0
	2	0.821	1.000	1.000	0.687	0.989	0.926	1.000	0.000	0.660	0.000	0.0
	4	0.823	1.000	1.000	0.556	0.990	0.929	1.000	0.000	0.649	0.000	0.0
	8	0.809	0.926	0.958	0.462	0.990	0.931	1.000	0.000	0.028	0.000	0.0
	16	0.807	0.986	0.992	0.451	0.989	0.933	1.000	0.000	0.123	0.000	0.0
4M	1	0.783	1.000	1.000	0.849	0.995	0.929	1.000	0.000	0.000	0.000	0.0
	2	0.821	1.000	1.000	0.687	0.997	0.930	1.000	0.000	0.000	0.000	0.0
	4	0.823	1.000	1.000	0.557	0.998	0.932	1.000	0.000	0.000	0.000	0.0
	8	0.809	1.000	1.000	0.462	0.997	0.933	1.000	0.000	0.036	0.000	0.0
	16	0.807	1.000	1.000	0.451	0.996	0.934	1.000	0.000	0.015	0.000	0.0

Table A.34: Hit ratios for `mmult()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.554	0.000	0.051	0.950
	2	11.9	14.0	8.7	10.8	3.1	3.1	40.1	0.567	0.013	0.099	0.889
	4	47.1	51.8	16.7	21.0	30.4	34.2	50.4	0.608	0.033	0.137	0.831
	8	115.8	128.7	19.6	34.3	96.2	111.4	76.1	0.731	0.061	0.141	0.798
	16	243.5	267.1	20.8	48.8	222.7	249.1	141.8	0.714	0.119	0.146	0.735
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.537	0.000	0.034	0.965
	2	9.5	11.0	7.0	8.5	2.5	2.5	38.2	0.548	0.010	0.083	0.908
	4	35.7	40.0	12.5	17.6	23.2	27.3	50.6	0.581	0.024	0.117	0.859
	8	97.7	106.2	16.3	28.4	81.4	92.9	76.7	0.699	0.050	0.124	0.826
	16	204.2	219.2	18.0	28.9	186.2	203.6	139.4	0.674	0.099	0.131	0.771
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.534	0.000	0.029	0.971
	2	1.4	1.4	1.1	1.1	0.3	0.3	33.0	0.530	0.001	0.060	0.938
	4	6.0	6.5	2.1	3.0	3.8	4.2	51.8	0.547	0.004	0.085	0.911
	8	19.9	21.5	3.5	5.6	16.4	18.2	73.8	0.640	0.010	0.091	0.899
	16	117.6	120.2	10.0	12.6	107.6	110.3	132.3	0.618	0.051	0.111	0.838
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.534	0.000	0.029	0.971
	2	1.1	1.2	0.9	0.9	0.2	0.2	42.2	0.529	0.001	0.060	0.939
	4	4.2	4.4	1.5	1.9	2.6	3.3	50.2	0.544	0.003	0.082	0.915
	8	10.1	10.9	1.7	3.2	8.4	8.9	77.1	0.631	0.005	0.085	0.910
	16	25.3	27.2	2.2	5.7	23.2	25.1	132.2	0.578	0.011	0.096	0.893
4M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.533	0.000	0.027	0.973
	2	0.2	0.2	0.1	0.2	0.0	0.0	0.0	0.526	0.000	0.056	0.944
	4	0.8	0.9	0.3	0.4	0.5	0.6	77.1	0.539	0.001	0.077	0.922
	8	3.3	3.5	0.6	0.7	2.7	2.9	90.8	0.553	0.002	0.092	0.907
	16	10.3	10.6	0.9	1.2	9.3	9.8	145.9	0.557	0.004	0.093	0.902

Table A.35: Traffic and timing for `mmult()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.783	0.993	0.999	0.845	0.864	0.915	1.000	0.000	0.556	0.000	0.0
	2	0.821	0.991	0.998	0.685	0.886	0.909	1.000	0.000	0.677	0.000	0.0
	4	0.823	0.989	0.998	0.555	0.883	0.909	1.000	0.000	0.744	0.000	0.0
	8	0.809	0.914	0.955	0.461	0.884	0.911	1.000	0.000	0.294	0.000	0.0
	16	0.807	0.974	0.990	0.449	0.883	0.907	1.000	0.000	0.593	0.000	0.0
256K	1	0.783	1.000	1.000	0.849	0.985	0.929	1.000	0.000	0.633	0.000	0.0
	2	0.821	0.999	1.000	0.687	0.986	0.926	1.000	0.000	0.616	0.000	0.0
	4	0.823	0.998	1.000	0.556	0.986	0.922	1.000	0.000	0.624	0.000	0.0
	8	0.809	0.924	0.957	0.462	0.982	0.925	1.000	0.000	0.062	0.000	0.0
	16	0.807	0.984	0.992	0.450	0.960	0.928	1.000	0.000	0.407	0.000	0.0

Table A.36: Hit ratios for `mmult()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.313	0.000	0.074	0.926
	2	20.7	24.1	15.3	18.7	5.4	5.4	40.5	0.327	0.022	0.137	0.841
	4	76.9	91.0	26.5	51.4	50.4	63.8	50.7	0.361	0.053	0.184	0.764
	8	168.9	195.7	28.3	65.8	140.6	172.2	78.2	0.476	0.091	0.171	0.738
	16	366.8	406.5	31.9	85.3	334.9	378.8	151.9	0.468	0.195	0.177	0.628
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.293	0.000	0.037	0.963
	2	2.5	2.5	1.9	2.0	0.5	0.5	45.7	0.291	0.003	0.077	0.921
	4	10.2	11.5	3.6	5.3	6.6	7.4	51.9	0.303	0.007	0.107	0.886
	8	32.7	35.6	5.6	9.2	27.1	30.3	75.3	0.391	0.016	0.105	0.878
	16	187.2	195.9	16.0	23.0	171.2	179.2	139.4	0.369	0.087	0.135	0.778

Table A.37: Traffic and timing for `mmult()`, 200Mhz CPU clock.

A.1.9 paths()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
64K	1	1045	8	5266	353	15072
	2	1035	6	5206	349	14906
	4	1035	5	5201	348	14894
	8	1036	4	5201	348	14898
	16	1028	3	5154	345	14767
128K	1	1045	8	5266	353	15072
	2	1035	6	5206	349	14906
	4	1035	5	5201	348	14894
	8	1036	4	5199	348	14895
	16	1027	3	5152	345	14761
256K	1	1045	8	5266	353	15072
	2	1035	6	5206	349	14906
	4	1035	5	5199	348	14892
	8	1036	4	5200	348	14896
	16	1027	3	5151	345	14759
512K	1	1045	8	5266	353	15072
	2	1035	6	5206	349	14906
	4	1035	5	5199	348	14892
	8	1036	4	5200	348	14896
	16	1028	3	5153	345	14763
4M	1	1045	8	5266	353	15072
	2	1035	6	5206	349	14906
	4	1035	5	5199	348	14892
	8	1036	4	5200	348	14896
	16	1028	3	5153	345	14763

Table A.38: Per node reference counts for `paths()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	0.984	1.000	0.597	0.969	0.203	1.000	0.000	0.514	0.982	1.0
	4	0.924	0.983	1.000	0.494	0.919	0.091	1.000	0.000	0.612	0.945	2.3
	8	0.923	0.975	0.998	0.412	0.878	0.022	1.000	0.000	0.685	0.971	3.5
	16	0.923	0.976	0.998	0.362	0.803	0.008	1.000	0.000	0.805	0.970	4.1
128K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	1.000	1.000	0.597	0.986	0.167	1.000	0.000	0.476	1.000	1.0
	4	0.924	1.000	1.000	0.494	0.965	0.058	1.000	0.000	0.372	0.994	2.5
	8	0.923	0.990	1.000	0.413	0.961	0.026	1.000	0.000	0.325	0.992	4.3
	16	0.923	0.988	1.000	0.363	0.902	0.006	1.000	0.000	0.662	0.989	6.2
256K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	1.000	1.000	0.597	0.986	0.140	1.000	0.000	0.476	1.000	1.0
	4	0.924	1.000	1.000	0.497	0.979	0.088	1.000	0.000	0.280	1.000	2.4
	8	0.923	0.998	1.000	0.415	0.972	0.023	1.000	0.000	0.202	0.999	4.4
	16	0.923	0.996	1.000	0.364	0.911	0.006	1.000	0.000	0.695	0.993	6.1
512K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	1.000	1.000	0.597	0.986	0.167	1.000	0.000	0.476	1.000	1.0
	4	0.924	1.000	1.000	0.497	0.979	0.088	1.000	0.000	0.280	1.000	2.4
	8	0.923	1.000	1.000	0.415	0.974	0.024	1.000	0.000	0.179	1.000	4.4
	16	0.923	1.000	1.000	0.364	0.972	0.006	1.000	0.000	0.125	1.000	6.3
4M	1	0.925	1.000	1.000	0.701	0.999	0.954	1.000	0.000	0.489	0.000	0.0
	2	0.924	1.000	1.000	0.597	0.986	0.139	1.000	0.000	0.476	1.000	1.0
	4	0.924	1.000	1.000	0.497	0.979	0.087	1.000	0.000	0.279	1.000	2.4
	8	0.923	1.000	1.000	0.415	0.974	0.023	1.000	0.000	0.178	1.000	4.4
	16	0.923	1.000	1.000	0.364	0.972	0.006	1.000	0.000	0.124	1.000	6.3

Table A.39: Hit ratios for `paths()`.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.195	0.000	0.076	0.924
	2	11.6	12.2	8.9	9.5	2.7	2.7	44.5	0.208	0.012	0.124	0.865
	4	76.2	81.7	29.3	37.1	46.9	52.6	59.8	0.237	0.051	0.183	0.766
	8	281.9	295.3	52.5	59.8	229.3	244.5	95.1	0.291	0.151	0.222	0.627
	16	695.3	701.9	63.8	78.0	631.5	639.4	205.6	0.461	0.401	0.208	0.391
128K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.195	0.000	0.076	0.924
	2	9.0	9.1	6.9	6.9	2.1	2.1	42.3	0.203	0.009	0.114	0.877
	4	38.7	41.9	14.7	20.5	24.0	25.1	58.7	0.218	0.026	0.149	0.825
	8	103.1	108.4	19.1	30.1	84.0	88.0	88.3	0.230	0.050	0.167	0.783
	16	484.4	489.8	44.6	61.0	439.8	452.0	184.5	0.320	0.250	0.192	0.558
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.195	0.000	0.076	0.924
	2	7.5	7.6	5.3	5.4	2.2	2.2	36.4	0.203	0.008	0.115	0.878
	4	24.0	24.1	8.5	8.8	15.6	15.8	50.2	0.211	0.017	0.138	0.845
	8	69.6	70.1	12.1	13.5	57.6	60.4	78.1	0.222	0.036	0.159	0.805
	16	395.1	402.2	33.8	42.2	361.3	370.8	156.1	0.299	0.212	0.196	0.592
512K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.195	0.000	0.076	0.924
	2	9.0	9.1	6.9	6.9	2.1	2.1	42.3	0.203	0.009	0.114	0.877
	4	29.3	29.4	11.1	11.5	18.3	19.1	58.7	0.212	0.019	0.138	0.843
	8	81.2	81.6	15.2	17.5	66.0	69.4	88.4	0.222	0.040	0.157	0.804
	16	179.3	180.5	16.8	21.0	162.6	168.3	156.8	0.233	0.077	0.162	0.761
4M	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.195	0.000	0.076	0.924
	2	7.5	7.6	5.3	5.4	2.2	2.2	36.5	0.203	0.008	0.115	0.878
	4	24.0	24.1	8.4	8.8	15.6	15.8	50.3	0.211	0.017	0.138	0.845
	8	67.2	67.7	11.7	13.2	55.5	58.1	77.7	0.221	0.035	0.158	0.807
	16	149.4	151.4	12.9	15.3	136.5	141.0	139.8	0.231	0.069	0.164	0.767

Table A.40: Traffic and timing for `paths()`.

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
64K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	0.984	1.000	0.597	0.970	0.242	1.000	0.000	0.515	0.980	1.0
	4	0.924	0.983	1.000	0.494	0.920	0.081	1.000	0.000	0.616	0.941	2.2
	8	0.923	0.975	0.998	0.412	0.879	0.031	1.000	0.000	0.689	0.965	3.5
	16	0.923	0.976	0.998	0.362	0.805	0.017	1.000	0.000	0.811	0.969	4.1
256K	1	0.925	1.000	1.000	0.701	0.999	0.953	1.000	0.000	0.496	0.000	0.0
	2	0.924	1.000	1.000	0.597	0.987	0.217	1.000	0.000	0.474	0.999	1.0
	4	0.924	1.000	1.000	0.497	0.979	0.093	1.000	0.000	0.280	1.000	2.3
	8	0.923	0.998	1.000	0.415	0.973	0.030	1.000	0.000	0.205	1.000	4.3
	16	0.923	0.996	1.000	0.364	0.910	0.006	1.000	0.000	0.693	0.993	6.1

Table A.41: Hit ratios for `paths()`, 200Mhz CPU clock.

cSz	N	lkav	lkmx	txav	txmx	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
64K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.104	0.000	0.096	0.903
	2	20.1	21.2	15.4	16.6	4.7	4.7	45.9	0.115	0.021	0.162	0.817
	4	131.3	139.4	49.7	57.8	81.6	92.3	60.7	0.142	0.089	0.238	0.673
	8	423.1	443.0	78.4	91.4	344.7	368.3	100.4	0.194	0.237	0.264	0.498
	16	854.8	863.3	78.4	95.8	776.4	787.3	223.7	0.373	0.537	0.208	0.255
256K	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.104	0.000	0.096	0.903
	2	15.6	15.7	11.9	12.0	3.7	3.7	44.8	0.111	0.017	0.145	0.838
	4	52.7	53.0	19.9	21.1	32.8	34.2	59.6	0.117	0.035	0.175	0.789
	8	142.6	143.8	26.9	29.8	115.7	120.1	91.6	0.127	0.072	0.196	0.732
	16	656.6	661.9	60.9	87.6	595.7	610.0	203.8	0.221	0.377	0.205	0.418

Table A.42: Traffic and timing for `paths()`, 200Mhz CPU clock.

A.2 SCI Meshes

A.2.1 chol()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	9113	2007	2814	611	29007
	16	2732	553	1135	160	10603
256K	4	6720	2044	2010	574	20499
	16	2383	558	804	155	8290
512K	4	6620	2042	2064	576	20428
	16	2343	553	766	160	7972

Table A.43: Per node reference counts for `chol()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.791	0.998	0.999	0.773	0.952	0.973	0.996	0.171	0.539	0.239	1.0
	16	0.865	0.996	0.995	0.796	0.967	0.980	0.998	0.312	0.519	0.559	1.0
256K	4	0.726	0.999	1.000	0.694	0.988	0.990	0.999	0.916	0.601	0.721	1.0
	16	0.815	0.998	0.998	0.767	0.984	0.986	0.999	0.894	0.519	0.864	1.0
512K	4	0.732	0.999	1.000	0.690	0.988	0.990	0.999	0.953	0.557	0.814	1.0
	16	0.800	1.000	0.999	0.764	0.985	0.986	0.999	0.925	0.488	0.913	1.0

Table A.44: Hit ratios for `chol()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	13.4	23.1	11.1	20.1	2.3	2.9	63.1	0.628	0.050	0.446	0.504
		Y	23.8	41.4	17.1	30.6	6.7	10.9					
	16	X	31.6	41.9	7.9	10.1	23.7	35.8	112.1	0.189	0.057	0.321	0.622
		Y	33.8	37.8	12.7	16.0	21.1	24.2					
256K	4	X	9.0	9.7	6.3	6.7	2.7	3.1	76.3	0.445	0.032	0.417	0.551
		Y	12.9	13.3	9.9	10.5	3.0	3.1					
	16	X	33.9	41.9	8.7	11.1	25.2	33.5	109.9	0.157	0.061	0.351	0.587
		Y	37.2	40.7	14.0	17.3	23.2	25.3					
512K	4	X	9.1	10.2	6.2	6.7	2.9	3.5	73.1	0.442	0.031	0.414	0.555
		Y	12.7	13.4	9.8	10.5	2.8	2.9					
	16	X	33.4	45.8	8.6	12.6	24.7	36.9	108.8	0.153	0.060	0.361	0.579
		Y	36.3	40.9	13.8	19.8	22.5	24.6					

Table A.45: Traffic and timing for `chol()`.

A.2.2 mp3d()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	8051	2013	2064	371	23556
	16	5740	897	2091	432	17488
	64	6375	367	1707	373	17178
256K	4	7506	2000	2054	369	22426
	16	5827	891	2077	429	17622
	64	6306	367	1707	373	17043
512K	4	7380	1997	2051	369	22163
	16	5509	891	2078	429	16989
	64	6132	367	1707	373	16694

Table A.46: Per node reference counts for `mp3d()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.828	0.994	0.999	0.768	0.880	0.884	0.998	0.354	0.501	0.892	1.0
	16	0.831	0.993	0.998	0.845	0.781	0.772	0.999	0.501	0.398	0.979	1.2
	64	0.830	0.993	0.998	0.936	0.578	0.533	0.999	0.563	0.302	0.994	1.4
256K	4	0.828	0.998	1.000	0.752	0.875	0.877	0.999	0.657	0.475	0.967	1.0
	16	0.831	0.997	0.999	0.848	0.777	0.767	0.999	0.869	0.397	0.992	1.2
	64	0.830	0.995	0.998	0.935	0.576	0.529	0.999	0.749	0.302	0.997	1.4
512K	4	0.828	1.000	1.000	0.748	0.872	0.873	0.999	0.696	0.466	0.987	1.0
	16	0.831	0.998	0.999	0.839	0.772	0.761	0.999	0.973	0.396	0.996	1.2
	64	0.830	0.996	0.998	0.933	0.573	0.527	1.000	0.910	0.302	0.998	1.4

Table A.47: Hit ratios for `mp3d()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	56.8	59.6	36.9	39.3	20.0	22.0	66.7	0.774	0.183	0.540	0.277
		Y	71.4	74.5	57.1	60.2	14.3	14.4					
	16	X	173.5	205.0	45.8	50.1	127.7	162.8	107.4	0.708	0.320	0.492	0.188
		Y	189.6	203.6	73.2	79.9	116.5	130.6					
	64	X	345.2	603.3	40.1	52.4	305.1	566.7	206.9	0.794	0.463	0.432	0.104
		Y	345.1	387.3	67.2	82.8	277.9	312.2					
256K	4	X	62.4	70.7	40.8	44.7	21.5	26.0	68.5	0.780	0.202	0.528	0.270
		Y	79.8	87.4	63.6	69.8	16.2	17.6					
	16	X	175.0	209.7	46.4	49.7	128.6	173.3	107.7	0.716	0.323	0.494	0.182
		Y	192.2	207.2	74.1	78.9	118.1	132.0					
	64	X	348.6	608.2	40.4	54.2	308.1	571.1	207.2	0.796	0.467	0.430	0.103
		Y	346.8	381.5	67.8	85.9	279.0	308.8					
512K	4	X	64.2	72.8	41.9	45.9	22.3	26.8	67.9	0.786	0.207	0.527	0.267
		Y	81.4	89.7	65.1	71.8	16.3	17.9					
	16	X	182.1	220.4	47.6	52.5	134.5	182.7	107.8	0.716	0.332	0.486	0.182
		Y	194.4	211.2	76.1	83.0	118.3	135.2					
	64	X	352.9	517.4	40.9	61.5	311.9	489.5	205.3	0.790	0.469	0.428	0.104
		Y	350.5	395.2	68.6	98.4	281.9	318.7					

Table A.48: Traffic and timing for `mp3d()`.

A.2.3 water()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	9974	2280	4165	805	30871
	16	7941	1274	7997	1770	34156
	64	4736	197	13510	3129	38800
256K	4	9964	2280	4165	805	30851
	16	8000	1274	7997	1770	34275
	64	4829	197	13510	3129	38987
512K	4	9977	2280	4165	805	30877
	16	8021	1274	7997	1770	34317
	64	4967	197	13510	3129	39263

Table A.49: Per node reference counts for `water()` (×1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.913	0.997	1.000	0.905	0.977	0.993	0.992	0.963	0.528	0.742	1.0
	16	0.910	0.998	1.000	0.930	0.948	0.982	0.993	0.952	0.538	0.696	1.1
	64	0.909	0.994	0.998	0.971	0.755	0.907	0.994	0.936	0.417	0.863	1.1
256K	4	0.913	1.000	1.000	0.904	0.978	0.993	0.992	0.966	0.529	0.780	1.0
	16	0.910	0.999	1.000	0.931	0.962	0.986	0.994	0.980	0.485	0.918	1.1
	64	0.909	0.996	0.999	0.972	0.804	0.909	0.994	0.962	0.421	0.901	1.1
512K	4	0.913	1.000	1.000	0.905	0.979	0.993	0.992	0.966	0.524	0.791	1.0
	16	0.910	1.000	1.000	0.931	0.963	0.986	0.994	0.980	0.482	0.925	1.1
	64	0.909	0.998	0.999	0.973	0.845	0.912	0.994	0.977	0.397	0.952	1.1

Table A.50: Hit ratios for `water()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	6.9	7.2	4.5	4.7	2.4	2.5	75.8	0.510	0.023	0.292	0.685
		Y	9.0	9.5	7.2	7.5	1.8	2.1					
	16	X	25.6	33.6	6.2	9.1	19.4	28.3	102.8	0.577	0.042	0.276	0.683
		Y	23.9	29.6	9.7	13.9	14.2	17.3					
	64	X	68.1	109.6	7.4	12.9	60.7	101.5	156.1	0.674	0.067	0.201	0.732
		Y	53.2	61.8	11.5	19.7	41.7	47.8					
256K	4	X	6.9	7.1	4.5	4.7	2.4	2.5	75.0	0.509	0.023	0.291	0.686
		Y	8.9	9.6	7.2	7.5	1.8	2.0					
	16	X	22.4	26.7	5.3	8.1	17.1	22.8	103.0	0.570	0.037	0.275	0.688
		Y	20.8	25.6	8.4	12.6	12.4	15.0					
	64	X	58.0	104.0	6.2	13.0	51.9	97.1	154.8	0.664	0.057	0.204	0.739
		Y	45.7	54.2	9.7	20.1	36.0	44.1					
512K	4	X	6.9	7.1	4.5	4.7	2.4	2.4	75.4	0.509	0.023	0.291	0.686
		Y	8.9	9.5	7.1	7.4	1.7	2.0					
	16	X	22.3	26.4	5.3	8.1	17.0	22.6	102.3	0.570	0.036	0.276	0.688
		Y	20.6	25.5	8.3	12.6	12.2	14.9					
	64	X	50.7	90.7	5.3	11.6	45.4	84.7	155.1	0.658	0.050	0.208	0.742
		Y	40.0	50.3	8.5	18.1	31.6	36.9					

Table A.51: Traffic and timing for `water()`.

A.2.4 ge()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	1709	852	11971	878	33351
	16	1700	848	11906	865	33169
256K	4	1709	852	11971	878	33351
	16	1700	848	11906	865	33169
	64	1698	848	11893	860	33134
512K	4	1709	852	11971	878	33351
	16	1700	848	11906	865	33169

Table A.52: Per node reference counts for `ge()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.925	0.998	0.999	0.487	0.989	0.994	1.000	0.093	0.533	0.253	1.0
	16	0.925	0.998	0.999	0.483	0.983	0.990	1.000	0.050	0.562	0.588	1.0
256K	4	0.925	0.999	1.000	0.487	0.992	0.996	1.000	0.121	0.460	0.413	1.0
	16	0.925	0.999	1.000	0.484	0.987	0.993	1.000	0.148	0.503	0.767	1.0
	64	0.925	0.999	0.999	0.478	0.975	0.987	1.000	0.043	0.583	0.800	1.0
512K	4	0.925	1.000	1.000	0.487	0.993	0.997	1.000	0.121	0.455	0.451	1.0
	16	0.925	0.999	1.000	0.484	0.988	0.993	1.000	0.202	0.392	0.863	1.0

Table A.53: Hit ratios for `ge()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	2.0	2.1	1.3	1.4	0.7	0.8	82.0	0.481	0.007	0.152	0.842
		Y	2.5	2.9	2.0	2.3	0.5	0.7					
	16	X	14.7	17.8	3.4	4.4	11.3	14.9	99.3	0.489	0.022	0.157	0.821
		Y	11.9	14.4	5.4	6.6	6.6	10.2					
256K	4	X	1.6	1.7	1.1	1.2	0.5	0.6	74.9	0.478	0.005	0.149	0.846
		Y	2.0	2.1	1.7	1.8	0.3	0.4					
	16	X	12.1	14.1	2.7	3.7	9.3	11.4	97.3	0.484	0.018	0.154	0.828
		Y	9.4	12.4	4.3	6.0	5.1	9.0					
	64	X	59.7	72.8	5.9	9.9	53.7	66.2	146.8	0.511	0.055	0.161	0.784
		Y	39.2	58.2	9.1	15.2	30.1	50.1					
512K	4	X	1.3	1.5	0.8	1.1	0.5	0.5	76.9	0.477	0.004	0.148	0.847
		Y	1.6	2.1	1.3	1.7	0.3	0.4					
	16	X	10.7	12.5	2.4	3.2	8.3	10.1	96.6	0.482	0.015	0.153	0.832
		Y	8.1	10.7	3.8	5.2	4.3	7.7					

Table A.54: Traffic and timing for `ge()`.

A.2.5 mmult()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	2010	6	12129	2029	33348
	16	2097	4	12628	2109	34723
	64	2000	2	12028	2008	33073
256K	4	2010	6	12129	2029	33348
	16	2097	4	12628	2109	34723
	64	2000	2	12028	2008	33073
512K	4	2010	6	12129	2029	33348
	16	2097	4	12628	2109	34723
	64	2000	2	12028	2008	33073

Table A.55: Per node reference counts for `mmult()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.824	0.995	0.999	0.471	0.967	0.920	1.000	0.000	0.634	0.000	0.0
	16	0.808	0.979	0.991	0.465	0.909	0.922	1.000	0.000	0.576	0.000	0.0
	64	0.821	0.991	0.999	0.458	0.899	0.919	1.000	0.000	0.803	0.000	0.0
256K	4	0.824	0.997	1.000	0.472	0.988	0.927	1.000	0.000	0.463	0.000	0.0
	16	0.808	0.982	0.992	0.465	0.951	0.930	1.000	0.000	0.446	0.000	0.0
	64	0.821	0.996	0.999	0.458	0.920	0.929	1.000	0.000	0.841	0.000	0.0
512K	4	0.824	0.997	1.000	0.472	0.992	0.930	1.000	0.000	0.402	0.000	0.0
	16	0.808	0.985	0.992	0.465	0.989	0.932	1.000	0.000	0.122	0.000	0.0
	64	0.821	0.998	1.000	0.458	0.959	0.932	1.000	0.000	0.726	0.000	0.0

Table A.56: Hit ratios for `mmult()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	4.4	6.7	2.9	4.5	1.6	2.2	77.6	0.575	0.017	0.112	0.871
		Y	6.0	8.5	4.5	7.0	1.4	1.5					
	16	X	54.4	86.4	14.1	21.4	40.3	70.5	111.1	0.701	0.089	0.134	0.777
		Y	52.9	66.5	21.9	34.4	30.9	37.3					
	64	X	150.2	190.9	18.6	26.7	131.6	172.4	175.0	0.709	0.160	0.136	0.705
		Y	135.7	150.1	28.4	42.7	107.3	121.3					
256K	4	X	1.6	2.5	1.1	1.7	0.5	0.8	83.7	0.557	0.007	0.097	0.896
		Y	2.3	3.2	1.7	2.6	0.6	0.6					
	16	X	30.9	42.1	8.3	11.0	22.6	33.3	112.6	0.651	0.053	0.113	0.833
		Y	32.9	37.9	12.9	17.6	20.1	25.1					
	64	X	130.3	166.0	16.3	23.1	114.0	150.8	172.5	0.674	0.136	0.127	0.737
		Y	115.9	130.4	24.6	35.8	91.3	104.0					
512K	4	X	1.0	1.6	0.6	1.1	0.3	0.5	79.3	0.553	0.004	0.094	0.902
		Y	1.4	2.0	1.0	1.7	0.4	0.4					
	16	X	6.4	11.6	1.6	3.5	4.8	9.5	113.2	0.604	0.010	0.094	0.896
		Y	6.3	8.7	2.6	5.4	3.6	5.0					
	64	X	72.4	102.0	8.8	11.9	63.6	94.8	166.2	0.608	0.073	0.112	0.815
		Y	62.5	71.4	13.5	19.4	49.0	55.1					

Table A.57: Traffic and timing for `mmult()`.

A.2.6 paths()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	4	1035	5	5200	348	14894
	16	1027	3	5152	345	14761
	64	1031	1	5163	345	14799
256K	4	1035	5	5199	348	14892
	16	1027	3	5152	345	14761
	64	1031	1	5164	345	14801
200MHz	64	1031	1	5164	345	14801
512K	4	1035	5	5199	348	14892
	16	1027	3	5152	345	14761
	64	1031	1	5164	345	14799

Table A.58: Per node reference counts for `paths()` ($\times$1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	4	0.924	1.000	1.000	0.494	0.965	0.054	1.000	0.000	0.370	0.994	2.5
	16	0.923	0.988	1.000	0.363	0.901	0.004	1.000	0.000	0.657	0.990	6.2
	64	0.923	0.984	0.999	0.339	0.610	0.001	1.000	0.000	0.900	0.984	8.3
256K	4	0.924	1.000	1.000	0.497	0.979	0.098	1.000	0.000	0.278	1.000	2.4
	16	0.923	0.996	1.000	0.364	0.910	0.005	1.000	0.000	0.689	0.995	6.4
	64	0.923	0.994	1.000	0.340	0.647	0.000	1.000	0.000	0.900	0.995	9.3
200M	64	0.923	0.994	1.000	0.340	0.648	0.000	1.000	0.000	0.902	0.993	8.9
512K	4	0.924	1.000	1.000	0.497	0.979	0.098	1.000	0.000	0.278	1.000	2.4
	16	0.923	1.000	1.000	0.364	0.971	0.004	1.000	0.000	0.120	1.000	6.6
	64	0.923	0.996	1.000	0.340	0.964	0.002	1.000	0.000	0.117	0.998	11.1

Table A.59: Hit ratios for `paths()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	4	X	9.7	12.8	6.9	9.1	2.9	3.8	76.5	0.220	0.034	0.148	0.819
		Y	13.9	16.9	11.1	14.0	2.8	2.9					
	16	X	102.0	114.7	27.0	35.4	75.0	89.3	115.2	0.290	0.173	0.212	0.615
		Y	104.1	114.6	42.9	55.6	61.1	67.2					
	64	X	446.5	523.3	53.8	75.4	392.7	468.1	188.3	0.719	0.530	0.220	0.250
		Y	369.9	419.8	79.9	112.0	289.9	343.2					
256K	4	X	7.2	8.0	5.2	5.7	1.9	2.3	76.2	0.213	0.025	0.137	0.838
		Y	10.7	12.0	8.4	9.5	2.3	2.5					
	16	X	97.9	117.5	25.8	31.8	72.0	90.0	115.7	0.283	0.167	0.208	0.625
		Y	100.6	105.7	41.0	49.4	59.5	63.3					
	64	X	434.5	594.2	52.6	100.8	381.9	533.3	187.6	0.670	0.515	0.219	0.266
		Y	362.2	429.8	78.3	150.3	283.9	332.5					
200M	64	X	512.2	687.0	62.5	110.3	449.7	615.4	0.0	0.568	0.629	0.207	0.164
		Y	429.2	483.2	93.0	163.1	336.2	390.2					
512K	4	X	7.2	8.0	5.2	5.7	1.9	2.3	76.2	0.213	0.025	0.137	0.838
		Y	10.7	12.0	8.4	9.5	2.3	2.5					
	16	X	35.9	41.8	9.3	11.1	26.6	32.3	119.4	0.229	0.060	0.165	0.774
		Y	37.8	40.4	16.2	19.0	21.6	26.8					
	64	X	105.1	129.9	11.3	17.6	93.7	118.9	180.8	0.245	0.107	0.167	0.726
		Y	89.9	113.3	20.1	29.1	69.8	93.8					

Table A.60: Traffic and timing for `paths()`.

A.3 SCI Cubes

A.3.1 chol()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	4616	1199	1112	158	14301
256K	8	4017	1208	790	149	11516
512K	8	4089	1198	841	160	11885

Table A.61: Per node reference counts for `chol()` ($\times$1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.866	0.998	0.998	0.738	0.971	0.983	0.997	0.298	0.544	0.476	1.0
256K	8	0.822	0.999	1.000	0.701	0.988	0.989	0.999	0.921	0.552	0.827	1.0
512K	8	0.820	0.999	0.999	0.709	0.988	0.989	0.999	0.956	0.523	0.871	1.0

Table A.62: Hit ratios for `chol()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	12.4	15.0	9.6	11.3	2.8	3.7	90.7	0.293	0.046	0.439	0.515
		Y	13.2	14.1	10.4	11.2	2.7	2.9					
		Z	14.5	16.3	11.6	13.0	3.0	3.3					
256K	8	X	13.7	16.3	10.7	12.4	3.1	4.0	92.9	0.247	0.051	0.450	0.499
		Y	16.0	16.8	12.8	13.4	3.2	3.5					
		Z	15.3	16.4	12.2	13.3	3.1	3.1					
512K	8	X	13.4	15.7	10.4	11.9	3.1	3.9	92.2	0.251	0.050	0.444	0.506
		Y	15.5	16.9	12.4	13.7	3.1	3.4					
		Z	15.2	16.7	12.1	13.4	3.1	3.3					

Table A.63: Traffic and timing for `chol()`.

A.3.2 mp3d()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	3959	885	4110	889	18587
	64	5424	367	1705	373	15273
256K	8	5013	881	4095	885	20655
	64	5597	367	1706	373	15620
512K	8	3950	868	4035	872	18374
	64	5588	367	1706	373	15603

Table A.64: Per node reference counts for `mp3d()` ($\times$1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.827	0.994	0.997	0.764	0.785	0.759	0.999	0.570	0.454	0.969	1.1
	64	0.830	0.993	0.998	0.925	0.574	0.528	0.999	0.561	0.305	0.994	1.4
256K	8	0.827	0.995	0.998	0.814	0.775	0.742	0.999	0.809	0.449	0.983	1.1
	64	0.830	0.995	0.998	0.927	0.575	0.527	0.999	0.753	0.309	0.997	1.4
512K	8	0.827	0.998	0.999	0.766	0.758	0.718	0.999	0.977	0.443	0.994	1.1
	64	0.830	0.997	0.999	0.927	0.574	0.529	1.000	0.915	0.306	0.999	1.4

Table A.65: Hit ratios for `mp3d()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	68.0	76.2	52.0	56.7	16.0	19.7	83.4	0.703	0.256	0.461	0.283
		Y	77.0	89.4	60.3	68.8	16.8	21.2					
		Z	71.1	85.0	55.6	67.7	15.5	17.3					
	64	X	178.6	311.1	74.0	131.2	104.6	209.9	159.5	0.686	0.418	0.461	0.121
		Y	193.6	248.3	90.1	107.4	103.5	148.8					
		Z	186.5	212.4	84.8	102.4	101.7	114.3					
256K	8	X	68.5	84.5	52.3	62.6	16.2	22.1	81.8	0.750	0.251	0.487	0.263
		Y	81.1	95.2	63.5	71.8	17.6	23.5					
		Z	58.6	63.7	44.9	49.6	13.6	14.1					
	64	X	176.6	317.5	73.1	133.6	103.4	215.7	159.5	0.691	0.416	0.465	0.119
		Y	193.7	247.6	90.1	106.4	103.6	147.5					
		Z	184.8	208.6	83.9	99.0	100.9	114.4					
512K	8	X	85.9	122.5	66.1	91.3	19.8	31.5	81.2	0.737	0.270	0.468	0.262
		Y	76.3	83.6	59.5	63.4	16.8	20.2					
		Z	62.9	67.8	48.5	52.5	14.4	15.3					
	64	X	176.6	318.5	73.2	135.0	103.5	214.8	159.7	0.691	0.417	0.465	0.118
		Y	193.6	257.2	90.1	112.1	103.6	154.1					
		Z	185.9	217.5	84.6	104.7	101.4	117.7					

Table A.66: Traffic and timing for `mp3d()`.

A.3.3 water()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	9594	2305	4048	782	29718
	64	4782	197	13510	3129	38892
256K	8	9594	2305	4048	782	29718
	64	4839	197	13510	3129	39006
512K	8	9594	2305	4048	782	29718
	64	4999	197	13510	3129	39326

Table A.67: Per node reference counts for `water()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.918	0.998	1.000	0.899	0.984	0.994	0.992	0.917	0.543	0.097	1.4
	64	0.909	0.994	0.998	0.972	0.755	0.908	0.994	0.936	0.417	0.862	1.1
256K	8	0.918	1.000	1.000	0.900	0.990	0.998	0.992	0.974	0.592	0.283	1.4
	64	0.909	0.996	0.999	0.972	0.804	0.909	0.994	0.962	0.421	0.901	1.1
512K	8	0.918	1.000	1.000	0.900	0.991	0.998	0.992	0.974	0.582	0.297	1.4
	64	0.909	0.998	0.999	0.973	0.845	0.912	0.994	0.977	0.397	0.952	1.1

Table A.68: Hit ratios for `water()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	3.1	4.5	2.5	3.5	0.7	0.9	98.6	0.482	0.011	0.264	0.725
		Y	3.4	5.5	2.8	4.6	0.6	0.9					
		Z	2.9	3.5	2.3	2.8	0.6	0.7					
	64	X	32.0	68.8	12.9	27.6	19.1	47.7	139.2	0.671	0.060	0.204	0.736
		Y	28.1	37.7	12.9	18.7	15.3	23.3					
		Z	22.1	29.4	9.6	15.2	12.5	17.3					
256K	8	X	3.0	4.3	2.4	3.4	0.7	0.9	95.9	0.475	0.010	0.260	0.730
		Y	3.2	5.3	2.7	4.4	0.6	0.9					
		Z	2.7	3.3	2.2	2.7	0.5	0.6					
	64	X	27.3	65.4	10.8	25.7	16.4	45.1	137.4	0.660	0.051	0.206	0.743
		Y	23.7	36.7	10.7	18.0	13.0	22.7					
		Z	18.6	25.7	8.0	16.2	10.6	15.7					
512K	8	X	3.0	4.3	2.3	3.4	0.7	0.9	96.0	0.475	0.010	0.260	0.730
		Y	3.2	5.3	2.7	4.4	0.6	0.9					
		Z	2.6	3.3	2.1	2.7	0.5	0.6					
	64	X	23.5	60.5	9.3	24.6	14.3	43.4	138.2	0.655	0.045	0.210	0.745
		Y	20.8	30.6	9.4	15.7	11.5	19.1					
		Z	16.0	21.0	6.8	10.5	9.2	12.8					

Table A.69: Traffic and timing for `water()`.

A.3.4 ge()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	1700	848	11908	869	33177
256K	8	1700	848	11908	869	33177
	64	1698	848	11893	860	33134
512K	8	1700	848	11908	869	33177

Table A.70: Per node reference counts for `ge()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.925	0.998	0.999	0.485	0.987	0.993	1.000	0.062	0.538	0.472	1.0
256K	8	0.925	0.999	1.000	0.485	0.990	0.995	1.000	0.133	0.454	0.664	1.0
	64	0.925	0.999	0.999	0.478	0.975	0.987	1.000	0.043	0.583	0.800	1.0
512K	8	0.925	1.000	1.000	0.485	0.991	0.996	1.000	0.134	0.419	0.729	1.0

Table A.71: Hit ratios for `ge()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	4.9	7.0	3.8	5.5	1.0	1.5	97.4	0.483	0.014	0.153	0.833
		Y	4.1	6.7	3.4	5.6	0.8	1.1					
		Z	3.3	6.1	2.7	5.1	0.6	0.9					
256K	8	X	3.7	4.7	2.9	3.7	0.8	0.9	90.6	0.478	0.010	0.151	0.839
		Y	3.0	4.8	2.5	4.0	0.5	0.8					
		Z	2.1	3.9	1.7	3.3	0.4	0.6					
	64	X	28.6	38.8	11.1	17.3	17.6	24.7	127.3	0.507	0.048	0.162	0.790
		Y	21.3	35.8	9.4	16.5	11.9	19.3					
		Z	15.4	27.5	6.6	14.0	8.8	17.5					
512K	8	X	3.6	4.3	2.8	3.4	0.8	0.9	88.3	0.477	0.009	0.150	0.841
		Y	2.8	4.3	2.3	3.6	0.5	0.7					
		Z	1.9	3.0	1.5	2.5	0.4	0.5					

Table A.72: Traffic and timing for `ge()`.

A.3.5 mmult()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	2000	5	12055	2015	33146
	64	2000	2	12028	2008	33073
256K	8	2000	5	12055	2015	33146
	64	2000	2	12028	2008	33073
512K	8	2000	5	12055	2015	33146
	64	2000	2	12028	2008	33073

Table A.73: Per node reference counts for `mmult()` ($\times 1000$).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.810	0.921	0.957	0.468	0.939	0.923	1.000	0.000	0.188	0.000	0.0
	64	0.821	0.991	0.999	0.458	0.899	0.919	1.000	0.000	0.803	0.000	0.0
256K	8	0.810	0.923	0.958	0.469	0.972	0.930	1.000	0.000	0.093	0.000	0.0
	64	0.821	0.996	0.999	0.458	0.920	0.929	1.000	0.000	0.841	0.000	0.0
512K	8	0.810	0.923	0.958	0.469	0.991	0.931	1.000	0.000	0.023	0.000	0.0
	64	0.821	0.998	1.000	0.458	0.959	0.932	1.000	0.000	0.726	0.000	0.0

Table A.74: Hit ratios for `mmult()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	13.6	18.9	10.8	15.7	2.8	3.9	96.5	0.687	0.049	0.112	0.839
		Y	15.8	21.4	12.9	17.9	2.8	3.5					
		Z	16.1	21.0	13.2	17.5	2.9	3.4					
	64	X	68.5	92.0	29.2	42.3	39.3	59.7	153.4	0.695	0.143	0.138	0.719
		Y	72.3	96.9	34.6	56.5	37.7	59.2					
		Z	62.7	86.6	28.9	45.0	33.8	45.9					
256K	8	X	6.3	8.6	5.0	7.3	1.3	2.0	101.6	0.652	0.025	0.094	0.881
		Y	8.7	10.0	7.2	8.5	1.5	1.7					
		Z	8.1	9.4	6.7	7.9	1.4	1.6					
	64	X	60.2	90.7	25.8	40.8	34.3	53.0	156.2	0.666	0.125	0.129	0.746
		Y	63.2	87.8	30.4	44.2	32.8	51.3					
		Z	57.6	72.3	26.7	44.7	30.9	39.0					
512K	8	X	1.8	3.5	1.4	2.8	0.4	0.7	99.2	0.631	0.006	0.083	0.910
		Y	1.9	2.6	1.6	2.0	0.4	0.5					
		Z	1.9	2.3	1.5	2.0	0.3	0.3					
	64	X	32.2	48.0	13.8	22.9	18.4	34.3	151.1	0.604	0.066	0.113	0.821
		Y	34.3	52.0	16.4	25.0	17.9	29.7					
		Z	28.8	37.5	13.3	22.2	15.5	20.8					

Table A.75: Traffic and timing for `mmult()`.

A.3.6 paths()

cSz	N	shdRD	shdWR	lclRD	lclWR	i-fetch
128K	8	1036	3	5199	348	14895
	64	1031	1	5163	345	14799
256K	8	1036	3	5199	348	14894
	64	1031	1	5164	345	14800
200M	64	1031	1	5164	345	14800
512K	8	1036	3	5199	348	14894
	64	1031	1	5164	345	14800

Table A.76: Per node reference counts for `paths()` ($\times$1000).

cSz	N	lrpch	lrcch	lwcch	srpch	srcch	swcch	ifpch	ifcch	flush	purge	shl
128K	8	0.923	0.990	1.000	0.413	0.961	0.026	1.000	0.000	0.324	0.992	4.4
	64	0.923	0.984	0.999	0.339	0.609	0.000	1.000	0.000	0.897	0.989	8.9
256K	8	0.923	0.998	1.000	0.415	0.972	0.021	1.000	0.000	0.199	0.999	4.5
	64	0.923	0.994	1.000	0.340	0.647	0.000	1.000	0.000	0.899	0.994	10.4
200M	64	0.923	0.994	1.000	0.340	0.647	0.000	1.000	0.000	0.900	0.994	9.7
512K	8	0.923	1.000	1.000	0.415	0.974	0.019	1.000	0.000	0.176	1.000	4.4
	64	0.923	0.996	1.000	0.340	0.963	0.002	1.000	0.000	0.116	0.998	11.1

Table A.77: Hit ratios for `paths()`.

cSz	N	n	lkav	lkmx	txav	txmv	psav	psmx	rtdly	rtime	ntwF	shdF	lclF
128K	8	X	17.8	25.1	14.2	20.3	3.6	4.7	99.3	0.232	0.056	0.166	0.778
		Y	17.1	21.1	14.1	17.5	3.0	3.6					
		Z	17.7	21.2	14.6	17.9	3.1	3.3					
	64	X	238.1	327.8	101.5	161.3	136.6	198.3	151.8	0.646	0.476	0.246	0.278
		Y	224.2	297.0	107.8	153.9	116.4	161.8					
		Z	186.9	236.3	84.2	119.6	102.7	126.2					
256K	8	X	13.0	14.3	10.4	11.7	2.6	2.8	100.2	0.224	0.046	0.157	0.797
		Y	16.2	18.5	13.4	15.5	2.7	3.0					
		Z	14.9	18.0	12.3	15.0	2.6	3.0					
	64	X	232.8	405.7	99.4	212.3	133.4	252.3	151.9	0.605	0.463	0.243	0.295
		Y	217.0	324.3	104.4	198.4	112.5	174.5					
		Z	181.9	232.5	82.3	127.7	99.6	138.5					
200M	64	X	283.0	465.7	120.8	243.8	162.2	286.3	0.0	0.493	0.571	0.240	0.189
		Y	265.3	370.2	127.4	212.8	137.9	196.9					
		Z	224.0	288.6	101.2	153.2	122.7	170.3					
512K	8	X	12.5	14.4	10.0	11.7	2.5	2.7	99.4	0.223	0.045	0.156	0.799
		Y	15.6	19.3	12.9	16.2	2.6	3.1					
		Z	14.5	17.3	11.9	14.5	2.5	2.8					
	64	X	45.8	60.5	19.3	31.3	26.5	40.5	157.4	0.242	0.096	0.170	0.735
		Y	48.2	60.1	23.5	34.1	24.6	33.3					
		Z	43.9	71.7	20.9	34.7	23.0	38.2					

Table A.78: Traffic and timing for `paths()`.