University of London

Imperial College of Science, Technology and Medicine

Department of Computing

Shared-Memory Multiprocessors with Stable Performance

Sarah Anne Margaret Talbot

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Engineering of the University of London and
the Diploma of Imperial College, March 1999

Abstract

The shared-memory programming model is attractive to programmers of parallel computers because they are not required to control the placement and communication of application data. Unfortunately, access to data is the root cause of performance problems on distributed shared-memory multiprocessors. Severe performance degradation can occur when there are many access requests competing for service at one or a few processing nodes. The request messages must join a queue for service, thereby increasing the delay between the original issue of a remote access request and the receipt of the data.

This thesis presents a new architectural mechanism for remote read requests which reduces excessive queueing. The reduction in queueing comes from using intermediate nodes, called proxies, to service read requests. By combining read requests for data at a proxy, only one request has to be forwarded to the data's home node.

The use of proxies introduces overheads, primarily due to the extra messages which can arise from sending read requests via a proxy, and the local storage space which a proxy uses to retain data to satisfy later requests from its clients. The overheads can cause performance degradation if proxies are used when they are not needed. To avoid such adverse effects, several variants of the proxy technique are investigated: invoking proxies automatically when excessive queueing occurs at run-time (reactive and adaptive proxies), not holding proxy data copies, and using separate storage buffers for proxy data copies.

The different implementations of the proxy technique are evaluated using execution-driven simulations of eight benchmark applications. The results show that substantial performance benefits can be obtained by using proxies for applications which have widely-shared data structures, and using adaptive proxies in conjunction with separate proxy buffers can improve the performance for all the benchmark applications.

Acknowledgements

I would like to thank my supervisor Paul Kelly for many things, most of all for being a reliable source of challenging ideas. My second supervisor Tony Field also deserves thanks for invaluable encouragement and comments on the initial draft of the "work" chapters in this thesis.

The ALITE simulator is the creation of Ashley Saulsbury, and I am very grateful to him for letting me use and modify it.

Wing To has not only been the ideal lodger, but he has also been an unfailing source of enthusiasm (even when he was getting used to having a "real" job) and he provided useful feedback on an earlier draft of this thesis. My sister Kathy and her husband Chris Long have also done sterling work reviewing the draft thesis, and this was done in their precious time-off, *i.e.* when their twin babies (Maggie and Tom) were asleep.

My parents provided both moral support and a generous loan which helped me to go on living in my "Barnes Hut". They deserve further plaudits for surviving the cost and trauma of having at least one child at school or university for thirty-five years.

Ariel Burton and Olav Beckmann endured sharing an office with me, and Ariel deserves extra thanks for doing the Linux configuration work on my "Lisa" computer. I would also like to thank Madhu Bhabuta, Vishaka Nanayakkara, Zully Grant-Duff, and Gwenllian Grafton-Robinson for their moral support, and the "lunch crowd" - including Justin Cormack, Anthony Mayer, Steve Newhouse, and Dave Thornley - for daily entertainment.

Finally, Andrew Bennett deserves a medal for his constructive criticism and remarkable tolerance.

This work was supported by an EPSRC research studentship, and used simulation equipment provided for the CRAMP project (GR/J99117).

to my parents: Bon and Peter Talbot

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The current economics of computing make it viable in cost terms to connect many powerful microprocessors together to solve a problem in parallel [128]. Parallel computing, however, suffers from the problem that the performance of applications is often disappointing or unpredictable. The same application can have dramatically different run times on different parallel architectures, and even the same application with different problem sizes can show seemingly unpredictable variations in performance on the same multiprocessor [24]. This anomalous behaviour has led potential users to be suspicious of parallel computing, despite their need to run ever larger and more sophisticated applications. An additional problem with parallel computing is portability, and the related issue of separating the application from the underlying architecture. For portability, users wish to be able to write applications which are machine independent. Unfortunately, many of the approaches to addressing the performance problems of parallel architectures involve tailoring algorithms or inserting application program directives, both of which compromise program portability [78].

The shared-memory programming model is attractive to users of parallel computers, because it spares them the worry of controlling the placement of application data [89]. This model is inherently portable because the code is not tailored to a particular implementation, but it comes with the cost of the extra access time that is needed to retrieve data which is held at a remote location. Access to data, the placement of which is not under the control of the application programmer, is the root cause of the performance anomalies which occur on these architectures. In distributed shared-memory (DSM) designs, where the shared data is distributed around the processing nodes which make up the system, severe problems can occur when there are many remote access requests competing for service at one or a few processing nodes [22]. The request messages have to join a queue for service, and this increases the time it takes from the original issue of a remote access request to receiving the data, *i.e.* it increases the latency of the response.

1.1 Motivation

The performance that can be achieved by a particular program running on a specific computer is the result of many interrelated factors. The contributing factors include the program code, the algorithm it is based on, the problem instance being solved, the high-level language the program is written in, the level of human effort used to optimise the program's performance, the compiler's ability to optimise the program, the operating system, the computer's architecture, and the hardware characteristics. As a result it is a complicated matter to track down the cause of performance problems, given that there are so many levels at which the problems can occur.

With parallel computers it becomes even harder to identify the cause of performance problems, and it has proved difficult to provide high-level programming and portability of applications. This is in contrast to the uniprocessor market where, despite the wide range of systems that exist, there are some basic requirements that users have come to expect, and which they also want to apply to multiprocessors [78]:

Programming Familiarity: programs can be written in familiar languages or an easy to understand variant.

Performance Stability: the performance of applications must be stable, *i.e.* from one run to the next the performance of the same program should not show a wide variation. Some variation is inevitable, but the variance should be at the same level as is seen on current uniprocessors.

Performance Portability: it must be easy to move code from one system to another, and the performance of these programs must remain stable.

The shared-memory parallel programming model, where the programmer does not have to manage the "low-level" movement of data in the multiprocessor, might seem to support these requirements. However in practice it has proved necessary to write applications which exploit the specific architecture in use [24]. This need for performance tuning can only be avoided if the architecture is changed to reduce the architectural bottlenecks that lead to performance problems. The aim should be to provide the user with stable and portable performance. This will not necessarily be the *best* performance that could be achieved on a particular multiprocessor, and programmers will still be able to fine-tune applications, but such tuning would be done by choice rather than being essential. Once stable and portable performance is achieved, it becomes possible for the shared-memory parallel programming model to achieve the "programmability" requirements outlined above.

1.1.1 Objectives

This thesis is motivated by the need to provide *architectural* mechanisms to reduce the requirement for performance tuning of applications. The specific questions which have to be addressed include:

1. What are the causes of the performance anomalies observed for applications running on distributed shared-memory multiprocessors?
2. Have any of these causes been neglected by the research to date?
3. Can an architectural technique be found to alleviate such a performance problem?
4. Can this technique be designed to minimise its side-effects on existing performance-enhancing techniques such as caching?

1.2 Background

The work reported in this thesis builds on an existing body of knowledge which is introduced briefly here. A more comprehensive discussion of shared-memory parallel architectures, and the performance problems associated with them, is left until Chapter 2.

1.2.1 Uniprocessor Architectures and Caches

There are three basic components in the uniprocessor system model: a central processing unit (CPU) which does the work, a memory which stores instructions and data, and an input/output (I/O) system which moves information into and out of the system [54]. The process of executing a program involves the CPU retrieving instructions from the memory, fetching any data that it requires, performing an operation, and writing the results to the memory. The execution time of a program is critically dependent on the rate at which instructions and data can be fetched from and written to the memory. Unfortunately, while processor speeds have increased at around 50% a year over the last two decades, main memory speeds have grown at a much slower rate. As a result, the ability to execute instructions and process data far outstrips the rate at which main memory can provide them.

To rectify this mismatch, many computers now include caches: small, fast memories which are physically close to the CPU and which provide the instructions and data needed at a rate more in line with the CPU's demands [44]. Caches work by automatically retaining information that the CPU has recently used or generated. Memory references made by programs tend

not to be to randomly distributed addresses, rather the accesses are usually clustered in time and space [118]. The slower main memory is only accessed when the cache does not contain the necessary information or when that memory has to be updated (*i.e.* when a data line is "evicted" from the cache). Caches can dramatically reduce the average time it takes for the CPU to obtain the information it requires.

1.2.2 Shared-Memory Parallel Architectures

In the taxonomy of all possible computer types proposed by Flynn [36], the Multiple Instruction Multiple Data (MIMD) category covers many of the multiprocessors in use today. In MIMD systems, each processor fetches its own instructions and operates on its own data, with the processors often being off-the-shelf microprocessors [54]. For the MIMD architecture, an additional distinction can be made based on the programming model. In the *message-passing* model, the memory space is partitioned between processors. This means that the programmer has to arrange explicitly for data transfer between nodes if the nodes are to work together on a single problem, where each node contains a processor and some memory. In the *shared-memory* model, all processing elements can access the same address space, which may be physically realised as a central memory, or can be distributed among the processors in the same way as message-passing systems. An important difference between the message-passing and shared-memory approaches is the level of effort required from the programmer. Controlling the transfer of data in the message-passing model adds to the programming effort and can be error-prone. In contrast, the shared-memory model relieves the programmer from managing the data, but may reduce the programmer's ability to maximise the performance of an application on a particular multiprocessor.

Shared-memory can be implemented in a structure similar to the architectures supporting message-passing, *i.e.* a number of nodes connected by a network, with each node containing a processor and memory. This distributed shared-memory (DSM) structure can work well when the vast majority of accesses are to the local memory, but repeated access to a remote item will always entail the delay of traversing the interconnection network. Many shared-memory systems use caches, to reduce communication and to provide processors with the information they need as fast as is practicable. It is then possible that more than one processor will have a cache copy of the same data line. In cache-coherent non-uniform memory access (cc-NUMA) systems, the system ensures that if the value of one copy is changed, then all other copies reflect that change. However, keeping multiple cache copies of a data line "coherent" introduces a new source of delay. For example, in Figure 1.1, suppose that Processors 1, 2 and 3 all hold a copy of variable A in their caches. If Processor 1 updates variable A, what

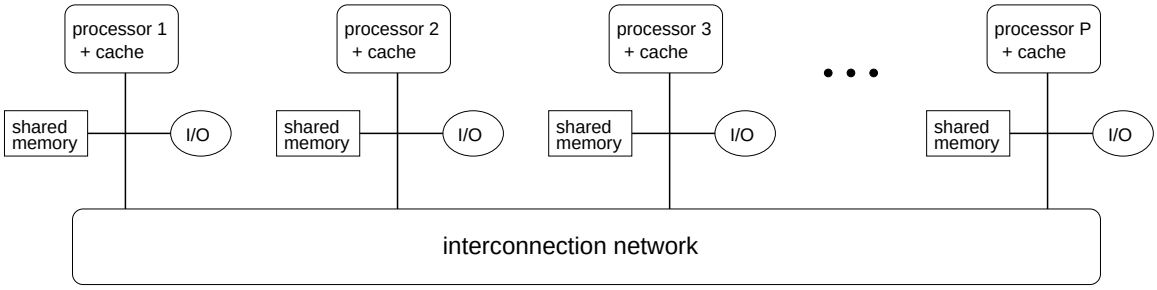


Figure 1.1: A distributed shared-memory architecture

should happen to the other cache copies? Solutions include invalidating or updating all other copies, or hybrid schemes which swap between update and invalidation [89].

In addition to the overheads of maintaining cache coherence, the example architecture shown in Figure 1.1 can suffer from three problems: memory contention, communication contention and latency time [123]. Memory contention occurs because a memory module can only handle one request at a time, so several concurrent requests from different processors will be serialised. Communication contention happens because contention for individual links in the network can occur even if requests are directed to different memory modules. In addition, multiprocessors with a large number of processors tend to have complex interconnection networks, so the latency for such networks (*i.e.* the time a memory request takes to travel across the network) can be very long.

These problems all conspire to increase memory access times, and hence they slow down the overall execution time of tasks running on the processors. The challenge is to minimise the causes of contention, *i.e.* to keep data in the local cache whenever possible and to avoid using the network. Unfortunately, it is often not apparent which parts of a program are causing excessive contention, and forcing the programmer to tune a program to avoid contention on a particular multiprocessor can limit the portability of the application.

1.3 The Approach of the Thesis

This thesis explores the performance anomalies in distributed shared-memory (DSM) systems. The current state of DSM is examined by a wide-ranging survey of the research and commercial literature. The cache coherence policy extensions proposed in this thesis are then evaluated using execution-driven simulations of eight application programs running on a cc-NUMA system. Simulation provides a cost-effective way of evaluating the viability of architectural changes before moving on to the expensive task of implementing a system in

hardware. In addition, simulation allows for monitoring many aspects of system behaviour, such as queue lengths and message traffic, without disturbing the execution of the application programs. Execution-driven simulation is used because it allows the study of the effects on individual program behaviour of changing the memory system (see Appendix B for further information on the simulation of computer systems).

1.4 Statement of Originality

The original contributions of this thesis are:

- An analysis of the causes of queueing for access to widely-shared data. This gives an insight into the effect on such queueing of varying one or more of the architectural parameters of a cc-NUMA system.
- The basic proxy protocol, which uses existing techniques (including the combining of read requests and two-phase random routing) in a novel way to reduce the number of read requests queueing for access to widely-shared data. Read requests for widely-shared data are sent via proxy nodes, where combining is used to reduce the number of read requests sent on to the home node.
- The reactive and adaptive proxy schemes, which invoke the proxy protocol automatically in response to run-time queueing. The reactive scheme only re-routes the message which triggers the proxy processing, whereas the adaptive scheme will proxy all further read requests destined for the congested home node until the adaptive proxying period expires.
- An investigation into two alternatives to holding proxy data copies in the local cache at proxy nodes. These alternatives are not to hold proxy copies at all, or to use a separate buffer for proxy copies. The non-caching approach reduces the storage requirements for proxying, at the cost of a reduction in combining. The separate proxy buffer approach keeps the level of combining high, and avoids cache pollution from proxy copies.

The basic proxy scheme described in Chapter 4 was presented at Euro-Par 96 [13]. Preliminary results for the reactive proxy scheme described in Chapter 5 were presented at Euro-Par 98 [130]. A study of the interaction between different page placement and proxy policies was presented at the Twelfth Annual International Symposium on High Performance Computing Systems and Applications (HPCS'98) [131].

1.5 Overview of the Thesis

In Chapter 2, important background material is introduced describing existing shared-memory implementations, cache coherence protocols, and the performance problems associated with using distributed shared-memory. The solutions which have been proposed to deal with the performance problems are examined, and an example is given of how some of these solutions are used together.

Chapter 3 presents the distributed shared-memory architecture used in this thesis. Simulation results, for eight application programs, are then used to demonstrate the problem of queueing bottlenecks in the cc-NUMA system. Finally, analysis shows how queueing for read access to widely-shared data is affected by varying the architectural parameters.

The access bottleneck that can arise at the home node for widely-shared data is addressed in Chapter 4 with the introduction of basic proxies. The basic proxy protocol distributes read requests for widely-shared data to intermediary nodes known as proxies: if the proxy node has the data it is sent to the requesting client, otherwise the proxy combines read requests for the same data line and sends one read request on to the home node (*i.e.* the node where the data is held in memory). When the data line is sent to the proxy node, the proxy forwards the data on to all the waiting clients.

The basic proxy technique has some shortcomings, the most significant of which are the requirement that the programmer identifies the widely-shared data structures, and the use of local cache to hold copies of proxy data. Chapter 5 investigates two automatic forms of proxying - reactive and adaptive - which are triggered when a queueing bottleneck is detected at run-time. The problem of cache pollution is tackled in Chapter 6 by not caching the proxy data, and in Chapter 7 by using a separate proxy buffer at each node to hold copies of proxy data.

Finally, Chapter 8 summarises the work presented in this thesis, draws conclusions, and suggests the direction of further work.

In addition to the main body of the thesis, the appendices contain further details on uniprocessor caches, the ALITE execution-driven simulator, the implementation of the cache coherence protocol, and the order notation.

Chapter 2

Shared-Memory Multiprocessors

The shared-memory programming model has the advantage of being more straightforward to use when compared to the message-passing approach [89]. In a shared-memory system, there is no need for the application programmer to control the movement of data. Memory is accessible to all processors and communication between the processors is through shared variables. The access to shared-memory data is managed by the hardware and requires no intervention by the operating system. However the scalability of a multiprocessor is affected by how well the system manages the increasing speed mismatch between fast processors and the latency of retrieving remote data. Data has to be fetched and returned to memory, and a processor may have to remain idle during such transfers.

This chapter examines the current state of shared-memory multiprocessor technology, the performance problems associated with these systems, and various solutions which have been proposed to remedy the performance problems. The first section surveys the range of shared-memory multiprocessors, starting with the small-scale symmetric multiprocessors. The use of caching in these multiprocessors introduces the issue of managing multiple copies of the same data item. To scale multiprocessors beyond tens of processors it is necessary to employ more advanced interconnection networks, and to distribute the shared memory. An overview is given of some architectural approaches to distributed shared memory, including cache coherent non-uniform memory access (cc-NUMA), cache-only memory architecture (COMA), and virtual distributed shared-memory. This thesis focuses on cc-NUMA multiprocessors, and in that context further consideration is given to using distributed directories in the cache coherence protocol, the issues raised by keeping data copies consistent, and the use of node controllers to manage all accesses to a node's portion of the distributed shared-memory.

Scalable shared-memory multiprocessors have failed to make much commercial impact, partly because of their cost, but also because of the performance problems that have been observed

on such systems. The second section of this chapter examines the various causes of the performance problems found on cc-NUMA systems. These include system resource bottlenecks (such as network bandwidth and node controller service rate), data locality, the overheads incurred by the cache coherence protocol, and the suitability of the application programs.

The third section of this chapter examines the wide range of solutions which have been proposed to deal with the performance problems. The section gives a flavour of the many varied and overlapping schemes which have been employed in commercial systems or which have been evaluated by research projects. The solutions surveyed include hiding the latency of accessing data, reducing that latency, optimising the cache coherence protocol, managing the placement of data, and improving the design of application programs. The design trade-offs which have to be made in practice between performance enhancing techniques (and their costs) are then illustrated by examining the architecture of the Silicon Graphics Origin2000 multiprocessor [85].

2.1 Implementing Shared-Memory

Figure 2.1 depicts the memory model for shared-memory programmers. The processing for an application is split between a collection of processes running on different processors, all of which can access the shared data. In practice, the shared-memory programming model is supported by a wide range of hardware and software implementations. This section examines that range: from the small-scale symmetric multiprocessors (SMPs) where a number of processors share centralised memory, to large-scale distributed shared-memory (DSM) systems.

2.1.1 Symmetric Multiprocessors

A popular architectural approach for shared-memory systems with a small number of processors is a single shared memory with multiple processors connected by a bus [53]. This approach is illustrated in Figure 2.2. These systems are often referred to as symmetric multiprocessors because they are an example of systems where all the processors have an equal

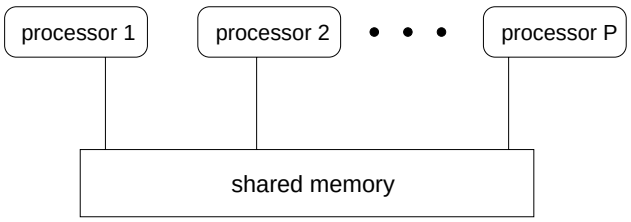


Figure 2.1: The programmer's view of shared-memory multiprocessors

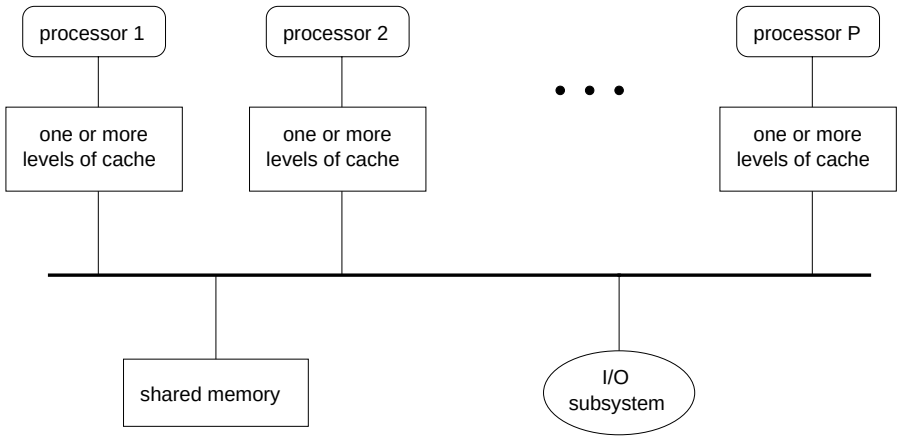


Figure 2.2: A bus-based symmetric multiprocessor (SMP)

relationship with the central shared-memory. In practice the shared-memory is usually split into a number of separate banks, to reduce contention when accessing memory. This uniform memory access (UMA) architecture was typified in the 1980's by commercial systems such as the Sequent Symmetry [93]. The SMP approach is found today in many systems, including the Silicon Graphics Challenge Series [54].

In order to reduce the latency of accessing data items in memory, SMPs usually employ some form of caching. This involves keeping copies of recently-used data in a small but fast storage area associated with each processor. Caching has been used for a long time in uniprocessor systems to exploit data locality (see Appendix A), and is equally useful in shared-memory multiprocessors where access to memory usually entails greater latency than in uniprocessors. Unfortunately the use of caches in multiprocessors introduces a new problem: cache coherence. Copies of the same data item may be held in more than one processor's cache, and if the value of the data item is changed then some action must be taken to ensure that the stale copies are updated or deleted. The sequence of operations in Figure 2.3 illustrates the write-invalidate approach to keeping the data item "A" coherent by removing the out-of-date copies. The example shows how (a) two processors P2 and P3 load in copies of data item A, but when (b) processor P1 stores a new value in data item A all the other copies of A are invalidated, including the value in memory. Processor P1 can go on changing the value of its copy of A, as shown in (c), but the value will not be copied back to the memory until it is evicted from P1's cache or (d) another processor such as P3 loads in data item A.

An alternative to write-invalidate is to update all the copies of a data item when it is changed, *i.e.* write-update. The performance differences between the write-invalidate and write-update protocols arise because [54]:

1. A sequence of writes to the same data line, with no intervening read, requires multiple

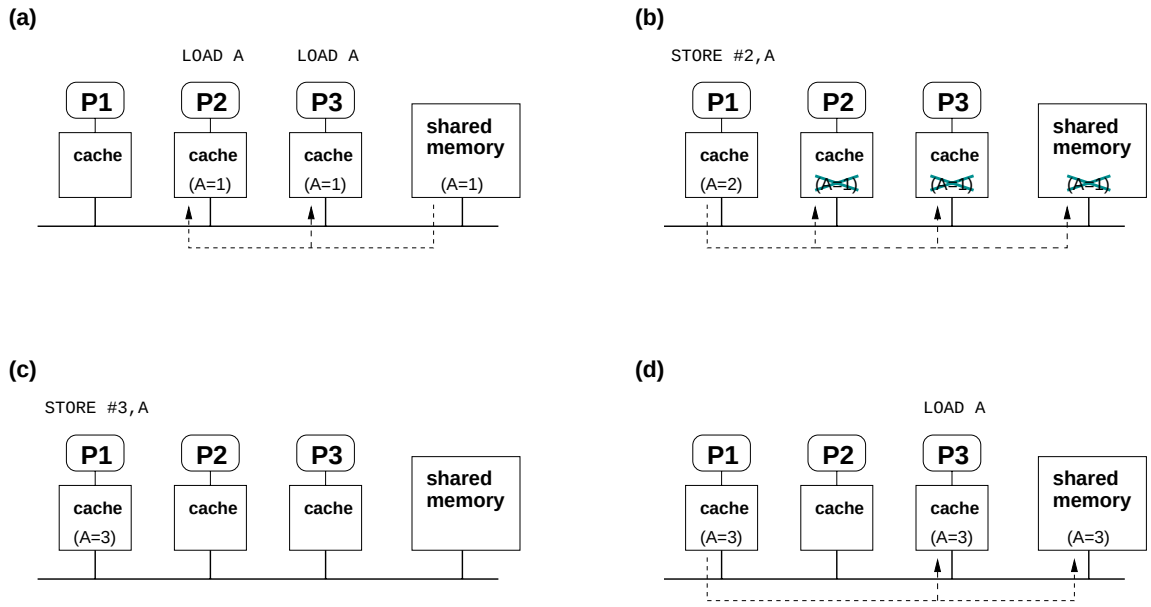


Figure 2.3: An example of invalidation-based multiprocessor cache coherence (from [89])

updates in the update protocol, but only one initial set of invalidations in the invalidate protocol.

2. The delay between writing to a data item in one processor and reading the data item in another processor can be less in the write-update scheme. If the reader already has a copy of the data line, the modified data item is automatically updated in the reader's cache. Under the write-invalidate scheme the reader's copy is invalidated, and the later read will stall until a fresh copy of the data line can be obtained.

The difference in performance between the two approaches therefore depends on both the access patterns of individual applications, and the available interconnection bandwidth. In general write-invalidate has been preferred because it has lower bandwidth requirements, *i.e.* it leads to fewer messages because a processor does not have to keep sending long update messages to other processors when it makes a number of changes to the same data line. There is recent work on adaptive protocols which use both write-update and write-invalidate, *e.g.* by Anderson and Karlin [7], and Raynaud *et al.* [108]. Designing an adaptive protocol which swaps between write-update and write-invalidate at the "correct" time is complicated: the subject is discussed further in Section 2.3.3 of this thesis.

The coherence protocols on bus-based SMPs are implemented by broadcasting transactions on the bus. The cache controllers for each processor snoop on the bus, *i.e.* they "eavesdrop" for any transactions affecting data in their cache [44]. In the example shown in Figure 2.3(b), processor P1 will only have to issue one invalidation message to the memory on the bus;

the cache controllers for processors P2 and P3 will be snooping on the bus and will pick up the invalidation for their copies of data item A. Using a bus interconnection also has the advantage that only one message can be on the bus at a time, so the order in which transactions are observed is the same for all the nodes. The foundations for the current bus-based coherence protocols were set out in the 1980's, and a useful comparison of such protocols is given by Archibald and Baer [9]. The Illinois protocol is an example of a bus-based invalidation protocol [103]. This is often referred to as the MESI protocol, because of its four cache line states: Modified, Exclusive-Clean, Shared, and Invalid. An example of an update-based protocol is the protocol used in the Digital Equipment Corporation's Firefly workstation [133].

2.1.2 Scalable Shared-Memory

The scalability of a parallel system is a measure of its capacity to reduce the execution time of an application as the number of processors working on the task is increased [79]. In practice, the scalability of an architecture is determined by its ability to exploit the inherent parallelism in application programs, by how effective the load balancing is between processors, and how much delay is incurred because of the communication costs for data transfer and protocol handling.

The SMP systems are restricted from scaling beyond a few tens of processors by both the bus bandwidth and the bottleneck which can occur when accessing the memory. Using the fastest available bus technology or splitting the memory into a number of separate banks can extend the number of processors to around 30 to 40, but the cost "per processor" of these systems is significantly higher. An example of a high-end bus-based SMP is the SGI Challenge XL which can support up to 36 processors [54].

Over the years there have been a number of research and commercial projects to implement the shared-memory model on multiprocessors which aim to scale to hundreds of processors and beyond. Such systems, although different in their design details, all start with a more flexible interconnect than a bus. For example, the Sun Microsystems' Enterprise 10000 Server (the Starfire) still uses address buses (four) for snooping, but has a 16×16 crossbar interconnect for data transfer [23]. This results in an SMP which can scale to up to 64 processors, *i.e.* sixteen processing nodes with four processors per node.

2.1.3 Interconnection Networks

The choice of interconnection network is critical to the performance of scalable shared-memory systems. The network has to cope with at least two message lengths (*i.e.* with or without a data line payload), and the message traffic will depend on both the coherency protocol and the individual application programs. This subsection gives an overview of the networks applicable to large-scale systems, and it introduces the design trade-offs that have to be made between cost, latency, and bandwidth.

An interconnection network is characterised by four factors:

Topology: the layout of the interconnection network. Figure 2.4 illustrates some common topologies used for networks. Indirect networks concentrate the switches together, leaving the processing nodes at the edges of the network, *e.g.* the butterfly, binary tree, crossbar, and fat tree networks. Direct networks distribute the routing switches with the processing nodes, *e.g.* the 2D torus, and hypercubes[1]. The indirect approach aims to optimise uniform message traffic, whereas direct networks tend to be better at supporting nearest neighbour communication [89].

Routing Algorithm: the method used to choose the path from one node to another. This can be adaptive to avoid faulty or congested switches, or can use approaches such as two-phase random routing [137]. However using such dynamic routing implies that messages sent between the same two points can arrive out of order. In addition, there is a choice between receiving all of a message at an intermediate network node before forwarding it on to the next node (store-and-forward) or starting to pass on the contents of a message while it is still arriving (cut-through routing schemes) [79].

Flow Control: the method used to regulate traffic in network, *i.e.* arbitrating between messages competing for the same resource. The performance of interconnection networks can be improved by organising the buffers associated with each network channel into several lanes or virtual channels rather than having a single first-in-first-out (FIFO) queue [27]. Using virtual channel flow control allows other messages to pass blocked messages.

Switching Technique: this determines the way in which the data in a message traverses the route. Circuit-switched communication will first establish a connection between two

[1]Most direct network topologies have been built with k-ary n-cubes or are isomorphic to them, where n = dimension, k = radix, $P = k^n$ = number of nodes. A k-ary n-cube has k nodes in each dimension, each node can be labelled by an n digit number of radix (base) k, and each node is connected to every node which has a label which differs in only one digit by one. A k-ary n-cube can be constructed by connecting together k k-ary $(n-1)$-cubes together in a ring. For example rings are k-ary 1-cubes, a 2D torus is a k-ary 2-cube, and hypercubes are 2-ary n-cubes [28].

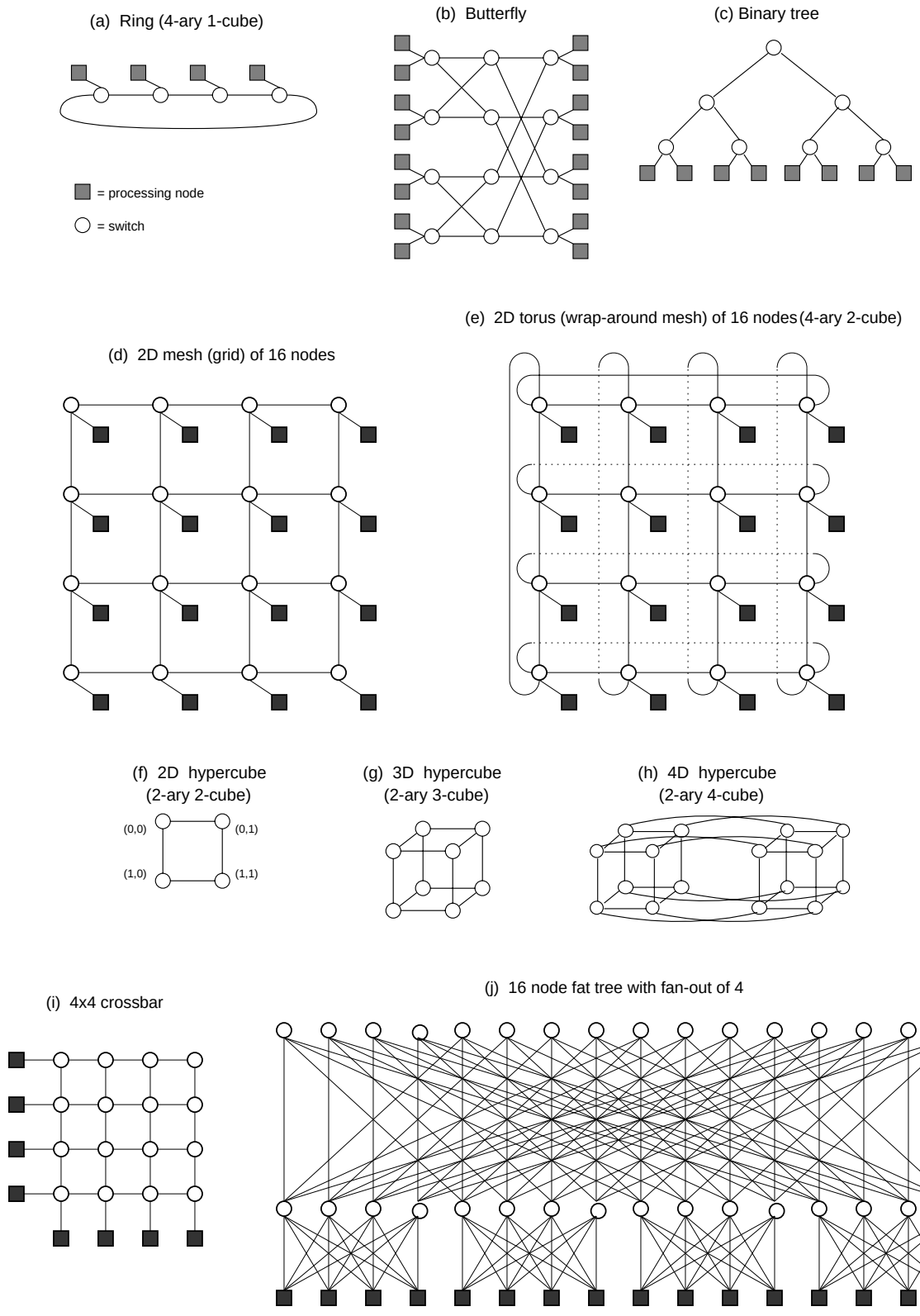


Figure 2.4: Examples of network topologies

nodes, and then use the already established connection to give lower latency for subsequent messages. In contrast, packet-switching establishes a route for each message sent and so it yields a lower startup overhead per communication. Packet-switching is more appropriate for shared-memory systems, where messages are short and communication patterns are very dynamic [54].

An ideal, scalable network would have [89]:

- a low cost which grows linearly with the number of processors. The cost is determined by the number of switches and connections between the switches. In addition, faster switches and/or connections will be more expensive.
- minimal latency that is independent of P (the number of processors). Given the physical limitations on the number of connections coming into and going out of a switch (*i.e.* the fan-in and fan-out limitations), the best that can be achieved is to limit the latency growth to $O(\log P)$ (see Appendix D for the definition of the "order" notation used in this thesis). If the fan-in and fan-out are bounded by some constant, then the latency will grow as a function of $\log P$ where the base of the logarithm is equal to the fan-in and fan-out constraint. The only way to remove the dependence on $\log P$ is to have unbounded fan-in and fan-out, which is not achievable in practice [128].
- bandwidth that grows linearly with P. If communication is uniformly distributed then half of all the message traffic will pass across any partition of the network into two equal parts. The bisection bandwidth is the minimum bandwidth required across any of these partitions [79]. To achieve linear scaling of the bandwidth, the bisection bandwidth must grow linearly with the number of processors. In practice many algorithms result in non-uniform distribution of messages, with nearest neighbour communication often dominating [89]. As a result many networks are designed to have better local communication bandwidth than bisection bandwidth.

The Massachusetts Institute of Technology's M-Machine provides an example of how virtual channels and a throttling mechanism can be used to ensure that transactions complete in the presence of finite network resources [35]. Two network priorities are provided, one each for requests and replies. Messages are routed in dimension order (the network is a 3D mesh) using up to four virtual channels. In order to prevent a processor from injecting messages into the network at a rate that is higher than they can be consumed, the M-Machine implements a return-to-sender throttling protocol. When a message is sent, buffer space is reserved in case the message is returned. If no buffer space is available then no additional messages can be sent: threads attempting to initiate a transaction will stall. When the message reaches its

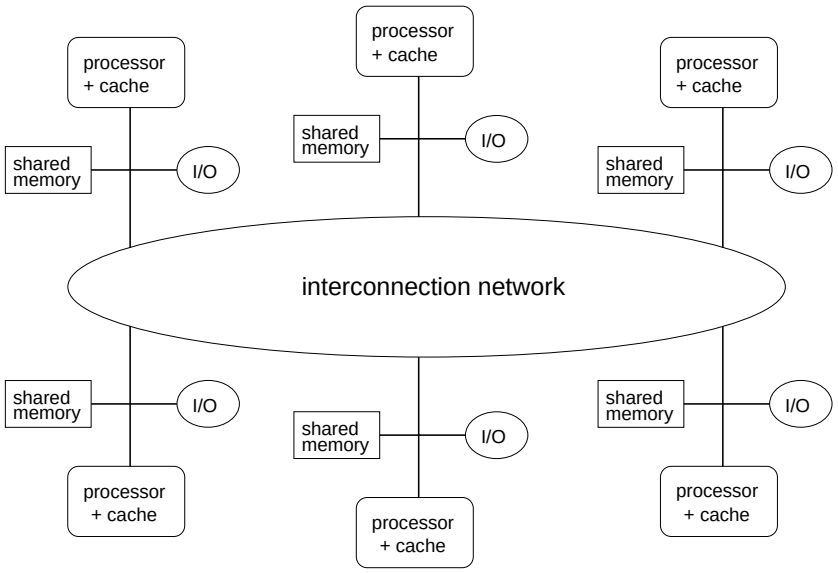


Figure 2.5: Distributed shared-memory

destination, a reply is sent indicating whether the destination was able to handle the message. If the message was consumed, the reply instructs the source processor to release the reserved buffer space. Otherwise, the reply contains the contents of the original message, which are copied into the buffer and sent again later.

2.1.4 Non-Uniform Memory Access

The trend in scalable shared-memory systems has been to distribute the physical memory (dynamic random-access memory - DRAM) among the processing nodes[2] as illustrated in Figure 2.5. This introduces non-uniform memory access (NUMA) because a processor at a given node will be able to access data in its local portion of the shared-memory in less time that it will take to access data held in memory at another node.

Early attempts at producing a scalable shared-memory multiprocessor using the NUMA approach included a number of systems which did not support cache coherence, *e.g.* the Cm* [38], the BBN Butterfly [86], the NYU Ultracomputer [45], and the IBM RP3 [104]. The lack of cache coherence complicated the programming model for these systems. For example, the Cm* was a cluster-based multiprocessor with a distributed memory and non-uniform access time, so the absence of caches and a long remote access latency made data placement critical. The RP3 and the Ultracomputer both included facilities in the network to combine references to the same data line in order to reduce the average latency of remote data access. The BBN

[2]The UMA model is still used in some approaches such as PRAM simulations, *e.g.* the Saarbrücken SB-PRAM [37] and the Tera MTA [5], where many overlapping processing threads are run on each CPU with the aim of masking the latency of accessing memory. Further information is given on these architectures in the discussion of multi-threading in Section 2.3.1 of this thesis.

Butterfly, introduced in 1981, was named for the "butterfly" multi-stage switching network around which it was built. An extra path was provided from the processors to memory by pairing up each processor with one of the memory modules, so each processor had a "favoured" memory unit. The processor could access this memory directly without going through the switch. Up to 256 CPUs, each with local memory, could be connected to allow every CPU access to every other CPUs memory, albeit with a substantially greater latency than for its own, a ratio of roughly 15:1. A current example of a non-cache-coherent NUMA system is the Cray T3E [114].

Given that the non-caching approaches require programmer intervention for accessing remote data, and typically experience long latency for such remote data accesses, a number of caching approaches to distributed shared-memory have been explored in the last two decades. Broadly speaking these fall into two categories: cache-coherent non-uniform memory access (cc-NUMA) and cache only memory architecture (COMA).

2.1.5 Cache-Coherent Non-Uniform Memory Access

In a NUMA architecture, every memory line has a fixed mapping from its address to the memory at one node (its home node) [89]. The data is allocated to memory in pages, each of which contains a number of data lines. For cache-coherent non-uniform memory access (cc-NUMA) systems, data lines retrieved from memory are held in the local cache hierarchy, and a cache coherence policy is used. This approach reduces the latency of accessing remote data lines where temporal and spatial locality apply (as with all caching schemes - see Appendix A). However the cache coherence policy comes with the overheads of keeping track of the cached copies, and the additional messages needed to enforce cache coherence. The cc-NUMA architecture was first explored by research projects such as the Stanford DASH system [88]. In the last four years a number of commercial systems have appeared which use the cc-NUMA approach: these include the SGI Origin2000 [85], the HP/Convex Exemplar [1], the Data General Aviion [29], and the Sequent NUMA-Q [92].

As was seen in Section 2.1.1, the cache coherence protocols used in SMPs rely on one-to-all broadcasting and snooping on the bus. This approach is not scalable because it leads to the network being flooded by protocol messages and/or intolerable message latency. cc-NUMA systems use directory-based protocols to keep track of the location of the cache copies of each data line, so that the update or invalidate messages are only sent to the relevant processors. Directory-based protocols explicitly maintain a list of processors which have a cached copy of each data line, and they identify each line's current owner [91].

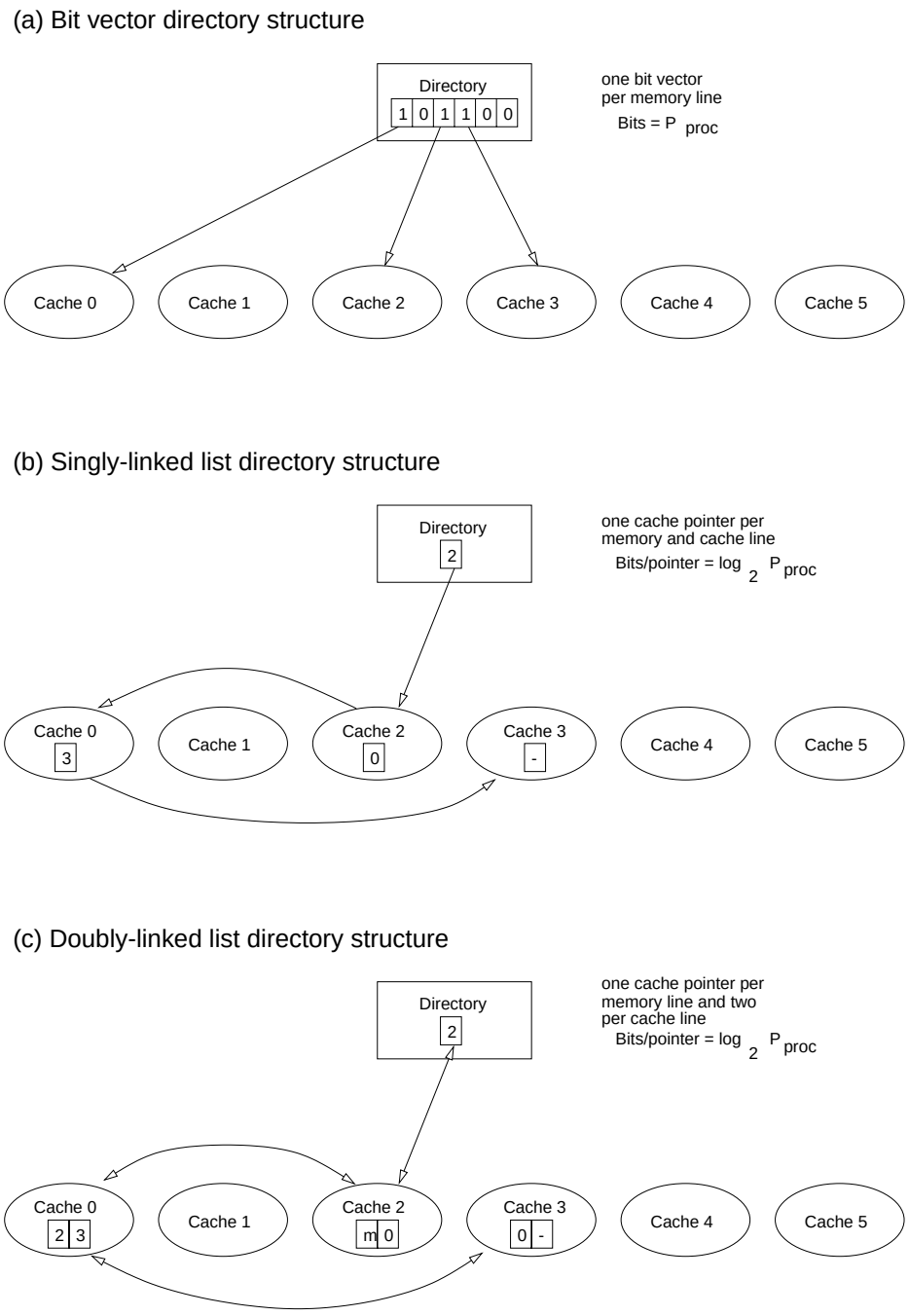


Figure 2.6: Examples of directory structures

It would be impractical to hold a large centralised directory in a scalable multiprocessor, because access to it would be a serious bottleneck. Therefore directory-based protocols are generally implemented by distributing the directory information in association with the data lines in memory. Figure 2.6 shows three different examples of distributed directory structures: bit vector, singly-linked list, and doubly-linked list.

In a full bit vector directory, a centralised version of which was first proposed by Censier and Feautrier [21], a bit vector with one bit per processor is associated with each data line in

memory. Each bit indicates whether that data line is currently cached by the corresponding processor. When data line copies have to be invalidated, messages are sent to all the processors whose bit is set on. This scheme requires the lowest level of coherence messages because the sharers are all known to the home node. The main drawback to the scheme is that the storage required to hold the bit vectors increases with $\Theta(P^2)$ (see Appendix D for the order notation definitions). In addition, if a data line is widely-shared, then invalidations will tie up the home node as it sends invalidation messages to all the sharers. The bit vector approach proved effective for the Stanford DASH prototype, where the modest number of processing nodes (sixteen) meant that the directory size was not a problem [88]. Coarse vectors are a more scalable form of bit vectors in which a bit represents more than one node: this approach is used in the SGI Origin2000 [85].

An alternative to the bit vector approach is to hold the directory information as linked lists. These lists occupy less space than bit vectors, although there is a trade-off when implementing the list between more pointers for faster maintenance of the list (*e.g.* the doubly-linked lists used, for example, in the HP/Convex Exemplar [1]) versus fewer pointers to save on storage space (*e.g.* as for singly-linked lists).

Sun's Scalable Shared-memory MultiProcessor (S3.mp) is a research prototype based on cache-coherent distributed shared-memory which uses singly-linked sharing lists to support a distributed directory [102]. Figure 2.7 shows the generic S3.mp components: each node is essentially equivalent to a workstation with one or more processors, memory, and I/O. The two components specifically designed for the S3.mp are the interconnect controller (IC) and the memory controller (MC). The MC processes requests for data from both the local processors and remote processors, and it is able to handle two simultaneous transactions by having two protocol engines: one for local data (LPE), and one for remote data (RPE). The S3.mp cache coherence protocol has roughly 30 stable or transient cache states and 20 memory states: these states represent branches in the microprogram implementation of the protocol. The number of states is far more than is typically needed for protocols using a bit

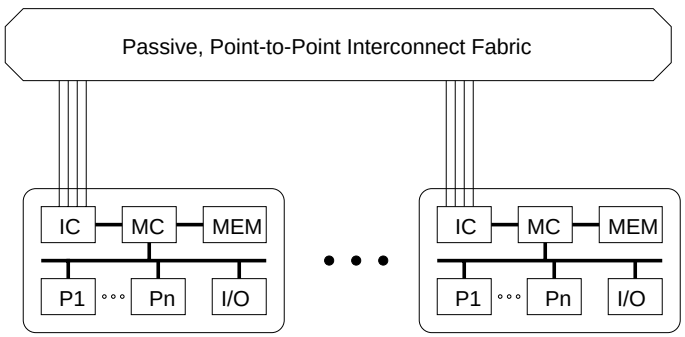


Figure 2.7: Sun's S3.mp prototype system

vector to represent the sharing list. This is because additional states are needed to represent the transient states which occur as the distributed sharing list is being amended [106].

A study by the FLASH team at Stanford, of four different distributed-directory protocols, found that there was no one protocol that would always give the best performance [51]. The bit-vector approach tended to get the best performance up to the point at which it became "coarse" (at 48 processors in their study). For larger-scale systems, the distributed doubly-linked list approach of the Scalable Coherent Interface (SCI) protocol[3] gave it an advantage for the applications which had not been carefully tuned to avoid contention when accessing the directory data at home nodes.

2.1.6 Cache Only Memory Architecture

In contrast to the home page approach of cc-NUMA designs, a cache only memory architecture (COMA) uses the distributed memory as another level of cache. This memory level of cache is often referred to as the attraction memory (AM). In the COMA approach, data lines are replicated and migrated at the memory level as well as in the cache hierarchy. Data lines will move (for write misses) or be copied (for read misses) to the nodes which use them. The ability of data to migrate means that the location of a memory address is decoupled from its physical address, unlike in cc-NUMA where the physical address indicates the home node. A COMA system needs additional hardware to find the current location of a required data line. It is also essential that the AM replacement policy ensures that it does not evict a line when it is the only remaining instance of that data line in memory.

The advantage of using COMA over cc-NUMA is that data is automatically copied or moved to the AM at the node(s) where it is being used. This dynamic re-balancing of the data allows COMA systems to run applications that do not map well to cc-NUMA architectures, *e.g.* applications which have a per-node working set larger than the size of each node's cache, and applications with dynamic data access patterns in which data cannot be effectively statically partitioned across the physical memory. Unfortunately, COMA's flexibility requires non-standard memory system hardware, *i.e.* it is not supported by commodity microprocessors. This introduces a price-performance tradeoff, *i.e.* more sophisticated hardware leads to better performance but at a higher cost and with longer development times [24].

The Data Diffusion Machine (DDM) uses a COMA approach, *i.e.* data migrates around the system according to its use. There are two main "variants" of the DDM: at the Swedish

[3] SCI is an ANSI/IEEE standard which defines a hardware-based approach to scalable shared-memory multiprocessors in terms of the physical network interfaces, a packet-based point-to-point communication protocol, and a cache coherence protocol for distributed shared-memory systems [41].

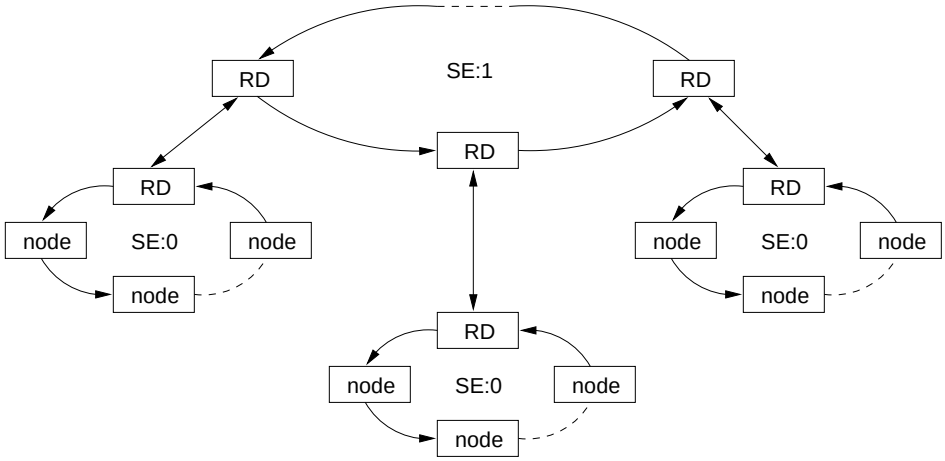


Figure 2.8: KSR-1 architecture

Institute of Computer Science (SICS), and at Bristol University. The original DDM papers envisaged a system with a hierarchical tree-shaped interconnect built of buses, with the processing nodes at the leaves of the tree, and higher levels of the hierarchy holding directory information for their sub-trees [50]. A hierarchical network is employed to ensure that all data can be accessed in a time bounded by $O(logP)$. The hierarchy also supports the combining of requests. If a directory node receives a read request for a data line which another processing node is requesting, the later request is added to a list of nodes waiting for the data line. When the reply is received by the directory node, it sends the data line on to all the waiting processing nodes. The DDM hierarchy needs high connectivity and with large systems the wiring would be too complex to be practicable. However, by collapsing the hierarchy into a fat tree (an example of a fat tree was shown earlier in Figure 2.4(j)), it appears that it would be possible to support large-scale systems up to 4096 processing nodes [99].

The Kendall Square Research KSR-1 was the first commercially available COMA system [140]. Nodes in the system were connected together by rings. A small system with up to 32 nodes was connected in one ring (a selection engine, termed an SE:0). For larger configurations, a higher level ring (the SE:1) was used to connect together up to 34 SE:0's, forming a two level hierarchy of rings, as shown in Figure 2.8. In the larger configurations, each SE:0 ring included at least one ring directory (RD) cell, which supported the routing between the SE:0 and the SE:1 rings. Each node in the system had a processor with a 512 Kbyte local first level cache, and a 32 Mbyte attraction memory which was the node's share of the overall ALLCACHE. The KSR-1 suffered from variable performance, because the architecture required programmers to be careful about the allocation to processors of processes sharing the same data. The ordering of writing and reading processes on the same SE:0 affected performance, as did having some or all of the reading processes located on a separate SE:0 to the writing process [140].

2.1.7 Software Implementations

This thesis focuses on hardware implementations of scalable shared-memory. However it should be noted that there are also systems where the shared-memory model is implemented in software on distributed processing nodes. The distributed systems can be message-passing parallel processors or networks of workstations. Ivy is an early example of such a distributed virtual shared-memory (DVSM) system, where the shared-memory is implemented using software on a message-passing distributed-memory architecture [90]. A memory manager on each node implements the mapping between the physical local memory and the shared virtual memory address space, and keeps the address space consistent. The memory manager treats its local memory as a large cache (of pages rather than lines) for the shared virtual address space. A problem with DVSM systems such as Ivy and Munin [14] is that by using software to implement shared-memory on a message-passing architecture, they place a processing overhead on the processor at each node which slows down the actual application. To reduce these overheads, the systems usually transfer data between nodes in page size units. This reduces the number of messages, and so lowers the overall message startup costs. However it means that the unit of data coherence has to be at the page level (*i.e.* much larger than at data line size) or additional mechanisms such as page-merging are needed to provide more fine-grained coherence.

An active area of research is into how individual workstations or personal computers (PCs) can be connected together to form a cost-effective scalable shared-memory system. Such a network of workstations (NOW) or "pile of PCs" is attractive as an inexpensive platform for solving large-scale problems which have a high computation to communication ratio. These systems are becoming more feasible given the emergence of low-latency switch-based local area networks. However supporting a true shared physical address space in the presence of unreliable nodes and networks is still an open research area. The workstations also have to be configured to support the shared-memory model, and this is usually done in software running on each CPU (or on one CPU within each node in cluster-based systems). Research in this area includes the Berkeley NOW [8], the Princeton SHRIMP [18], and the Beowulf project [126].

2.1.8 Simple-COMA

A comparison by Stenström *et al.* between cc-NUMA and COMA architectures observed that a cc-NUMA system with page migration or replication is similar to COMA but with page-sized blocks in memory [125]. The simple-COMA (S-COMA) architecture, proposed by Saulsbury *et al.* , is an amalgam of COMA and DVSM which reduces the hardware and soft-

ware overheads associated with those approaches [112]. It addresses the page locality problems associated with straightforward cc-NUMA implementations by providing page replication and migration in the attraction memory. The design uses software to handle page allocation and hardware to handle coherence at the data line level. The result is a hybrid software and hardware COMA system which can be built using standard components. It should be noted that S-COMA has the drawback that allocating memory in page-sized units (as opposed to the COMA approach which allocates in line-sized units) can lead to wasted space where an application's access pattern does not display enough spatial locality [24].

To illustrate the workings of S-COMA, consider what happens on a read miss. The virtual address is translated to a physical address by the memory management unit. If the data line is present and valid in the local memory then the data is supplied to the caches and processor. Otherwise, if a page fault occurs (*i.e.* the page containing the data does not currently have space allocated for it) then space for the new page is allocated by the operating system in the local memory, although none of the data for the page is loaded in. If the allocation requires an existing page to be replaced, then any data in local cache that belong to the evicted page must also be evicted. All the data lines on the new page are set to invalid. If the data line matching the read miss is invalid (which it will be if a new page has had to be allocated) then the physical address is translated to a global identifier, and a read request is sent to the appropriate remote node. The remote node supplies the valid data line, which is then loaded into the appropriate page in local memory. The transaction is completed by loading the data to the local caches and processor.

2.1.9 Consistency

The need for coherent data was discussed in Section 2.1.1, where cache coherence policies were introduced to ensure that all copies of a data item hold the same value. In practice it takes time for the cache coherence policy to either invalidate or update a data item, and during that time the copies can be inconsistent. One way of ensuring that the consistency issue is dealt with in a predictable way is to enforce Lamport's guidelines for sequential consistency [82]. Lamport starts with the situation on uniprocessors where operations may be rearranged by a compiler (or at run-time) to improve the execution time, but the program is still correct if the result is the same as that obtained when the operations are executed in the order specified by the program. He points out that rearranging the order of operations running on multiprocessors can cause problems because of the sharing of data between processors. As a result, the correctness of programs can only be guaranteed if the operations on each individual processor are executed in the order specified by the program. A sequentially

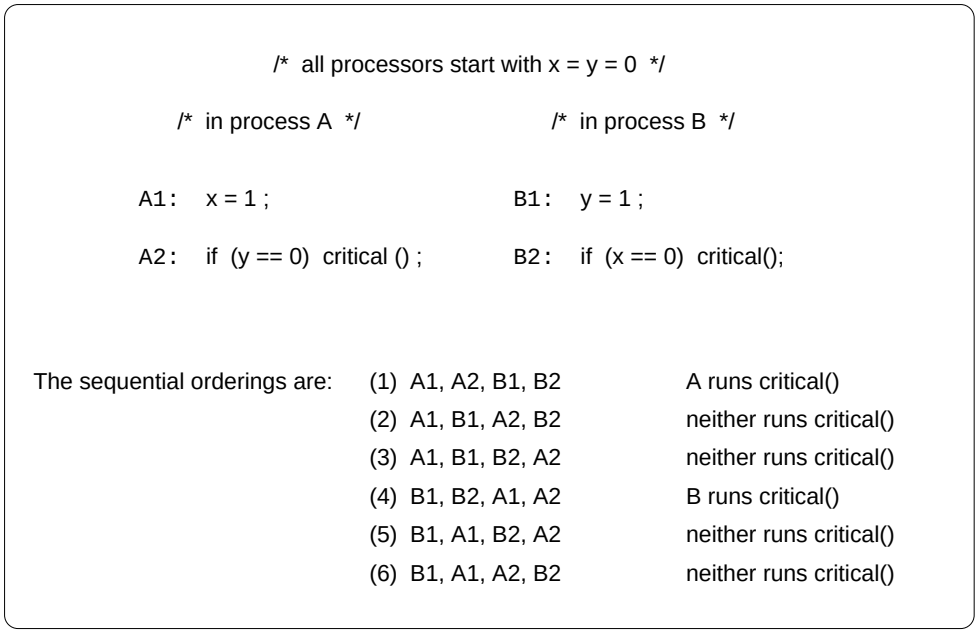


Figure 2.9: An example of sequential consistency

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

Figure 2.9 illustrates the sequential orderings where two processes A and B are executing on separate CPUs of a multiprocessor. In the example there are six possible sequentially consistent orderings of the statements A1, A2, B1, and B2. Four of the sequential orderings lead to neither process entering the critical function, whereas of the remaining orderings one only lets in A, and the other only lets in B. Without the ordering imposed by sequential consistency, *e.g.* if operation B2 executed before B1, it is possible that both processors would run the critical process.

The important thing about sequential consistency is that it enforces an ordering of accesses to shared data. Lamport recognised this and gave two requirements for sequential consistency [82]:

1. Each processor should issue memory requests in the order specified by its program.
2. Memory requests from all processors issued to a particular memory *module* should be serviced from a single first-in-first-out (FIFO) queue. This requirement can safely be "softened" to only require that all requests to the same memory *item* must be serviced in the order which they appear in the queue.

Sequential consistency does not preclude compiler or runtime optimisations which re-order

the instruction sequence for individual processors, but it does restrict these optimisations by imposing an ordering on shared-memory accesses. As a result the performance may suffer as the price of ensuring the correct execution of the application, because processors may have to stall while they wait for data to become available. Alternative approaches to consistency will be discussed in Section 2.3.3.

2.1.10 Node Controllers

A feature common to many scalable shared-memory designs, including the Stanford DASH [88], the Sun S3.mp [102], and the HP/Convex Exemplar [1], is a separate "controller" which manages access to the local memory. This controller is known by many names including memory controller, coherence controller, and hub; in this thesis it will be referred to as the node controller.

Using a separate node controller to handle all accesses to memory, whether from the local CPUs or those on remote nodes, allows commodity microprocessors to be used for the local CPUs, and leaves the local CPUs free to run the application code. The node controller is responsible for accessing the local memory, maintaining the directory information, handling all access requests from remote and local CPUs, and enforcing the cache coherence protocol. The protocol processing can be implemented in hardware on the node controller to obtain the best performance, as for example is done in the SGI Origin2000 [85]. Alternatively there can be facilities for programming the node controller which enable software correction of protocol errors, and allow extra functions to be added to the protocol. This approach is popular because of the flexibility it offers, and examples include the Sequent NUMA-Q [92], the Stanford FLASH project [81], and the Wisconsin Typhoon project [110].

Using a separate node controller does have some disadvantages. All accesses to the local memory by CPUs have to go via the node controller, which may already be occupied with processing a message received from a remote node. The local CPU's request may have to wait for service: the length of this delay depends on the node controller's architecture, and the design tradeoffs are discussed further in Section 2.3.2 of this thesis. In addition, there is the cost of dedicating a processor to act as node controller.

2.1.11 Summary of Scalable Shared-Memory

This section has demonstrated that a wide range of architectural designs have been proposed over the last two decades with the aim of providing scalable shared-memory. However it should be noted that there is a convergence towards systems which are made up of a collection of

essentially complete computers, each consisting of one or more microprocessors with private caches, and a large node memory managed by a node controller. The nodes are connected together by a robust communication network. This convergence is driven by the increase in microprocessor performance and memory capacity, and the very high cost of developing new processor technology. The high development costs for microprocessors can be offset against the extremely large market for commodity uniprocessors, whereas the cost of developing custom processors for large scale systems is becoming prohibitive given the long development time and the relatively small market.

2.2 Performance Problems with Scalable Shared-Memory

Although shared-memory systems are now enjoying commercial success, particularly the smaller SMP designs, the larger scale systems have a much smaller market. This is in part due to their cost, but it is also because of the performance problems that have been observed on these systems. The problems are not only caused by the system architecture, both in terms of non-uniform data access times and the various bottlenecks in the system, but are also the result of application design. Even more worrying for potential users of such systems is the different performance that the same program can obtain on different systems, and seemingly inexplicable variations in performance for the same application running on the same system.

This section examines the various causes of the performance problems found in cc-NUMA systems. The investigation covers the effects of:

- Finite system resources, *e.g.* network bandwidth and node controller service rate.
- Data locality, including page placement.
- The cache coherence protocol, *e.g.* the overheads of protocol messages and cache line conflicts.
- Algorithm design.

This catalogue of problems will be used in Section 2.3 as the framework for introducing the various solutions which have been proposed to alleviate the performance problems.

2.2.1 Finite System Resources

Multiprocessor performance may be limited by the same resource constraints which occur in uniprocessors, *e.g.* buffer sizes, memory access latency, and processor speeds. However for

shared-memory multiprocessors there are additional performance limitations arising from the interconnection network and node controllers.

Communication latency is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination [79]. Many factors affect the communication latency of a network, such as its topology and bandwidth. The network is a fixed performance resource that must bear a heavier load with each additional device placed on it. At some point the network becomes overloaded and adding more processing nodes provides no additional systems performance, or can even reduce the total throughput. The problems can occur because of congestion at intermediate switches on the route a message takes through the network (*i.e.* "hot spots") [105], or, as noted by Holt *et al.*, because of end-point contention where the node controller at the destination cannot service incoming messages fast enough to prevent a queue building up [57]. Occupancy is the time for which the node controller is tied-up performing one action and cannot perform another.

2.2.2 Data Locality

The programmer using a shared-memory system may view the memory as a centralised resource which is equally accessible to all processors. In practice, for distributed shared-memory architectures such as cc-NUMA, the memory is distributed among the processing nodes. Access to data held on remote memory will incur a greater latency than access to local memory, with typical ratios of between 2:1 and 14:1 on current cc-NUMA systems [60]. The use of caches aims to avoid the remote access latency by holding copies of data lines close to processors which have recently accessed the data. In addition, techniques such as tightly integrating processors and local memory may minimise the data miss latency [111]. However the initial placement of data can have the decisive effect on the overall performance of an application. In cc-NUMA multiprocessors, shared data is partitioned into virtual memory pages. Each page of shared data is then physically allocated to a node by the operating system when it handles the page fault associated with the first access to data on that page. The default policy for choosing this "home" node is usually based on the first node to touch a page.

First-touch allocates the page to the node which first accesses it, and this strategy aims to achieve data locality. It is important to distinguish between naïve first-touch and first-touch-after-initialisation policies. A naïve policy will allocate pages on a first-touch basis from the start of program execution. This is a problem for applications where one process initialises everything before parallel processing commences: all the pages end up on the same node (with overflow to its neighbours). It is preferable to employ a first-touch-after-initialisation policy, where shared memory pages are only permanently allocated to nodes once parallel processing has commenced.

A drawback of cc-NUMA is that each node has a relatively small second level cache (SLC). If a processor is accessing a large data structure from a remote home node (*i.e.* the structure is too large to fit into the SLC) then it has to keep accessing that home node to pick up data that has been evicted from the local cache. In contrast, the COMA approach would copy such a data structure into local memory, but there are then the overheads of keeping track of the data to ensure that at least one memory copy always exists and where to find it.

2.2.3 Issues Related to the Cache Coherence Protocol

A cache coherence protocol is essential for cc-NUMA systems to ensure that copies of data lines are up-to-date, but achieving this entails a number of overheads which can degrade the performance of the multiprocessor. These overheads include the messages needed to implement the protocol, conflicting access to cache lines, and how sharing patterns in applications are handled.

Protocol Overheads

Protocol overheads are the additional memory and messages which are required to support the coherence protocol. The memory overhead is the space needed to hold the directory information. The message overhead comes from controlling access to the directory information. Protocols which distribute the sharing list among the sharing nodes have to be carefully defined to ensure that the sharing list remains coherent when more than one node needs to update the list. For example, in SCI the following basic operations are defined for modifying a sharing list [41]:

(a) a node may join the list, becoming the head;

(b) a node in the list may delete itself from the list by serially communicating with its upstream and downstream neighbours, informing each in turn of its new neighbour (*i.e.* unhooking);

(c) a head node may purge (*i.e.* invalidate) all the other elements of the list (one at a time) to become a single element list.

To obtain a local copy of a data line, a node executes operation (a). It receives a copy of the data either from the home node's memory or from the previous head of the sharing list. If a node wishes to modify the line, it must first perform operations (b) and (a) if it is not at the head of the sharing list; once it is at the head of the list it performs operation (c). SCI carefully defines these operations to permit arbitrary concurrency in their execution.

In contrast, on the Sun S3.mp prototype, transactions which modify the sharing list are serialised by the use of a lock maintained in the directory entry at the home node [101]. This ensures that only one modification to a distributed sharing list is being made at a time, but this conservative approach comes at the cost of delaying some transactions which could have safely been applied.

Multiprocessor Cache Line Sharing

There are caching problems similar to those found in uniprocessors, for example cache line conflicts, and lack of locality (see Appendix A). However there are additional problems arising in multiprocessors because there can be more than one processing node with a cached copy of a data line. There are three types of cache line sharing:

1. **Active sharing:** this is where a data item is accessed by more than one processor during the execution of a program. Contention can occur where one processor updates the data item: all other cached copies of the item have to be updated or invalidated, and this introduces delays. In addition, where many processors share the same data line, there can be contention problems when obtaining the data line from the home node; this problem with widely-shared data is examined in more detail below.

2. **False sharing:** this is where processors share a line without sharing data items within the line, *i.e.* the line contains a number of items, none of which is used by more than one processor [136]. The problem occurs with writes to an item in a shared line: all cached copies of that data line have to be updated or invalidated, although the other processors never use the data item which was changed. False sharing results in more updates/invalidations being sent than are needed by the parallel application and its data sharing requirements. In addition, with an invalidation-based protocol, false sharing introduces further cache miss latency for the data line which has to be obtained again after invalidation.

3. **Passive sharing:** this occurs where shared data still remains in a processor's cache even though no objects on the line will be accessed by that processor again [34]. Since a write by another processor to any item in that line will require all other copies of the line to be updated or invalidated, it is desirable that the redundant data line is ejected from the cache after its last use.

The extent to which these sharing problems affect the performance depends on the memory access characteristics of the individual application programs and the shared-memory architecture. Cache line size is particularly relevant: larger cache lines would allow many data items

to be held in each cache line, which could be helpful if an application has locality of access. However, the larger line size can lead to contention for lines, especially if false sharing plays a significant role in the behaviour of an application. Agarwal and Gupta studied the influence of cache line size on performance for a four processor bus-based system, and found that false sharing had a significant impact when line sizes greater than 64 bytes were used (because the number of transactions and invalidations increased significantly) [4]. A more recent study by Torrellas *et al.* showed that poor spatial locality in the data and false sharing both caused variations in the miss rate of shared data as the cache line size was changed [136].

Widely-Shared Data

The types of access to data which occur in shared-memory applications have an effect on performance, and various protocols have been suggested to tackle the different types of access patterns: for example by Bennett *et al.* in Munin [14], and by Weber and Gupta [142]. Widely-shared data, where many (and often all) processors require a copy of the same data item, has tended to be ignored because such data usually forms a very small part of the overall shared-data used by an application. This was the case with Munin which characterised the main types of sharing for data objects to drive the use of appropriate protocols. The Munin categories included *write-mostly*, *producer-consumer*, and *read-mostly*. They identified the Gaussian Elimination (GE) benchmark as *write-mostly* because of the high level of writes by all processors to the matrix. However GE is known to widely-share the current pivot row data in each iteration, and the categorisation as *write-mostly* masked the bottleneck *producer-consumer* behaviour on the pivot row which occurs once in every iteration. Kaxiras *et al.* have demonstrated experimentally that a significant performance degradation can occur in each iteration of GE when a great many processors need to simultaneously access the pivot row's data items [69]. As the number of nodes in the system is increased, the bottleneck caused by widely-shared data becomes worse: requests for the same data line made simultaneously by more nodes will increase the contention at the home node. This bottleneck leads to performance problems which are not only connected with the number of nodes in the system, but also with the rate at which a node controller can deal with incoming messages, and with the network bandwidth.

2.2.4 Algorithm Design

The shared-memory programming model may relieve the programmer from managing data placement, but there is still some onus on the programmer to write a "parallel" program rather than being totally oblivious to the problems that can occur. For example, an application

where much of the work is done by one processor, leaving the other processors idle for most of the time, cannot be expected to achieve a good overall parallel speedup. Load balancing is complicated because in many parallel architectures it is difficult to predict the size of the subtasks assigned to various processors, and so it is not possible to divide the program statically among the processors and be able to guarantee that the processors will all have the same workload [79]. This can result in some processors being idle for part of the time, and this idle time is an overhead which can increase the overall execution time of the application.

The problems in algorithm design lead to the question of what is the "best" model of the parallel environment for programmers to use when designing parallel algorithms? The answer is the one that leads to both good performance and portability, but in practice these are conflicting goals. Programs written under an incorrect or inaccurate model often perform poorly, and their performance does not scale because of bottlenecks and poor utilisation of hardware. For example a model which is too abstract will not consider the realities of there being a hierarchy of memory access, and it will fail to capture the advantages of accessing local rather than remote data [84]. The performance is generally better when the programming model matches the underlying architecture, but architecture-specific applications have limited portability [24]. There is confusion over how to relate a parallel algorithm, the programming model which is used to implement it, and the architecture on which the program will run. To add to the problem, there is a lack of standards, which limits the portability of even "good" shared-memory applications [124].

2.3 Solutions to the Performance Problems

The performance problems with scalable shared-memory systems which were identified in the preceding section have been the spur for a wide range of research work. This has produced a wealth of partial solutions, some of which can be used together, although others may work against each other. This section covers the spectrum of the proposed solutions, although it does not claim to be an exhaustive survey. The aim is to provide a flavour of the many varied and overlapping schemes which are currently employed in commercial systems or which have been proposed by research projects. The solutions include:

- Latency hiding: prefetching, data forwarding, access decoupling, and multi-threading.
- Latency reduction: sharing list representation, node controller optimisations, compile-time analysis, victim caches, and cache bypassing.
- Coherence policy optimisations: message prediction, relaxed consistency models, and combining accesses to widely-shared data.

- Data placement: static data placement strategies (including randomisation and hashing), and dynamic page placement.
- Algorithm design: abstract models, load balancing, and performance tuning.

2.3.1 Latency Hiding

Latency hiding schemes aim to conceal the latency of data access from the CPUs. The latency hiding approaches include prefetching, data forwarding, access decoupling, and multithreading.

Prefetching and Data Forwarding

In prefetching schemes, data is retrieved from the memory hierarchy before a CPU needs it [139]. The simplest form of prefetching is instruction scheduling, where the sequence of instructions is re-arranged so that, for example, a load instruction is issued well in advance of arithmetic instructions which use the data. Static instruction scheduling, where the compiler rearranges the instruction order, was first used in the 1960's, and became popular when the use of RISC (reduced instruction set computer) systems and pipelining became widespread [54]. Dynamic instruction scheduling, where the hardware rearranges the instruction execution to reduce stalls at run-time, is found today in many microprocessors including the MIPS R10000 [146]. It should be remembered that these instruction scheduling techniques, when used in the microprocessors which are part of a multiprocessor, may compromise the sequential consistency model (as outlined in Section 2.1.9 of this thesis) by altering the timing of load and store operations when other processors are accessing the same data lines.

Some modern instruction sets, such as Hewlett Packard's PA-RISC [65], include a separate fetch instruction which enables the compiler to implement data prefetching without having to re-arrange the order of instructions. However it can be quite hard for a compiler to accurately predict where to insert a "fetch" in the instruction stream. If a fetch is performed too late then the data will not yet be in the cache when the CPU needs it. If the fetch is scheduled too early then the data may be evicted from the cache before it is used, to make room for a conflicting data line. In addition, using explicit fetch instructions increases the processing load, although the aim is that this will be more than balanced by a reduction in stall time. An alternative way of invoking fetch instructions is to allow the programmer to include prefetch annotations, for example as was proposed for the Wisconsin CICO programming model [84].

Hardware-based prefetching techniques rely on speculation about future memory access patterns based on previous access patterns. If the speculation is incorrect then superfluous data

will be brought into the cache. Such unnecessary prefetching can cause cache pollution, and will consume memory and network bandwidth. An example of hardware-based prefetching is the adaptive sequential prefetching proposed by Dahlgren *et al.* [26]. In this scheme the number of data lines speculatively prefetched into second level cache is varied by an adaptive mechanism. The adaptive mechanism uses an upper and lower threshold to decide whether to increment or decrement (or leave untouched) the LookaheadCounter. However, there is no indication that an upper limit is placed on the value of the LookaheadCounter. This suggests that the reason why the adaptive scheme fared little better than their fixed scheme was at least partly due to the counter getting to such a high value that it would take a long time to adjust back down in response to a change in conditions which meant that prefetching was no longer appropriate.

Data forwarding takes a different approach to moving data into cache ready for local processing. In this scheme a processor which modifies data (a "producer") hides the latency of cache misses from the users of this data (the "consumers") by sending the data to the consumers before they request it. To use the strategy a processor needs to know which other nodes are likely to need the data which it is modifying. The technique can be seen as the "complement" of prefetching, and the two techniques can be used together [76].

Access Decoupling

A different approach to latency hiding is access decoupling, where the responsibility for obtaining data is given to a separate instruction stream [119]. In a decoupled architecture, responsibility for data access and data processing is split between two separate processors at each node. The two processors, the access processor (A-processor) and the execution processor (X-processor) execute programs which have the same structure. The A-processor performs all the operations related to transferring data to and from the memory hierarchy, whereas the X-processor performs arithmetic operations on the data provided by the A-processor. The two processors exchange data by means of queues. The access decoupling technique has been investigated for a distributed virtual shared-memory (DVSM) architecture by the DELTA project at the University of Manchester [141]. A different approach was taken in the ACRI system where an additional control processor at each node was responsible for dispatching instructions and evaluating conditional branches ahead of the X and A processors [17].

Multi-Threading

Another strategy for latency hiding is multi-threading, where each CPU is given a number of processing threads between which it can swap, with the aim of keeping the processor

busy rather than having it stalled waiting for data. Swapping between processing threads (*aka.* context switching) involves preserving the program counter and register values for each thread, and may complicate the cache design. There are two main approaches to multi-threading: blocked and interleaved. In the blocked approach one thread is executed until it stalls waiting for data: at that point execution is switched to another thread. Alternatively, in interleaved multi-threading, the system regularly switches between threads so that each thread is steadily "moved forward". The choice of method depends on the tradeoffs between the cost of context switching and the fairness of the scheme.

The Massachusetts Institute of Technology's Alewife system is a 32-node research prototype which uses blocked multi-threading to switch between threads on synchronisation faults and remote data accesses; this permits good single thread performance and requires less hardware support than interleaved multi-threading [3]. The Communication and Memory Management Unit (CMMU) implements most of the Alewife features, including ensuring that the cache remains lockup-free. Multi-threading requires that a cache can cope with multiple cache misses from different threads, including resolving conflicts between multiple outstanding requests for different data lines which map to the same cache line [77].

The Tera Computer Corporation's Multi-Threaded Architecture (MTA) is a commercial system[4] which uses interleaved multi-threading [5]. The custom-built processors switch context every cycle between up to 128 threads (called instruction streams). In addition, each instruction stream can have up to eight outstanding memory references before the stream has to stall, which further increases the latency tolerance. The system supports fetch-and-add instructions, although these are implemented at the memory modules rather than in the network. The network connecting the processors to the memory modules has a 3D torus topology. Messages are processed by a randomised routing scheme that may detour packages if the output link in the correct direction is not working or is overloaded.

Grün and Hillebrand [47] have reported that both the Tera MTA and the Saarbrücken Parallel Random Access Machine (SB-PRAM) [37] show very good performance on the Integer Sort benchmark of the NAS parallel benchmark suite, completing the calculations in an order of magnitude fewer CPU cycles than reported for general purpose message-passing or shared-memory systems. It remains to be seen whether this reduction in cycles will translate into faster performance or will be cost effective. In addition it appears from the work in [47] that the multi-threading systems are getting good performance on the integer sort as much because they have fetch-and-add combining operations as because of the latency hiding resulting from multi-threading.

[4] At the time of writing, a four-processor system is being evaluated at the San Diego Supercomputer Center.

The argument for multi-threaded systems is that they place the burden of handling data access latency firmly on the architecture, and leave the application programmers free to concentrate on creating algorithms which have sufficient parallelism and which balance the processing load evenly across the available CPUs. The approach relies on there being sufficient execution threads to mask the latency, and so such systems have to be heavily multiprogrammed or applications must be heavily multi-threaded to achieve reasonable performance. The approach also sacrifices the tremendous cost benefits of using commodity microprocessors, facing head-on the enormous effort of designing and manufacturing the non-standard high-performance processors and the associated system software [24]. Given the relatively small market for scalable shared-memory systems, it is questionable whether the multi-threaded multiprocessor approach can survive.

2.3.2 Latency Reduction

The designers of cc-NUMA systems aim to keep the latency of retrieving data as short as possible, but this inevitably involves tradeoffs which aim to keep the common case fast and minimise costs. For example, the use of a bit vector in the directory to hold the sharing list for a data line allows the home node to send invalidation messages to all sharers when a write miss occurs for the line, but the bit vector is expensive in memory and is not scalable. The alternative of using distributed sharing lists reduces the storage overheads, but can increase the latency of write misses when there is a long list of sharers which has to be invalidated node-by-node.

Caching in multiprocessors introduces the data conflict patterns of active, false and passive sharing. These latency-adverse effects can be alleviated by compile time and execution time strategies. The techniques which can reduce the latency of cache misses include node controller optimisations, compile-time analysis, victim caches, and cache bypassing.

Node Controller Optimisations

In many cc-NUMA multiprocessors, the node controller handles all access to memory. Therefore it is not surprising that the node controller can become a performance bottleneck when access requests from remote nodes and local CPUs coincide. The node controller's occupancy can be reduced by implementing the controller's functions completely in hardware, although this sacrifices the flexibility offered by programmable node controllers [97]. It is also possible to reduce the occupancy by providing separate hardwired coherence handlers for local and remote data access requests; this was done in the Sun S3.mp prototype [101]. A different

approach is to improve the overall throughput of access requests by using pipelining and multiple issue techniques; the SCI Cache Link Interface Controller (SCLIC) in the Sequent NUMA-Q uses a three-stage pipeline and issues up to two instructions every cycle [92].

A significant part of node controller occupancy is due to the latency of accessing directory information, which is usually kept in memory. This latency can be reduced by caching directory information within the node controller, for example by using a fast directory cache within a hardwired node controller [96]. The results presented by Michael *et al.* indicate that communication-intensive applications can show a performance improvement of 40% or more using a hardwired node controller with a 4-way associative directory cache, each cache line containing between 4 and 8 directory entries. The optimum size for the directory cache was found to vary depending on the application programs and problem sizes. However the study showed that the optimum size for the directory cache grows sub-linearly with the problem size. The results also indicate that using an off-chip SRAM (static random access memory) directory cache realises most of the performance benefits seen with a directory cache that is on-chip with the node controller.

Compile-Time Analysis

Static analysis at compile-time can be used to transform the algorithm and/or the data layout to reduce the run-time delays that result from data misses. For example, false sharing can be reduced by grouping together data items that are only (or mainly) used by one processor ("group & transpose"), and by ensuring that items do not share a data line if they are written to by a number of processors ("pad & align") [59].

The SUIF (Stanford Universal Intermediate Format) parallelising compiler employs a number of techniques that aim to improve the locality of data use for applications [48]. The compiler tries to minimise the number and effect of cache misses by making the data accessed by each processor contiguous in the shared address space, and by ensuring that processors reuse data as quickly and as often as possible to improve the chance of the data still being held in cache. Other recent work in the area of compile-time data transformations includes McKinley [95] and Kandemir *et al.* [64].

Victim Caches

A problem with caches is that some data may be evicted only to be needed by the CPU soon afterwards. Jouppi proposed the victim cache approach to address this problem, with recently evicted lines being placed in the victim cache [63]. A victim cache is a small (2 to

15 entries) fully-associative cache into which is placed any line that is displaced from the direct-mapped cache that it supports. A new line put into the victim cache uses the "least recently used" (LRU) approach to choose which of the current victims it will replace. In the case where a miss in the main cache hits in the victim cache, the contents of the two cache lines are swapped, *i.e.* the victim goes into the main cache and the line it displaces swaps into the victim cache. The victim cache alleviates the problem of conflict misses in the cache hierarchy.

Jouppi's work was on relatively small cache sizes. For larger caches (128 Kbytes and above) the small size of the victim cache meant that it could get swamped, with the result that the benefit of using a victim cache was reduced. A more recent scheme, selective use of victim caches, has been investigated by Stiliadis and Varma [127]. This scheme employs a history of data line use to decide whether to use the victim cache for a particular data line. In addition, interchanges of data lines between the main cache and the victim cache are performed selectively, and incoming data lines destined for the FLC may instead be placed in the victim cache if the prediction algorithm determines that they are unlikely to be accessed in the future.

Victim caches have been studied in the context of uniprocessor memory hierarchies. For multiprocessors, the use of victim caches could have latency penalties from having to check for the presence of data lines in the victim cache when processing invalidation or update requests generated by other nodes.

Cache Bypassing

One solution to the problem of passive sharing is to not put a data line into the cache hierarchy if it is known to only be needed once. For example, Johnson and Hwu use an adaptive policy to determine whether data should bypass cache [62]. Their work is on uniprocessor cache hierarchies, and aims to minimise cache miss latency by reducing cache conflict and capacity misses. They target the cache pollution caused by infrequently-used data, *i.e.* data which is read into cache and then not used again while it remains cached. They detect such infrequently-used data by keeping track of the access patterns. The tracking is done at the level of macroblocks, which are somewhere between a data line and page in length, because it would give too large an overhead to track at data line level, and they find the results are still good using this coarser tracking. A counter for each macroblock is incremented when data is accessed, and decremented when a conflicting data line would like to evict the data from the cache. Comparison of counters for conflicting data lines determines whether the new data line replaces the resident data line or bypasses the cache.

2.3.3 Cache Coherence Policy Optimisations

There are performance issues associated with cache coherence protocols because of the overheads of maintaining the multiple copies of a data line, and the network traffic caused by the additional messages. These can be partially addressed by techniques such as message prediction, which aims to reduce the number of messages, and relaxed consistency, which aims to allow a process to continue executing before, for example, its update transaction has been seen by all the other processes. There is also the possibility of optimising the choice between the write-update and write-invalidate strategies.

Message Prediction

A hardware approach to avoiding excessive invalidations is provided in the Cosmos coherence message predictor which monitors access patterns and uses the information to accelerate coherence protocols [98]. Predictors sit beside each directory and cache module to monitor coherence activity and request appropriate actions. If, for example, a directory predictor anticipates that a processor asking to "share" data line B will subsequently ask for data line B to be "exclusive", the directory can respond to the "shared" request by providing "exclusive" access to data line B. Cosmos is able to predict the source and type of the next coherence message with an accuracy of between 62% and 93% for the results reported in the paper, the high prediction accuracy being the result of detecting regular coherence message patterns associated with specific data lines. Cosmos' lower prediction accuracy for the Barnes application (between 62% and 69%) occurs because Barnes periodically rebuilds its principal data structure [11, 144], thereby moving logical nodes with stable sharing patterns to different memory addresses, and this eludes Cosmos' sharing pattern detection.

Compile-time analysis can also be used to reduce the coherence policy overhead. For example, Skeppstedt and Stenström have proposed compiler algorithms which reduce the overhead of transferring ownership of a data line [117]. The algorithms work by detecting a load miss request followed by a write request for the same data line from the same processor. The initial read is marked to provide a hint to the cache to obtain an exclusive rather than shared copy of the data line.

Relaxed Consistency Models

The ordering of memory accesses imposed by sequential consistency ensures that a shared-memory program will execute correctly [82]. However it disallows some hardware and compiler optimisations that would change the ordering of shared-memory accesses. For this reason, a

number of relaxed consistency models have been proposed. These support the uniprocessor consistency model for accesses made by a single processor but relax the ordering constraints on how a processor's accesses are observed by other processors [2].

Relaxed consistency models weaken the ordering requirements to allow for more aggressive optimisations; the aim being to reduce the cost of memory access by masking the latency of write operations. Under the relaxed models, memory is only consistent at certain synchronisation events, which allows the protocol to buffer, merge, and pipeline write requests as long as it respects the consistency constraints specified in the model. As a result, a system with relaxed consistency changes the programming model and so programmers have to take into account that an update cannot be relied upon to be immediately visible to all other processors. An example of a relaxed consistency model is the weak consistency described by Dubois, Scheurich and Briggs [32]. In a weakly ordered system, coherence is only enforced at synchronisation points (which are implemented via hardware-recognised synchronising variables). Between synchronisations no assumption can be made on the order in which stores will be applied, except that the order of successive stores by a processor to the same address is respected.

The release consistency model proposed by Gharachorloo *et al.* extends weak consistency [39, 40]. Each memory access is classified as either an ordinary access, or an acquire, or a release. A *release* indicates that the processor is completing an operation on which other processors may depend: all of the releasing processor's writes must now be made visible to any processor which performs a later acquire. An *acquire* indicates that the processor is beginning an operation that may depend on some other processor: all other processors' writes must now be made locally visible. An implementation, lazy release consistency, has been proposed by Kontothanassis *et al.*, in which write operations are overlapped with processing, and invalidations are delayed until *acquire* operations [75]. The paper reported that the lazier protocols degrade performance when implemented in hardware.

In the last decade there has been a considerable amount of work done on the performance benefits that may accrue from using a more relaxed approach to consistency. However Hill has recently argued that multiprocessors should support simple consistency models such as sequential consistency because the more complicated models do not give a sufficient improvement in performance to justify exposing programmers to their complexity [56]. This judgement is based on recent hardware optimisations which have reduced the performance gap between various consistency models, and the lack of quantitative data on the benefits of relaxed consistency models. The argument for sequential consistency is bolstered by the results for database workloads presented by Ranganathan *et al.* [107]. They note that using

the hardware techniques of prefetching and speculative loads enabled a sequential consistency model to achieve performance which approaches that of the release consistency model with the same hardware optimisations.

Support for Widely-Shared Data

A number of protocol enhancements have been suggested to tackle the performance problems which can occur when there are many accesses to the same data item by different processors. In cc-NUMA multiprocessors the main bottleneck is access to the directory information at the home node, but there may also be problems with congestion in the network, and delays in updating distributed sharing lists. One tactic is to combine requests which affect the same data line in the network, so that fewer requests go on to the home node. This approach was used in the NYU Ultracomputer, where it applied to both read and write requests [45]. In that system, the combining was limited to requests which were concurrently held in an intermediate input buffer. Similar combining of requests is also performed in the network hierarchy of the SICS DDM [50] and KSR-1 [140] COMA systems.

Another form of combining is the "eager combining" proposed by Bianchini and LeBlanc [16]. Accesses to "hot" data, *i.e.* data lines which have been marked as widely-shared by the programmer, are treated differently by the cache coherence protocol. Each hot data line is given a fixed number of server nodes. When a node makes a read request for a hot data line, the request is sent to a server node (selected using the requesting node's identity and the physical page number which contains the data line) rather than to the home node. If the server has the data line it is sent to the client, otherwise the client is noted as waiting for the data, and the server sends a read request to the home node. Subsequent requests from other clients for the same data line are queued at the server until the data arrives from the home node, at which point the server sends the data on to all the waiting clients. When a write request is issued for a hot data line which is in the shared state, the home node sends invalidation messages to all the server nodes for that data line, and the servers then pass the invalidation on to their clients. When a hot data line has its state at the home node changed from "modified" to "shared", the home node sends the updated data line to all its server nodes (this being the eager sharing which gives the protocol its name).

Eagerness is also used in the SESAME project where an eager sharing mechanism is provided in the hardware to work in conjunction with compile-time analysis [143]. Each eager-sharing interface uses a directory of the shared memory regions which contain shared variables to decide whether to broadcast any updates made to its local memory to other nodes. As with eager combining, the scheme relies on being given information about which data is

to be included in the eager sharing. Both the eager combining and eager sharing schemes can be regarded as providing a hybrid update-invalidate coherence scheme for writes to the marked data, because the underlying write-invalidate protocol is extended with a mechanism to selectively broadcast updates for widely-shared data.

In the Scalable Coherent Interface standard (SCI), the cache coherence protocol uses distributed doubly-linked sharing lists to keep track of the sharers of each data line [41]. The sharing lists are sequential structures, and so there is a problem with scalability for widely-shared data, particularly if there are many updates, because the invalidation of the sharing list is sequentialised. To address this problem, proposals have been made to modify the SCI standard to cater for widely-shared data [68]. The initial scheme relied on "recursive doubling" to impose a tree structure on top of the doubly-linked list used in SCI [61]. This proved rather cumbersome and has effectively been replaced by the GLOW extensions to SCI which use an explicit tree structure to hold the sharing list for widely-shared data [70].

The GLOW extensions to SCI intercept requests for widely-shared data by providing "agents" at selected network switch nodes. The intercepted requests are used to build sharing trees for these data items. The tree structure supports the combining of read requests, and allows for faster invalidation/update on writes. The original proposal relied on static analysis by the programmer or compiler to identify the widely-shared data. The more recent work on dynamic GLOW has found that the best performance improvements are achieved by detecting the program data access instructions which suffer long latency [69]. This instruction-based prediction relies on some customisation of the CPUs to supply program counter information along with data access requests.

Adaptive Write-Update and Write-Invalidate

In Section 2.1.1 of this thesis it was noted that neither the write-update nor the write-invalidate approach is best for all situations. The optimum strategy depends partly on the network characteristics and partly on the individual applications. Adaptive schemes in this area aim to obtain a balance between update and invalidate by having update as the default policy, and swapping to invalidate after more than x updates in a row, with x being modified dynamically to suit the application. Examples of such schemes include the distance-adaptive update protocols of Raynaud *et al.* [108] and the adaptive hybrid protocols proposed by Anderson and Karlin [7].

Anderson and Karlin describe two adaptive schemes - Random-Walk and Last-Three-Samples - for swapping between write-update and write-invalidate. The work builds on algorithms

introduced earlier by Karlin *et al.* for spin-locks [66], and uses the concept of "write runs" proposed by Eggers and Katz [34] to drive the adaptive algorithms for the individual data lines. A write run is a sequence of write references to a shared address by a single processor. The run begins with the processor's first write to the address, is incremented by one for each write by the processor, and ends with the first access (read or write) by any other processor. The two adaptive schemes modify the threshold value, *i.e.* the length of the write run at which write-update will swap to write-invalidate, on a line by line basis. This ensures that data lines with consistently long write runs will have their threshold reduced to zero (which favours write-invalidate), while those with consistently short write runs will have their threshold increased to allow for active sharing (*i.e.* the balance tips in favour of write-update).

The Random-Walk protocol maintains a separate threshold T_b and counter C_b for each data line, as well as an overall threshold invalidation ratio R. At the start of running an application, the T_b and C_b for all the data lines are set to zero. During a write run the default policy is write-update, but the C_b for the data line is decremented by 1 for each write, and when it reaches zero write-invalidate is used instead for the rest of the write run. At the end of each write run, the T_b for the data line is adjusted as follows: if the write run $> R$ then T_b is decremented by 1 (but will not go below zero), otherwise T_b in increased by 1 (but will not go above R). The C_b for the data line is then set to the new T_b value.

The other protocol presented in the paper, Last-Three-Samples, differs only in how the T_b value is adjusted at the end of each write run, aiming to use the last three write run lengths together with probability distribution information to adjust the data line's threshold more quickly than the ± 1 of Random-Walk. However the overheads of storing the last three write run lengths for every data line, together with a static table of optimum thresholds at every node, are not rewarded with further performance gains over Random-Walk.

Anderson and Karlin conclude that one needs to weigh the benefit of using an adaptive protocol to obtain performance equal to the better of write-update and write-invalidate, against the additional complexity and hardware needed to implement the adaptive protocols. They also note that the performance advantages of the adaptive protocols increased with the number of processors and larger cache size [7].

2.3.4 Data Placement

The conflict which can occur when several processors require access to the same data line, *i.e.* access to widely-shared data, has been addressed in the preceding section. However problems can also occur on cc-NUMA systems when several processors simultaneously require

access to different data items that have the same home node. When these items are held on the same page the conflict problem is reminiscent of the false sharing of data on cache lines, and the solutions are similar: *e.g.* separate the data items on to different pages, or use a smaller page size. Conflicts also occur where the data items are held on different pages with the same home node. This problem can occur when a first-touch page placement strategy is used, *i.e.* where the first node which accesses data on a page becomes the home node. Such inter-page conflicts can be addressed by using a different static page allocation policy, or by employing dynamic page placement.

Static Page Placement

A common alternative to the first-touch policy is to use a round-robin approach, where the allocation routine cycles through the nodes allocating a page in turn to each node. This approach gives a more even distribution of data across the system, and so reduces hot spots. Unfortunately, this policy can lead to problems with applications which have been written with locality in mind, since it is likely that few of the pages accessed by a node will have been allocated to it. It is also possible that the policy could be still be vulnerable to "stride" access patterns in the application program, leading to inter-page access conflicts at a home node. As a result, first-touch is generally the default page placement policy, with round-robin being available as an option for improving the performance of some applications. Swapping between the strategies can solve performance problems on particular applications, but the performance problems caused by static page placement may indicate the need for more flexible placement policies.

The first-touch and round-robin policies are generally the only static policies provided in commercial systems, for example the SGI Origin2000 [85], primarily because they are relatively easy to implement. It should be noted, however, that there has been a considerable amount of work on randomised shared-memory where data is allocated to memory in smaller units than a page, *e.g.* at data line or even word size. Allocating data randomly to memory in smaller units reduces "inter-unit" access conflicts, but at the same time removes data locality. Examples of randomised shared-memory include the proposal by Hellwagner illustrated in Figure 2.10 [52], and the SB-PRAM [71]. The SB-PRAM uses simple linear bijective hash functions[5] to distribute data to different memory modules, and dynamic re-hashing to avoid run-time stride access patterns. Techniques to combat stride access patterns are also used in

[5] A hash function takes a key as input and returns an integer within a prescribed range as the result. The function is designed so that the integer values it produces are uniformly distributed throughout the range. A bijective hashing function will only map one input key to a particular location in the target range, *i.e.* several input addresses will *not* map to the same output address, and it ensures that every element in the function's target range will be mapped on to by some element in the input domain.

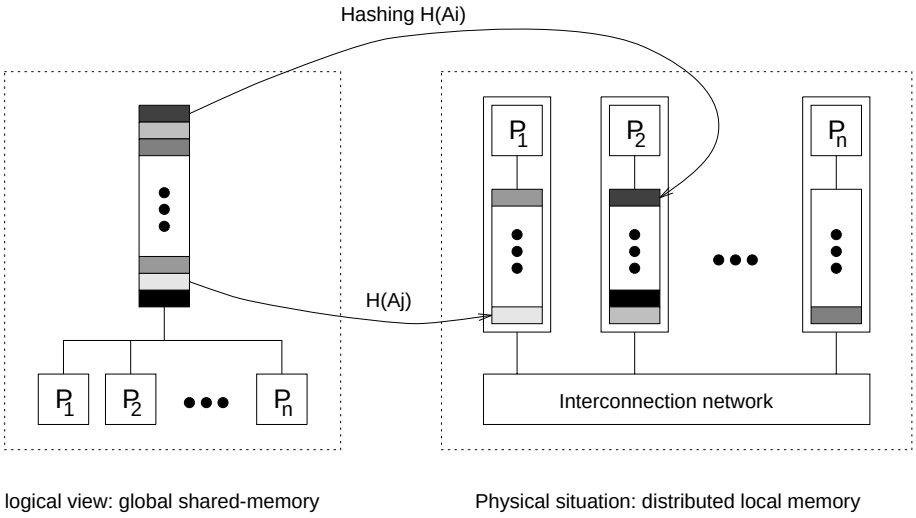


Figure 2.10: Randomised shared-memory [52]

systems that split the memory into banks to speed up access to data, for example the modulo interleaving described by Hennessy and Patterson on page 436 of [54].

Dynamic Page Placement - Page Migration and Replication

Dynamic page migration is similar to the migration of data to attraction memory in COMA systems, but involves moving whole pages rather than individual data lines. An early example of a migration scheme is given by Scheurich and Dubois, whose adaptive algorithm uses a principle of "geographic locality", *i.e.* it suits the quite common pattern in parallel algorithms where a node shares data with its immediate neighbours (*e.g.* north, south, east and west neighbours on a 2D grid) [113]. Another approach is to copy (*i.e.* replicate) complete pages to remote nodes which have suffered a data miss. Scheurich and Dubois made the important point that the initial data partitioning, *i.e.* how the data is split into pages, is just as critical as where these pages are then allocated [113]. Given that migration is page-based, poor initial partitioning can defeat the best migration mechanism because two data structures on the same page may be updated by different processors.

The extensive experimental study of LaRowe and Schlatter Ellis focuses on the policies which determine the physical placement of pages, including dynamic migration and replication [83]. The study finds that there is no one policy that gives the best results for all the applications they tested. They note that improving performance is a complex issue, where, for example, the decision to migrate a page may overload the destination node, or occur too late to be useful, or cause network contention if too many pages are migrated at once. Addressing these problems leads to more complex dynamic page placement algorithms, and they can never perfectly predict future use. There is a tradeoff between the problem of preventing

unwanted page migration and the desire to migrate pages as soon as possible. Even in their small selection of benchmarks there are applications for which one is more important than the other, suggesting that it may be difficult to find a single best tradeoff point. LaRowe and Schlatter Ellis observed the problem of initial partitioning, noted earlier by Scheurich and Dubois, with their "fish" application (fishes and sharks in a 2D ocean): none of their dynamic paging policies were able to overcome the page level false sharing effect *i.e.* there were many data structures on each page, and the dynamic policies all led to pages thrashing between processors.

There is a large body of work dealing with dynamic page placement, mainly in the context of distributed virtual shared-memory systems (DVSM), *i.e.* systems which implement shared-memory in software, and which enhance the existing operating system paging functions to include moving/copying pages between the distributed memories. Recent approaches to dynamic page management in DVSM systems include the Cashmere project at the University of Rochester [74] and the TreadMarks project at Rice [6]; both systems use page replication. Cashmere and TreadMarks are compared in [73], each implemented on a 32 processor DEC Alpha cluster (8 nodes, 4 processors each). The main conclusion is that low-latency networks make fine-grain DVSM systems (*e.g.* Cashmere) more competitive with more coarse-grained approaches (*e.g.* TreadMarks), although further hardware improvements will be needed before systems such as Cashmere can consistently show superior performance.

2.3.5 Algorithm Design

Although there are various architectural strategies which can partially address the performance problems of cc-NUMA multiprocessors, there still has to be some parallelism in the user applications if the architecture is to have a chance of performing well. Programmers need some guidance on how to write a "parallel" program, albeit one which is portable and not tied to a specific implementation [124]. To this end, a number of abstract models have been suggested, including the parallel random access machine (PRAM), the bulk synchronous parallel model (BSP), and the LogP model. In addition, the programmer needs to consider the processing balance within an application. It may also be necessary to tune the performance of an application to suit a specific multiprocessor, and the effects of such performance tuning have to be considered.

Abstract Models of Parallel Architectures

The question of how to guide algorithm designers without tying the resulting programs to a particular system is a vexed issue. Too high a level of parallel programming model can result in applications which ignore all communication delays or assume infinite system resources. At the other extreme are overly specialised models which are based on a particular system design, for example a specific network topology. The Parallel Random Access Machine (PRAM) is the most popular model for analysing parallel algorithms [67]. This model assumes that a collection of processors compute synchronously in parallel, and can access any part of the global shared-memory in unit time. The PRAM is not physically realisable as a scalable system, although the SB-PRAM is an effort to get the same effect by hiding the memory access latencies through multi-threading [37]. The PRAM model has proved useful for gross classification of algorithms, but in general it does not favour algorithms which are suited to a cc-NUMA environment.

An alternative is the Bulk Synchronous Parallel (BSP) model which was originally proposed by Valiant [138]. This model attempts to provide a bridge between theory and practice by imposing restrictions on the programming model which represent the key performance bottlenecks. A BSP computer consists of a set of memory-processor pairs, a global communications network, and a mechanism for the efficient barrier synchronisation of the processors. The approach allows for latency to be hidden by running several jobs or "virtual processors" on each physical processor (*i.e.* multi-threading). A job which accesses memory is de-scheduled until the reply from the remote processor is received. This requires a minimum number of ready-to-run processes, termed the "parallel slackness". Valiant proposes a programming environment in which algorithms are designed for the PRAM model assuming an unlimited number of processors, and are then implemented by simulating a number of PRAM processors on each BSP processor (*i.e.* to exploit the parallel slackness).

The LogP model proposed by Culler *et al.* also aims to represent the key performance bottlenecks, but without imposing a programming structure [25]. LogP attempts to capture the performance bottlenecks with four parameters, which are illustrated in Figure 2.11:

L an upper bound on the *latency*, or delay, incurred in communicating a message containing a word (or a small number of words) from its source module to its target module.

o the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations. Holt *et al.* use a slightly modified LogP to model distributed shared-memory systems where **o** is the *occupancy* suffered by the node controller [57].

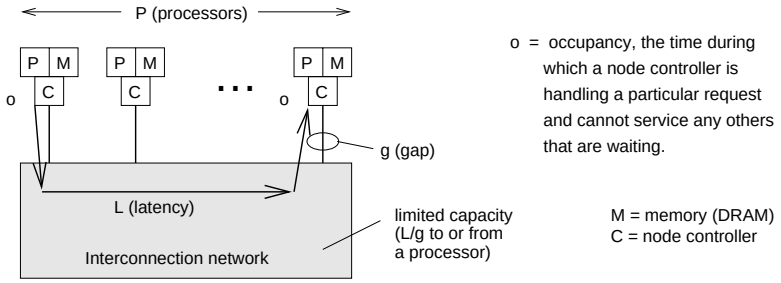


Figure 2.11: LogP based abstract system used in [57], adapted from [25]

g the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of **g** corresponds to the available per-processor communication bandwidth.

P the number of processor/memory modules. The model assumes unit time for local operations and calls it a cycle.

It is assumed that the network has a finite capacity, such that at most $[\mathbf{L}/\mathbf{g}]$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit. The LogP model avoids specifying the communication protocol or the programming style, and so is equally applicable to shared-memory and message-passing multiprocessors. At the same time the four parameters provide enough information to highlight some performance bottlenecks in parallel algorithms [25].

The models discussed so far have not included any performance penalty for conflicting data accesses. This has in part been addressed by Dwork *et al.* with a formal complexity model which reflects the fact that the performance of multiprocessor algorithms is heavily influenced by contention, *i.e.* the extent to which processes access the same location at the same time [33]. The trade-offs in their model capture the notion that one can reduce contention when concurrency is high only by paying a cost when contention is low, and vice versa. In the model, simultaneous accesses (resulting from cache misses) to a single memory location are serialised. Only one operation succeeds at a time, and other pending operations must stall. Their measure of contention is the worst case number of stalls that can occur. They freely admit that the model, like all complexity models, represents an abstraction of real architectures. For example, all memory accesses are serialised at the granularity of individual data locations, whereas in practice serialisation occurs at coarser-grained levels of memory (*e.g.* at data line and at page level).

There is as yet no consensus on the best parallel programming model, but the trend is towards models which include some element of communication costs without tying the model down

to a specific multiprocessor system. The aim is to guide the algorithm design so that the resulting program can perform reasonably across a range of architectures.

Load Balancing

An important aspect of algorithm design is the time spent waiting at synchronisation events. The simplest form of load balancing would be to ensure that every processor does the same amount of work and is busy at the same time. In practice the amount of "work" done by a processor also includes the latency of any data accesses, and this can vary from processor to processor. Using models such as LogP can help the programmer to partition the processing evenly between the available processors. However there are algorithms which are not amenable to such static load balancing because the amount of processing depends on information which is only available at run-time.

Dynamic partitioning techniques adapt to load imbalance at run-time. A semi-static example of this can be found in the Barnes-Hut galaxy simulation where the work done in one phase is used to assign stars to processors for the following phase [11]. A more dynamic approach is to provide a task queue, from which idle processors take the next available task. The task queue can be centralised (simple, but problems with contention) or distributed (more complicated to control, but less contention). More sophisticated still are algorithms where idle processors can "steal" work from overly busy processors, for example the Radiosity application in Stanford University's Splash-2 benchmark suite [144]. Task stealing implies some form of communication and can generate contention, so several interesting issues arise, *e.g.* how to minimise stealing, which processor to steal work from, how many and which tasks to steal at a time. A detailed survey of the techniques currently employed to achieve "partitioning for performance", including task stealing, can be found in Section 3.1 of Culler *et al.* [24].

It should also be noted that research is being done into using hardware to speculatively execute parts of applications which static analysis cannot guarantee has no data dependence effects. This has the overhead of coping with "undoing" the work if a dependence is found. However it has the benefit that code which seems hard to parallelise can be run on nodes that might otherwise be left idle, so it can be regarded as a form of load balancing. An example of speculative parallelisation is the research currently being carried out by the IACOMA group at Illinois [147].

Performance Tuning

There still remains the practical problem that when a specific program performs poorly on a given architecture, the programmer needs some guidance on how to optimise the application. The confusing interactions of the various types of sharing and locality, together with the characteristics of the interconnection network and the coherence protocol, mean that a programmer may have to make educated guesses on how to restructure the application.

The memory reference behaviour of an application, at the most basic level, depends on the intrinsic nature of the application. However, the programmer still has considerable flexibility in manipulating the algorithm, data structures, and program structure to change the memory reference patterns in order to better exploit the memory hierarchy [94]. There has been a surge of interest in recent years in developing tools to support application performance tuning. The purpose of a performance debugging tool is to focus the user's attention on where a program is spending its time and to give as much insight as possible into how to reduce the time spent in performance bottlenecks. Many performance debuggers now exist, ranging along a spectrum from summary-level, low-overhead tools to more detailed, high-overhead tools.

At one end of the spectrum, performance monitoring tools such as Gprof [46] and Mtool [42] are intended to produce simple, high-level statistics with minimal overhead. At the other extreme are tools like SHMAP [31], which provides a reference-by-reference animated picture of program memory behaviour. In addition, tools such as MemSpy [94], SM-prof [20], and Clarissa [129] provide a range of detail levels, *i.e.* starting with a summary, the user can then focus on more detailed information.

LaRowe and Schlatter Ellis report that when first-touch page placement is used, the performance is better for all tuned applications (*i.e.* those written with the underlying architecture in mind) than for the untuned equivalents written for the UMA memory model where the application programmer does not have to worry about data placement [83]. However they emphasise that the tuned programs take longer to write and are less portable.

2.4 Performance Trade-Offs in the SGI Origin2000

The preceding section gave an overview of the many partial solutions which have been proposed to address the performance problems in scalable shared-memory systems. In reality, multiprocessors use some subset of the performance enhancing techniques, partly because of manufacturing cost considerations, but also because while some techniques work well together, others are not complementary. The SGI Origin2000 is a commercial cc-NUMA multiprocessor,

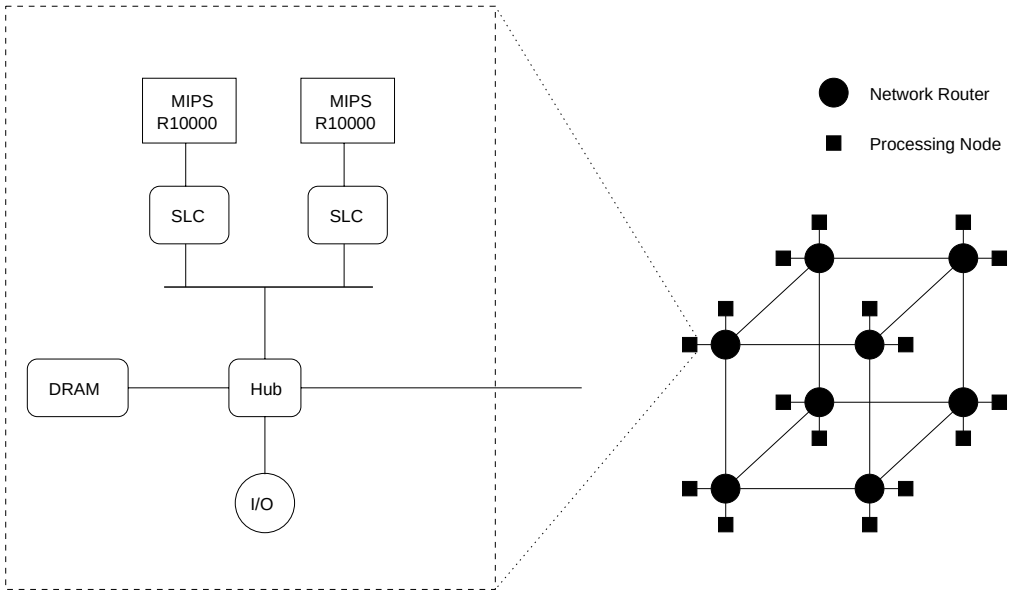


Figure 2.12: An SGI Origin2000 with 16 nodes (32 processors)

the design of which illustrates the practical trade-offs between cost and performance [85].

The Origin2000 system is capable of scaling to 512 nodes connected by a hypercube-based network. The topology is referred to as a "bristled" hypercube, because each network router has two nodes attached to it. Figure 2.12 shows the topology of a sixteen node system. Each node contains two MIPS R10000 processors (with integral FLC and off-chip SLC), a node controller connected to the network, a share of the overall DRAM, and an I/O interface. The node controller, known as the "hub", handles all the SLC misses by the local processors, as well as access to the local memory, and the processing of all incoming messages from remote nodes.

An important design goal for the system was to keep the remote access latency reasonable in comparison to local accesses, without slowing local accesses. A hypercube topology was used to reduce the maximum number of hops between any two nodes. The local processors in a node are connected by a bus to the hub, but there is no snooping between the two processors: the loss of this opportunity for fast access to data in the other SLC was a deliberate design decision taken to minimise the latency of accessing local memory and to simplify the cache coherence protocol [85].

Data placement is important on the Origin2000 because of its hypercube topology, and so the default page placement policy is first-touch to give some locality for data. The system also provides round-robin and dynamic migration policies as options which can be turned on for individual applications. The page migration facility has overheads in capturing and acting on page usage information, and should therefore only be used when tuning the performance

of particular applications [60].

The dynamically scheduled MIPS R10000 microprocessors [146] allow independent memory access instructions to issue out of program order, and multiple access requests can be outstanding at any one time. However the processor ensures that the operations complete in program order, and the writes are made visible to the memory system in program order. In conjunction with the coherence protocol, which provides transaction atomicity (*i.e.* a transaction affecting a data line will be fully completed before any other transaction affecting that data line can proceed), this satisfies the conditions for sequential consistency as discussed in Section 2.1.9 of this thesis.

The invalidation-based cache coherence protocol is similar to the protocol used in the Stanford DASH prototype system [88]. However the Origin2000 protocol has several enhancements which are designed to reduce the size and number of messages, and it also has an improved deadlock avoidance scheme [85]. A distributed bit vector directory is employed, and the 64-bit vector scales by being used as a "coarse vector" when an application is running on more than 64 nodes, *i.e.* with N nodes each bit will correspond to $N/64$ nodes. This coarse vector approach can result in nodes receiving invalidations for data lines which are not held locally, so the protocol has to handle such spurious invalidations, and the spurious invalidation messages will increase the overall network traffic.

Jiang and Singh have carried out a performance study [60] of the Origin2000 using applications from the Stanford Splash-2 benchmark suite [144]. Their study found that the FFT application (with 1 million points) suffered from large memory delays on a 32 processor system. These delays were due to contention, and were unbalanced across the 32 processors. They were able to get some improvements on the Origin2000 by using prefetching, or by forcing only one CPU at each node to run the application, but the scalability was still quite poor. They suggested that this contention problem on the Origin2000 was caused by two processors sharing each hub controller. Their results also demonstrated that the contiguous Ocean application needed a larger problem size than the Splash-2 default of 258×258 to be scalable on the Origin2000, with scaling problems showing at only 32 processors. They attributed this to the large (128 byte) line size and the memory access stall time.

2.5 Conclusions

This chapter has examined the current state of shared-memory architectures, the performance problems associated with the approach, and various strategies which have been proposed to deal with the performance problems. One thing that is clear from the survey is that a

solution proposed for one problem is often also useful for alleviating other problems. For example, caching in the memory hierarchy was the inspiration for using directory caches within the node controller. Another example was the widespread applicability of adaptive algorithms, although commercial examples of adaptive strategies being used are rare because of the overheads of capturing and acting upon the usage data. Such opportunities for re-using a strategy in different contexts was the prime motivation for the wide scope of the chapter. By understanding the many proposals which have been put forward to improve the performance of shared-memory systems, it is then possible to avoid any known pitfalls and to grasp how some strategies could be used together in new ways.

This thesis is concerned with reducing the need to tune application programs, by providing stable performance on cc-NUMA systems. Stable performance for scalable shared-memory multiprocessors can only be achieved by addressing the architectural problems which lead to long latency when accessing remote data. For example, it was shown in Section 2.2.3 that access to widely-shared data has tended to be ignored by previous researchers because data access patterns have been categorised by how often they are used rather than by the severity of the performance problems that they can cause. Although, as discussed in Section 2.3.3, some architectural proposals have been made to support widely-shared data, the node controller contention caused by such data is still an open research area.

One thesis cannot hope to find a solution for all the performance woes which hamper the goal of scalable shared-memory. However, by concentrating on a part of the cc-NUMA architecture which is still causing problems despite all the research work and commercial implementations of the last ten to fifteen years, the aim is to provide a novel solution to one of the main causes of node controller contention.

Chapter 3

Node Controller Contention

Given that the node controller handles all accesses to the local memory at a node, it has the potential to cause performance problems. It was seen in the previous chapter that although node controllers represent an obvious bottleneck, and despite strenuous attempts to reduce node controller occupancy in order to improve the throughput of requests, there are still performance problems. An example of this is found on the SGI Origin2000 [60].

Contention at node controllers occurs when there is a burst of incoming messages, and can be exacerbated by requests for service from local CPUs arising from local cache misses. To examine the problem, this chapter gives an overview of the cc-NUMA architecture used to evaluate contention, followed by details of the cache coherence protocol and eight benchmark applications. These applications are then used to demonstrate, by means of simulation, the performance anomalies which arise from node controller contention. Finally the causes of the contention are discussed and analysed.

3.1 System Architecture

This section presents the representative cc-NUMA design which is simulated for this thesis. The system is illustrated in Figure 3.1, and consists of a set of nodes connected by a network. Each node contains a processor (CPU), with integral first level cache (FLC) and translation look-aside buffer (TLB), a large second-level cache (SLC), some local memory (dynamic random access memory - DRAM), and a node controller. The processor, SLC, DRAM, and node controller are interconnected by two decoupled buses. The node controller sends messages to, and receives messages from, the network, and also handles all SLC misses from the local CPU. The node configuration is similar to a number of research and commercial systems, including Stanford's FLASH [81], Sun's S3.mp prototype [102], and the SGI Origin2000 [85].

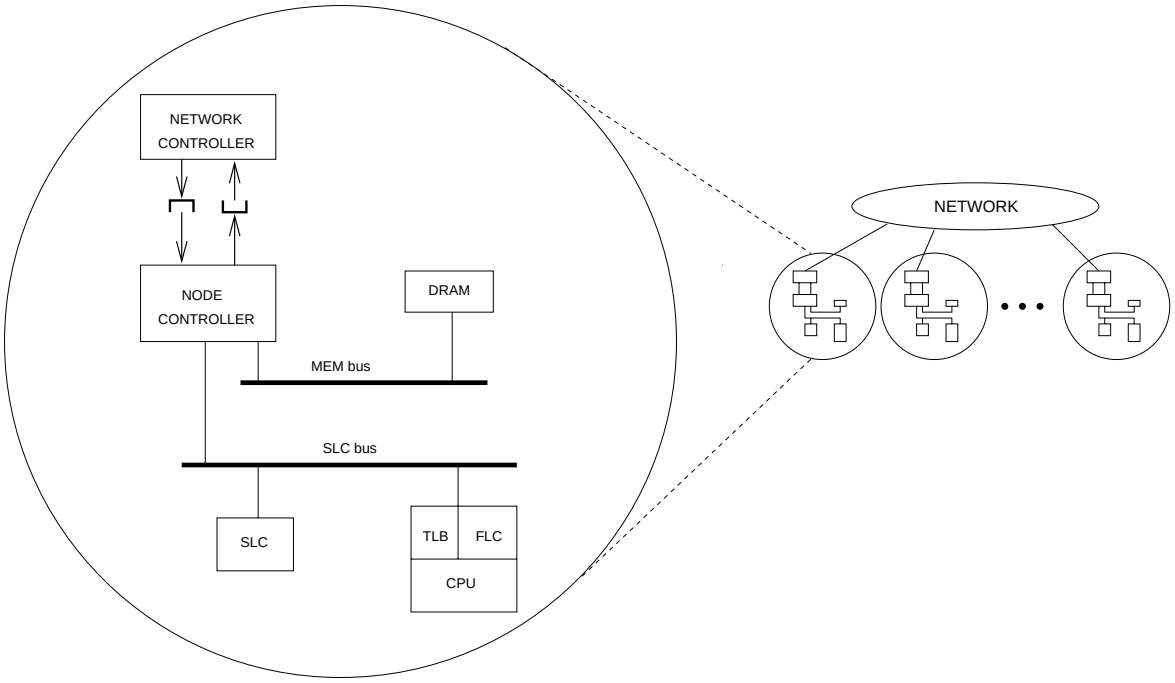


Figure 3.1: The system architecture

The processor and the caches are attached to the SLC bus, whereas the DRAM and the node controller are attached to the MEM bus. This decoupled bus arrangement allows the processor to access the SLC at the same time as the node controller accesses the DRAM. This can be expected to occur frequently because the FLC is small and write-through: many memory references which cannot be satisfied by the FLC will be satisfied by the SLC. The decoupled bus arrangement therefore reduces contention between the processor and the controller. The contention for the buses and the node controller which will occur, because of the conflicting demands from incoming messages and the local CPU, is simulated in detail.

A large direct-mapped cache is used for the SLC. Direct-mapped caches have more conflict misses due to their lack of associativity because any one data line will only map to one specific cache line rather than the n choices of location found in an n-way associative cache. However the performance of direct-mapped caches for hits is better than set-associative caches because of the simpler hardware required to match the address with the cache location [63].

The assumption that all instruction accesses hit in the FLC has been made in order to focus this study on the effects of data misses. The assumption is based on a "rule-of-thumb" described by Hennessy and Patterson, *viz.* that application code has a very high level of locality (90% of execution time uses only 10% of instruction code) so the instruction misses are small in comparison to data misses [54]. This assumption holds true for the benchmarks used in this thesis.

3.1.1 Cache Coherence and Consistency

The caches are kept coherent using an invalidation-based, distributed directory protocol. The protocol is the Stanford distributed-directory protocol proposed by Thapar and Delagi [134], and is similar to that used in the Sun S3.mp prototype [102][1]. The caches that share a copy of a data line are linked together in a list. The protocol is described in detail in Section 3.2.

The cc-NUMA architecture is modelled with sequential consistency (see Section 2.1.9). Sequential consistency does not accommodate some of the latency-hiding techniques which are possible under more relaxed consistency models (see Section 2.3.3 of this thesis). However, unlike the relaxed consistency models, sequential consistency does not require the application programmer to mark the requests that have consistency implications; nor do barriers have to be inserted just to ensure that consistency has been restored. In addition, recent studies suggest that advances in hardware have reduced the performance gap between sequential consistency and the more relaxed models [107], which starts to tip the balance back in favour of sequential consistency given its simpler programming model and coherence protocol [56].

3.1.2 Network Configuration

The network is a contention-free full crossbar; therefore a given type of message will take the same time to travel between any two nodes. This straightforward network topology was chosen for two reasons: to have a clear notion of the effect of network delays on the overall results, and to avoid tying the results to a specific network topology. This approach is also taken by Culler *et al.* in the LogP model, and stems from there not yet being consensus on the "best" network topology [25]. The networks of new commercial systems are typically different from their contemporaries and also from their predecessors. In addition, production systems allow for network faults, and this means that the actual interconnect experienced by an application may vary from run to run on the same machine.

3.1.3 Page Placement Policy

The page placement policy used in this work, first-touch-after-initialisation, was chosen after an investigation into the effects of different page placement policies [131]. The study also examined the interaction of the page placement policy with the contention-reducing strategies proposed in this thesis. It should be noted that the first-touch-after-initialisation policy does not determine the home page for a node until parallel processing starts: in this way it avoids

[1] The differences between the Stanford distributed-directory protocol and that used in S3.mp are implementation details.

the shortcoming of naïve first-touch policies, which can result in the node which initialises all data structures becoming the only home node.

3.2 The Base Cache Coherence Protocol

The Stanford distributed-directory cache coherence protocol is based on invalidations, *i.e.* a write to a data line may only proceed when all other cached copies of that data line have been removed [134]. A distributed directory structure, implemented as a singly-linked distributed sharing list, is used to keep track of the identity of the nodes which have cached a particular data line. Each cached copy of the line has a pointer to the next node on the sharing list. The address of a given data line uniquely determines the home node for that line, based on the page which contains the data line. In the absence of any sharers the data line exists solely in the memory associated with the home node. The home node's SLC may contain a copy of a data line from the local memory, but this copy does not form part of the sharing list for the data line.

The sharing lists are in the form of singly-linked lists. This has the advantage of requiring less storage than other distributed sharing list implementations such as double linked lists or tree structures. Singly-linked lists have the disadvantage that traversing the sharing list, which is needed when one or all entries need to be removed from the list, takes time proportional to the number of entries on the list. This overhead is generally a fair tradeoff when sharing lists are short, given the saving in storage [135]. For long sharing lists the use of single linking can add to the latency of invalidation (removing all entries from the sharing list) in comparison to a tree-structure list, and also in comparison with a bit-vector approach (see Section 2.1.5 of this thesis). In addition, removing one entry from a long sharing list when a data line is evicted from cache is more efficient in doubly-linked lists. Although singly-linked lists have these overheads, the linking policy has been chosen for this work because its lower storage requirements generally outweigh the drawbacks.

A read miss to a data line by a remote processor will prompt the home node to send a copy of the data line to the requesting processor, which will place the copy in its second-level cache. In addition, the remote node is added to the distributed sharing list for the data line. Subsequent readers will be added to the sharing list in a similar way, with new entries added at the head of the sharing list. Figure 3.2 shows the situation when nodes j and k each have a cached copy of a line whose home is at node i.

The home node:

- either holds a valid copy of the line (in SLC, DRAM, or both), or knows the identity of

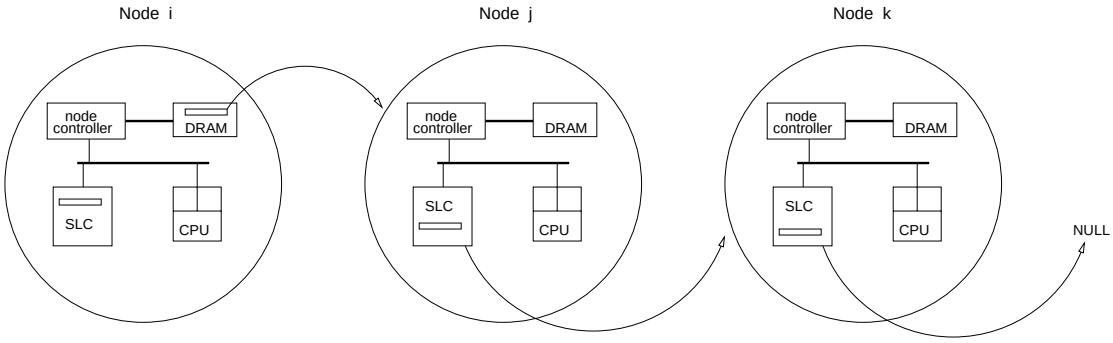


Figure 3.2: Example of a distributed sharing list

the node which does (the current "owner"),

- has pre-allocated space in memory to which the final replacement of the line from cache can take place, and
- holds directory information for the line (head and state of the sharing list) in DRAM.

3.2.1 State Sets and Transitions

The home maintains the state for all its data lines. There are four possible memory states for each data line:

Home-Invalid: the data is not held at the home node, but the home knows the identity of the owner of the data.

Home-Exclusive: the up-to-date data is held only at the home node, in the SLC and/or DRAM.

Home-Shared: the data is held at the home node and at one or more clients, all copies being consistent with the copy in DRAM.

Home-Locked: the sharing list is being updated. No other transactions are allowed to traverse the sharing list until the list is unlocked. This is done to avoid either using an out-of-date list pointer or having conflicting updates to the list pointers.

The state transitions for the DRAM directory entries are illustrated in Figure 3.3. A disadvantage with this state set is that, in the state Home-Exclusive, it is not possible to tell from the directory entry's state whether there is a more up-to-date copy held in the home node's SLC. This was a deliberate choice, because it saves the local processor (when it has Home-Exclusive ownership of a line) from having to update the DRAM as well as the SLC until the data line is evicted from the SLC. This choice is deliberately made at the expense

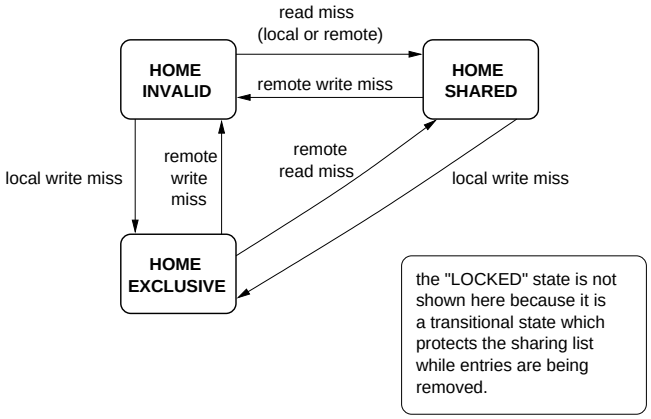


Figure 3.3: DRAM state transitions

of remote read requests. A similar state set is used for the same reason in other systems, *e.g.* Sun's S3.mp prototype [102].

The second-level cache line states are:

Shared: There is more than one valid copy of the line.

Exclusive: This is the only valid cached copy.

Invalid: The line contains no usable information.

The state transitions for the SLC cache lines are shown in Figure 3.4.

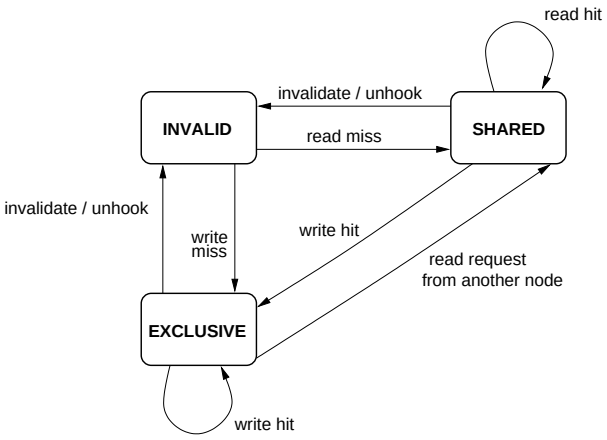


Figure 3.4: SLC state transitions

The message types needed to support the protocol are described in Appendix C.2. Extra message types were added to the ALITE simulator to handle the contention-reducing strategies described in later chapters, and these are also included in the appendix. There are two different classes of message: short messages which contain only control information (*e.g.* for

managing updates to a sharing list), and long messages which contain both control information and a line of data.

3.2.2 Example Transactions

To illustrate how the protocol works, Figures 3.5, 3.6, 3.7, and 3.8 show four examples of transactions: client read miss (home node invalid), client read miss (home node valid), client unhook, and client write miss. The key to the examples is shown in Figure 3.9.

Client Read Miss, Home Node Invalid

If a client tries to read a data line that is currently owned by another node, the `read-request` message will find the home node in state Home-Invalid. This prompts the forwarding of the request to the owner (`read-request-fwd`). The owner node will then change the state of the SLC line from Exclusive to Shared, and will send copies of the data line to both the client and the home node. The arrival of the `take-shared` message at the client will change the cache status from Invalid to Shared, and the home node's status will change from Home-Invalid to Home-Shared. The sequence of messages and the changes to the sharing list are shown in Figure 3.5.

Client Read Miss, Home Node Valid

The `read-request` message from the client will find the home node's state for the line to be Home-Shared. The home node will add the client at the head of the sharing list, and will send a `take-shared` message containing the data line to the client. The sequence of messages and the changes to the sharing list are shown in Figure 3.6.

Client Unhook

If an SLC data line copy is evicted, following a miss on another location which maps to the same cache line, the cached copy has to be "unhooked" from the sharing list. This is done by sending a `client-unhook-request` message to the home node, which will lock the directory entry. The unhook request is then forwarded along the sharing list until it reaches the entry preceding the node which requested the unhook. The identifier of this prior node is then passed to the requesting node which removes itself from the list by sending back the identifier of its successor. This is used to update the list pointer to skip the unhooking node. Finally a `client-unhooked` message is sent to the home node to prompt it to unlock the directory entry for the data line. The operations are illustrated in Figure 3.7.

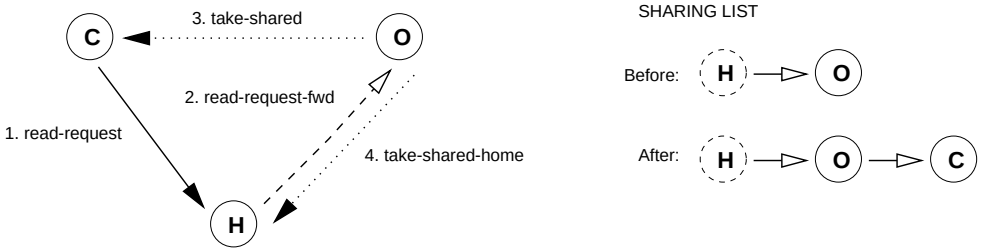


Figure 3.5: Client read miss, home is invalid

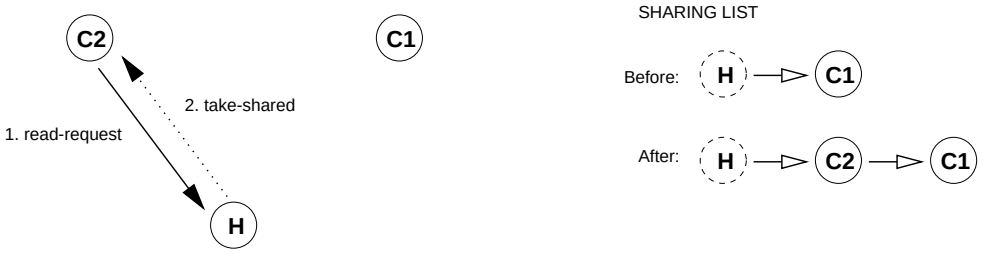


Figure 3.6: Client C2 read miss, home is valid

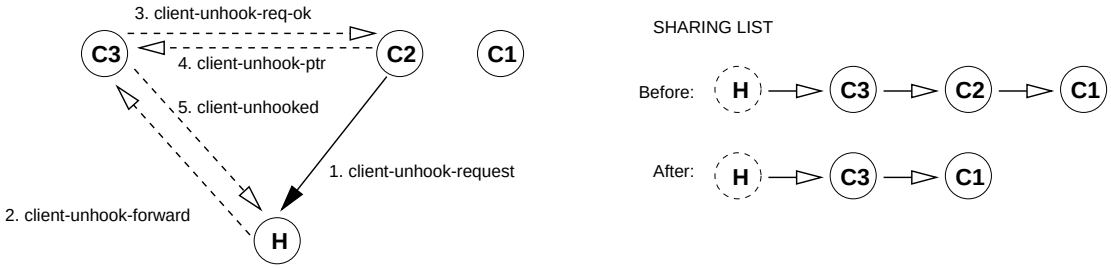


Figure 3.7: Client C2 unhook

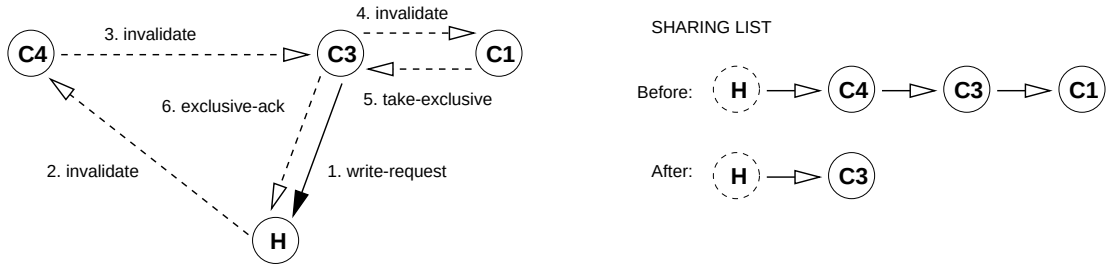


Figure 3.8: Client C3 write miss

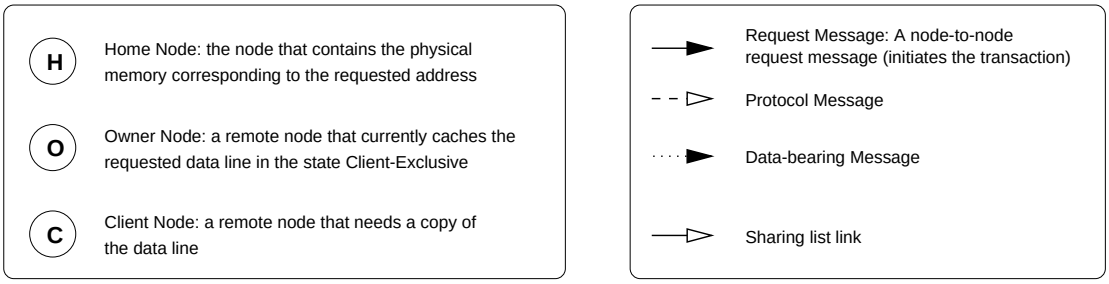


Figure 3.9: Message flow diagram notation

Client Write Miss

When a client processor wants to write to a data line, all other copies (including the home copy) must be invalidated before the update is allowed to proceed. This involves sending an `invalidate` request to the home node, which marks any home copy as invalid, and forwards the request along the sharing list. All sharing list entries will be invalidated, apart from the copy at the writing node. The sharing list is locked for the duration of this operation by setting the home node's directory entry status to Home-Locked. The last entry on the sharing list will respond to the invalidate request by sending a `take-exclusive` message to the requesting node. If the original write request involved a miss (*i.e.* the processor did not have a copy of the data line in its SLC) this message will also carry a copy of the data line. A `exclusive-ack` message is then sent by the requester to the home node to release the lock on the sharing list. While the line remains in the Exclusive state, the processor may write to it repeatedly without incurring coherence message traffic. The sequence of messages and the changes to the sharing list are shown in Figure 3.8.

3.3 Benchmark Applications

The applications are summarised in Table 3.1, along with the problem sizes used in this work. GE implements a Gaussian Elimination algorithm [128]. CFD is a computational fluid dynamics application modelling laminar flow [132]. The remaining six applications were taken from Stanford's Splash-2 suite [144], and were selected to give a representative cross-section of scientific and engineering shared-memory applications. A brief description of each of the applications is given below.

3.3.1 Barnes

The Barnes application simulates the interaction of a system of bodies (*e.g.* galaxies or particles) in three dimensions over a number of time-steps using the Barnes-Hut hierarchical

application	problem size
Barnes	16K particles
CFD	64×64 grid
FFT	64K points
FMM	8K particles
GE	512×512 matrix
Ocean-Contig	258×258 ocean
Ocean-Non-Contig	258×258 ocean
Water-Nsq	512 molecules

Table 3.1: Benchmark applications

N-body algorithm [11]. The Splash-2 implementation allows for multiple particles to be stored in each leaf cell of the space partition [58].

3.3.2 CFD - Computational Fluid Dynamics

Computational fluid dynamics is a major application area of high performance computing. The problems studied are concerned with the way fluids (liquids or gases) deform under the action of shear stress. The parallelisation comes from distributing the region of fluid to be studied among the available processing elements. The fluid flow algorithm implemented in the CFD application is capable of solving two-dimensional laminar and turbulent incompressible flows. In laminar flows, the fluid moves in smooth layers, or laminae, and the shear stress is the result of the microscopic action of the molecules. Turbulent flow is characterised by large scale, observable fluctuations in fluid and flow properties, and the shear stress is the result of these fluctuations.

The fluid dynamics situation being modelled in CFD is that of laminar flow in a square cavity with a sliding lid causing friction. The lid moves across the cavity, which introduces a zone of re-circulatory fluid. If the analysis grid of the square cavity is sufficiently fine, then small counter-rotating eddies should be observed in the corners of the cavity. The CFD algorithm was originally implemented on a distributed-memory message-passing architecture [132], but the version used in this thesis had already been adapted to run as a shared-memory application and optimised to reduce false sharing [129]. The program is written in FORTRAN, so it is converted to C (using "`f2c`") before being run on the ALITE simulator. The structure of the program is illustrated in Figure 3.10. The grid size for analysing the fluid is a run-time parameter.

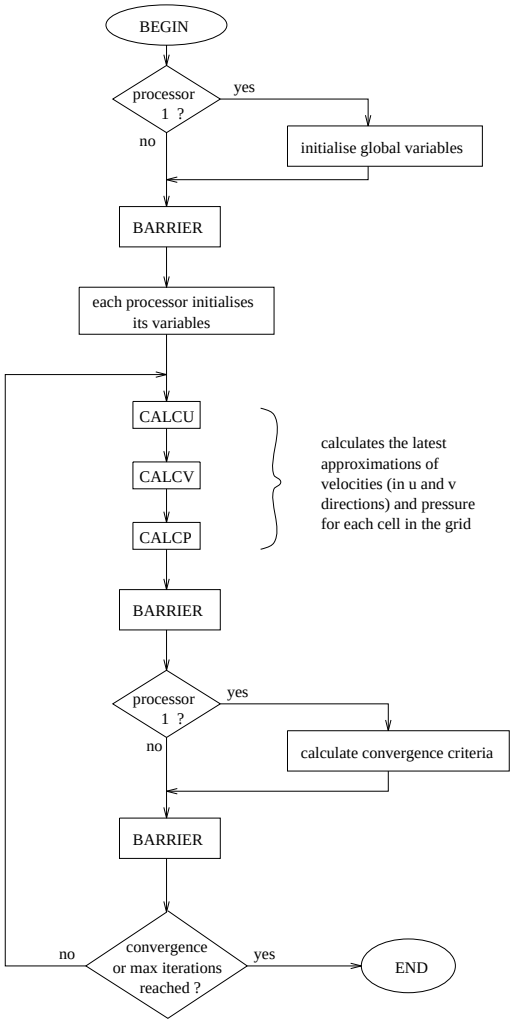


Figure 3.10: Structure of the CFD application

3.3.3 FFT - Fast Fourier Transform

The FFT kernel program is a complex, one-dimensional version of the "Six-Step" Fast Fourier Transform described by Bailey [10]. The FFT kernel is widely used in applications ranging from signal processing to climate modelling. Specific optimisations in the Splash-2 implementation include [145]:

1. Performing staggered blocked transposes that exploit cache-line reuse.
2. The roots of unity data structure is arranged and distributed for only local accesses during application of the roots of unity step.
3. A small set of roots of unity elements are replicated locally at each processor for computation of the 1D FFTs.
4. The matrix data structures are padded to reduce cache mapping conflicts.

3.3.4 FMM - Fast Multipole Method

FMM, like Barnes, simulates a system of bodies over a number of time-steps. However it simulates interactions in two dimensions using a different hierarchical N-body method called the Fast Multipole Method [115]. FMM was run for a two-cluster Plummer distribution with cost zones partitioning, and the precision set at $1.0e$-6, in line with the Splash-2 guidelines [144].

Analysis of running the application on a 32 node simulated system showed that queues of length 31 were occurring for access to elements of the `f_array`, which forms part of the `G_Memory` data structure [129]. This array is used in each iteration of FMM to enable all the processors to obtain the current overall dimensions of the $x - y$ grid containing the 2-dimensional model of the particles. The queueing occurs immediately after a barrier, when all the processors read all the elements of the `f_array`. This was the most significant case of read contention detected in FMM and it scales with the number of processors. It is independent of the number of particles, so simulations were run for a small problem size of 8192 particles and for three iterations to reduce simulation time.

3.3.5 GE - Gaussian Elimination

GE is a simple Gaussian elimination application, similar to that used by Bianchini and LeBlanc in their study of eager combining [16]. The application is based on a vector system algorithm described by Stone [128]. At the end of each iteration a single processor updates a row of the matrix which is designated as the pivot row. Following a barrier, all processors read this row and use it to update the set of rows which they maintain. It is immediately clear that this will cause contention since the data lines holding the pivot row will all have the same owner, and are also likely to have the same home node. The responsibility for updating the rows is re-allocated each iteration, with each processor handling approximately the same number of rows (in contiguous data lines) of the remaining matrix data.

3.3.6 Ocean

The Ocean program simulates large-scale ocean movements based on eddy and boundary currents. It uses a Red-Black Gauss-Seidel Multigrid technique to solve the elliptic equations associated with the problem, and partitions the grids into squarish subgrids rather than strips of columns to improve the communication to computation ratio. There are two versions of Ocean in Splash-2: using contiguous and non-contiguous allocations of partitions.

1. **Contiguous partition allocation**:

   This version implements the grids to be operated on as 3-dimensional arrays. The first dimension specifies the processor which owns the partition, and the second and third dimensions specify the x and y offset within a partition. This data structure allows partitions to be allocated contiguously and entirely in the local memory of processors that "own" them, thus enhancing the data locality.

2. **Non-contiguous partition allocation**:

   This version implements the grids to be operated on as two-dimensional arrays. This data structure prevents partitions from being allocated contiguously, but leads to a conceptually simple programming implementation.

The base problem size for a system with up to 64 processors is a 258×258 grid. The default values are used for other parameters (except the number of processors, which is varied up to 64). In addition, sample output files for the default parameters for each version of the code are contained in the file `correct.out` in each subdirectory. The Splash-2 distribution guidelines indicate that the less optimised versions of applications, where available, should only be used in addition to their optimised counterpart [122]. It can be assumed, therefore, that other work which only refers to "Ocean" will be using Ocean-Contig.

3.3.7 Water-Nsq

This N-body molecular dynamics application evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a user-specified number of time-steps, with the aim of reaching a steady state. The code in Splash-2 is an improvement over the Water code in the original Splash suite, but is mostly the same. Therefore the best source of information about this application is the original Splash report [116]. The main change found in the Splash-2 version is an improvement to the locking strategy which protects the updates to the water accelerations (in `interf.C`): a process updates a local copy of the relevant particle accelerations, and then accumulates into the shared copy at the end [144].

3.4 Experimental Design

The experimental results in this thesis were generated by ALITE, an execution-driven simulator[2]. The simulator parameterises the cc-NUMA design and provides detailed reporting on

[2]Please see Appendix B for more information about simulation techniques and the ALITE simulator.

the run-time characteristics of the benchmark applications, both at summary and node-by-node level. Using a simulator allowed for a detailed evaluation of the effects of changing the network bandwidth characteristics, and of altering the cache coherence policy to tackle node controller contention.

The node specification is summarised in Table 3.2. The CPU is a generic reduced instruction set computer (RISC), using a subset of the DEC Alpha instructions [15]. The first level caches are write-through, direct-mapped, and 8 Kbytes in size. Instruction accesses are assumed to be dealt with by a separate, perfect memory system. Second level caches (SLC) are write-back, direct-mapped, and 4 Mbytes in size. The line size is 64 bytes throughout. The simulator times all events in terms of clock cycles; for example Table 3.3 shows the latencies for the most significant node actions. In order to relate these "clock tick" timings to the real world, one only needs to specify a clock speed. For example, three network bandwidths are provided by the simulator (fast, medium, and slow); given a 100 MHz clock these bandwidths equate to 160 Mbytes/sec, 16 Mbytes/sec and 1.6 Mbytes/sec respectively. Unless otherwise stated, the network bandwidth is the fast bandwidth. Messages consist of a header (containing a message type, and the identities of the source, destination, requester and home nodes) and optionally a 64 byte data line payload.

The simulator models in detail the data miss and node controller actions. For example, consider the CPU on a client node making a read request for data which is not present in its FLC or SLC. The processor acquires the SLC bus, does a lookup of the tag in the SLC, acquires the MEM bus, and then instructs the node controller to initiate the transmission of a `read-request` message to the home node. While this is taking place the resources at the node (buses, SLC, node controller) are occupied, and therefore other requests which have been sent to this node will be denied service for some period of time.

The detailed level of simulation also captures the node controller actions at the home node. Consider again the `read-request` message which has been sent to the home node. When a home node services a `read-request`, the node controller performs a lookup in DRAM to determine the state of the line. Assuming that the state is Home-Exclusive, the SLC bus is acquired and a lookup in the SLC is done. If the data is not present, the SLC bus is released, the data is read from DRAM, directory information is updated in DRAM, the MEM bus is released, and the reply message is dispatched. An example of this is given in Figure 3.11, where the timings represent the relative length of each operation. While the request is being handled, other requests may arrive from remote nodes which cannot be serviced until the node controller has finished this transaction. In addition, the home node's CPU is prevented from accessing the SLC for part of the transaction (*i.e.* when SLC bus is in use by the node controller).

CPU	CPI	1.0
	Instruction set	generic RISC; subset of DEC Alpha
Instruction cache	All instruction accesses are assumed to be first level cache hits	
First level data cache (FLC)	Capacity	8 Kbytes
	Line size	64 bytes
	Direct mapped, write-through, on-chip.	
Transaction look-aside buffer (TLB)	Capacity	64 entries
	Fully associative	
Second-level cache (SLC)	Capacity	4 Mbytes
	Line size	64 bytes
	Direct mapped, write-back, off-chip.	
Memory (dynamic random access memory - DRAM)	Capacity	Large enough to hold all the data required by an application. Given that P=number of nodes, each node has $\frac{1}{P}$ share of the total DRAM.
	Data line size	64 bytes
	Page size	8 Kbytes
Node controller	Non-pipelined	
	Service time and occupancy	See Table 3.3 and Appendix B.2.
Interconnection network	Topology	Full crossbar
	Incoming message queues	Infinite for Chapters 3 & 4. Limited to 8 for incoming **read-request** messages in Chapters 5, 6, & 7.
Cache coherence protocol	Invalidation-based, sequentially-consistent cc-NUMA, home nodes assigned to first CPU to reference each page (*i.e.* "first-touch-after--initialisation"). Distributed directory, using singly-linked sharing list: based on the Stanford distributed-directory Protocol, described by Thapar and Delagi [134]	

Table 3.2: Details of the simulated architecture

Operation	Time (cycles)
Acquire SLC bus	2
Release SLC bus	1
SLC lookup	6
SLC line access	18
Acquire MEM bus	3
Release MEM bus	2
DRAM lookup	20
DRAM line access	24
Initiate message send	5

Table 3.3: Latencies of the most important node actions

The local memory (DRAM) at each node is part of an overall pool of shared-memory which is large enough to hold all of the data required for each application: the memory size at each node is an input parameter to the simulator. The pool of memory is divided evenly across the nodes, so in a 64 node system each node will have $\frac{1}{64}$th of the overall DRAM. However the memory at an individual node may not be sufficient to hold all of the pages which the

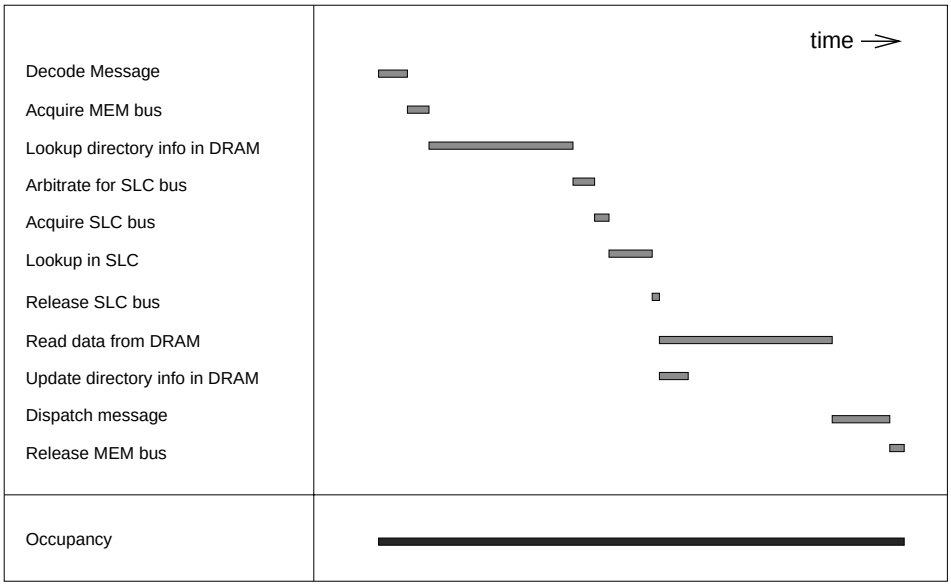


Figure 3.11: Occupancy of an example client read request at the home node

first-touch placement policy would like to allocate to that node: in this overflow case the pages will be allocated to the next node which has space available in its DRAM.

The ALITE implementation of the coherence operations handled by the node controller, and the resulting state transitions, have been cross-validated with an analytical model [12]. It should be noted that the node controller state machine includes many intermediate states, which are derived from the directory and SLC states: these are documented in detail in Appendix C of this thesis.

The pointers associated with the second-level cache lines are stored in the network controller so that pointer maintenance and traversal can be performed without generating internal bus traffic. This is a common way of implementing distributed directory protocols, because the performance benefits outweigh the additional storage requirements in DRAM, and it avoids complicating the SLC. Most SCI-based architectures follow this approach [120].

In the experimental work presented in this chapter and Chapter 4, the buffers at the node controllers are assumed to be infinite in size. The later experiments use bounded-length queues for incoming `read-request` messages. The default network bandwidth is the fast network (equivalent to 160 Mbytes/sec with a 100MHz clock speed). In addition to this fast bandwidth, the medium and slow bandwidth settings are used in this chapter to investigate the interaction between the network bandwidth, the resulting transaction latency, and the buildup of queues in the message buffers which can lead to node controller contention.

3.5 Bandwidth, Occupancy, and Contention

In order to understand node controller contention, and how it is affected by varying the bandwidth characteristics of the network, the eight applications were run for the three network bandwidths described in Section 3.4. Figure 3.12 shows the performance results in terms of relative speedup. Relative speedup is the ratio of the execution time for the fastest algorithm running on one processor to the execution time for $\mathcal{P}$ processors. It is not surprising that using a slow bandwidth results in the worst performance, given that any accesses to remote data will only travel at 1.2 Mbytes per second. However even with the fastest network some of the applications show a disappointing speedup with 64 processing nodes, with CFD and GE showing speedups of less than 30, and the best speedup (of 55.3 for Water-Nsq) is still well short of the "ideal speedup" of 64.

The breakdown of the execution times shown in Figure 3.14 helps to explain the performance problem. Taking FFT as an example, the faster the network, the smaller the percentage of overall execution time that is spent by the CPUs waiting for load misses. However even with the fastest network, 35.6% of the execution time is spent by CPUs stalled waiting for read data. In addition, the percentage of execution time spent waiting for store misses increases as the network gets faster. Of course the overall execution time improves with faster networks, but there is a performance bottleneck occurring for remote data accesses.

The queueing delay results presented in Figure 3.15 indicate that there is a problem with the distribution of queueing times across the nodes. The maximum and minimum mean delays are, respectively, the longest and shortest mean incoming message queue delay experienced by individual nodes. The overall value is the mean incoming queue delay taken across all the nodes. Once again taking FFT as an example it can be seen that, as the network bandwidth increases, all three mean values increase. However the minimum mean queueing time for individual nodes stays low, while the maximum is very high for the fast network (note the change of vertical scale). Figure 3.13 shows that the maximum individual node mean queueing delay is affected by the number of processing nodes for all three network bandwidths. However with the fast network the delay rises noticeably for all applications as the number of nodes increases, with FFT and GE showing particular "hot spots". This indicates that there are one or more nodes which have a high level of queueing when the network bandwidth is increased.

3.5.1 The Causes of Queueing

A study by Chandra *et al.* investigated where the time is spent in message-passing and shared-memory implementations of a number of algorithms [22]. They included a shared-memory

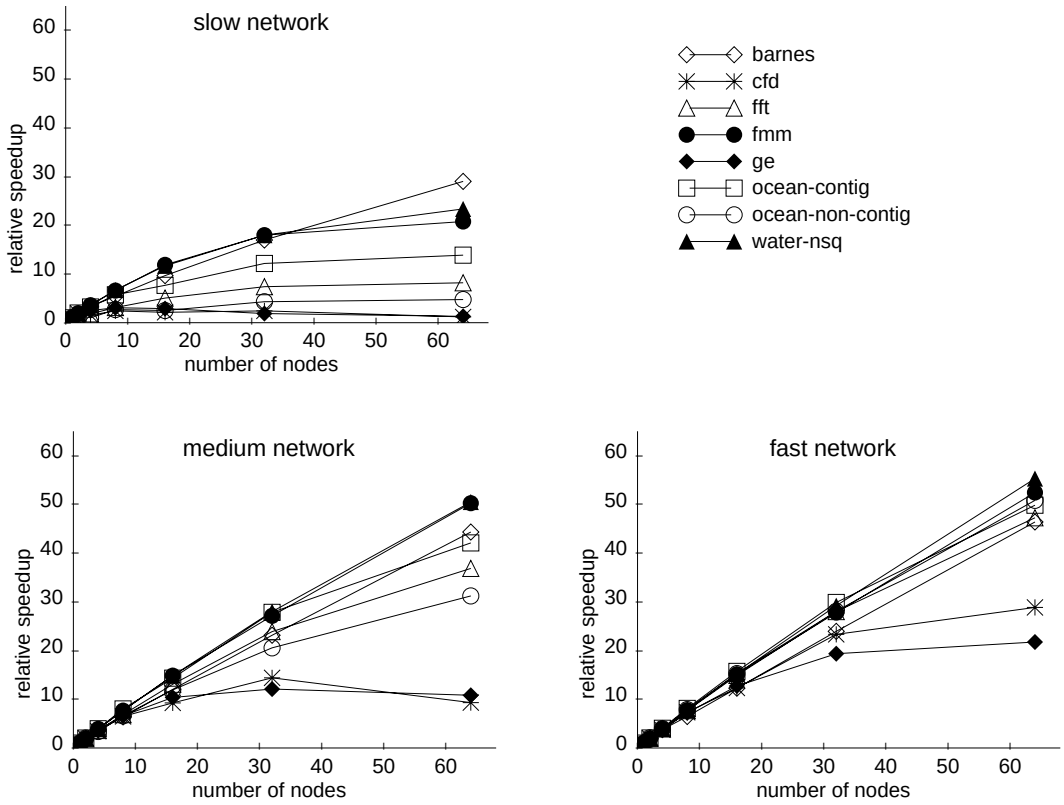


Figure 3.12: Performance speedup with different network bandwidths

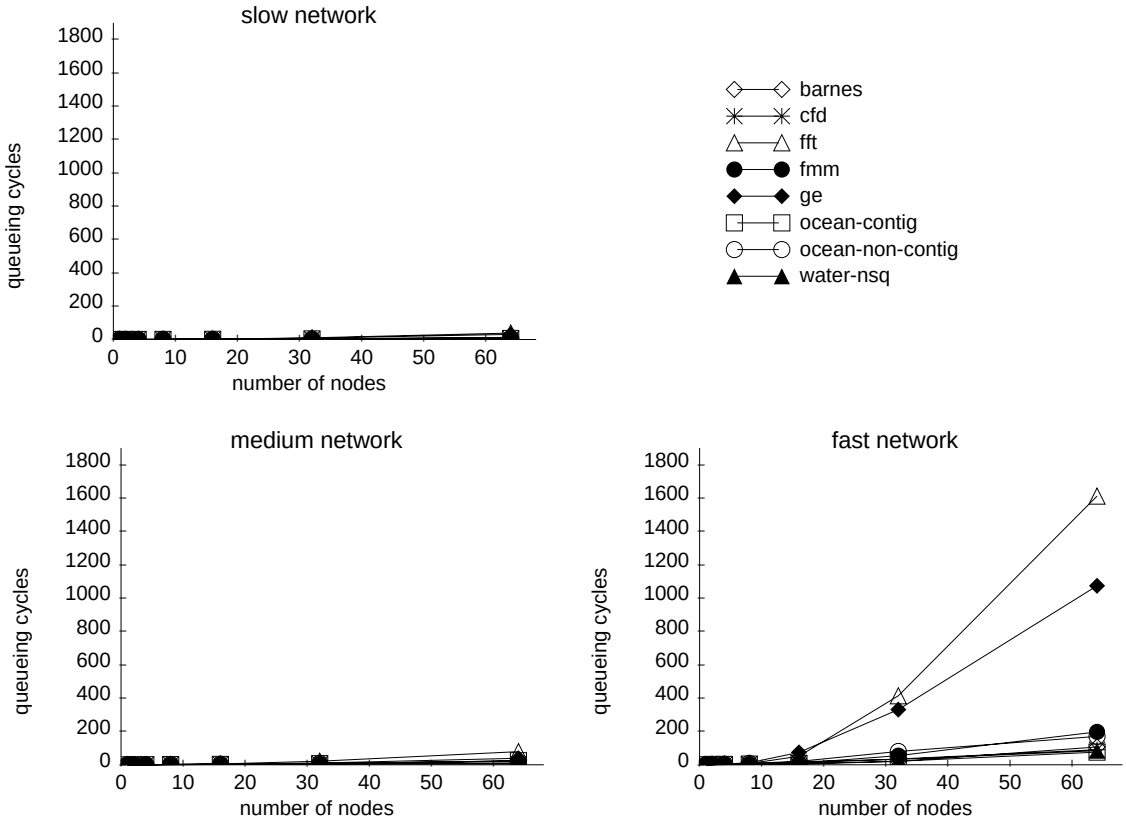


Figure 3.13: Maximum individual node mean queueing cycles with different network bandwidths

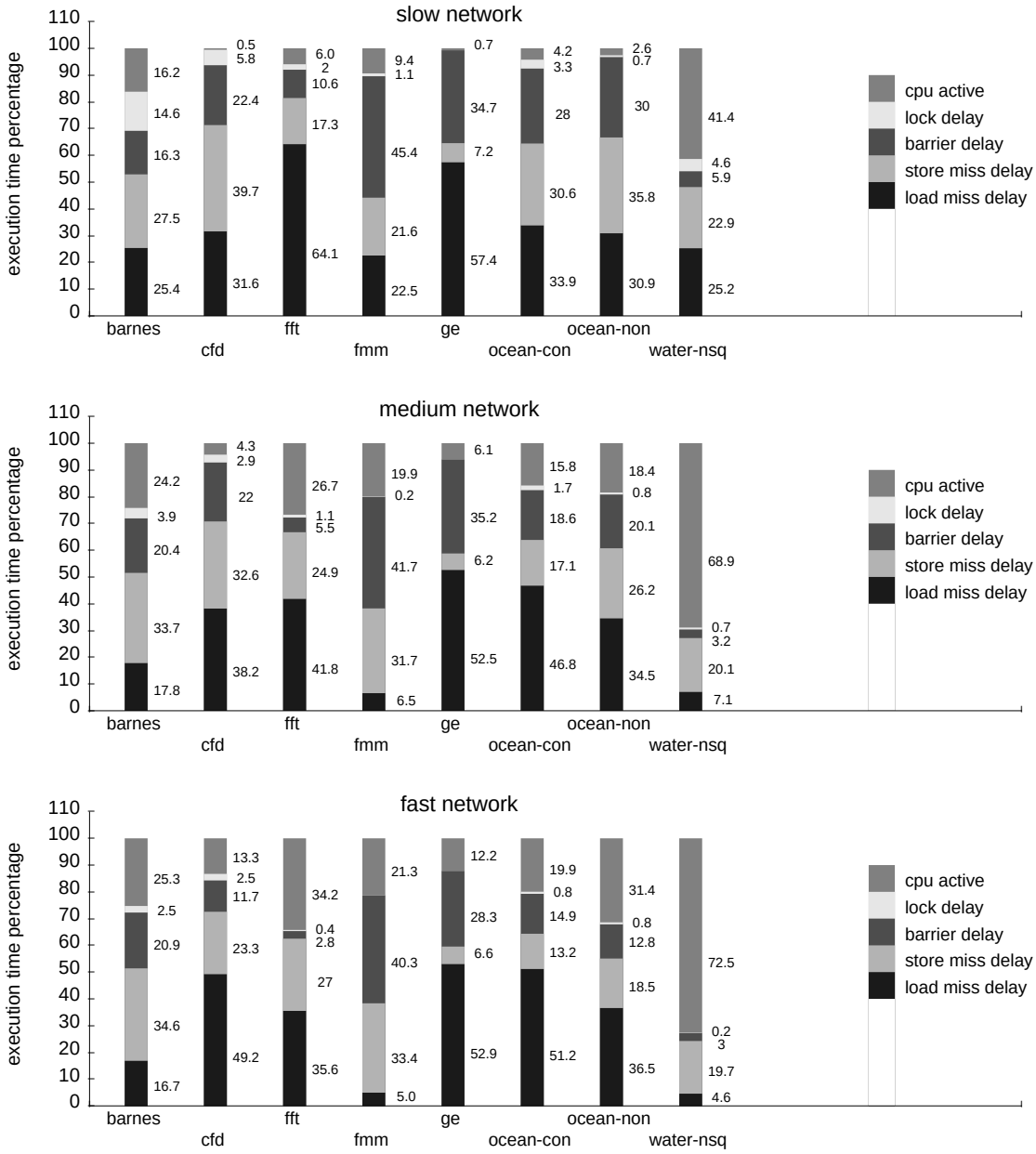


Figure 3.14: Execution time profiles with different network bandwidths (64 nodes)

version of the Gaussian Elimination algorithm which, like the one used in this thesis, has the problem of disseminating the pivot row to all the nodes. For this application the study found that the delays were due to directory contention, *i.e.* accessing the directory data held at the home node. Their simulations were for 32 processing nodes using the Wisconsin Wind Tunnel [109], and they noted that the queueing delays observed for GE would become untenable for larger systems because even more nodes would have to queue up for read access to the data lines of the current pivot row.

There are two factors at work in the buildup of an incoming message queue. The first is the rate at which messages arrive, and the second is the rate at which the messages are

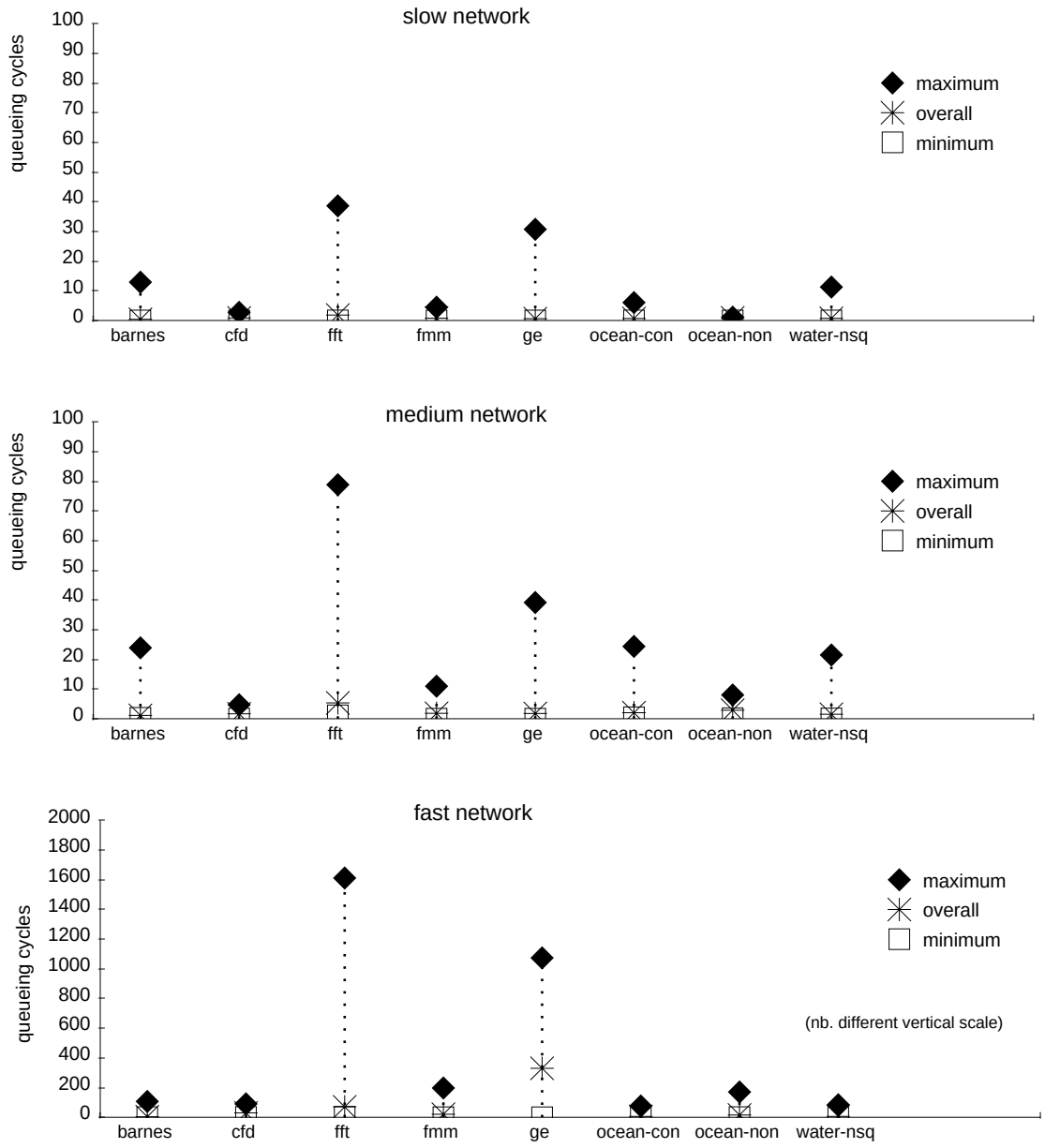


Figure 3.15: Mean queueing cycles for incoming messages (64 nodes)

serviced. The latter depends on the occupancy of the node controller, *i.e.* the time for which the node controller is tied up with one action and cannot perform another [57]. It is possible to reduce the occupancy for node controllers by introducing the ability to overlap the processing of requests, *e.g.* by pipelining or by splitting the processing of local CPU misses and remote requests (*e.g.* the Sun S3.mp). However these techniques reduce rather than avoid the occupancy, and occupancy will still be a problem when there is contention for a node controller. Contention can occur for either homes or owners. Directory traffic for a large set of lines, whose ownership is dispersed, may be concentrated in a single home node due to home allocation. Ownership of many data lines with different homes may become

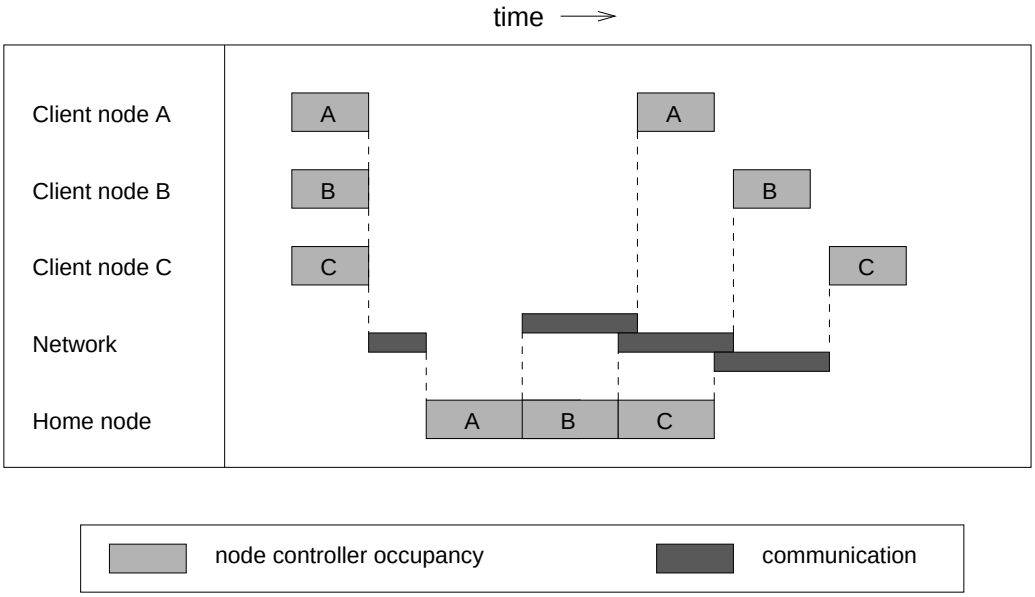


Figure 3.16: Overall transaction latency for an example of three simultaneous requests arriving at a home node

concentrated in a single cache because of the application's write behaviour.

The severity of node controller contention is both application and architecture dependent. Some contention is inevitable and will result in the latency of transactions being elongated. The key problem is that queue lengths at node controllers, and hence contention, are non-uniformly distributed around the system. Some node controllers may have short queues, whereas others have long, or full queues. The result is that many requests are waiting in buffers for long periods of time, and processors are stalled unnecessarily. For example see Figure 3.16, where the simultaneous arrival of three requests at the home node results in a queue forming in the incoming message buffer, and the serialisation of the requests increases the overall latency before the reply is received at Client C.

The communications access pattern is non-uniform primarily because of the way homes and ownership are allocated. It is the non-uniform distribution of requests made by the application which causes the variation in contention over the execution time of the program. The characteristics of the architecture determine how effectively the non-uniform distribution of requests can be resolved.

3.5.2 Widely-Shared Data

In Section 2.2.3 of this thesis, it was noted that applications which use widely-shared data often send many simultaneous `read-request` messages to the same home node. GE is a benchmark application which is already known to widely-share the current pivot row data

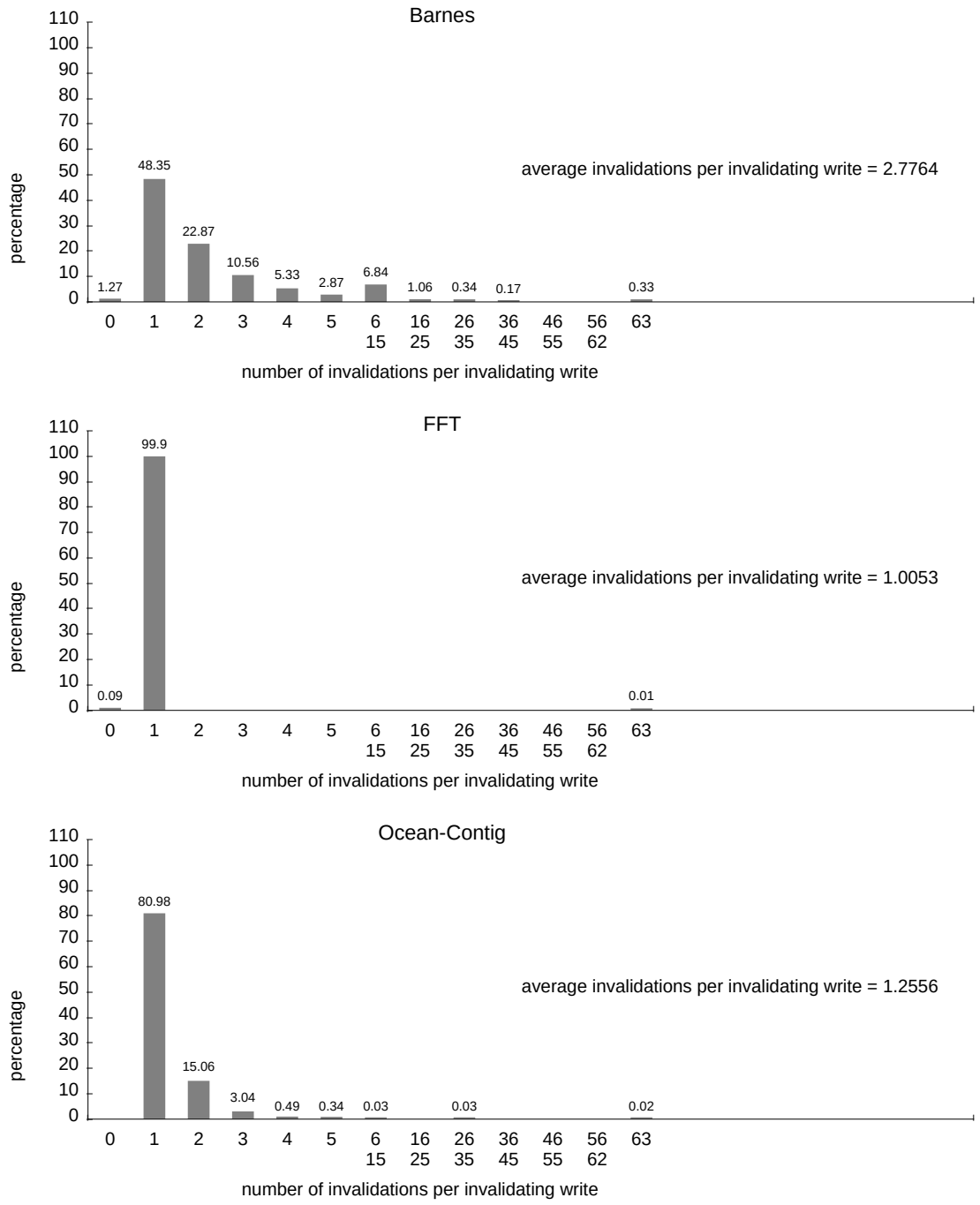


Figure 3.17: Invalidation profiles for Barnes, FFT, and Ocean-Contig (data from [121])

in each iteration. Kaxiras *et al.* have shown that there is modest mean degree of sharing in GE of 2.75 sharers per data item in a 128 node system, but about half of all the reads issued by the application are for widely-shared data. The very few but large read-runs for widely-shared data represent around half the overall number of reads [69].

As the number of nodes in the system is increased, the bottleneck caused by widely-shared data becomes worse: requests for the same data line made simultaneously by more nodes

will increase the contention at the home node. This will lead to performance degradation, as is seen for GE in Figure 3.12 for the fast network, where the speedup tails off as the number of nodes increases, and the queueing imbalance between nodes becomes very large (see Figure 3.15). This bottleneck leads to performance problems which are not only connected with the number of nodes in the system, but also with the rate at which a node controller can deal with incoming messages, and with the network bandwidth. If the network is unable to deal with a rush of messages all with the same destination node then there are likely to be hot-spot problems within the network as well as the contention at the home node [105].

The Stanford Splash-2 benchmark suite also contains some applications with widely-shared data: these include Barnes, FFT, and Ocean-Contig [144]. The sharing can be identified from the invalidation patterns provided in the Splash-2 results database [121]. Looking at Figure 3.17 it can be seen that all three applications have a "tail" in the number of invalidations needed before a node has obtained exclusive access to the data line and the write can proceed. Although the average number of invalidations in an invalidating write transaction is low, there are data structures where $(P-1)$ invalidations are needed before the write can proceed, *i.e.* copies of the same data line are held in SLC at all the nodes on the system. The widely-shared data can lead to a buildup of incoming messages which have to queue at the home node controller as they wait to access the same directory information.

It is always possible to tune a program to avoid high degrees of sharing, but this tuning may be time consuming, and tends to increase the complexity of a program [24]. However, by providing efficient hardware support for programs that use widely-shared data, Johnson believes that significantly less time will be needed to get good performance out of shared-memory multiprocessors with thousands of processing nodes [61]. Given the performance bottleneck which can result from widely-shared data, a scalable distributed shared-memory system needs to provide a mechanism to support this sharing pattern if it is to avoid seemingly unpredictable performance problems.

3.6 An Analysis of Widely-Shared Data Accesses

The experimental runs in the preceding section demonstrated that there is a problem with node controller contention, in particular where there are widely-shared data structures. This section presents an analysis of the node controller behaviour, with the aim of gaining insight into the effects of simultaneous `read-request` messages for widely-shared data arriving at the home node. The analysis is consistent with the work done by Holt *et al.* on node occupancy [57], and the analysis of contention by Dwork *et al.* [33].

The main latency elements in the access of widely-shared data are the network traversal time, the queueing delay and occupancy (both at the client itself and at the home node), and the interval between `read-request` messages arriving at the home node. Consider the following example, where there are P processors, one at each node, each executing the same fragment of pseudo-code:

```
barrier()
for i := 1 to N do
   read A[i]
end
```

i.e. each processor is going to reference the entire array A. This example captures the data access pattern of the GE application when all processors require read access to the pivot row.

It is assumed that:

- Array A is held in one page of DRAM with home node processor P_{home}, and is n bytes in length.
- Each data item is i bytes long, where $i = n/N$.
- A data line is d bytes long, and contains q consecutive data items from A.
- $A[1]$ is at the start of a data line.
- The yth data line in A will contain $A[((y-1) \times q + 1) : (y \times q)]$.
- There are no (valid) cached copies of data from A after the barrier, except at the home node.
- The home node is initially in state Home-Exclusive for each data line in A: there is a more up-to-date copy of the data line in the home node's SLC. After processing the `read-request` from the first client, a data line's state will be updated to Home-Shared and the DRAM will be up-to-date.

3.6.1 Node Controller Occupancy

Using the approach of Holt *et al.* [57], the node controller occupancy is characterised as follows:

- When the node controller is simply generating a request into the network or receiving a reply from the network, it incurs a fixed occupancy t_o.

- When the node controller is at the home node for a request, it incurs a fixed occupancy of $2 \times t_o$, because it has to retrieve data from memory and/or manipulate coherence state information.
- If the state lookup at the home reveals that the requested line is dirty in the home node's cache, the node controller incurs an extra fixed occupancy of C.

3.6.2 Network

Using the approach of Holt *et al.* [57], the time for a message to traverse the network can be characterised as follows:

- Startup time t_s is the time required to handle a message at the sending processor. It includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node controller and the router. This delay is only incurred once for each message.
- Per-hop time t_{hop} is the time taken by the header of a message to travel between two directly-connected processors in the network.
- Per-byte transfer time t_b : if the channel bandwidth is r bytes per second, then each byte takes time $t_b = 1/r$ to traverse the link.

Many factors influence the communication latency of a network, such as the topology of the network and the switching techniques (see Section 2.1.3). For the purpose of analysing the effects of contention, the simplifying assumption is made here that the network is a completely non-blocking fully-switched crossbar. This results in each message only having one "hop", and the further assumption is made that a given message will always take the same time to travel between any two processors regardless of locality and network load. This is the same simplification as is made in the ALITE simulator.

Given these assumptions, the latency for a `read-request` message to traverse the network $t_{read} = t_s + t_{hop}$, and the latency of a `take-shared` message $t_{take} = t_s + t_{hop} + d \times t_b$.

3.6.3 Hit to Miss Interval

The CPU will sooner or later (later given that the processor first has to process the data line containing items $A[1 : q]$ which has just arrived, sooner if it has a prefetching mechanism) request the node controller to obtain the next data line containing items from A, *i.e.* the data

line containing $A[q+1:2q]$. t_x represents the time that passes between the CPU receiving the data for one cache miss, and when it requests the next data line. This value can be negative if the system uses prefetching.

3.6.4 Contention

After the barrier, the $(P-1)$ client processors will each send a `read-request` message to the home node P_{home} for the data line containing $A[1]$. Given the network assumptions, this will cause a queue of $(P-1)$ messages in the input queue at the node controller for P_{home}. Only one message can be serviced at a time, in time $(2 \times t_o + C)$ for the first message, and time $2 \times t_o$ for the remaining $(P-2)$ messages. Depending on its original position in the input queue of messages, it will take between $(2 \times t_o + C)$ and $((P-1) \times 2 \times t_o + C)$ for a request to be serviced by the node controller at P_{home}. Dwork *et al.* have noted this serialisation effect, and the increase in latency which it causes [33].

The processor whose request message is initially at the head of the incoming message queue at P_{home} will receive the data line line containing $A[1]$ in time $t_o + t_{read} + (2 \times t_o + C) + t_{take} + t_o$, *i.e.* request occupancy + `read-request` message latency + home node controller occupancy + `take-shared` message latency + reply occupancy.

The next `read-request` message from that client node will be sent to the same home node, P_{home}, and will have to join the input queue waiting for service from the node controller. If the home node controller has not finished processing the remaining $(P-2)$ `read-request` messages for the first data line of A, then the request for the next data line will have to wait. Meanwhile, further requests for this new data line will arrive from the other processors. Given this pattern of requests, the measure of node controller contention at the home node, $Contention_{home}$, is given by how long requests from a particular client node will have to wait for service at the home node.

$$Contention_{home} = [(P-2) \times 2t_o] - [(t_s + t_{hop} + d \times t_b) + t_o + t_x + t_o + (t_s + t_{hop})]$$

If $Contention_{home} \leq 0$ then there is no incoming message contention at the home node for access to the widely-shared data structure. If $Contention_{home} > 0$ then there is contention. If the occupancy time taken to service the remaining $(P-2)$ read requests for a data line is any longer than the mean time between `read-request` messages from the first client, then the queue will build up to a maximum of $(P-2)$ read messages (or even worse if prefetching is allowed) causing processors to stall for longer as they wait for the arrival of their `take-shared` messages.

The equation for $Contention_{home}$ gives a straightforward way of seeing the effect of varying the characteristics of the cc-NUMA system. Reducing the node controller occupancy t_o will help delay the point where the bottleneck occurs. However increasing the number of nodes P in the system will make the queueing bottleneck more likely, as will decreasing t_x using pre-fetching of data lines or faster CPUs. It can also be seen that an increase in network bandwidth will reduce the per-byte transfer time t_b, which will in turn reduce the amount of time before the next request arrives from a given client. An increase in bandwidth will also reduce the per-hop time t_{hop}, *i.e.* the time taken by the message header to travel between two directly connected nodes. Therefore improving the network bandwidth, or the arrival rate of successive data line `read-request` messages, or increasing the number of nodes in the system will exacerbate the queueing at the input buffer of the home node's controller.

It should also be noted that the local CPU at the home node is loading the data items of A, which may add to the contention for the SLC bus. In addition, given that the home node's CPU will obtain the data for A sooner than the remote nodes, it is likely to "race ahead" to the next synchronisation point which will increase the barrier delay as the home node waits for the other nodes to "catch up". In addition, client node controllers will be idle while they are waiting for each `take-shared` message, given that their local CPUs have stalled waiting for that data.

The analysis is unaffected by the choice of mechanism for holding directory data about the sharers of a data line (*e.g.* as a bit vector or as a distributed list) because in all cases it is necessary to access the state information and add a new sharer, all of which is done at the home node. The simplifying assumption of a full-crossbar network does affect the analysis. However the t_{hop} values in the $Contention_{home}$ equation can be multiplied by a suitable value to represent the mean number of hops which are taken by a message in another type of network.

This analysis has shown that the usual mechanisms for improving system performance, such as increasing the network bandwidth or the number of processors or the local CPU speed, will result in accesses to a widely-shared data structure becoming a performance bottleneck because of queueing for service at the home node. The client CPUs will spend a considerable amount of time stalled waiting for data, and using prefetching of data lines will only increase the bottleneck. In addition, the client node controllers will be mostly idle during the access to the widely-shared data structure.

3.7 Conclusions

This chapter has presented the representative cc-NUMA architecture used throughout this thesis, and has demonstrated the performance problems which arise from contention for node controllers. A specific problem was identified with widely-shared data, where many `read-request` messages are sent simultaneously to the home node. Although only a small part of the shared-memory may be widely-shared, the widely-shared data can give rise to significant performance problems.

The next four chapters investigate a technique for alleviating node controller contention, which uses combining to reduce the number of `read-request` messages arriving at a home node. The approach builds on the knowledge gained in this chapter, including the observation that client node controllers are often idle during the transactions to access widely-shared data.

Chapter 4

The Basic Proxy Scheme

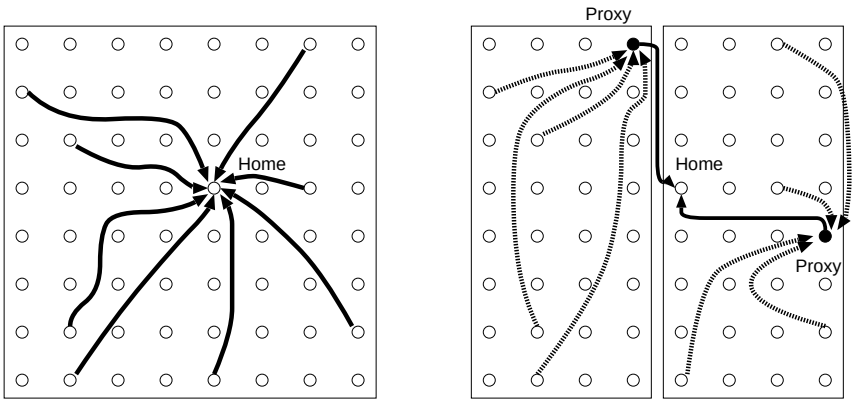
It has been shown in Chapter 3 that contention for node controllers can lead to performance degradation. These problems are particularly severe for the class of shared-memory applications where a large data structure is read simultaneously by many processors, and whose data lines have a common home node. This chapter presents a protocol modification which aims to reduce the node controller contention at the home node which results from access to this widely-shared data. The protocol modification was first presented in [13], but this chapter includes more detailed experimental results, investigates the effects of proxying for five more benchmark applications, and uses an improved representation of the pending chain of proxies.

4.1 Proxies: an overview

In the analysis of node controller contention presented in Section 3.6 it was noted that the client node controllers are often idle, while a long queue of incoming `read-request` messages builds up at the home node. Given that this contention problem arises because many nodes are accessing the same data lines, it would be better if some of the idle clients obtained the data and then provided that data to the rest of the clients. This would shift some of the `read-request` messages from the home node's input queue to the input queues of the under-utilised client node controllers.

4.1.1 Proxies

The high concentration of read requests at home nodes can be reduced by distributing the messages to other node controllers, using them as intermediaries. These are referred to as *proxies*. In the proxy scheme, a processor issuing a read request for remote data sends the request message to another node, which is known to act as a proxy for that data line, rather



(a) Without proxies (b) With two proxy clusters

Figure 4.1: Contention is reduced by routing loads via a proxy

than going directly to the data line's home node. As an example, Figure 4.1 shows how more than one proxy could service requests for a given data line, each handling requests from a different subset of the processing nodes. If the proxy has the data, it is supplied to the client; otherwise a read request is made by the proxy node to the home node. The home node either supplies the data to the proxy, or forwards the request to the owner of the line, which then replies to the proxy. The sequence of messages is illustrated in Figure 4.2. The proxy scheme also combines read requests. If multiple requests for the same data line are sent to the same proxy, only the first requires a request to be made to the home. When the proxy receives the data, it is added to its second level cache (SLC) and sent to all the waiting clients. This basic form of proxies was first described by Bennett *et al.* [13], and builds on the idea of eager combining suggested by Bianchini and LeBlanc [16].

It should be noted that every node in the system can act as a proxy, *i.e.* there are no separate specialised proxy nodes. The use of proxies is expected to be sporadic, depending on the nature of individual applications, and it makes sense to use existing nodes rather than build in the hardware overheads of specialised nodes.

When a write occurs, the cache copies used for proxying are treated in the same way as any other cache copy: they appear on the sharing list, and so are invalidated automatically on a write.

4.1.2 Potential Benefits and Costs of Proxying

The proxy scheme will shift the balance of processing for proxied data, and so it should have a number of effects. Among the benefits one would expect are:

Incoming message queue length reduction at home node: queueing for directory in-

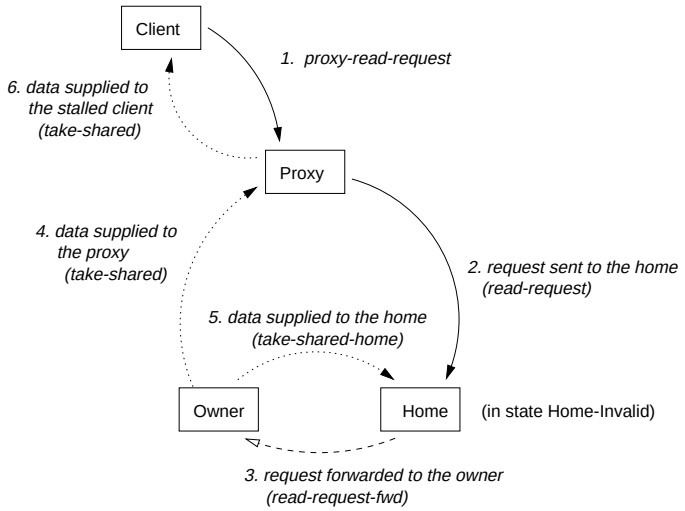


Figure 4.2: A simple proxy read request

formation served by a single node controller, due to unfortunate home node allocation, should be reduced.

Combining of read requests: when the proxy has, or has already requested, a copy of the data, further read requests are serviced at the proxy node.

The potential costs are:

Indirection: every load (for addresses subject to proxying) must now go via a proxy, whereas with the original protocol no indirection would be involved. For example, an original two message sequence of a `read-request` message to the home node followed by a `take-shared` message back to the client could become a sequence of four messages when proxies are used. The `proxy-read-request` from the client to the proxy node could be followed by a `read-request` from the proxy to the home when the proxy does not have a copy of the data. The home node would then send a `take-shared` message back to the proxy, and the proxy would send a `take-shared` message on to the client node.

Cache pollution: allocating proxy data in the SLC may displace another line, and lead to a later cache miss. Extra unhooking messages are required, which will increase the latency of satisfying the original remote read request.

Longer sharing lists: the distributed sharing list for a data line will have one extra entry for each proxy currently holding a copy of the data where the proxy node does not need the data for its local processing. This will increase the number of invalidation messages needed when a write is pending.

These expected benefits and costs are considered further in Section 4.4 which analyses the results of running the eight benchmark applications with the proxy protocol.

4.2 Design Issues

There are a number of issues that need to be addressed when designing a proxy scheme. They are how to decide which data structures should be subjected to proxying, how to choose the proxy node(s), how combining of read requests can be achieved, and when to supply proxy nodes with modified data. These issues are discussed in this section.

4.2.1 Selective Use of Proxying

To avoid using proxies on data which is not widely-shared, the protocol will only use proxying for pages of the shared-memory which have been "marked" for proxying. If the proxy does not have a copy of the data, two more messages are required (labelled 1 and 4 in the example given in Figure 4.2) than is the case where proxying is not used. The advantages of proxies come when further read requests are made via the proxy. However, if pages of shared memory are marked for proxying when their data is not widely-shared, then this message overhead is likely to degrade performance rather than improve it.

The marking of pages would be achieved by using directives inserted by the programmer (or potentially by a compiler). This approach was used by Bianchini and LeBlanc with pages being marked as "hot" for eager combining [16]. The original version of the GLOW protocol extensions for widely-shared data also used program directives to mark data: however Kaxiras *et al.* have since recognised that there are a number of drawbacks to statically marking widely-shared data [69]. Firstly the user involvement in marking data compromises the ideal of shared-memory programming, *i.e.* that the user does not have to worry about how data structures are accessed. Secondly it may not always be possible to statically identify the widely-shared data. Finally, a mechanism would be needed to transfer the marking information to the hardware: such a mechanism is hard to implement if the system is to be constructed from commodity parts.

In this chapter it is assumed that a mechanism is available to transfer the marking information from the application program to the node controller hardware[1]. The choice of which data structures should be proxied can be made by using the application programmer's knowledge of the application, or by analysing memory usage traces to determine which data lines and

[1] Chapter 5 will introduce proxy strategies which do not require the data to be marked.

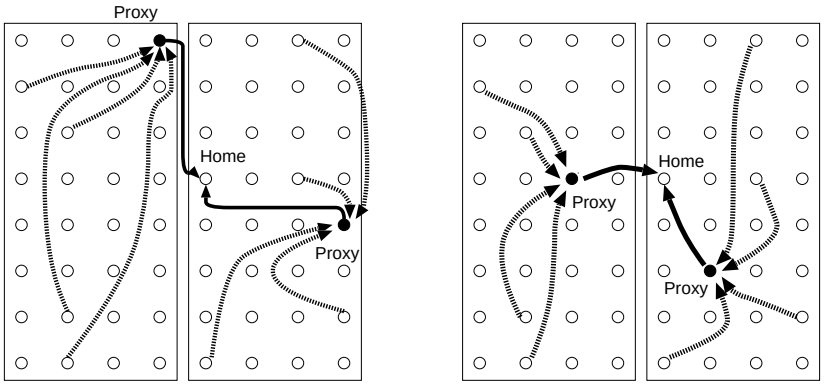
pages are widely-shared (*e.g.* using the Clarissa tool [129]). To evaluate the cost of proxying in circumstances when it is not beneficial, the results in Section 4.4 include applications for which *all* of the shared data is proxied.

4.2.2 Choosing the Proxy

The choice of proxy node can be random or, as shown in Figure 4.1, on the basis of locality. The following definitions are used to describe how a client node decides which node to use as a proxy:

- $\mathcal{P}$: the number of processing nodes.
- $\mathcal{H}(l)$: the home node of location l. This is determined by the operating system's page placement policy.
- $\mathcal{NPC}$: the number of proxy clusters, *i.e.* the number of clusters into which the nodes are partitioned for proxying. In the example shown in Figure 4.1, $\mathcal{NPC}$=2. The choice of $\mathcal{NPC}$ depends on the balance between the degree of combining and the length of the proxy pending chain. $\mathcal{NPC}$=1 will give the highest combining rate, because all proxy read requests for a particular data line will be directed to the one proxy node. As $\mathcal{NPC}$ increases, combining will reduce but the number of clients for each proxy will also be reduced, which will lead to shorter proxy pending chains.
- $\mathcal{PCS}(C)$: the set of nodes which are in the cluster containing client node C. In this work, $\mathcal{PCS}(C)$ is one of $\mathcal{NPC}$ disjoint clusters, each containing $\mathcal{P}/\mathcal{NPC}$ nodes, with the nodes grouped into clusters based on their node number. In a system where "geographic locality" affects the communication latency between nodes, it would be sensible to form each proxy cluster set from neighbouring nodes.
- $\mathcal{PN}(l, C)$ the proxy node chosen for a given client node C when reading location l. The proxy is chosen from the proxy cluster $\mathcal{PCS}(C)$. If the chosen node is the client or the home node (*i.e.* $\mathcal{PN}(l, C) = C$, or $\mathcal{PN}(l, C) = \mathcal{H}(l)$), then client C will send a read request directly to the home node $\mathcal{H}(l)$.

The choice of proxy node is, therefore, a two stage process. When the system is configured, the nodes are partitioned into $\mathcal{NPC}$ clusters. Then, whenever a client wants to issue a proxy read, it will use the function $\mathcal{PN}(l, C)$ to select one proxy node from $\mathcal{PCS}(C)$. This mapping ensures that requests for a given location are routed via a proxy (so that combining occurs), and that reads for successive data lines go to different proxies, as shown in Figure 4.3. This should reduce network contention and balance the message load across all the node controllers



(a) Load Line K (b) Load Line $K+1$

Figure 4.3: The proxy is different for successive data lines

in a way that is akin to Valiant's two-phase random routing [137]. Figure 4.4 shows an example of the split into proxy clusters when $\mathcal{P}$=10 and $\mathcal{NPC}$=4.

Given the partition into clusters, the actual choice of proxy node for a given data line tag and client node has been implemented as a round-robin of the available nodes in the client's proxy cluster set. For example, in Figure 4.4, if node 9 is the client, then a given tag value K could map to having node 10 as its proxy, and tag value $K+1$ would map to node 8. Although this round-robin approach is vulnerable to stride access patterns, extensive tests using the eight benchmark applications with pseudo-random and incremental offset round-robin functions failed to get better results than the simple round-robin. This is because the size of the proxy cluster set used in this work is relatively small (≤ 64), so the pseudo-random and offset functions make poor use of the available nodes.

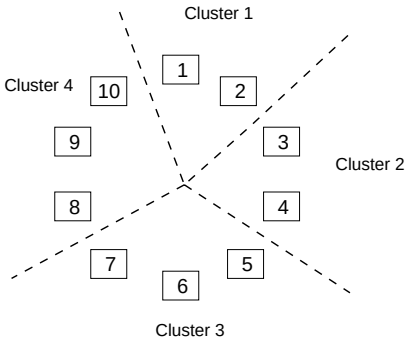


Figure 4.4: Example partition of nodes into proxy clusters, $\mathcal{P}$=10 and $\mathcal{NPC}$=4

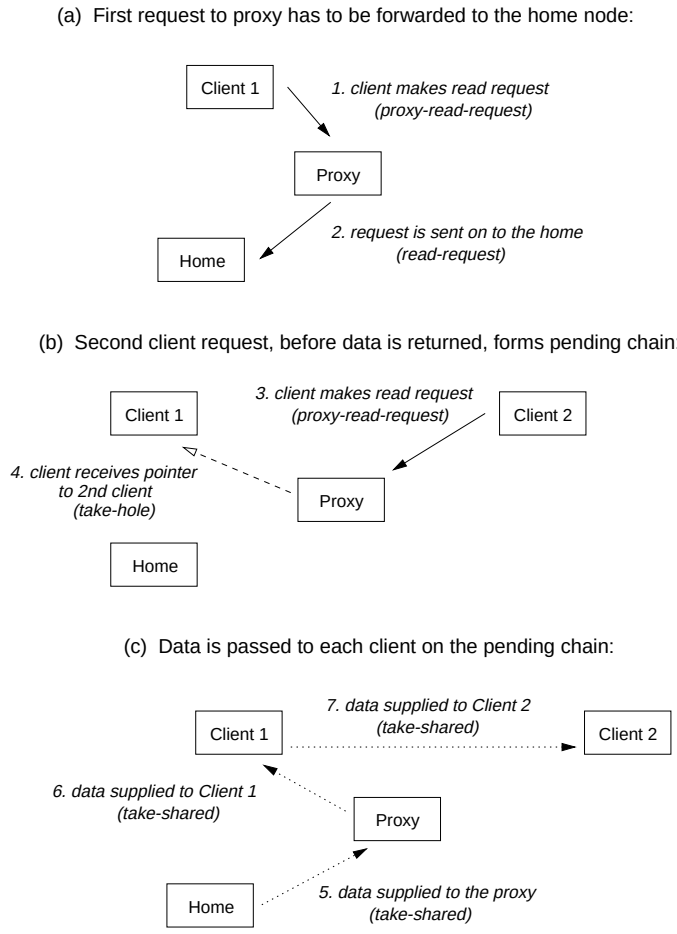


Figure 4.5: Combining of proxy requests

4.2.3 Combining

The proxy scheme combines read requests. If multiple requests for the same data line are sent to the same proxy, only the first request has to be forwarded on to the home. When the proxy receives the data, it is added to its SLC and sent on to the list of waiting clients. The list of waiting clients is held as a "pending chain", *i.e.* a distributed singly-linked list of the nodes which have requested a particular line of data. The process is illustrated in Figure 4.5. The implementation of the pending chain as a distributed list (rather than the proxy node keeping a local list of the pending clients) means that the proxy's node controller occupancy is kept to a minimum because it does not have to send a separate `take-shared` message to each of the waiting clients. The distributed list approach also has the advantage of being more scalable than a pending-client bit vector[2]. For a different network, *e.g.* a mesh of buses where broadcast facilities are available, one might choose a different implementation of the pending chain.

[2] Although the singly-linked list used in this work is not fully scalable, because of the time taken to traverse the list, other distributed implementations such as a tree are scalable.

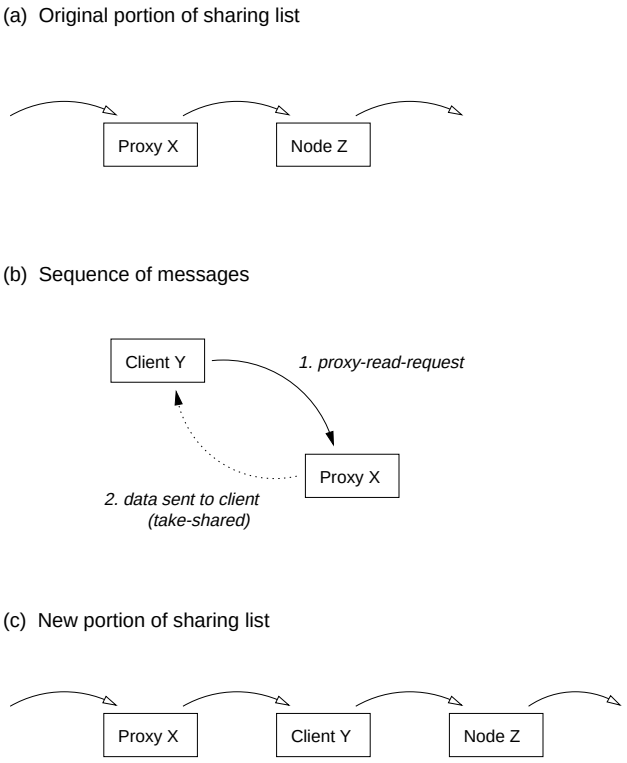


Figure 4.6: Read request when proxy has the data

When a client requests data, and the proxy already has a copy in its SLC, the request can be satisfied immediately by the proxy node. The proxy is able to add the new sharer to the sharing list, without having to exchange messages with the home node, by updating the pointers appropriately as illustrated in Figure 4.6. In this example Proxy X changes its SLC entry to point to Client Y, and the message sent to Client Y includes the pointer to Node Z (*i.e.* the next node on the sharing list).

The way that proxies allow read requests for data to be combined in controllers, away from the node suffering from contention, is reminiscent of the combining within the network used for the atomic read-modify-write operations proposed for the NYU Ultracomputer [45] and the IBM RP3 [104]. Combining of requests within the network is also used in COMA systems such as the Swedish Institute of Computer Science's (SICS) DDM [50], and it is also found in the SB-PRAM [37]. The GLOW extensions for widely-shared data use combining and are, like proxies, designed to be added to existing cache coherence protocols [70]. GLOW uses *agents* to intercept requests for widely-shared data at selected network switch nodes. In all these cases the combining opportunities are restricted to requests which go through a common switch on the network, so the combining often occurs close to the home node. This is in contrast to the proxy scheme which routes requests via specific proxy nodes, increasing the likelihood of early combining.

Eager combining, suggested by Bianchini and LeBlanc, uses intermediate nodes which act like proxies for "hot" pages, *i.e.* the programmer is expected to mark the "hot" data structures [16]. The choice of server node is based on the page address rather than data line address, so their scheme does not spread the load of messages around the system, unlike the more fine-grained approach of proxies. Bianchini and LeBlanc's scheme eagerly updates all proxies whenever a newly-updated value is read. In contrast, the basic proxy scheme allocates copies of data at proxies on demand. This approach was chosen to incur lower overheads because it only provides data when it is needed, reducing cache pollution at the proxies.

4.2.4 Caching the Proxy Data

In the combining scheme used for the NYU Ultracomputer [45] and the IBM RP3 [104], there is no retention of data at the combining node. In the proxy scheme, it was decided to explore the option of retaining a copy of proxied data at the proxy node. This should achieve a higher level of combining by allowing for later client read requests to be satisfied at the proxy. Combining requests for the same data lines at intermediate nodes, to improve the retrieval time for remote accesses, has been explored for hierarchical architectures, such as the SICS DDM [50]. The proxies approach is different because it does not use a fixed hierarchy: requests for copies of successive data lines are serviced by different proxies, and later requests (*i.e.* after an earlier proxied read transaction has completed) may be satisfied by data retained at the proxy.

As was noted in Section 4.1.2, the problems with holding a copy of proxy data at a node include cache pollution and a potential increase in the number of invalidations required before a write. In this chapter, the evaluation is done on a system where the proxy data is held in the proxy's SLC. The issue is explored further in Chapter 6 and Chapter 7, with schemes that do not cache the proxy data or hold it in a separate proxy buffer.

4.2.5 Adding Proxy Protocol Handlers

It is envisaged that the proxy protocol extensions would be implemented in software on a programmable node controller, with the original cache coherence protocol implemented in hardware. The way in which the node controller is implemented may be critical to performance of the whole multiprocessor, and so it is important to understand the performance tradeoffs between using customised hardware and/or a programmable protocol processor to implement the coherence protocol. Michael *et al.* have carried out a study of the performance and convenience tradeoffs between using hardwired or programmable node controllers [97]. Their

conclusions were that: (1) programmable node controllers have higher occupancy; (2) protocol errors can be easily fixed in programmable node controllers; and (3) having more than one protocol engine (*i.e.* the node controller can process more than one action at a time) improves performance.

The Wisconsin Typhoon [110] and Stanford Flash [81] are research examples of programmable network controllers which aim to combine the speed and concurrency of existing hardware mechanisms with the flexibility of software coherence. The Sequent NUMA-Q is a commercial system with a programmable node controller [92]. These all show the benefits of combining hardwired protocol handlers for common actions and programmable protocol handlers for correcting errors or adding complicated but less frequently used actions.

4.3 Modifications to the Protocol and Architecture

The proxy scheme requires a change to the hardware to allow a small amount of extra storage to be associated with the node controller, shown as the "Proxy Transit Cache" in Figure 4.7. Specifically the node controller needs to be able to identify which data lines have outstanding proxy transactions, and record the head and finish of each pending chain. This could be implemented using a small, fast associative store on the MEM bus, or as a reserved area within DRAM, or within the node controller. In the simulated system, the Proxy Transit Cache is assumed to be within the node controller.

The Proxy Transit Cache is similar in function to the Remote Access Cache (RAC) in Stanford's DASH [89]. The RAC is a 128 Kbyte direct-mapped cache which acts as a staging area to receive and buffer replies from remote clusters. RAC entries are allocated when a remote request is issued (by one of the four local processors) and persist until all inter-cluster transactions relating to the request have completed. If there is a conflict for a RAC entry, the later request is delayed and then retried after the earlier request has finished with the RAC entry. The RAC also enables the DASH system to detect when local processors are accessing the same remote location: in this case the RAC combines the later request with the earlier one, and satisfies both requests when the reply to the earlier request is returned. To use the RAC for proxy transit data, each entry would need to have two extra fields to hold the node numbers of the head and finish of the proxy pending chain.

Adding proxying to the protocol outlined in Section 3.2 requires the addition of three new message types. Relatively minor changes are also needed in the protocol state machine to handle the new message types. These protocol changes could be made in software on a programmable node controller such as MAGIC [80]. The protocol changes needed to support

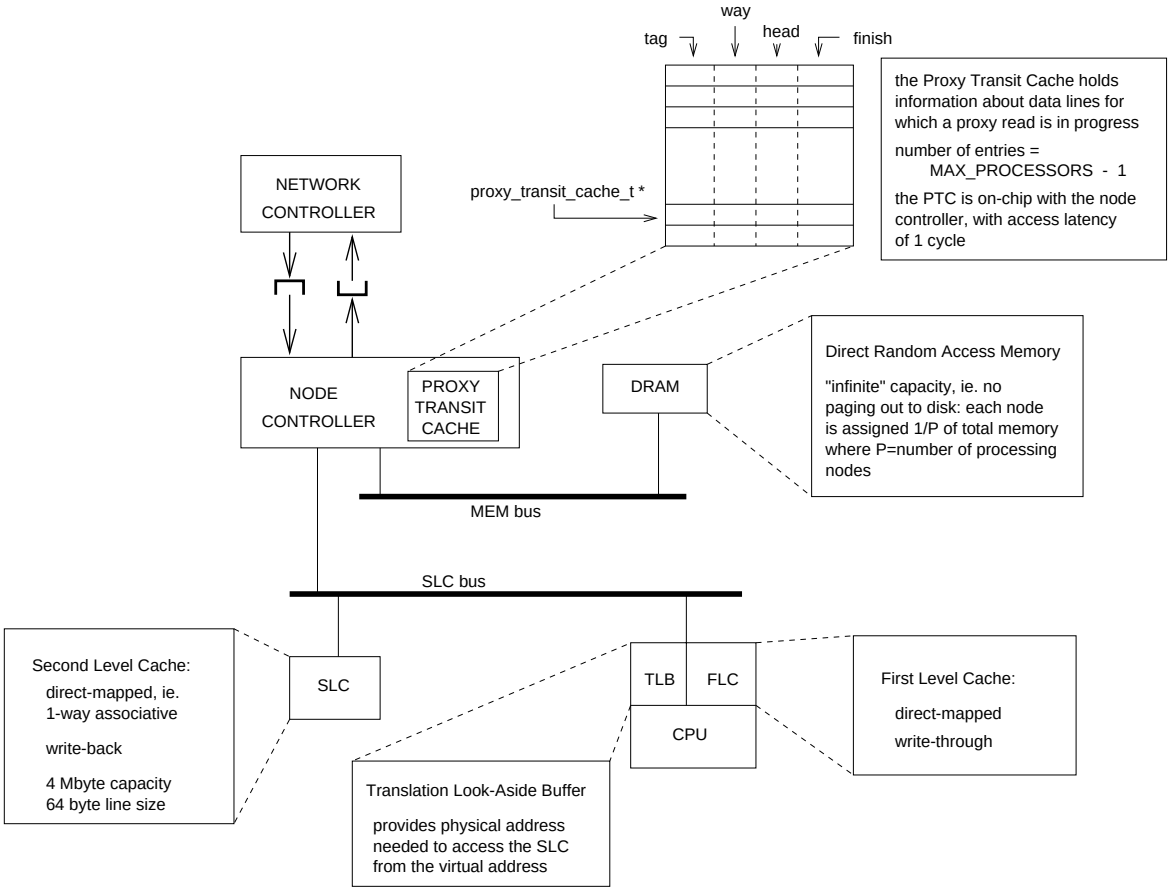


Figure 4.7: Memory model for a cc-NUMA node with basic proxies

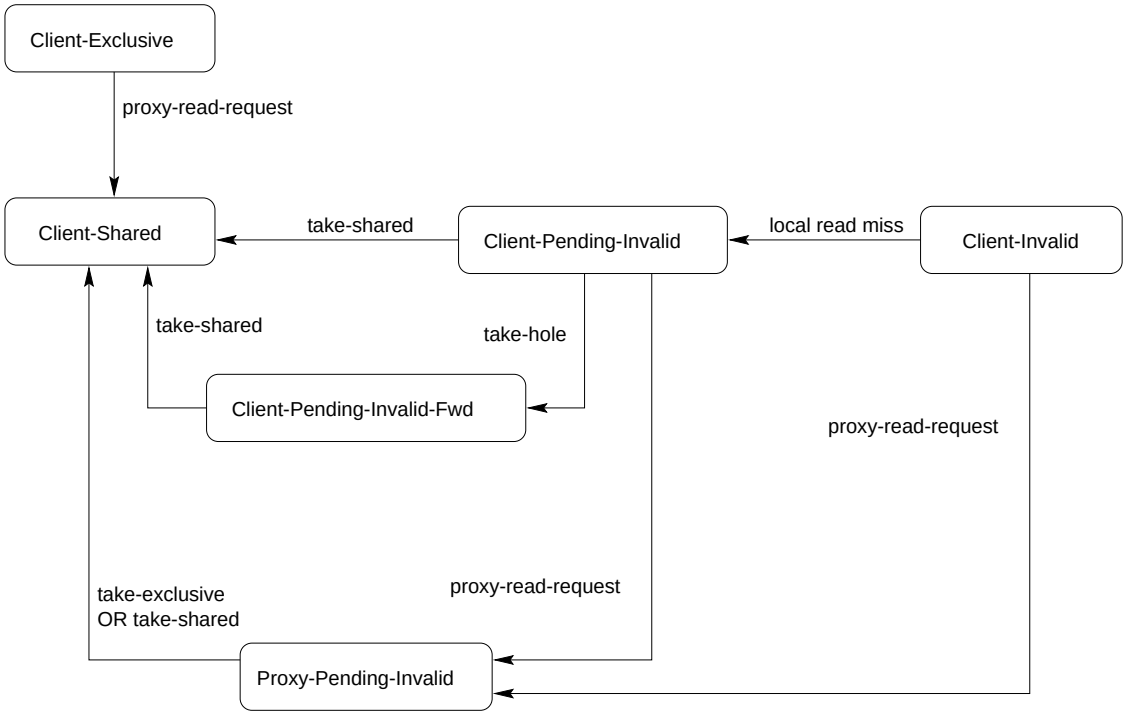


Figure 4.8: Extra node controller state transitions for client node actions

the proxy processing are illustrated in Figure 4.8. There are no extra node controller state transitions for home node actions. The new message types are:

proxy-bounced-read-request: this message is sent in response to a `proxy-read-request` when the proxy node is in the process of obtaining another data line which conflicts with the new request in the proxy transit cache. The client will re-send the `proxy-read-request`, but after 10 failed attempts it will revert to sending a `read-request` to the home node.

proxy-read-request: this message is sent by a client to a proxy node once the client has decided that it will use a proxy to service a remote read request.

take-hole: used to build the pending chain of clients. When a `proxy-read-request` arrives from a client, and the proxy is already in the process of obtaining the data for another client, this message is sent to the old "tail" of the pending chain to link it to the latest client (that latest client then becomes the new "tail"). The message is called "take-hole" because the client will be added to the pending chain, which is a hollow sharing list that contains no data until it receives a `take-shared` message. The use of the `take-hole` message was illustrated in Figure 4.5(b).

4.4 Results

This section presents the results obtained from execution-driven simulations of the basic proxy protocol. The objective of this experimental work was to investigate the potential benefits and costs of proxies, as outlined in Section 4.1.2. For details of the architecture simulated, refer to Section 3.4.

It has been shown in Section 3.5 that contention only becomes an important issue when more than a few tens of nodes are used. For this reason the detailed results presented below are from simulations of a 64 node design. Table 4.1 shows how the shared data was marked for

application	problem size	shared data marked for basic proxying
Barnes	16K particles	all
CFD	64×64 grid	all
FFT	64K points	all
FMM	8K particles	`f_array` (part of `G_Memory`)
GE	512×512 matrix	entire matrix
Ocean-Contig	258×258 ocean	`q_multi` and `rhs_multi`
Ocean-Non-Contig	258×258 ocean	`fields`, `fields2`, `wrk`, and `frcng`
Water-Nsq	512 molecules	`VAR` and `PFORCES`

Table 4.1: Benchmark problem sizes, and data marked for basic proxies

application	relative speedup no proxies	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
		1	2	3	4	5	6	7	8
Barnes	46.4	-0.3	0.0	0.0	0.0	-0.2	-0.1	-0.2	-0.4
CFD	28.8	+8.4	+7.0	+7.2	+11.8	+9.0	+8.2	+5.0	+13.7
FFT	47.2	+9.4	+9.1	+8.8	+9.0	+9.5	+8.7	+9.6	+8.5
FMM	52.4	+0.4	+0.3	+0.4	+0.4	+0.3	+0.4	+0.4	+0.4
GE	21.7	+28.8	+28.9	+29.0	+29.0	+29.1	+29.2	+29.2	+29.2
Ocean-Contig	49.8	-0.4	-2.1	-2.1	-5.0	+1.0	-0.8	-3.0	-3.9
Ocean-Non-Contig	50.8	-2.8	-1.6	-12.8	+0.1	+0.7	-9.4	-2.5	-1.5
Water-Nsq	55.3	-0.7	-0.6	-0.6	-0.6	-0.6	-0.5	-0.7	-0.5

Table 4.2: Benchmark relative speedups for 64 processing nodes

proxying for each of the eight benchmarks. The choice of data was based partly on knowledge of the applications, and partly as a result of data usage profiling [129]. For applications where there was no obvious sharing of data, all the shared data pages were marked for proxying.

The performance results for the eight benchmarks are summarised in Table 4.2. The performance speedup results are presented in terms of relative speedup, *i.e.* the ratio of the execution time for the fastest algorithm running on one processor to the execution time on $\mathcal{P}$ processors. The main point to note is that proxying improves the performance of four of the applications, but the other applications show mixed or slightly worse performance. In addition, the number of proxy clusters affects performance, but there is no "winning" value of $\mathcal{NPC}$ which gives the best result for all the applications. Figure 4.9 shows that the general trend is for the benefits of proxying to increase with the number of processing nodes.

In the following sub-sections, the results for each of the applications are examined in more detail in order to understand their different reactions to the introduction of proxies. For each application, the detailed results are presented as two graphs (one of relative changes and one of message ratios) and two histograms (one showing the execution time profile and the other showing the message category profile).

The relative changes results show four different metrics:

messages: the ratio of the total number of messages to the total without proxies,

execution time: the ratio of the execution time (excluding startup) to the execution time (also excluding startup) without proxies,

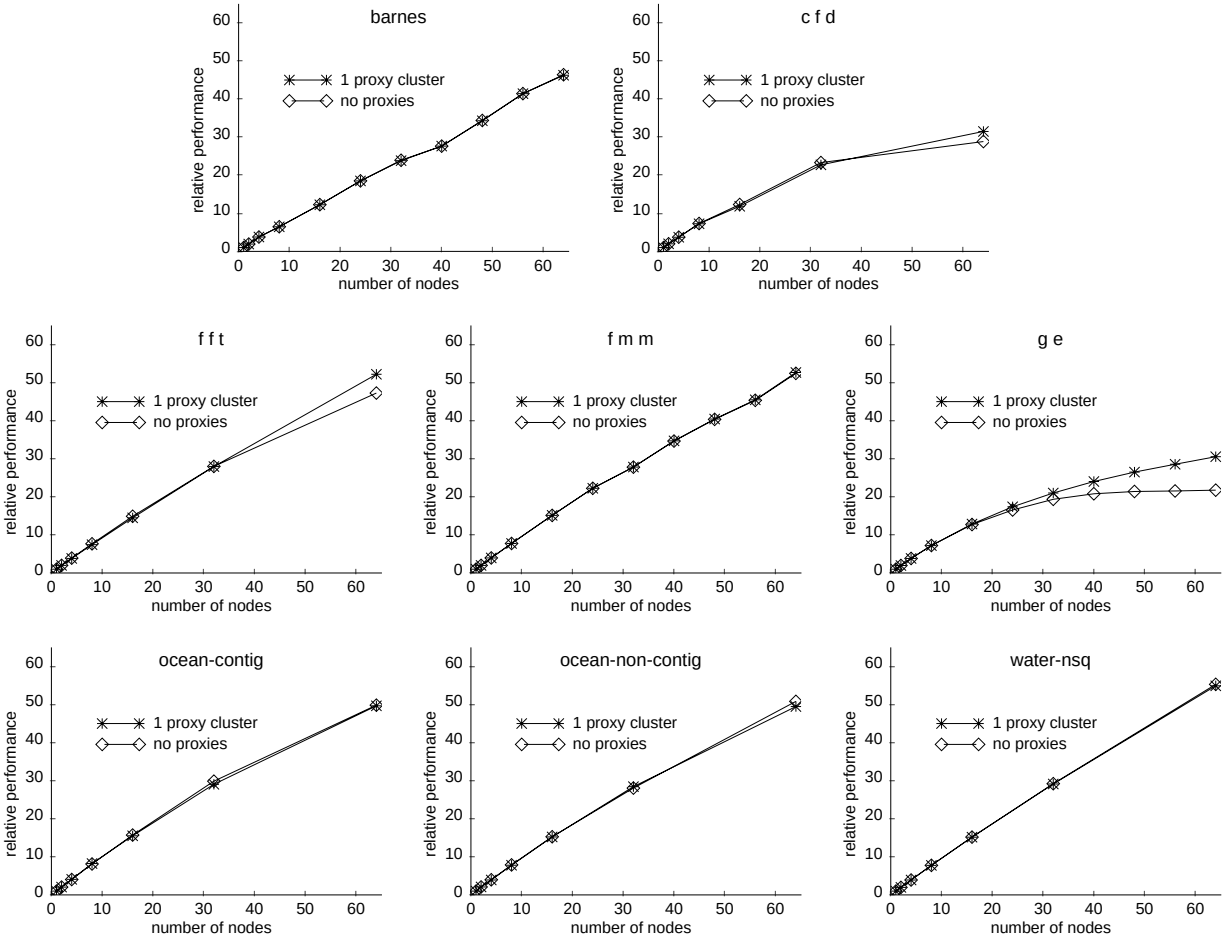


Figure 4.9: Performance speedup graphs

queueing delay: the ratio of the total time that messages spend waiting for service to the total without proxies, and

remote read delay: the ratio of the mean delay between issuing a `read-request` and receiving the data, to the same mean delay when proxies are not used.

The message ratios are:

proxy hit rate: the ratio of the number of proxy read requests which are serviced directly by the proxy node, to the total number of proxy read requests (in contrast, a proxy miss would require the proxy to request the data from the home node),

proxy read ratio: the ratio of the total number of `proxy-read-request` messages to the total number of `read-request` messages - this gives a measure of how much proxying is used by an application.

The execution time profile presents the overall execution time split into CPU active time and

the time spent waiting because of delays. The delays are further split into load, store, barrier and lock delays. The times are normalised with respect to the execution time without proxies.

The message category profile shows how the total number of messages breaks down into four categories: read, write, unhook, and proxy messages. The allocation of message types to message categories is given in Appendix C.2. It should be noted that read messages include all the `read-request` and `take-shared` messages, write messages include all the `write-request`, `invalidate` and `take-exclusive` messages, and unhooks are all the messages needed to handle the eviction of conflicting data lines from the SLCs.

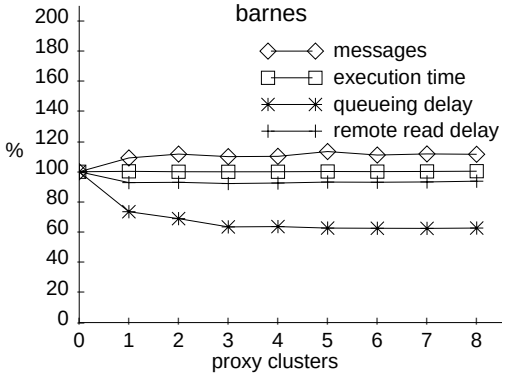
4.4.1 Barnes

This application does not benefit from the use of proxies, and for a number of values of $\mathcal{NPC}$ the performance worsens (by up to -0.4%). It should, however, be noted that proxies did achieve their aim of reducing node controller contention, as shown in Figure 4.10(a) by the reduction in mean queueing delay. This effect is reflected in the reduction in the remote read delay, despite the overall increase in the number of messages. The use of proxies is high, as shown by the proxy read ratio in Figure 4.10(b): this was expected because all the shared data was marked for proxying. The proxy hit rate indicates that there is a reasonable level of combining at the proxies, *i.e.* when `proxy-read-request` messages arrive at the proxy node there is up to 50% chance that the data is already there or has already been requested.

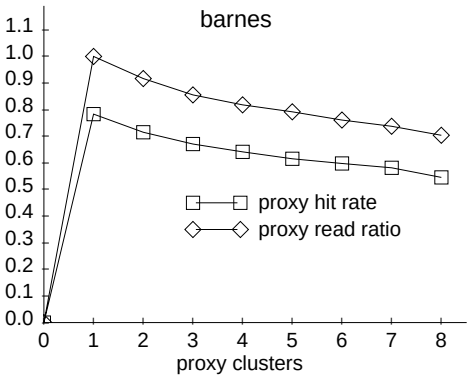
Although the adverse effect on performance is small, it is important to understand why it occurs. Using proxies reduces the number of read messages but adds the overhead of proxy messages, and this causes a slight increase in the overall load miss delay. In addition, cache pollution effects cause an increase in unhooking, as shown in Figure 4.10(d). The number of write messages increases because proxy nodes are added to the distributed sharing lists, so additional invalidation messages are needed before a write can complete, although this effect does not affect the store miss delay because of the overall reduction in queueing time. As a result of the increase in unhook messages, together with the introduction of proxy messages, the load miss delay increases despite the improvement in remote read delay. In addition the timing differences for individual node processing introduced by the additional messages have a slight knock-on effect on barrier and lock delays.

The Barnes application is an example of where the indiscriminate use of proxying can lead to performance degradation. All of the shared data was marked for proxying. Although combining of requests was achieved for at least 50% of the `read-request` messages sent to proxies, this was not sufficient to outweigh the delays introduced by the indirection of sending requests via proxies.

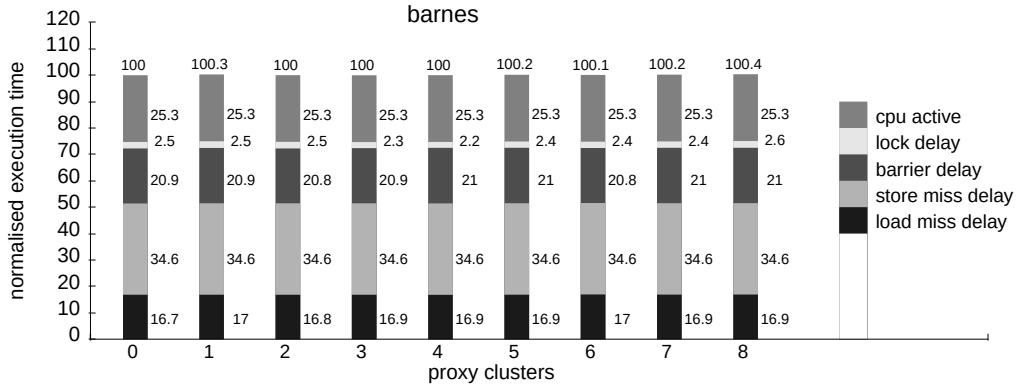
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

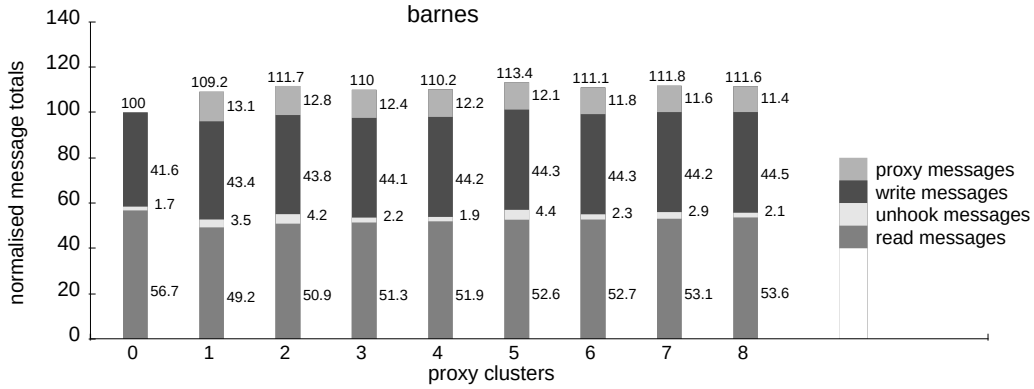


Figure 4.10: Barnes

4.4.2 CFD

Using proxies improves the performance of CFD by between 5.0% and 13.7%. The performance improvement and proxy hit rate exhibit a "stride" pattern with powers of 2, *i.e.* they are better for 2, 4, and 8 proxy clusters. This is an example of proxy selection being "in step" with data line ownership. In particular, at $\mathcal{NPC}$=8, when a processing node requires corona data from the "east" and "west" nodes, $\mathcal{PN}(l, C)$ picks the owner node for l. This results in a quite dramatic drop in the number of read messages, because there is no longer a need to go via the home node. By using a round-robin function for $\mathcal{PN}(l, C)$ the high combining rate and the reduction in `read-request` messages are an artifact of the data ownership being in-step with the proxy node selection function[3].

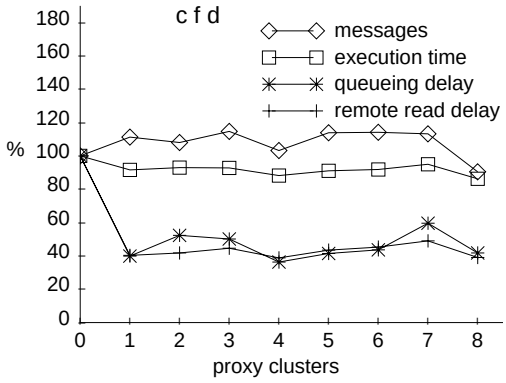
When $\mathcal{NPC}$ is not in step with the algorithm, there is still a considerable benefit to be had from proxies. Contention is reduced at the home node, and combining helps reduce the number of `read-request` and `read-request-fwd` messages, which in turn reduces the remote read delay and overall load miss delay. The least improvement is at $\mathcal{NPC}$=7, and is the result of a higher mean queueing delay than is seen for the other values of $\mathcal{NPC} \geq 1$ (see Figure 4.11). That increase in queueing delay is caused by the spread of `proxy-read-request` messages around the nodes in the system, resulting from the way the nodes are partitioned into clusters at $\mathcal{NPC}$=7. The best performance improvements are obtained when the problem size, cache line size, and $\mathcal{NPC}$ are all in step: change any one of these and the performance benefits for CFD are reduced.

4.4.3 FFT

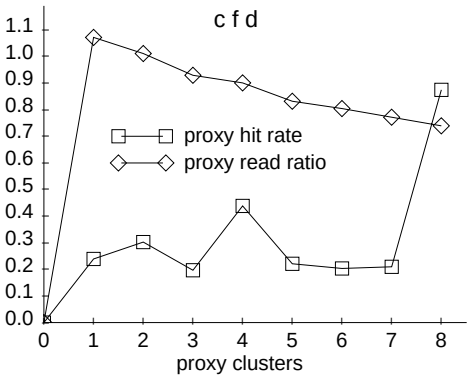
This application benefits from proxies, with performance improvements of between 8.5% and 9.6%, at $\mathcal{NPC}$=8 and 7 respectively. Figure 4.12 shows the dramatic decrease in queueing delay, which gives a corresponding decrease in node controller contention. As with the first two applications, all the shared data has been marked for proxying, and this generally leads to a high level of `proxy-read-request` messages. However, there is a low proxy hit rate indicating that there is very little combining. The performance benefit for FFT comes from spreading out the input queue waiting across the nodes, rather than from combining read requests for the same data line. The effect of spreading the queueing of messages around the system by using proxies is analogous to Valiant's two-phase random routing scheme which aims to reduce queueing at network switches [137]. It should also be noted that the level of unhooking increases when proxies are used with FFT. This indicates that there is cache

[3] Simulations using a more random $\mathcal{PN}(l, C)$ function avoided the artificial benefits observed for some values of $\mathcal{NPC}$ with CFD, but the randomness unfortunately worsened the results for all the other applications.

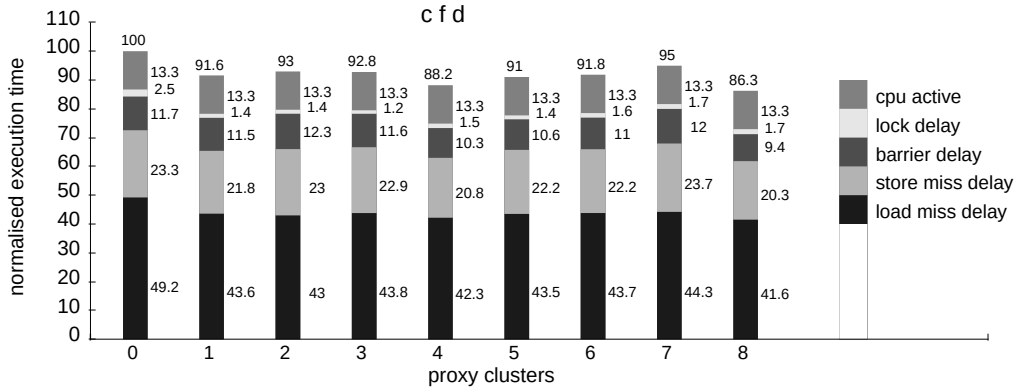
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

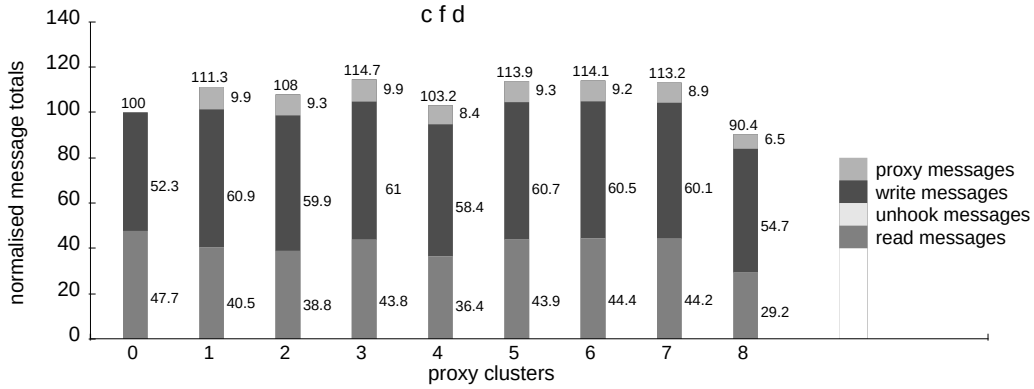
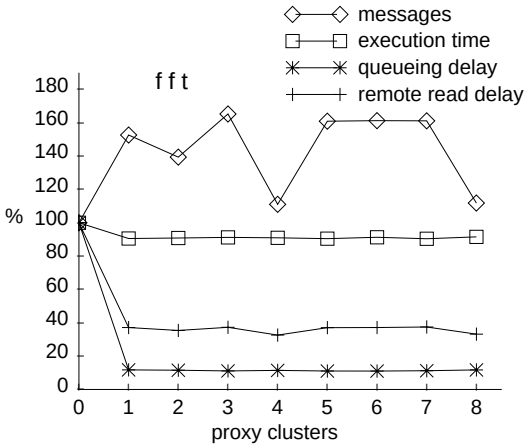
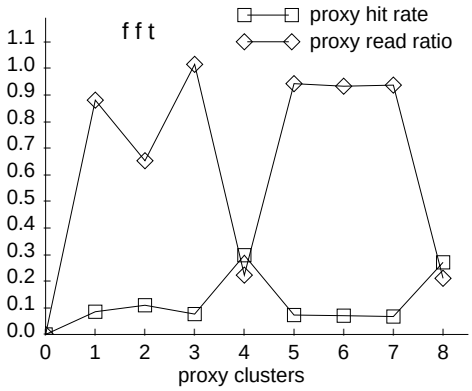


Figure 4.11: CFD

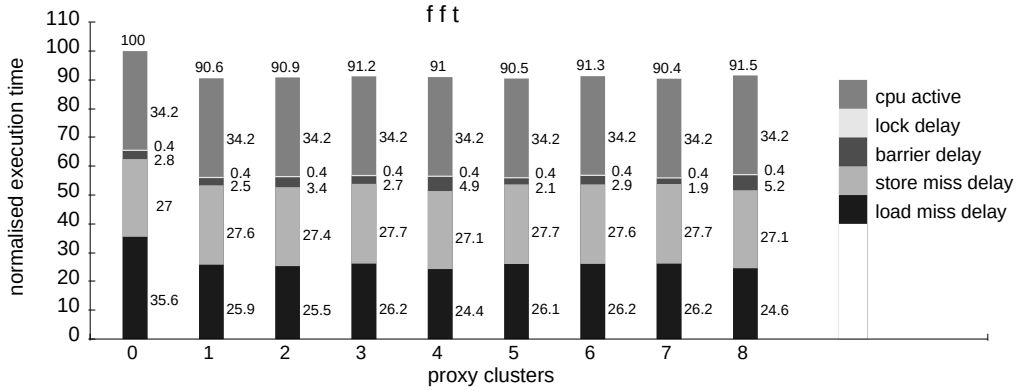
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

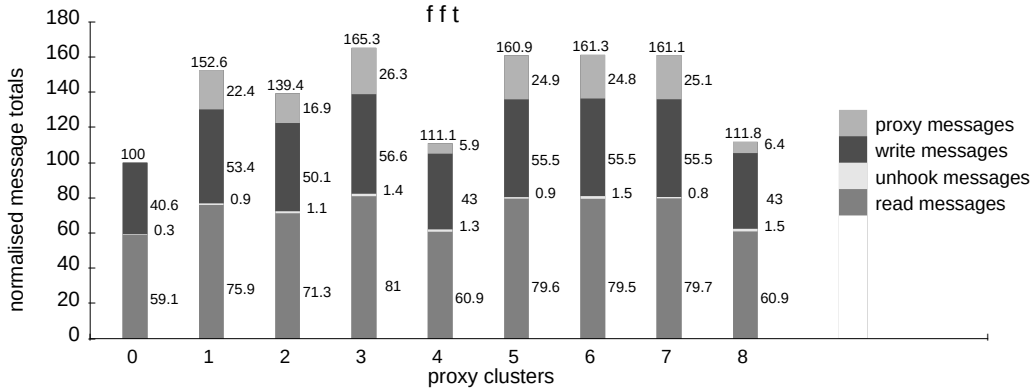


Figure 4.12: FFT

pollution in the SLCs as a result of holding proxy data lines.

There is an oscillation in the number of messages and proxy read ratio, with an inverse oscillation in the proxy hit rate. The lower message totals at $\mathcal{NPC}$=2,4&8 occur where the client node is the proxy within its cluster for the data line it requires. As with CFD, the effect is an artifact of the simple round-robin function that is used to select proxies.

The benefits of proxying are also shown by the reductions in both the load miss delay (all read misses by CPUs), and the reduction in remote read delay. However, there is a slight increase in store miss delay, attributable to the increase in invalidation messages now that nodes are holding data as proxies: this is reflected in the increase in write messages as shown in Figure 4.12(d). The number of read category messages also increases when proxies are used. This is due to a sizeable increase in the number of `take-shared` messages, because (bearing in mind the low rate of combining) using proxies for FFT usually replaces one `take-shared` message to the client with two `take-shared` messages: one to the proxy and another one from the proxy to the client.

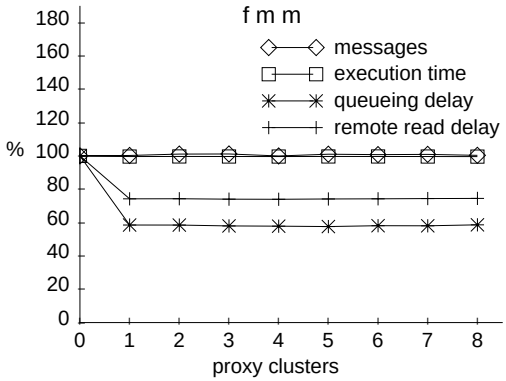
4.4.4 FMM

This application gets a marginal improvement in performance by using proxies, with performance gains of between 0.3% and 0.4%. Only one data structure has been marked for proxying in FMM, chosen because it was the only obvious example of widely-shared data in the application. As a result there is very little use of proxies, as shown by the low proxy read ratio and small number of proxy messages (see Figure 4.13), but the proxying which does occur is very effective, and has a high proxy hit rate. The queueing delay is reduced by more than 40%, with corresponding reductions in remote read delay, and load miss delay. However it is not surprising that there is only a small improvement in performance, because proxying is aimed at reducing load miss delays, and these account for only 5% of the overall execution time.

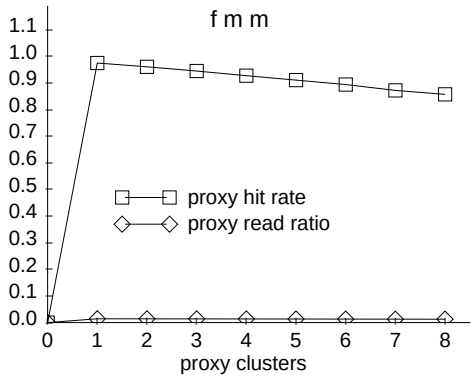
4.4.5 GE

The Gaussian Elimination application benefits greatly from the use of proxies, with performance improvements of around 29%. The overall queueing delay is reduced dramatically when proxying is used, dropping by over 90%. This is a result of the application's behaviour: in each iteration, all nodes need a copy of the current pivot row, which will have one home node (or possibly more, if the row crosses page boundaries). When proxies are used, the read messages are diverted to proxy nodes, resulting in a more uniform distribution of messages,

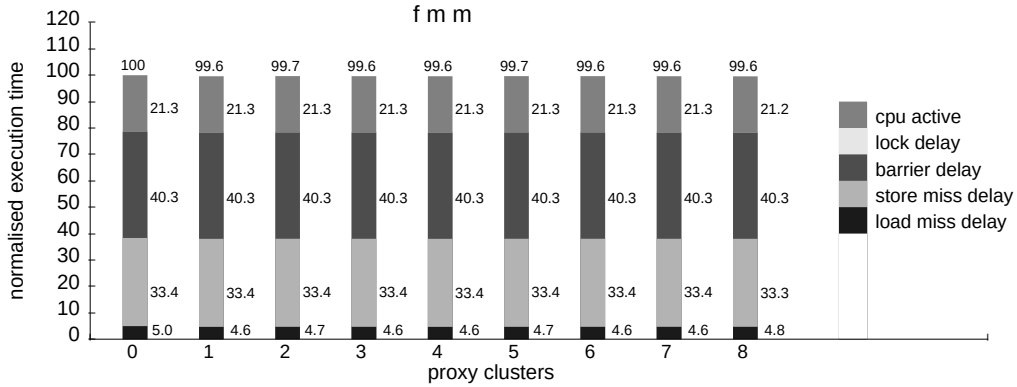
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

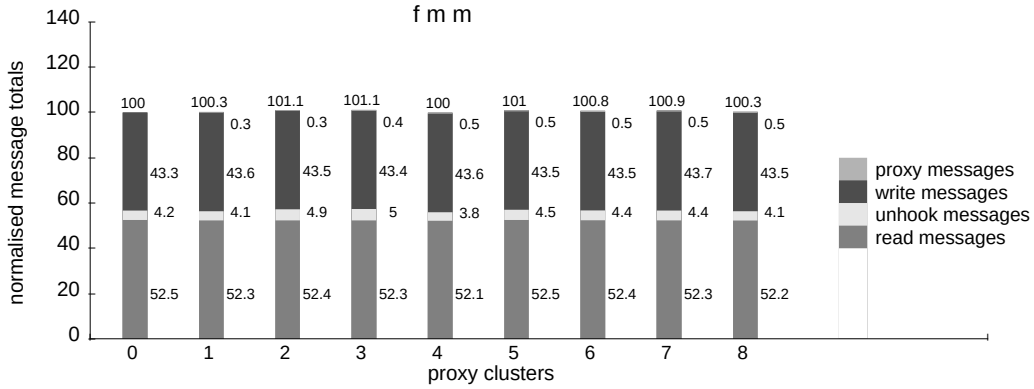
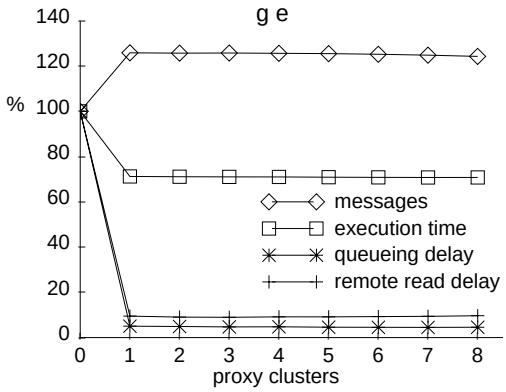
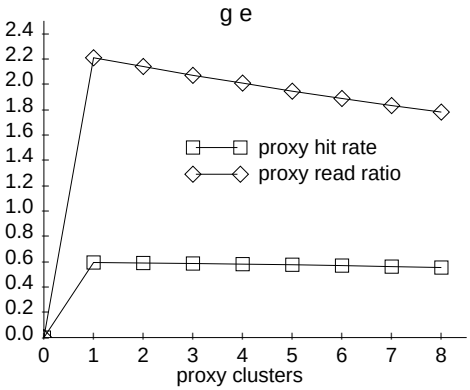


Figure 4.13: FMM

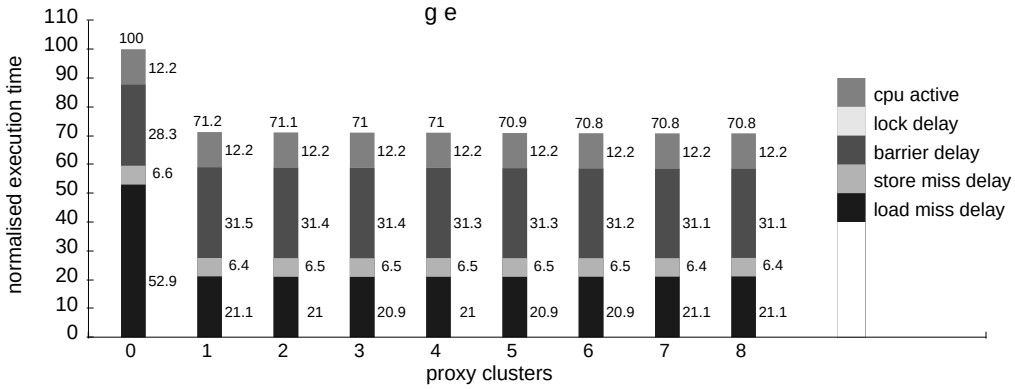
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

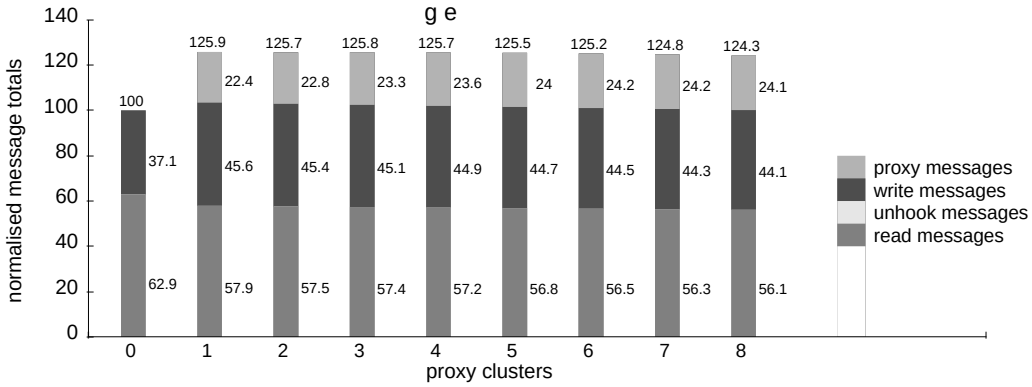


Figure 4.14: GE

reduced queueing delay, and an overall reduction in remote read delay. This has the effect of reducing the overall load miss delay by more than 50%, as shown in Figure 4.14(c). There is a slight increase in barrier delay, due to some nodes checking-in earlier: *i.e.* there has been an increase in barrier delay because the decrease in load delay can introduce more processing imbalance.

The store miss delay is slightly reduced, reflecting the shorter queueing delays because messages related to writes are no longer caught in long queues at the hot-spot nodes. That being said, the number of write messages increases because all updates to matrix data first have to invalidate any proxy copies in addition to the "genuine" members of the sharing list. Although the current pivot row data is required by all the nodes, any other read to shared data will also be proxied, resulting in proxy nodes being on the sharing list which would not have been there without the proxy scheme. The overall increase in write category messages does not result in a longer store miss delay because the large reduction in queueing delay means that the messages suffer much shorter queueing penalties even though there are more messages.

The proxy read ratio is very high for this application, reflecting the fact that the vast majority of `read-request` messages have been converted to `proxy-read-request` messages. The value tails off as $\mathcal{NPC}$ increases, because there are fewer clients issuing proxy read requests and more proxy nodes issuing read requests to the home nodes.

Figure 4.14(b) shows the proxy hit rate to be highest with one proxy, and then it tails off slowly as the number of proxies is increased. This is as expected, because every node accesses each data line of the pivot row; therefore a high rate of combining is possible, but the best combining rate is achieved when only one location is acting as proxy for a given data line. When $\mathcal{NPC}$=1, all `proxy-read-request` messages for a particular data line are sent to the same node, and this will give the best combining, since at most one `read-request` has to be sent on to the home node.

The proxy hit rate is not as high as it is for some applications, *e.g.* FMM, because in GE all reads for matrix data are converted into proxy read requests, but those reads not relating to the current pivot row are not widely-shared. For example, a read from the new owner of the data will be followed by a write, so combining for this data line is likely to be very low (*i.e.* zero unless the proxy is the current owner). Contrast this with pivot row data lines where, with $\mathcal{NPC}$=1, the hit rate will be very high, because the first proxy read request will trigger the read to the home node, and all the subsequent proxy read requests will combine at the proxy.

As $\mathcal{NPC}$ increases, the pivot row's proxy hit rate will decrease as more `proxy-read-request` messages suffer an initial "miss" at different proxies, but the chances of a proxy being the owner node will increase, so for non pivot row reads, the proxy hit rate should increase slightly: this leads to an slow decline in the overall proxy hit rate.

The performance does not vary much with changes in the value of $\mathcal{NPC}$ because the performance benefit comes from directing the read requests away from the home node, and varying $\mathcal{NPC}$ does not really affect this for GE. With $\mathcal{NPC}$=1, there is maximum combining, but it takes longer to send the data to all the clients on the single pending chain. As $\mathcal{NPC}$ increases, the burden of being proxy for a particular data line is spread over more nodes, and the barrier delay (which is symptomatic of processing imbalance) improves slightly. As a result, the overall execution time shows gradually greater improvements.

4.4.6 Ocean-Contig

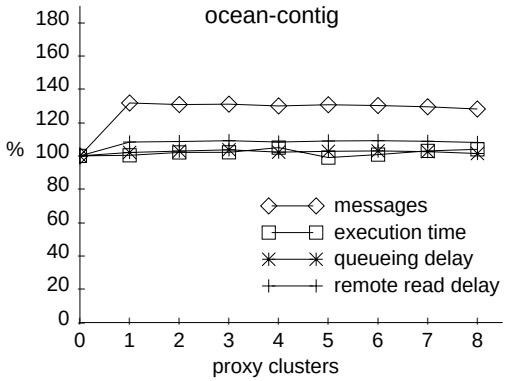
The results for Ocean-Contig are summarised in Figure 4.15. This application is unusual among the benchmarks, because the queueing delay and remote read delay both increase, *i.e.* proxies degrade rather than improve these performance measures. The application has been specifically written to exploit data locality. The introduction of proxies increases the sharing list by one for each proxy, because the proxy node does not require the data itself, *i.e.* it is not on the sharing list when $\mathcal{NPC}$=0. This increases the `take-shared` messages, and also increases the number of invalidation messages required when a node wants to write to a proxied data line. Figure 4.15(d) shows that there is a significant increase in the number of read and write messages. The read messages total includes all `take-shared` messages, and the write messages total includes the invalidation messages required to get exclusive ownership of a data line.

The data locality of Ocean-Contig means that using proxies is likely to increase indirection without the compensation of combining at the proxy nodes. In addition, proxying shifts the timing of processing at the individual CPUs and so affects the barrier delays. This change of barrier delays results in the only performance improvement, which occurs when $\mathcal{NPC}$=5 and the barrier delay is at its lowest level.

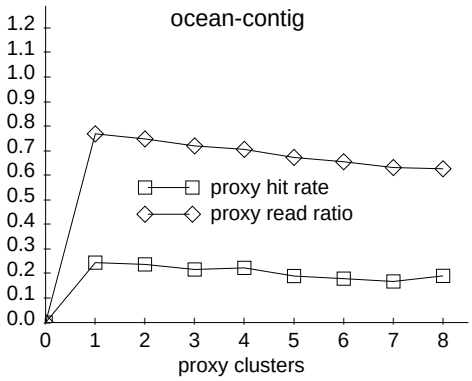
4.4.7 Ocean-Non-Contig

The performance of this application varies, depending on the number of proxy clusters, between an improvement of 0.7% and a slowdown of -12.8% (see Figure 4.16). The performance varies according to the effect proxies have on the queueing delay: when queueing delay is worse

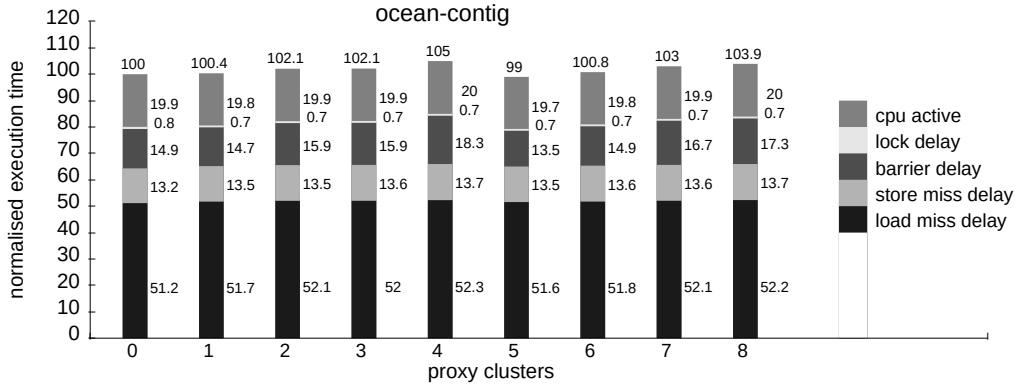
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

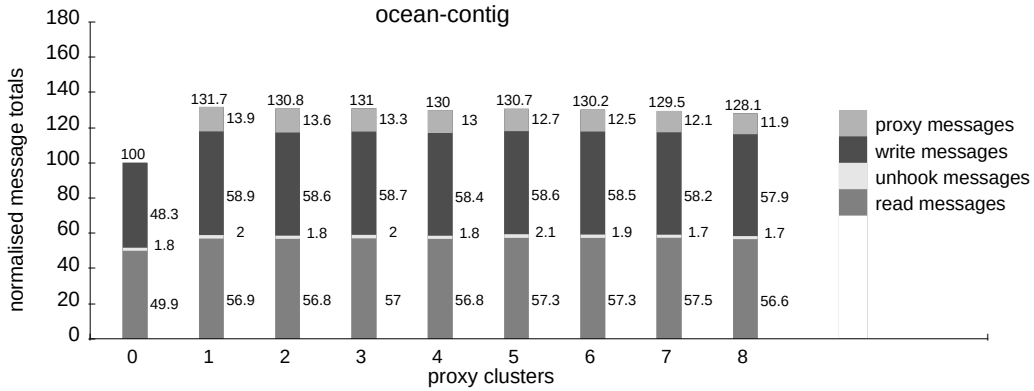
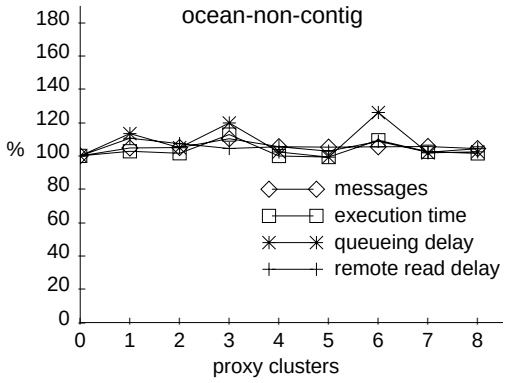
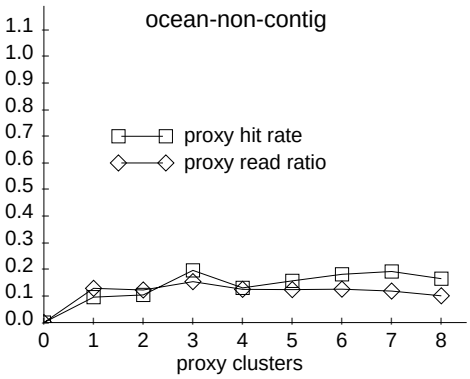


Figure 4.15: Ocean-Contig

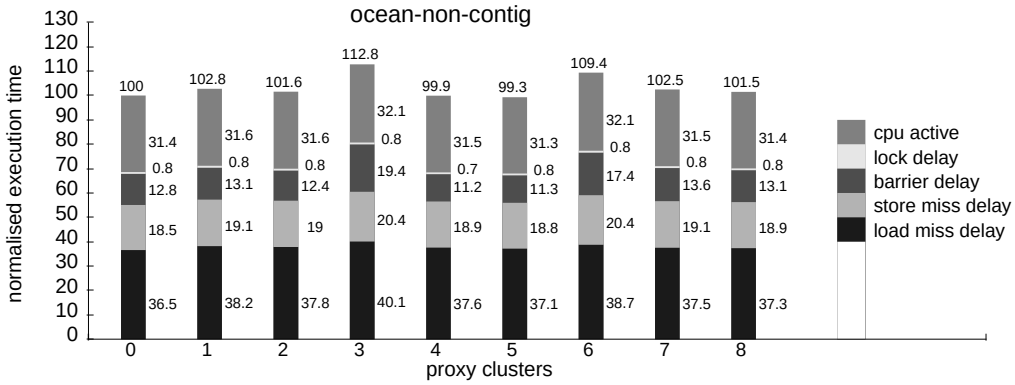
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

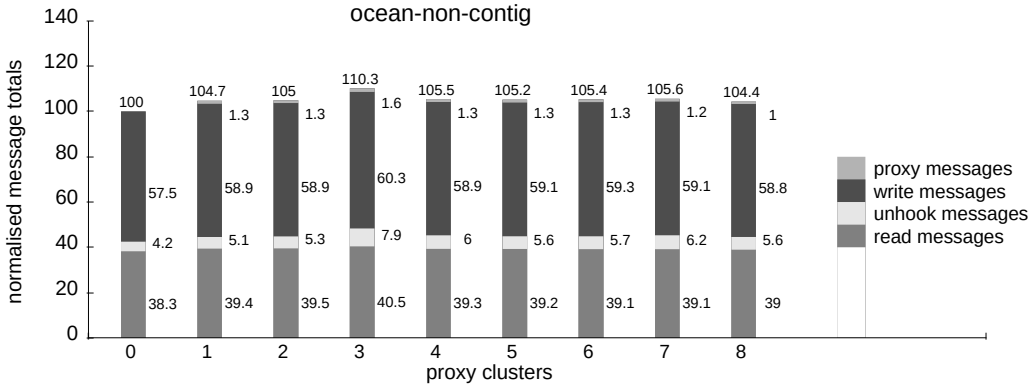


Figure 4.16: Ocean-Non-Contig

than the base case, performance suffers; when it is better than the base case, performance improves. The question is, why does the queueing delay oscillate depending on the value of $\mathcal{NPC}$?

The performance is at its best when $\mathcal{NPC}$=4&5, *i.e.* when the queueing delays are spread evenly around the system and the individual nodes do not receive more than their fair share of proxy read requests. For the other values of $\mathcal{NPC} \geq 1$, the mean queueing time for some nodes is affected more than others, and when two or more nodes have queueing times that stand out from the pack, the performance suffers. In Ocean-Non-Contig, poor data locality is the price paid for having an implementation that is easier to understand than Ocean-Contig. As a result, there is already a high level of messages in the network, and certain values of $\mathcal{NPC}$ have the unfortunate effect of creating new message queue bottlenecks.

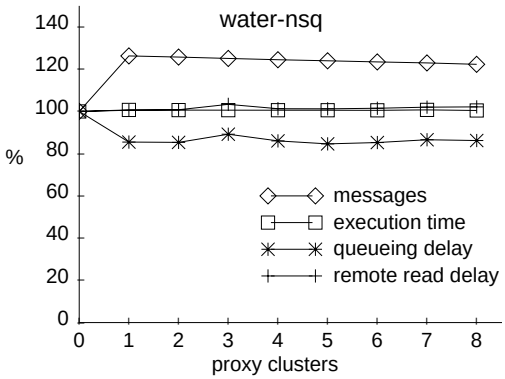
It should also be noted from Figure 4.16(d) that there is an increase in unhook messages for all values of $\mathcal{NPC} \geq 1$. This indicates that proxies are causing cache pollution for this application.

4.4.8 Water-Nsq

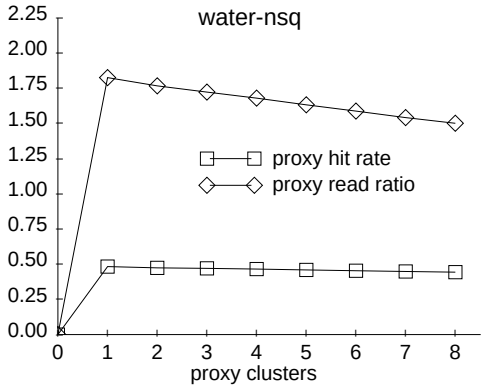
This application shows a slight worsening in performance when proxies are used, ranging between -0.5% and -0.7%. As shown in Figure 4.17, proxies reduce queueing but they have the effect of slightly increasing the load and store miss delays. Most of the `read-request` messages from clients are converted to `proxy-read-request` messages, and this is reflected by the high proxy read ratio. However, combining is below 50%, and so many read requests are forwarded on from proxy nodes to home nodes. Taken together with the extra `take-shared` messages needed to pass the data to proxy nodes, the overall level of read category messages remains close to the level seen without proxies. By routing the messages via proxies the queueing delay is slightly reduced, which is the aim of proxying. However, the remote read delay shows a slight increase due to the indirection of going via a proxy node and the relatively low level of combining. In addition, there is a marked increase in the number of write messages, primarily because of the extra invalidation messages needed to remove proxy copies from sharing lists. This results in a slight increase in the overall store miss delay.

The slight "blip" in queueing delay, when $\mathcal{NPC}$=3, is due to a very uneven distribution of proxy read requests in cluster 1 (nodes 1 to 21) and cluster 2 (nodes 22 to 42), as shown in Figure 4.18. The distribution of `proxy-read-request` messages is very even for all the other values of $\mathcal{NPC} \geq 1$. However, although the uneven distribution of `proxy-read-request` messages at $\mathcal{NPC}$=3 slightly increases the overall remote read delay, it does not have any extra effect on the execution time.

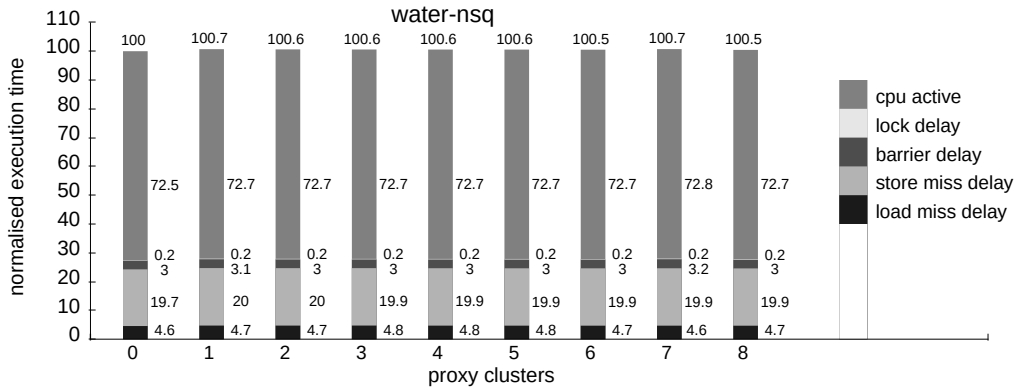
(a) Changes (relative to no proxies case)



(b) Message ratios



(c) Execution time profile



(d) Message category profile

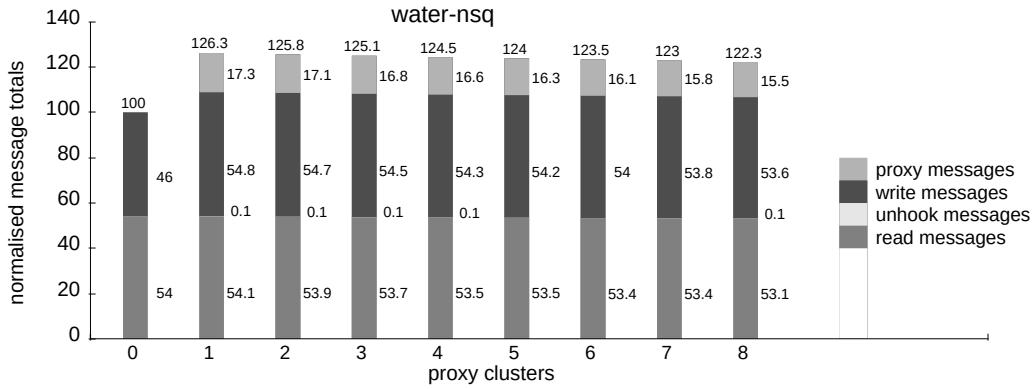


Figure 4.17: Water-Nsq

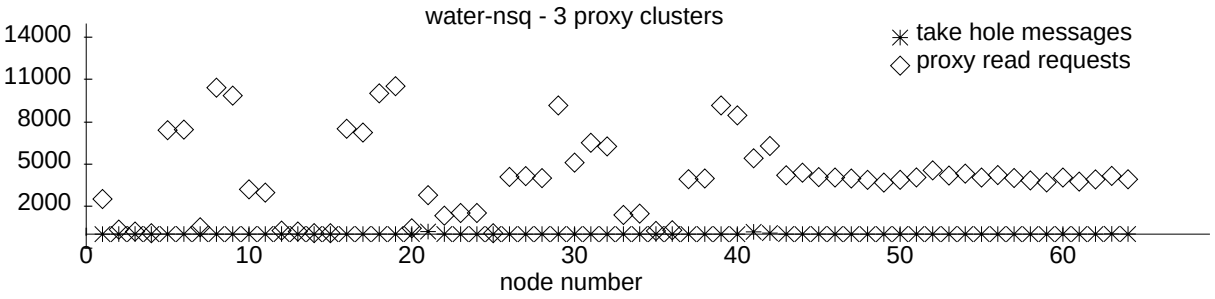


Figure 4.18: Individual incoming `take-hole` and `proxy-read-request` message totals for Water-Nsq, with $\mathcal{NPC}$=3

4.4.9 Summary of Results

Using the basic proxy protocol, which routes read requests via proxy nodes when the data is marked for proxying, does not have the same effect on all the applications. Four out of the eight benchmarks had better performance when proxies were used, although this ranged from a peak improvement of 29.2% with GE, to a minimum improvement of 0.3% with FMM. Of the remaining four benchmarks, the performance of Ocean-Contig and Ocean-Non-Contig oscillated between being worse and being better depending on the value of $\mathcal{NPC}$, while using proxies had a slightly adverse effect on Barnes and Water-Nsq. There were a number of reasons for this variance: the selection of data structures for proxying, the level of remote reads in the application (*i.e.* algorithm locality), the interaction of the proxy selection function with the data accesses, unhooking caused by cache pollution, the level of combining, and the trade-off between higher combining and shorter pending chains.

The GE benchmark showed a significant performance improvement using basic proxies. This was to be expected because the application was specifically chosen as an example of read contention. The application ran around 29% faster on 64 nodes with proxies. For CFD, when $\mathcal{NPC}$ was in step with the data ownership for a particular problem size and cache line size, the performance benefits were enhanced. However, the application still benefited from the use of proxies for other values of $\mathcal{NPC}$.

Ocean-Non-Contig showed oscillating behaviour, *i.e.* the performance was usually worse but sometimes better with proxies. This oscillation was caused by the interaction of the queueing redistribution resulting from proxies (different for each value of $\mathcal{NPC}$) with the high level of messages needed during the execution of the application. As a result, new queueing bottlenecks were created for some values of $\mathcal{NPC}$. Barnes and Water-Nsq both showed a slight drop in performance. This serves as a warning that basic proxies can have a detrimental effect on performance if the wrong data structures are marked for proxying.

Although only GE showed a substantial improvement in execution time, the results have demonstrated that more random message delivery can result in reduced message queueing, even when there is an overall increase in network traffic. Although the overall message traffic level is somewhat higher, the distribution is more uniform.

There are some overheads associated with proxies. Every load (for addresses subject to proxying) goes via a proxy, whereas with the underlying protocol no indirection would be involved. This can be detrimental to performance where inappropriate data structures are marked as "hot". In addition, there can be cache pollution *i.e.* allocating space in the cache for proxied data may displace another line, and lead to a later cache miss, with an unhooking overhead for the displaced line. In the simulations, these delays were more significant for some applications than for others, and this effect is studied further in Chapter 6 and Chapter 7.

To what extent do these results depend on details of the simulator? The experimental setup was designed to give results that were relatively straightforward to interpret, and so some simplifying assumptions were made. The most important is assuming a fully-interconnected network. This means that blocking occurs only when there is contention for network interfaces. With proxies, the overall traffic level is increased, but the traffic pattern is partially randomised so that the load tends to be spread more evenly. This is highly desirable for some, but not all network designs. This simplification has allowed the study of the interaction between the proxy selection algorithm and the data access patterns of the applications.

The wide range of performance results, the difficulty of selecting the correct data for proxying, and the hardware and software costs of implementing proxies, lead to the conclusion that this basic form of proxies is not suitable for general use. However, proxies can give a noticeable improvement in performance in some cases.

4.5 Conclusions

This chapter has introduced the proxy protocol extensions, and evaluated the basic form of proxies. The results are encouraging, in that some applications show a marked performance improvement. However, the slight drop in performance suffered by other applications, such as Water-Nsq, indicates that the basic form of proxies is not suitable for general use in cc-NUMA architectures. In the next three chapters, different implementations of the proxy strategy are considered to determine whether proxies can be provided in a way that is beneficial for all applications.

Chapter 5

Automatic Invocation of Proxies

The basic form of proxies, described in Chapter 4, has the disadvantage that data has to be marked for proxying by the application programmer, or possibly by a compiler. This chapter introduces two automatic forms of proxying, *reactive* and *adaptive* proxies, which detect run-time node congestion and make this the trigger for using proxies. The mechanism for detecting node congestion exploits the handling of full message buffers which will be present in any real system. Some of the material in this chapter first appeared in [130], which presented preliminary results for the reactive proxies scheme.

5.1 Finite Buffers

In real systems, message buffers have a limited size. This limitation affects all buffers, including those in the network, and the input and output buffers of each node controller. In addition, in large-scale systems it is impractical to provide enough buffering at each node to hold all the incoming messages, because all (or many) of the nodes in the system could simultaneously send a request to one node. These physical limitations complicate the task of ensuring the correct operation of distributed systems: inter-node operations must be guaranteed to complete in a finite amount of time, and the system as a whole must make forward progress [89]. The memory system must be free from deadlock, and individual requests must not starve. Figure 5.1 illustrates two of the problems that can occur with finite buffers.

A commonly adopted strategy for tackling these problems (*e.g.* as used in the Stanford DASH system [88] and the MIT M-Machine [35]) has four parts [54]:

- A separate network (physical or virtual [27]) is used for requests and replies, where a reply is any message that a controller waits for before moving to a new state. This ensures that new requests cannot block replies that will free up buffers.

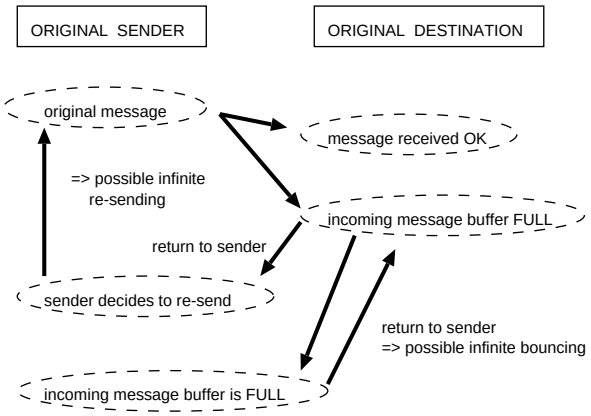


Figure 5.1: The problem of starvation with bounded input queues

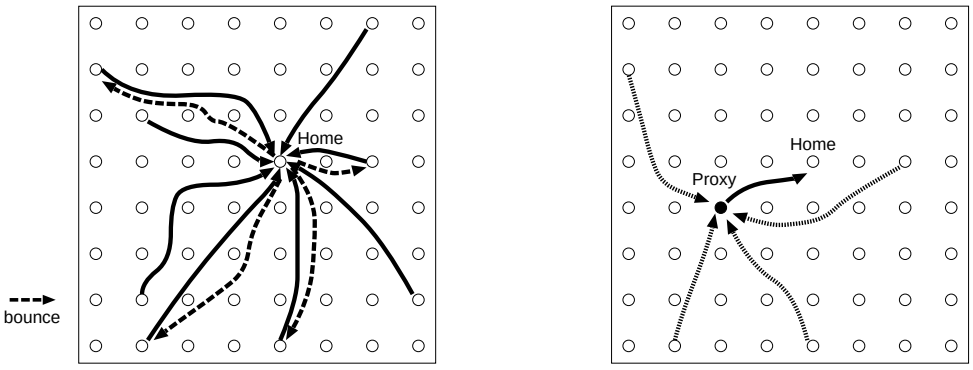
- Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.
- Any controller can reject (usually with a *negative acknowledge* or NAK) any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.
- Any request that receives a NAK in response is simply retried.

A result of adopting such a strategy is that when a `read-request` reaches a full buffer, a NAK will be sent back to the requester. The requester will be able to accept the NAK because it will already have allocated buffer space in the reply buffer.

This approach will avoid the problem of possible infinite bouncing of messages due to full incoming message buffers. However there is still the possibility that the original sender will suffer "node starvation" if it keeps having to re-send the request message. This problem can be avoided either by using timestamp-based priority or by having a higher class of request message which is used when the normal request becomes stale. Either way, when a request has failed to be accepted by the destination node over a "long time", there has to be a way of forcing the destination to accept the request [89].

5.2 Reactive Proxies

In the basic form of proxies, the application programmer uses program directives to mark data structures for handling by the proxy protocol - all other shared data will be exempt from proxying. If the application programmer makes a poor choice of data structures to



(a) Input buffer full, some read requests bounce (b) Reactive proxy reads

Figure 5.2: Bounced read requests are retried via proxies

be marked for proxying, then the overheads incurred by proxies may outweigh any benefits and degrade performance, as was noted in Section 4.4.9. These overheads include the extra work done by the proxy nodes handling the messages, proxy node cache pollution, and longer sharing lists. In addition, the programmer may fail to mark data structures that would have given noticeable performance benefits if they had been proxied. An alternative approach is to detect node controller contention when it occurs at run time, and use this to trigger the proxy protocol extensions.

As explained in Section 5.1, any system with limited buffering has to provide a mechanism to handle bounced requests, *i.e.* those messages which cannot be inserted into a full buffer. When a remote `read-request` reaches a full buffer, a NAK (negative acknowledgement) message is sent back across the network. The receipt of the NAK message indicates to the requester that there is a queueing bottleneck, and so this can be used to trigger proxy processing. This *reactive* form of proxying only operates when there is congestion: if no NAKs are received then no `proxy-read-request` messages will be sent. When a NAK is received in response to a `read-request`, the requester will then send a `proxy-read-request` rather than repeating the `read-request`, as illustrated in Figure 5.2. The proxy node will be selected in the same way as for basic proxies (see Section 4.2.2). Reactive proxies were first described in [130].

The reactive proxies protocol, therefore, differs from the basic proxies protocol, where the user has to decide whether all or selected parts of the shared data are proxied, and proxy reads are *always* used for data marked for proxying. Instead, the proxying only takes effect when congestion occurs. A proxy read is only done in direct response to the arrival of a `buffer-bounced-read-request`, so as soon as the queue length at the destination node has reduced to below the limit, read requests will no longer be bounced and no proxying will be employed. This removes the obvious overhead of the basic protocol, where proxying may add at least one message to every remote read on shared data marked for proxying.

A possible variation on this scheme would be to re-route a read request directly to a proxy from the bouncing node, rather than just sending a NAK back to the client. The problem with this idea is that it compromises the simplicity of the NAK approach, and it would mean that each network controller would have to have the capability of calculating $\mathcal{PN}(l, C)$. The simple approach has been used in this work to avoid complicating the underlying protocol and the hardware needed to implement it.

Using the arrival of `read-request` NAKs is a straightforward way of detecting home node congestion. However the congestion will not always be due to many clients requesting the same data line from the home node: the full buffer(s) may be due to a hot spot in the network, or many clients requesting different data lines from the same home node. Although the use of proxies in such circumstances may not get much benefit from combining, it is expected that the re-routing of requests via proxies will have the effect of avoiding hot spots and staggering the arrival of requests at the home node, in the same way as the two-phase random routing described by Valiant [137].

The need to dynamically identify widely-shared data at run-time, rather than relying on static marking of data, has independently been identified by Kaxiras *et al.* in their recent work on the GLOW extensions to SCI [69]. GLOW intercepts requests for widely-shared data by providing agents at selected network switch nodes. In their dynamic detection schemes, agent detection achieves better results than the combining of the NYU Ultracomputer [45] by using a sliding window history of recent read requests, but does not improve on the static marking of data. Their best results are with program-counter based prediction (which identifies load instructions that suffer very large miss latency) although this approach has the drawback of requiring customisation of the local node CPUs.

It should be noted that there are other approaches to ensuring forward progress, such as that used in the Thinking Machines CM-5 [87], where a delivery contract guarantees that any message accepted by the network will be delivered to its destination. The data network promises to eventually accept and deliver all messages injected into the network by the processors as long as the processors promise to eventually remove all messages from the network when they are delivered to the processors. The contract relies on the use of a split request-response network. Such an approach will not have the NAK messages to indicate that there is a problem with full buffers. However it would be possible in such an environment to record the response time for read requests, and to use the detection of an overly long response as the trigger for using a `proxy-read-request`.

The reactive proxy scheme has the twin virtues of low overheads and simplicity. No information needs to be held about past events, and no decision is involved in using a proxy:

the protocol state machine is simply set up to trigger a `proxy-read-request` in response to the receipt of a NAK'd read request. However, the strategy is "dumb", in as much as when another `read-request` needs to be sent soon after a NAK from a particular home node, it will be sent to that home node even though it is likely to still be congested.

5.3 Adaptive Proxies

A more sophisticated strategy, adaptive proxies, uses the arrival of a NAK'd `read-request` message to trigger the start of a proxy-period, *i.e.* a time during which any further `read-request` messages destined for the home node are replaced with `proxy-read-request` messages. The proxy-period is modified according to the level of buffer-bouncing. Adaptive algorithms cover all areas where a system can adapt to suit run-time conditions, with examples including adaptive coherency protocols, cache bypassing, and page management. The schemes have a common theme of a threshold value (which itself may be varied) which is used to decide whether to use one strategy or another. A selection of adaptive schemes were examined in Section 2.3 of this thesis, including sequential prefetching [26], the adaptive predictors for accelerating coherence protocols in Cosmos [98], and the random walk policy.

The *random walk* policy is a simple and effective adaptive algorithm that has been used in a number of domains including hybrid update/invalidate cache coherency protocols [7] (see Section 2.3.3 of this thesis). The random-walk policy has low overheads, and so has been used in this thesis as the basis for adaptive proxying. The algorithm is based on the assumption that the probability of a buffer-bounce (from a particular home node) occurring within an upper time limit of the last buffer-bounce (from that home) is high if the last inter-bounce period was less than the upper time limit. Given:

- last buffer bounce time $LB_{(x,y)}$, *i.e.* the time at which the last buffer-bounce message was received at client node x from home node y,
- current time T_{curr},
- the most recent inter-bounce time $IB_{(x,y)}$, *i.e.* the gap between buffer-bounces from home node y to client node x, set to $(T_{curr} - LB_{(x,y)})$,
- PP_{unit} is one unit of proxy-period time,
- PP_{max} is the maximum proxy-period,
- the maximum inter-bounce threshold IB_{max}, set to $(PP_{unit} \times PP_{max})$,
- PP_{min} is the minimum proxy-period,

- the proxy-period for reads from client node x to home node y is $PP_{(x,y)}$. All $PP_{(x,y)}$ are initialised to PP_{min},

then the arrival at client node x of a `buffer-bounced-read-request` from home node y will trigger the adjustment of $PP_{(x,y)}$ as follows:

if $(IB_{(x,y)} < IB_{max})$

then add 1 unit to $PP_{(x,y)}$,

otherwise subtract 1 unit from $PP_{(x,y)}$,

but do **not** adjust $PP_{(x,y)}$ if the resulting value would be outside the range $PP_{min} \leq PP_{(x,y)} \leq PP_{max}$

The $PP_{(x,y)}$ are upper-bounded by PP_{max}, to ensure that the level of proxying does not get so high that it takes a long time for the system to react to a reduction in home node contention. If there was no upper limit on $PP_{(x,y)}$, then at points in the algorithm where there was a concentration of home node contention, such as the acquisition of pivot row data in GE, the $PP_{(x,y)}$ value would go so high that subsequent reads requests from client x to home node y would be proxied unnecessarily. The choice of suitable values for PP_{max} and PP_{unit} depends on the architecture, and the values used in the simulations were selected after experiments with a range of values.

To decide whether proxying is appropriate, there has to be an extra check before each `read-request` is issued by a client x to a home node y. The test is as follows:

if $[LB_{(x,y)} > 0]$ **and** $[(PP_{(x,y)} \times PP_{unit}) > (T_{curr} - LB_{(x,y)})]$

then send a `proxy-read-request`,

otherwise send a normal `read-request`.

The adaptive proxies scheme is more flexible than reactive proxies, because it adjusts according to the level of congestion at individual home nodes. However it has the storage overheads of holding the $LB_{(x,y)}$, $PP_{(x,y)}$, PP_{unit}, PP_{max}, and PP_{min} values at each node. There are also the processing overheads of the tests to adjust $PP_{(x,y)}$, and checking before issuing each remote `read-request` whether a `proxy-read-request` should be sent instead.

5.4 Potential Benefits and Costs of Automatic Proxying

The two schemes for triggering proxying in response to run-time message buffer congestion should have a number of effects. Among the benefits one would expect are:

No marking of widely-shared data: the new strategies avoid the need for the programmer to correctly identify all the widely-shared data structures.

Proxies are only used during buffer congestion: the new proxy strategies only kick-in when incoming message buffers are full. At other times during the execution of an application the `read-request` messages are sent directly to the home node.

The potential costs are:

Delay in deploying proxies: the `proxy-read-request` is only made after a NAK is received. This delay of two messages (`read-request` and NAK) may outweigh any small performance gain to be had from sending the request via a proxy. The problem is less acute for adaptive proxies, because once a proxying-period has been triggered the subsequent `read-request` messages for the congested home node are automatically converted to `proxy-read-request` messages.

Unnecessary use of proxies: the congestion may have eased at the home node by the time a NAK is received at the requester. In these circumstances, the requester could get a faster response by re-sending the `read-request` to the home node rather than sending a `proxy-read-request`.

Moving rather than alleviating congestion: the use of proxies may just move the problem of full buffers from the home node to the proxy node(s). Using a different proxy for successive data lines, and partitioning the system into more than one proxy cluster (*i.e.* $\mathcal{NPC} > 1$) should help to avoid the problem. However, applications with a high level of messages might still encounter performance degradation because of re-directed messages.

These potential costs and benefits are considered as part of analysing the results in Section 5.7

5.5 Design Issues

There are a number of factors which have to be considered as part of implementing the reactive and adaptive proxy schemes. The most significant are how to represent finite buffers, and how to handle the messages which are "bounced" when a message buffer is full. In addition, the adaptive scheme requires that historical data about such buffer-bounces is recorded and used by node controllers. These issues are discussed in this section.

5.5.1 Finite Buffers

In this work, to simulate finite size input message buffers, the incoming message queue at each node is limited to eight for `read-request` messages. There may be more messages in an input buffer, but once the queue length has risen above eight, all `read-request` messages will be bounced back to the sender until the queue length has fallen below the limit. This is done because this research is concerned with the effect of finite buffering on read requests rather than all messages, and the approach also ensures that all transactions will complete in the protocol. The implementation in effect splits `read-request` messages away from other messages, in the spirit of split read-response networks (*e.g.* as used in Stanford's DASH project [89]), and curtails the number of reads. The limit of eight may seem low, but it was chosen to reflect the limitations in queue length that one would expect in large cc-NUMA configurations. The queue length of $\sqrt{P}$, where P is the number of processing nodes, is an arbitrary but reasonable limit.

5.5.2 Handling Bounced Messages

When the reactive and adaptive proxy protocols are in use, the return of a "buffer-bounced" NAK of a `read-request` will trigger a proxy read. For reactive proxies, a read request is only sent to a proxy in direct response to the arrival of a NAK. This is implemented by a straightforward modification to the client state machine: the arrival of a `read-request` NAK will now result in sending a `proxy-read-request` rather than repeating the `read-request`.

In addition, for adaptive proxies, proxy reads will be done instead of direct `read-request` messages to a home node when there has recently been a buffer-bounce from that home node. Two changes are needed to the node controller processing to support this. When a NAK is received, the value of $PP_{(x,y)}$ needs to be adjusted. In addition, the controller needs to check before any `read-request` message is issued to see whether a proxying period is in effect for the home node: if it is, then a `proxy-read-request` message has to be sent instead.

5.5.3 Adaptive Proxy Data

Every node controller needs to keep track of the last time a `read-request` was buffer-bounced back to it from each of the other nodes in the system. This last buffer-bounce information ($LB_{(x,y)}$) has to be updated whenever a `buffer-bounced-read-request` message is received. This information is used, in conjunction with $PP_{(x,y)}$ (the current proxy period for each node) to decide whether a `proxy-read-request` should be sent instead of a `read-request`. $PP_{(x,y)}$ is updated whenever a `buffer-bounced-read-request` is received.

The minimum proxy-period PP_{min} could be set to zero, which would initialise the algorithm to behave like reactive proxies when a buffer-bounce is first received. However for this work PP_{min} was initialised to one, so the receipt of a `buffer-bounced-read-request` always triggers a period of proxying. This was done in order to distinguish clearly between the effects of the adaptive and reactive proxy schemes..

The choice of suitable values for PP_{max} and PP_{unit} depends on the architecture, and the values used in the simulations were selected after experiments with a range of values. Setting too large a range between PP_{min} and PP_{max}, or a high value of PP_{unit} results in proxying being over-used when a home node is no longer congested, because the algorithm takes too long in adjusting $PP_{(x,y)}$ down to PP_{min}. However, setting PP_{max} at too low a value prevents proxying from being used for as long as it should be when a home node is congested over a long period. After experiments with a range of values (both higher and lower than those chosen to produce the results presented in this chapter) the PP_{max} and PP_{unit} parameters were set to 50 and 1000 respectively, *i.e.* PP_{unit} is one unit of proxy-period time, and is set to 1000 node controller cycles, and PP_{max} is the maximum proxy-period, and is set to 50 units. These values produced balanced performance results for the eight benchmark programs using adaptive proxies. The performance balance was between making the maximum use of proxies for applications such as GE (*i.e.* with higher values of both PP_{max} and PP_{unit}), and not over-using proxies for applications such as Ocean-Contig where the performance degrades for larger values of PP_{max} or PP_{unit}.

5.6 Modifications to the Protocol and Architecture

The architecture needed to support reactive proxies is the same as for basic proxies. However, adaptive proxies need to keep track of the timestamp of the last NAK'd `read-request` received from each of the other nodes in the system, and the current value of $PP_{(x,y)}$ for each of the other nodes. These values are held in the Adaptive Proxy Table, which is part of the node controller, as illustrated in Figure 5.3.

There are no extra node controller states or state transitions needed to support reactive and adaptive proxies, *i.e.* the state machine is as for basic proxies. However the processing of local read misses changes for adaptive proxies because there has to be a check on whether the proxying period is in effect. In addition, the introduction of finite buffers requires the node controller to handle the receipt of `buffer-bounced-read-request` messages: for basic proxies the will result in the `read-request` being re-sent to the home node, whereas for the automatic proxying schemes this is the trigger for sending a `proxy-read-request` instead.

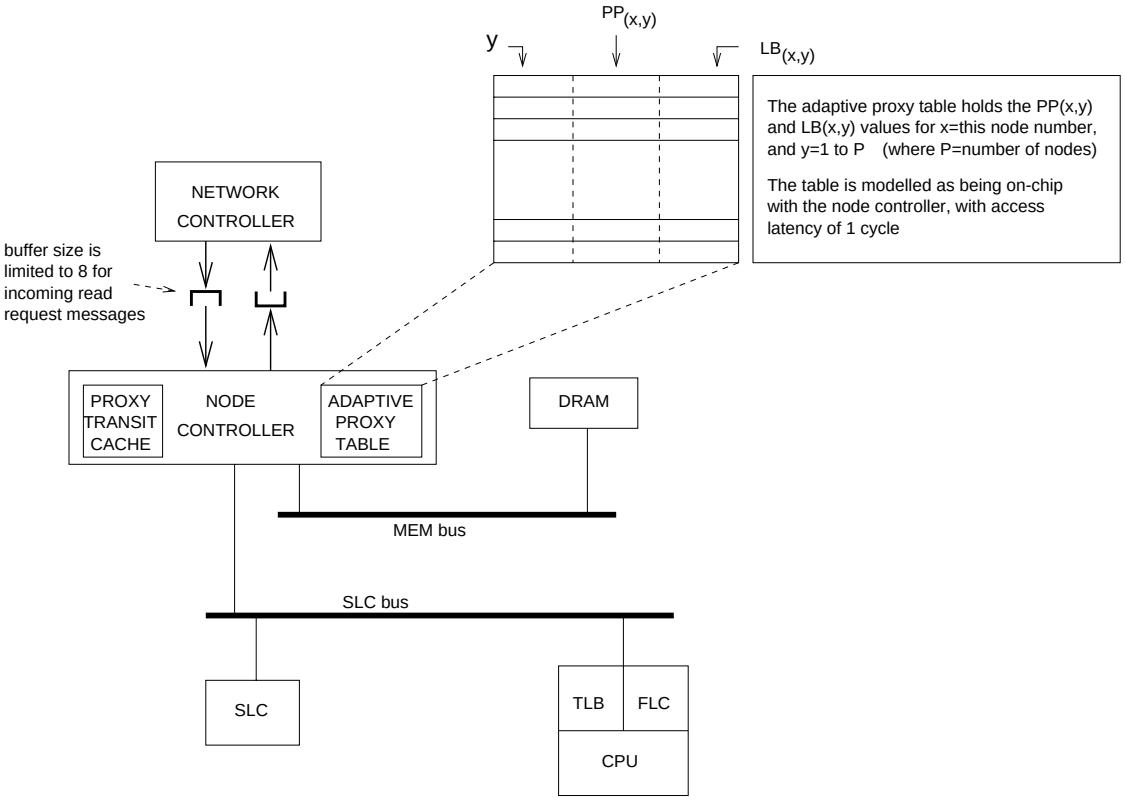


Figure 5.3: Memory model for a cc-NUMA node with finite message buffers

5.7 Results

This section presents the results obtained from execution-driven simulations of the basic, reactive, and adaptive proxy strategies. It has been shown in Section 3.5 that contention only becomes an important issue when more than a few tens of nodes are used. For this reason the detailed results presented below are from simulations of a 64 node design. For details of the simulated architecture, please refer back to Section 3.4. For basic proxies, the shared data marked for proxying is shown in Table 5.1: the same marking was used in Chapter 4.

application	problem size	shared data marked for basic proxying
Barnes	16K particles	all
CFD	64×64 grid	all
FFT	64K points	all
FMM	8K particles	`f_array` (part of `G_Memory`)
GE	512×512 matrix	entire matrix
Ocean-Contig	258×258 ocean	`q_multi` and `rhs_multi`
Ocean-Non-Contig	258×258 ocean	`fields`, `fields2`, `wrk`, and `frcng`
Water-Nsq	512 molecules	`VAR` and `PFORCES`

Table 5.1: Benchmark problem sizes, and data marked for basic proxies

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	+0.2	0.0	0.0	+0.2	-0.1	-0.3	0.0	-0.1
		reactive	+0.4	+3.3	+0.2	0.0	+0.3	0.0	+0.5	+0.4
		adaptive	+0.1	+3.2	+0.4	+0.4	+0.4	+0.2	-0.1	+0.2
CFD	28.3	basic	+10.4	+11.3	+8.1	+11.8	+9.8	+9.3	+9.2	+14.8
		reactive	+7.6	+7.4	+8.3	+7.6	+8.6	+7.6	+7.6	+6.1
		adaptive	+9.2	+13.1	+11.3	+11.6	+11.2	+10.4	+10.6	+12.1
FFT	47.3	basic	+9.4	+8.7	+8.7	+9.6	+9.5	+8.6	+10.0	+8.5
		reactive	+11.7	+11.2	+10.9	+11.0	+11.2	+11.8	+11.2	+10.7
		adaptive	+11.9	+11.6	+11.3	+11.4	+11.2	+11.5	+11.0	+11.0
FMM	52.4	basic	+0.4	+0.4	+0.5	+0.3	+0.4	+0.3	+0.3	+0.4
		reactive	+0.3	+0.4	+0.4	+0.4	+0.3	+0.4	+0.4	+0.4
		adaptive	+0.4	+0.4	+0.4	+0.4	+0.4	+0.5	+0.4	+0.4
GE	21.6	basic	+29.3	+29.3	+29.3	+29.4	+29.4	+29.5	+29.6	+29.6
		reactive	+28.4	+28.6	+28.9	+28.8	+28.8	+28.8	+28.7	+28.9
		adaptive	+30.5	+30.7	+31.4	+31.2	+31.7	+31.6	+31.4	+31.6
Ocean-Contig	49.7	basic	-2.6	-0.9	-1.1	-4.7	-2.1	+0.4	-5.4	+0.9
		reactive	-0.6	-4.4	+1.8	+3.3	-0.9	+2.5	+1.8	+2.6
		adaptive	-1.3	-2.8	-6.1	-3.5	-1.4	-3.6	-0.4	-3.6
Ocean-Non-Contig	48.2	basic	+5.3	0.0	+2.4	-0.7	+6.4	+1.8	+5.7	-1.2
		reactive	+5.8	+3.1	+2.0	+4.7	+4.0	-1.9	+2.5	+5.5
		adaptive	+7.8	+7.6	-6.3	+2.0	+4.1	+6.6	-8.3	-1.5
Water-Nsq	55.3	basic	-0.6	-0.6	-0.6	-0.6	-0.6	-0.5	-0.7	-0.5
		reactive	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

Table 5.2: Benchmark relative speedups for 64 processing nodes

The performance results for the eight benchmark applications are summarised in Table 5.2. The results are presented in terms of relative speedup, *i.e.* the ratio of the execution time for the fastest algorithm running on one processor to the execution time on $\mathcal{P}$ processors[1]. There is no overall "winner" among basic, reactive, and adaptive proxies: no policy improves the performance of all the applications for all values of the number of proxy clusters ($\mathcal{NPC}$). Looking at the results for different values of $\mathcal{NPC}$, for basic proxies there is no value which has a positive effect on the performance of all the benchmarks. However, for reactive proxies, there are values of $\mathcal{NPC}$ which improve the performance of all the benchmarks, *i.e.* $\mathcal{NPC}$=3,7&8 all achieve a balance between combining, the length of the proxy pending chains for the simulated system, and an even distribution of queue lengths. Reactive proxies may not always deliver the best possible performance improvement, but by providing stable points for $\mathcal{NPC}$ they may be of more use to system designers. Adaptive proxies usually perform as well as, and often better than, reactive proxies; this includes obtaining the best performance for GE. However adaptive proxies are "let down" by their performance with the Ocean-Contig application.

In the following sub-sections, the results for each of the applications are examined in more detail, in order to understand their different reactions to the use of proxies with finite buffering. For each application, the detailed results are presented as two graphs (one of relative changes

[1]It should be noted that the relative speedup and percentage change values for basic proxies differ from those presented in Chapter 4. This is because the results in Chapter 4 were from simulations with infinite buffers, as opposed to the current results where `read-request` messages are liable to be buffer-bounced.

and one of message ratios) and two histograms (one showing the execution time profile, and the other showing the message category profile).

The relative changes graphs show four different metrics:

messages: the ratio of the total number of messages to the total without proxies,

execution time: the ratio of the execution time (excluding startup) to the execution time (also excluding startup) without proxies,

queueing delay: the ratio of the total time that messages spend waiting for service to the total without proxies, and

remote read delay: the ratio of the mean delay between issuing a `read-request` and receiving the data, to the same mean delay when proxies are not used.

The message ratios are:

proxy hit rate: the ratio of the number of `proxy-read-request` messages which are serviced directly by the proxy node, to the total number of `proxy-read-request` messages (in contrast, a proxy miss would require the proxy to request the data from the home node),

buffer bounce ratio: the ratio of the total number of buffer bounce messages to read requests. This gives a measure of how much bouncing there is for an application. This ratio can go above one, since only the initial `read-request` is counted in that total (*i.e.* the retries are excluded), and

proxy read ratio: the ratio of the proxy read messages to read requests - this gives a measure of how much proxying is used in an application.

The execution time profile presents the overall execution time split into CPU active time and the time spent waiting because of delays. The delays are further split into load, store, barrier and lock delays. The times are normalised with respect to the execution time without proxies.

The message category profile shows how the total number of messages breaks down into four categories: read, write, unhook, and proxy messages. The allocation of message types to message categories is given in Appendix C.2, but it should be noted that read messages include all the `read-request`, `buffer-bounced-read-request`, and `take-shared` messages, write messages include all the `write-request`, `invalidate` and `take-exclusive` messages, and unhooks are all the messages needed to handle cache line replacements.

5.7.1 Barnes

For this application, the basic proxy protocol sometimes degrades the performance, whereas the reactive and adaptive proxy protocols usually improve the performance. The performance changes resulting from the use of proxies can, for this application, be as much from the effects of changes in barrier and lock delays than from the changes in load and store miss delays. Changing the balance of processing by routing read requests via proxy nodes can have more impact than the direct effects of reducing home node congestion. For example, when $\mathcal{NPC}$=2 for adaptive proxies, the store and load miss delays are the same as with no proxies, yet a reduction of 3.2% in barrier delay results in an overall performance improvement of 3.2% (see Figure 5.5).

The low level of buffer-bouncing shown in Figure 5.6 indicates that there is no need for a high degree of proxying in this application. However, all the shared-data was marked for basic proxies because there was no obvious data structure that would specifically benefit from proxying. As a result, there is a high proxy read ratio for basic proxies, but the proxy hit rate is not as good as that achieved by reactive and adaptive proxies (which are only triggered when buffer-bouncing occurs), and basic proxies have little effect on the level of buffer-bouncing. Taken together with the very low proxy read ratio for reactive and adaptive proxies (close to zero for $\mathcal{NPC}\geq 1$), this shows that there is very little reactive/adaptive proxying taking place in Barnes, but that which occurs is significant enough to affect the overall performance of the application.

For basic proxies, the remote read delay initially drops and then rises as $\mathcal{NPC}$ is increased up to 8. This is in line with the increase in messages (see Figure 5.4) which mainly occurs because the number of `read-request` messages increases. The steady rise in `read-request` messages is due to the increasing number of proxies needing copies of data. This leads in turn to an increase in the level of unhooking because of SLC conflicts (see Figure 5.7), which leads to later local SLC misses and an increase in the overall load miss delay, as shown in Figure 5.5. Given that all the shared data in Barnes is marked for basic proxying, this results in basic proxies having a higher load miss delay than with no, reactive, or adaptive proxies.

Reactive and adaptive proxies both have a high proxy hit rate (see Figure 5.6), *i.e.* when a proxy read is received at a node then with $\mathcal{NPC}$=1 there is $> 90\%$ chance that the data is already held at the proxy or has been requested. This suggests that the data which is subjected to reactive/adaptive proxies is widely-shared.

Unhooking delays resulting from proxy cache pollution can be significant for this application. The effect is most marked for adaptive proxies when $\mathcal{NPC}$=7, as shown in Figure 5.7. In this

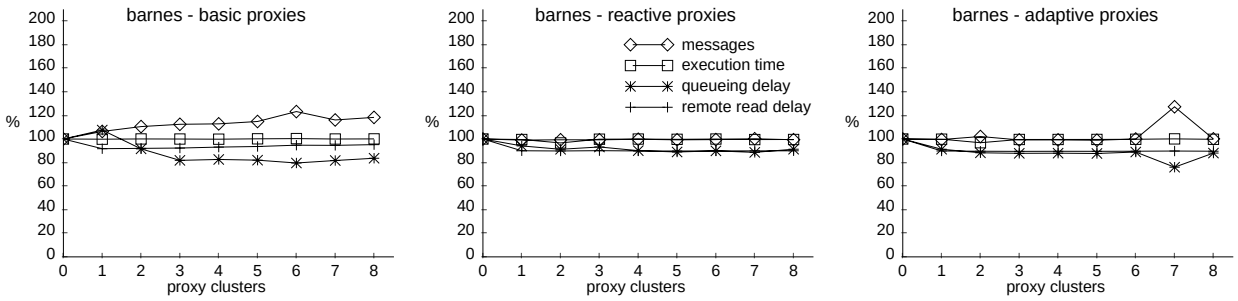


Figure 5.4: Barnes: changes (relative to no proxies case)

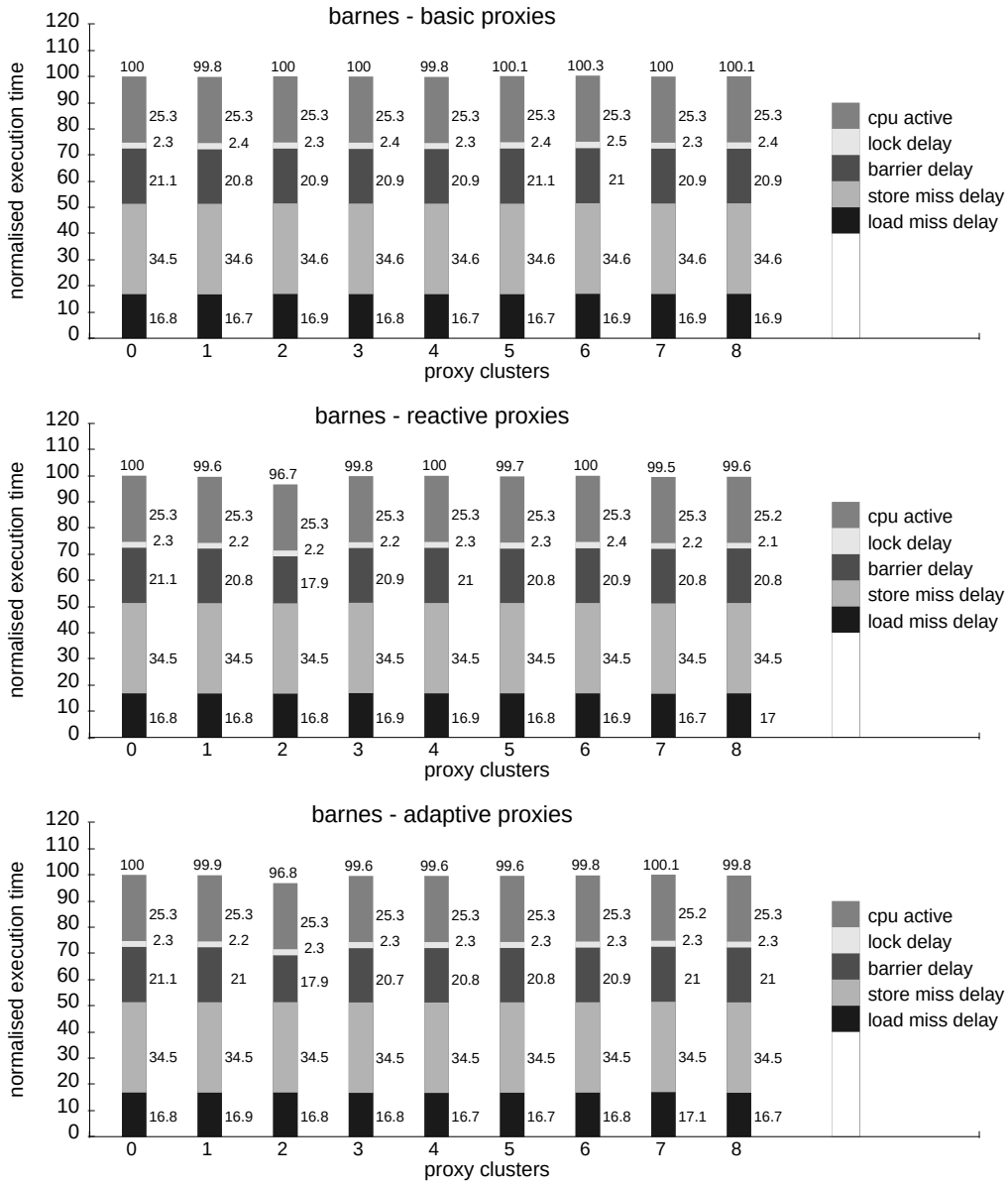


Figure 5.5: Barnes: execution time profiles

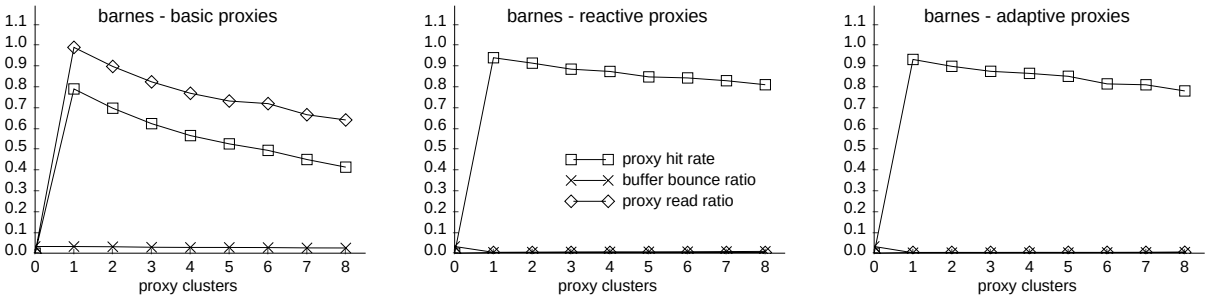


Figure 5.6: Barnes: message ratios

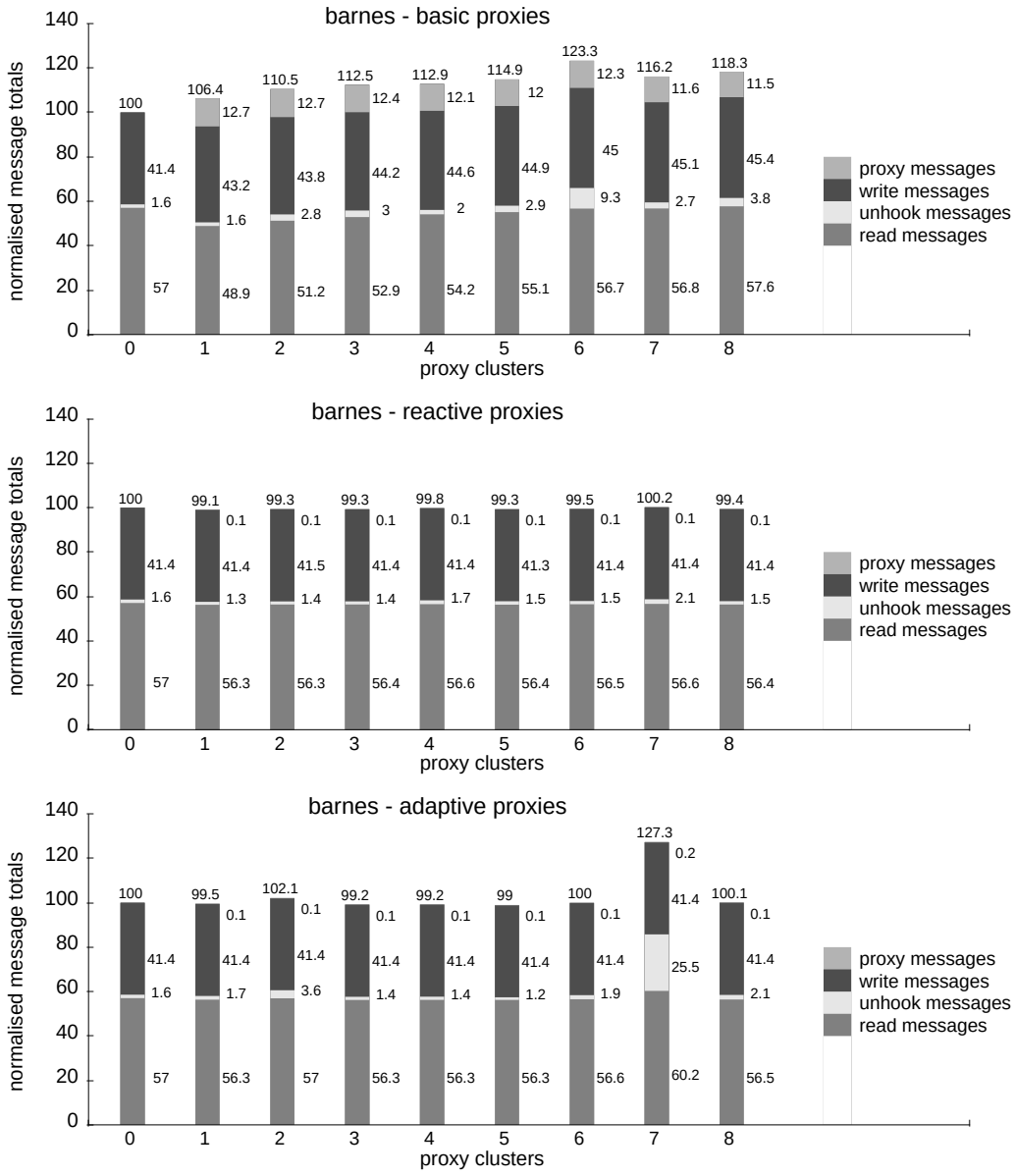


Figure 5.7: Barnes: message category profiles

case, unhooking increases dramatically because there is a conflict between proxy data lines. In each proxy cluster, one node (A) sends a `proxy-read-request` for data line X to node B, and at the same time node B sends a `proxy-read-request` for data line Y to node A. Data lines X and Y happen to map to the same SLC cache line, and so the `proxy-read-request` messages are bounced because each node has already reserved the cache line for its local processing. In an early implementation of the protocol this caused an infinite loop of requests and bouncing: the protocol was amended to break the cycle by giving up on the `proxy-read-request`, once it had bounced ten times, by sending a `read-request` directly to the home node. However the cache line conflict still causes a high level of unhooking because, for example, node A may obtain data line X directly from the home node, but then a `proxy-read-request` arrives from node B for data line Y, and this will cause X to be evicted from the SLC. The "frenzy" of unhook messages, resulting from this unfortunate correspondence between data usage by the algorithm and the partitioning into seven proxy clusters, increases the load miss delay and results in slightly worse overall performance. However the performance degradation is only -0.1% because the problem only occurs sporadically during the execution of the application, and the nodes which are not caught up in the SLC cache line conflict benefit from the lower queueing delays.

The application can benefit marginally from automatic proxying, but only if it is timely. The benefit can be lost with adaptive proxies, because in the time it takes for the client to react, send the `proxy-read-request`, and receive the data, the congestion may have eased at the home node; to continue proxying for the proxy period is counter-productive. Reactive proxies achieve the most stable performance, which is never worse and often better than without proxies, because they avoid the proxying of too many data structures seen with basic proxies, and they do not suffer from the unhooking conflict seen for $\mathcal{NPC}$=7 with adaptive proxies.

5.7.2 CFD

All three forms of proxying improve the performance of the CFD application. Adaptive and basic proxies obtain the best performance. Reactive proxies, although successful in improving the performance over no proxies, generally have a lower benefit. This reflects their "on demand" strategy, *i.e.* a proxy read is only used in direct response to a NAK'd read request, rather than all the read requests being proxied (basic proxies) or proxying for a proxy period after a NAK (adaptive proxies).

As was observed in Section 4.4.2, Figure 5.10 shows that this application has a "stride" in proxy hit rate for values of $\mathcal{NPC}$ which are powers of 2. This is because the proxy tends to already be using the data (as a result of the algorithm and problem size). Basic proxies

display the same knock-on effect on the performance as was seen with infinite buffers, but reactive and adaptive proxies do not have the same performance variations, as is shown in Figure 5.8. The basic proxy protocol has its highest level of combining and lowest number of `read-request-fwd` messages from home to owner nodes when $\mathcal{NPC}$=4&8. Although adaptive proxies show a similar proxy hit rate, proxies are used for considerably fewer read requests, as shown by the lower proxy read ratio in Figure 5.10. As a result, the adaptive strategy does not get the same level of reduction in `read-request-fwd` messages, and hence does not get such large improvements in remote read delay.

For this application, reactive proxies make relatively little use of proxies, as shown by the proxy read ratio and proxy message category profile in Figures 5.10 and 5.11. Although proxies are not often used, the technique results in $> 40\%$ drop in the remote read delay, and this has the effect of reducing the overall load miss delay (see Figure 5.9).

Adaptive proxies make greater use of `proxy-read-request` messages than reactive proxies (as shown by the proxy read ratios in Figure 5.10), and get an even better reduction in remote read delay. Although, as shown in Figure 5.11, the proxy and invalidation messages increase, there is a decrease in read messages which results in a reduction of both the overall queueing delay and the load miss delay.

The buffer bounce ratio, as shown in Figure 5.10, is affected by all three types of proxying, but it oscillates rather than falls away. In fact the absolute number of `buffer-bounced-read-request` messages falls by up to 50% with proxies, but because the number of `read-request` messages also falls (replaced by `proxy-read-request` messages) the buffer-bounce ratio is unaffected. The drop in read category messages is shown in Figure 5.11.

On balance, although the best performance is obtained by basic proxies when $\mathcal{NPC}$=8, the adaptive proxies scheme gives the most reliable performance improvement for this application, and should not be so vulnerable to variations in performance resulting from particular combinations of problem size, cache line size, and $\mathcal{NPC}$ value.

5.7.3 FFT

This application shows a performance improvement with all three forms of proxying. There is a high level of buffer-bouncing without proxies: this is shown by the buffer bounce ratio, which is 70% without proxies, *i.e.* on average for every 100 `read-request` messages there are 70 NAK's. This ping-ponging effect (NAK followed by retrying the read) will continue until there is room for the `read-request` in the input message buffer at the home node. When proxies are introduced, the buffer bounce ratio falls dramatically, but differently, for each of

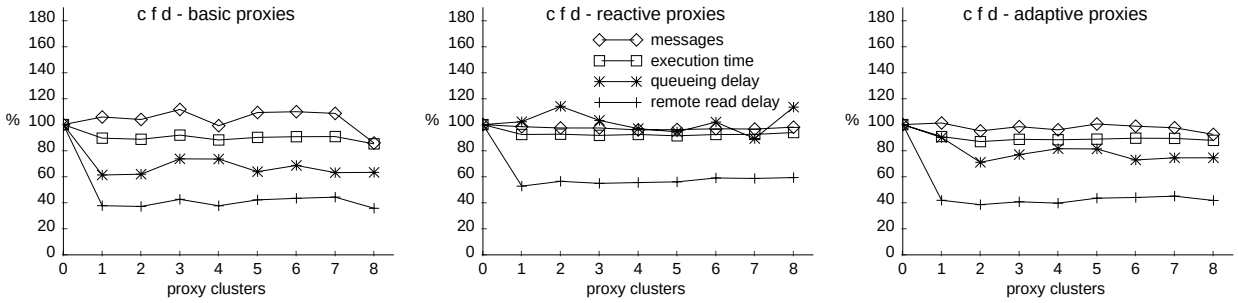


Figure 5.8: CFD: changes (relative to no proxies case)

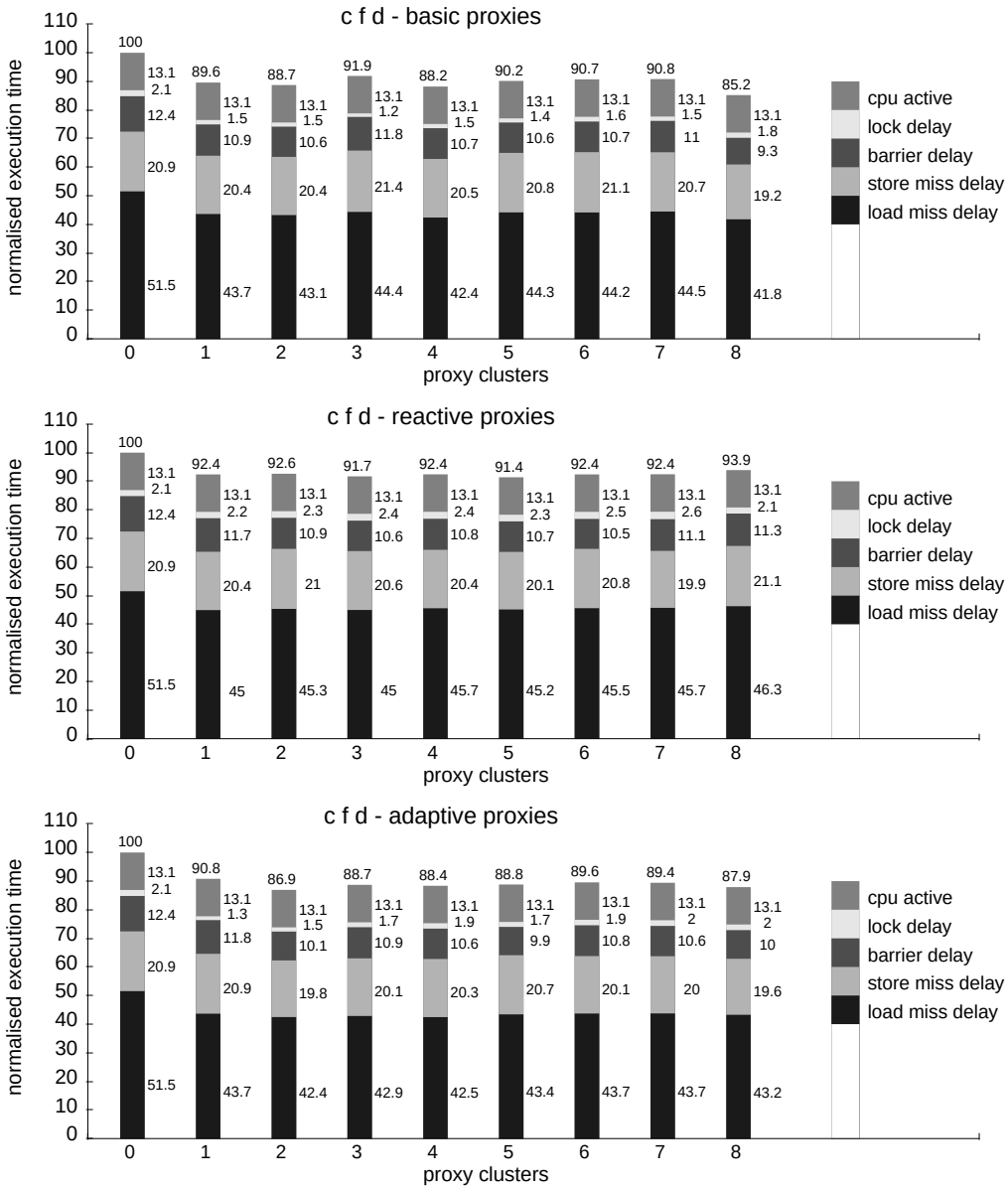


Figure 5.9: CFD: execution time profiles

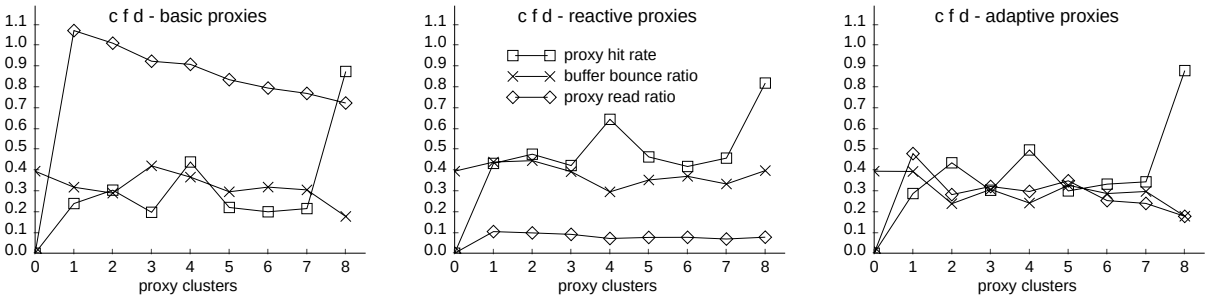


Figure 5.10: CFD: message ratios

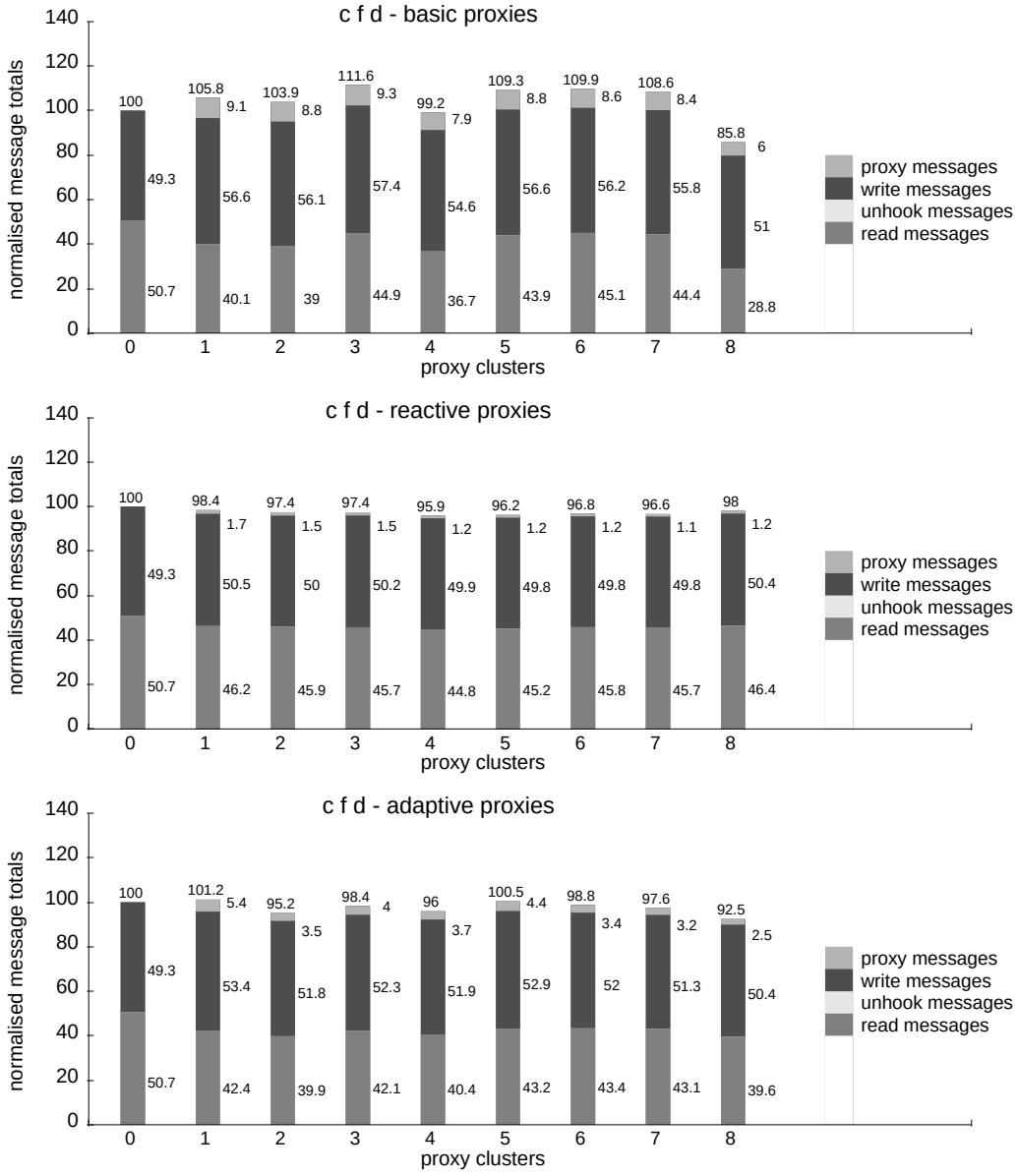


Figure 5.11: CFD: message category profiles

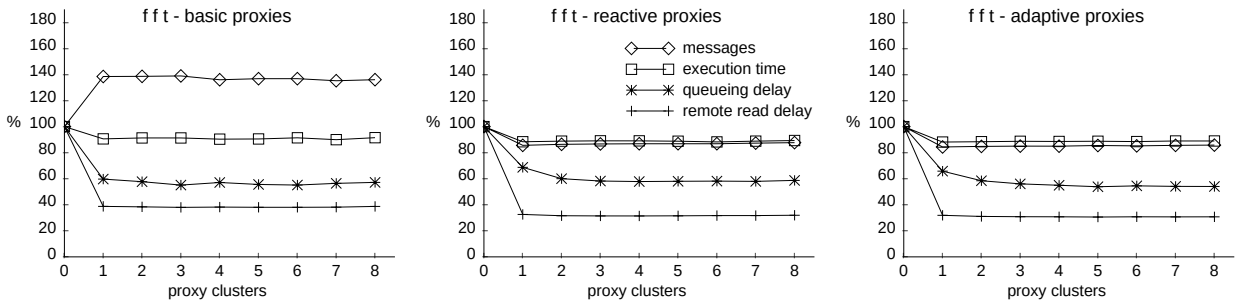


Figure 5.12: FFT: changes (relative to no proxies case)

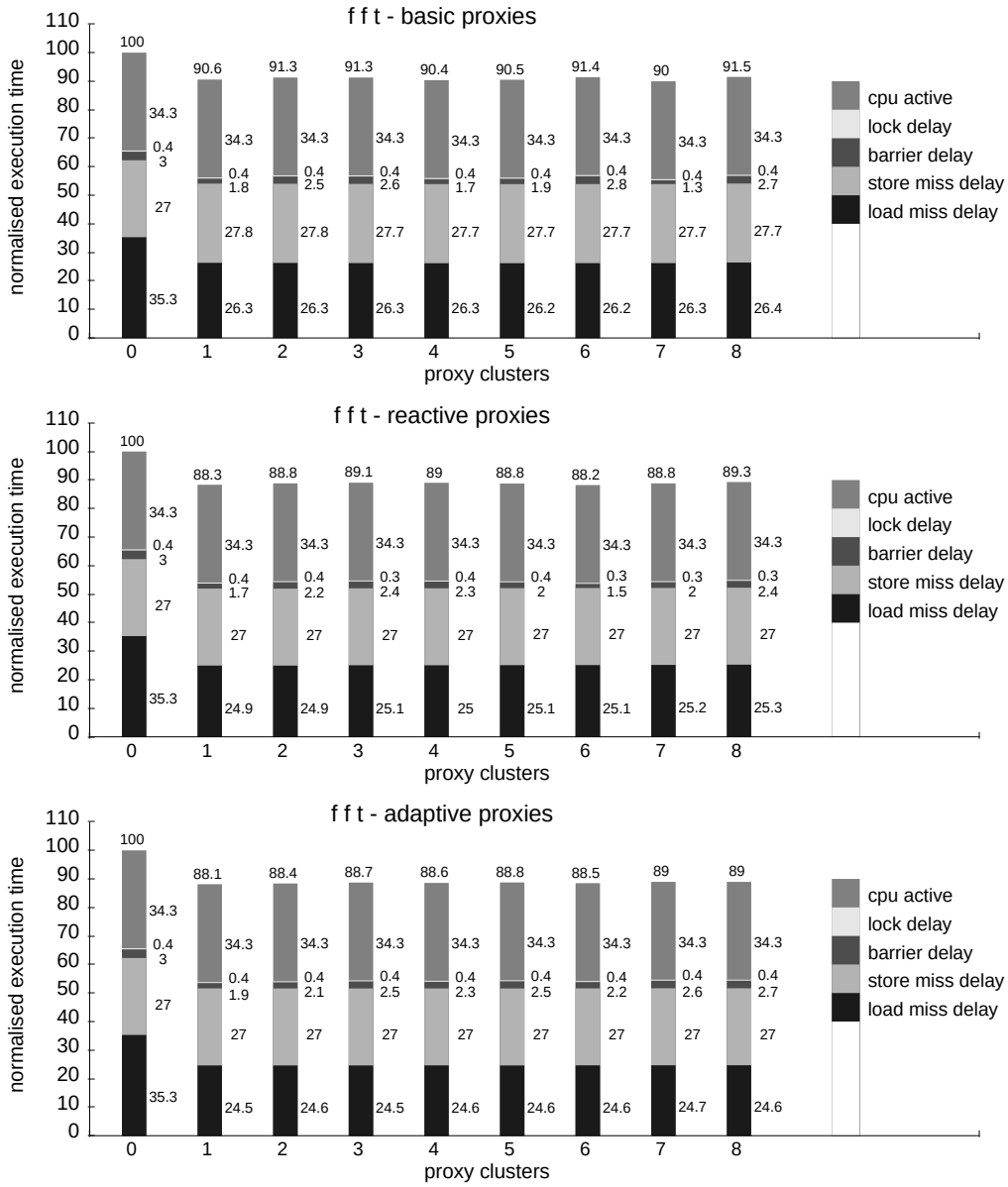


Figure 5.13: FFT: execution time profiles

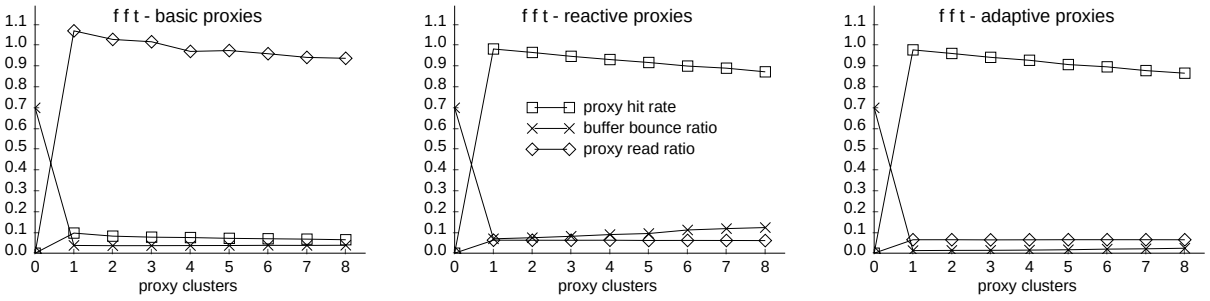


Figure 5.14: FFT: message ratios

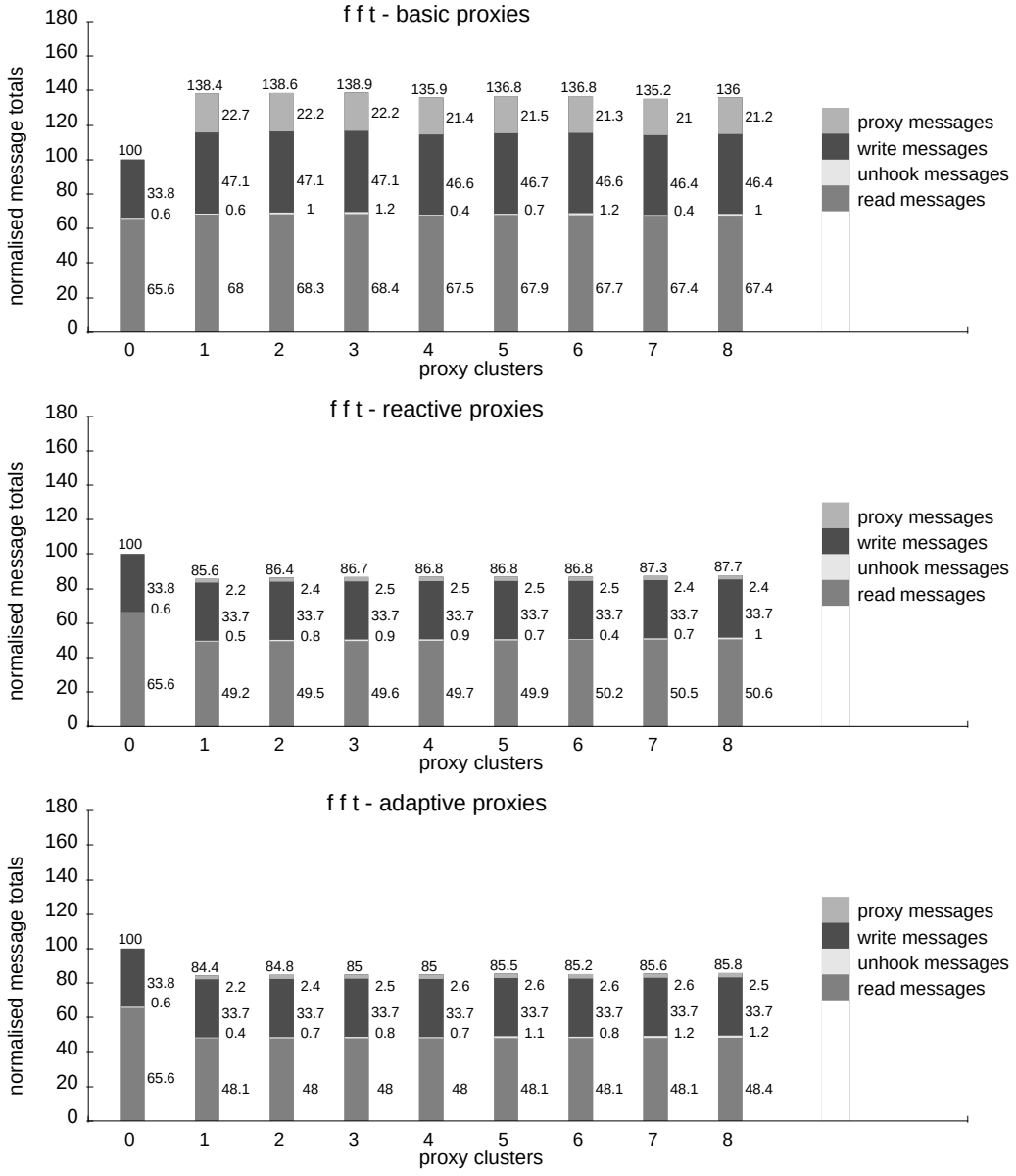


Figure 5.15: FFT: message category profiles

the three proxy strategies.

The basic proxy scheme, with all the shared-data marked for proxying, gets the least performance benefit: this is the result of its "overuse" of proxies which leads to a low rate of combining and a large increase in the number of messages. The technique also leads to a worsening of the store miss delay mainly because of the additional invalidation messages needed to remove proxy copies before writes can proceed. However, the technique is still very effective in reducing queueing delays, and the resulting improvements in load miss delay lead to performance gains of between 8.5% and 10.0% depending on the value of $\mathcal{NPC}$.

The more selective use of proxies under the reactive and adaptive schemes results in greater reductions in load miss delay and has no adverse effect on store miss delay (see Figure 5.13). Adaptive proxies are able to reduce the load miss delay further than reactive proxies because they are more successful at avoiding NAK'd read requests. This is because once a home node NAK triggers proxying, all further `read-request` messages to that home node are changed to `proxy-read-request` messages during the proxying period. As a result, there are fewer `read-request` messages and NAKs, and this is reflected in the lower level of read messages shown in Figure 5.15. The high proxy hit rate for reactive and adaptive proxies (see Figure 5.14) confirms that for this application it is a good idea to target the proxy reads at the data experiencing home node congestion, rather than marking all data for proxying.

The FFT application shows that an application which benefits from the use of basic proxies can obtain even better performance when proxying is triggered by run-time congestion. In addition, the adaptive proxy approach generally gets the best performance by continuing to use proxies during the proxying period after a triggering NAK.

5.7.4 FMM

All three forms of proxying improve the performance of FMM, and reduce both the number of messages and the overall queueing delay. However, FMM has a marginal speedup compared with no proxies (between 0.3% and 0.5%). This is as expected given that only the `f_array` data structure (part of `G_Memory`) is widely-shared, which was why it was marked for basic proxies. However, the performance improvement with the automatic proxying schemes is comparable to that achieved using basic proxies, so the new methods are able to detect dynamically the opportunities for read combining without the need for code inspection and/or profiling tools. The proxy read ratios (see Figure 5.18) show that reactive and adaptive proxies make slightly more use of proxy reads than does the basic proxy strategy. This increased use of proxies leads to a marginally greater decrease in load miss delay with the two automatic schemes.

The adaptive proxy strategy usually gives better performance than the reactive scheme for FMM, reflecting its ability to be best at reducing buffer bouncing, remote read delay, and overall load miss delay for this application, by continuing to use proxies for read requests during the proxying period.

5.7.5 GE

This application benefits from the use of any of the three proxy strategies: the best performance gains are with adaptive proxies (around 31.0%), followed by basic proxies (up to 29.6%), with reactive proxies getting performance improvements of up to 28.9%. It was to be expected that the performance benefit using reactive proxies would not be as good as that obtained using basic proxies, because the proxying is no longer targeted by marking the widely-shared data structure. Instead proxying is triggered when a read is rejected because a buffer is full, and so there will be two messages (the read and its NAK) before a `proxy-read-request` is sent by the client. The delay incurred by the first two messages means that the performance improvement is not as good as can be obtained with basic proxies, and this is confirmed by the execution time profiles in Figure 5.21. Adaptive proxies, however, are able to get better performance improvements than basic proxies because once congestion triggers a proxy read at a client, all further read requests for that home node made by the client during the proxying period will be replaced by `proxy-read-request` messages. This strategy is eminently suited to pivot row acquisition, and avoids unnecessarily proxying the matrix data at other points during the execution.

Using basic proxies reduces the remote read delay to around 10% of the no proxy level. By marking all of the matrix for proxying, nearly all the `read-request` messages are converted into `proxy-read-request` messages. As a result these requests go via a proxy, and avoid the home node's full buffer. The level of read messages (including NAKs) drops dramatically, and despite the introduction of proxy messages the overall number of messages falls by around 35% (see Figure 5.23). It should be noted that the number of write messages increases with basic proxies because of the extra invalidation messages needed to remove proxy copies from sharing lists.

Reactive proxies also reduce the remote read delay, but not by as much as is seen with basic proxies. This is because there is the initial delay of sending a `read-request` to the home node and receiving its NAK before a `proxy-read-request` is sent instead. In addition, the overall queueing delay initially increases with reactive proxies. This increase is due to a much longer queueing delay for one node, which now receives `proxy-read-request` messages for the global data field "`el`" which all nodes read once every iteration. The node (which happens to be

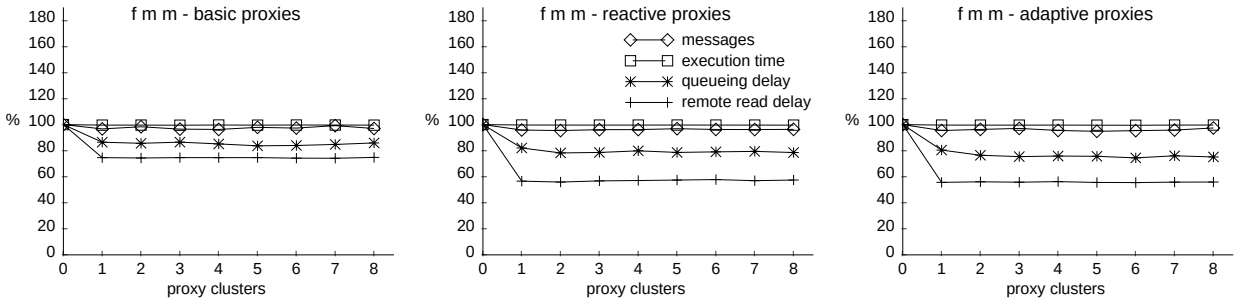


Figure 5.16: FMM: changes (relative to no proxies case)

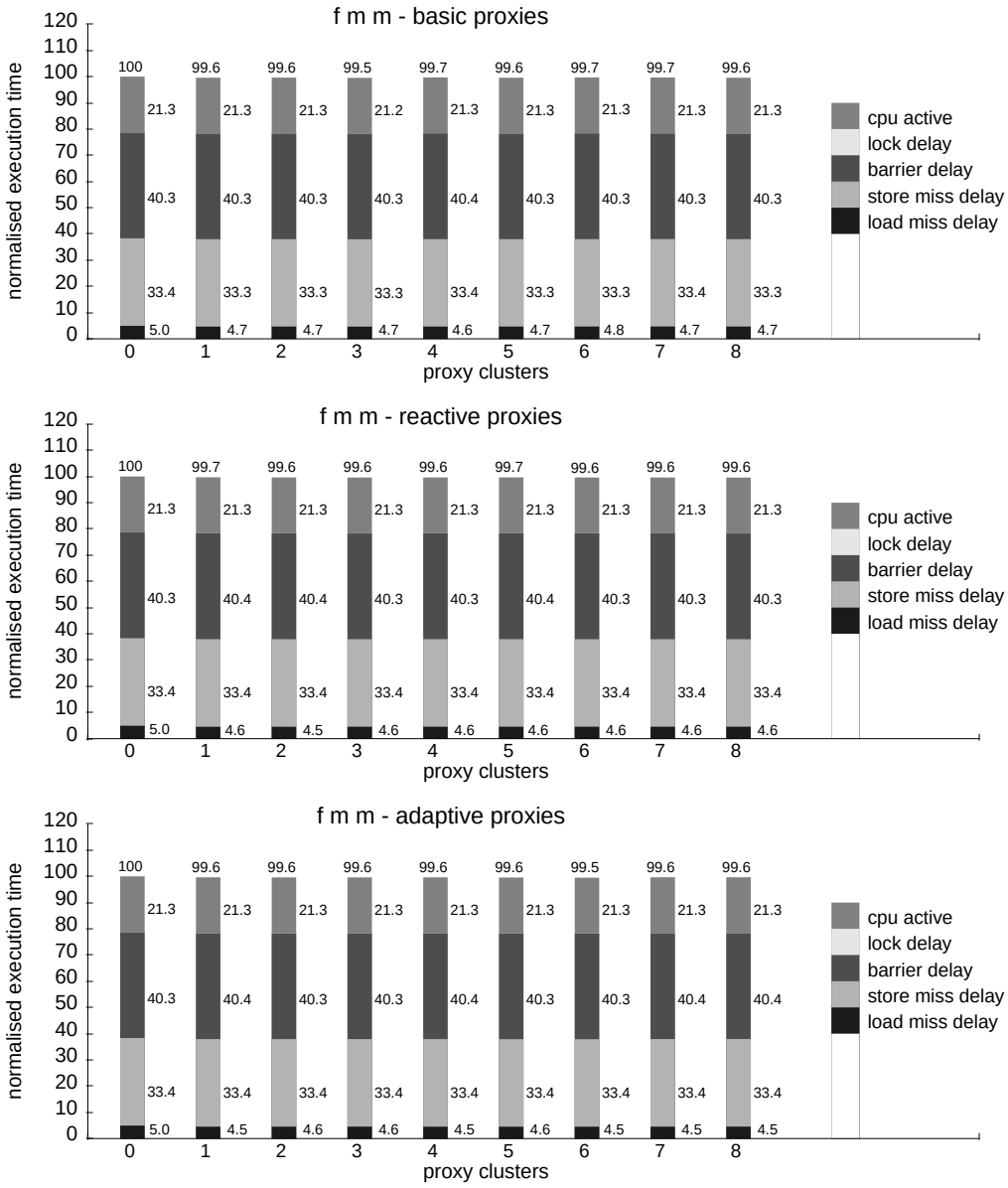


Figure 5.17: FMM: execution time profiles

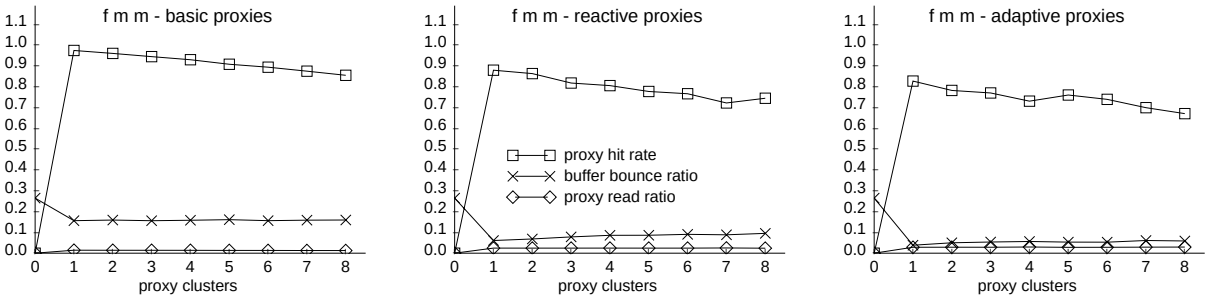


Figure 5.18: FMM: message ratios

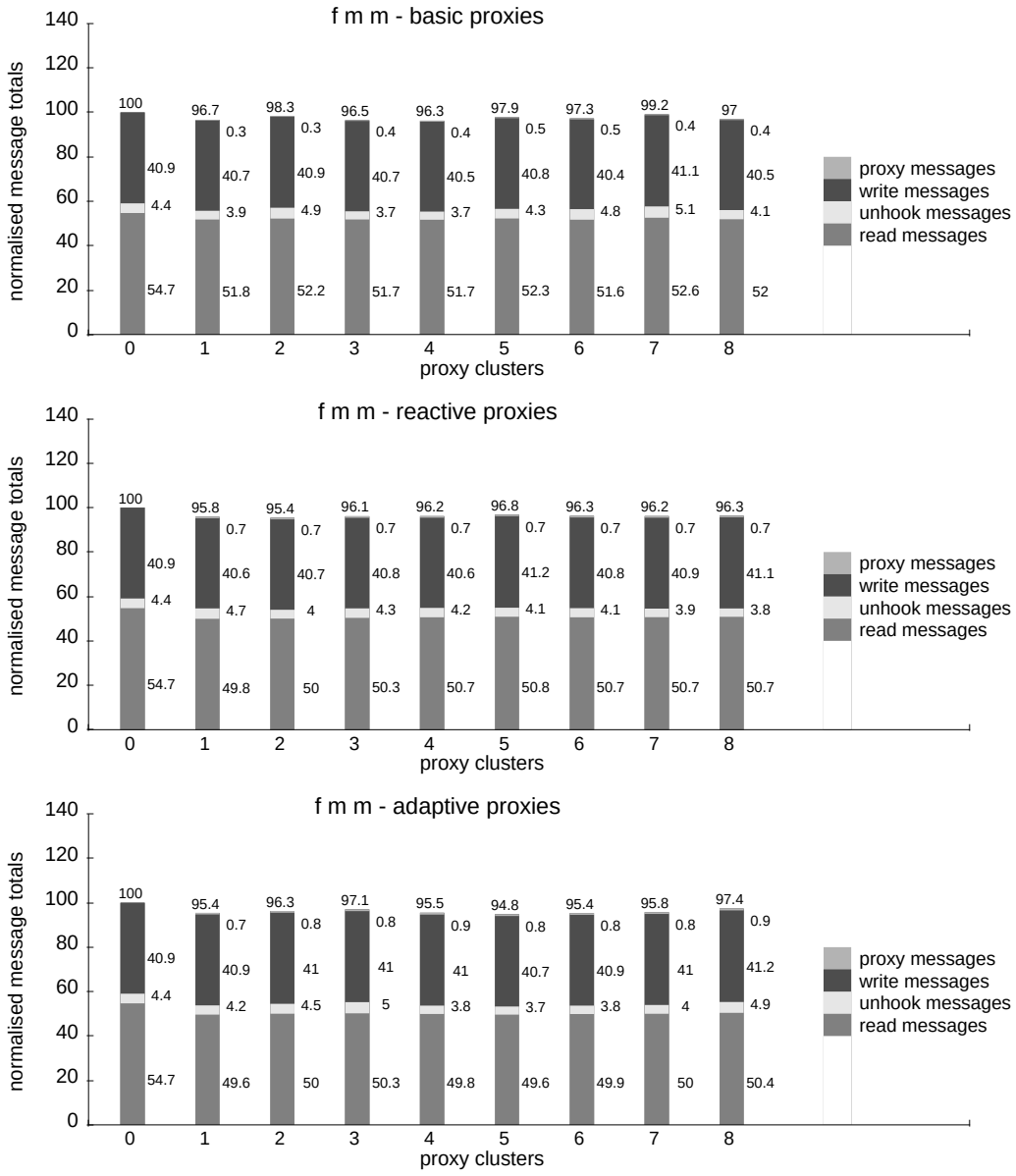


Figure 5.19: FMM: message category profiles

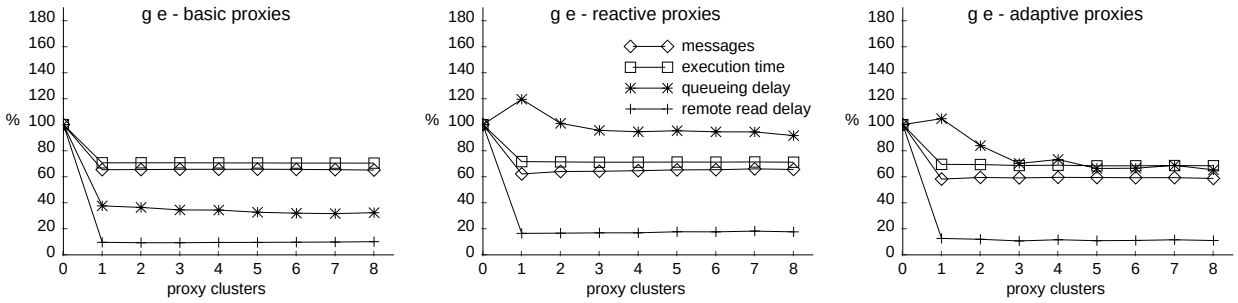


Figure 5.20: GE: changes (relative to no proxies case)

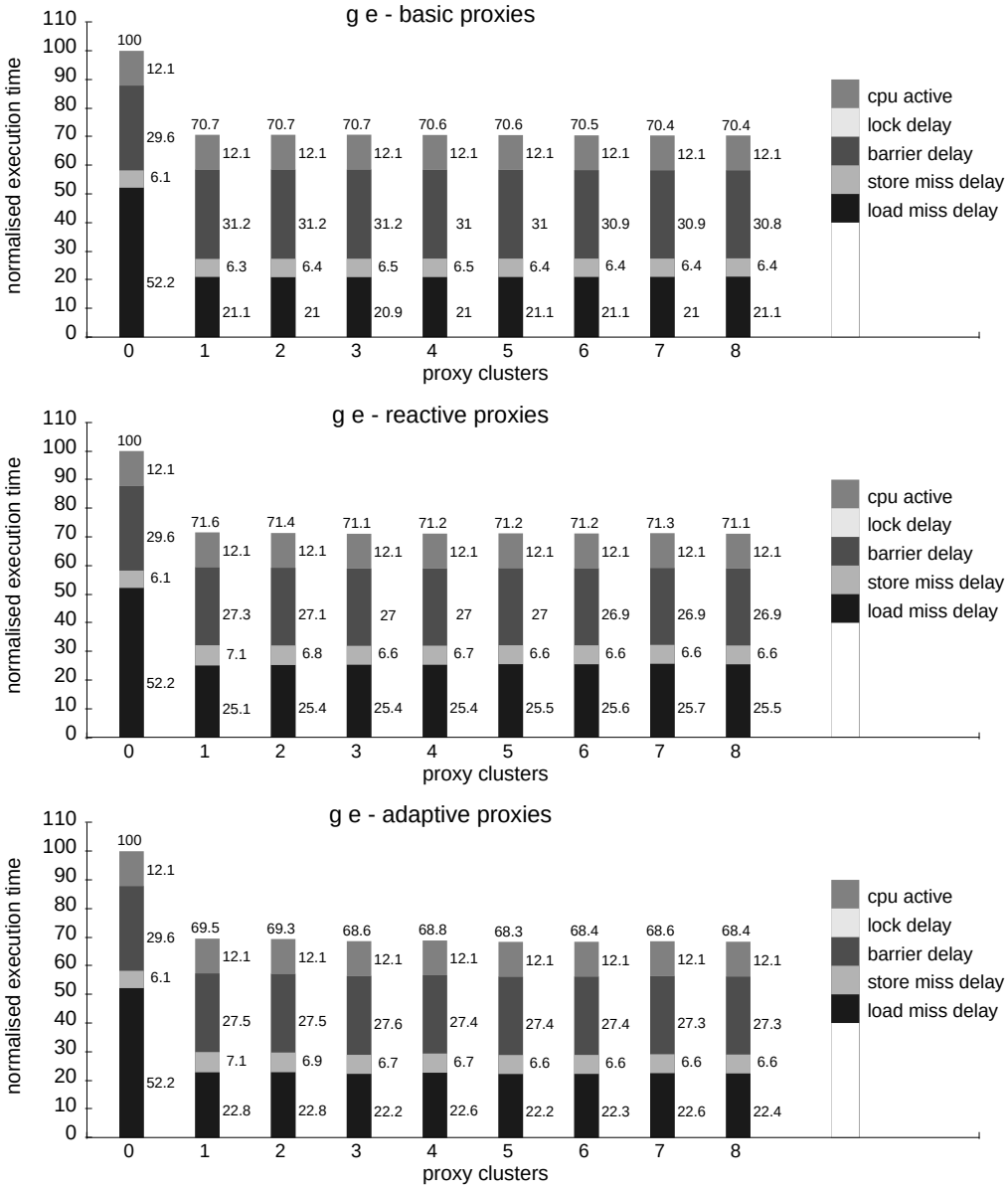


Figure 5.21: GE: execution time profiles

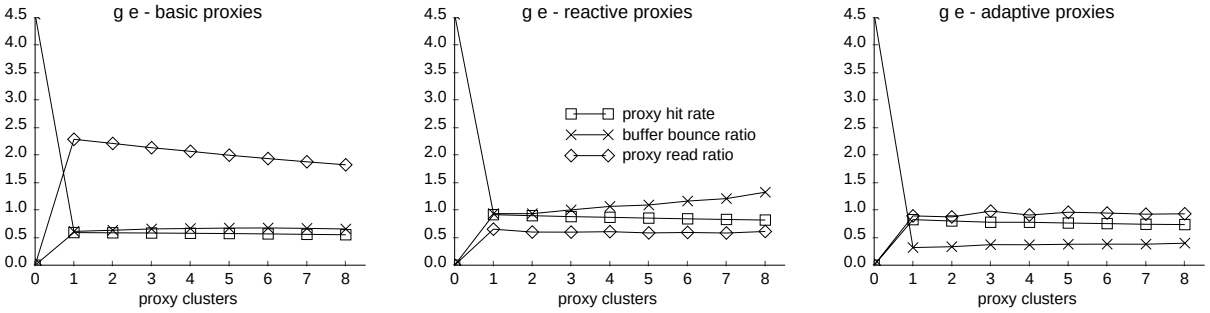


Figure 5.22: GE: message ratios

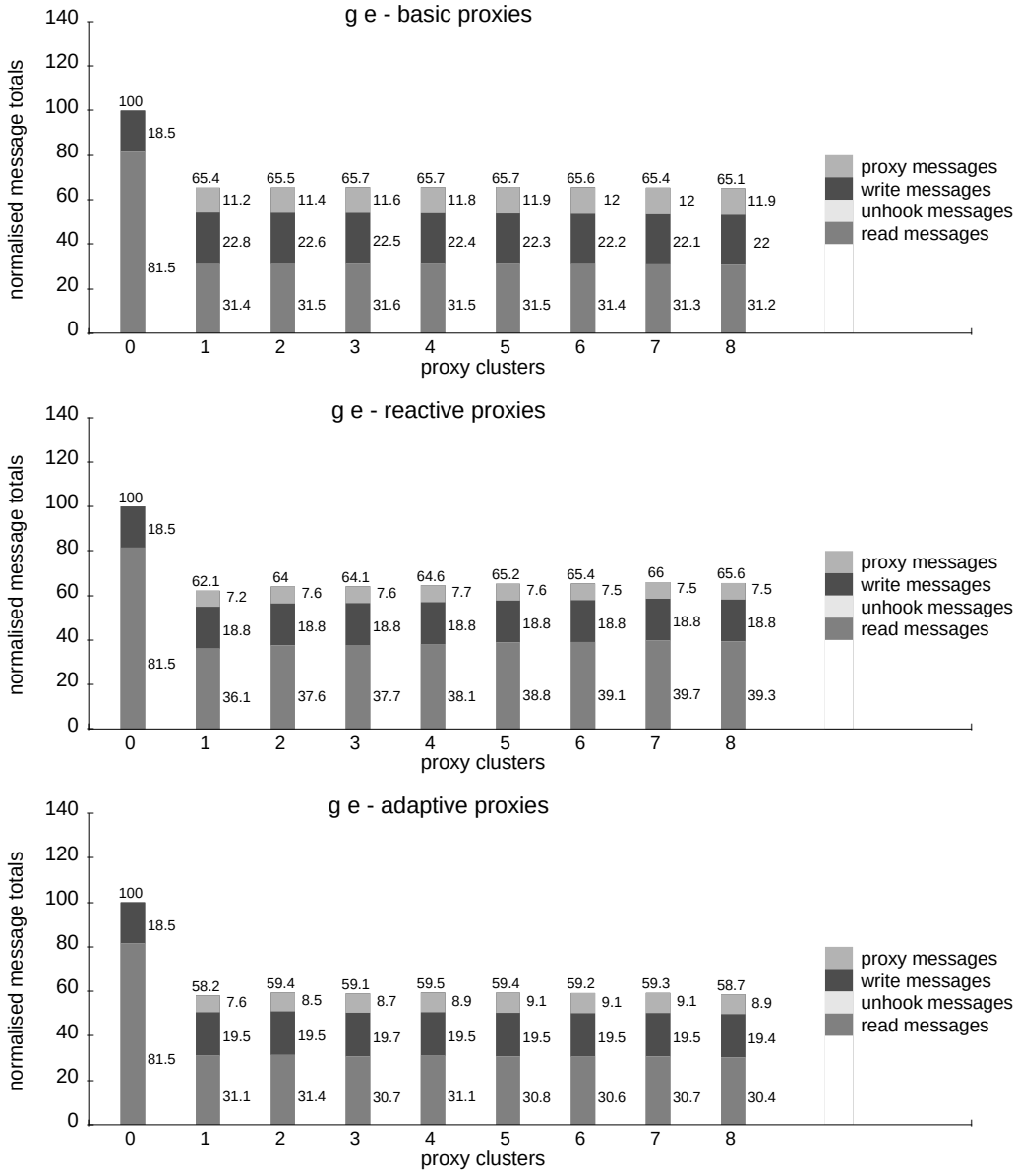


Figure 5.23: GE: message category profiles

running the master process) is also the home node to about 25% of the data pages containing the main matrix, and is often owner for other lines, and as a result its input message queue is very heavily used. Despite this bottleneck, the overall remote read delay and load miss delay drop, but the store miss delay increases. However the overall improvement in performance with reactive proxies shows the benefits for this application of spreading the `read-request` load around the system. In addition, the reactive scheme achieves a better level of combining then the basic scheme, as shown by the proxy hit rate in Figure 5.22, and this indicates that the proxying is being targeted at the widely-shared data.

With adaptive proxies, there is a similar drop in service for write messages with an increase in store miss delay, once again due to one of the nodes having a longer mean queueing delay than the other nodes. However the queueing delay is not as badly affected as for reactive proxies because read requests for matrix data are automatically converted into `proxy-read-request` messages during the proxying period and there is a high level of combining at the proxies which relieves some of the queueing pressure on the bottleneck node. These results highlight the complicated balance between re-distributing the messages to avoid home node hot-spots, and not overloading the other nodes to the detriment of their normal processing.

For all three proxy strategies, the proxy hit rate falls off after one proxy because it is more likely that the proxy node has to send a `read-request` to the home node as the number of proxies increases. This increase in `read-request` messages from proxy nodes in turn increases the buffer bounce ratio, because these requests are more likely to get bounced because there is no room for read messages in the home node's input message buffer. However, the reactive and adaptive schemes make better use of the proxy nodes, with a higher hit rate than that obtained with basic proxies.

The GE application illustrates how even the careful marking of a widely-shared data structure under the basic proxies scheme does not get the best performance. By using the adaptive proxy strategy, performance gains are obtained which are better than those achieved with either the "dumb" reactive strategy or basic proxies.

5.7.6 Ocean-Contig

This application displays a very mixed response to the use of proxies. None of the three proxy policies is able to avoid performance degradation at some value of $\mathcal{NPC}$, and there is no value of $\mathcal{NPC}$ which gets a performance improvement for all three proxy strategies. The best performance improvement (of 3.3%) is obtained by reactive proxies at $\mathcal{NPC}$=4, and this scheme achieves performance improvements for more values of $\mathcal{NPC}$ than the other

proxy strategies. Adaptive proxies have the poorest results. Basic proxies usually degrade performance (by up to −5.4% at $\mathcal{NPC}$=4), although small performance gains are seen when $\mathcal{NPC}$=6&8.

Basic proxies cause increases in both the number of messages and the queueing delay (see Figure 5.24). These changes are due to increases in the number of `take-shared` messages (and subsequent invalidation messages) for proxy nodes. There is also a slight increase in buffer-bouncing of `read-request` messages, as a result of the longer queues at nodes. However, despite increasing the store and load miss delays, basic proxies are still able to obtain performance improvements (see Figure 5.25) when $\mathcal{NPC}$=6&8 because the redistribution of read requests via proxy nodes changes the balance of processing in the system. This can result in lower barrier delays, which more than compensate for the increases in store and load miss delays.

When Ocean-Contig is run with reactive or adaptive proxies, it makes very little use of `proxy-read-request` messages, as is shown by the low level of proxy messages in Figure 5.27. Despite the low usage, these proxy reads have the effect of reducing the buffer-bounce ratio and the number of read messages, although there is a slight increase in the number of write messages because the number of invalidations goes up. The reduction in the number of read messages, and the re-routing of some read requests via proxy nodes, have the effect of lowering the overall queueing delay (see Figure 5.24). However, the redistribution of messages also has the effect, for some values of $\mathcal{NPC}$, of changing the balance of processing. This results in increased barrier delay and CPU active times which, at $\mathcal{NPC}$=1,2&5 for reactive proxies and $\mathcal{NPC}\geq$ 1 for adaptive proxies, result in an overall increase in execution time (*i.e.* performance suffers).

On balance, Ocean-Contig has its best performance with reactive proxies. The tiny proxy read ratio shows that there is very little use of proxies, although proxying does reduce the buffer bouncing to almost zero. This application is best suited to the "on demand" strategy of reactive proxies: overuse of proxying by the adaptive or basic policies has an adverse effect on the overall performance.

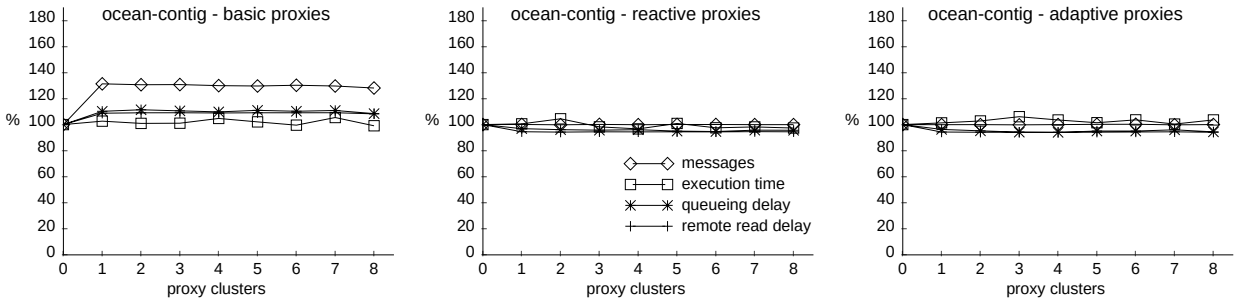


Figure 5.24: Ocean-Contig: changes (relative to no proxies case)

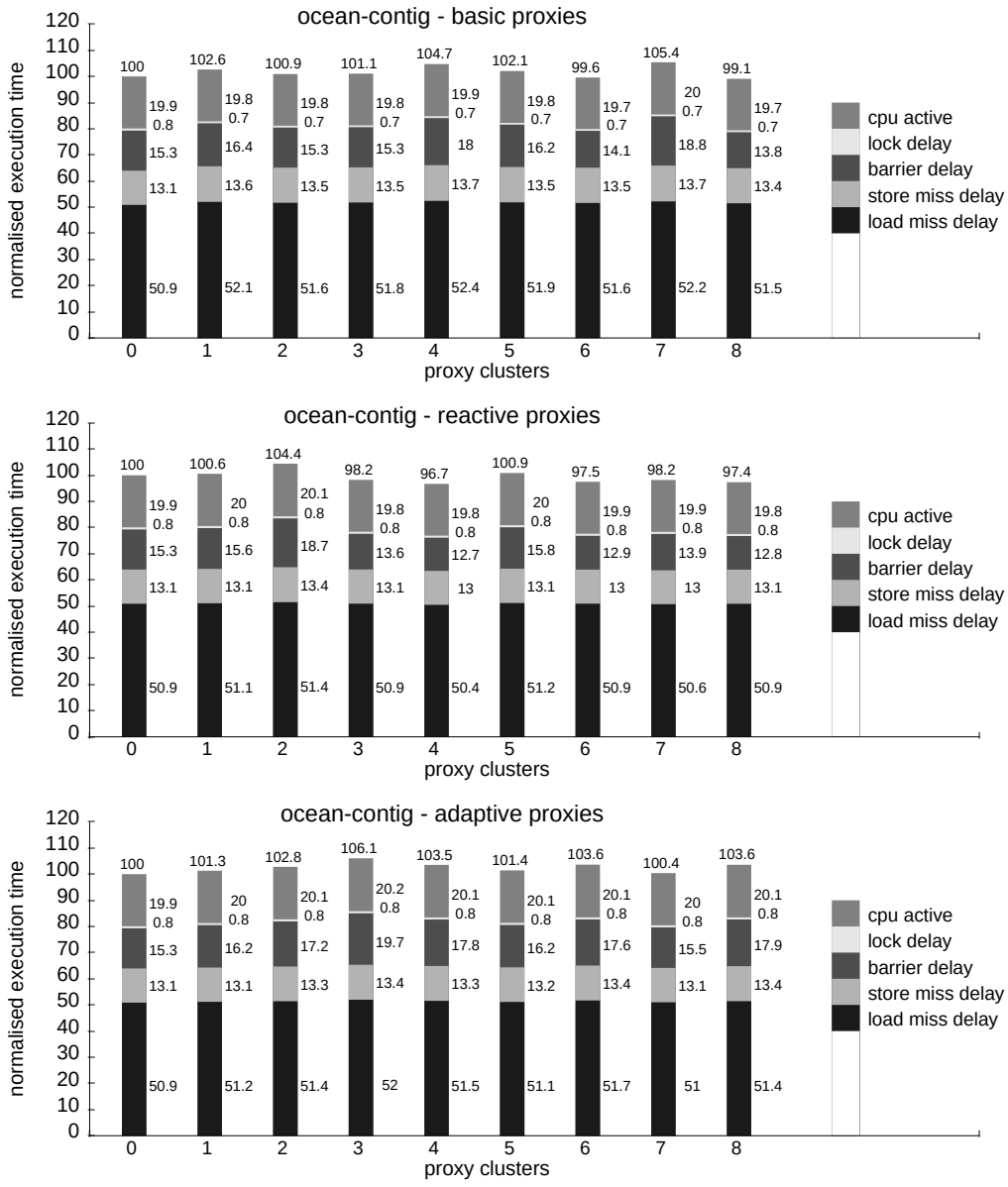


Figure 5.25: Ocean-Contig: execution time profiles

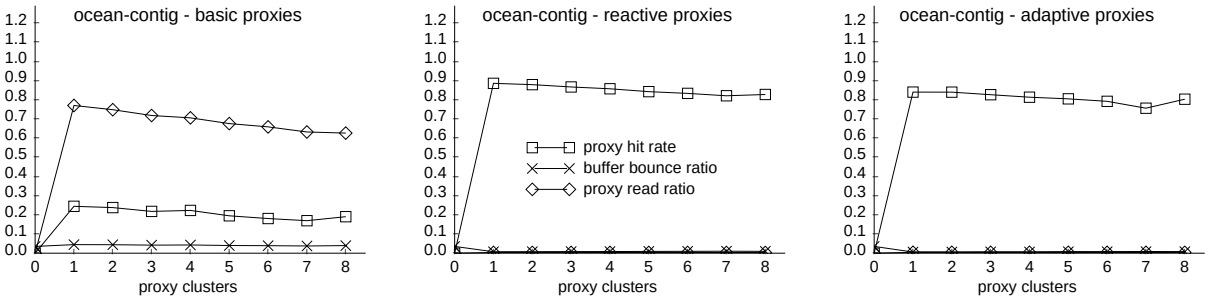


Figure 5.26: Ocean-Contig: message ratios

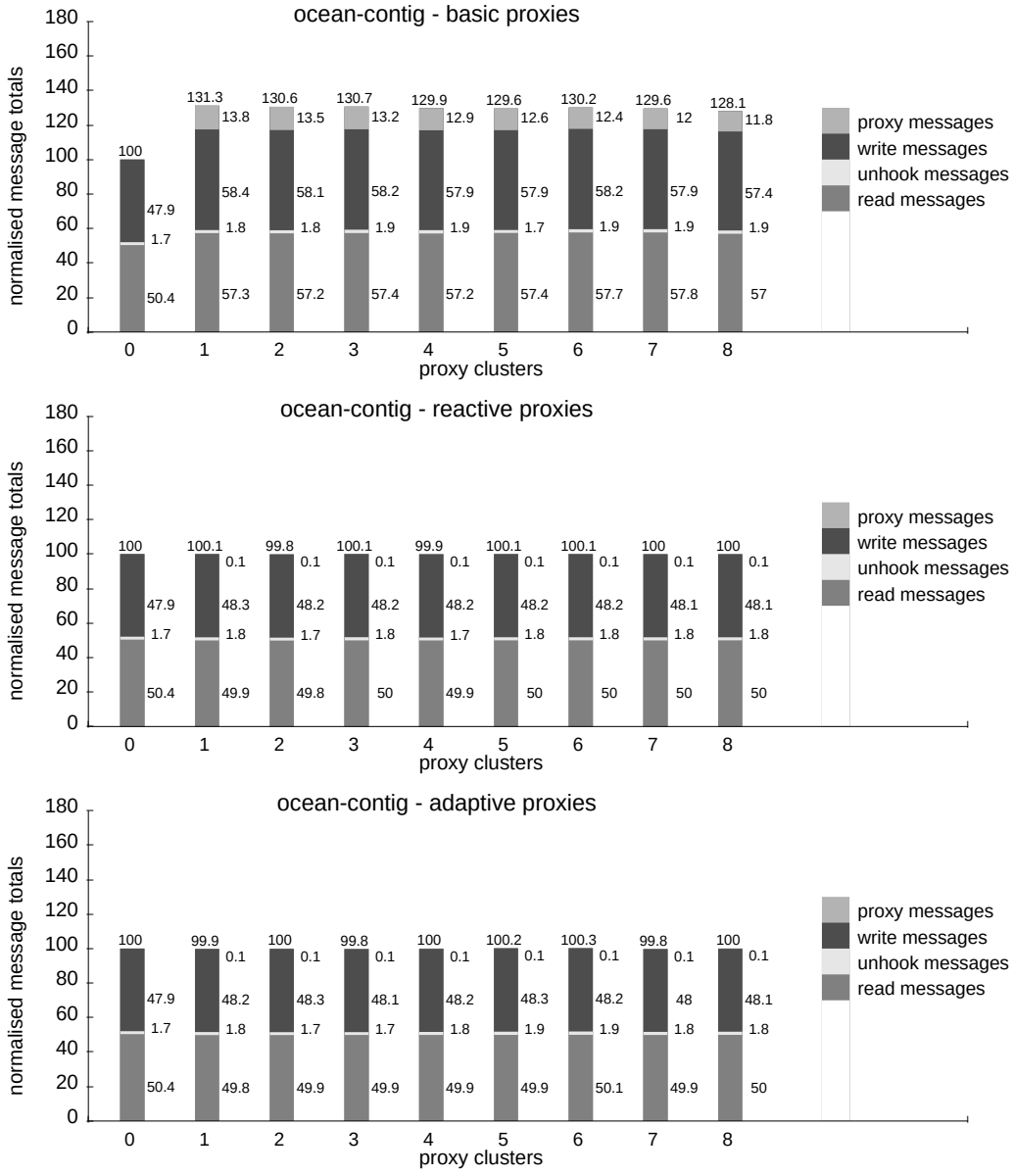


Figure 5.27: Ocean-Contig: message category profiles

5.7.7 Ocean-Non-Contig

This application can benefit from the use of proxies, with performance gains of up to 7.8%, but the overhead of additional unhooking and invalidation messages can cause a degradation in performance of up to −8.3% (see Figure 5.29 and Figure 5.31). Reactive proxies keep the number of messages below (or in the case of $\mathcal{NPC}$=6, only 0.1% above) the message level without proxies, and as a result reactive proxies nearly always improve the performance of Ocean-Non-Contig. The performance oscillations show that a relatively small increase in messages puts up the queueing delay significantly and degrades the performance.

Ocean-Non-Contig has a high level of remote read requests because the algorithm was written to preserve understandability rather than to exploit data locality. These remote `read-request` messages result in a high level of buffer-bounces, which in turn invoke the reactive and adaptive proxy protocols. Unfortunately the data is seldom widely-shared, so there is little combining at the proxy nodes, as is shown by the low proxy hit rates in Figure 5.30. The low proxy hit rates indicate that even for reactive and adaptive proxies there is less than a 20% probability that the proxy has (or has already requested) the data required by a client's `proxy-read-request`. The buffer-bouncing which is used to trigger reactive and adaptive proxying is the result of the already high level of remote read requests caused by the application prizing understandability over optimum data placement, rather than indicating a concentration of read requests for a particular data line. However, for reactive proxies, the application usually shows improved performance despite the lack of combining: the benefit comes from the two phase "random routing" effect of sending the read request via a proxy.

Like Ocean-Contig, this application is best served by reactive proxies, because `proxy-read-request` messages are only used in direct response to the buffer-bouncing of read requests. As a result, reactive proxies has only one value of $\mathcal{NPC}$ which displays adverse performance, unlike the basic and adaptive proxy strategies.

5.7.8 Water-Nsq

This application benefits from triggering the use of proxies in response to home node congestion: both reactive and adaptive proxies improve the performance, whereas basic proxies degrade the performance. Despite the very low level of proxying by the reactive and adaptive schemes (see the proxy read ratio in Figure 5.34), both strategies succeed in reducing the buffer-bounce ratio and remote read delay. This can lead to a slight reduction in the overall load miss delay, as shown in Figure 5.33. In contrast, the basic proxy scheme increases the number of messages in the system, as shown in Figure 5.35, and fails to achieve the drop

in remote read delay obtained by the other proxy strategies (see Figure 5.32). Using basic proxies also causes more disturbance to the balance of processing between the nodes, leading to an increase in the CPU active time. The results indicate that the wrong data structure was chosen for basic proxying, whereas the reactive and adaptive schemes are able to improve the application's performance with only a low use of `proxy-read-request` messages.

For both the reactive and adaptive schemes there are values of $\mathcal{NPC}$ where the load miss delay is at the same level as when $\mathcal{NPC}$=0, despite a drop in the remote read delay. This is due to the increased delay for local load misses (*i.e.* those wanting to get to access the local DRAM). Local misses experience an increased delay because they have to wait for service from their node controller while it is dealing with the current incoming message (local requests only have to wait for the current message to be serviced, rather than having to wait for all the currently queued incoming messages). This is an example of how the proxy schemes can affect the order of processing at node controllers.

The Water-Nsq application is suited to either of the automatic proxying schemes, getting a slight performance improvement (of around 0.2%) for all values of $\mathcal{NPC}$. This is in contrast to the performance degradation caused by using the basic proxy strategy. The behaviour of this benchmark clearly illustrates the advantages of using automatic proxying rather than taking the risk that the programmer (or compiler) will incorrectly mark data structures for proxying.

5.7.9 Summary of Results

These results show that there are certain values of $\mathcal{NPC}$ for which the reactive proxy scheme improves the performance of all the applications. These balance points for reactive proxies occur when $\mathcal{NPC}$=3,7&8. Reactive proxies always suffer a delay of `read-request` and `buffer-bounce-read-request` before invoking the `proxy-read-request`, but overall the latency benefits from the complex interaction of the higher level of combining when $\mathcal{NPC}$ is low, and the shorter proxy pending chains and queue lengths when $\mathcal{NPC}$ is high. The balance points occur when the partitioning into proxy clusters results in a distribution of `proxy-read-request` messages which strikes a balance between the length of proxy pending chains, the degree of combining, the minimum disturbance to other processing at each proxy node, and little if any cache pollution.

For the adaptive proxy scheme, once proxying has been invoked the reads benefit from spreading the messages around the system during the proxying period. However the scheme suffers from over-using proxies for the Ocean-Contig application, and so has no overall balance point for the eight benchmark applications.

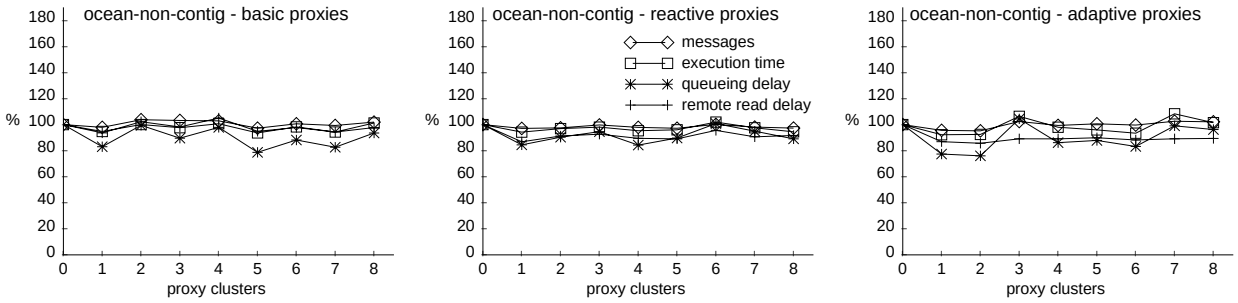


Figure 5.28: Ocean-Non-Contig: changes (relative to no proxies case)

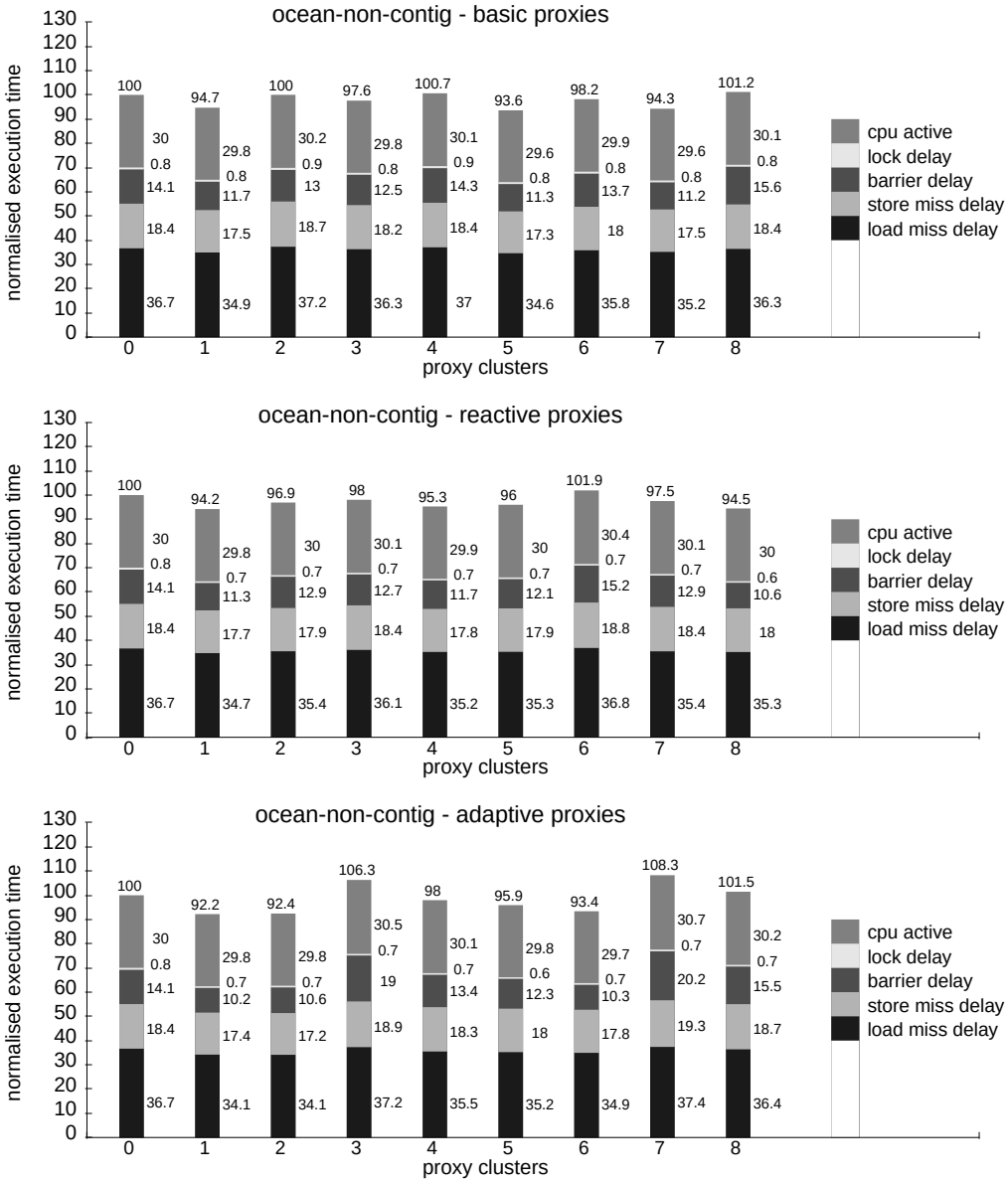


Figure 5.29: Ocean-Non-Contig: execution time profiles

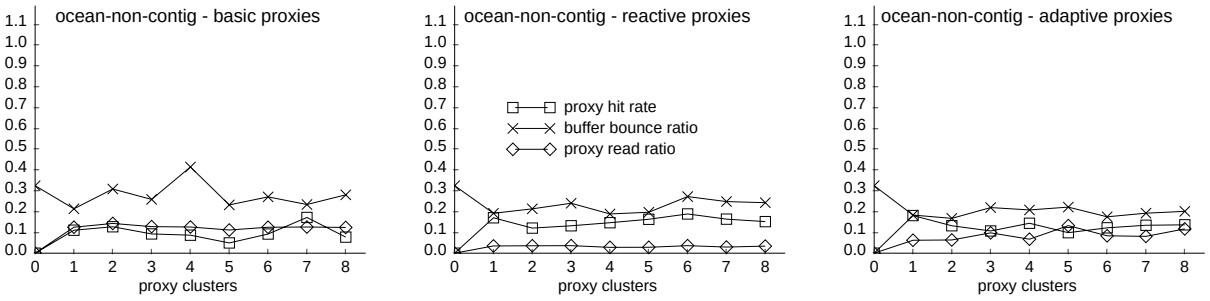


Figure 5.30: Ocean-Non-Contig: message ratios

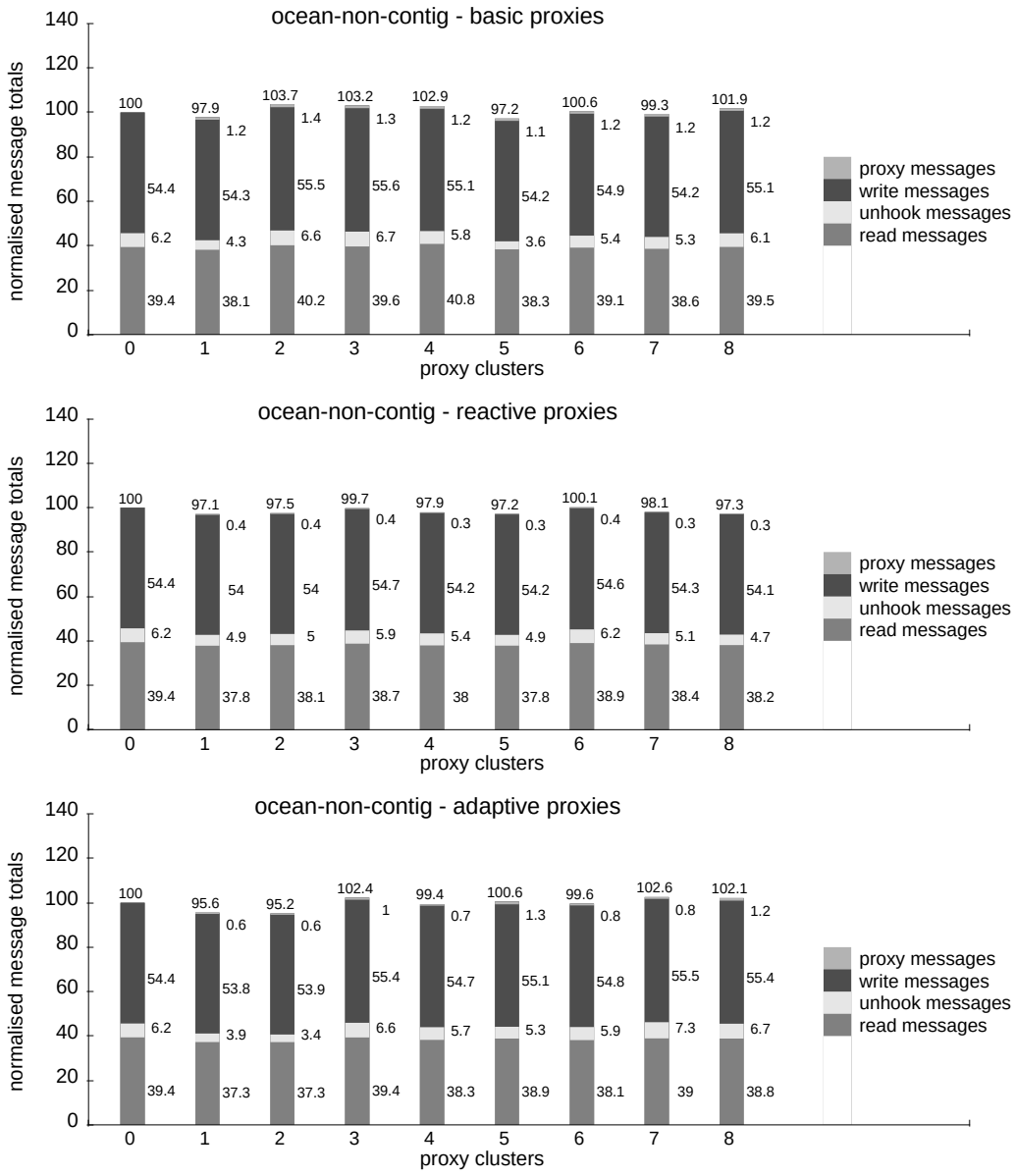


Figure 5.31: Ocean-Non-Contig: message category profiles

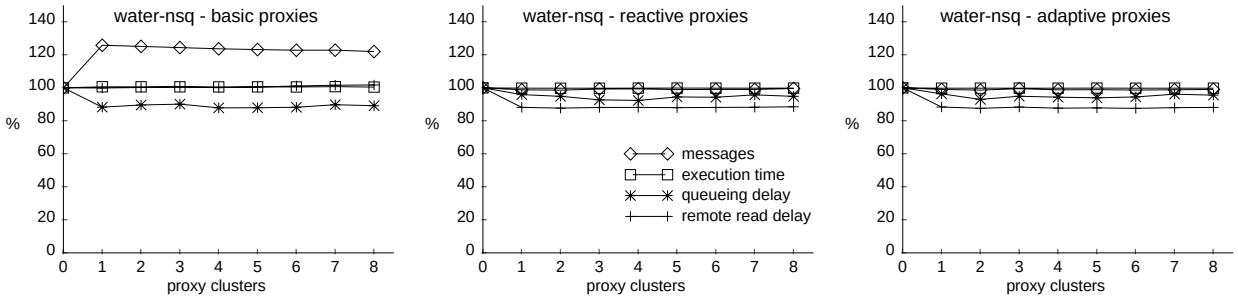


Figure 5.32: Water-Nsq: changes (relative to no proxies case)

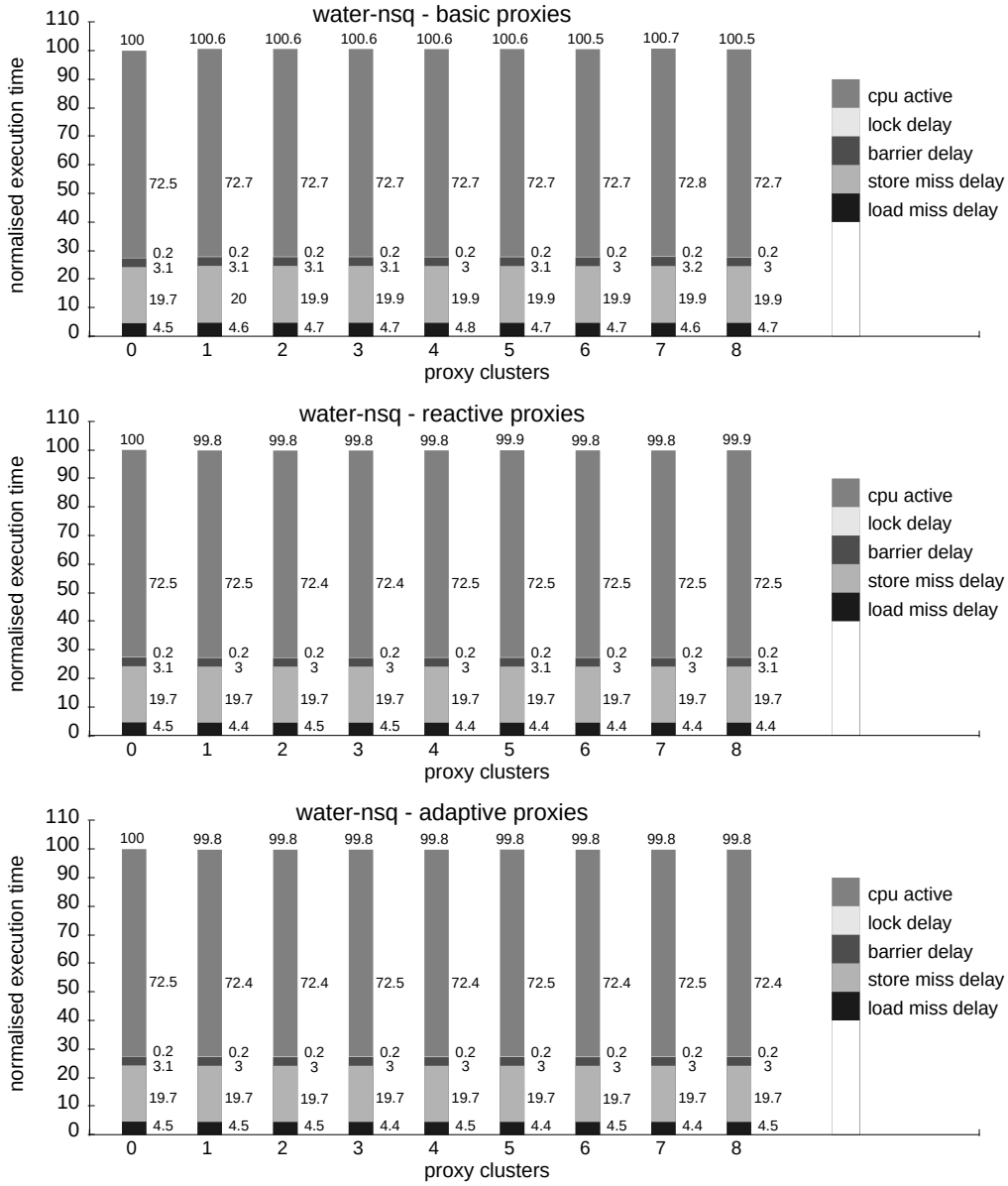


Figure 5.33: Water-Nsq: execution time profiles

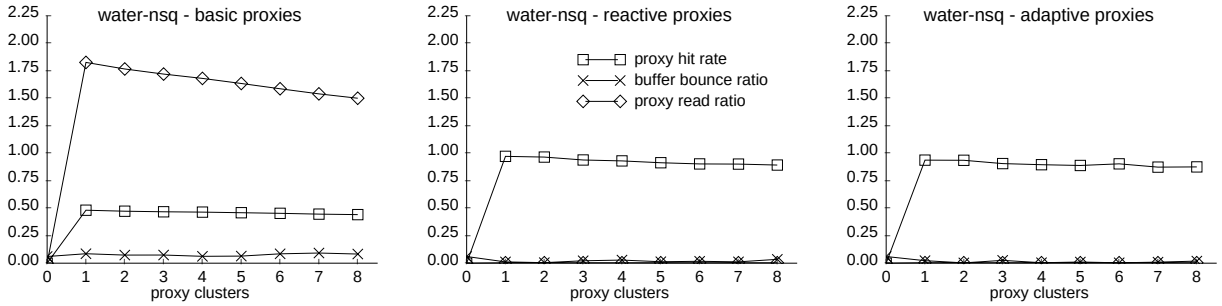


Figure 5.34: Water-Nsq: message ratios

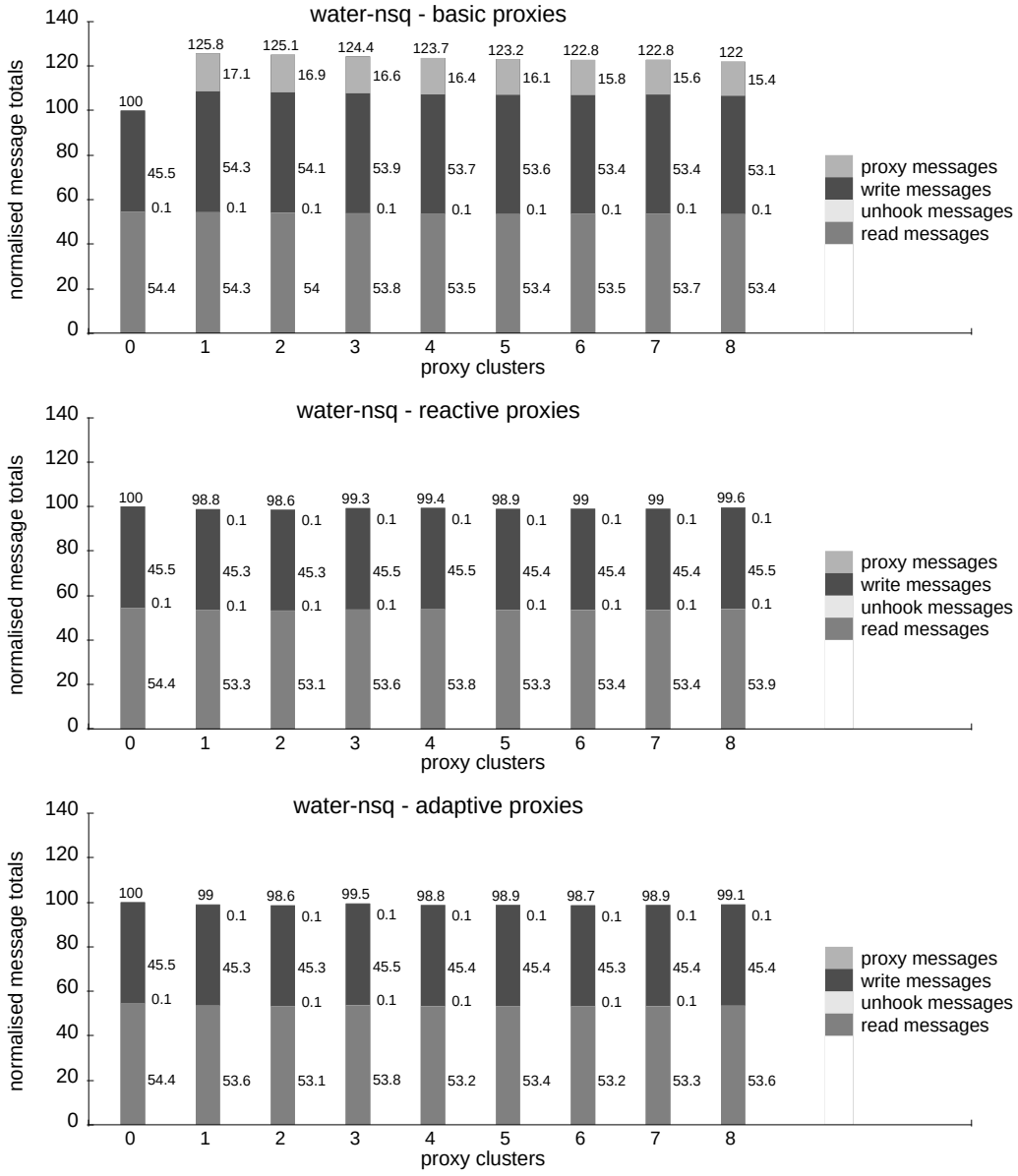


Figure 5.35: Water-Nsq: message category profiles

The existence of balance points for reactive proxies is very encouraging, because the scheme can obtain sizeable performance improvements for three benchmarks (GE, FFT, and CFD), with no detrimental effect on the other applications, and this has been achieved without having to mark data structures. By selecting a suitable $\mathcal{NPC}$ for an architecture, the system designers can provide a cc-NUMA system with more stable performance for portable programs. This is in contrast to basic proxies, where although better performance can be obtained for some benchmarks, the strategy relies on judicious marking of widely-shared data for each application.

The detailed examination of the results also showed that the potential costs of the automatic schemes identified in Section 5.4 can affect the performance of the applications. For example, Ocean-Non-Contig is an application with a high level of messages, and performance suffers when the number of messages increases with proxies: the bottlenecks have just been shifted to other nodes. However, for this application, the reactive proxy scheme keeps the message count close to or well below the no proxy level, and as a result the scheme alleviates the message queueing bottlenecks and gets better performance.

The GE application illustrated how even the careful marking of a widely-shared data structure under the basic proxies scheme will not necessarily get the best performance. Using the adaptive proxy strategy gave performance gains which were better than those obtained with either basic proxies or the "dumb" reactive strategy. The adaptive proxy scheme benefited from only using proxies when congestion occurred, and then continuing to use proxies during the adaptive proxying period.

The GE application also exhibited some bottleneck problems for both of the automatic proxying schemes, where `proxy-read-request` messages were sent to an already congested node leading to a rise in overall queueing delay (although this was more than compensated for by the gains at other nodes). In an ideal situation, the proxying scheme would not send `proxy-read-request` messages to a node which is currently buffer-bouncing other read requests. It would be possible to extend the automatic schemes to check whether they had received a buffer-bounce from the proxy candidate. However this would then raise the issue of how to choose an alternative proxy, which is the subject of further work.

It is hard to say which of the automatic proxying schemes is the best. The reactive strategy has the advantage of being inexpensive to implement and use. It only requires a simple change to the client state machine processing at each node controller to handle the generation of `proxy-read-request` messages in response to the NAK of a read request. However the reactive scheme has no "memory" of past events, and therefore subsequent `read-request` messages are still sent direct to a home node even when it is likely that it is still congested.

The results showed that for the CFD and GE applications it was better to use the adaptive scheme (see Table 5.2). The downside of the adaptive scheme is that it is more complicated to implement because it needs changes to the NAK receipt processing, and all outgoing read requests have to check against the adaptive proxy table to see if the `read-request` should be converted to a `proxy-read-request` message. This check would really have to be implemented in hardware otherwise normal processing would be slowed down by too much, and such hardware changes would be expensive. In addition the Ocean-Contig application showed performance degradation with adaptive proxies because it was not appropriate to go on using `proxy-read-request` messages for the proxy period. On balance it is likely that the reactive proxy scheme would be chosen for implementation even though it does not always get the best possible performance. This is because reactive proxies only require minor changes to the node controller's protocol processing, and these could reasonably be implemented in software on a programmable node controller (because `read-request` NAK's are an exception rather than being regular processing tasks which need to be implemented in hardware).

5.8 Conclusions

This chapter has introduced two strategies for proxying which are invoked automatically at run-time. The schemes operate in a realistic environment where message buffers are finite, and `read-request` messages are "returned to sender" when there is no room for them in an incoming message buffer. The new reactive proxy policy was shown to have values of $\mathcal{NPC}$ (number of proxy clusters) for which the performance of all eight benchmark application was improved. This suggests that reactive proxies could be used to avoid the bottlenecks which result from full message buffers, and so provide more stable performance for shared-memory systems.

Some of the applications still exhibited problems with cache pollution which were exacerbated by proxies. The next two chapters investigate proxy caching strategies which aim to reduce the effect of proxying on SLC cache pollution.

Chapter 6

Non-Caching Proxies

In the work described so far, any data obtained by a node when it is acting as a proxy is cached in its second level cache (SLC). This was done deliberately to increase the combining effect, *i.e.* further read requests for that data can be satisfied immediately from the proxy node's cache. There are, however, a number of drawbacks to this approach, including increased sharing list length, cache pollution, and delays to the local CPU and node controller processing.

These delays and conflict problems lead one to ask whether it is worthwhile caching the proxied data lines, *i.e.* would it have been just as effective to have non-caching proxy read operations, thereby avoiding the unhooking of conflicting data lines? Alternatively, would it be better to have a separate "proxy buffer" to hold the data obtained by a node when it is acting as a proxy? This chapter examines the first of these strategies, *i.e.* not caching proxy data at proxy nodes. The approach of using a separate buffer for proxy data is investigated in Chapter 7.

6.1 The Non-Caching Approach to Proxies

The original proposal for proxies included caching data obtained by a node (while acting as proxy) in its SLC, in order to increase the potential for combining (because further proxy read requests for that data line are more likely to be immediately satisfied from the proxy node's SLC). Unfortunately this approach has some drawbacks, which include increased sharing list length, cache pollution, and delays to local CPU and node controller processing. The sharing list increases by the number of proxies holding a copy of a data line which they will not use for local processing. This leads to increased latency for invalidations of the entire sharing list, and also gives rise to longer unhooking latency when the proxy node evicts the proxied data

from its SLC.

The cache pollution occurs because the proxy data in the cache may displace another data line. At the very least this will cause a delay while the displaced line is removed from its sharing list, and at worst it will be displacing data which is about to be required by the local CPU, in which case the proxy data will soon be displaced (causing yet another unhook delay).

There may also be delays associated with accessing the SLC. In the cc-NUMA architecture simulated in this thesis the node controller first has to gain control of the SLC bus, which will cause a delay if the SLC bus is already in use by the local CPU. In addition, while the node controller has control of the SLC bus the local CPU cannot access its SLC, which will increase the latency of any FLC miss which occurs during this period. The delay to the CPU would be even worse in systems where the SLC can only be accessed by the CPU because the CPU might have to be interrupted from its local processing to load the proxy data into the SLC.

In the non-caching form of proxies, data which is obtained by a node acting as proxy will not be cached locally unless the local CPU also needs the data. In some ways this looks like a minor change to the protocol, with a similar sequence of messages and state changes as were needed for the original proxy protocol described in Chapter 4. However, because a proxy node must not be added to the sharing list there has to be a way for the home node to distinguish between "normal" read requests and non-caching read requests from proxies. Figure 6.1 shows an example of a proxy read under the non-caching proxy scheme. It is similar to Figure 4.2, but differs at steps 2 and 3 because non-caching proxy messages are used to ensure that the proxy node is not added to the sharing list. The detailed design issues associated with non-caching proxies are discussed further in Section 6.2.

It should be noted that the non-caching policy can be used with all three forms of proxying described so far in this thesis. The basic, reactive, and adaptive proxy schemes are all concerned with deciding whether a client node will send a `proxy-read-request` to a proxy. The caching policy at the proxy is independent of the decision of when to use proxies, so the non-caching policy will be evaluated for all three proxy strategies.

It is likely that there will be less combining in the non-caching scheme than was observed under the original caching approach. When a node receives a `take-shared` message for data which is destined for its clients, the node will send the data on to all the clients but will not cache the data in its SLC (unless the local CPU is also stalled waiting for that data). As a result, a `proxy-read-request` for the same data which arrives soon afterwards at the proxy node cannot be serviced immediately because the proxy does not have a copy of the data line.

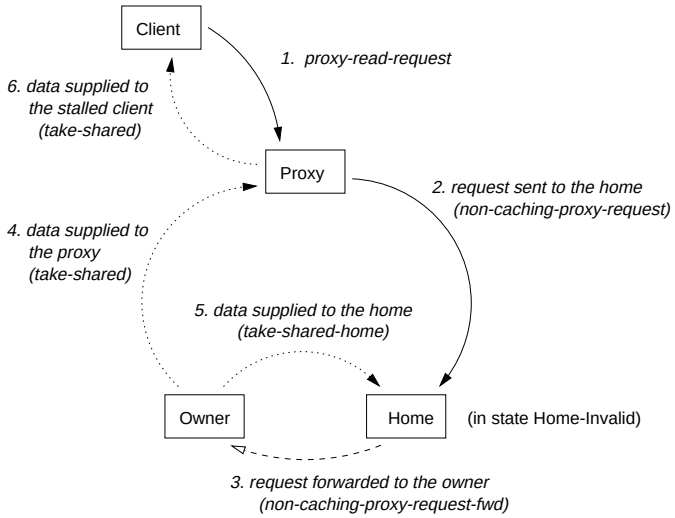


Figure 6.1: A non-caching proxied read request

The proxy will have to issue another `non-caching-proxy-request` to the home node, and the client will have to wait longer for its data.

This reduction in the potential for combining is the main drawback to the scheme. Combining does occur for any "duplicate" `proxy-read-request` messages which arrive during the period between the proxy requesting the data from the home node and its arrival in a `take-shared` message. In addition, if a proxy node has an SLC copy of the data for its local CPU's processing, then it will be able to satisfy `proxy-read-request` messages. This potential for combining is better than schemes where combining only occurs for requests which are simultaneously in a switch's input buffer (*e.g.* the NYU Ultracomputer [45]). However, the level of combining will be no better than schemes which combine requests for the same data at intermediate nodes in the network hierarchy (*e.g.* hierarchical COMA systems such as the SICS DDM [50]), and will not be as good as schemes which hold a copy of widely-shared data in intermediate network nodes (*e.g.* the GLOW protocol extensions [69]).

6.1.1 Potential Benefits and Costs of Non-Caching Proxies

The non-caching proxy scheme should have a number of effects. Among the benefits one would expect are:

No SLC pollution from using proxies: a better SLC hit rate for the local CPU, fewer unhooking messages, and less CPU stall time because SLC misses caused by a conflict with proxy data are avoided.

Shorter sharing lists: because the proxy no longer has a copy; leading to fewer invalidation messages.

Reduced usage of SLC bus: the node controller will not have to load or invalidate proxy data in the SLC.

The potential costs are:

Less combining: proxies will no longer retain a copy of the proxied data lines, so a subsequent `proxy-read-request` arriving from a client will require another `non-caching-proxy-request` to be send to the home node.

Complication of the coherence protocol: special actions are needed to handle the receipt of a "non-caching" read request from a proxy (to add the first client rather than the proxy to the head of the sharing list).

These potential costs and benefits are considered as part of analysing the results in Section 6.4.

6.2 Design Issues

There are a number of factors which have to be considered in order to implement the non-caching proxy scheme. These are how to support non-caching read requests, and the need to modify the handling of `proxy-read-request` messages to prevent the reservation of an SLC cache line to hold the data which will arrive with the `take-shared` message.

6.2.1 Non-Caching Read Requests from Proxies

The non-caching proxy policy relies on the proxy node not being added to the sharing list. However, in the original protocol, `read-request` messages result in the requester being added to the sharing list. As a result, it is necessary to introduce a new message type, the `non-caching-proxy-request`, to indicate to a home node controller that the receipt of such a message should result in the *client* being added to the sharing list, rather than the requester. Figure 6.2 shows an example of the actions required to handle a straightforward proxy read transaction using the new message type.

The decision to have the home node send the `take-shared` message to the proxy node, rather than directly to the client node, was taken to maximise the combining of requests. Further clients (and possibly the proxy node itself) may be added to the proxy's pending chain while the `non-caching-proxy-request` is being processed by the home node. By routing the `take-shared` via the proxy, one can be certain that all the opportunities for combining are realised in the non-caching proxy scheme. This approach does have the drawback that the

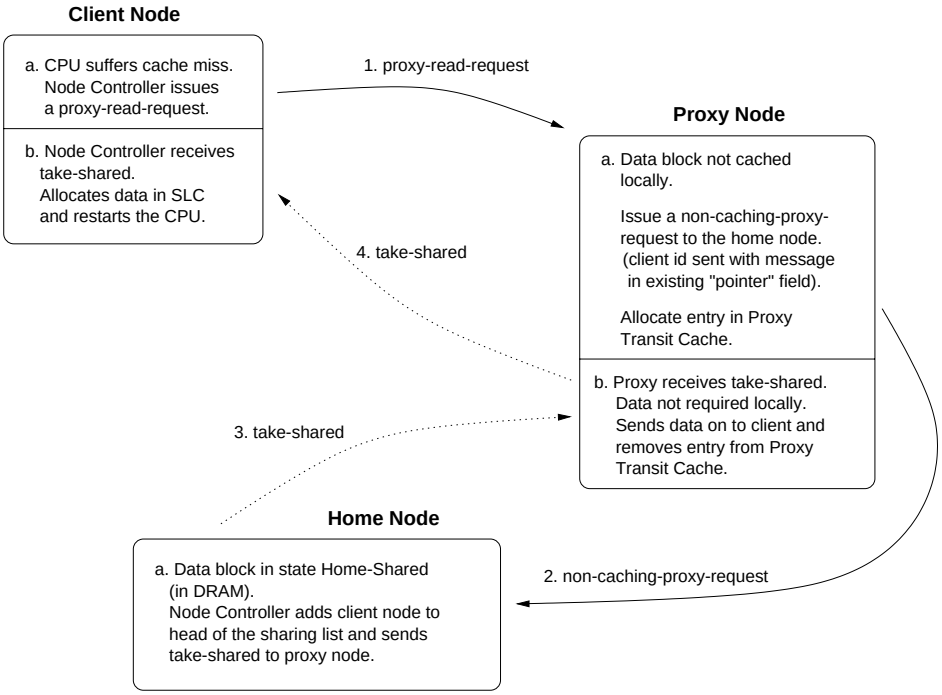


Figure 6.2: Actions required to handle a non-caching proxied read request

client will be at the head of the sharing list for some time before it is sent a `take-shared` message. This can increase the probability that other messages (such as an `invalidate` or `client-unhook-forward`) traverse the sharing list and reach the client before it knows it is part of the sharing list. Invalidation or unhook-forward messages will have to be "buffered" at a client when the client has requested to be added to the sharing list, but has not yet received the `take-shared` message from the proxy which provides the link to the tail of the list.

There was a further change needed to handle non-caching read requests from proxies. In cases where the home node is in the Home-Invalid state, *i.e.* where there is a more up-to-date copy of the data at the current owner node, read requests are forwarded to the owner node. The non-caching processing was triggered by providing a new `non-caching-proxy-request-fwd` message type, which prompts the owner node to add the client to the sharing list and send a `take-shared` message to the proxy (and a `take-shared-home` message to the home node).

It should be noted that `non-caching-proxy-request` messages are not to be confused with the "normal" non-caching operations which are allowed for in some instruction sets, for example the cache bypass "hints" held in the 2-bit cache control field provided in Hewlett Packard's PA-RISC instruction set [65]. Such non-caching operations are provided to avoid caching data which the CPU will not need again before the data is evicted. In contrast, for `non-caching-proxy-request` messages, although the data is not cached in the proxy's SLC the data line *is* cached at the client node.

6.2.2 Reduction of SLC Conflicts

The introduction of non-caching proxies means that a proxy node no longer has to reserve space in the SLC for a data line which is currently being obtained for a client. This differs from the original proxy protocol, where the arrival of a `proxy-read-request` would trigger the eviction of a conflicting data line (*i.e.* a different data line which maps to the same cache line) from the SLC. This unhooking is no longer required in the non-caching protocol, because the cache line does not have to be "reserved" for the arrival of the proxy data. This change in the use of the SLC has ramifications for the handling of SLC misses from the proxy node's CPU which occur before the arrival of the `take-shared` message with proxy data.

The first relevant cache miss instance is where the proxy node's CPU suffers a cache miss on a data line which maps to the same SLC cache line as a data line for which the node is currently acting as proxy. Under the original proxy scheme, the local CPU would be stalled until the proxy data was loaded into the SLC when the `take-shared` message arrived: the proxy data would then be unhooked, and only then could the node controller start processing the local CPU's cache miss. With the non-caching proxy scheme, there is no longer a conflict between the proxied data and the data the local CPU requires, so the local cache miss can be dealt with immediately by the node controller. As a result, the local CPU's stall time will be reduced, and there is no knock-on unhooking of a proxy data line.

The other situation is where the local CPU suffers a cache read miss on a data line for which it is currently acting as proxy. Unlike the situation under the original proxy protocol, the `non-caching-proxy-request` will not cause the proxy node to be added to the sharing list, and there may be conflicting data in the SLC. The possible conflict is easily dealt with by the underlying protocol, which will evict the conflicting data. However getting the local node added to the sharing list is slightly more complicated. The solution adopted was to use a flag in the proxy transit cache entry to indicate that the local CPU now needs the data. This local-read flag is set to "off" when entries are added to the proxy transit cache. If the local CPU suffers a cache read miss for that data line, the local-read flag is set "on". When a `take-shared` message arrives for which there is a matching entry in the proxy transit cache, the local-read flag can be checked to see if the data should be copied into the SLC.

There still remains the question of how to link the local node into the sharing list. It seems sensible not to change the existing protocol which passes the `take-shared` message along the pending chain of client nodes. However, by getting the local SLC's entry to point to the tail of the sharing list, and by putting the local node's ID as the "pointer" to the tail of the list which is sent to the clients, the local node is inserted on the list after the client nodes. The

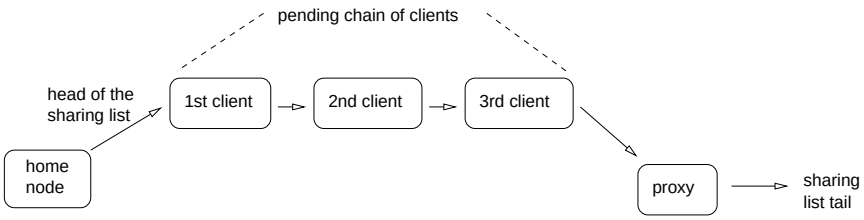


Figure 6.3: Example ordering of sharing list when the proxy's CPU also needs the data line

resulting sharing list is illustrated by the example in Figure 6.3. This approach allows the local CPU to be restarted as part of processing the `take-shared` message at the proxy node.

6.3 Modifications to the Protocol and Architecture

The main changes needed to support non-caching proxies are to the protocol. To enable the home (or owner) node to add the first client rather than the proxy to the sharing list, two new message types are required: the `non-caching-proxy-request` and the `non-caching-proxy-request-fwd`. The arrival of these messages can result in the state changes illustrated in Figure 6.4. Two additional messages are needed to handle certain node controller states: a `bounced-non-caching-proxy-request` is sent back to the proxy when the home node's directory entry for the data line is locked, and a `buffer-bounced-non-caching-proxy-request` supports the simple finite buffer simulation (it is sent by the home node when there are eight or more messages in its incoming message buffer). The introduction of the latter message type ensures that a fair comparison is made with the "SLC proxy caching" results in Section 5.7, where any `read-request` sent by a proxy node is liable to be buffer-bounced. In addition to handling the new message types and state transitions, changes must be made to the node controller's processing of local read misses, and to the processing for `proxy-read-request` and `take-shared` messages (as discussed in Section 6.2).

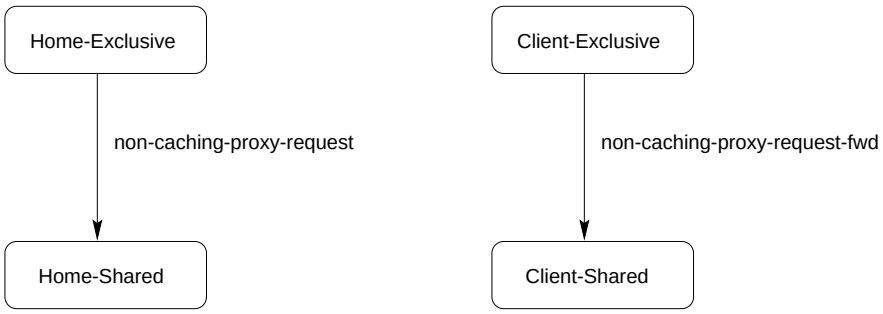


Figure 6.4: Extra node controller state transitions for non-caching proxies

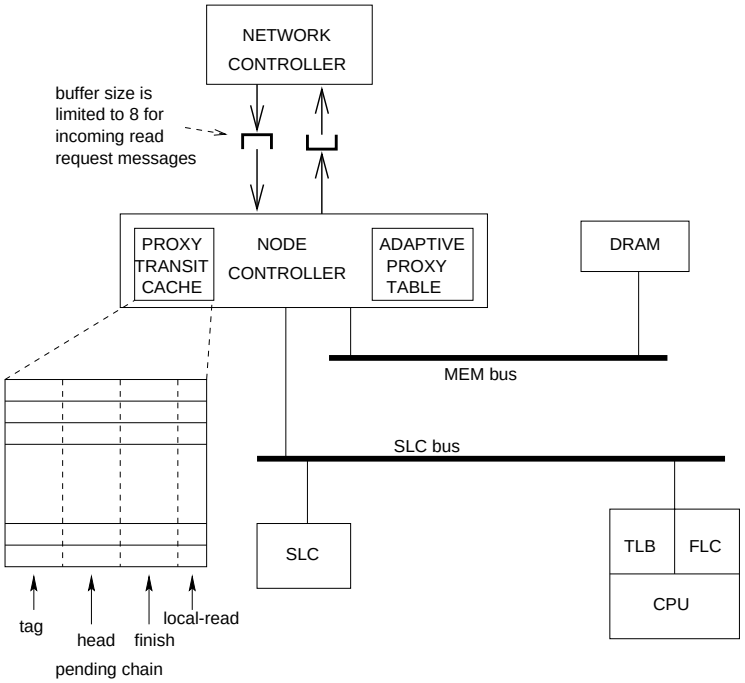


Figure 6.5: Memory model for a cc-NUMA node with non-caching proxies

Compared with the hardware structure used in Chapter 5 (see Figure 5.3), the only architectural modification needed to support non-caching proxies is the addition of a local-read flag in the proxy transit cache. This is used to indicate whether the local CPU is stalled on a read miss for the proxy data. The change is illustrated in Figure 6.5.

6.4 Results

This section presents the results obtained from execution-driven simulations[1] of the basic, reactive, and adaptive proxy strategies using the non-caching proxy policy. It was shown in Section 3.5 that contention only becomes an important issue when more than a few tens of nodes are used. For this reason the detailed results presented below are from simulations of a 64 node design. For basic proxies, the shared data marked for proxying is shown in Table 6.1: the same marking was used in the preceding chapters.

The results from using non-caching proxies are summarised in Table 6.2. The performance speedup results are presented in terms of relative speedup, *i.e.* the ratio of the execution time for the fastest algorithm running on one processor to the execution time on $\mathcal{P}$ processors. To simplify the comparison of these results with those from the SLC caching scheme, Table 6.3 is a copy of Table 5.2 from Chapter 5. Basic proxies tend to have slightly improved performance

[1] The details of the simulated system were given earlier in Section 3.4.

application	problem size	shared data marked for basic proxying
Barnes	16K particles	all
CFD	64×64 grid	all
FFT	64K points	all
FMM	8K particles	`f_array` (part of `G_Memory`)
GE	512×512 matrix	entire matrix
Ocean-Contig	258×258 ocean	`q_multi` and `rhs_multi`
Ocean-Non-Contig	258×258 ocean	`fields`, `fields2`, `wrk`, and `frcng`
Water-Nsq	512 molecules	`VAR` and `PFORCES`

Table 6.1: Benchmark problem sizes, and data marked for basic proxies

using the non-caching strategy compared to that seen in Chapter 5. The only "balance point" for reactive proxies is at $\mathcal{NPC}$=2. For adaptive proxies, there is now a balance point at $\mathcal{NPC}$=1. Overall there are no dramatic changes in performance using the non-caching scheme.

In order to understand the changes exhibited by the individual applications, the results are examined in detail in the sub-sections which follow. For each application, the detailed results are presented as two graphs, one of relative changes and one of message ratios, and two histograms, one showing the execution time profile, and the other showing the message category profile. In addition, node-by-node graphs showing the mean queueing cycles are given where appropriate.

The relative changes results show four different metrics:

messages: the ratio of the total number of messages to the total without proxies,

execution time: the ratio of the execution time (excluding startup) to the execution time (also excluding startup) without proxies,

queueing delay: the ratio of the total time that messages spend waiting for service to the total without proxies, and

remote read delay: the ratio of the mean delay between issuing a `read-request` and receiving the data, to the same mean delay when proxies are not used.

The message ratios are:

proxy hit rate: the ratio of the number of proxy read requests which are serviced directly by the proxy node, to the total number of proxy read requests (in contrast, a proxy miss would require the proxy to request the data from the home node),

buffer bounce ratio: the ratio of the total number of buffer bounce messages to read requests. This gives a measure of how much bouncing there is for an application. This

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	-0.3	+0.4	-0.1	+0.2	+0.2	+0.2	+0.1	-0.1
		reactive	+0.1	+3.4	0.0	+0.2	+0.1	+0.1	-0.2	+0.4
		adaptive	+0.4	+3.7	0.0	0.0	+0.5	+0.3	+0.1	+0.2
CFD	28.3	basic	+14.3	+14.9	+14.7	+15.7	+14.8	+15.0	+14.3	+15.3
		reactive	+10.5	+9.4	+8.9	+8.1	+9.5	+7.4	+8.4	+6.3
		adaptive	+12.9	+13.7	+13.6	+12.7	+12.9	+13.5	+12.7	+12.5
FFT	47.3	basic	+10.7	+10.0	+9.8	+10.4	+9.6	+9.6	+10.5	+9.5
		reactive	+11.2	+10.8	+10.7	+10.7	+10.7	+11.4	+10.7	+10.3
		adaptive	+11.7	+11.2	+11.3	+11.4	+11.3	+11.1	+11.2	+10.8
FMM	52.4	basic	+0.4	+0.4	+0.5	+0.5	+0.4	+0.4	+0.4	+0.4
		reactive	+0.3	+0.4	+0.4	+0.4	+0.3	+0.4	+0.4	+0.3
		adaptive	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4	+0.4	+0.4
GE	21.6	basic	+30.9	+30.8	+30.7	+30.7	+30.6	+30.5	+30.3	+30.1
		reactive	+27.5	+27.8	+28.1	+28.0	+27.9	+27.9	+27.9	+27.9
		adaptive	+30.3	+30.5	+31.4	+31.0	+31.3	+31.3	+31.0	+31.0
Ocean-Contig	49.7	basic	-2.0	+1.0	+1.0	-2.9	-1.0	-1.5	-1.6	-0.7
		reactive	-1.1	+0.2	-7.0	+3.0	-3.3	-1.5	+0.7	-0.4
		adaptive	+3.2	+0.5	-1.0	-2.3	0.0	-2.6	-0.1	-1.1
Ocean-Non-Contig	48.2	basic	+4.8	+7.1	+1.4	+4.1	+2.2	+2.1	+6.1	+2.6
		reactive	+4.1	+0.9	+4.2	-19.4	-6.9	+7.4	+6.2	+4.6
		adaptive	+0.5	-3.6	+4.4	-11.3	+3.7	+4.7	+7.4	+3.3
Water-Nsq	55.3	basic	-0.4	-0.4	-0.3	-0.4	-0.3	-0.3	-0.4	-0.3
		reactive	+0.2	+0.2	+0.2	+0.1	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2

Table 6.2: Benchmark relative speedups for non-caching proxies (64 nodes)

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	+0.2	0.0	0.0	+0.2	-0.1	-0.3	0.0	-0.1
		reactive	+0.4	+3.3	+0.2	0.0	+0.3	0.0	+0.5	+0.4
		adaptive	+0.1	+3.2	+0.4	+0.4	+0.4	+0.2	-0.1	+0.2
CFD	28.3	basic	+10.4	+11.3	+8.1	+11.8	+9.8	+9.3	+9.2	+14.8
		reactive	+7.6	+7.4	+8.3	+7.6	+8.6	+7.6	+7.6	+6.1
		adaptive	+9.2	+13.1	+11.3	+11.6	+11.2	+10.4	+10.6	+12.1
FFT	47.3	basic	+9.4	+8.7	+8.7	+9.6	+9.5	+8.6	+10.0	+8.5
		reactive	+11.7	+11.2	+10.9	+11.0	+11.2	+11.8	+11.2	+10.7
		adaptive	+11.9	+11.6	+11.3	+11.4	+11.2	+11.5	+11.0	+11.0
FMM	52.4	basic	+0.4	+0.4	+0.5	+0.3	+0.4	+0.3	+0.3	+0.4
		reactive	+0.3	+0.4	+0.4	+0.4	+0.3	+0.4	+0.4	+0.4
		adaptive	+0.4	+0.4	+0.4	+0.4	+0.4	+0.5	+0.4	+0.4
GE	21.6	basic	+29.3	+29.3	+29.3	+29.4	+29.4	+29.5	+29.6	+29.6
		reactive	+28.4	+28.6	+28.9	+28.8	+28.8	+28.8	+28.7	+28.9
		adaptive	+30.5	+30.7	+31.4	+31.2	+31.7	+31.6	+31.4	+31.6
Ocean-Contig	49.7	basic	-2.6	-0.9	-1.1	-4.7	-2.1	+0.4	-5.4	+0.9
		reactive	-0.6	-4.4	+1.8	+3.3	-0.9	+2.5	+1.8	+2.6
		adaptive	-1.3	-2.8	-6.1	-3.5	-1.4	-3.6	-0.4	-3.6
Ocean-Non-Contig	48.2	basic	+5.3	0.0	+2.4	-0.7	+6.4	+1.8	+5.7	-1.2
		reactive	+5.8	+3.1	+2.0	+4.7	+4.0	-1.9	+2.5	+5.5
		adaptive	+7.8	+7.6	-6.3	+2.0	+4.1	+6.6	-8.3	-1.5
Water-Nsq	55.3	basic	-0.6	-0.6	-0.6	-0.6	-0.6	-0.5	-0.7	-0.5
		reactive	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

Table 6.3: Benchmark relative speedups for SLC caching proxies (64 nodes)

ratio can go above one, since only the initial read request is counted in that total, *i.e.* the retries are excluded, and

proxy read ratio: the ratio of the proxy read messages to read requests - this gives a measure of how much proxying is used in an application.

The execution time profile presents the overall execution time split into CPU active time and the time spent waiting because of delays. The delays are further split into load, store, barrier and lock delays. The times are normalised with respect to the execution time without proxies.

The message category profile shows how the total number of messages breaks down into five categories: read, non-caching proxy requests, write, unhook, and proxy messages. The allocation of message types to message categories is given in Appendix C.2.

6.4.1 Barnes

This application has a variable response to the introduction of non-caching proxies. The performance under basic proxies is usually slightly better than seen in Chapter 5, although there are still values of $\mathcal{NPC}$ where the performance is worse with proxies (see Figure 6.7). The proxy hit rate (see Figure 6.8) is much lower than with SLC caching, and this loss of combining leads to a smaller reduction in remote read delay, as shown in Figure 6.6.

The reactive proxies scheme generally improves the performance of Barnes. However the level of unhook messages shown in Figure 6.9 is higher than was observed in Chapter 5. This is a surprising result, given that the non-caching policy will reduce the direct cache pollution caused by caching proxy data. It is caused by an increase in `client-unhook-forward` messages. The delays introduced by the indirect approach of requesting data via a proxy are increased by the reduction in combining. This has a knock-on effect of changing the ordering of sharing lists, and as a result the "normal" unhook requests have to traverse more links from the home node to the node preceding the unhooker. For reactive proxies this effect is most marked at $\mathcal{NPC}$=7.

The side-effect increase in unhook category messages is also observed for adaptive proxies. However it is never as bad as seen for reactive proxies at $\mathcal{NPC}$=7, and, with the overall message total keeping close to the no proxy level, the scheme always improves the performance compared to not using proxies.

On balance, the performance of the Barnes application under the non-caching scheme is comparable to that seen with SLC caching of proxies. However it is interesting that the side-effect, which rearranges the ordering of the sharing list, causes an increase in the number of unhooking category messages despite the reduction in direct cache pollution by proxy data.

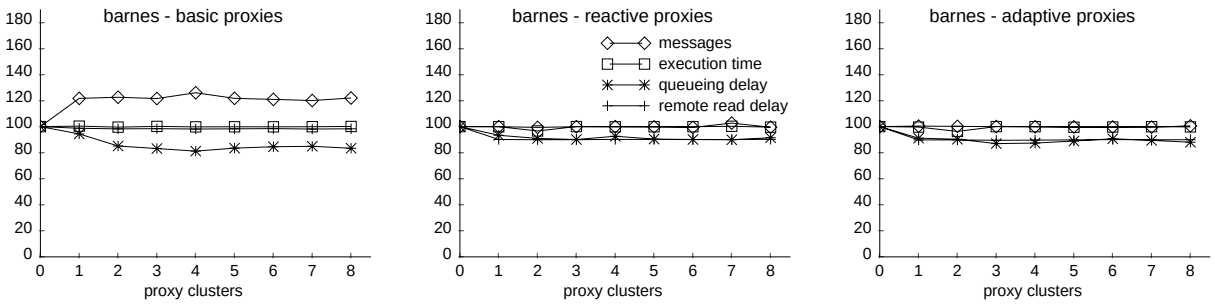


Figure 6.6: Barnes: changes (relative to no proxies case)

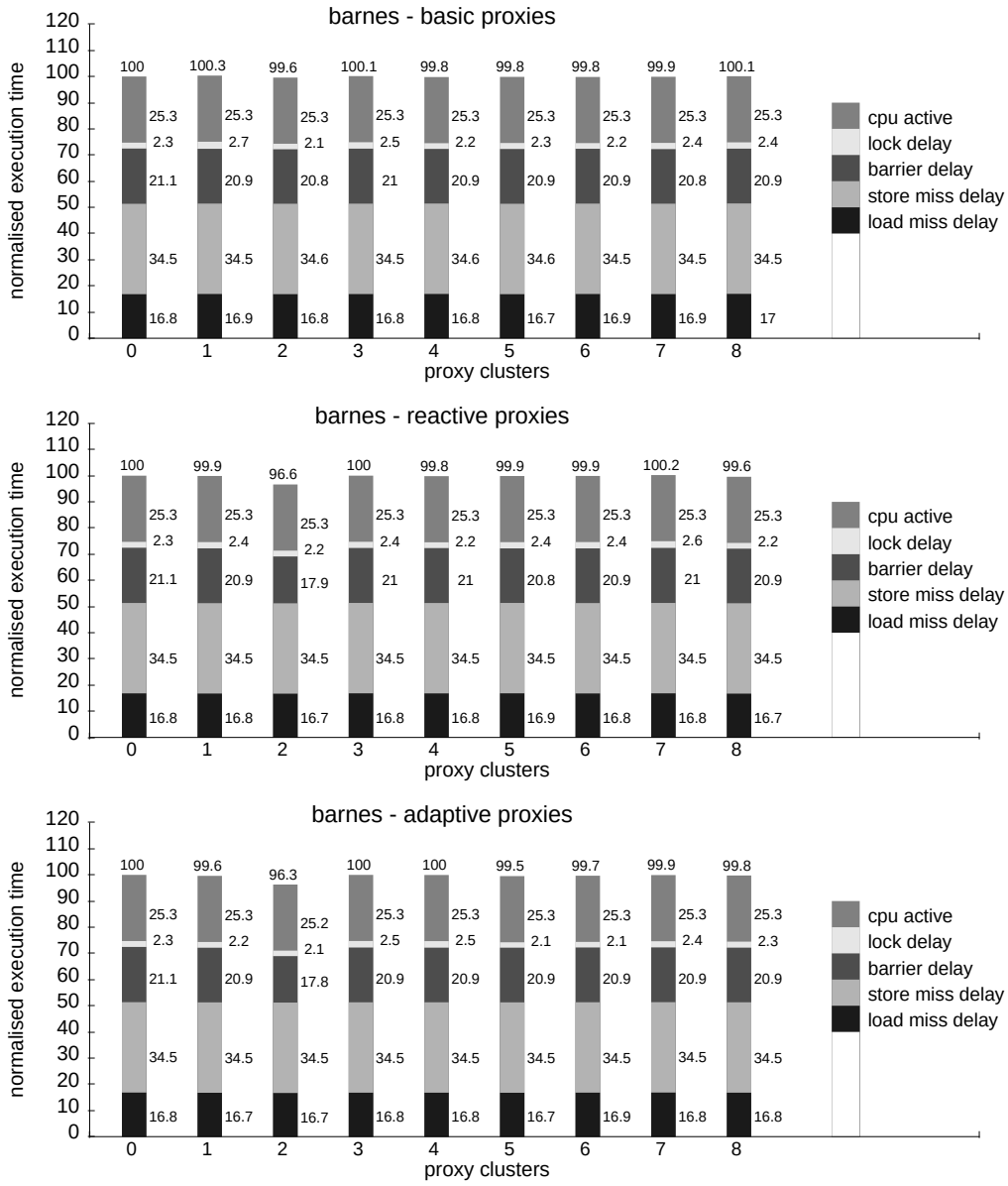


Figure 6.7: Barnes: execution time profiles

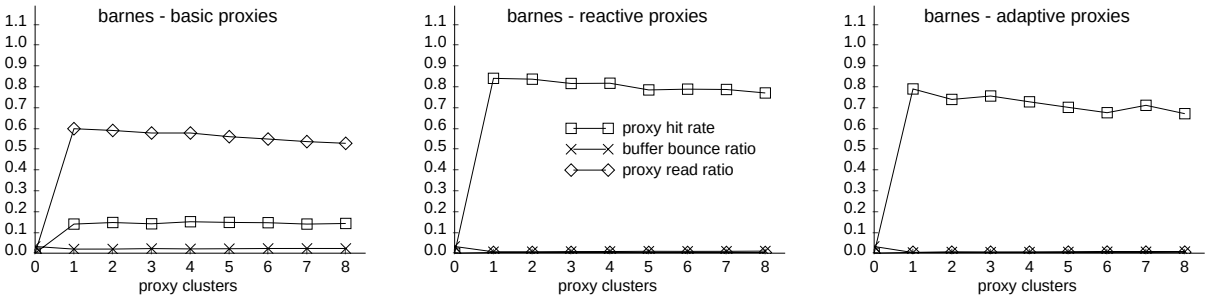


Figure 6.8: Barnes: message ratios

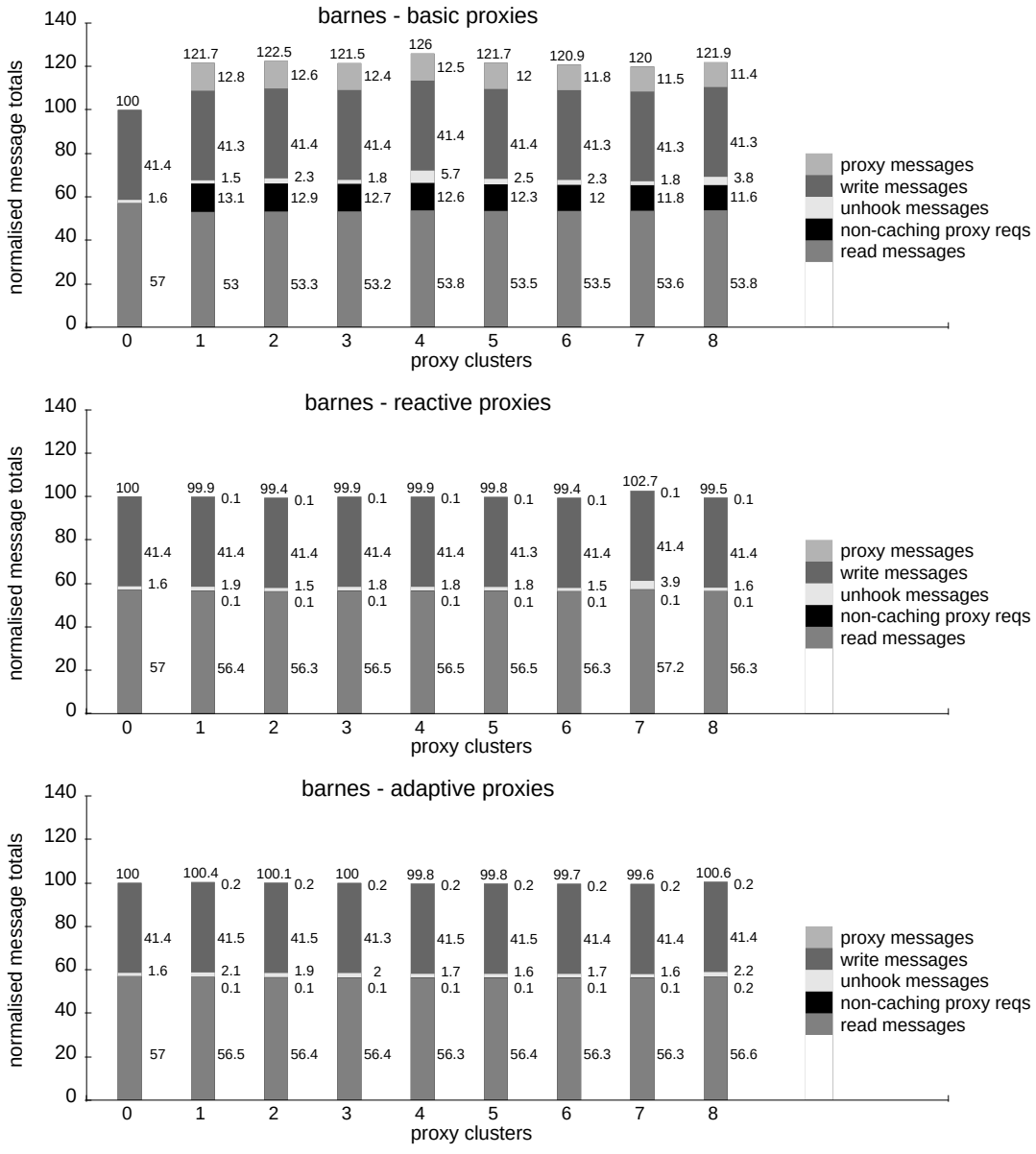


Figure 6.9: Barnes: message category profiles

6.4.2 CFD

This application performs slightly better with non-caching proxies than with SLC caching proxies, *i.e.* the speedups are in the range 6.3% to 15.7% compared to speedups of between 6.1% and 14.8% with the SLC caching scheme. The improvement might seem surprising given that, as shown in Figure 6.13, there is no unhooking for this application. The improvements come from reducing the usage of the SLC bus by the node controller. When a `take-shared` message is received for proxy data, the node acting as proxy no longer has to obtain control of the SLC bus to place the data into the SLC. As a result the local CPU suffers less delay when accessing the SLC, and this is reflected in the larger reduction in load miss delay (shown in Figure 6.11). In addition, because proxy nodes are now only added to the sharing list when the local CPU needs the data, the number of invalidation messages is kept low. This results in the write messages total remaining closer to the level seen with no proxies (see Figure 6.13). Using proxies still brings the benefit of reduced node congestion, which leads to shorter remote read delays, as shown in Figure 6.10.

It should be noted that the proxy hit rate curves shown in Figure 6.12 are similar to those seen for SLC caching proxies (see Figure 5.10). As was observed in the earlier chapter, much of the combining exhibited by CFD comes when the proxy node is also the current owner node for a data line. The non-caching policy for proxies only loses some of the combining which comes from data that is not needed by the local CPU, and this is not a significant factor in the performance of this application.

6.4.3 FFT

For basic proxies, using the non-caching scheme results in slightly better performance improvements than were obtained using the SLC caching approach (see Figure 6.15). The extra improvement is in the range 0.1% to 1.3%. It comes from the reduced store miss delay which results from fewer invalidation messages because there are no proxy copies on the sharing list.

Using the non-caching approach with reactive or adaptive proxies gives good performance improvements when compared with not using proxies. However the improvements are usually not quite as good as were observed using SLC caching of proxy data, with the performance shortfall being up to 0.5%. The shortfall is attributable to different barrier delays resulting from a change to the delays experienced by the local CPUs. This change occurs because there is less competition for the SLC bus since the node controller, when acting as a proxy, no longer has to use the SLC bus to access proxy data lines. However, as shown in Figure 6.15, the barrier delays are always less than without proxies because of the reduction in mean queueing

delay for $\mathcal{NPC}\geq 1$ (see Figure 6.14).

For all three proxy strategies (basic, reactive, and adaptive) the level of combining is comparable to that achieved with SLC caching, as shown in Figure 6.16. This is because the widely-shared data in FFT tends to be accessed by many nodes at the same time, so the non-caching scheme is still able to combine requests at the proxy nodes by allowing requests to join the pending chain. However the hit rate is very low for basic proxies (where all the shared data was marked for proxying) regardless of whether the SLC caching or non-caching approach is used, indicating that some data has been inappropriately marked for proxying.

It should be noted that the results show a slight increase in unhooking category messages compared to the SLC caching scheme examined in Chapter 5. As with Barnes, this increase is caused by an increase in `client-unhook-forward` messages, which occurs because timing delays introduced by using a proxy can affect the ordering of sharing lists.

Overall, FFT achieves performance improvements with the non-caching approach which are similar to those obtained when the proxy data is cached in the SLC. In addition, the basic proxy scheme benefits from the reduction in store miss delay which comes from not needing to invalidate proxy copies (see Figure 6.17) and shorter queueing delays (see Figure 6.14).

6.4.4 FMM

The performance of this application with non-caching proxies is similar to that observed in Chapter 5, with a small performance improvement in the range 0.3% to 0.5%. The performance improvements are once again due to the slight reduction in the load miss delay, as shown in Figure 6.19. There are a few cases, *e.g.* when $\mathcal{NPC}$=8 for reactive proxies, where the load miss delay is not reduced by as much as when SLC caching is used. This shortfall is attributable to the lower proxy hit rate (see Figure 6.20) achieved by non-caching proxies because a proxy node no longer retains a copy of proxied data.

6.4.5 GE

Using the non-caching policy results in the basic proxy scheme attaining even better performance speedups than were observed for SLC caching in Section 5.7.5. The improvements stem from the lower store miss delay that comes from there no longer being proxy copies which have to be invalidated before a write can proceed. As a result there is no increase in the number of write messages (see Figure 6.25), the store miss delay remains at or falls below the level of 6.1% observed without proxies (see Figure 6.23), and nodes spend less time waiting at

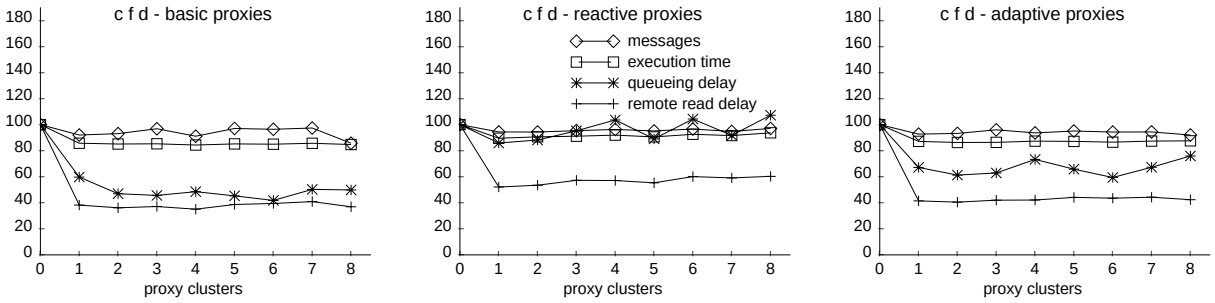


Figure 6.10: CFD: changes (relative to no proxies case)

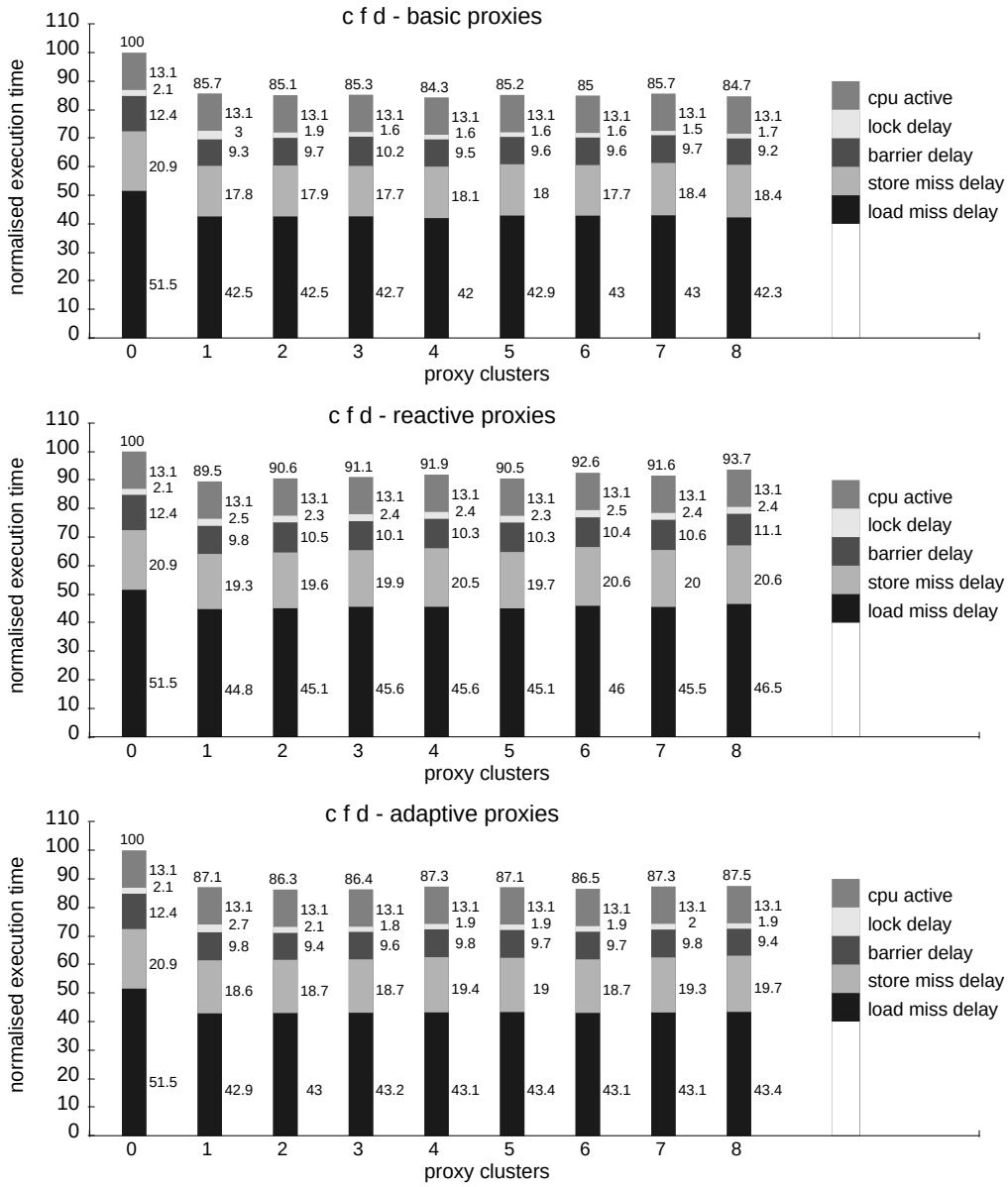


Figure 6.11: CFD: execution time profiles

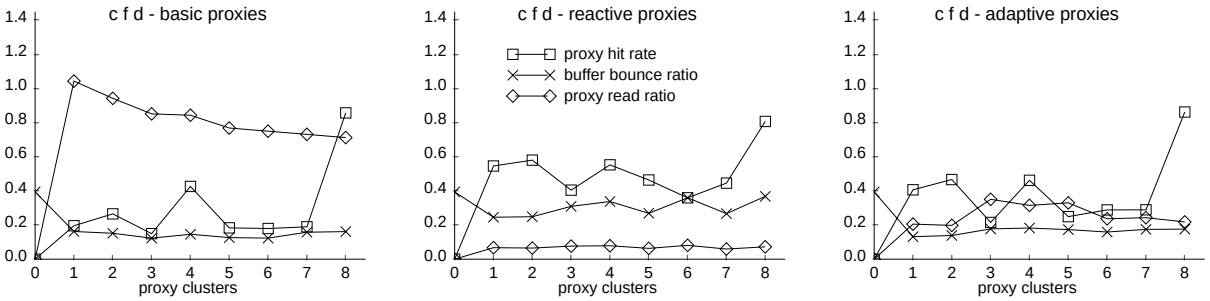


Figure 6.12: CFD: message ratios

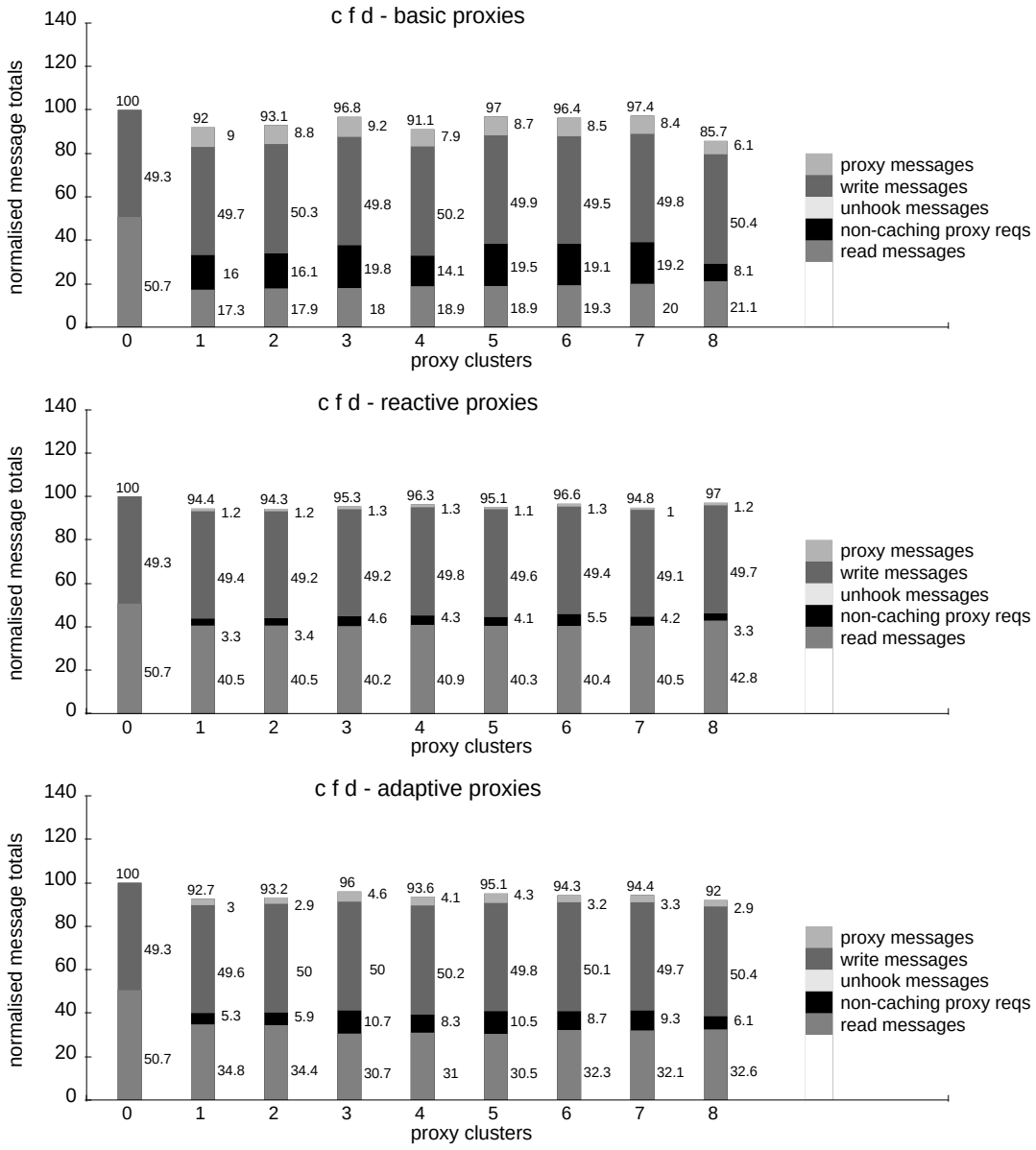


Figure 6.13: CFD: message category profiles

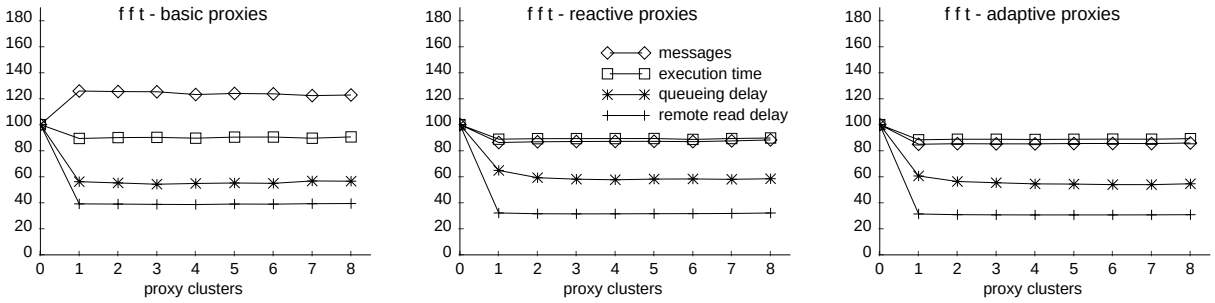


Figure 6.14: FFT: changes (relative to no proxies case)

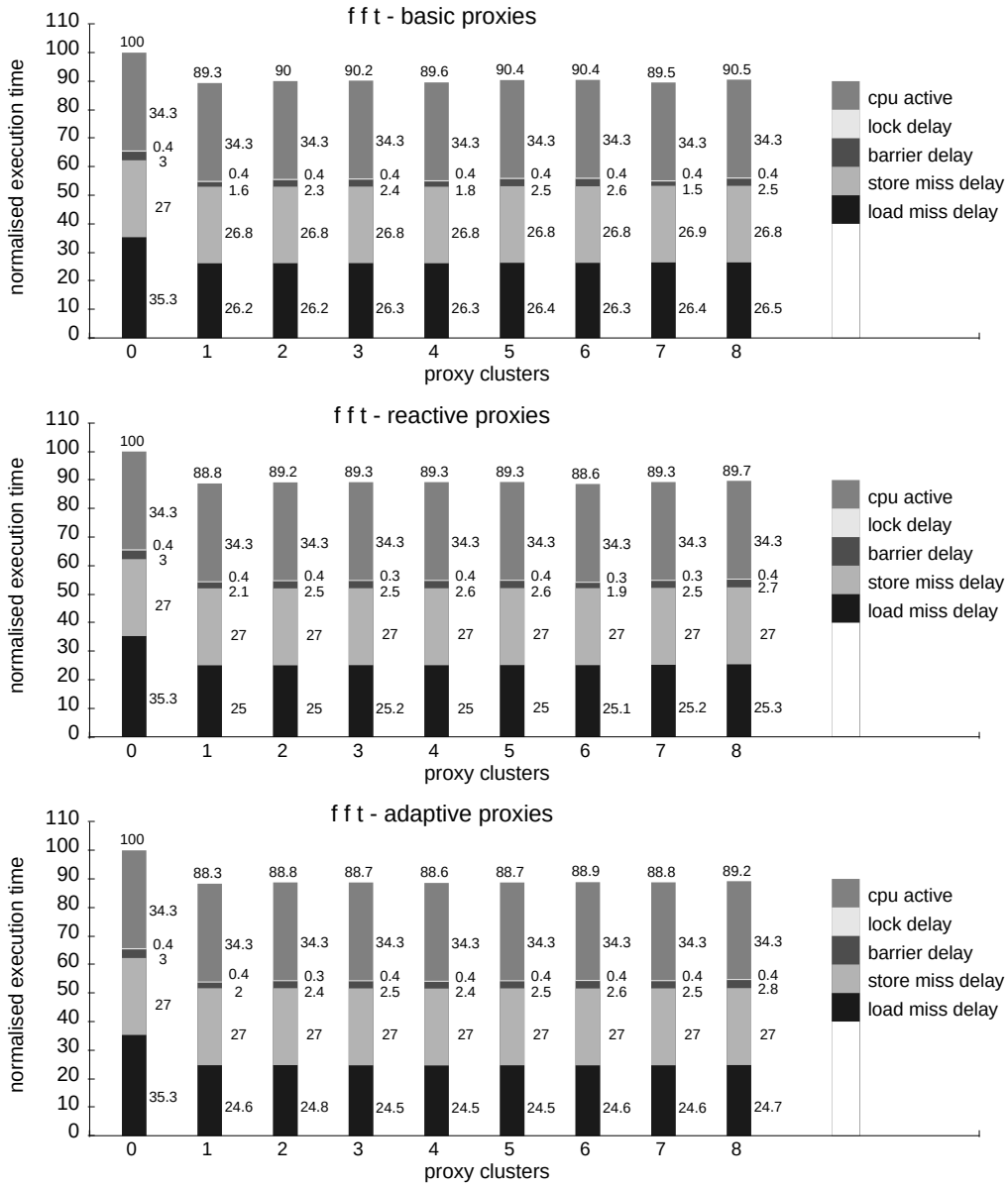


Figure 6.15: FFT: execution time profiles

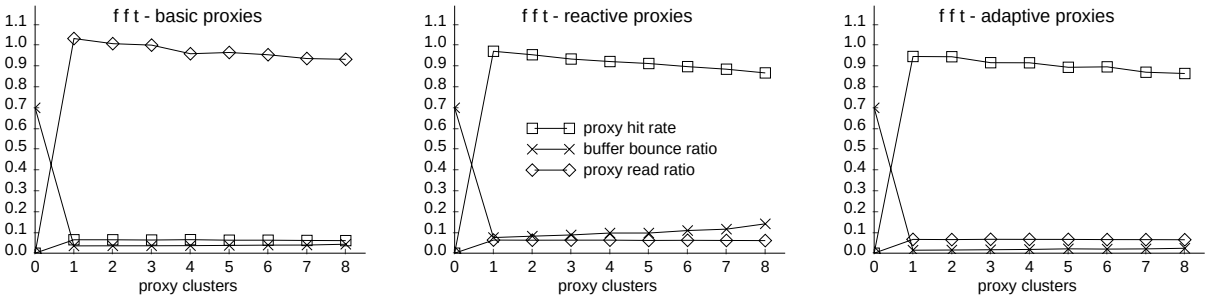


Figure 6.16: FFT: message ratios

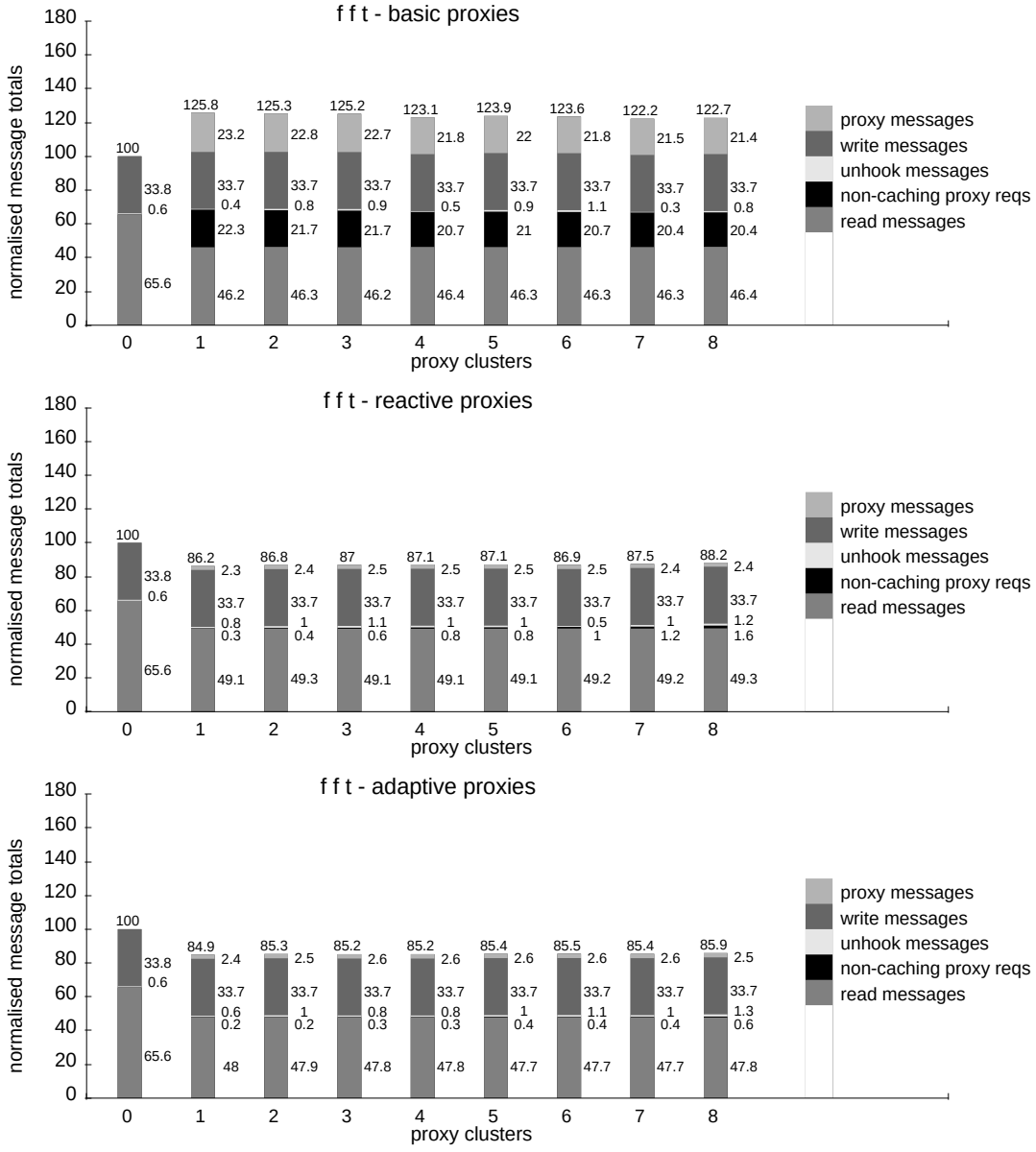


Figure 6.17: FFT: message category profiles

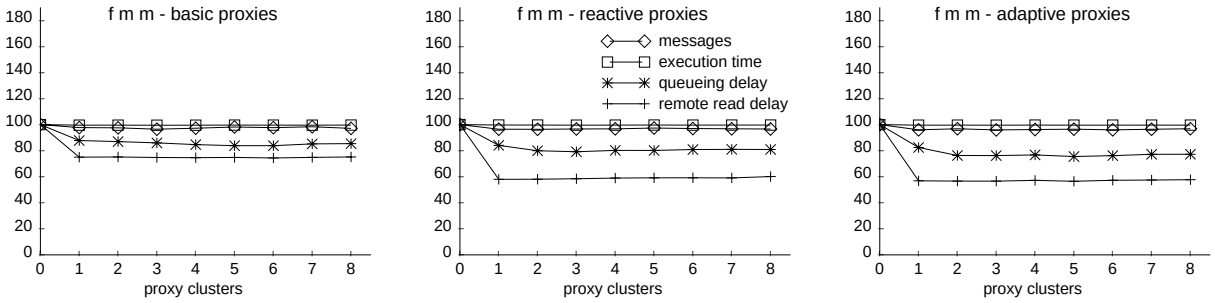


Figure 6.18: FMM: changes (relative to no proxies case)

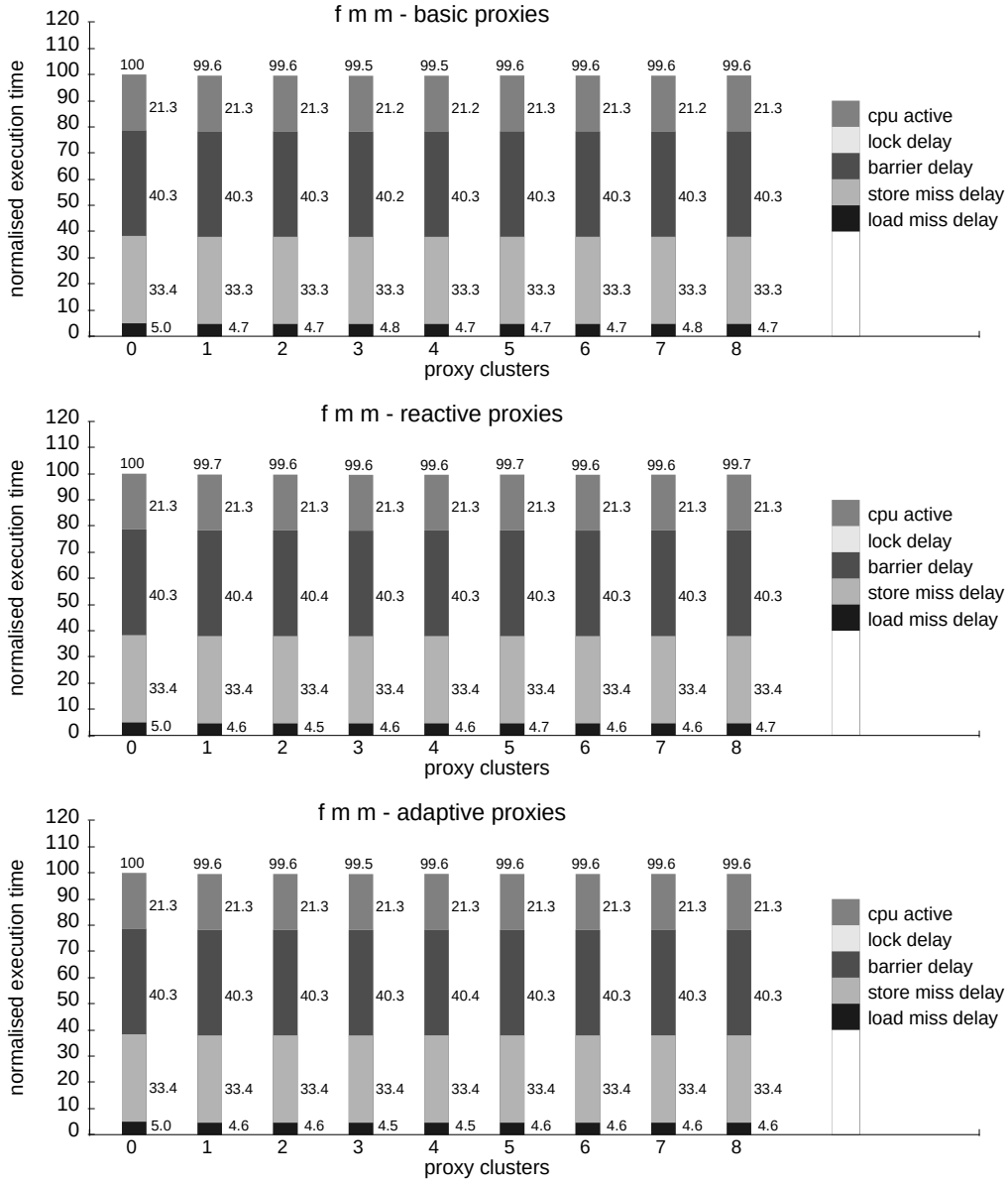


Figure 6.19: FMM: execution time profiles

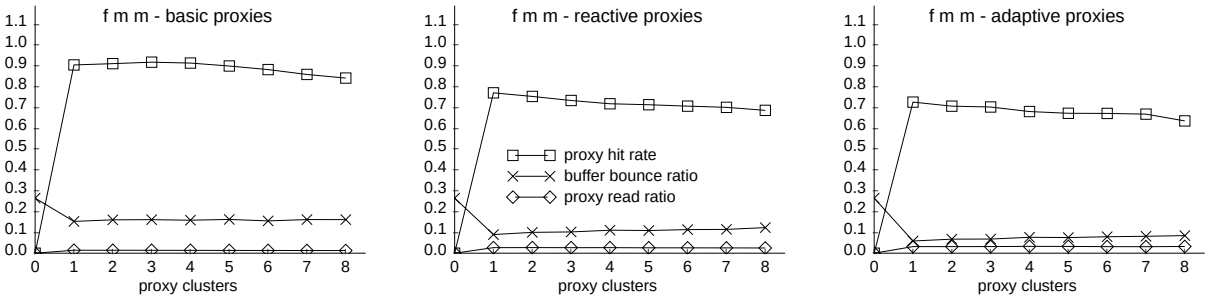


Figure 6.20: FMM: message ratios

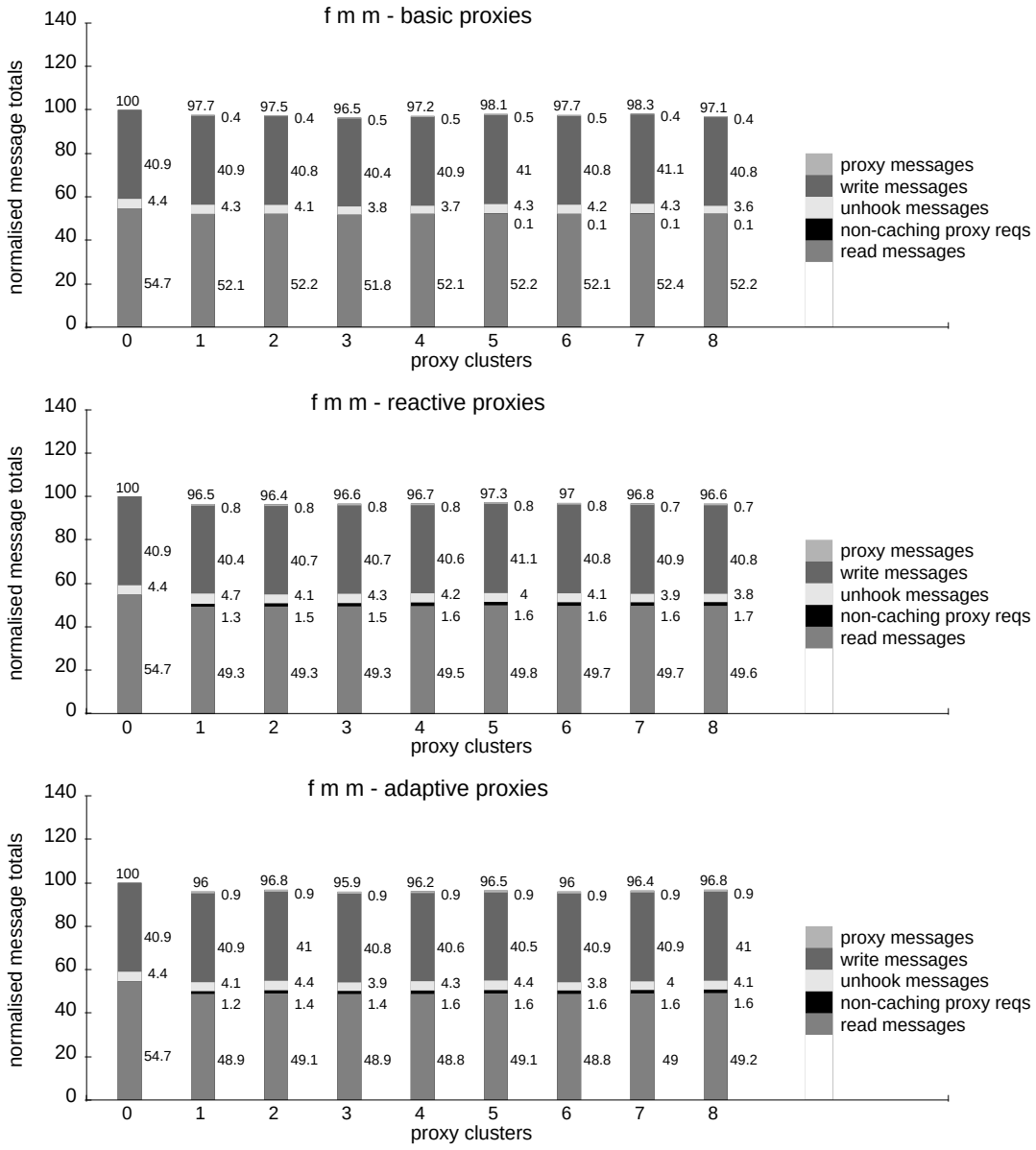


Figure 6.21: FMM: message category profiles

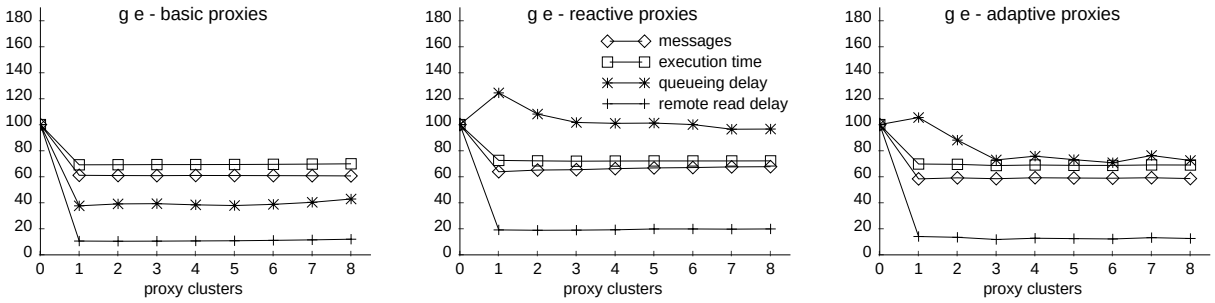


Figure 6.22: GE: changes (relative to no proxies case)

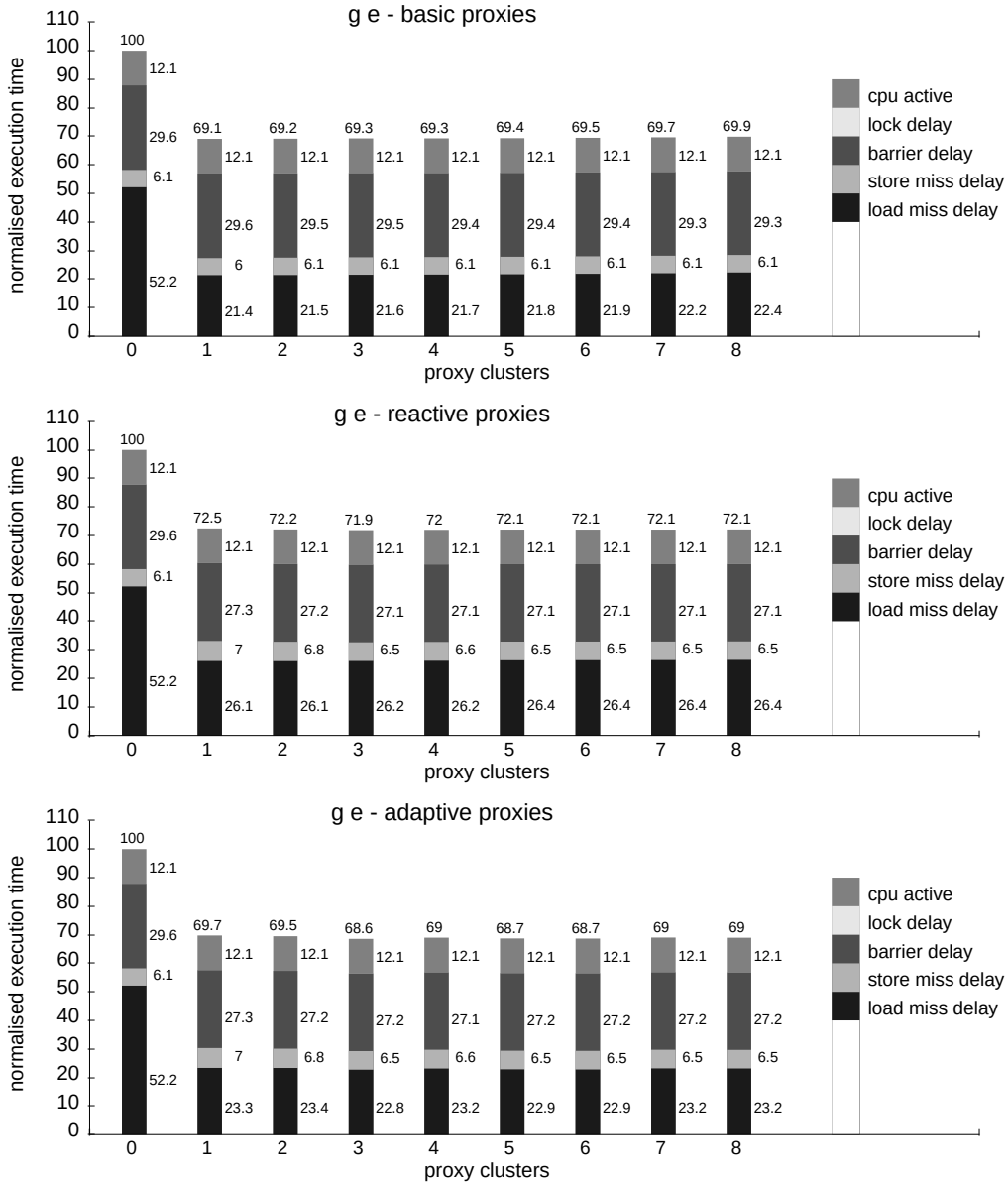


Figure 6.23: GE: execution time profiles

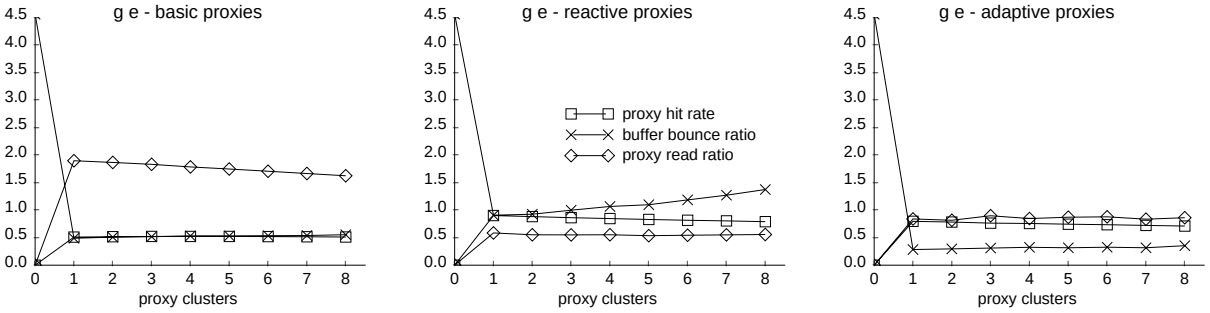


Figure 6.24: GE: message ratios

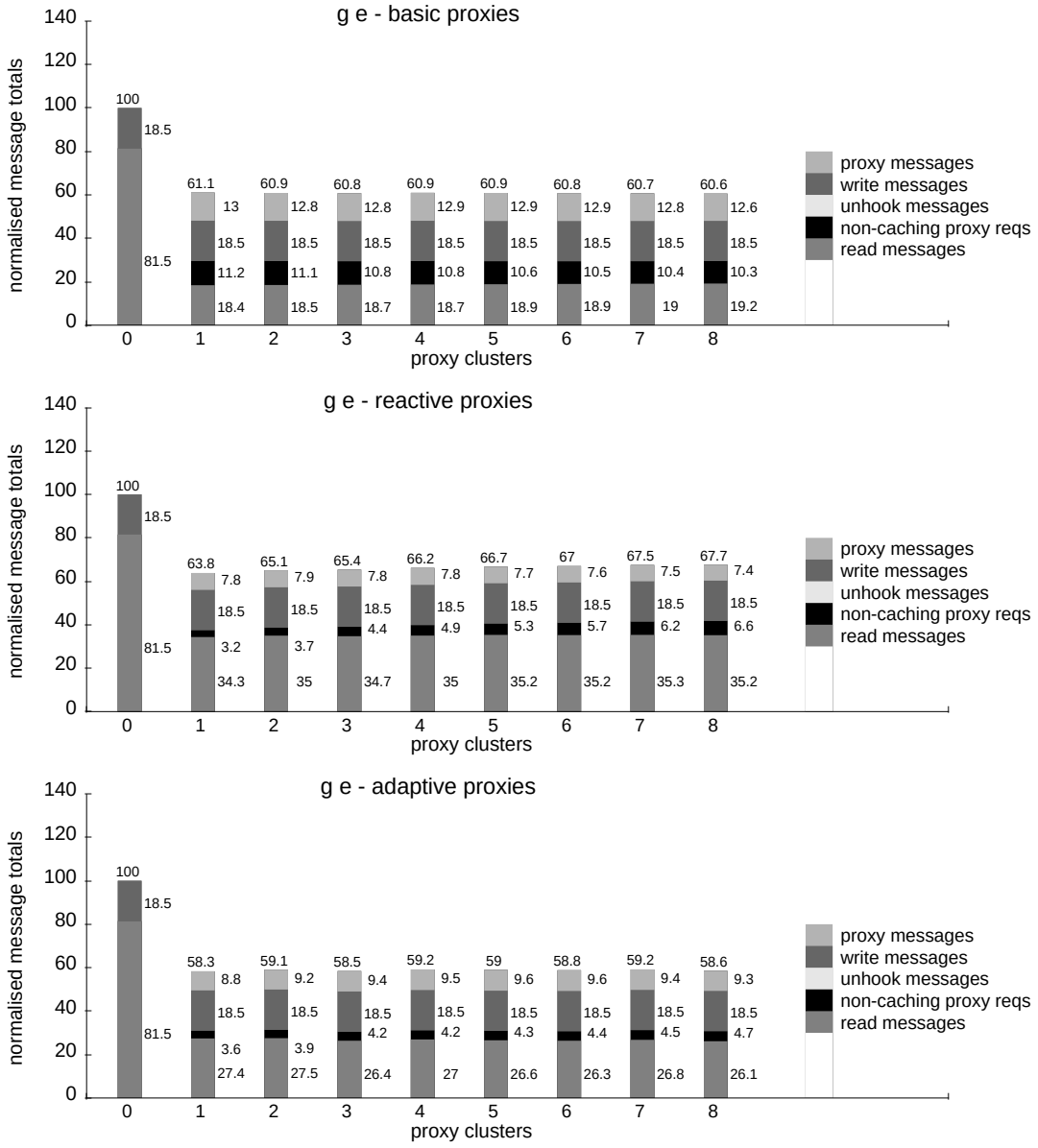


Figure 6.25: GE: message category profiles

barriers for writes to complete. These improvements outweigh the increased number of proxy messages and read messages (including non-caching reads) which are needed now that there is a lower level of combining of `proxy-read-request` messages (shown by the proxy hit rate in Figure 6.24).

The non-caching proxy scheme is not so successful for reactive proxies. Although there is still a large performance improvement, in the range 27.5% to 28.1%, this is not as good as was observed when proxy data was held in the SLC. Although the policy succeeds in keeping the level of write messages the same as without proxies, the overall number of read messages (including non-caching reads) is high compared to the basic proxy policy. This is because of both the higher level of `buffer-bounced-read-request` messages which are used to trigger the reactive proxy policy, and the higher number of `take-shared` messages as the proxy node has to keep re-acquiring a data line for clients rather than using a copy from its SLC. The slightly higher level of proxy messages in comparison to the SLC caching scheme is also a direct result of the non-caching policy. More `take-hole` messages are needed to build the proxy pending chains while the proxy is obtaining data via the home node (data for which a copy would already have been available at the proxy node under the SLC caching scheme).

Adaptive proxies also suffer, albeit only slightly, in comparison to the performance improvements seen in Chapter 5. This small reduction in the performance gain is again due to the slight reduction in combining.

Overall, using non-caching proxies for GE improves the performance of basic proxies further than under the SLC caching scheme because the reduction in proxy copies improves the store miss delay. However the reactive and adaptive proxy schemes suffer in comparison to their performance with SLC caching of proxies, because the reduction in combining restricts the improvement in remote read delay.

6.4.6 Ocean-Contig

For basic proxies, the introduction of non-caching proxies makes little difference to the relative changes profile when compared to the SLC caching scheme, with both messages and queueing delay increasing, and with an execution time which is sometimes better and sometimes worse than without proxies (see Figure 6.26). The effect of non-caching proxies is seen more clearly in Figure 6.28, where the proxy hit rate for basic proxies is almost zero, *i.e.* there is now very little combining at proxy nodes. This results in a slightly higher remote read delay than was observed under the SLC caching scheme. However the increase in load miss delay tends to be balanced in performance terms by the decrease in overall store miss delay which results from no longer having to invalidate proxy copies before a write can proceed.

The reactive proxies strategy also shows mixed performance results in response to the non-caching policy. Although the level of combining is quite high, as shown by the proxy hit rate in Figure 6.28, it is not as good as that achieved with SLC caching, and this causes an increase in the load miss delay. The main influence on performance is the fluctuation in barrier delay, which results from changes in the timing of the arrival of messages at node controllers.

The performance for adaptive proxies shows an improvement over the SLC caching scheme, although the performance is often worse than not using proxies. As with reactive proxies, there is a performance drop when unfortunate proxy selection patterns have a knock-on effect by increasing node controller processing at the proxy node, which delays the local CPU and increases the barrier delay. However the non-caching strategy generally improves the load miss delay (because it results in fewer buffer-bounced read messages), and does not have the adverse effect on the store miss delay which was observed under the SLC caching approach (because there are no proxy copies which need to be invalidated prior to the completion of a write request).

In general, Ocean-Contig is slightly better suited to the non-caching form of proxies than to SLC caching. Most significantly, when $\mathcal{NPC}$=1&2 for adaptive proxies the performance improves, which allows adaptive proxies to get an overall balance point at $\mathcal{NPC}$=1.

6.4.7 Ocean-Non-Contig

Running the Ocean-Non-Contig application under the non-caching scheme with basic proxies improves the performance for all values of $\mathcal{NPC} \geq 1$. This improvement is in contrast to the SLC caching scheme used in Chapter 5, where the performance oscillated. The main factor in the success of the non-caching policy is the reduction in store miss delays, which results from no longer having to invalidate proxy copies. In addition, there is a reduction in unhook category messages because proxy cache pollution has been avoided. By reducing the level of messages in the system, for both invalidate and unhook messages, the application no longer suffers from the buffer saturation experienced for some values of $\mathcal{NPC}$ under the SLC caching strategy.

In contrast, both the reactive and adaptive policies have instances where the store miss delay increases, and these are the values of $\mathcal{NPC}$ for which the overall performance is worse than not using proxies. As shown in Figure 6.32, the cases where the performance degrades (for $\mathcal{NPC} \geq 1$) are those where the buffer bounce ratio is at its highest, namely when $\mathcal{NPC}$=4&5 for reactive proxies, and when $\mathcal{NPC}$=2&4 for adaptive proxies. These cases have the highest level of `buffer-bounced-non-caching-proxy-request` messages, *i.e.* the home node is bouncing

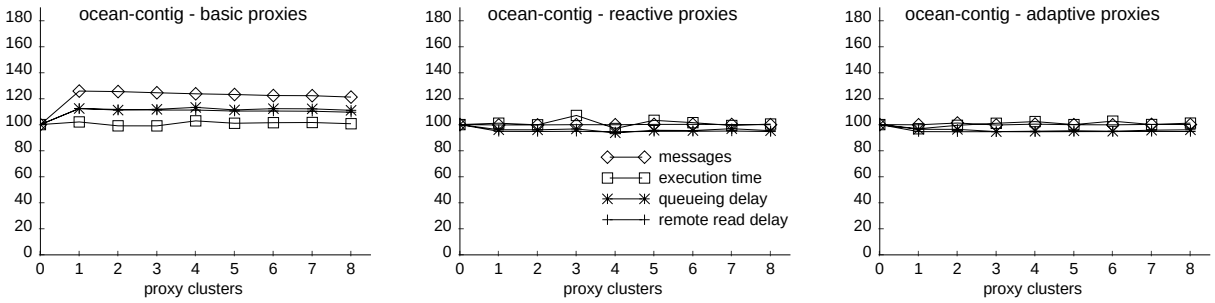


Figure 6.26: Ocean-Contig: changes (relative to no proxies case)

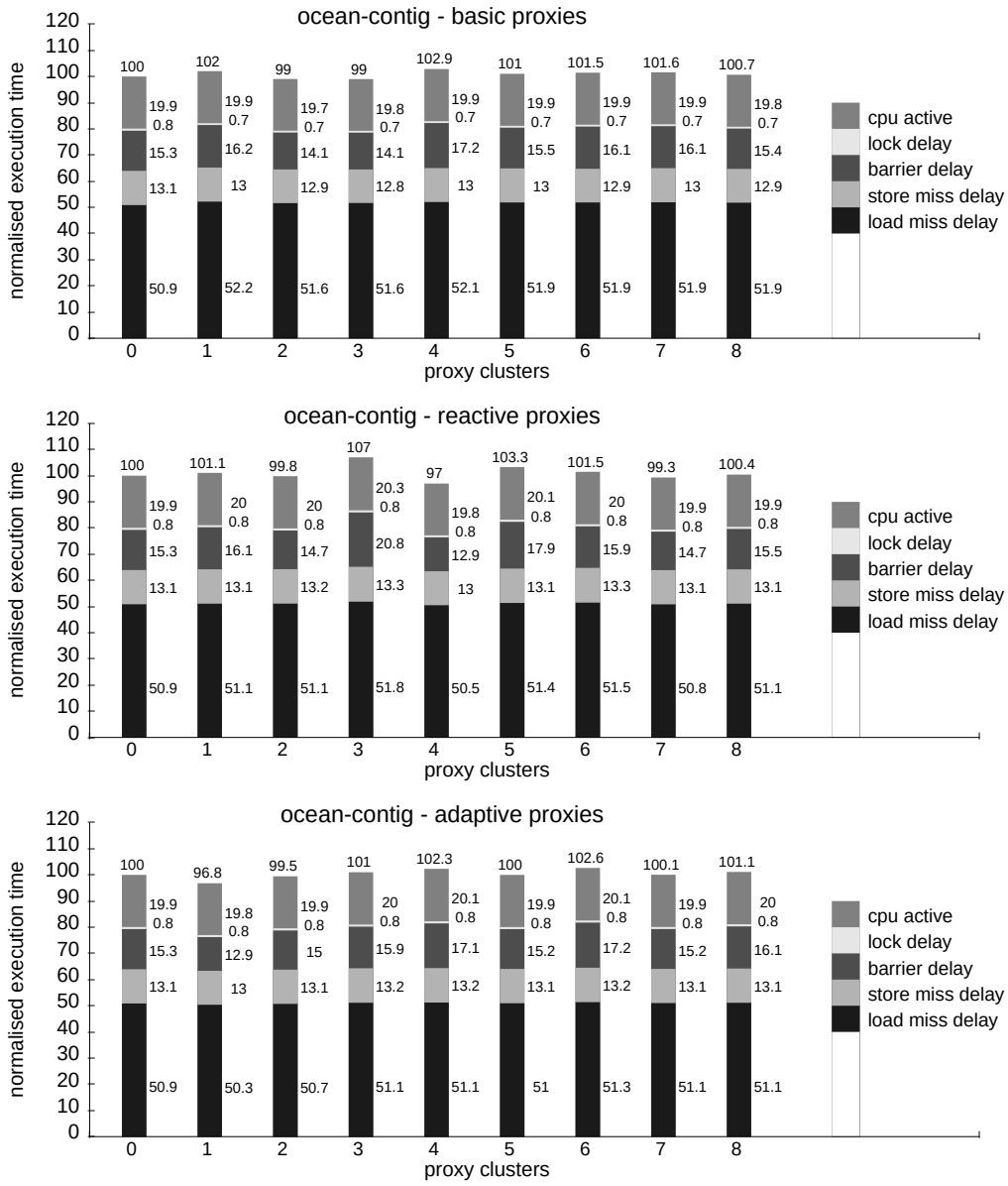


Figure 6.27: Ocean-Contig: execution time profiles

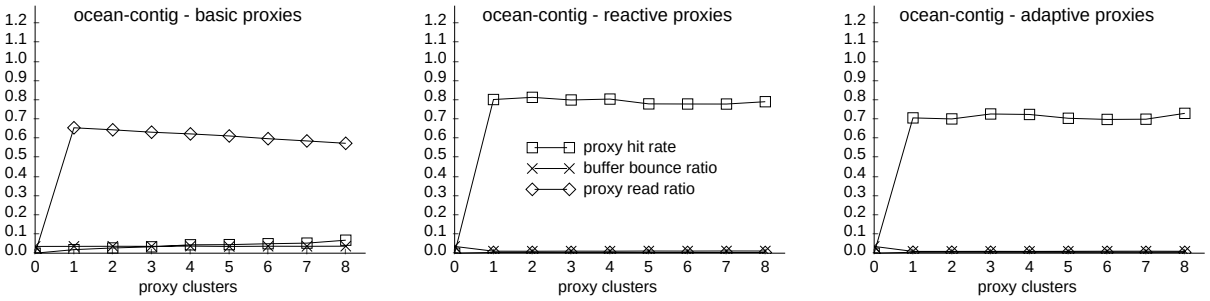


Figure 6.28: Ocean-Contig: message ratios

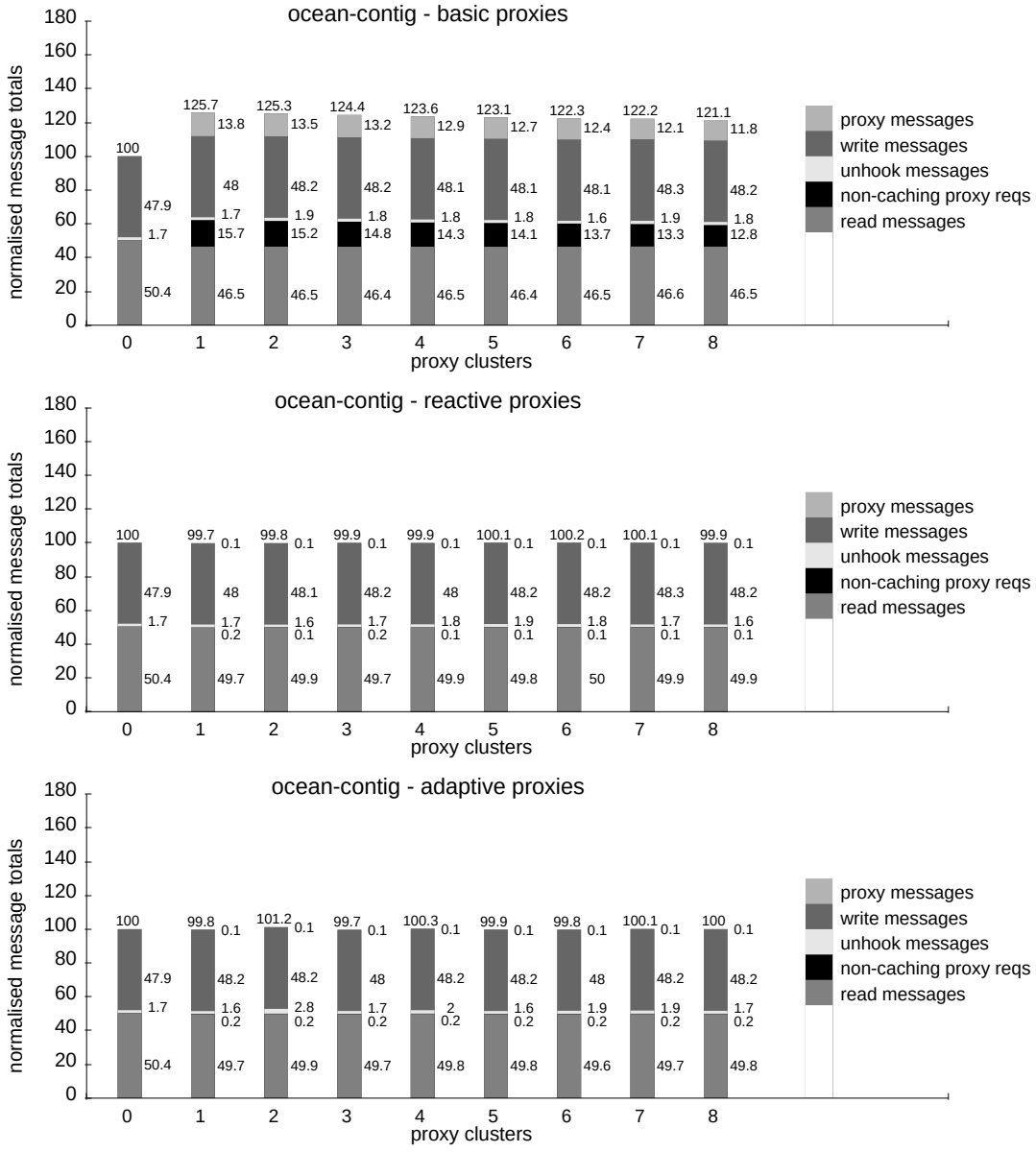


Figure 6.29: Ocean-Contig: message category profiles

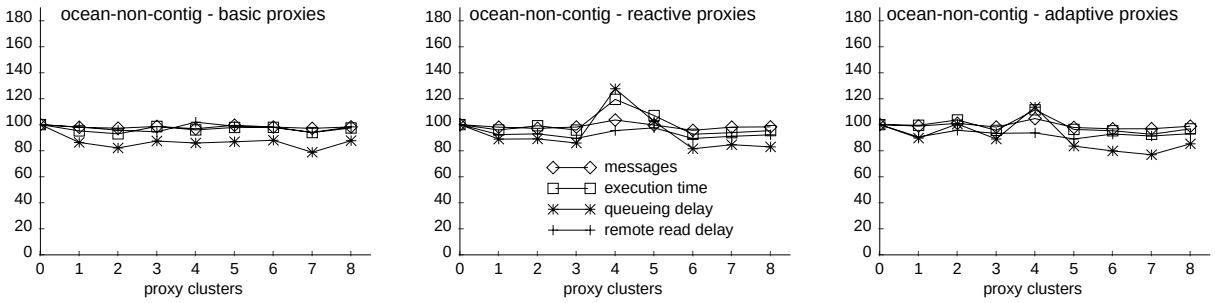


Figure 6.30: Ocean-Non-Contig: changes (relative to no proxies case)

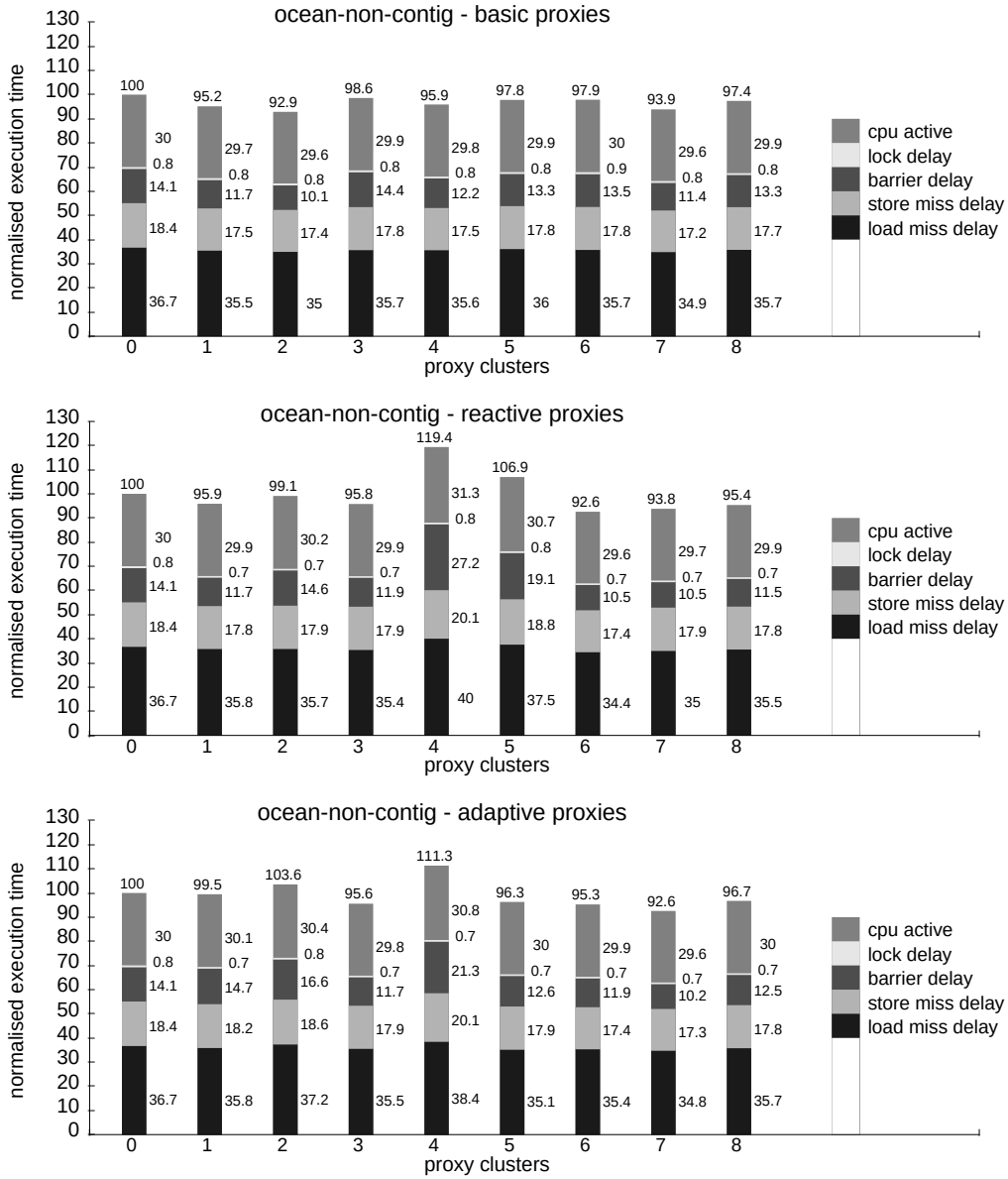


Figure 6.31: Ocean-Non-Contig: execution time profiles

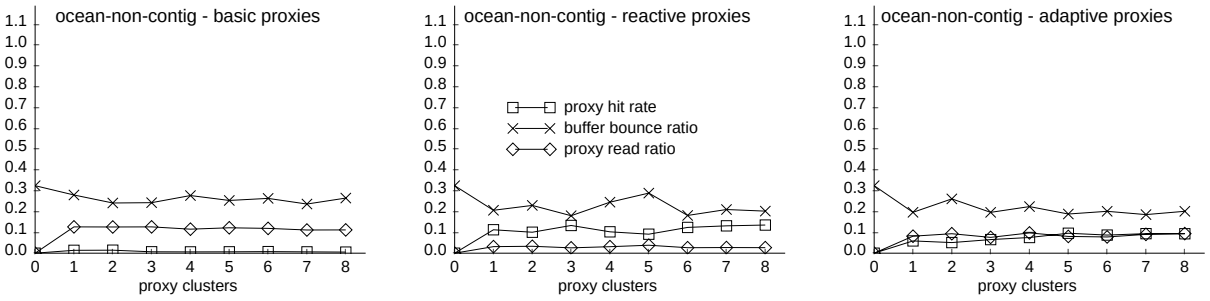


Figure 6.32: Ocean-Non-Contig: message ratios

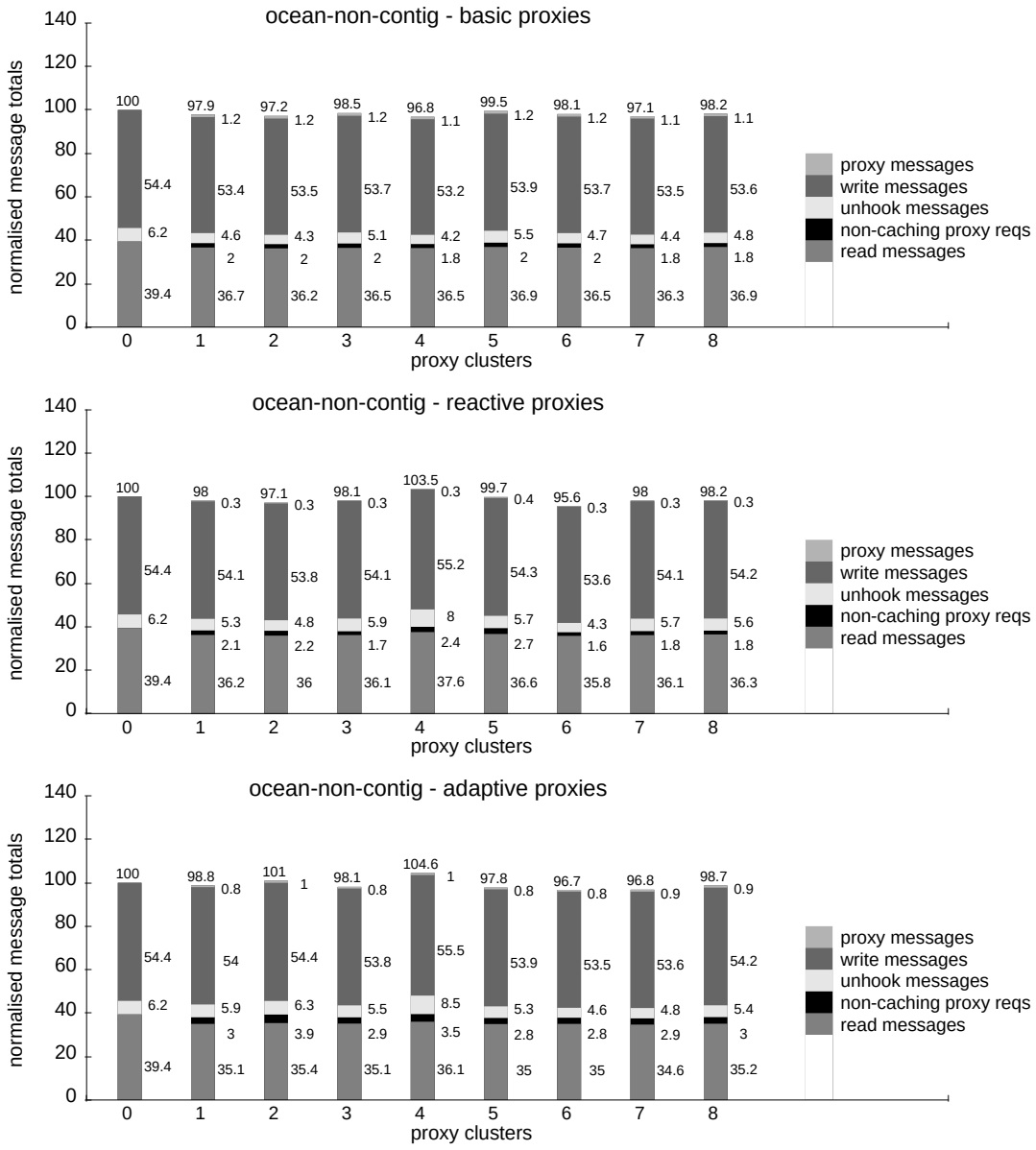


Figure 6.33: Ocean-Non-Contig: message category profiles

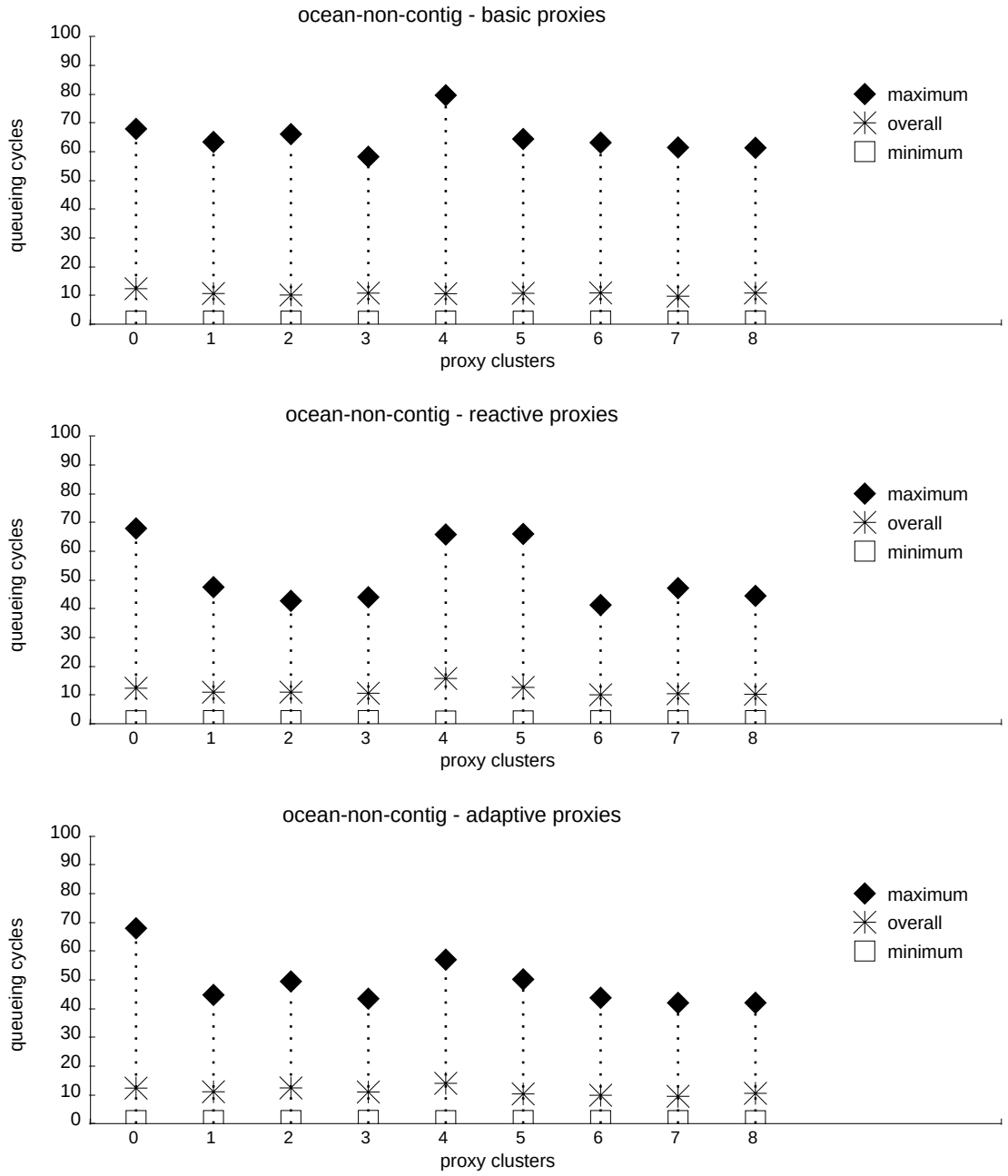


Figure 6.34: Ocean-Non-Contig mean queueing cycles

the requests from a proxy because there are still at least eight messages in the home node controller's input message buffer. This buffer saturation also delays the processing of other messages, and this is reflected by the elevated maximum individual node mean queue lengths shown in Figure 6.34.

An additional feature of the $\mathcal{NPC}$=4 results for reactive and adaptive proxies is the increase in unhook messages (see Figure 6.33). This increase is due to there being more `client-unhook-request` and `client-head-unhook` messages than are seen when $\mathcal{NPC}$=0. Given that the non-caching proxy technique was specifically designed to avoid cache pollu-

tion, it was worrying to observe an increase in the level of SLC unhooks. Careful investigation revealed that the effect is due to a change in home node allocation using the first-touch page placement policy. For this application when $\mathcal{NPC}$=4, the delays introduced by the buffer congestion affect which CPU is first to "touch" some pages of the shared memory. As a result, nodes which were the home node when $\mathcal{NPC}$=0 are no longer the home node. Cache line conflicts which previously caused a replacement to the local memory now require an unhook request to be sent to the remote home node.

Ocean-Non-Contig is best suited to the basic proxies protocol when non-caching proxies are used, because the performance improves for all values of $\mathcal{NPC} \geq 1$. However this result does rely on appropriate data structures having been marked for proxying by the programmer.

6.4.8 Water-Nsq

The performance of Water-Nsq using non-caching proxies is similar to that obtained when proxy data was cached in the local SLC. For basic proxies the performance is slightly worse than not using proxies, whereas both reactive and adaptive proxies get slightly better performance than without proxies.

For basic proxies, comparing the non-caching strategy with SLC caching, the lower proxy hit rate, increase in read messages (including non-caching reads) and the increased latency for overall load miss delays are more than compensated for by the reduction in store miss delay. There are no longer proxy copies which have to be invalidated before a write request can complete. Although the performance is still worse than not using proxies, using the non-caching strategy with basic proxies is less harmful to the performance of Water-Nsq than SLC caching.

The level of combining achieved for the reactive and adaptive proxy schemes (see Figure 6.37) is similar to that seen in Chapter 5. This indicates that the combining for these schemes comes from `proxy-read-request` messages arriving while the proxy node is in the process of obtaining the data from the home node, rather than from requests which arrive later and rely on there being a copy of the proxy data still held at the proxy.

6.4.9 Summary of Results

The simulation results for the eight applications have shown that the proxying technique is still effective even when the opportunities for combining are restricted because proxied data is not held in the local SLCs. The non-caching proxy policy was introduced as a way of reducing

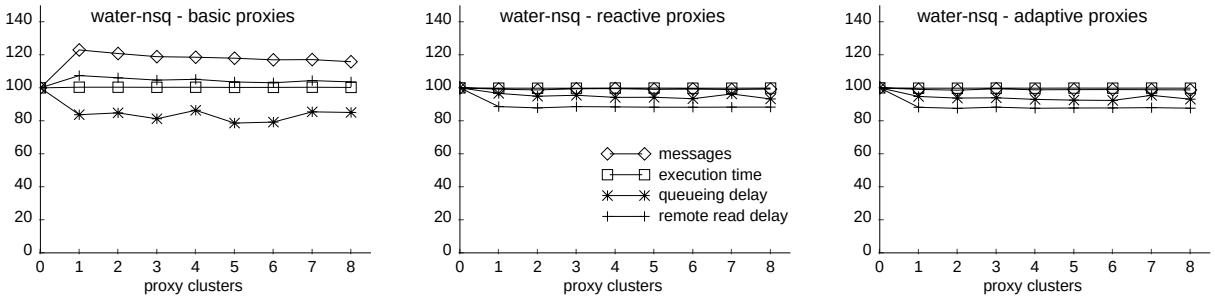


Figure 6.35: Water-Nsq: changes (relative to no proxies case)

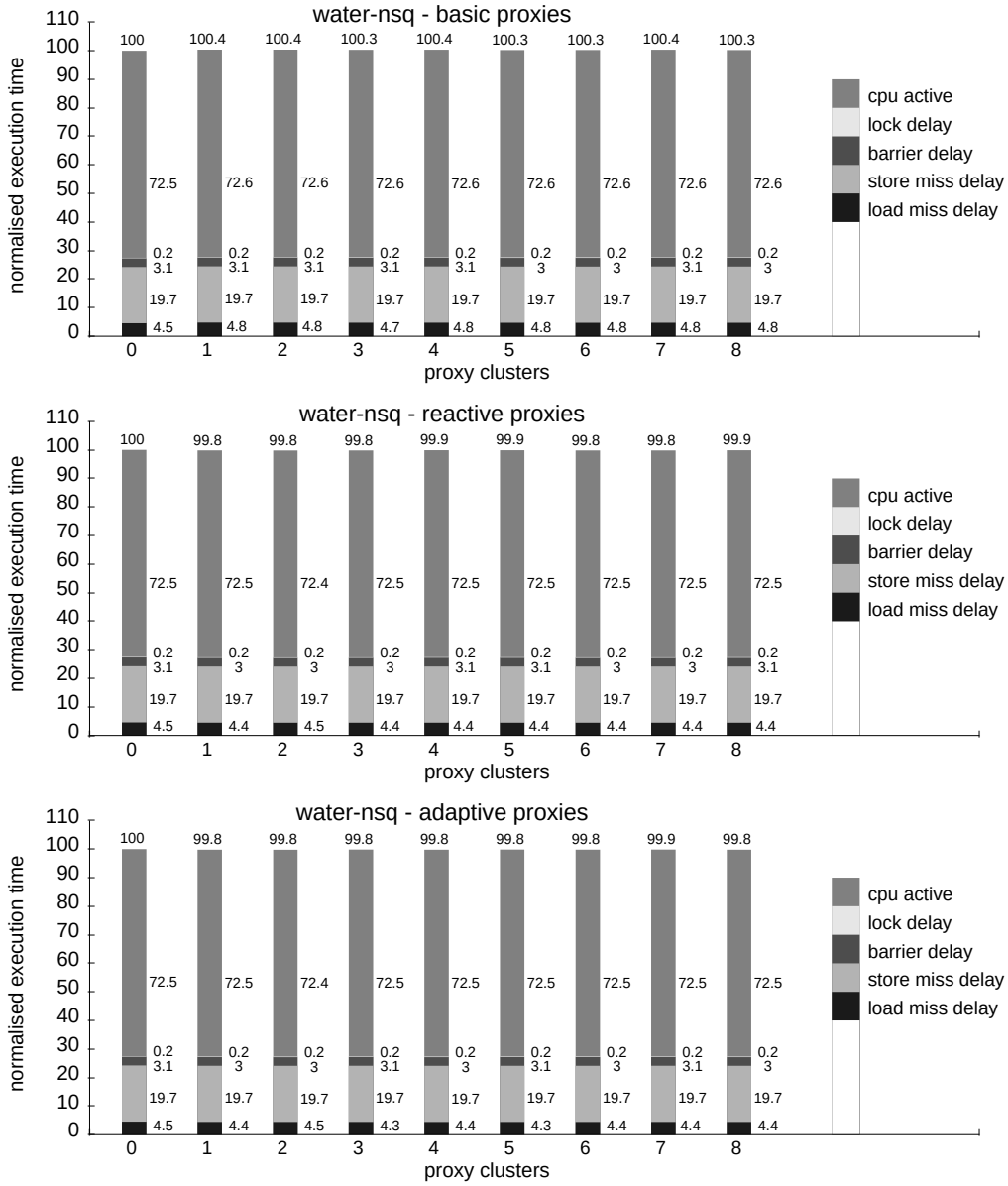


Figure 6.36: Water-Nsq: execution time profiles

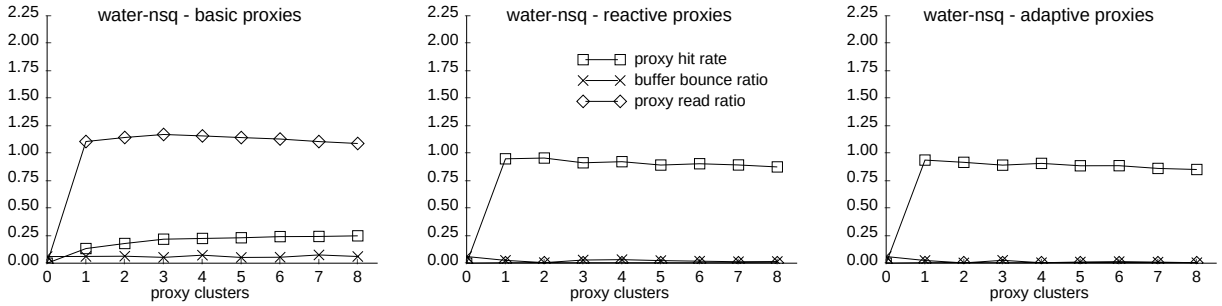


Figure 6.37: Water-Nsq: message ratios

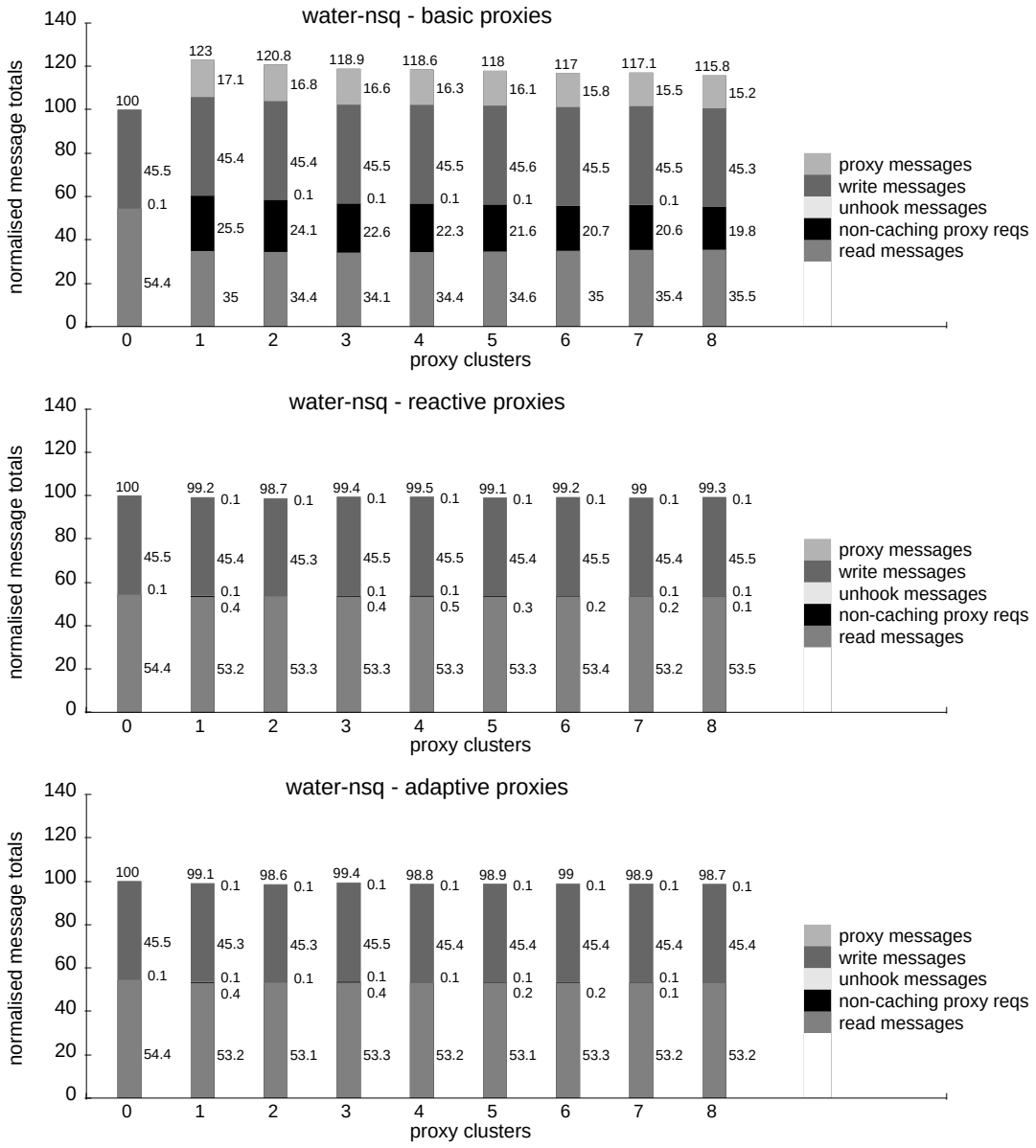


Figure 6.38: Water-Nsq: message category profiles

cache pollution, and this was effective for the basic proxies strategy. However the performance improvements were also due to a reduction in contention for the SLC bus because the node controllers no longer needed access to the bus in order to load proxied data lines into the SLC.

The level of messages related to unhooking rose for some of the simulations, *e.g.* for the Barnes and Ocean-Non-Contig applications using adaptive proxies. Investigations into the effect showed that the use of proxies introduces delays which can affect (1) the order in which entries appear on sharing lists, and (2) the allocation of pages to home nodes. These side-effects can cause the overall level of unhooking messages (including the forwarding of unhooking messages along the sharing list) to increase.

The increase in client-unhook-forward messages which arises because the unhooking node is further along the sharing list, is an effect of using a singly-linked list to represent the sharing list in the Stanford distributed-directory protocol [134]. If the sharing list had been represented in another way, for example as a doubly-linked list or a bit vector, then the "position" of a node in the list would not have changed the number of messages required to complete an unhook transaction. Such alternative implementations might improve the performance results for applications such as Barnes, and an investigation of the alternatives should be the subject of further work.

Reactive proxies have a single balance point at $\mathcal{NPC}$=2, *i.e.* the value of $\mathcal{NPC}$ where the performance of all eight applications is improved. The lack of balance points for $\mathcal{NPC}$> 2 is due to the lower combining opportunities when there are more proxies for a particular data line. It should be noted that adaptive proxies also have a balance point using non-caching proxies, unlike when the SLC caching scheme was used. The balance point occurs when $\mathcal{NPC}$=1, *i.e.* when the chances of combining are greatest because there is only one proxy node for a given data line. The Ocean-Contig application, which suffered under adaptive proxies with SLC caching, is able to benefit from the reduced cache pollution at $\mathcal{NPC}$=1, and so achieves a balance point.

6.5 Conclusions

The non-caching strategy can be used to support proxying in a system where the node controller does not have direct access to the local SLC. The strategy reduces contention for the SLC bus between node controllers acting as proxies and the local CPUs, and avoids polluting the SLC with proxy data. However the timing and sharing list order side-effects introduced by the reduction in combining (because proxied data is no longer retained) result in variable

performance improvements. Given that the reduction in combining can exacerbate some of the side-effects of proxies, it is prudent to investigate the other policy which avoids using the SLC, namely having a separate buffer to hold proxied data. That approach is examined in the next chapter.

Chapter 7

Using a Separate Proxy Buffer

The effects of not holding a copy of proxied data were examined in the preceding chapter. It was found that SLC pollution was reduced, but the reduction in combining of `proxy-read-request` messages increased the number of read requests sent by proxies to home nodes. This increase in the overall number of messages sometimes degraded the performance. An alternative way of avoiding SLC cache pollution is to have a separate small buffer for proxy data at each node. This has the potential to deliver a greater level of read request combining than was achieved with the non-caching approach. The separate proxy buffer approach is investigated in this chapter, and the results are compared with both the original SLC caching scheme and the non-caching scheme.

7.1 The Proxy Buffer

The proxy buffer is a separate area of storage used to hold the data lines which a node has obtained when acting as a proxy. When a copy of a data line arrives in a `take-shared` message, and the data is not needed for local processing, the data line will be inserted into the proxy buffer. It will stay there until it is invalidated, or it is "promoted" to the local SLC (in response to a local read miss for the data), or it is evicted to make room for a new data line of proxied data. The existence of the proxy buffer means that the node controller has to be modified for local read misses, and for handling `proxy-read-request`, unhooking, and invalidation messages: there has to be an additional check in the proxy buffer to see if there is a matching entry. Using the proxy buffer rather than the SLC to hold proxy copies of data should reduce both cache pollution and contention for the SLC bus. In addition the scheme will maximise the opportunity for satisfying read requests at the proxy node, unlike the non-caching policy evaluated in Chapter 6.

7.1.1 Related Work

The idea of using a small fully-associative buffer to hold data lines which conflict with current cache entries, termed a victim cache, was introduced by Jouppi [63]. Victim caches are small (*e.g.* 8 entries) fully-associative caches which are used to hold data lines which have just been evicted from the FLC because of conflict or capacity misses. If the processor requires the victim data soon after its eviction, it can be retrieved from the victim cache with a latency which is only slightly higher than a cache hit.

Using a separate cache area to hold a copy of data obtained from remote nodes is an idea which has been implemented in a number of forms, with examples including the Remote Access Cache in Stanford's DASH [89], the Interconnect Cache (IC) in the HP/Convex Exemplar [1], and the Remote Data Cache in the Sequent NUMA-Q [92]. In the DASH system, the Remote Access Cache (RAC) is a 128 Kbyte direct-mapped cache which acts as a staging area to receive and buffer replies from remote clusters. RAC entries are allocated when a remote request is issued (by one of the four local processors) and persist until all inter-cluster transactions relating to the request have completed. If there is a conflict for a RAC entry, the later request is delayed and then retried after the earlier request has finished with the RAC entry. The RAC is also able to detect when more than one of the local processors are accessing the same remote location: in this case the RAC combines the later request with the earlier one, and satisfies both requests when the reply to the earlier request is returned.

In the HP/Convex Exemplar, the Interconnect Cache (IC) is a dedicated section of each node's memory which is designed to improve locality of reference for data with a remote home node [1]. When a remote shared-memory access misses in both a processor's data cache and the node's IC, a memory line is retrieved over the SCI ring interconnect which passes through the home node. This line is then stored in the local IC as well as in the processor's data cache. As a result, subsequent references to this line from other processors at the node can be satisfied locally from the IC, until the line is replaced or invalidated by a remote node. The size of the IC is set by the system administrator to achieve the best performance for frequently executed applications.

The Sequent NUMA-Q is another design which uses a separate cache to hold copies of data lines obtained from remote nodes [92]. The Remote Data Cache (RDC) is a 32 Mbyte 4-way associative cache. Local node data accesses which hit in the RDC are satisfied with a latency which is similar to a local memory hit. Data is copied into the RDC as part of supplying remote data to a local processor.

7.1.2 Potential Benefits and Costs of a Separate Proxy Buffer

The proxy buffer scheme should have a number of effects. Among the benefits one would expect are:

More opportunities for combining: node controllers will use the proxy buffer to hold a copy of each data line obtained while acting as a proxy. This data will then be available to satisfy subsequent `proxy-read-request` messages from clients. A data line will remain in the proxy buffer until it is out-of-date (invalidation), or it is evicted to make room for a new entry, or it is promoted to the local SLC in response to a local read miss.

No extra cache pollution from using proxies: the local SLC does not hold proxy data.

The potential costs are:

Additional protocol processing: needed to manage the proxy buffer, *i.e.* for adding, invalidating, and evicting entries.

Data storage: the proxy buffer will take up a (small) part of the local memory (DRAM), or be part of the node controller, or there will need to be a small buffer added to the MEM bus. The latency of accessing the proxy buffer depends on its location.

These potential costs and benefits are considered as part of analysing the results in Section 7.4.

7.2 Design Issues

The issues surrounding the implementation of the proxy buffer are concerned with how to incorporate the existence of the separate buffer into the protocol, and how to manage the buffer.

7.2.1 Using the Proxy Buffer

Using a separate proxy buffer to hold the data which a node has obtained as proxy introduces the problem that there is now an additional location which has to be checked for the existence of a data line copy. This extra checking is needed for local cache misses, and for the processing of various external requests such as invalidations, unhooks, and `proxy-read-request` messages. These checks of the proxy buffer are required when:

1. **A local read miss occurs**: the node controller needs to check if the data is currently in the proxy buffer. If it is, the data line is "promoted" to the SLC, *i.e.* the tag, data, and sharing list pointer are copied to the SLC and the proxy buffer entry is deleted. No further action is needed before the CPU is restarted because the node is already on the sharing list for the data line.

2. **A local write miss occurs**: the node controller needs to check if the data is currently in the proxy buffer. If it is, the data line is "promoted" to the SLC, *i.e.* the tag, data, and sharing list pointer are copied to the SLC and the proxy buffer entry is deleted. Processing then continues as if the write had hit in the SLC on a clean shared data line, *i.e.* the node will have to obtain exclusive access to the data line before the write can proceed. This promotion of the proxy buffer entry to the SLC minimises the changes to the original protocol processing.

3. **An invalidate message arrives**: the node controller must check to see if there is a matching entry in the proxy buffer. If there is, the entry must be deleted from the proxy buffer. As for a "normal" invalidate match in the SLC, the node controller must then pass the invalidation on to the rest of the sharing list.

4. **An unhooking message arrives** (`client-unhook-forward` or `client-unhook-ptr`): if there is a match in the proxy buffer, the node controller processing is similar to a match in the SLC, apart from using the proxy buffer pointer data rather than the pointer from the SLC.

For this work it was decided that accesses to the proxy buffer would have the same latency as accesses to the local DRAM. This can be seen as implementing the proxy buffer in DRAM, which would be possible using the IC of the HP/Convex Exemplar [1], or in the Stanford FLASH architecture where the MAGIC chip can adapt the use of the DRAM to suit each protocol [120]. This approach is more realistic than making the data instantly accessible within the node controller. It also allows the buffer size to be tuned to suit the multiprocessor's configuration, *e.g.* the buffer size might need to be increased as more processing nodes are added to a system.

7.2.2 Managing the Proxy Buffer

In order to use the proxy buffer, decisions had to be made on the policies for adding, deleting, and evicting entries from the buffer, and how to handle the potential overflow given that the buffer is very small. The proxy buffer is fully associative, *i.e.* a data line can be placed anywhere in the buffer. Entries will be deleted from the proxy buffer in response to an

invalidation message or because the entry is being "promoted" to the SLC to satisfy a local read miss. When an entry has to be evicted to make room for a new entry (*i.e.* when the buffer is full), it was decided, for simplicity, to use a FIFO policy.

An entry is chosen for eviction when it is the oldest entry and there is no room in the buffer for a new entry. The data line copy will have to be removed from the sharing list, so an unhook request is sent to the home node. The data line being evicted is moved to an overflow region for entries which are in the process of being unhooked, and this leaves room in the buffer for the new entry. Entries held in the overflow region can still be accessed by the node controller, so the data can be used to satisfy `proxy-read-request` messages and local read misses for the data line. When the unhook request completes, the entry will be deleted from the overflow region.

A new entry will be added to the proxy buffer whenever a `take-shared` message arrives at a node and the node does not require the data for local processing (*i.e.* the data is only needed by the client nodes). The data line will be inserted in the first available slot in the buffer. If it is not possible to evict the oldest entry when the buffer is full, because the overflow area is currently filled with entries pending unhook, the processing of the `take-shared` message will be suspended until there is room in the proxy buffer for the new entry.

7.3 Modifications to the Protocol and Architecture

The extra states, state transitions, and message types needed to support the use of the proxy buffer are illustrated in Figure 7.1 and Figure 7.2. A separate set of unhooking messages has been introduced for the proxy buffer. This was done for two reasons: to avoid over-complicating the existing processing for normal SLC unhooking, and to make it easier to distinguish between proxy buffer and SLC unhooking for reporting purposes. In fact, separating the processing for the two types of unhooking might well reflect the implementation on a programmable node controller, where the standard SLC unhook protocol processing would be implemented in hardware, with the much less common processing for proxy buffer unhooks being implemented in software. The proxy buffer unhook message types are described in detail in Appendix C.2.6.

The proxy buffer has been implemented in ALITE as a wraparound first-in-first-out (FIFO) buffer with 32 entries. The first 16 buffer positions hold the current entries, with state Proxy-Buffer-Valid. The remaining 16 entries are reserved for data lines which are currently being unhooked (*i.e.* in state Proxy-Buffer-Pending-Invalid). In effect, the buffer is limited to 16 entries, with the "overflow" region providing the opportunity for further combining (or

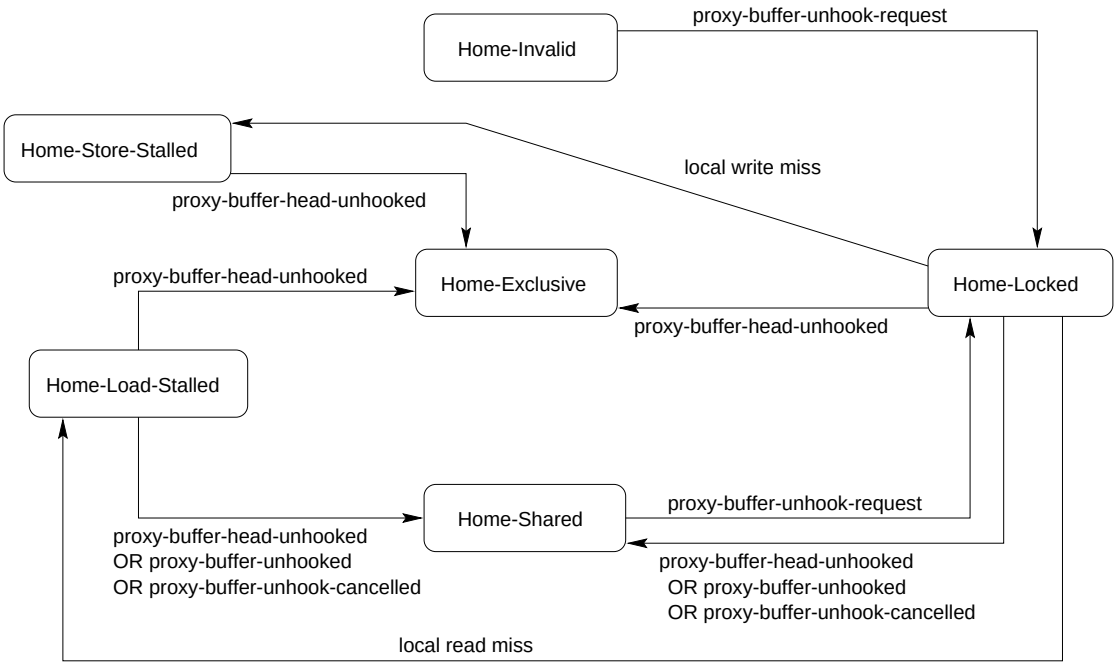


Figure 7.1: Extra node controller state transitions for home node actions

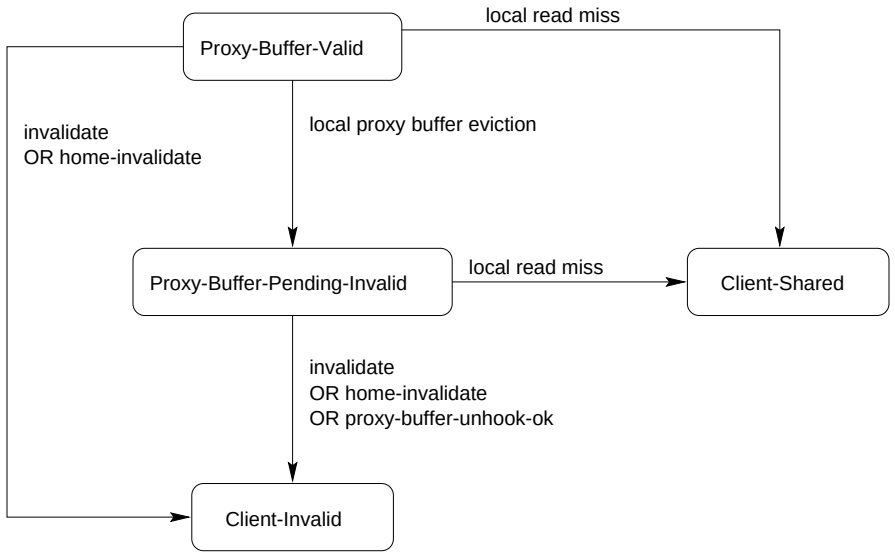


Figure 7.2: Extra node controller state transitions for client node actions

satisfying a local read miss) before the unhooking transaction completes. The sizes of the "current" and "unhooking" regions were set after experimental evaluation. They keep the overall buffer size low, while also keeping evictions from the proxy buffer reasonably low for most applications, and avoiding stalling evictions (due to the "unhooking" region being full) in most cases.

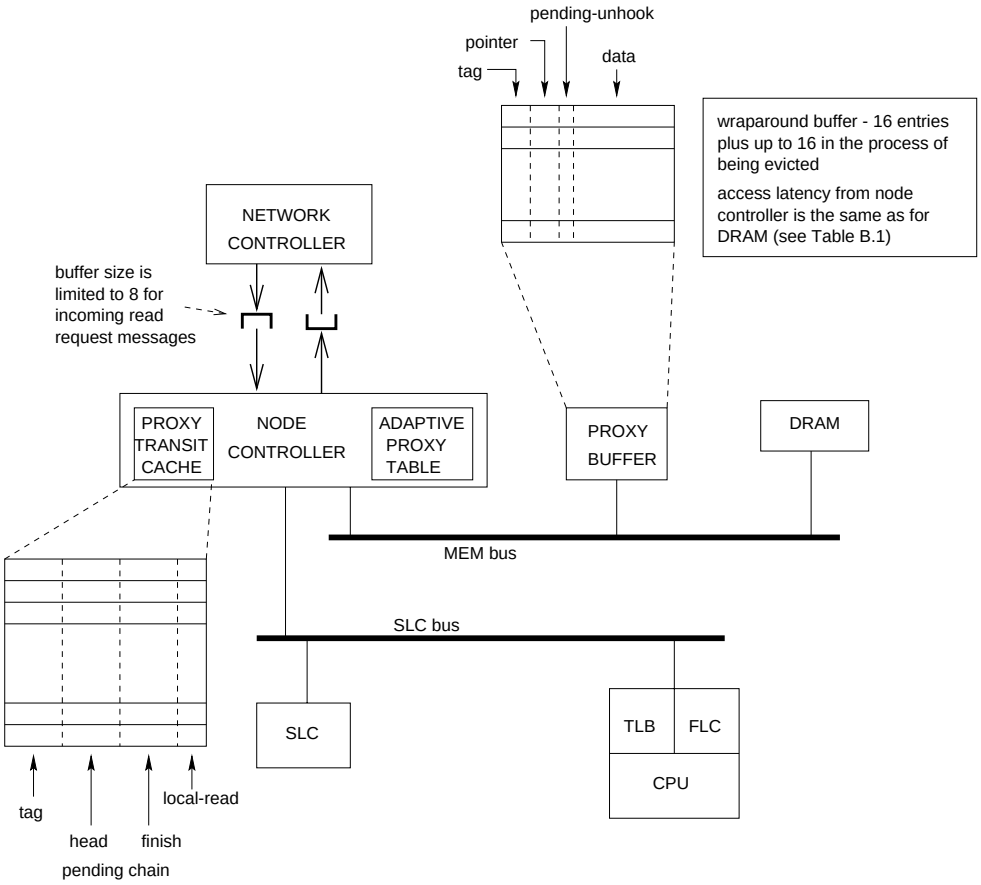


Figure 7.3: Memory model for a cc-NUMA node with a separate proxy buffer

The proxy buffer is simulated as a separate structure accessible via the MEM bus. This is a conservative approach because the latency of accessing its data from the node controller is similar to accessing DRAM data. A more aggressive approach would model the proxy buffer data as being directly accessible to the node controller either on-chip or as an off-chip cache (*i.e.* similar to the "directory caching" options investigated by Michael and Nanda [96]). The revised node architecture is illustrated in Figure 7.3.

7.4 Results

This section presents the results obtained from execution-driven simulations of the basic, reactive, and adaptive proxy strategies using the separate proxy buffer policy. It was shown in Section 3.5 that contention only becomes an important issue when more than a few tens of nodes are used, so the detailed results presented below are from simulations of a 64 node design. For details of the architecture simulated, refer to Section 3.4. For basic proxies, the shared data marked for proxying is shown in Table 7.1: the same marking was used in the preceding chapters.

application	problem size	shared data marked for basic proxying
Barnes	16K particles	all
CFD	64×64 grid	all
FFT	64K points	all
FMM	8K particles	`f_array` (part of `G_Memory`)
GE	512×512 matrix	entire matrix
Ocean-Contig	258×258 ocean	`q_multi` and `rhs_multi`
Ocean-Non-Contig	258×258 ocean	`fields`, `fields2`, `wrk`, and `frcng`
Water-Nsq	512 molecules	`VAR` and `PFORCES`

Table 7.1: Benchmark problem sizes, and data marked for basic proxies

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC} = 1$ to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	+0.2	+0.3	0.0	+0.1	0.0	-0.5	-0.3	-0.1
		reactive	+0.4	+3.2	-0.3	0.0	+0.3	+0.2	+0.3	0.0
		adaptive	0.0	+3.3	+0.4	+0.2	+0.2	+0.2	+0.4	+0.4
CFD	28.3	basic	+9.9	+12.1	+10.9	+13.9	+10.4	+9.0	+9.3	+15.5
		reactive	+9.8	+5.6	+8.5	+6.8	+8.4	+7.7	+7.5	+4.4
		adaptive	+9.4	+9.4	+9.0	+12.5	+10.7	+10.8	+10.5	+12.7
FFT	47.3	basic	+7.5	+8.0	+8.6	+7.9	+8.8	+7.1	+8.6	+6.9
		reactive	+11.6	+11.0	+10.9	+11.2	+10.9	+10.9	+10.9	+10.5
		adaptive	+11.9	+11.9	+11.6	+11.8	+11.4	+11.4	+11.0	+10.8
FMM	52.4	basic	+0.4	+0.4	+0.4	+0.4	+0.4	+0.4	+0.4	+0.4
		reactive	+0.4	+0.4	+0.4	+0.4	+0.4	+0.4	+0.3	+0.4
		adaptive	+0.4	+0.3	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4
GE	21.6	basic	+30.2	+30.4	+30.8	+30.4	+30.5	+30.6	+30.7	+30.5
		reactive	+28.4	+28.7	+28.9	+28.9	+28.8	+28.9	+28.9	+28.8
		adaptive	+30.7	+30.9	+31.8	+31.3	+31.8	+31.8	+31.5	+31.7
Ocean-Contig	49.7	basic	-2.1	+3.1	-0.4	-6.5	-1.3	-2.0	-4.8	-2.1
		reactive	-1.8	+0.4	-3.5	+4.4	-3.4	-1.5	-1.6	-2.3
		adaptive	-2.4	+1.5	-1.5	-6.8	-0.2	+1.9	+0.8	-0.7
Ocean-Non-Contig	48.2	basic	+7.5	+5.3	+0.2	+2.2	+3.9	+2.0	+0.9	+1.0
		reactive	+3.7	+7.8	+2.0	+4.4	+7.5	+6.6	+6.3	+3.5
		adaptive	+4.5	+6.5	+5.8	+2.3	-0.2	+3.0	+3.7	+6.8
Water-Nsq	55.3	basic	-0.4	-0.4	-0.4	-0.3	-0.3	-0.3	-0.5	-0.3
		reactive	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

Table 7.2: Benchmark relative speedups with a separate proxy buffer (64 nodes)

The results for the eight applications using a separate proxy buffer are summarised in Table 7.2. To simplify the comparison with the SLC and non-caching schemes, Table 7.3 repeats the results from Chapter 5, and Table 7.4 repeats the results from Chapter 6. The performance results are presented in terms of relative speedup, *i.e.* the ratio of the execution time for the fastest algorithm running on one processor to the execution time on $\mathcal{P}$ processors.

Basic proxies show similar performance characteristics to those seen with SLC caching proxies. Some applications fare slightly better using the proxy buffer, *e.g.* GE, while other applications such as FFT get slightly better performance using SLC caching. The exception is CFD which gets its best performance under basic proxies by using the non-caching approach.

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	+0.2	0.0	0.0	+0.2	-0.1	-0.3	0.0	-0.1
		reactive	+0.4	+3.3	+0.2	0.0	+0.3	0.0	+0.5	+0.4
		adaptive	+0.1	+3.2	+0.4	+0.4	+0.4	+0.2	-0.1	+0.2
CFD	28.3	basic	+10.4	+11.3	+8.1	+11.8	+9.8	+9.3	+9.2	+14.8
		reactive	+7.6	+7.4	+8.3	+7.6	+8.6	+7.6	+7.6	+6.1
		adaptive	+9.2	+13.1	+11.3	+11.6	+11.2	+10.4	+10.6	+12.1
FFT	47.3	basic	+9.4	+8.7	+8.7	+9.6	+9.5	+8.6	+10.0	+8.5
		reactive	+11.7	+11.2	+10.9	+11.0	+11.2	+11.8	+11.2	+10.7
		adaptive	+11.9	+11.6	+11.3	+11.4	+11.2	+11.5	+11.0	+11.0
FMM	52.4	basic	+0.4	+0.4	+0.5	+0.3	+0.4	+0.3	+0.3	+0.4
		reactive	+0.3	+0.4	+0.4	+0.4	+0.3	+0.4	+0.4	+0.4
		adaptive	+0.4	+0.4	+0.4	+0.4	+0.4	+0.5	+0.4	+0.4
GE	21.6	basic	+29.3	+29.3	+29.3	+29.4	+29.4	+29.5	+29.6	+29.6
		reactive	+28.4	+28.6	+28.9	+28.8	+28.8	+28.8	+28.7	+28.9
		adaptive	+30.5	+30.7	+31.4	+31.2	+31.7	+31.6	+31.4	+31.6
Ocean-Contig	49.7	basic	-2.6	-0.9	-1.1	-4.7	-2.1	+0.4	-5.4	+0.9
		reactive	-0.6	-4.4	+1.8	+3.3	-0.9	+2.5	+1.8	+2.6
		adaptive	-1.3	-2.8	-6.1	-3.5	-1.4	-3.6	-0.4	-3.6
Ocean-Non-Contig	48.2	basic	+5.3	0.0	+2.4	-0.7	+6.4	+1.8	+5.7	-1.2
		reactive	+5.8	+3.1	+2.0	+4.7	+4.0	-1.9	+2.5	+5.5
		adaptive	+7.8	+7.6	-6.3	+2.0	+4.1	+6.6	-8.3	-1.5
Water-Nsq	55.3	basic	-0.6	-0.6	-0.6	-0.6	-0.6	-0.5	-0.7	-0.5
		reactive	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

Table 7.3: Benchmark relative speedups for SLC caching proxies (64 nodes)

application	relative speedup no proxies	proxy type	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	basic	-0.3	+0.4	-0.1	+0.2	+0.2	+0.2	+0.1	-0.1
		reactive	+0.1	+3.4	0.0	+0.2	+0.1	+0.1	-0.2	+0.4
		adaptive	+0.4	+3.7	0.0	0.0	+0.5	+0.3	+0.1	+0.2
CFD	28.3	basic	+14.3	+14.9	+14.7	+15.7	+14.8	+15.0	+14.3	+15.3
		reactive	+10.5	+9.4	+8.9	+8.1	+9.5	+7.4	+8.4	+6.3
		adaptive	+12.9	+13.7	+13.6	+12.7	+12.9	+13.5	+12.7	+12.5
FFT	47.3	basic	+10.7	+10.0	+9.8	+10.4	+9.6	+9.6	+10.5	+9.5
		reactive	+11.2	+10.8	+10.7	+10.7	+10.7	+11.4	+10.7	+10.3
		adaptive	+11.7	+11.2	+11.3	+11.4	+11.3	+11.1	+11.2	+10.8
FMM	52.4	basic	+0.4	+0.4	+0.5	+0.5	+0.4	+0.4	+0.4	+0.4
		reactive	+0.3	+0.4	+0.4	+0.4	+0.3	+0.4	+0.4	+0.3
		adaptive	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4	+0.4	+0.4
GE	21.6	basic	+30.9	+30.8	+30.7	+30.7	+30.6	+30.5	+30.3	+30.1
		reactive	+27.5	+27.8	+28.1	+28.0	+27.9	+27.9	+27.9	+27.9
		adaptive	+30.3	+30.5	+31.4	+31.0	+31.3	+31.3	+31.0	+31.0
Ocean-Contig	49.7	basic	-2.0	+1.0	+1.0	-2.9	-1.0	-1.5	-1.6	-0.7
		reactive	-1.1	+0.2	-7.0	+3.0	-3.3	-1.5	+0.7	-0.4
		adaptive	+3.2	+0.5	-1.0	-2.3	0.0	-2.6	-0.1	-1.1
Ocean-Non-Contig	48.2	basic	+4.8	+7.1	+1.4	+4.1	+2.2	+2.1	+6.1	+2.6
		reactive	+4.1	+0.9	+4.2	-19.4	-6.9	+7.4	+6.2	+4.6
		adaptive	+0.5	-3.6	+4.4	-11.3	+3.7	+4.7	+7.4	+3.3
Water-Nsq	55.3	basic	-0.4	-0.4	-0.3	-0.4	-0.3	-0.3	-0.4	-0.3
		reactive	+0.2	+0.2	+0.2	+0.1	+0.1	+0.2	+0.2	+0.1
		adaptive	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2

Table 7.4: Benchmark relative speedups for non-caching proxies (64 nodes)

For reactive proxies, using a separate proxy buffer gives a balance point at $\mathcal{NPC}$=2, *i.e.* the same balance point obtained with non-caching proxies. The performance effects are similar to those seen with SLC caching, with the exception of the Ocean applications.

Overall the adaptive proxy scheme gets the best performance by using a separate proxy buffer, with balance points at $\mathcal{NPC}$=2,6&7. This increase in the number of balance points, compared to there being no balance point with SLC caching, and only one balance point at $\mathcal{NPC}$=1 for non-caching proxies, arises from the improved performance of the Ocean-Contig application.

The following sub-sections contain a more detailed analysis of the results of using a separate proxy buffer for each of the eight applications. The results for each application are presented in the form of two graphs and two histograms for each of the basic, reactive, and adaptive proxy schemes. The graphs show relative changes and message ratios, while the histograms show the execution time profiles and message category profiles. In addition, node-by-node graphs showing mean queueing cycles are included where appropriate.

As in previous chapters, the relative changes graphs show four different metrics:

messages: the ratio of the total number of messages to the total without proxies,

execution time: the ratio of the execution time (excluding startup) to the execution time (also excluding startup) without proxies,

queueing delay: the ratio of the total time that messages spend waiting for service to the total without proxies, and

remote read delay: the ratio of the mean delay between issuing a `read-request` and receiving the data, to the same mean delay when proxies are not used.

The message ratios are:

proxy hit rate: the ratio of the number of proxy read requests which are serviced directly by the proxy node, to the total number of proxy read requests (in contrast, a proxy miss would require the proxy to request the data from the home node),

buffer bounce ratio: the ratio of the total number of buffer bounce messages to read requests. This gives a measure of how much bouncing there is for an application. This ratio can go above one, since only the initial read request is counted in that total, *i.e.* the retries are excluded, and

proxy read ratio: the ratio of the proxy read messages to read requests - this gives a measure of how much proxying is used in an application.

The execution time profile presents the overall execution time split into CPU active time and the time spent waiting because of delays. The delays are further split into load, store, barrier and lock delays. The times are normalised with respect to the execution time without proxies.

The message category profile shows how the total number of messages breaks down into five categories: read, write, unhook, proxy, and proxy buffer unhook messages. The allocation of message types to message categories is given in Appendix C.2, but it should be noted that read messages include all the `read-request`, `buffer-bounced-read-request`, and `take-shared` messages, and write messages include all the `write-request`, `invalidate`, and `take-exclusive` messages.

7.4.1 Barnes

For basic proxies this application has worse performance when $\mathcal{NPC} \geq 6$ than not using proxies, whereas the reactive and adaptive schemes generally improve the performance compared to not using proxies (see Figure 7.5). The most marked effect with basic proxies is the increase in queueing delay when $\mathcal{NPC} \geq 1$, as shown in Figure 7.4. This increase is caused by the increase in the total number of messages because of the high level of proxy buffer unhook messages (Figure 7.7). The queueing delay is worst when $\mathcal{NPC}=1$ because a bottleneck occurs for a few proxy nodes. The queueing delay across the nodes evens out as the number of proxies is increases, as shown in Figure 7.8.

The proxy buffer unhook messages are generated when node controllers need to evict the oldest entry from the proxy buffer to make room for the newest line of data to be proxied. All the shared data was marked for basic proxying in Barnes, and as a result there is a high turnover of entries in the proxy buffer. This lowers the proxy hit rate in comparison to the SLC caching scheme, because proxy buffer entries are evicted earlier than when they were held in the SLC. However the hit rate is still better than that observed using the non-caching approach. The turnover of entries in the proxy buffer increases with $\mathcal{NPC}$ because for a given data line there will be more nodes acting as proxy and retaining a copy of the data line in their proxy buffers. The resulting increase in the number of evictions is reflected by a rise in the number of proxy buffer unhook messages, as shown in Figure 7.7. The processing of the extra unhook messages introduces delays for other messages, and so the performance degrades for basic proxies when $\mathcal{NPC} \geq 6$.

The reactive and adaptive schemes only invoke proxying when home node congestion causes the buffer-bouncing of read requests. There is very little buffer-bouncing in Barnes, as shown by the buffer bounce ratio in Figure 7.6. As a result there are hardly any `proxy-read-request`

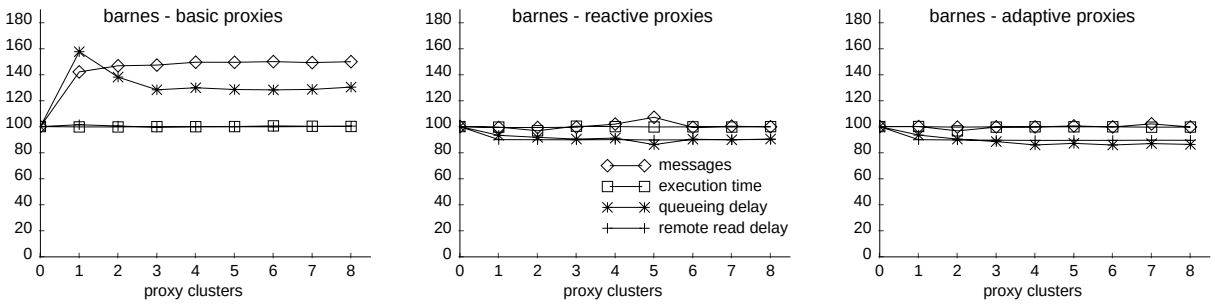


Figure 7.4: Barnes: changes (relative to no proxies case)

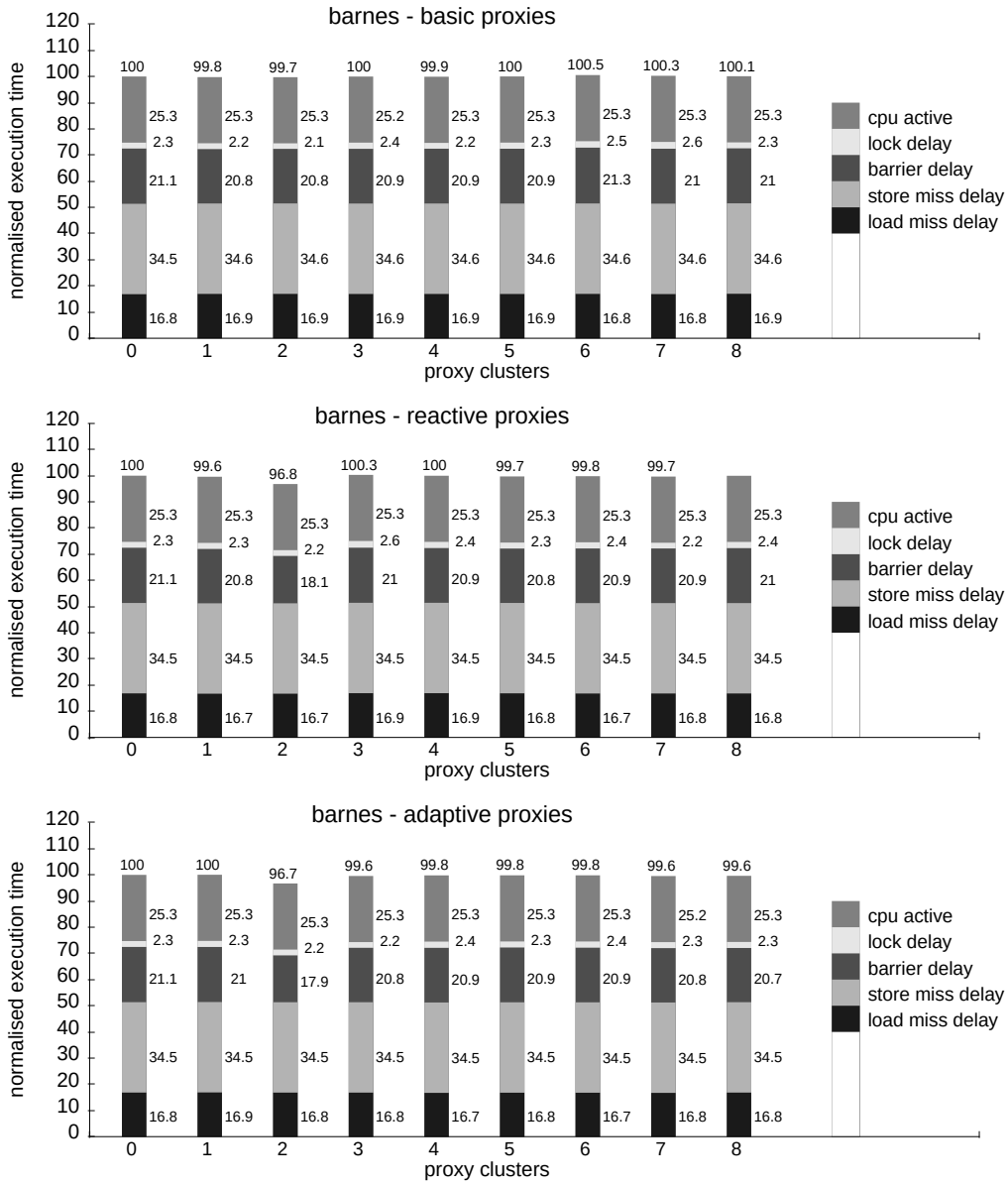


Figure 7.5: Barnes: execution time profiles

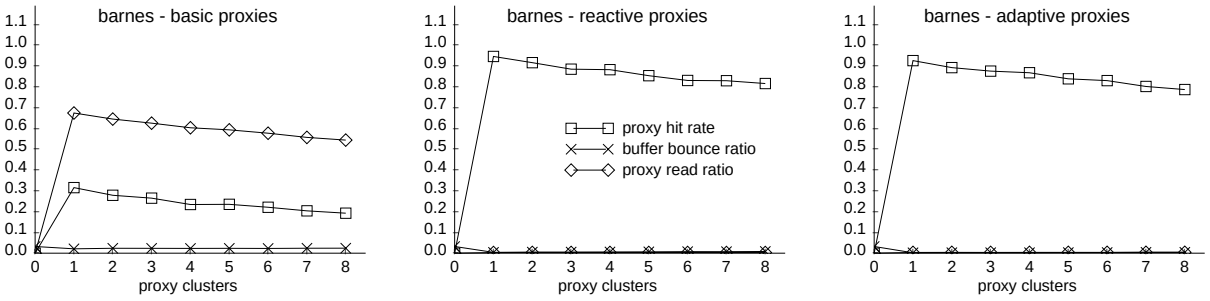


Figure 7.6: Barnes: message ratios

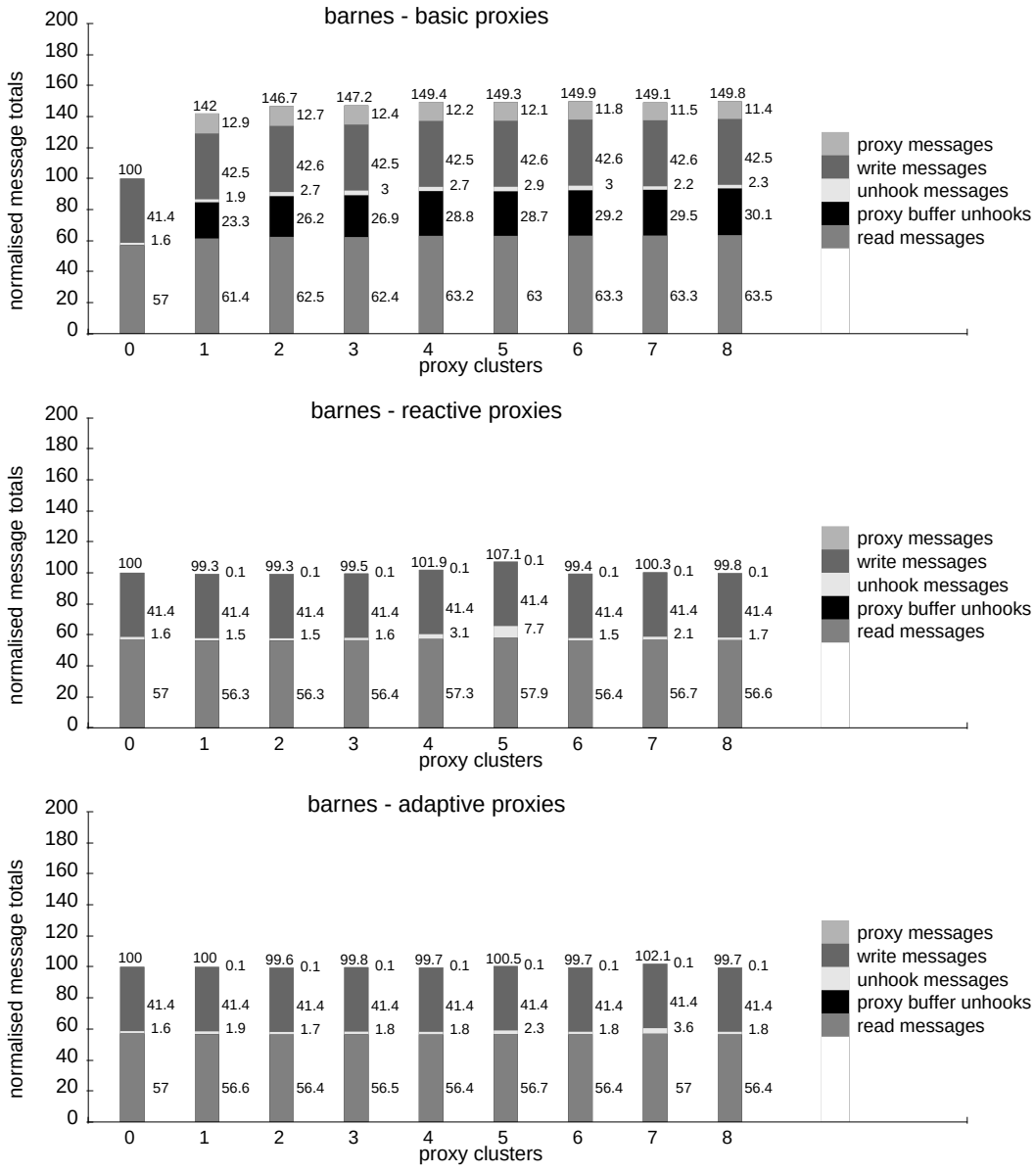


Figure 7.7: Barnes: message category profiles

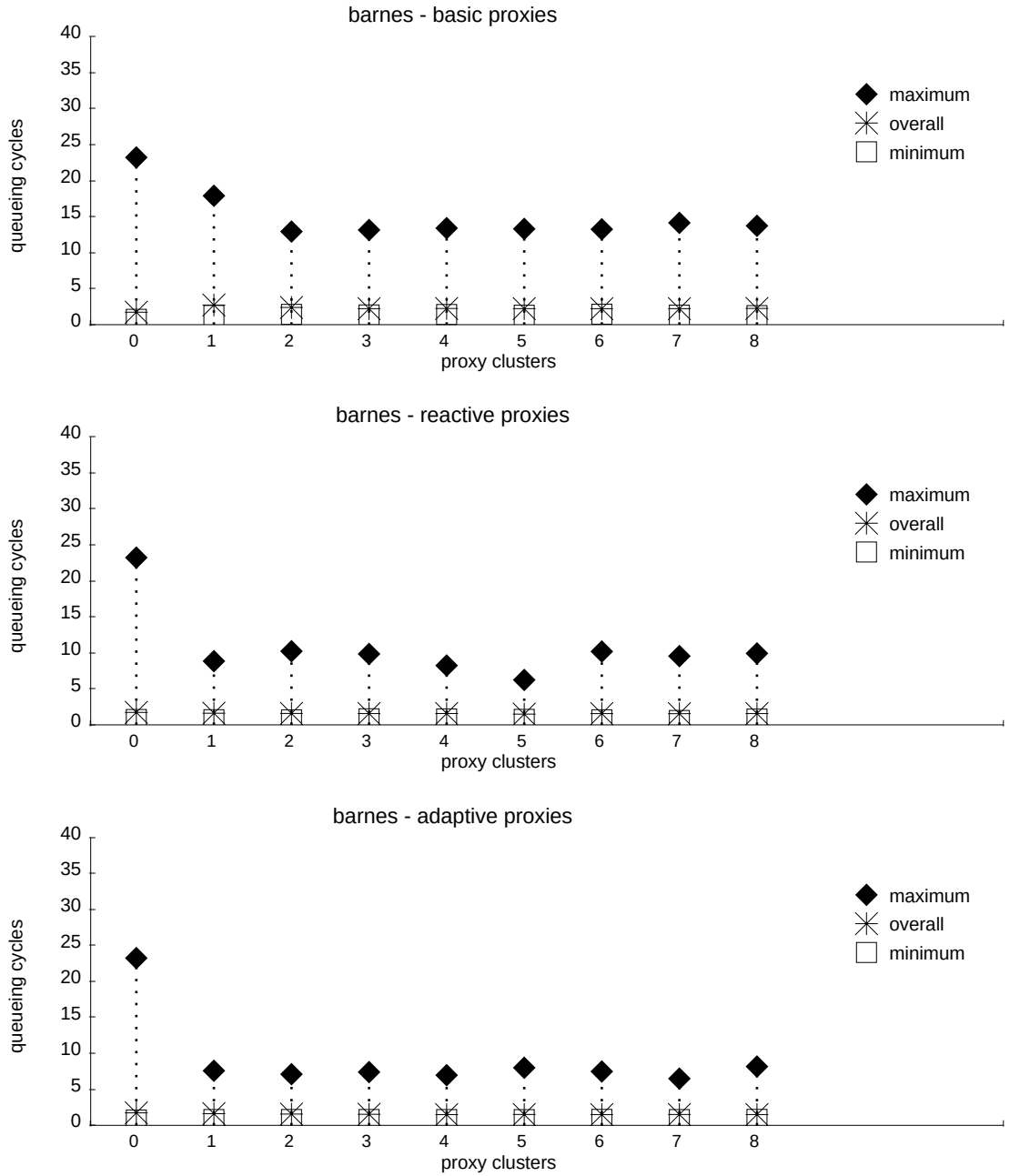


Figure 7.8: Barnes mean queueing cycles

messages, and the much lower usage of the proxy buffers means that no entries have to be evicted. Both the reactive and adaptive schemes achieve a high rate of combining for `proxy-read-request` messages, as shown by the proxy hit rate in Figure 7.6.

Reactive proxies usually improve the performance of Barnes. However when $\mathcal{NPC}$=3 the performance is −0.3% worse than not using proxies. This drop in performance stems from higher lock and barrier delays. For $\mathcal{NPC}$=3 on this application, the redistribution of processing via proxies has an adverse effect on the "normal" processing which needs to be done by node controllers. As a result, the proxy messages delay the handling of other messages, and the overall performance suffers.

Adaptive proxies improve the performance of Barnes for $\mathcal{NPC} \geq 2$. For $\mathcal{NPC}=1$ the overall performance is the same as when proxies are not used. As shown in Figure 7.5, the overall load miss delay increases very slightly compared to when $\mathcal{NPC}=0$. This happens because there is a slight increase in the number of unhooking messages attributable to an increase in `client-unhook-forward` messages, *i.e.* normal unhook requests have to traverse the extra proxy nodes in a sharing chain. However the marginal increase of 0.1% in the load miss delay is balanced by a matching improvement in the barrier delay resulting from the redistribution of node controller processing.

On balance, the best performance for this application with a separate proxy buffer is obtained by using adaptive proxies. The adaptive strategy of using proxying for the proxy period once node congestion has been detected, in conjunction with avoiding SLC pollution, improves the performance for $\mathcal{NPC} \geq 2$, and does not degrade performance when $\mathcal{NPC}=1$.

7.4.2 CFD

Using a separate proxy buffer improves the performance with all three proxy strategies. As shown in Figure 7.10, the speedups are in the ranges 9.0% to 15.5% for basic proxies, 4.4% to 9.8% for reactive proxies, and 9.0% to 12.7% for adaptive proxies. The results are similar to those observed for the SLC caching scheme examined in Chapter 5, but are generally not as good as were seen for the non-caching scheme used in Chapter 6. The use of a separate proxy buffer avoids unnecessary conflict for the SLC bus between the local CPU and the node controller. However the number of write messages increases with proxies (see Figure 7.12) because invalidations are needed to remove proxy copies prior to a write. In addition, evictions from proxy buffers generate proxy buffer unhook requests. These increases in the number of messages, for the write and proxy buffer unhook categories, put the proxy buffer approach at a disadvantage to the non-caching scheme. This is because the increase in messages generally leads to the queueing delay being longer than with the non-caching scheme.

Although the performance improvements are not as good as those seen with non-caching proxies, the results indicate that the proxy buffer scheme is still very successful at reducing home node contention for this application.

7.4.3 FFT

The performance of the FFT application is improved by using a separate proxy buffer for all three proxy strategies. For basic proxies, the performance improvement is in the range 6.9% to 8.8%. However, this is not as good as was observed for basic proxies using either of the SLC

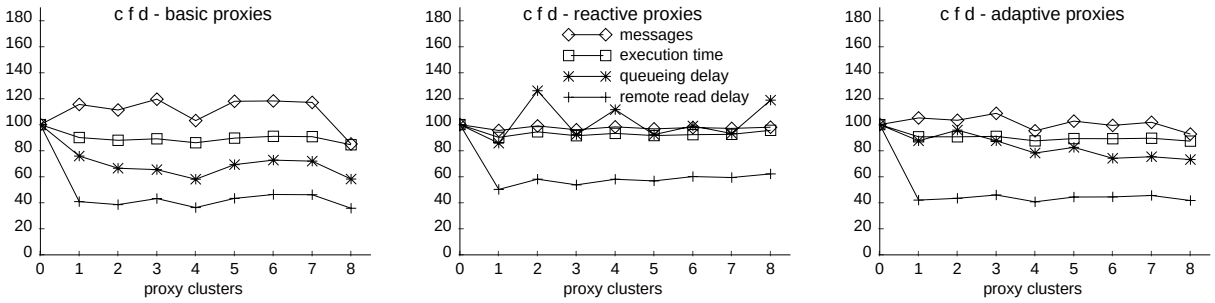


Figure 7.9: CFD: changes (relative to no proxies case)

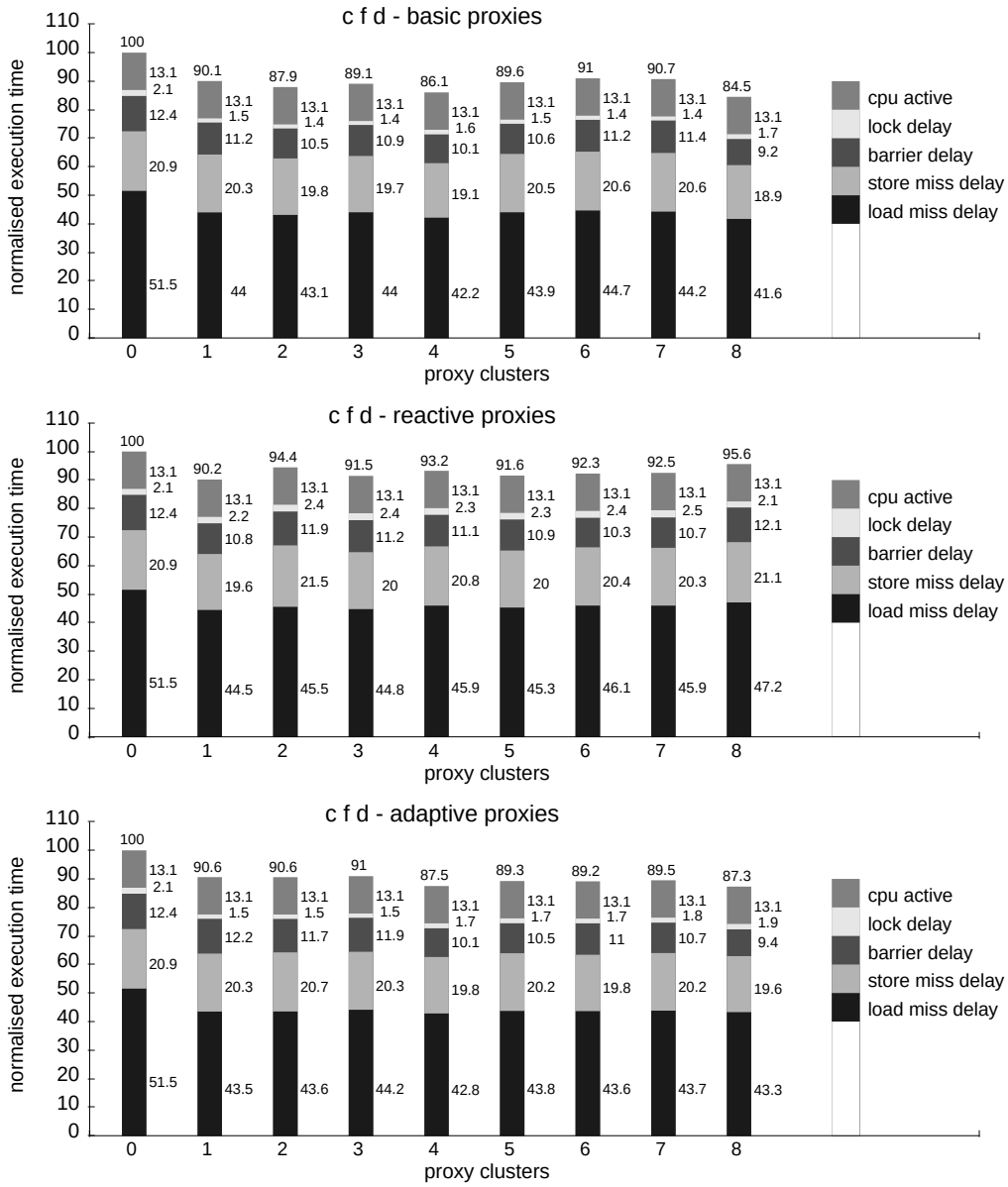


Figure 7.10: CFD: execution time profiles

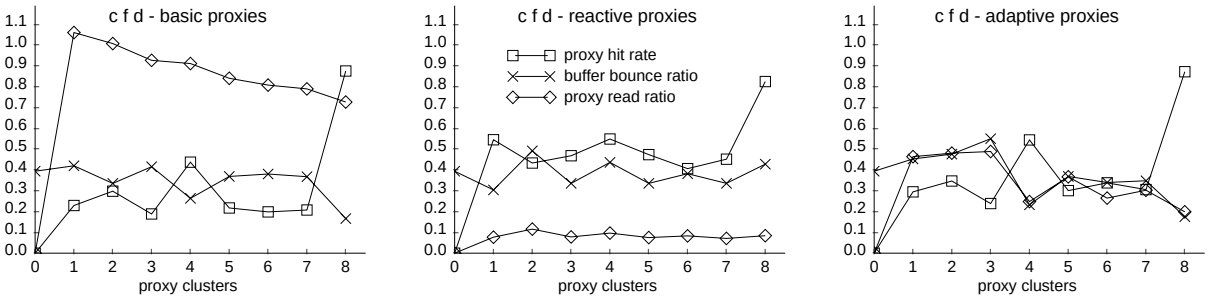


Figure 7.11: CFD: message ratios

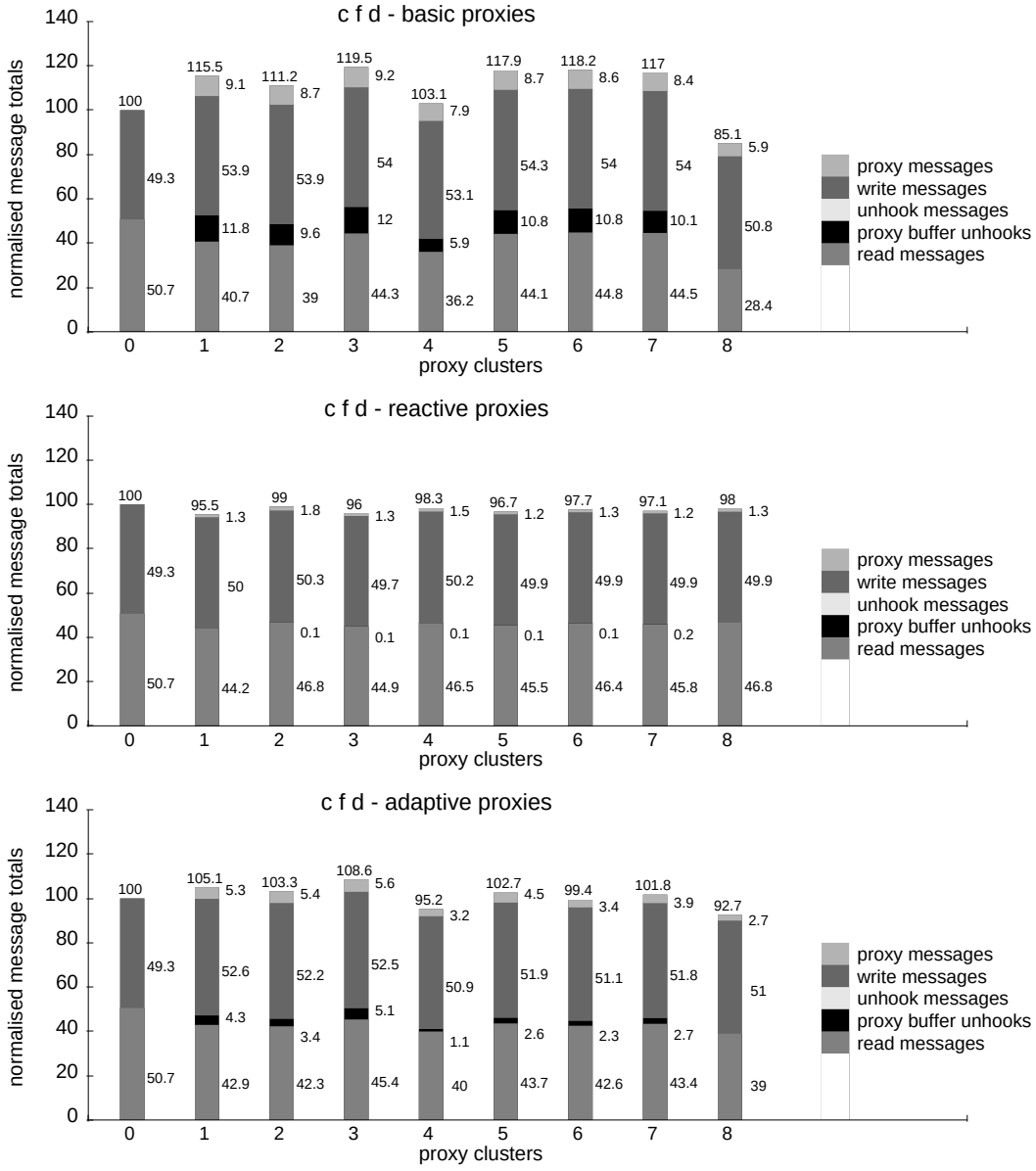


Figure 7.12: CFD: message category profiles

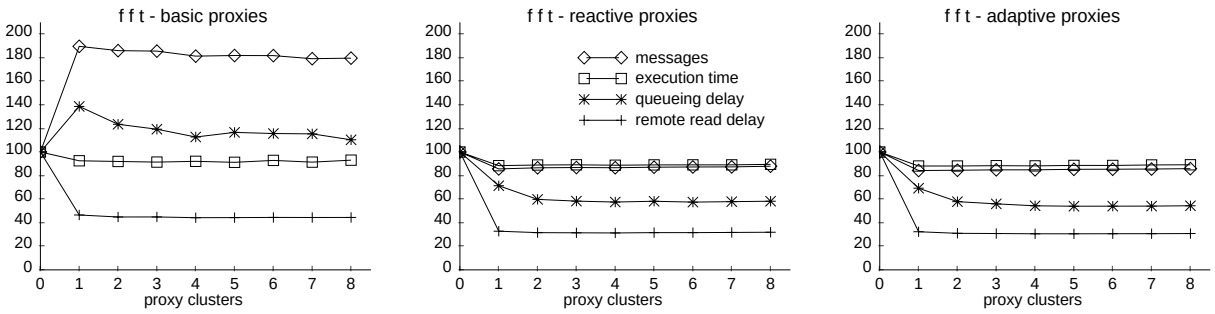


Figure 7.13: FFT: changes (relative to no proxies case)

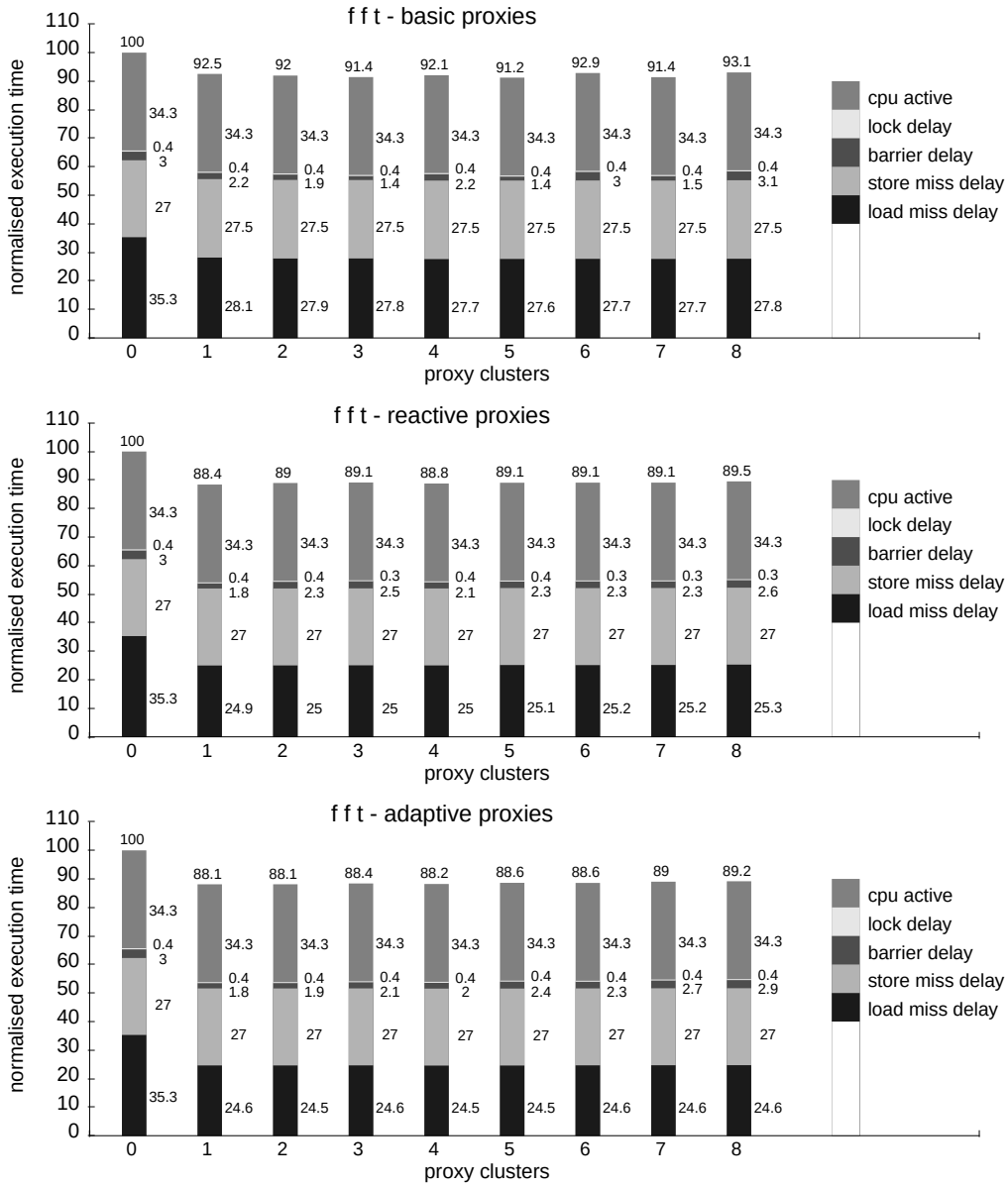


Figure 7.14: FFT: execution time profiles

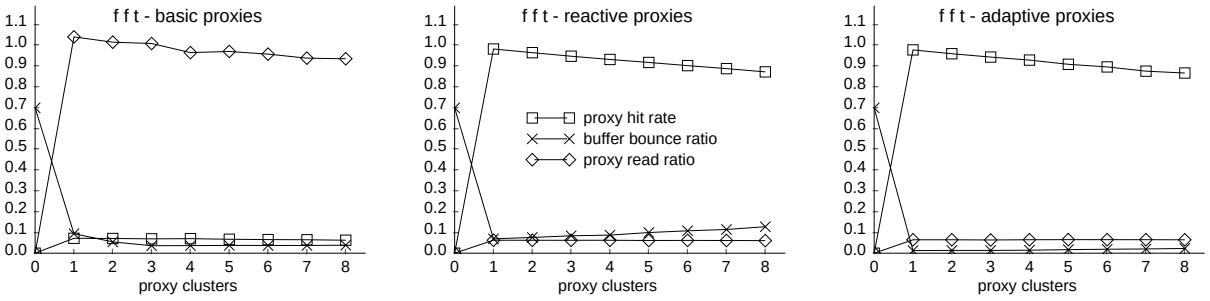


Figure 7.15: FFT: message ratios

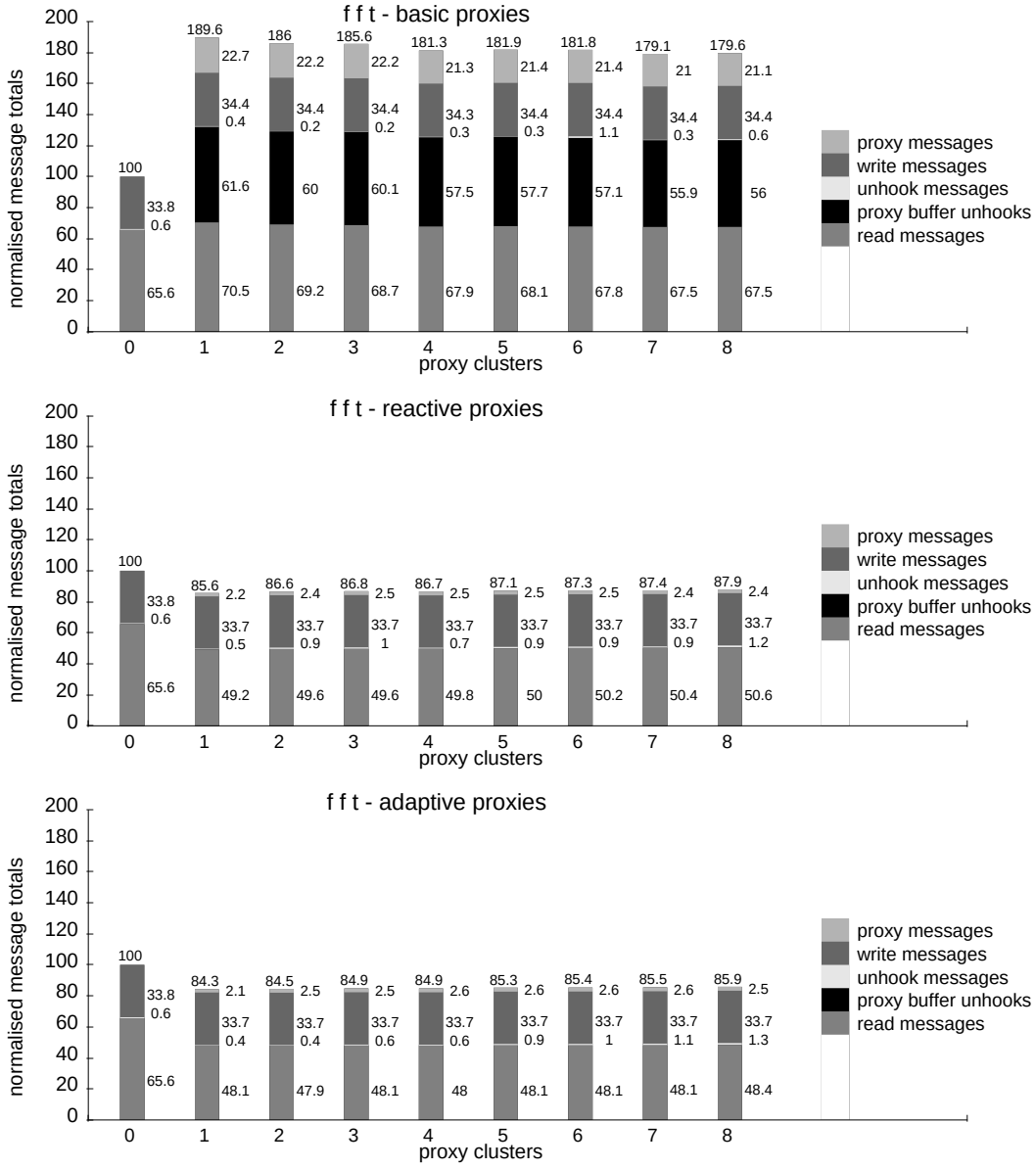


Figure 7.16: FFT: message category profiles

caching or non-caching approaches. Looking at Figure 7.13, the queueing delay increases for basic proxies which is in contrast to the earlier chapters where the queueing delay decreased with basic proxies. The increase stems from the high level of proxy buffer unhooks for this application (see Figure 7.16). The proxy buffers are greatly in demand, and as a result there is a high turnover of entries in the small buffers, as old entries are evicted to make room for the newest line of proxied data.

Using a separate proxy buffer for reactive and adaptive proxies gives performance improvements which are very similar to those obtained with the SLC caching and non-caching approaches. There are no evictions from the proxy buffers, reflecting the much lower use of proxies when `proxy-read-request` messages are triggered by home node congestion rather than marking all the shared data for basic proxying. The best performance is obtained using adaptive proxies, with speedups in the range 10.8% to 11.9%.

7.4.4 FMM

The performance of FMM is improved for all three types of proxying, with reductions in the overall queueing delay and the remote read delay (see Figure 7.17). The speedup is very small, being in the range 0.3% to 0.5%, but this was to be expected given that there is only one small widely-shared data structure in FMM. The results are very similar to those obtained using the SLC caching approach to holding proxied data. Both the use of a separate proxy buffer and the SLC caching strategy obtain a better proxy hit rate than the non-caching approach (see Figure 7.19, Figure 5.18, and Figure 6.20 respectively). This is because retaining a copy of a proxied data line allows subsequent `proxy-read-request` messages for this data line to be satisfied at the proxy node.

7.4.5 GE

All three forms of proxying improve the execution time for GE when a separate proxy buffer is used. The adaptive proxy scheme achieves the best performance, with speedups in the range 30.7% to 31.8%. The speedup with adaptive proxies is slightly better than that obtained when proxy data was cached in the SLC. In addition, the adaptive proxy performance is marginally better than that seen in Chapter 6 with the non-caching approach. This is because the retention of proxied data in the proxy buffer gives a slightly higher proxy hit rate (see Figure 7.23) and results in a lower load miss delay (see Figure 7.22).

The reactive proxy strategy gets the least improvement in performance, although the speedup is still substantial, being in the range 28.4% to 28.9%. This speedup is comparable to that

achieved with SLC caching of proxy data, and is better than that achieved with non-caching proxies. As with adaptive proxies, this shows the additional performance benefits for this application that come from maximising the proxy hit rate by retaining a copy of proxied data.

The basic proxy scheme achieves performance improvements in the range 30.2% to 30.8%. This is not quite as good as obtained using SLC caching, the difference being caused by the eviction of proxy data from the proxy buffer. The turnover of proxy buffer entries is at a much lower rate than seen earlier for the Barnes and FFT applications, and consequently has a less detrimental effect on the proxy hit rate.

7.4.6 Ocean-Contig

Using a separate proxy buffer with the Ocean-Contig application results in performance that is usually worse then not using proxies. However there are cases where reduced barrier delays give an overall performance speedup. As shown in Figure 7.26, this reduction in barrier delay occurs when $\mathcal{NPC}$=2 for all types of proxying, and also when $\mathcal{NPC}$=4 for reactive proxies, and when $\mathcal{NPC}$=6&7 for adaptive proxies.

For basic proxies, the high level of proxy buffer unhooks (see Figure 7.28) indicates that there is a high turnover of entries in the proxy buffers. This contributes to the high level of messages and the increase in overall and minimum mean queueing delays (see Figure 7.29). However, when $\mathcal{NPC}$=2 the lower load miss and barrier delays combine to improve the overall execution time.

There are only two values of $\mathcal{NPC}$ where reactive proxies improve the performance, *i.e.* at $\mathcal{NPC}$=2&4. This is in contrast to the SLC caching scheme investigated in Section 5.7.6, where the performance improved for $\mathcal{NPC}$=3,4,6,7&8. The deciding factor in all the performance improvements is a drop in barrier delay compared to not using proxies, as shown in Figure 7.26. The changes in barrier delay are not the result of uneven queueing distribution, because Figure 7.29 shows that this remains quite steady for $\mathcal{NPC} \geq 1$. The barrier delay changes stem from the nature of the Ocean-Contig application, which was written to maximise data locality. This results in the accesses to remote data being minimised, but these accesses tend to be "protected" by a barrier, *i.e.* updates to boundary data are followed by a barrier before the boundary data is read by the neighbouring processor. Any extra delay to an update transaction which precedes a barrier will delay *all* the processors which have already reached the barrier. Equally, any reduction in the update delay can reduce the barrier delay suffered by the other processors. The ordering of incoming messages is affected by the introduction

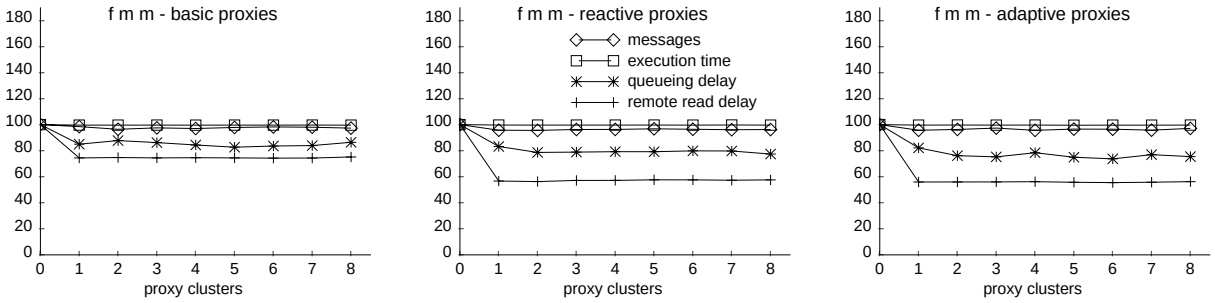


Figure 7.17: FMM: changes (relative to no proxies case)

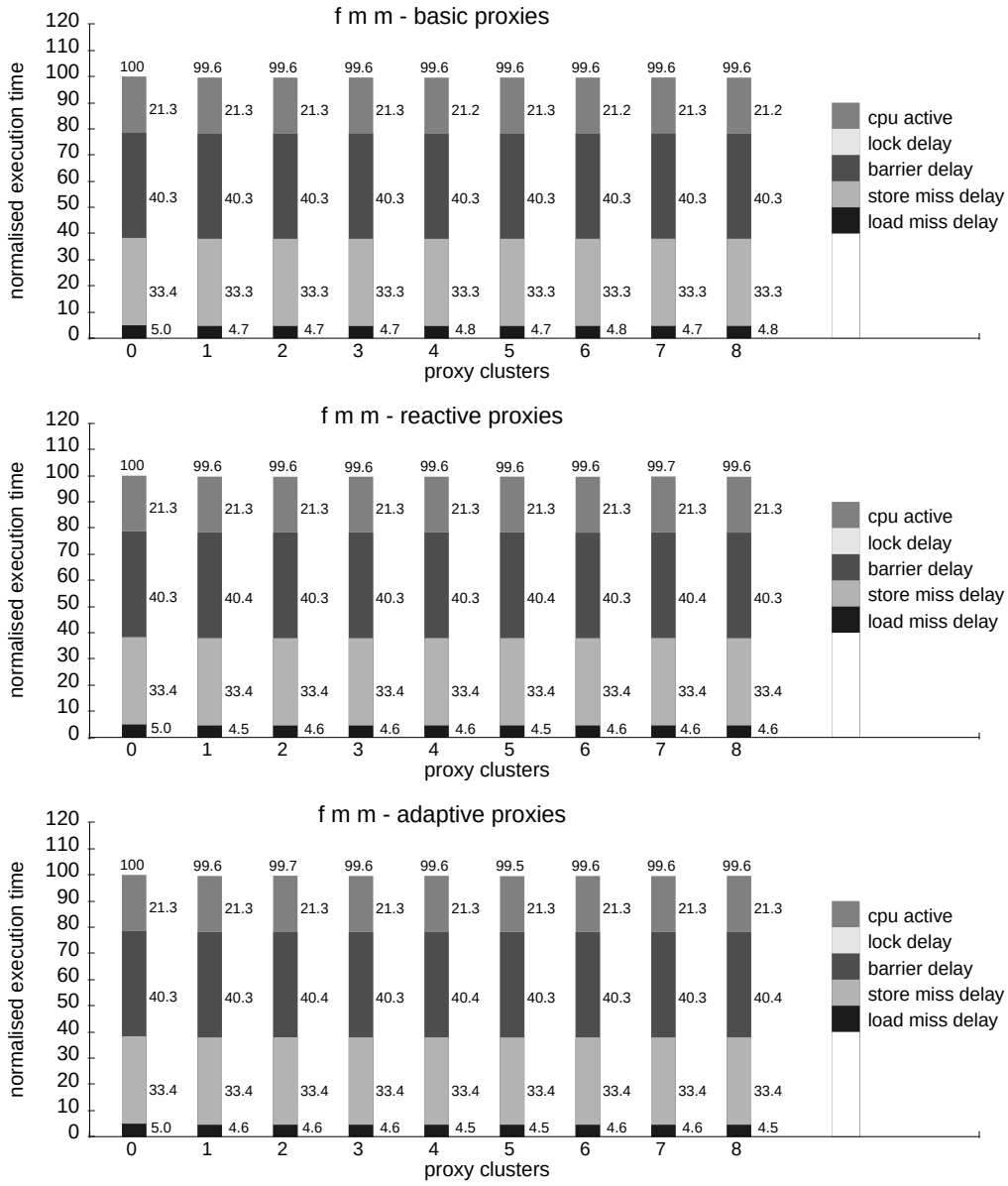


Figure 7.18: FMM: execution time profiles

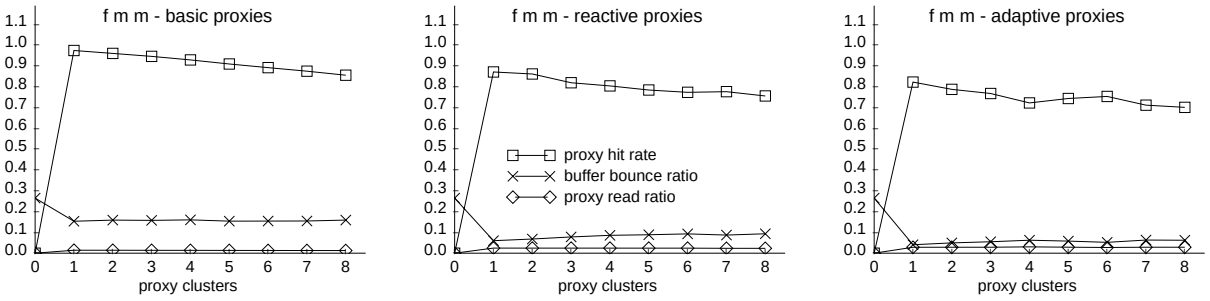


Figure 7.19: FMM: message ratios

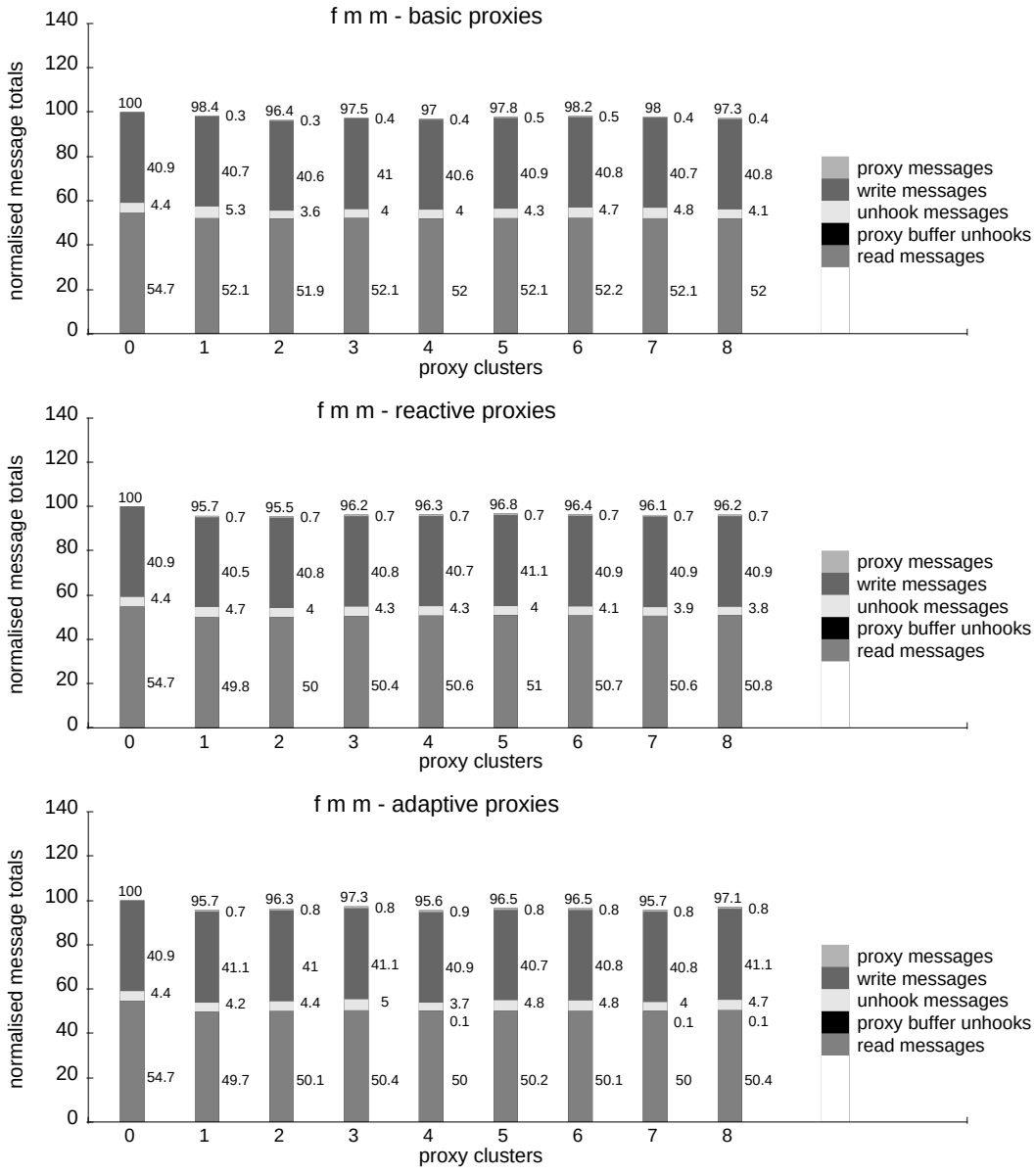


Figure 7.20: FMM: message category profiles

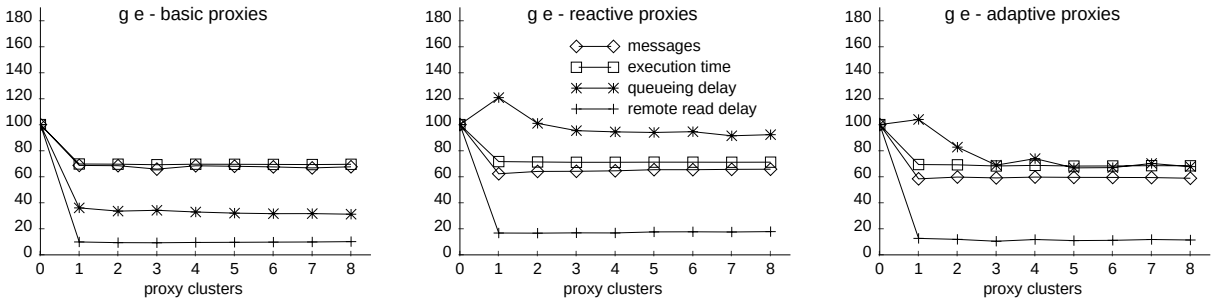


Figure 7.21: GE: changes (relative to no proxies case)

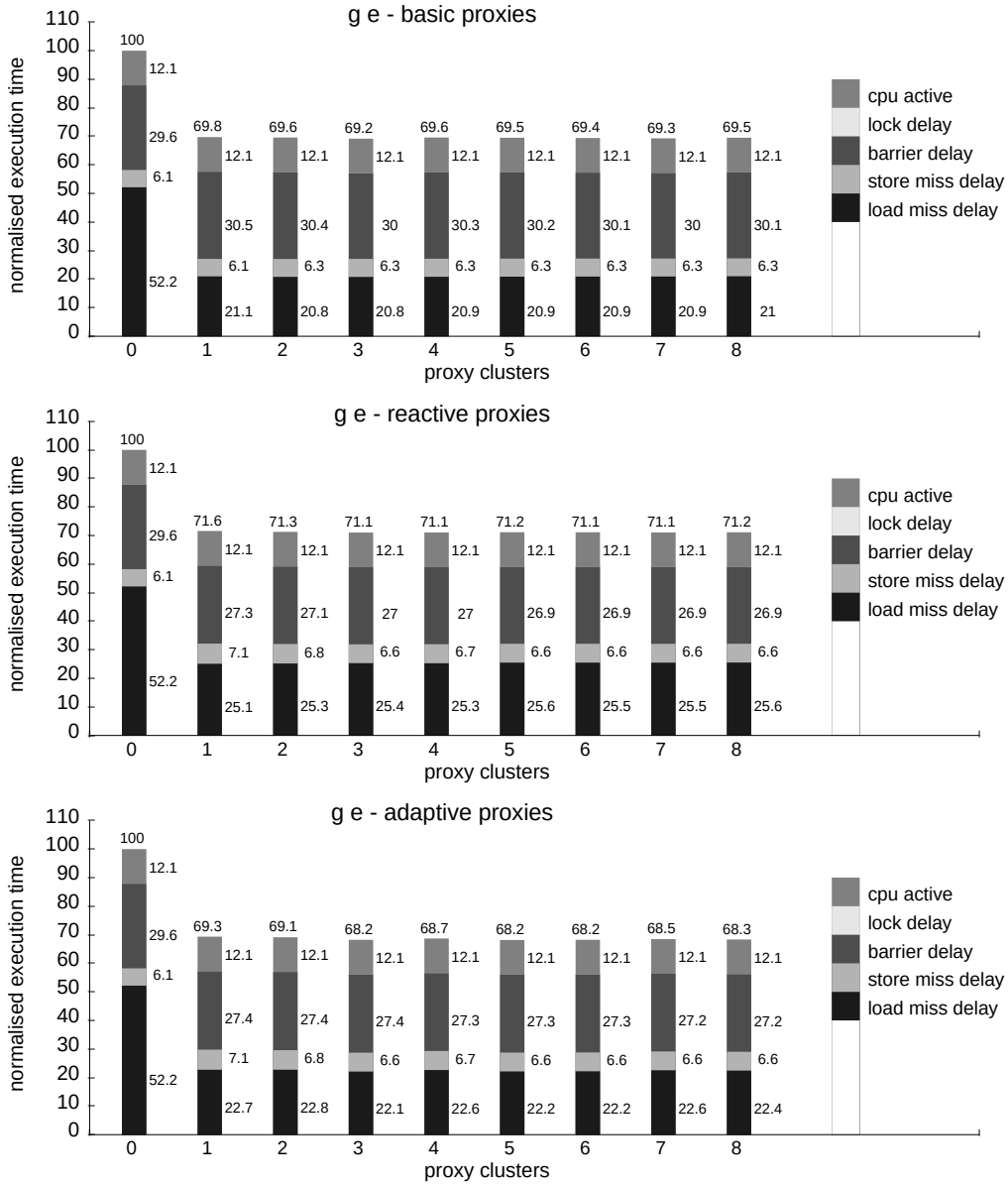


Figure 7.22: GE: execution time profiles

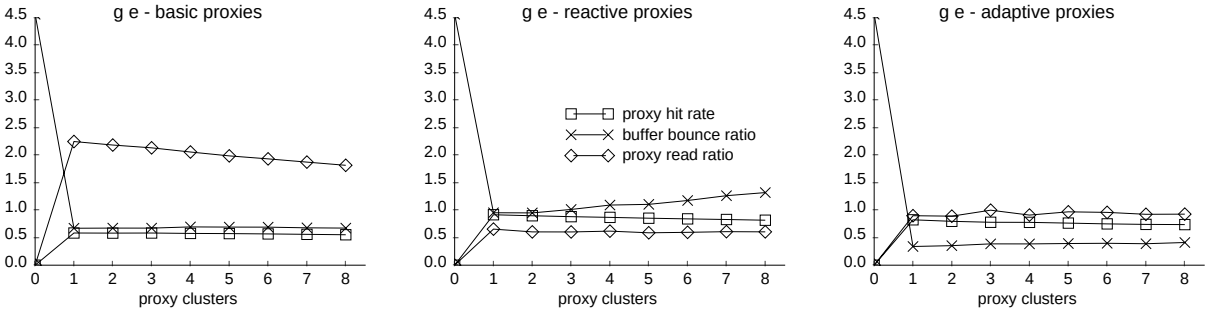


Figure 7.23: GE: message ratios

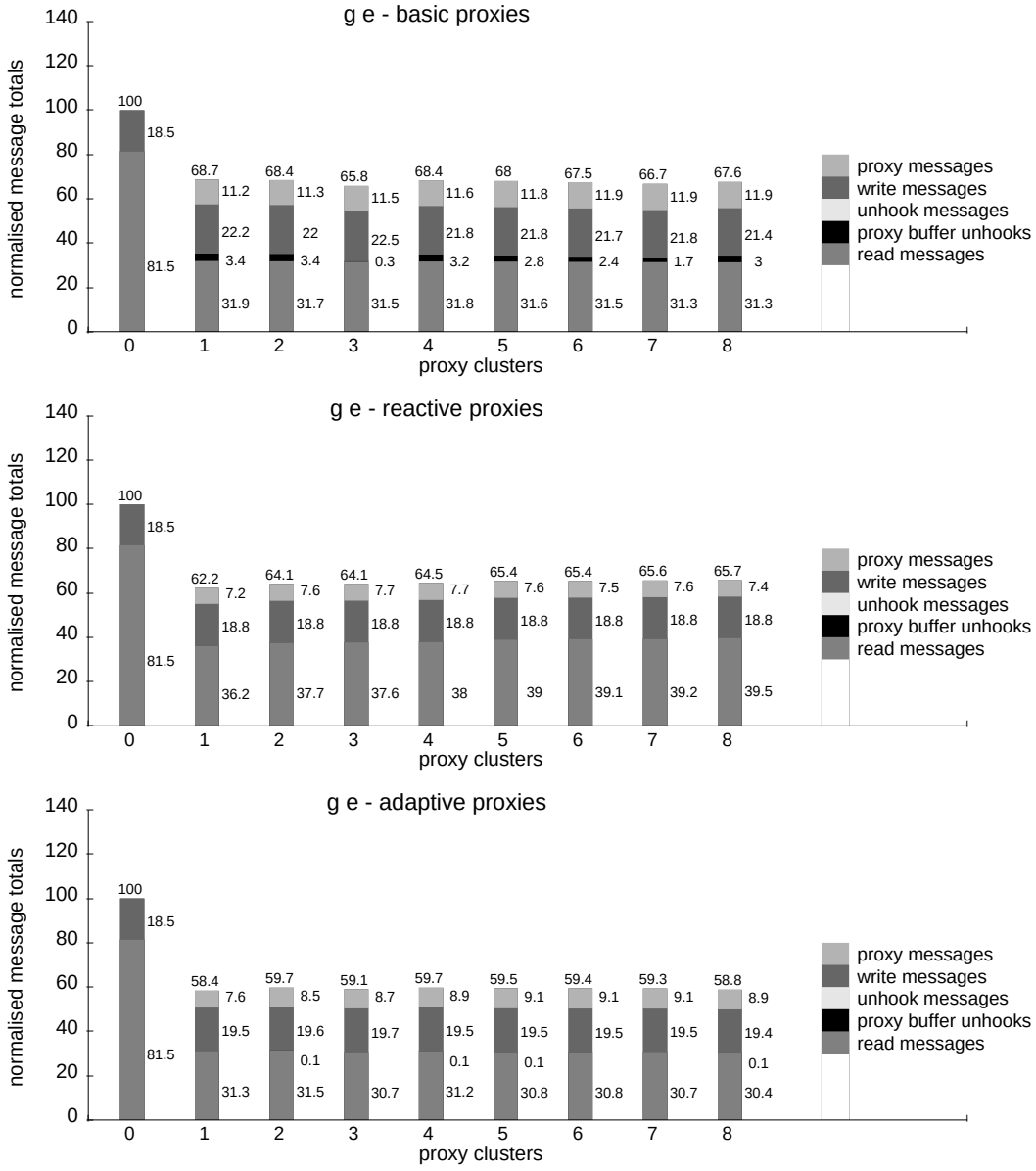


Figure 7.24: GE: message category profiles

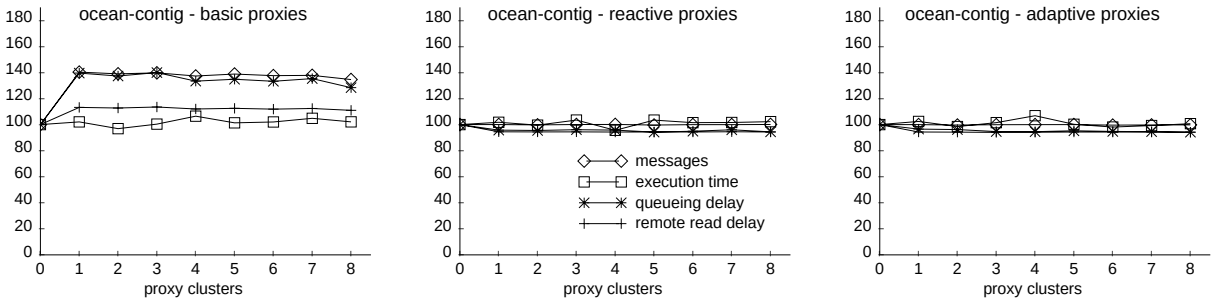


Figure 7.25: Ocean-Contig: changes (relative to no proxies case)

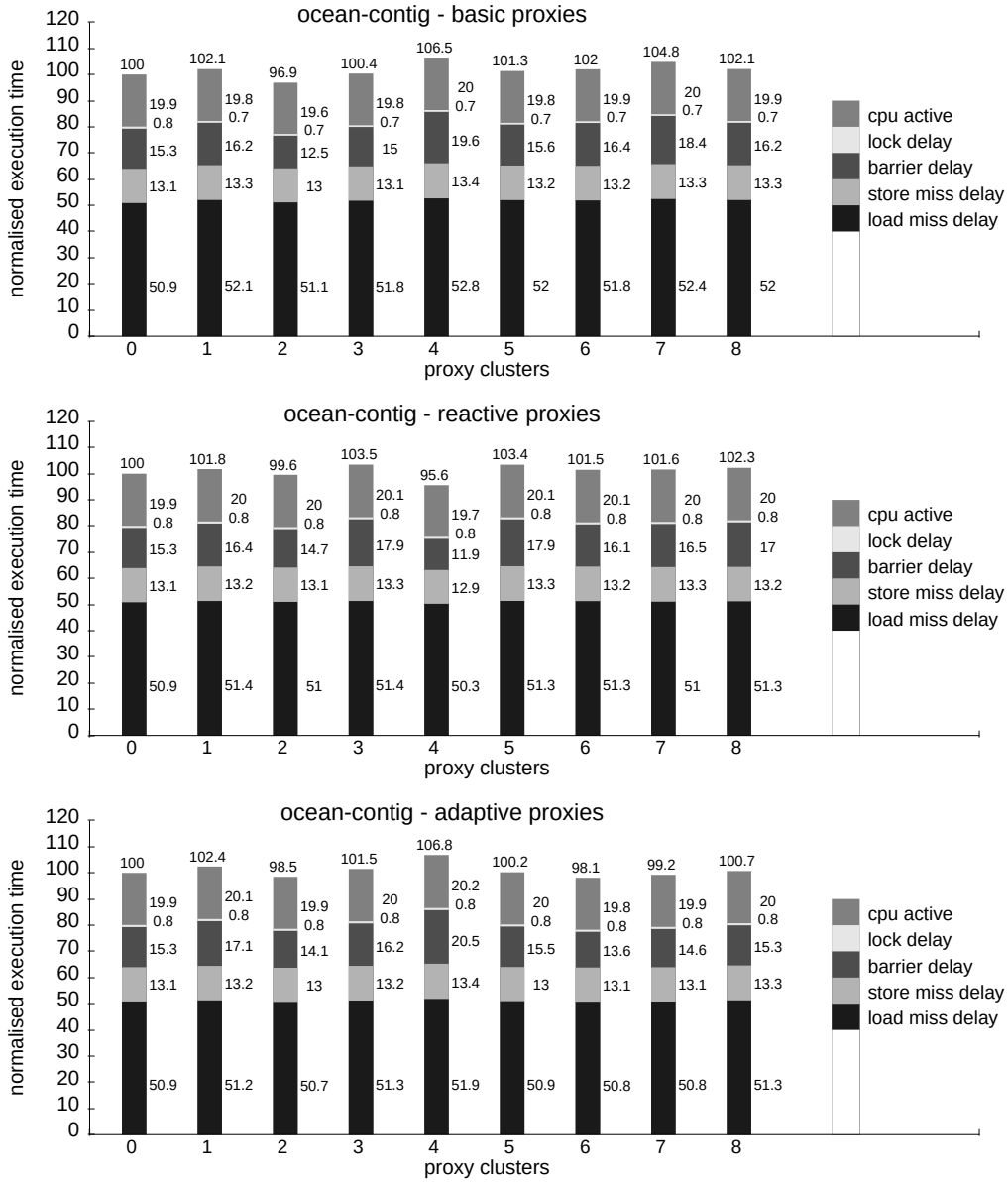


Figure 7.26: Ocean-Contig: execution time profiles

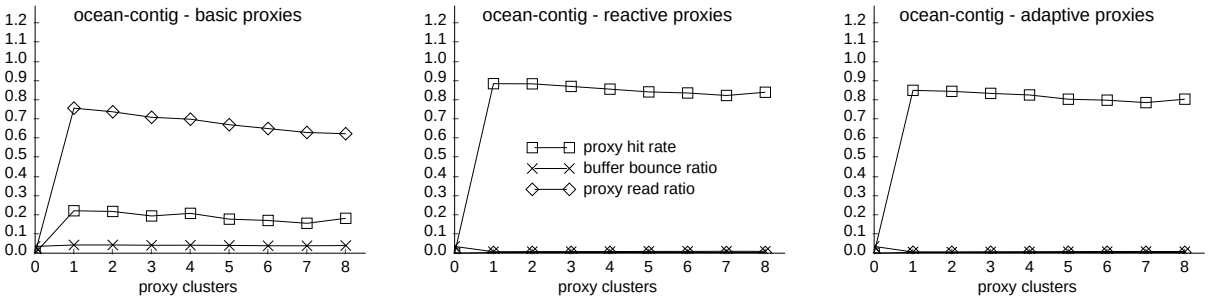


Figure 7.27: Ocean-Contig: message ratios

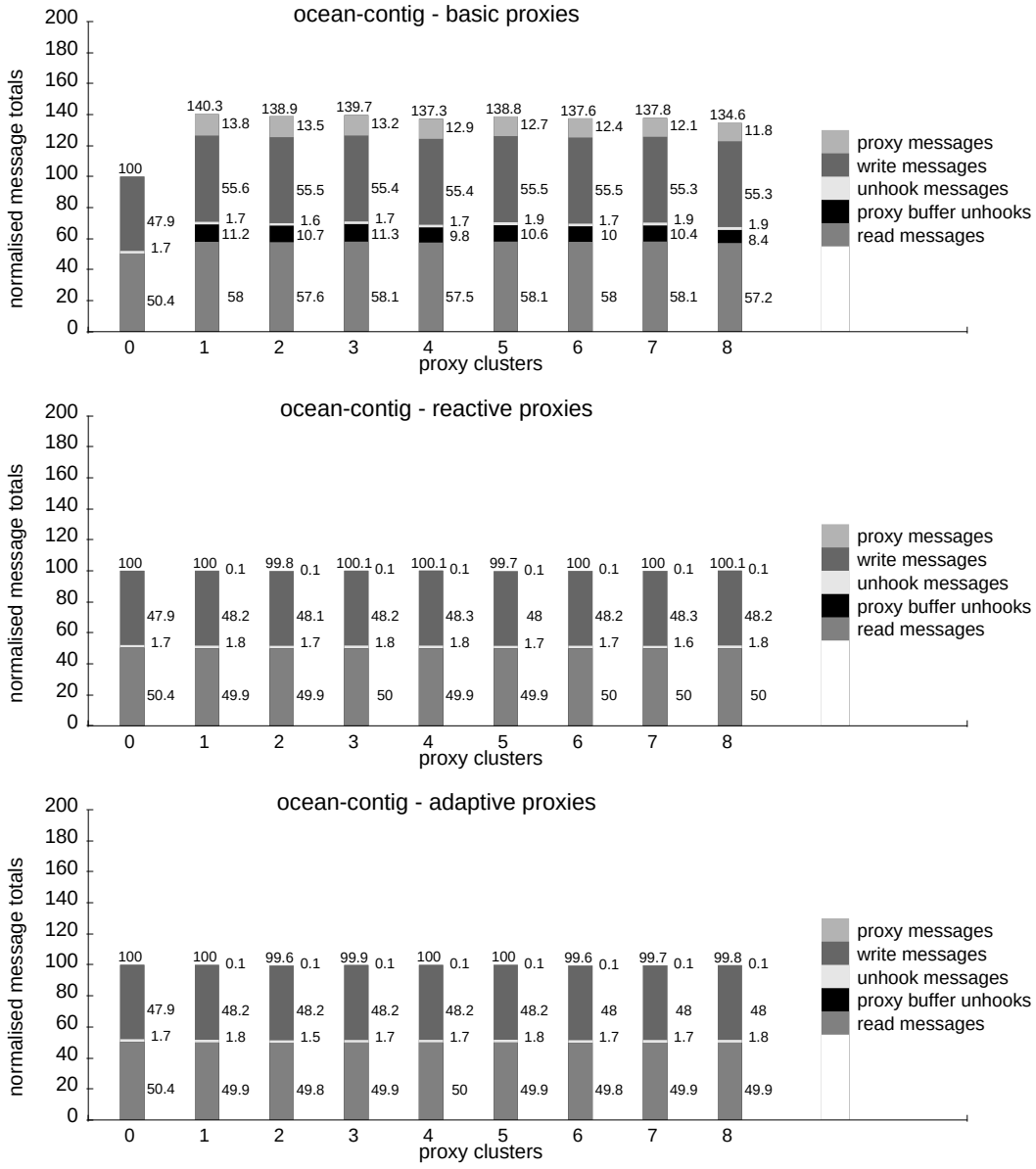


Figure 7.28: Ocean-Contig: message category profiles

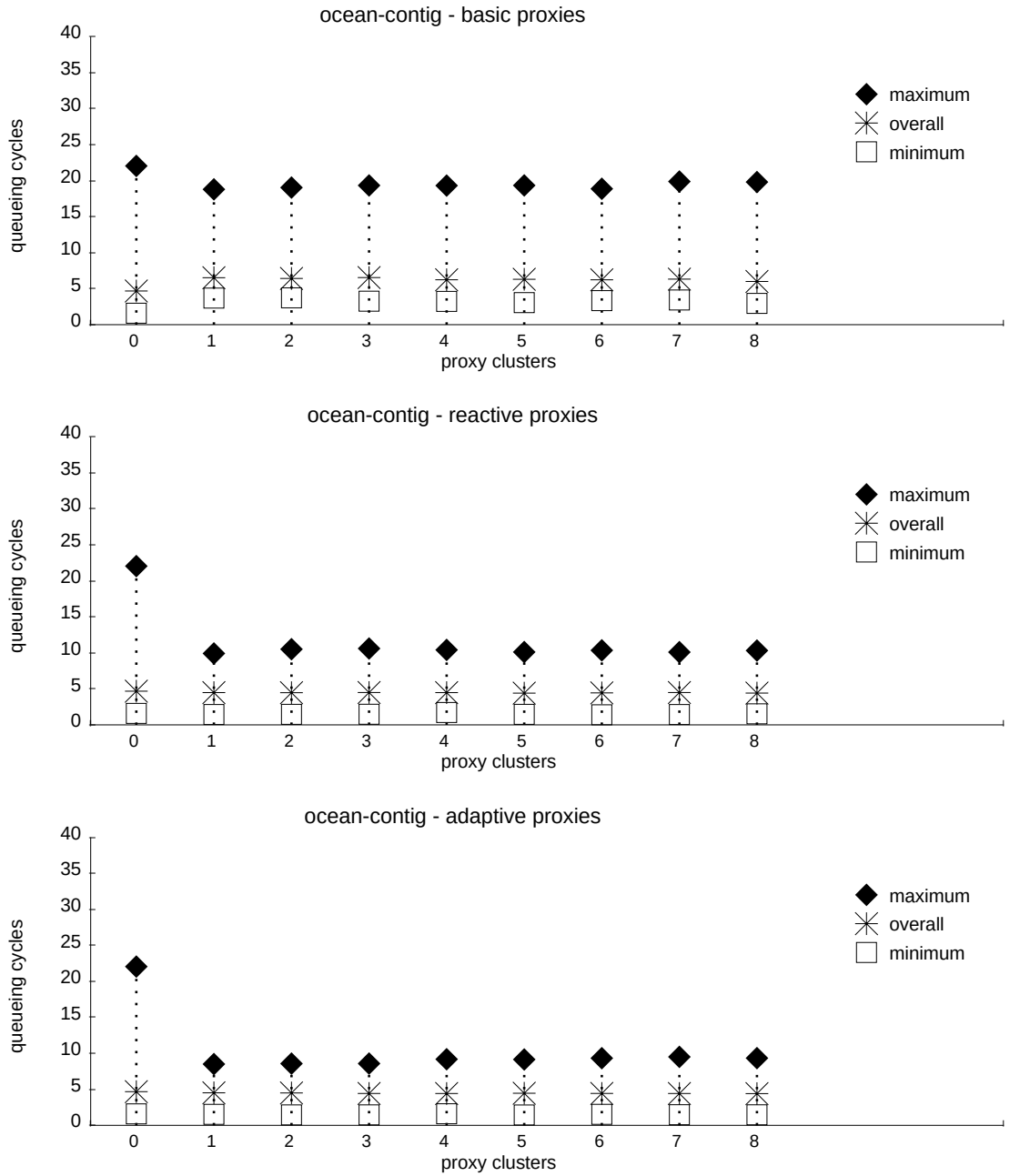


Figure 7.29: Ocean-Contig mean queueing cycles

of proxies, and changing the value of $\mathcal{NPC}$ affects which nodes receive `proxy-read-request` messages for a particular data line. Using a separate proxy buffer rather than SLC caching reduces the level of unhook messages for this application, but this can lead to a local CPU reaching a barrier earlier, which can increase the overall barrier delay. The side-effects are complex and highlight the experimental approach which would be needed before setting the value of $\mathcal{NPC}$ which would be most likely to improve the performance of a specific architecture.

For adaptive proxies, Ocean-Contig fares better using a separate proxy buffer than was the case with SLC caching (where the performance was worse for $\mathcal{NPC} \geq 1$). As with reactive

proxies, it is reduced barrier delay which produces the performance improvements. When the separate proxy buffer is used it is the adaptive proxy scheme which is best suited to the Ocean-Contig application.

7.4.7 Ocean-Non-Contig

The use of a separate proxy buffer nearly always gives performance speedups for the three proxy strategies. This is an improvement on the SLC caching and non-caching approaches, where there were a number of instances of performance degradation. With the separate proxy buffer there is only one case where the performance is impaired: for adaptive proxies when $\mathcal{NPC}$=5.

The poor performance observed when $\mathcal{NPC}$=5 for adaptive proxies stems from there being more proxy nodes on the sharing lists than is the case for other values of $\mathcal{NPC}\geq 1$. This increase is reflected by the increase in the number of write and unhook category messages seen in Figure 7.33. These higher message levels are caused by there being more `invalidate`, `home-invalidate`, and `client-unhook-forward` messages, these being the message types which are sent along sharing lists. The increase in proxy nodes on sharing lists occurs because of the interaction between the non-locality of access in the application, and the partitioning of nodes at $\mathcal{NPC}$=5. In Chapter 5 the resulting increase in unhook and invalidate messages did not lead to a drop in performance when $\mathcal{NPC}$=5 for adaptive proxies because the overall level of messages was kept relatively close to that observed when $\mathcal{NPC}$=0. In the current case, however, the lower proxy hit rate which results from turnover of entries in the proxy buffers leads to more `read-request` messages being sent to home nodes by proxies. Given that there are already many messages using the interconnection network (because of the poor locality of data access in Ocean-Non-Contig) these extra `read-request` messages lead to a rise in mean queueing delay (Figure 7.30) and the overall execution time suffers because of the queueing delays.

It is encouraging that the performance of Ocean-Non-Contig using a separate proxy buffer is much more stable than was seen with either SLC caching or non-caching proxies. The proxy buffer technique avoids the cache pollution seen with SLC caching, and improves on the proxy hit rate results seen for non-caching proxies. The separate proxy buffer provides the best performance environment for Ocean-Non-Contig, given that the application has poor data locality so is particularly vulnerable to increases in the total number of messages.

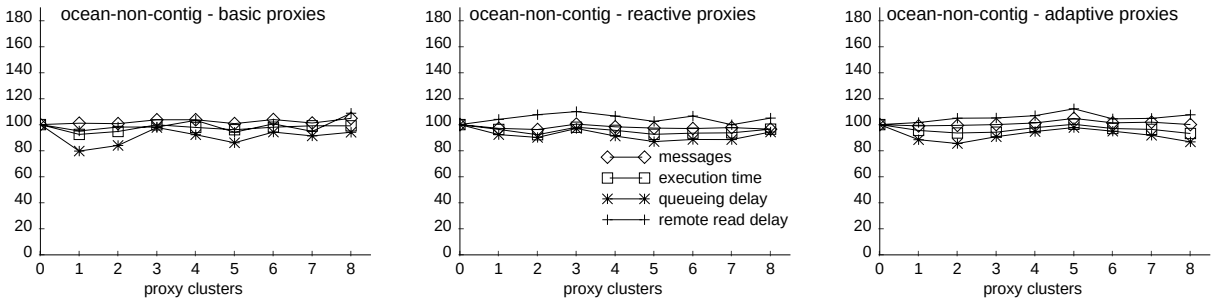


Figure 7.30: Ocean-Non-Contig: changes (relative to no proxies case)

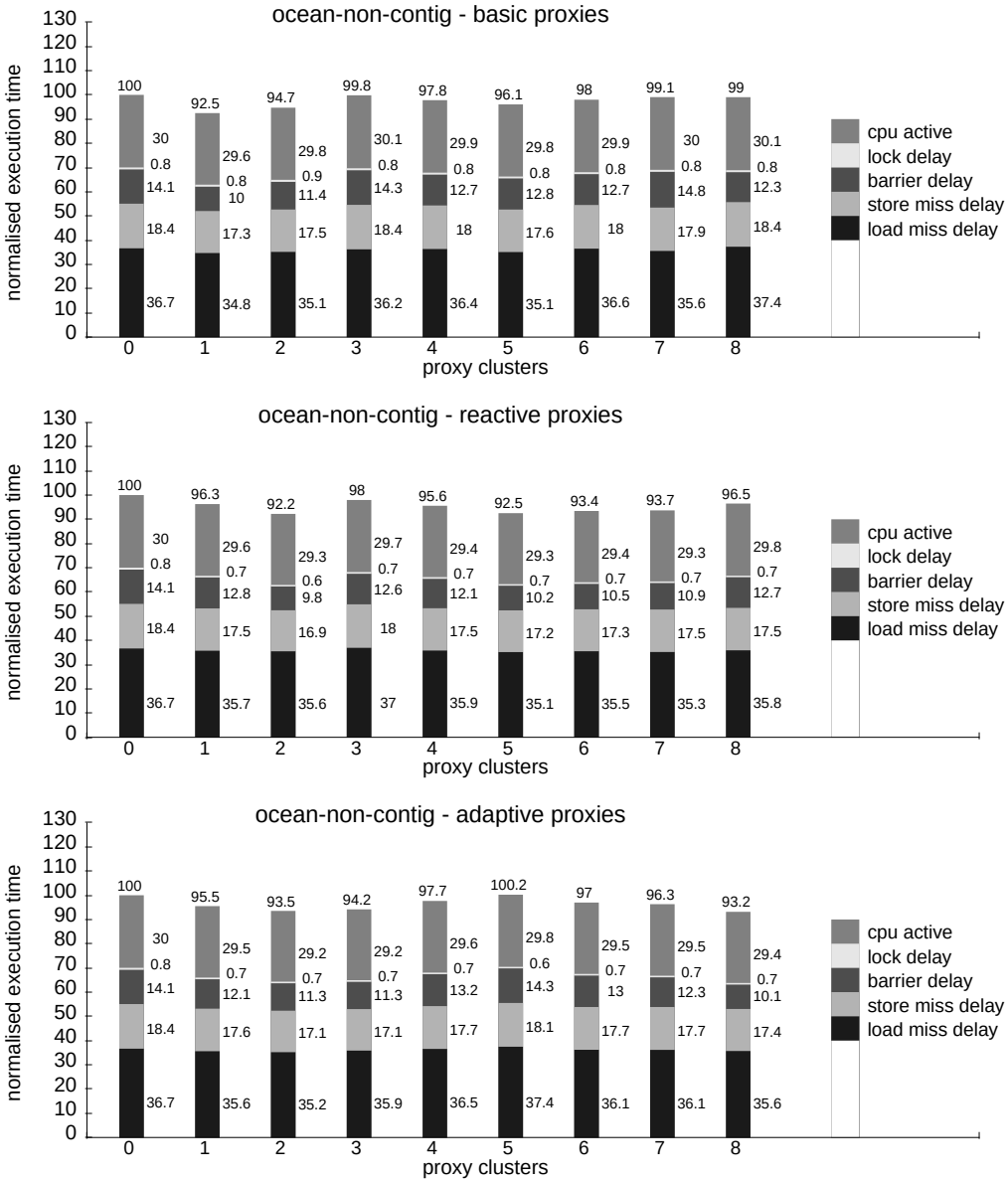


Figure 7.31: Ocean-Non-Contig: execution time profiles

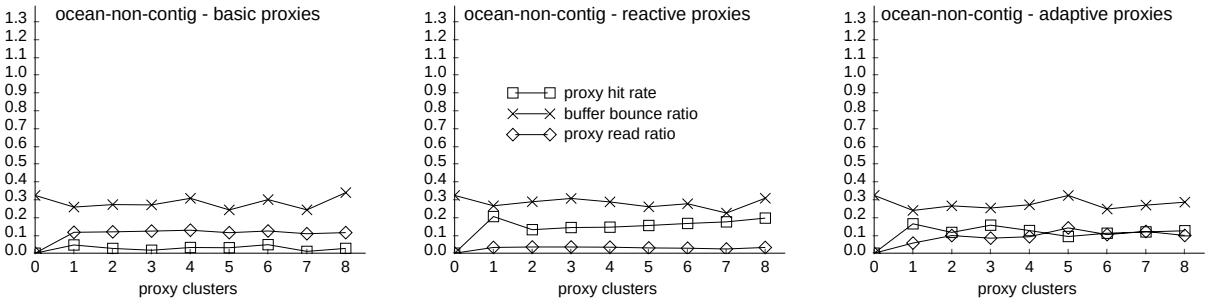


Figure 7.32: Ocean-Non-Contig: message ratios

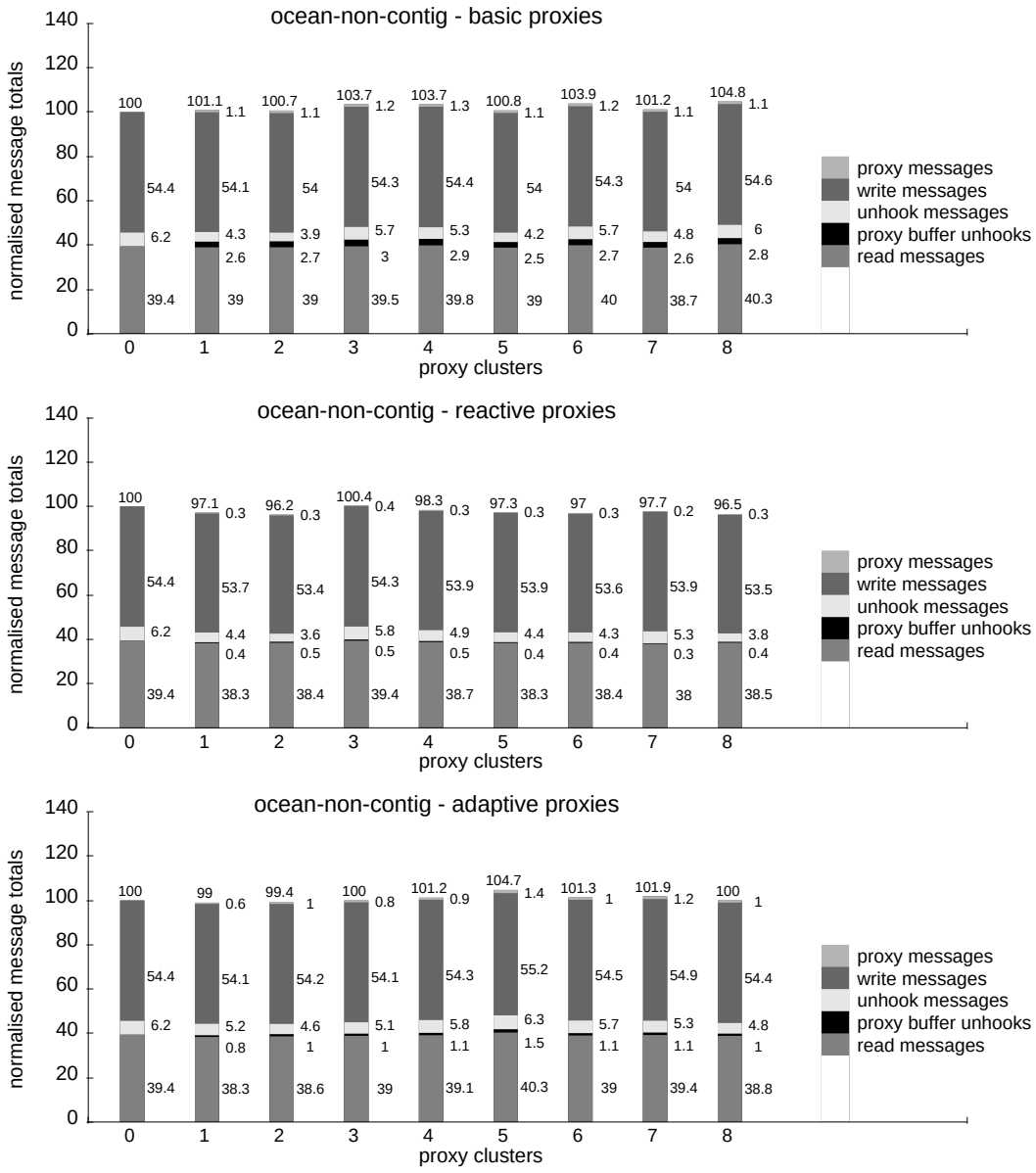


Figure 7.33: Ocean-Non-Contig: message category profiles

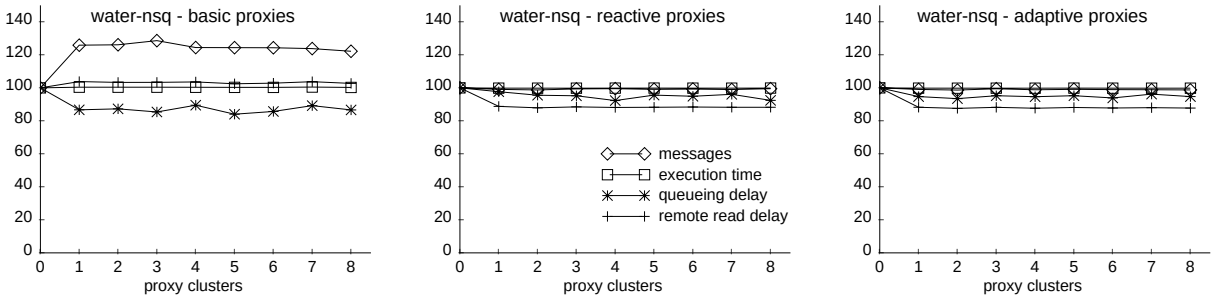


Figure 7.34: Water-Nsq: changes (relative to no proxies case)

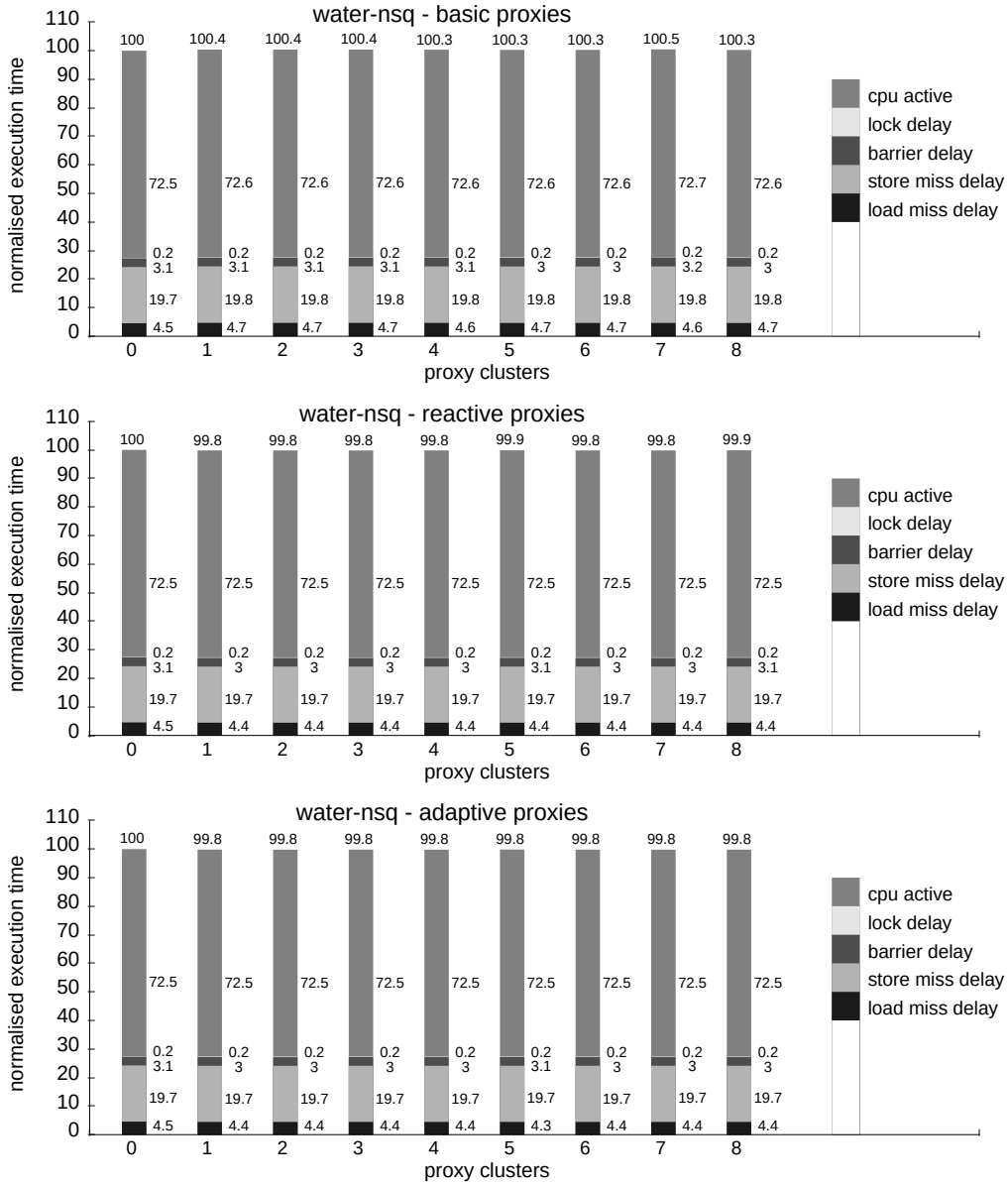


Figure 7.35: Water-Nsq: execution time profiles

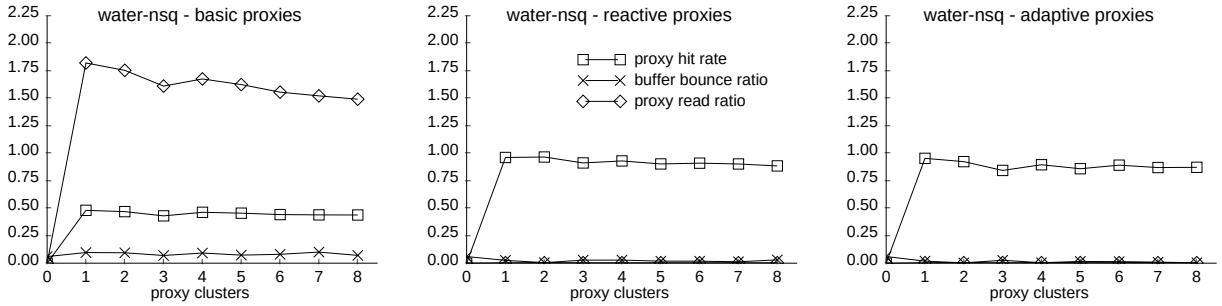


Figure 7.36: Water-Nsq: message ratios

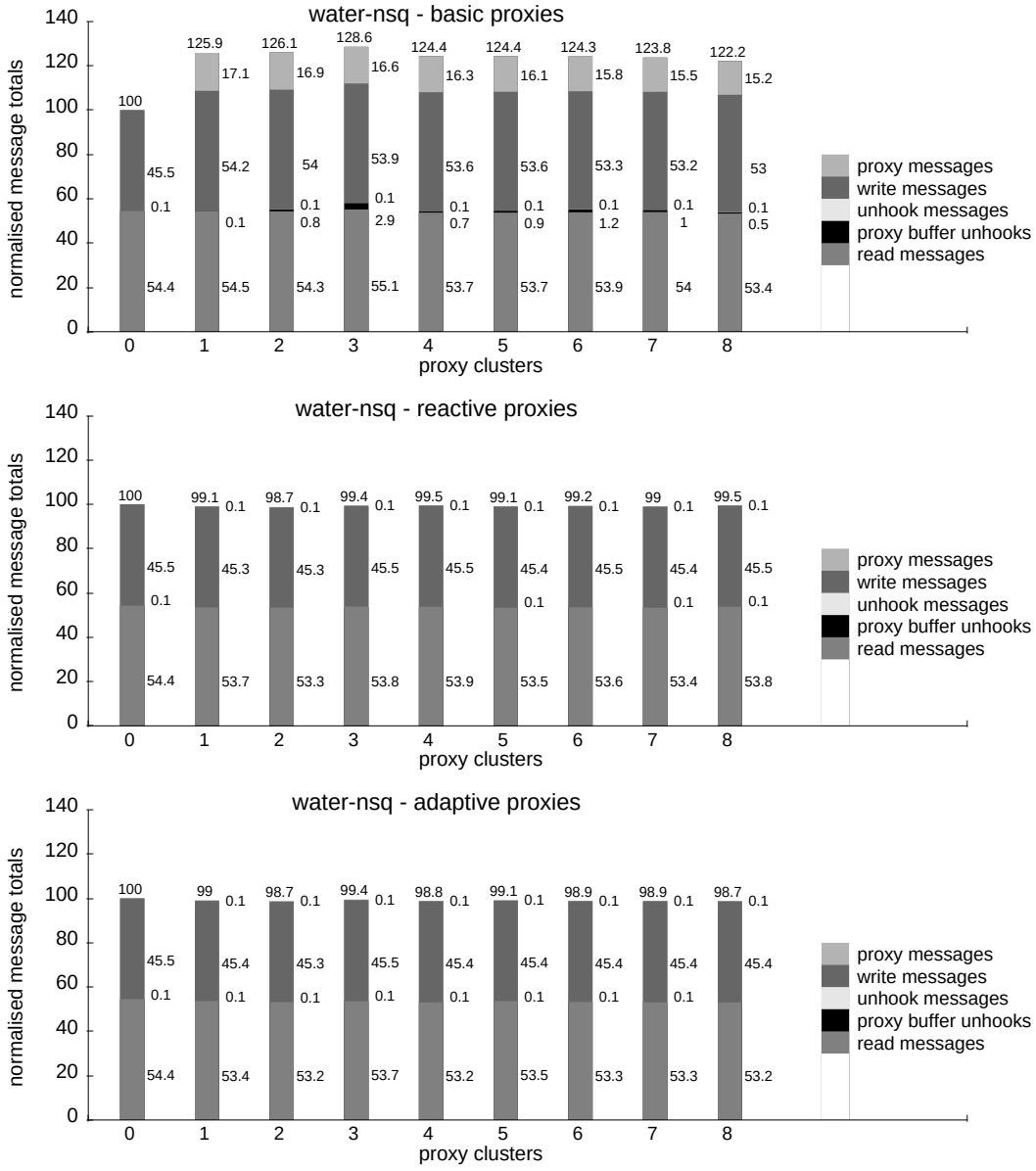


Figure 7.37: Water-Nsq: message category profiles

7.4.8 Water-Nsq

The performance of the Water-Nsq application using a separate proxy buffer is similar to that obtained with non-caching proxies. The performance degrades with basic proxies (in the range -0.3% to -0.5%), but it improves slightly with reactive and adaptive proxies by between 0.1% and 0.2%. This marginal effect of using proxies is due to the small part which load miss delays play in the overall execution profile of this application, *i.e.* around 4.5% without proxies as shown in Figure 7.35. However, it should be noted that the techniques which avoid cache pollution, *viz.* using a separate proxy buffer or not caching proxy data, have a smaller adverse effect for basic proxies than is seen when SLC caching is used.

7.4.9 Summary of Results

Using a separate proxy buffer with basic proxies can lead to a high turnover of entries in the proxy buffers. This was seen for the Barnes, FFT, and Ocean-Contig applications. The size of the proxy buffer was deliberately set to be low in order to (1) follow the approach taken with victim caches where increasing the buffer size had little effect on the hit rate [63], and (2) allow for putting the proxy buffer "on-chip" with the node controller (or minimise the amount of DRAM required). Despite the high turnover, these applications still obtained performance speedups, although they were not as good as might have been achieved by having a larger proxy buffer. Any further investigation into the use of the separate proxy buffer should consider the effects of a range of buffer sizes, bearing in mind that the performance should be better if the proxy buffer was on-chip with the node controller, although this approach is not so flexible or scalable.

Employing a separate proxy buffer keeps the proxy hit rate close to the level seen with SLC caching, *i.e.* combining is kept as high as possible. Even FFT, despite its high turnover in the proxy buffers for basic proxies, gets a slightly higher proxy hit rate than was seen with the non-caching approach. For reactive and adaptive proxies the small size of the proxy buffer did not result in a high turnover of proxy buffer entries.

It was noticeable that the proxy buffer technique avoided the cache interference patterns seen in Chapter 5 for Barnes and Ocean-Non-Contig, while keeping most of the benefits of combining (unlike the non-caching approach examined in Chapter 6). Ocean-Non-Contig in particular, which has poor data locality and so suffers when any increase in messages swamps the network, benefits from the reduction in SLC cache pollution and the combining of proxy read requests.

The results for Ocean-Contig highlighted a subtle side-effect of using proxies. For values of $\mathcal{NPC} \geq 1$ the performance was determined by the effect the use of proxies had on the overall barrier delay. The changes in barrier delay resulted from redistributing messages to proxy nodes and the delays experienced by other messages queueing for service at proxy nodes. This sensitivity to the redistribution of read requests makes it clear that careful experimentation would be needed before determining the appropriate value of $\mathcal{NPC}$ for a specific system configuration.

On balance, the adaptive proxy strategy gets the most out of the proxy buffer approach. Adaptive proxies have three balance points at $\mathcal{NPC}$=2,6,&7, which reflect the stability that can be achieved when the handling of read requests is spread around the system, and the opportunities for combining are maximised by holding copies of data at the proxy nodes without polluting the local SLCs.

The use of a small separate proxy buffer enables proxies to be implemented in a way which avoids SLC pollution and achieves most of the benefits of combining which were seen in Chapter 5. Implementing the proxy policy in this way, *i.e.* with a separate proxy buffer, would require a node controller which is capable of using a small area of the local memory for its own purposes, or which has some on-chip or off-chip storage accessible to the node controller. The first approach is the subject of current research at Stanford University, *i.e.* the MAGIC node controller within FLASH [80].

7.5 Conclusions

This chapter has investigated a scheme where proxy data is held in a separate proxy buffer. The approach avoids the SLC pollution caused by holding proxy data in the local SLC, but retains the potential for satisfying `proxy-read-request` messages at the proxy node. Given a node controller which can manage a small buffer (either on-chip or as part of the local memory), the proxy buffer technique in conjunction with adaptive proxies delivers the most stable performance, with balance points at $\mathcal{NPC}$=2,6,&7, *i.e.* there are three values of $\mathcal{NPC} \geq 1$ where the performance improves for all eight applications. Adaptive proxies do not always achieve the best performance speedup, but they provide stable performance without the need for the application programmer or compiler to mark widely-shared data structures.

Chapter 8

Summary, Conclusions, and Further Work

This chapter summarises the material presented in the preceding chapters, and assesses the extent to which the objectives set out at the start of the thesis have been met. Finally, some directions for future work in the area are considered.

8.1 Thesis Summary

The starting point of this thesis was that the unpredictable performance characteristics of shared-memory multiprocessors have hampered their acceptance. The performance anomalies stem from accessing non-local data, particularly in distributed shared-memory architectures, and are compounded by the characteristics of the interconnection network and the overheads of any cache coherence protocol. One approach to solving the problem has been to tune applications to run on a specific architecture, but this goes against the simple programming model of shared-memory, and compromises the portability of such tuned applications. The focus of this thesis is on the performance problems which arise when incoming messages have to queue for service at a processing node.

Chapter 2 presented a survey of shared-memory architectures, and considered the performance problems that can arise on cache-coherent non-uniform memory access (cc-NUMA) multiprocessors. These performance issues have been the subject of a wide range of proposed solutions, some of which are complementary (*i.e.* they can be used together) whereas others can have side-effects which work against each other. This results in system designers having to trade-off the benefits and drawbacks of various performance enhancing techniques to suit the target market for a new architecture. This was illustrated by considering the design trade-offs present in the Silicon Graphics Origin2000 architecture. Any new technique designed to

alleviate performance problems in distributed shared-memory multiprocessors must aim to minimise its side-effects on other aspects of the system.

Chapter 3 introduced the cc-NUMA architecture studied in this work, and used execution-driven simulations of eight application programs to illustrate how the level of queueing increases as the bandwidth of the interconnection network is increased. The buildup of messages was shown to be in part due to access to widely-shared data structures, *i.e.* data items which are accessed by many or all of the processing nodes. Read requests for data items are directed to the processing node which holds in its local memory the page containing the data. This home node suffers from a buildup of incoming messages when there are many simultaneous requests for the same data line. The performance problems caused by read access to widely-shared data was analysed in the chapter, and was shown to be dependent on the number of processors accessing the data, the characteristics of the network, the service time for a message at the home node, and the interval between successive requests to the same home node from the same client node. The effects of read access to widely-shared data were encapsulated in the $Contention_{home}$ equation.

Given that read access to widely-shared data is a cause of performance degradation in cc-NUMA multiprocessors, Chapter 4 introduced a protocol modification designed to reduce the number of `read-request` messages arriving at the home node. The basic proxy protocol distributes read requests to nodes other than the home node, using these other nodes as intermediaries. If these "proxy" nodes already have a copy of the data line it is sent to the client. Otherwise the proxy node will keep a note of the client, and send a `read-request` on to the home node. When the data arrives back at the proxy a copy is retained to satisfy any later requests, and the data line is sent on to all the waiting client nodes. Not all the shared-data used by an application will be widely-shared, so the basic proxy technique is only applied to read requests for data marked as widely-shared (by the application programmer or compiler). The proxy technique was evaluated using execution-driven simulations. It improved the performance of some applications, but was shown to have a small adverse effect on other applications where the indirection introduced by going via proxies outweighed any reduction in queueing times at the home node. It was also noted that relying on the application programmer to mark all the widely-shared data (and not mark any other data structures) meant that to get the best performance the programmer had to have detailed knowledge of the applications.

Chapter 5 introduced two forms of automatic proxying, where read requests are only sent via proxies when run-time queueing is detected at the home node. The trigger is the arrival of a `buffer-bounced-read-request` at a client node. Reactive proxies then send the read request to a proxy node, but later read requests destined for the home node will not be proxied

unless they are "buffer-bounced". Adaptive proxies continue to send any read requests for the congested home node via proxies until a proxying period has expired. The proxying period is adjusted depending on the interval between buffer-bounces from a home node. Execution-driven simulations of the eight application programs showed that it was possible to improve the performance of all eight applications using reactive proxies. However it was noted that using the local second level cache (SLC) to hold copies of proxied data could cause the eviction of data lines which were still needed by the local processor.

Chapter 6 and Chapter 7 explored two ways of avoiding this pollution of SLCs by proxy data. In Chapter 6, data copies were no longer held at proxy nodes unless they were needed for local processing. The results from execution-driven simulations showed that avoiding SLC usage by proxy data led to further performance improvements for some of the applications. Unfortunately, the non-caching form of proxies has the disadvantage that subsequent read requests arriving at a proxy require a fresh `read-request` message to be sent on to the home node, *i.e.* the proxy hit rate is lower with non-caching proxies. As a result the performance obtained for some applications was not as good as that seen in Chapter 5. That being said, it was possible to improve the performance of all eight applications using reactive or adaptive proxies, each of which had one balance point (*i.e.* a value of $\mathcal{NPC} \geq 1$ where the performance of all eight applications is improved).

Chapter 7 addressed the reduced proxy hit rate seen for some applications when non-caching proxies were used. By employing a small separate buffer (with a few tens of entries) to hold proxy data, subsequent proxied read requests from clients could be satisfied at the proxy nodes. The separate proxy buffer technique was evaluated using execution-driven simulations with the proxy buffer assumed to have the same access latency as the local memory at each processing node. As was to be expected, the proxy hit rate improved in comparison to non-caching proxies, and the SLC pollution seen in Chapter 5 was reduced. Overall, the best performance was obtained using adaptive proxies which benefited from reduced SLC pollution during the proxying period. Adaptive proxies improved the performance of all eight applications at three "balance points", *i.e.* three different ways of partitioning the processing nodes to decide which node will be the proxy for a given client node and data line. Reactive proxies also improved the performance of all eight applications, but only at one balance point.

8.2 Conclusions

The purpose of this thesis was to study the causes of erratic performance on distributed shared-memory multiprocessors. In particular, the following objectives were set out in Section 1.1.1:

1. What are the causes of the performance anomalies observed for applications running on distributed shared-memory multiprocessors?
2. Have any of these causes been neglected by the research to date?
3. Can an architectural technique be found to alleviate such a performance problem?
4. Can this technique be designed to minimise its side-effects on existing performance-enhancing techniques such as caching?

These four questions have been addressed in the thesis. Chapter 2 investigated the current state of shared-memory multiprocessors, with particular emphasis on the causes of the performance problems which have been observed on these systems (Section 2.2) and the techniques which have been proposed to address these performance problems (Section 2.3).

A combination of execution-driven simulation and analysis was used in Chapter 3 to demonstrate that the performance problems associated with contention for node controllers are exacerbated by read accesses to widely-shared data. The problem of widely-shared data has tended to be obscured by researchers categorising applications in terms of their dominant access patterns, rather than considering that relatively infrequent access patterns can have a significant effect on performance.

The proxy strategy is proposed as an architectural technique which is designed to alleviate the home node controller contention caused by access to widely-shared data. The technique builds on past research work, but it is novel because it uses proxies for read access to explicitly marked widely-shared data, uses a different proxy for successive data lines, combines the read requests at the proxy node, and does not require additional specialised processors to handle proxying.

The reduction of side-effects from using proxies was the subject of Chapters 5, 6, and 7. Employing reactive or adaptive proxies avoided any requirement for the programmer (or compiler) to mark widely-shared data, and so removed the risk of proxying the wrong data. Problems with SLC pollution were alleviated by using non-caching proxies, although this technique suffered from the reduction in combining of requests. Using a separate proxy buffer with adaptive proxies gave the most stable performance, with the reduction in side-effects leading to performance improvements for all eight application programs at three balance points (*i.e.* when $\mathcal{NPC}$=2,6,&7, as shown in Table 8.1).

Adaptive proxies with a separate proxy buffer give stable performance, allow the programmer to write portable applications which are less "architecture specific", and save on performance tuning because the widely-shared data access bottleneck is dealt with automatically by the

application	relative speedup no proxies	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
		1	2	3	4	5	6	7	8
Barnes	46.3	0.0	+3.3	+0.4	+0.2	+0.2	+0.2	+0.4	+0.4
CFD	28.3	+9.4	+9.4	+9.0	+12.5	+10.7	+10.8	+10.5	+12.7
FFT	47.3	+11.9	+11.9	+11.6	+11.8	+11.4	+11.4	+11.0	+10.8
FMM	52.4	+0.4	+0.3	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4
GE	21.6	+30.7	+30.9	+31.8	+31.3	+31.8	+31.8	+31.5	+31.7
Ocean-Contig	49.7	-2.4	+1.5	-1.5	-6.8	-0.2	+1.9	+0.8	-0.7
Ocean-Non-Contig	48.2	+4.5	+6.5	+5.8	+2.3	-0.2	+3.0	+3.7	+6.8
Water-Nsq	55.3	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

Table 8.1: Benchmark relative speedups for adaptive proxies with a separate proxy buffer (extracted from Table 7.2)

run-time system. It should be remembered that proxies have the overheads of representing the proxy pending chain, and require minor changes to the "state machine" in a programmable node controller to handle the extra states and messages associated with proxies. However, these costs are one-off in that they only apply when the system is being designed and built. Proxies can then reduce execution times, and they save the application programmers from having to write system-specific programs.

8.3 Further Work

The work reported in this thesis has raised a number of issues which deserve further attention:

- Using a more complicated architectural model in terms of the network topology and the number of processors at each node.
- Choosing alternative proxy nodes when the proxy candidate is itself suffering from node controller contention.
- Employing a different representation of the sharing list.
- Using more than one level of proxies for larger configurations.
- Investigating the effects of varying the proxy buffer size, replacement policy, and location.
- Checking the applicability of proxies to commercial workloads.

In order to study proxies in sufficient detail to detect their side-effects on other transactions, caches, and queue lengths it was necessary to make some simplifications to avoid being swamped by too much data and to avoid tying the results to a specific architecture. To this end the network was modelled as a contention-free full crossbar, and each processing node contained only one local CPU. This approach was successful because it enabled the differences between the various approaches to proxying to be evaluated. However any further work on, for example, the commercial viability of using adaptive proxies with a separate proxy buffer would need to assess the impact of different network topologies and of having more than one CPU at each processing node.

A separate simplification was the decision to only "buffer-bounce" the `read-request` type of messages when investigating automatic proxying. Any hardware implementation would have to cater for handling full buffer conditions for all the message types, and this will considerably increase the number of conditions which have to be handled by the coherence protocol. In such an environment it would be necessary to define the action to be taken when a client received a `buffer-bounced-proxy-read-request`. Rather than just retrying the request, or sending it directly to the home node, it might be sensible to select an alternative proxy. However it should be noted that there is the possibility that any alternative proxy might itself already be congested.

The increase in `client-unhook-forward` messages, which arises in Chapter 6 when the unhooking node is further along the sharing list, is an effect of using a singly-linked list to represent the sharing list in the Stanford distributed-directory protocol [134]. If the sharing list had been represented in another way, for example as a doubly-linked list or a bit vector, then the "position" of a node in the list would not have changed the number of messages required to complete an unhook transaction. Such alternative implementations should improve the performance results observed for applications like Barnes where the position of a proxy node in the singly-linked sharing list has a noticeable effect on performance.

The execution-driven simulation results reported in this thesis were for up to sixty-four processing nodes. For larger configurations of many hundreds or thousands of nodes it would be prudent to use a hierarchical approach to proxy selection to avoid the proxy nodes themselves becoming performance bottlenecks. This would complicate the proxy selection algorithm, but it appears to be feasible.

Using a separate proxy buffer, as reported in Chapter 7, shows encouraging results for adaptive proxies. However the results indicate that there are a number of areas which warrant further investigation. In particular, the following design parameters have a bearing on the performance of the strategy: the size of the proxy buffer, its replacement policy (*e.g.* using

"least-recently-used" rather than "first-in-first-out"), and its access latency. The values used in Chapter 7 were deliberately chosen to be conservative, in that the proxy buffer was small, no account was taken of the usage of entries when it came to eviction, and situating the buffer on the MEM bus gave long access latency. To evaluate the commercial viability of using adaptive proxies with a proxy buffer it would be necessary to investigate the performance effects of more aggressive designs.

The eight application programs used in this thesis were taken from the scientific and engineering domains. In the past such computation-intensive applications have provided the main market for large scale multiprocessors. There is now a trend towards designing systems which also support commercial applications such as large database systems (*e.g.* for data-mining and real-time response) and composite workloads [100]. The study of such applications and their interaction with distributed shared-memory systems is an active research area, and further evaluation of proxies should aim to include benchmarks from the commercial domain.

Appendix A

Uniprocessor Caches

The execution time of a program is critically dependent on the rate at which instructions and data can be fetched from and written to memory. Unfortunately, while processor speeds have increased at about 50% per year over the last two decades, main memory speeds have grown at a much slower rate [54]. As a result, the ability to execute instructions and process data far outstrips the rate at which main memory can provide them. To rectify this mismatch, most uniprocessor computers now include caches: small, fast memories which are physically close to the CPU and which provide the instructions and data needed with a shorter latency that is more in line with the CPU's needs [44].

Caches exploit the fact that the memory references made by programs tend to be clustered in time and space rather than being to randomly distributed addresses [118]. There are two forms of locality which are observed in uniprocessors: temporal and spatial.

Temporal locality: items which have been referenced recently are likely to be referenced again soon, *e.g.* loops, stacks, temporary variables.

Spatial locality: programs tend to access items whose addresses are near one another, *e.g.* arrays, code segments, stacks.

In addition, the working set of the program, *i.e.* the subset of addresses referenced in some time interval, only changes slowly during program execution [30].

Caches work by automatically retaining information that the CPU has used or generated recently, the aim being to keep as much of the working set as possible in fast storage. Data is brought into cache as a block of contiguous data, the length of which is determined by the cache line size, and this will exploit any spatial locality. The slower main memory only has to be accessed when the cache does not contain the necessary information or when that memory has to be updated, *i.e.* when a data line is "evicted" (*aka.* "flushed") from the cache.

The location in the cache of a data line containing a particular data item is based on the address of the item and the type of cache organisation. If each data line has only one place it can appear in the cache, the cache is said to be *direct mapped*. If a data line can be placed anywhere in the cache, the cache is said to be *fully associative*. If a data line can be placed in a restricted set of places in the cache, the cache is said to be *set associative*, where a set is the group of two or more cache lines where the data can be held. A data line is first mapped on to a set, and then the data line can be placed anywhere within that set. Figure A.1 shows the address partitioning used to determine the set number and identifying tag for a data item. Figure A.2 shows an example of matching an address with a four-way set associative cache. In the example there are N sets, each of which contain four lines, *i.e.* there are four places in the cache where the data line could be held.

An important cache design issue is the choice of write policy. A write-through policy sends all writes through to the memory. This keeps the memory hierarchy up-to-date, but the strategy has the drawback that each write-through of data requires the use of the connection between caches and memory. This takes time, and there is the added drawback that it prevents other uses of the connection during the update. Alternatively, a write-back policy will only update the next level of the memory hierarchy when a modified data line is evicted from the cache. A "dirty bit" is used to support this policy, and this is set when a cache line is modified. The problem with this approach is that main memory will not always contain the up-to-date version of the data line. A compromise in two-level cache systems is to employ the write-through policy for the first level cache (FLC), and to use the write-back policy for the second level cache (SLC). In this way the SLC is kept up to date, but the update traffic to the memory is reduced. If the memory is shared, it will have to have some mechanism for indicating that the SLC may hold a more up-to-date copy of the data line.

A cache miss occurs when the cache does not hold the data demanded by the CPU. The consequence of a cache miss is that the item must be fetched from the slower main memory. In uniprocessors, a cache miss will occur for one of the following reasons [55]:

Compulsory: the first access to a line of data requires it to be brought in from main memory.

Capacity: when the cache runs out of room a replacement policy will discard a data line, which may be required later.

Conflict: occurs when more than one line of main memory maps to the same cache line; the conflicting data line must be discarded, and it will have to be retrieved from memory if it is required again.

When a cache miss occurs, and all suitable locations in the cache are already in use, a data

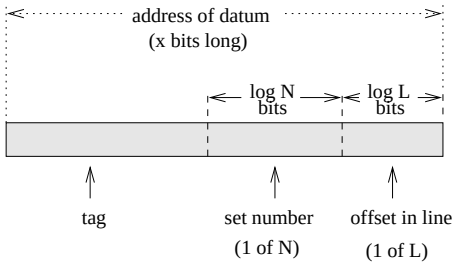


Figure A.1: Address partitioning for a cache search

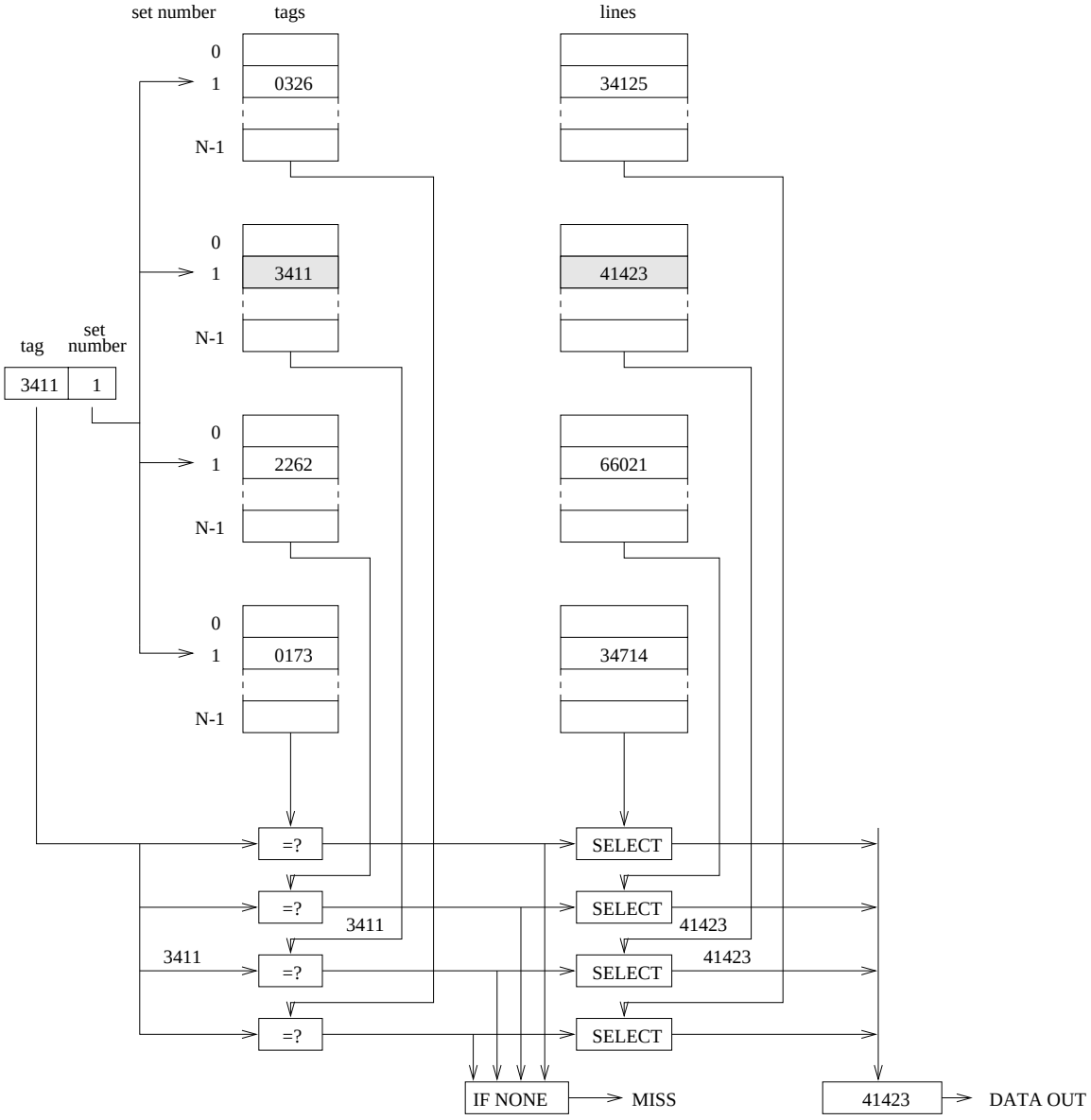


Figure A.2: An implementation of a four-way set-associative cache with N sets [128]

line must be selected for eviction to make room for the new data line. With a direct-mapped cache there is only one candidate for eviction. For caches with associativity of more than one a replacement policy is needed. Two of the common strategies for selecting the cache line for eviction are [49]:

- **Least-Recently Used (LRU):** the cache line selected is the one that has been unused for the longest time. The LRU policy exploits temporal locality, *i.e.* if recently used data lines are likely to be used again, then the best candidate for eviction is the least recently used data line. To implement this policy the system has to keep track of the latest access to each cache line.
- **Random:** to evenly distribute the replacements, the line to be evicted is chosen at random from the candidates. This policy has the advantage of being simple to implement, and for larger cache sizes the resulting miss rates are comparable to LRU [54].

Appendix B

The ALITE Execution-Driven Simulator

This project relied on a simulator to investigate the effects of the different forms of proxying. The simulator used was ALITE, an execution-driven simulator written by Ashley Saulsbury. This appendix gives an overview of the various techniques used to simulate computer architectures, and describes the ALITE simulator.

B.1 Simulation of Computer Architectures

A simulator is a piece of software which emulates the behaviour of input workloads in a particular environment. The environment includes all the system features that determine the system's behaviour under a given workload. System software, processor architecture, memory system, and connection network are all included in the environment. The system being studied is called the *target* to distinguish it from the *host*, which is the system on which the simulator runs. Parallel program execution can be simulated on a sequential host by using one of the three alternative schemes: full emulation, using application-derived (or stochastically generated) traces, or execution-driven simulation.

Full emulation is where the architecture under evaluation is simulated completely. The action of each machine instruction is modelled for the processor registers, pipeline, arithmetic and logic unit (ALU), and each level of the memory hierarchy. The advantage of this approach is that it is highly reliable because the results of the simulation can be expected to correspond closely to an implementation of the architecture being modelled. The main disadvantages are the huge computational resources required and the significantly longer time it takes to simulate the execution of a program.

The behaviour of a program executing on a particular architecture is largely determined by its stream of memory references. An address trace describes the accesses made by a program, *i.e.* the address and whether the access is a read or write. Address traces are used as the input to simulators which concentrate on modelling the behaviour of the memory hierarchy. The traces themselves are either derived from applications, *i.e.* they are produced by running real programs, or they are stochastically generated by a synthetic reference generator. The disadvantage of trace-driven simulators is that they do not allow for feedback effects from different memory hierarchies, *i.e.* the trace is static.

Currently the most commonly used simulators for shared-memory multiprocessors are based on *execution-driven* simulation, where non-memory instructions are executed directly on the host system (which can itself be a parallel computer), and only the memory instructions are actually simulated for the target architecture. Examples of such simulators include the TangoLite simulator used at Stanford [43], and the Wisconsin Wind Tunnel [109].

B.1.1 Execution-Driven Simulation

An execution-driven simulator is one where the simulation is driven by executing the applications so that the interaction between the target's processors can affect the course of the simulation. For multiprocessor systems, execution-driven simulation is important because it allows for accurate modelling of the effects of synchronisation and contention. In execution-driven simulations, the execution of the application program on the host is interleaved with the simulation of the target architecture. Each processor is represented by a process, and sequences of application instructions are directly executed on the simulation host until a global event is generated. A global event is a process interaction, *i.e.* an action which can alter the execution of another process, *e.g.* accessing a shared data structure, sending a message, or attempting to acquire a lock. When a process generates a global event, the architectural simulator takes over in order to determine how long the event will take, *i.e.* its latency.

Figure B.1 shows an example of how an execution-driven simulation proceeds for a four processor target system. When the simulator starts, the first process is allowed to execute until it generates a global event, at which point the process blocks. The other processes are then executed in turn, up to the point where they generate a global event. When all the processes are blocked, the situation is as shown in Figure B.1(a).

When all the application processes have generated global events, the architectural simulator is invoked. The simulator selects the process with the earliest timestamp because this cannot affect the behaviour of the other processes at the point at which they have blocked. Process

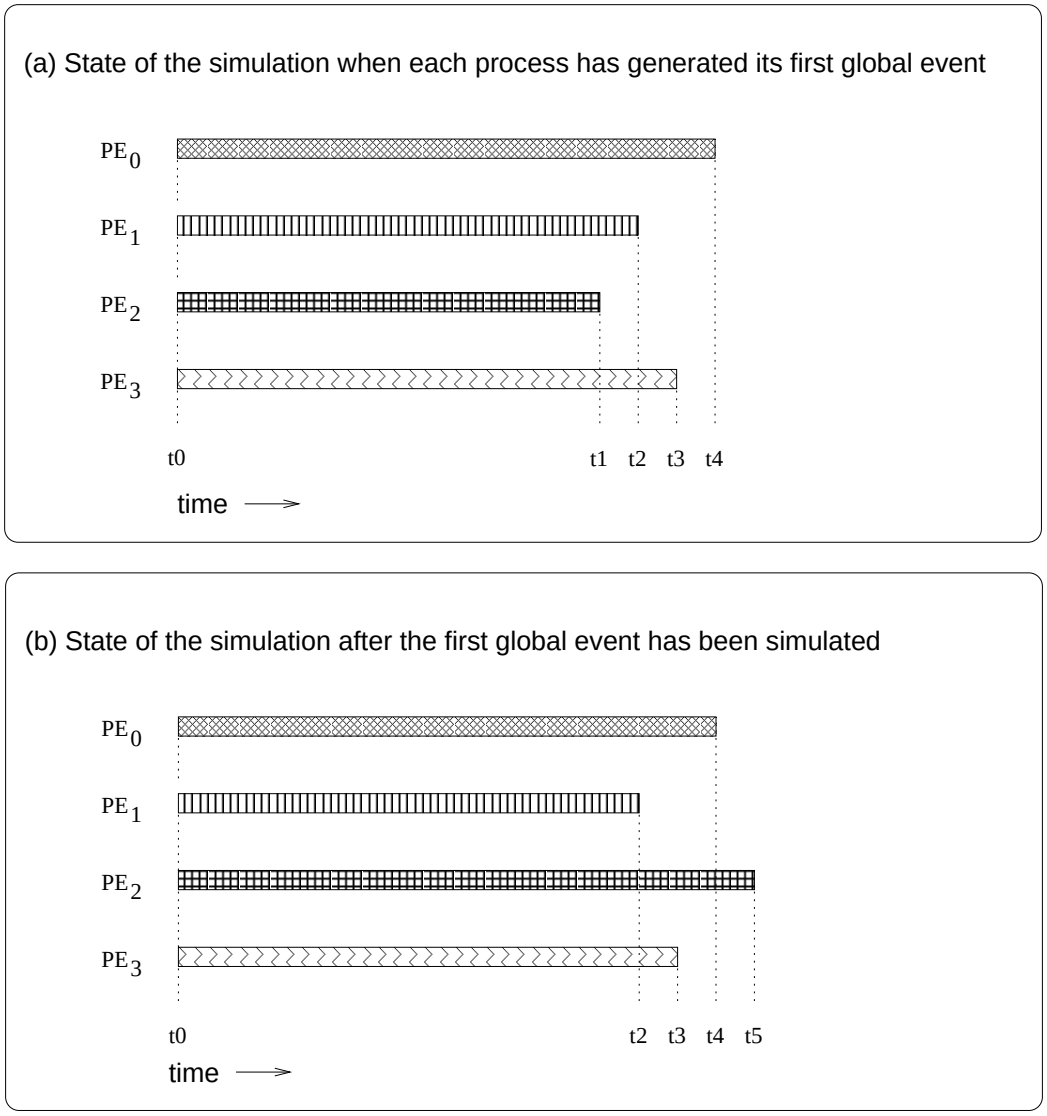


Figure B.1: Example of a four processor execution-driven simulation

PE_2 has the earliest timestamp, and so its event is simulated. If, for example, the event was a shared-memory write, and the simulator is modelling an invalidation-based cc-NUMA system, the invalidation will be observed by the other processes when they restart. This is the correct behaviour because the other processes will not need to access that data until at least the time at which they have blocked (any access to that data would generate a global event and cause the process to block).

When the global event has been simulated, the clock of process PE_2 is updated to reflect the simulated latency of the event, and process PE_2 is then restarted. Eventually it will generate another global event and it will block at time t5 as shown in Figure B.1(b). The simulator is again invoked and chooses the processor with the earliest timestamp for simulation, which in this case is process PE_1. This alternation between running processes and the architectural

simulation continues until all the processors have signalled that they have completed. When this happens the simulation finishes.

B.2 ALITE

The ALITE simulator was originally designed to run on a DEC Alpha host, but was adapted for the proxy work to run on Sun Sparc workstations and on PCs running Linux. It supports applications written in C or Fortran which use the Argonne National Laboratory's parallel macros package for parallel constructs such as locks, barriers, and shared data [19].

The cc-NUMA target architecture provided with ALITE consists of a number of processing nodes, each of which contain a CPU with on-chip FLC and TLB, an off-chip SLC, some local memory (DRAM), and a node controller. All timings in the system are measured in terms of clock cycles. Table B.1 shows the latencies defined for the simulations produced for this thesis. The size of the FLC, TLB, and SLC are parameters set at compile-time, *i.e.* when an application program is compiled to a simulation which will run on the host, the resulting object code includes all the ALITE functions needed to simulate the target architecture. The DRAM size and the number of processing nodes are run-time parameters. Cache coherence is maintained using the Stanford distributed-directory protocol [134]; this is described in Chapter 3 of this thesis, and further details about the protocol's implementation in ALITE are given in Appendix C. The line size has been set to 64 bytes throughout the memory hierarchy, *i.e.* for the FLC, SLC, and DRAM.

B.2.1 FLC

The first level caches (FLC) are direct-mapped, and use a write-through policy. The FLC size for each node was set to 8 Kbytes for the results reported in this thesis. The size of the FLC used by ALITE is defined by setting FLC-SIZE-BITS. Each cache line holds 64 bytes of data.

B.2.2 Translation Look-Aside Buffer

The translation look-aside buffer (TLB) is a buffer which holds address translations from logical to physical addresses. The TLB relies on the principle of locality to reduce access times for address translations. Page tables are usually so large that they are stored in the local memory (DRAM). By keeping the recent address translations in a special buffer, if the address translation is needed again it is usually still in the TLB. A TLB entry is like a

#define name	delay cycles	comments
CPU-SLC-GET-DELAY	2	Cycles it takes the cpu to arbitrate for the SLC bus
CPU-SLC-RELEASE-DELAY	1	Cycles it takes for the cpu to let go of the SLC bus
CPU-SLC-ACCESS-DELAY	6	Delay to address and tag check the SLC, and potentially read first bus word
CPU-SLC-LINE-ACCESS	18	Once SLC is accessed, number of cycles for cpu to read the rest of the line
CPU-MBUS-GET-DELAY	3	Delay for the cpu to get the mbus once it becomes available
CPU-MBUS-RELEASE-DELAY	2	number of cycles taken by the cpu to release the mbus
CPU-DRAM-ACCESS-DELAY	20	Delay to decode and address a data line in DRAM
CPU-DRAM-LINE-ACCESS-TIME	24	Time to read whole data line from DRAM - once it has been accessed via CPU-DRAM-ACCESS-DELAY
CTRLR-DRAM-ACCESS-DELAY	20	Delay to decode and address a data line in DRAM
CTRLR-DRAM-LINE-ACCESS-TIME	24	Time to read whole data line from DRAM - once it has been accessed via CTRLR-DRAM-ACCESS-DELAY
CPU-CTRLR-ACCESS-DELAY	5	number of cycles taken for the cpu to initiate an action (cache replacement/miss) by the controller
CPU-TLB-LOAD-TIME	80	time taken for a TLB entry to be loaded from DRAM - typically a DRAM access + some time for the handler
CTRLR-MBUS-GET-DELAY	3	Number of cycles for the controller to negotiate for the mbus once available
CTRLR-MBUS-RELEASE-DELAY	2	Number of cycles for the controller to release the mbus
CTRLR-SLC-GET-DELAY	2	Number of cycles for the controller to negotiate for the SLC bus once available
CTRLR-SLC-RELEASE-DELAY	1	Number of cycles for the controller to release the SLC bus
CTRLR-SLC-LINE-FILL-DELAY	6	Assuming the controller already knows the cache way, this is the number of cycles to fill an SLC line
CTRLR-SLC-FIRST-WORD-DELAY	6	number of cycles for the controller to place the critical word into the SLC upon a fill, and release the cpu
CTRLR-SLC-ACCESS-DELAY	6	Delay for controller to perform a tag check on the SLC to check its contents
CTRLR-SLC-LINE-READ-DELAY	24	Delay to read back data from SLC - does not include access and decode time - typically for remote write-back
CPU-PROXY-BUFFER-ACCESS-DELAY	20	Delay to decode and address a data line in the proxy buffer
CPU-PROXY-BUFFER-LINE-ACCESS-TIME	24	Time to read whole data line from the proxy buffer - once it has been accessed via CPU-PROXY-BUFFER-ACCESS-DELAY
CTRLR-PROXY-BUFFER-ACCESS-DELAY	20	Delay to decode and address a data line in the proxy buffer
CTRLR-PROXY-BUFFER-LINE-ACCESS-TIME	24	Time to read whole data line from the proxy buffer - once it has been accessed via CTRLR-PROXY-BUFFER-ACCESS-DELAY

Table B.1: Latencies of the node actions

cache entry, where the tag holds portions of the virtual address, and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit [54].

In the simulator, the TLB is modelled to be on-chip with the CPU and FLC. It is accessed in parallel with the FLC, so if there is an FLC miss then the physical address will already be available for accessing the SLC. The TLB size for each node was set to 64 entries for the results reported in this thesis. The size of the TLB used by ALITE is defined by setting TLB-HASH-ENTRIES.

B.2.3 Second Level Cache

The second level caches (SLC) are direct-mapped, and use a write-back policy. The SLC size for each node was set to 4 Mbytes for the results reported in this thesis. The size of the SLC used by ALITE is defined by setting SLC-SIZE-BITS.

B.2.4 Memory

The local memory (DRAM) at each node holds a number of pages of data; each page is 8 Kbytes in size, and contains 128 lines of data. Each data line has associated with it a state variable and a pointer to the head of the sharing list. The DRAM size at each node is a run-time parameter.

The ALITE simulator provides two page placement policies: first-touch-after-initialisation (the default) and round-robin. The round-robin policy is invoked by compiling with RR-PAGE-PLACEMENT defined; if it is not defined then first-touch is used. All the results presented in this thesis use the first-touch default, but the round-robin policy was used in a separate study to investigate the performance trade-offs between different page placement policies and the use of proxies [131].

Appendix C

The Cache Coherence Protocol

This appendix documents the cache coherence protocol implemented in the ALITE simulator. It also describes the different message types, and shows how the messages are grouped into the reporting categories used to present the simulation results in the main body of the thesis. The cache coherence protocol used in ALITE is based on invalidations, and it uses a write-back approach for updating the local memory (DRAM) from the SLC. The coherence policy is the Stanford distributed-directory protocol, described by Thapar and Delagi [134].

The caches that share a copy of a data line are linked together by a singly-linked list, which starts in the directory entry for the data line. The directory entry is held in the DRAM at the home node for the page containing the data line. In addition to the start of the sharing list, the directory entry also indicates the current state of the data line. A data line in local memory will be in one of four states:

Home-Exclusive: held at only the home node (may be modified in the local SLC).

Home-Shared: held in an unmodified state in the cache of one or more nodes (local and/or remote).

Home-Invalid: the data at the home node is out-of-date. The current owner of the data will have the up-to-date version of the data line.

Home-Locked: the sharing list is being updated. No other transactions are allowed to traverse the sharing list until the update has been completed.

The coherence protocol has the notion of an "owning" node for each data line. The owner is the node which has most recently updated the data line.

Cache lines in the processors' second-level caches will be in one of three states:

Invalid: not valid in the cache

Shared: unmodified in the cache, valid for reads only. May also be cached at other nodes.

Exclusive: either modified in the cache (dirty bit is set) or not yet modified but the node has requested exclusive access to the data line. The data line is only held by this cache, so it must be written back to the home node's DRAM if the line is replaced (*i.e.* if it is evicted from the SLC).

C.1 Protocol States

The cache coherence scheme is implemented in the node controller module of the ALITE simulator using two finite state machines: one for home node actions, and the other for client node actions. The state machines include a number of pseudo-states, which are derived from the actual local memory (DRAM) or SLC states plus additional information from the incoming message and (as appropriate) the proxy transit cache and the proxy buffer. The complexity of the state machines is a direct result of the non-atomicity of transactions in the distributed-directory protocol. For example, a client read miss cannot result in an atomic transaction, because there will be at least a `read-request` message to the home node and a `take-shared` message back from the home node, during which time other messages could arrive at the client node controller which affect the SLC cache line.

The implementation of the protocol in the Sun S3.mp prototype provides valuable insights about the complexity of verifying cache coherence protocols for distributed directory schemes [106]. The cache-coherence protocol for that system was verified by a combination of modelling and simulation. Due to the maintenance of linked lists by the distributed algorithm and the non-FIFO nature of the interconnect network (*i.e.* messages do not necessarily arrive in order), the complex S3.mp protocol is a challenge for verification. The protocol has roughly 30 stable/transient cache states and 20 memory states: these states represent branches in the microprogram implementation of the protocol. The number of states is far more than is typically needed for protocols using a central directory because the non-atomicity of transactions means that transient states are needed to represent the intermediate states which occur as the distributed sharing list is being amended.

The home and client state machines used in ALITE are illustrated in Figure C.1 and Figure C.2. Most of the state transitions are triggered by incoming messages, but the figures also show where local read or write misses cause state changes. For the sake of clarity, the Proxy-Buffer-Valid and Proxy-Buffer-Pending-Invalid states have been omitted from Figure C.2 (and all the state transitions associated with the separate proxy buffer have likewise been omitted). They are documented in detail in Section 7.3 of this thesis.

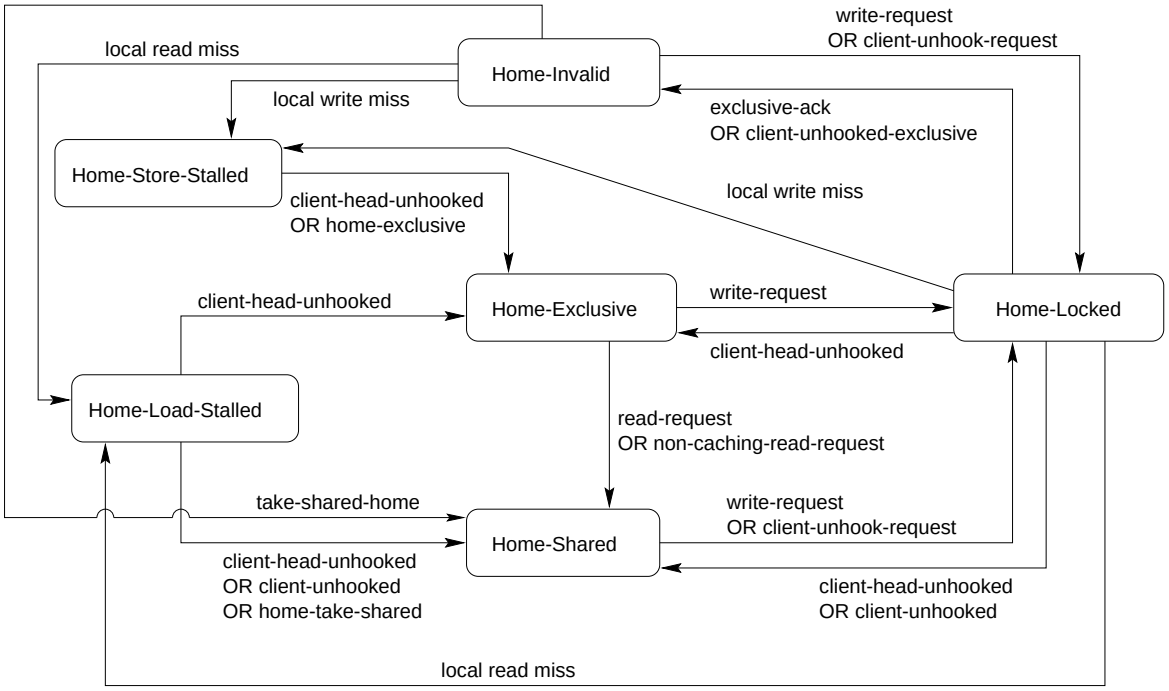


Figure C.1: Node controller state transitions for home node actions

The states are as follows:

Home-Exclusive: The home node has the only copy of the data line, *i.e.* it is currently the owner.

Home-Invalid: The up-to-date data is held at another node (the owning client, who has obtained exclusive access to write to the data line).

Home-Load-Stalled: This indicates that a local read miss found that its DRAM directory's state was Home-Invalid for this data line. A `home-read-request` message has been sent to the current owner of the data line.

Home-Locked: The sharing list for this data line has been locked in response to a `write-request` or a `client-unhook-request` message. The state will continue until the current transaction has finished amending the sharing list, *i.e.* the lock protects the sharing list until it is no longer unstable.

Home-Shared: The home node has a shared copy of the data line.

Home-Store-Stalled: This indicates that a local write miss found that its DRAM directory's state was Home-Invalid for this data line. A `home-invalidate` message has been sent along the sharing list.

Client-Exclusive: This node is the owner of the data line and has the most up-to-date version.

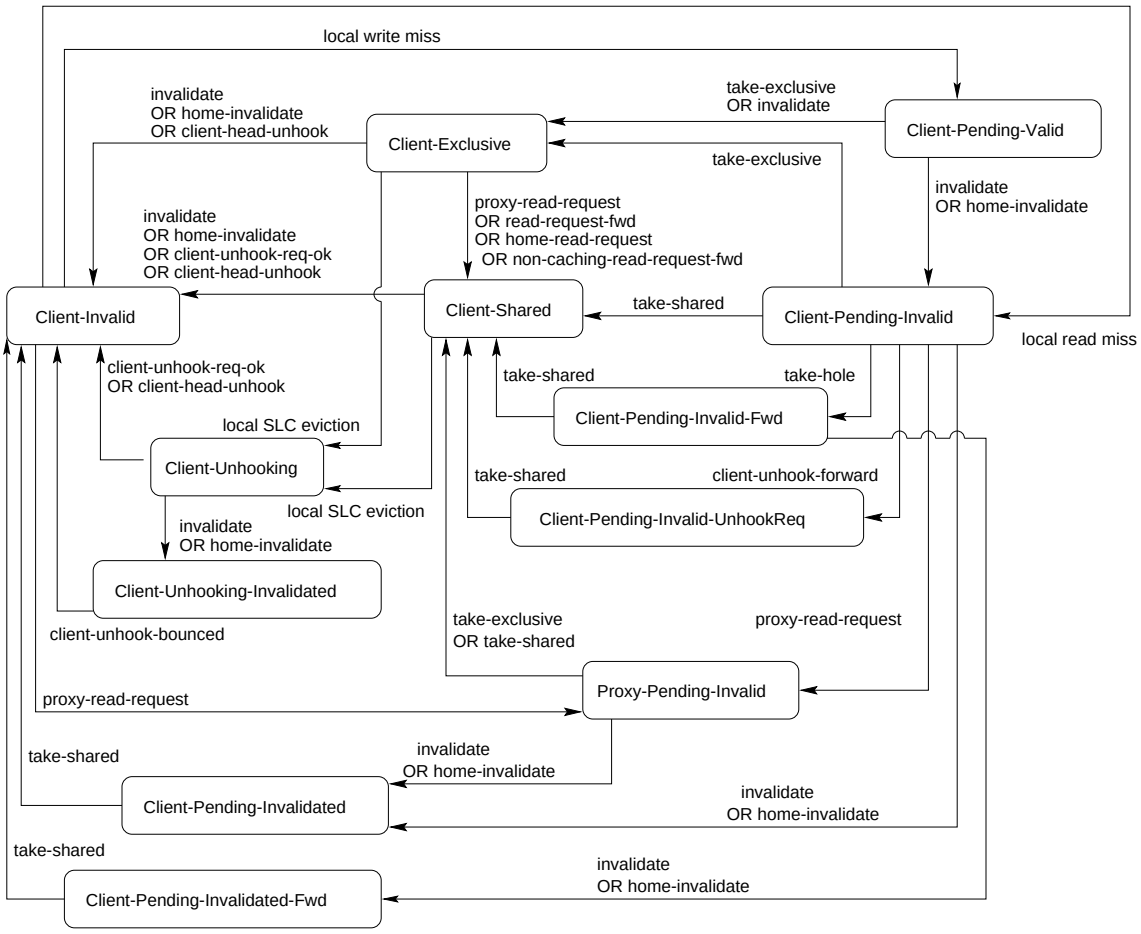


Figure C.2: Node controller state transitions for client node actions

Client-Invalid: There is no valid data held in the SLC cache line.

Client-Pending-Invalid: A `read-request` message has been sent for this data line, *i.e.* the cache line has been reserved, and is waiting for the data to arrive in a `take-shared` message.

Client-Pending-Invalid-Fwd: The client waiting for a `take-shared` message (in state Client-Pending-Invalid) has received a `take-hole` message from the proxy node. This state indicates that the node is not at the end of the proxy pending chain.

Client-Pending-Invalid-UnhookReq: An unhook request beat the `take-shared` message back to this node, *i.e.* the node controller will have to handle forwarding the unhook down the sharing list once the `take-shared` message has arrived and this node is part of the sharing list.

Client-Pending-Invalidated: An invalidation request beat the `take-shared` message back to this node (which was in state Client-Pending-Invalid). The node controller will have to handle the invalidation once the `take-shared` message arrives and this node is part of the sharing list.

Client-Pending-Invalidated-Fwd: An invalidation request has arrived before the `take-shared` message when the node was in state Client-Pending-Invalid-Fwd (*i.e.* it was on the proxy pending chain). The node controller will have to handle the invalidation once the `take-shared` message arrives and this node is part of the sharing list.

Client-Pending-Valid: A local write miss is being processed (a `write-request` message has been sent to the home node). It should be noted that the receipt of an invalidate message can lead to various transitions from this state, depending on whether the requester of the invalidation is this node, and its position on the sharing list.

Client-Shared: A clean copy of the data line is held in the SLC.

Client-Unhooking: The data in the cache line is being unhooked from its sharing list.

Client-Unhooking-Invalidated: An invalidation request has arrived at a line which has requested unhooking. The invalidation is held until the `client-unhook-bounced` message arrives (the sharing list will be locked at the home), at which point the invalidation can proceed.

Proxy-Buffer-Valid: There is a valid copy of the data line in the local proxy buffer. For the sake of clarity, this state is not shown in Figure C.2, but its use is documented in detail in Section 7.3.

Proxy-Buffer-Pending-Invalid: There is a valid entry in the local proxy buffer, but it is currently being evicted under the FIFO replacement policy. For the sake of clarity, this state is not shown in Figure C.2, but its use is documented in detail in Section 7.3.

Proxy-Pending-Invalid: the node has sent a `read-request` on behalf of a client *i.e.* there is an entry in the proxy transit cache.

C.2 Message Categories

This section contains a full list, with descriptions, of the message types used in the simulated protocol. The messages are grouped into the categories which are used to report the results in Chapters 4, 5, 6, and 7. The read, write, and unhook messages are used to implement the Stanford distributed-directory protocol [134]. The proxy, non-caching read, and proxy buffer unhook messages were added to support the various proxy strategies.

C.2.1 Read Messages

bounced-read-request: when a home node receives a `read-request` message for a locked directory entry, the "bounced" message is sent in response.

buffer-bounced-read-request: when the system is simulating a simple finite incoming message buffer, this buffer-bounce message is sent when a `read-request` arrives at a home node and there are already eight or more messages in the incoming message buffer.

home-read-request: the home node requests a new copy of the data line from the current owner (because of a read miss by the home node's CPU).

home-take-shared: the owner node sends an up-to-date copy of the data line to the home node (in response to a `home-read-request`).

read-request: this message is sent either (1) by a client node to the home node as a result of a local read miss, or (2) by a proxy node to the home node in response to a `proxy-read-request` from a client node.

read-request-fwd: the home node forwards a read request on to the current owner when the home node directory state is Home-Invalid.

take-shared: a new copy of the data line is sent to the client (or proxy).

take-shared-home: a new copy of the data line is sent to the home node by the owner (when it changes state from Client-Exclusive to Client-Shared in response to a `read-request-fwd`).

C.2.2 Write Messages

bounced-write-request: the directory entry at the home node is locked, so the `write-request` is "bounced" back to the client.

exclusive-ack: the new owner (which now has exclusive access to the data line) sends this acknowledgement to the home node to unlock the home's directory entry. The home node changes its directory state to Home-Invalid.

home-exclusive: this message is sent to the home node when a `home-invalidate` has reached the end of the sharing list (*i.e.* all entries have now been removed from the sharing list).

home-invalidate: the home node's CPU wants to write to the data line, so the sharing list has to be invalidated. This is similar to the `invalidate` message, but the processing is initiated by the home node.

invalidate: the home node sends this message along the sharing list in response to a `write-request` message. All clients (except the sender of the `write-request`) are to be removed from the sharing list.

take-exclusive: the client requiring write access is informed that it is now the owner by the last node to be invalidated (*i.e.* by the last node on the old sharing list). This message is not needed when the requesting node is itself the last node on the old sharing list.

write-request: a client node sends a `write-request` to the home node when it needs to obtain exclusive access to change a data item. This causes the home node to lock the sharing list and send an `invalidate` message down the sharing list.

C.2.3 Unhook Messages

client-head-unhook: the client which sent a `client-unhook-request` is at the head of the sharing list (*i.e.* it is the first entry). The home node locks the sharing list and sends this message back to the client. The client then knows it is the head of the list, and can simply unhook itself and send a `client-head-unhooked` message back to the home node.

client-head-unhooked: the client at the head of the list has unhooked itself, and sends this message to the home with the pointer to the tail of the sharing list. The home node can then update the directory entry to point to the tail of the sharing list, and unlock the directory entry.

client-unhook-bounced: the home node entry is locked, so the `client-unhook-request` is sent back to the client.

client-unhook-forward: a `client-unhook-request` is passed along the sharing list using this message, until it reaches the node *before* the client (*i.e.* the node which points to the requesting client). At this point a `client-unhook-req-ok` message is sent to the unhook requester.

client-unhook-not-found: the client which requested to be unhooked was not found on the sharing list. This message is sent by the node at the end of the sharing list to the node which requested to be unhooked. This could happen when an invalidation reached the unhooking client *before* the unhook, in which case the protocol handles the receipt of the `client-unhook-not-found` message in state `Client-Unhooking-Invalidated`. If the client which requested the unhook is in any other state, then the protocol signals that an error has occurred.

client-unhook-ptr: sent in response to a `client-unhook-req-ok`, this message contains a pointer to the tail of the sharing list (*i.e.* from after the unhook requester). The receiving node modifies its SLC entry to point to the tail of the sharing list, in effect removing the unhook requester from the sharing list, and then sends a `client-unhooked` message to the home node.

client-unhook-request: a client node sends this message to the home node when it wants to be taken off the sharing list (*i.e.* it no longer wants a copy of the data line).

client-unhook-req-ok: this message is sent to the requester by the preceding node in the sharing list. It prompts the requesting node to send a `client-unhook-ptr` message back to the preceding node, and remove the data line from its local SLC.

client-unhooked: this message indicates to the home node that the modifications to the sharing list are complete. The directory entry is unlocked, and the state is set back to Home-Shared.

client-unhooked-exclusive: this message handles the situation where an unhook request is "beaten" by an `invalidate` request from another node. It is possible for a `client-unhook-forward` request to arrive at the new owner node (*i.e.* the node which requested exclusive access to the data line). When this happens, a `client-unhooked-exclusive` message is sent to the home node to prompt the home node to unlock the directory entry (which was locked by the `client-unhook-request`). In addition, a `client-unhook-not-found` message is sent to the node which requested to be unhooked.

C.2.4 Proxy Messages

proxy-bounced-read-request: this message is sent in response to a `proxy-read-request` when the proxy node is in the process of obtaining another data line which conflicts with the new request in the SLC, or there is no room in the proxy transit cache. The client will re-send the `proxy-read-request` until it has bounced 10 times, at which point the client will revert to sending a `read-request` to the home node.

proxy-read-request: this message is sent by a client to a proxy node once the client has decided that it will use a proxy to service a read request.

take-hole: used to build the pending chain of clients. When a `proxy-read-request` arrives from a client, and the proxy is already in the process of obtaining the data for another client, this message is sent to the old "finish" of the pending chain to link it to the latest client (that latest client then becomes the new "finish"). The use of the `take-hole` message is illustrated in Figure 4.5(b).

C.2.5 Non-Caching Proxy Messages

bounced-non-caching-proxy-request: the matching directory entry at the home node is locked (because the sharing list is being updated) so the `non-caching-proxy-request` is bounced back to the proxy node.

buffer-bounced-non-caching-proxy-request: when the system is simulating a simple finite incoming message buffer, this buffer-bounce message is sent when a `non-caching-proxy-request` arrives at a home node and there are already eight or more messages in the incoming message buffer.

non-caching-proxy-request: this message is sent by a proxy node to a home node, in response to a `proxy-read-request` from a client, when there is no copy of the data in the local SLC and the non-caching proxy policy is in use. The client node's id is sent as part of the message.

non-caching-proxy-request-fwd: when the home node receives a `non-caching-proxy-request` and the matching directory entry has status Home-Invalid, then this forwarding message is sent on to the current owner node.

C.2.6 Proxy Buffer Unhook Messages

proxy-buffer-head-unhooked: the proxy node (*i.e.* a node where the data line copy is in the proxy buffer rather than the SLC) which requested to be unhooked from the sharing chain was at the head of the sharing list. In response to a `proxy-buffer-unhook-ok` message from the home node, it sends the `proxy-buffer-head-unhooked` message to the home node to confirm that it has deleted the proxy buffer copy, along with the pointer to the tail of the sharing list. The home node will then update the directory entry for the data line to use the "tail" pointer, and will also unlock the entry.

proxy-buffer-unhook-bounced: when a home node receives a `proxy-buffer-unhook-request` message for a locked directory entry, the "bounced" message is sent in response.

proxy-buffer-unhook-cancelled: the node which requested the unhook of its proxy buffer entry has subsequently experienced a local read miss for the same data line before the `proxy-buffer-unhook-ok` message arrived back at the requesting node. The proxy buffer entry has been "promoted" to the SLC, and so the node needs to remain on the sharing list. The `proxy-buffer-unhook-cancelled` is sent to the home node in response to the `proxy-buffer-unhook-ok` message, and it indicates that the unhook transaction has been aborted and the sharing list should be unlocked by the home node.

proxy-buffer-unhook-forward: this message is sent along the sharing list until it reaches the node *before* the (proxy) node which requested the unhook. When that happens a `proxy-buffer-unhook-ok` message is sent to the unhook's requester.

proxy-buffer-unhook-not-found: the proxy which requested to be unhooked could not be found on the sharing list. This message is sent to the requester by the node at the end of the sharing list (and it also sends a `proxy-buffer-unhooked` message to the home node to unlock the sharing list).

proxy-buffer-unhook-ok: this message is sent by the preceding node in the sharing chain to the node which requested the unhook. It prompts the proxy node to remove the matching entry from its proxy buffer, and then send back a `proxy-buffer-unhook-ptr` message (or `proxy-buffer-head-unhooked` if appropriate) containing a pointer to the rest of the sharing list.

proxy-buffer-unhook-ptr: sent in response to a `proxy-buffer-unhook-ok`, this message contains a pointer to the remainder of the sharing list. The receiving node modifies its SLC (or proxy buffer) entry to use this pointer, in effect removing the unhook's requester from the sharing list. A `proxy-buffer-unhooked` message is then sent to the home node.

proxy-buffer-unhook-request: this message is sent to the home node when a proxy needs to evict an entry from its proxy buffer. This will initiate the actions needed to remove the node from the sharing list.

proxy-buffer-unhooked: this message notifies the home node that the unhook modifications to the sharing list have finished. The directory entry is then unlocked.

Appendix D

Order Notation

The order notation used in this thesis for expressing asymptotic magnitudes is that proposed by Knuth [72]. Assume that f and g are functions over the domain of the natural numbers. Then $O(f(n))$ is the set of all $g(n)$ such that there exist positive constants c and n_0 so that $|g(n)| \leq c\, f(n)$ for all $n \geq n_0$. Informally this is the set of functions which "grow no faster" than f.

Similarly, $\Omega(f(n))$ is the set of all $g(n)$ such that there exist positive constants c and n_0 so that $|g(n)| \geq c\, f(n)$ for all $n \geq n_0$. This is the set of functions which "grow at least as fast" as f.

Finally, $\Theta(f(n))$ are those functions which "grow at the same rate" as f. This is the set of all $g(n)$ such that $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.

Bibliography

[1] Gheith A. Abandah and Edward S. Davidson. Effects of architectural and technological advances on the HP/Convex Exemplar's memory and communication performance. In *the Twenty-Fifth Annual International Symposium on Computer Architecture, Barcelona*, pages 318–329, June 1998.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Krantz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. *Twenty-Second Annual International Symposium on Computer Architecture, Santa Margherita Ligure, in Computer Architecture News*, 23(2):2–13, June 1995.

[4] Anant Agarwal and Anoop Gupta. Memory reference characteristics of multiprocessor applications under MACH. *Conference on Measurement and Modeling of Computer Systems, Santa Fe, in Performance Evaluation Review*, 16(1):215–225, May 1988.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *International Conference on Supercomputing, Amsterdam, in Computer Architecture News*, 18(3):1–6, September 1990.

[6] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[7] Craig Anderson and Anna R. Karlin. Two adaptive hybrid cache coherency protocols. In *the Second Annual Symposium on High Performance Computer Architecture, San Jose, California*, pages 303–313, February 1996.

[8] Thomas E. Anderson, David E. Culler, and David A. Patterson. The case for networks of workstations: NOW. *IEEE Micro*, 15(1):54–64, February 1995.

[9] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[10] David H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[11] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[12] Andrew J. Bennett, Anthony J. Field, and Peter G. Harrison. Development and validation of an analytical model of a distributed cache coherency protocol. *Performance Evaluation*, 27&28:541–563, October 1996.

[13] Andrew J. Bennett, Paul H. J. Kelly, Jacob G. Refstrup, and Sarah A. M. Talbot. Using proxies to reduce cache controller contention in large shared-memory multiprocessors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par 96 - Second European Conference on Parallel Processing, Lyon*, volume 1124 of *Lecture Notes in Computer Science*, pages 445–452. Springer-Verlag, August 1996.

[14] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. *Seventeenth Annual International Symposium on Computer Architecture, Seattle, in Computer Architecture News*, 18(2):125–134, June 1990.

[15] Dileep P. Bhandarkar. *Alpha Implementations and Architecture: complete reference and guide.* Digital Press, 1996.

[16] Ricardo Bianchini and Thomas J. LeBlanc. Eager combining: a coherency protocol for increasing effective network and memory bandwidth in shared-memory multiprocessors. In *the Sixth IEEE Symposium on Parallel and Distributed Processing, Dallas*, pages 204–213, October 1994.

[17] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectiveness of decoupling. In *the Seventh International Conference on Supercomputing, Tokyo*, pages 47–56, July 1993.

[18] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi, and Robert A. Shillner. Design choices in the SHRIMP system: an empirical study. In *the Twenty-Fifth Annual International Symposium on Computer Architecture, Barcelona*, pages 330–341, June 1998.

[19] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, 1987.

[20] Mats Brorsson. SM-prof: a tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *Proceedings of the ACM SIGMETRICS and Performance '95*, pages 178–187, May 1995.

[21] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[22] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? *Sixth International Conference on Architectural*

Support for Programming Languages and Operating Systems, San Jose, in SIGPLAN Notices, 29(11):61–73, October 1994.

[23] Alan Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, February 1998.

[24] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: a hardware/software approach*. Morgan Kaufman, 1998.

[25] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: a practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[26] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[27] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

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

[29] Data General Corporation. *NUMALiiNE Technology: the foundation for the AV 25000 server*. Technical white paper, available online as `http://www.dg.com/aviion/html/av_25000_foundation.html`. 1998.

[30] Peter J. Denning. On modeling program behaviour. In *Proceedings of the Spring Joint Computer Conference, Atlantic City*, pages 937–944, May 1972.

[31] J. Dongarra, O. Brewer, J.A. Kohl, and S. Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[32] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. *Thirteenth Annual International Symposium on Computer Architecture, Tokyo, in Computer Architecture News*, 14(2):434–442, June 1986.

[33] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[34] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. *Fifteenth Annual International Symposium on Computer Architecture, Honolulu, in Computer Architecture News*, 16(2):373–382, May 1988.

[35] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-machine multicomputer. In *Annual International Symposium on Microarchitecture, Ann Arbor*, pages 146–156. IEEE, December 1995.

[36] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[37] Arno Formella, Thomas Grün, and Christoph W. Keßler. The SB-PRAM: concept, design and construction. In *the Third Working Conference on Massively Parallel Programming Models MPPM-97, London*, pages 163–172. IEEE, November 1997.

[38] Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segal. *Parallel Processing: the Cm* experience*. Digital Press, 1987.

[39] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to 'Memory consistency and event ordering in scalable shared-memory multiprocessors'. Technical Report CSL-TR-93-568, Stanford University, Computer Systems Laboratory, April 1993.

[40] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Seventeenth Annual International Symposium on Computer Architecture, Seattle, in Computer Architecture News*, 18(2):15–26, June 1990.

[41] Stein Gjessing, David B. Gustavson, James R. Goodman, David V. James, and Ernst H. Kristiansen. The SCI cache coherence protocol. In Michel Dubois and Shreekant S. Thakkar, editors, *Workshop on Scalable Shared Memory Multiprocessors, Toronto*, pages 219–237. Kluwer Academic Publishers, May 1990.

[42] A.J. Goldberg and J.L. Hennessy. Mtool: an integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, Jan 1993.

[43] Stephen R. Goldschmidt. *Simulation of multiprocessors: accuracy and performance*. PhD thesis, Department of Electrical Engineering, Stanford University, June 1993.

[44] James R. Goodman. Using cache memory to reduce processor-memory traffic. *Tenth Annual International Symposium on Computer Architecture, Stockholm, in Computer Architecture News*, 11(3):124–131, June 1983.

[45] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[46] S.L. Graham, P.B. Kessler, and M.K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, Aug 1983.

[47] Thomas Grün and Mark A. Hillebrand. NAS integer sort on multi-threaded shared memory machines. In David Pritchard and Jeff Reeve, editors, *Euro-Par 98 - Fourth European Conference On Parallel Processing, Southampton*, volume 1470 of *Lecture Notes in Computer Science*, pages 999–1009. Springer-Verlag, September 1998.

[48] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[49] Jim Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.

[50] Seif Haridi and Erik Hagersten. The cache coherence protocol of the Data Diffusion Machine. In E. Odijk, M. Rem, and J.-C Syre, editors, *PARLE 89 Parallel Architectures and Languages Europe, Eindhoven*, volume 365 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, June 1989.

[51] Mark Heinrich, Vijayaraghavan Soundararajan, John Hennessy, and Anoop Gupta. A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols. *IEEE Transactions on Computing*, 48(2), February 1999.

[52] Hermann Hellwagner. On the practical efficiency of randomized shared memory. In Luc Bougé et al, editor, *Parallel Processing: CONPAR 92 - VAPP* V, *Lyon*, volume 634 of *Lecture Notes in Computer Science*, pages 429–440. Springer-Verlag, September 1992.

[53] John Hennessy, Anoop Gupta, and Mark Heinrich. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3):418–429, March 1999.

[54] John L. Hennessy and David A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufmann, second edition, 1996.

[55] Mark D. Hill. A case for direct mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[56] Mark D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, August 1998.

[57] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[58] Chris Holt and Jaswinder Pal Singh. Hierarchical N-body methods on shared address space multiprocessors. In *the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 313–318, February 1995.

[59] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, July 1995.

[60] Dongming Jiang and Jaswinder Pal Singh. A methodology and evaluation of the SGI Origin2000. In *SIGMETRICS and Performance '98*, pages 171–181, June 1998.

[61] Ross E. Johnson. *Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors*. PhD thesis, Computer Science Department, University of Wisconsin-Madison, February 1993.

[62] Teresa L. Johnson and Wen Mei Hwu. Run-time adaptive cache hierarchy management via reference analysis. *Twenty-Fourth Annual International Symposium on Computer Architecture, Denver, in Computer Architecture News*, 25(2):315–326, May 1997.

[63] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Seventeenth Annual International Symposium on Computer Architecture, Seattle, in Computer Architecture News*, 18(2):364–373, May 1990.

[64] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing spatial locality via data layout optimizations. In David Pritchard and Jeff Reeve, editors, *Euro-Par 98 - Fourth European Conference On Parallel Processing, Southampton*, volume 1470 of *Lecture Notes in Computer Science*, pages 422–434. Springer-Verlag, September 1998.

[65] Gerry Kane and Hewlett Packard. *PA-RISC 2.0 Architecture*. Prentice Hall, 1996.

[66] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *the Thirteenth ACM Symposium on Operating Systems Principles, Pacific Grove, CA*, pages 41–55, October 1991.

[67] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of theoretical computer science Vol. A: Algorithms and Complexity*, chapter 17, pages 871–941. Elsevier, 1990.

[68] Stefanos Kaxiras. *Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface*. IEEE Draft 0.36 Standard 1596.2-1996, available online from `ftp://ftp.cs.wisc.edu/galileo/kaxiras/stdbook.ps`, November 1996.

[69] Stefanos Kaxiras, Stein Gjessing, and James R. Goodman. A study of three dynamic approaches to handle widely shared data in shared-memory multiprocessors. In *the Twelfth International Conference on Supercomputing, Melbourne*, pages 457–464, July 1998.

[70] Stefanos Kaxiras and James R. Goodman. The GLOW cache coherence protocol extensions for widely shared data. In *the Tenth International Conference on Supercomputing, Philadelphia*, pages 35–43, May 1996.

[71] Jörg Keller. Fast rehashing in PRAM emulations. *Theoretical Computer Science*, 155(2):349–363, March 1996.

[72] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April-June 1976.

[73] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolas Hardavellas, Michael Cierniak, Srinivasan Parthasarathy, Wagner Meira, Jr., Sandhya Dwarkadas, and Michael L. Scott. VM-based shared memory on low-latency, remote memory access networks. *Twenty-Fourth Annual International Symposium on Computer Architecture, Denver, in Computer Architecture News*, 25(2):157–169, May 1997.

[74] Leonidas I. Kontothanassis and Michael L. Scott. Software cache coherence for large scale multiprocessors. In *the First Annual Symposium on High Performance Computer Architecture, Rayleigh, North Carolina*, pages 286–295, January 1995.

[75] Leonidas I. Kontothanassis, Michael L. Scott, and Ricardo Bianchini. Lazy release consistency for hardware-coherent multiprocessors. Technical Report TR547, University of Rochester, Department of Computer Science, December 1994.

[76] David Koufaty and Josep Torrellas. Comparing data forwarding and prefetching for communication-induced misses in shared-memory MPs. In *the Twelfth International Conference on Supercomputing, Melbourne*, pages 53–59, July 1998.

[77] David Kroft. Lockup-free instruction fetch/prefetch cache organisation. *Eighth Annual International Symposium on Computer Architecture, Minneapolis, in Computer Architecture News*, 9(3):81–87, May 1981.

[78] David J. Kuck. *High Performance Computing: challenges for future systems.* Oxford University Press, 1996.

[79] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: design and analysis of algorithms.* Benjamin Cummings, 1994.

[80] Jeffrey Kuskin. *The FLASH Multiprocessor: designing a flexible and scalable system.* PhD thesis, Computer Systems Laboratory, Stanford University, November 1997. Also available as technical report CSL-TR-97-744.

[81] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. *Twenty-first Annual International Symposium on Computer Architecture, Chicago, in Computer Architecture News*, 22(2):302–313, April 1994.

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

[83] Richard P. LaRowe Jr and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[84] James R. Larus, Satish Chandra, and David A. Wood. CICO: a shared-memory programming performance model. In Jeanne Ferrante and Tony Hey, editors, *Portability and Performance for Parallel Processors.* John Wiley & Sons, 1993.

[85] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. *Twenty-Fourth Annual International Symposium on Computer Architecture, Denver, in Computer Architecture News*, 25(2):241–251, May 1997.

[86] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown. Large-scale parallel programming: experience with the BBN Butterfly parallel processor. *ACM SIGPLAN Notices*, 23(9):161–172, September 1988.

[87] Charles Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33:145–158, 1996.

[88] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Seventeenth Annual International Symposium on Computer Architecture, Seattle, in Computer Architecture News*, 18(2):148–159, May 1990.

[89] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.

[90] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[91] David J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *Computing Surveys*, 25(3):303–338, September 1993.

[92] Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. *Twenty-Third Annual International Symposium on Computer Architecture, Philadelphia, in Computer Architecture News*, 24(2):308–317, May 1996.

[93] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *the International Conference on Parallel Processing, Pennsylvania State University*, volume 1 (Architecture), pages 303–310, August 1988.

[94] M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, April 1995.

[95] Kathryn S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, August 1998.

[96] Maged Michael and Ashwini Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *the Fifth Annual Symposium on High Performance Computer Architecture, Orlando*, pages 142–151, January 1999.

[97] Maged M. Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. *Twenty-Fourth Annual International Symposium on Computer Architecture, Denver, in Computer Architecture News*, 25(2):219–228, June 1997.

[98] Shubhendu S. Mukherjee and Mark D. Hill. Using prediction to accelerate coherence protocols. In *the Twenty-Fifth Annual International Symposium on Computer Architecture, Barcelona*, pages 179–190, June 1998.

[99] Henk L. Muller, Paul W. A. Stallard, and David H. D. Warren. Implementing the data diffusion machine using crossbar routers. In *the Tenth International Parallel Processing Symposium, Honolulu*, pages 152–158, April 1996.

[100] Jim Nilsson, Fredrik Dahlgren, Magnus Karlsson, Peter Magnusson, and Per Stenström. Computer system evaluation with commercial workloads. In *IASTED International Conference on Modelling and Simulation, Pittsburgh*, pages 293–297, May 1998.

[101] Andreas Nowatzyk, Günes Aybay, Michael Browne, Edmund Kelly, Michael Parkin, Bill Radke, and Sanjay Vishin. Exploiting parallelism in cache coherency protocol engines. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Euro-Par 95 - First European Conference On Parallel Processing, Stockholm*, volume 966 of *Lecture Notes in Computer Science*, pages 269–286. Springer-Verlag, August 1995.

[102] Andreas Nowatzyk, Günes Aybay, Michael Browne, Edmund Kelly, Michael Parkin, Bill Radke, and Sanjay Vishin. The S3.mp scalable shared memory multiprocessor. In *the International Conference on Parallel Processing, Vol. 1*, pages 1–10, August 1995.

[103] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *Eleventh International Symposium on Computer Architecture, Ann Arbor, June, in Computer Architecture News*, 12(3):348–354, June 1984.

[104] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfekder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): introduction and architecture. In *the International Conference on Parallel Processing, Pennsylvania State University*, pages 764–771, August 1985.

[105] Gregory F. Pfister and V. Alan Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[106] Fong Pong, Michael Browne, Günes Aybay, Andreas Nowatzyk, and Michel Dubois. Design verification of the S3.mp cache-coherent shared-memory system. *IEEE Transactions on Computers*, 47(1):135–140, January 1998.

[107] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose*, October 1998.

[108] Alain Raynaud, Zheng Zhang, and Josep Torrellas. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In *the Second Annual Symposium on High Performance Computer Architecture, San Jose*, pages 323–334, February 1996.

[109] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *Performance Evaluation*, 21(1):48–60, June 1993.

[110] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: user-level shared memory. *Twenty-First Annual International Symposium on Computer Architecture, Chicago, in Computer Architecture News*, 22(2):325–336, April 1994.

[111] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall: the case for processor/memory integration. *Twenty-Third Annual International Symposium on Computer Architecture, Philadelphia, in Computer Architecture News*, 24(2):90–101, May 1996.

[112] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An argument for simple COMA. In *the First Annual Symposium on High Performance Computer Architecture, Rayleigh, North Carolina*, pages 276–285, January 1995.

[113] Christoph Scheurich and Michel Dubois. Dynamic page migration in multiprocessors with distributed global memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.

[114] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Mass, in SIGPLAN Notices*, 31(9):26–36, September 1996.

[115] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A parallel adaptive fast multipole method. In *Supercomputing 93, Portland*, pages 54–65, November 1993.

[116] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[117] Jonas Skeppstedt and Per Stenström. Simple compiler algorithms to reduce ownership overhead in cache coherence protocols. *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, in SIGPLAN Notices*, 29(11):286–296, October 1994.

[118] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[119] James E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[120] Vijayaraghavan Soundararajan, Mark Heinrich, Ben Verghese, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors. In *the Twenty-Fifth Annual International Symposium on Computer Architecture, Barcelona*, pages 342–355, June 1998.

[121] Splash-2 Characterization Database. Available on-line from `http://liber.stanford.edu/~torrie/Splash2/Results.html`.

[122] Splash-2 Suite Distribution. Available on-line from `ftp://www-flash.stanford.edu/pub/splash2/`.

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

[124] Per Stenström, Erik Hagersten, David J. Lilja, Margaret Martonosi, and Madan Venugopal. Trends in shared memory multiprocessing. *IEEE Computer*, 30(12):44–50, December 1997.

[125] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. *Nineteenth International Symposium on Computer Architecture, Gold Coast, in Computer Architecture News*, 20(2):80–91, May 1992.

[126] Thomas Sterling. Beowulf-class clustered computing: harnessing the power of parallelism in a pile of PCs. In *the Third Annual Conference on Genetic Programming, University of Wisconsin-Madison*, 1998.

[127] Dimitrios Stiliadis and Anujan Varma. Selective victim caching: a method to improve the performance of direct-mapped caches. In *the Twenty-Seventh Hawaii International Conference on Systems Sciences, Volume 1*, pages 412–421, January 1994.

[128] Harold S. Stone. *High-Performance Computer Architecture*. Addison Wesley, third edition, 1993.

[129] Sarah A. M. Talbot, Andrew J. Bennett, and Paul H. J. Kelly. Cautious, machine-independent performance tuning for shared-memory multiprocessors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par 96 - Second European Conference on Parallel Processing, Lyon*, volume 1123 of *Lecture Notes in Computer Science*, pages 106–113. Springer-Verlag, August 1996.

[130] Sarah A. M. Talbot and Paul H. J. Kelly. Reactive proxies: a flexible protocol extension to reduce ccNUMA node controller contention. In David Pritchard and Jeff Reeve, editors, *Euro-Par 98 - Fourth European Conference On Parallel Processing, Southampton*, volume 1470 of *Lecture Notes in Computer Science*, pages 1062–1075. Springer-Verlag, September 1998.

[131] Sarah A. M. Talbot and Paul H. J. Kelly. Stable performance for cc-NUMA using first-touch page placement and reactive proxies. In Jonathan Schaeffer, editor, *High Performance Computing Systems and Applications*, pages 251–266. Kluwer Academic Publishers, May 1998.

[132] B. A. Tanyi. *Iterative Solution of the Incompressible Navier-Stokes Equations on a Distributed Memory Parallel Computer*. PhD thesis, University of Manchester Institute of Science and Technology, 1993.

[133] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite. Firefly: a multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[134] Manu Thapar and Bruce Delagi. Stanford distributed-directory protocol. *IEEE Computer*, 23(6):78–80, June 1990.

[135] Manu Thapar, Bruce A. Delagi, and Michael J. Flynn. Scalable cache coherence for shared memory multiprocessors. In Hans P. Zima, editor, *First Annual Conference*

on Parallel Computation, Salzburg, volume 591 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, September 1991.

[136] J. Torrellas, M.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[137] Leslie G. Valiant. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Transactions on Computers*, C-32(8):861–863, August 1983.

[138] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[139] Steven P. VanderWiel and David J. Lilja. When caches aren't enough: data prefetching techniques. *IEEE Computer*, 30(7):23–30, July 1997.

[140] T. D. Wagner, E. Smirni, A. W. Apon, M. Madhukar, and L. W. Dowdy. The effects of thread placement on the KSR-1. In *the Eighth International Parallel Processing Symposium, Cancun*, pages 618–624, April 1994.

[141] Ian Watson and Alasdair Rawsthorne. Decoupled pre-fetching for distributed shared memory. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, volume 5, pages 252–261, January 1995.

[142] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patters in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston*, pages 243–256, April 1989.

[143] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager sharing for efficient massive parallelism. In *the International Conference on Parallel Processing*, volume II, pages 251–255, August 1992.

[144] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Twenty-Second Annual International Symposium on Computer Architecture, Santa Margherita Ligure, in Computer Architecture News*, 23(2):24–36, June 1995.

[145] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, in SIGPLAN Notices*, 29(11):219–229, October 1994.

[146] Ken Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[147] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *the Fifth Annual Symposium on High Performance Computer Architecture, Orlando*, January 1999.