1 STABLE PERFORMANCE FOR CC-NUMA USING FIRST TOUCH PAGE PLACEMENT AND REACTIVE PROXIES

Sarah A. M. Talbot and Paul H. J. Kelly

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
{samt,phjk}@doc.ic.ac.uk

Abstract: A key problem for shared-memory systems is unpredictable performance. A critical influence on performance is page placement: a poor choice of home node can severely degrade application performance because of the increased latency of accessing remote rather than local data. Two approaches to page placement are the simple policies "first-touch" and "round-robin", but neither of these policies suits all applications. We examine the advantages of each strategy, the problems that can result from a poor choice of placement policy, and how these problems can be alleviated by using proxies. Proxies route remote read requests via intermediate nodes, where combining is used to reduce contention at the home node. Our simulation results indicate that by using reactive proxies with first-touch page placement, performance is always better than using either page placement policy without proxies. These results suggest that the application programmer can obtain stable performance without knowing the underlying implementation of cc-NUMA, and can avoid time-consuming performance tuning.

Keywords: cache coherence protocols, shared-memory, combining, page placement.

1.1 INTRODUCTION

Unpredictable performance anomalies have hampered the acceptance of coherent-cache non-uniform memory access (cc-NUMA) shared-memory architectures. One source of performance problems is the location of shared data: each page of shared data is allocated in distributed memory at a home node, by the

operating system, when it is first accessed. The choice of home node for a page is commonly on a first-touch or round-robin basis, but neither of these policies is suited to all applications, and a poor choice of page placement policy can have a marked effect on the performance of an application.

In this paper we examine the effects of simple page placement, and describe how a technique for reducing read contention in cc-NUMA machines can alleviate problems with inappropriate page placement. In the proxy protocol, we associate a small set of nodes with each location in shared memory, which act as intermediaries for remote read requests. Using execution-driven simulations, we show that using the *reactive* variant of proxies, in conjunction with first-touch page placement, yields performance which is always better than using either of the simple page placement strategies without proxies. This suggests that, by using first-touch page placement with reactive proxies, application programmers can be confident that they will obtain stable performance.

The rest of the paper is structured as follows: page placement is discussed in Section 1.2, and the proxy protocol is explained in Section 1.3. We describe our simulated architecture and experimental design in Section 1.4, and present the results in Section 1.5. The relationship to previous work is discussed in Section 1.6, and in Section 1.7 we summarise our conclusions and give pointers to further work.

1.2 PAGE PLACEMENT POLICIES

In distributed shared memory multiprocessors, shared data is partitioned into virtual memory pages. Each page of shared data is then physically allocated to a home node, by the operating system, as a result of the page fault resulting from the first access to the page. The choice of home node is commonly on a first-touch or round-robin basis. 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, because all the pages end up on the same node (with overflow to its neighbours). It is better to use a first-touch after initialisation policy, where shared memory pages are only permanently allocated to nodes once parallel processing has commenced, and this is the policy we use.

In the round-robin approach, allocation cycles around the nodes, placing a page in turn at each node. This approach distributes the data more evenly around the system. Unfortunately, for applications which have been written with locality in mind, it is likely that few, if any, of the pages accessed by a node will be allocated to it. 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 (*e.g.* on SGI's Origin2000 (Laudon and Lenoski, 1997)).

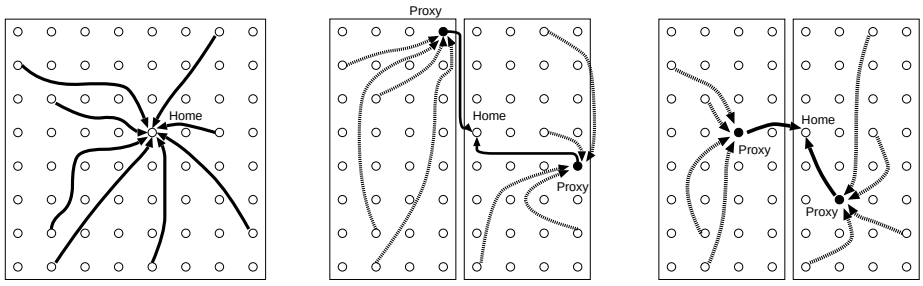
The operating system may also provide facilities for pages to migrate to a new home node, or have copies of the page at other nodes. In recent years, there has been much discussion of dynamic page migration and replication schemes, mainly in the context of distributed virtual shared memory (DVSM), where the software implementation of shared memory on a message-passing distributed memory architecture mandates the movement and/or copying of pages between processing nodes (*e.g.* Munin (Carter et al., 1995)). In contrast, where dynamic paging features are available on cc-NUMA systems, they are usually implemented as options; they are not the default because of the overheads of capturing and acting upon access patterns (Verghese et al., 1996). There is the problem that if two or more processors update data on the same page, then the page may "ping pong" between the new homes, or alternatively the coherence traffic will increase greatly. Even when a page is migrated to the node which uses it most, the average memory access times of other nodes may increase, and reacting too late to the need to migrate or replicate a page may be completely useless, and even costly (LaRowe and Schlatter Ellis, 1991). In addition, even when all the migrations and replications are chosen correctly but they occur in bursts, performance may suffer due to page fault handler, switch, and memory contention.

Given the pitfalls of dynamic page placement, can we get reasonable performance on cc-NUMA machines using simple page placement policies? The best performance for shared memory machines can be obtained by tuning programs based on page placement, but an important principle of shared memory is that it provides application programmers with a simple programming model, where they do not have to worry about the underlying machine, and which leads to more portable programs. We want to keep programmer involvement in page placement to a minimum, but still get reasonable performance.

1.3 PROXIES

Proxying is an extension to standard distributed directory cache coherence protocols, and is designed to reduce node controller contention (Bennett et al., 1996). Normally, a read request message would be sent to the location's home node, based on its physical page address. With proxying, a read request is directed instead to another node controller, which acts as an intermediary. Figure 1.1 illustrates how more than one proxy could serve read requests for a given location, each handling requests from a different processor subset. In this paper, the nodes are split into three proxy clusters (*i.e.* subsets): this split was chosen after experimentation and represents the best balance for our simulated system between contention for the proxy nodes and the degree of combining. The mapping of each data block to its proxy node ensures that requests for successive data blocks are served by different proxies. This balances the queueing of read request messages across the input buffers of all the node controllers.

If the proxy node has a copy of the requested data block in its cache, it replies directly. If not, it requests the value from the home node, allocates the



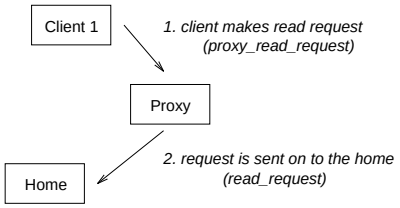
(a) Without proxies

(b) With two proxy clusters (read Line l)

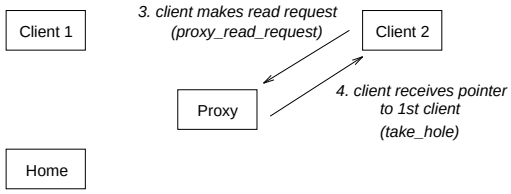
(c) Read next data block ($i.e.$ read Line $l+1$)

Figure 1.1 Contention is reduced by routing reads via a proxy

(a) First request to proxy has to be forwarded to the home node:



(b) Second client request, before data is returned, forms pending chain:



(c) Data is passed to each client on the pending chain:

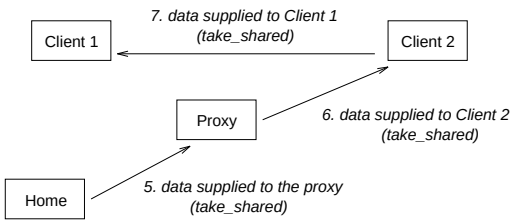


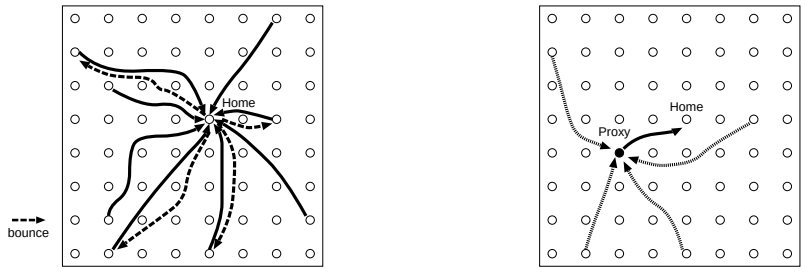
Figure 1.2 Combining of proxy requests

copy in its own cache, and replies to the client. Any requests for a particular block which arrive at a proxy before it has obtained a copy from the home node, are added to a distributed chain of pending requests for that block, and the reply is forwarded down the pending chain, as illustrated in Figure 1.2. The current implementation is slightly unfair in that the first client will be the last to receive the data: we are investigating the tradeoff between increasing the hardware overhead to hold an additional pointer to the tail of each pending proxy chain, and any performance benefits.

Proxying requires a small amount of extra store to be added to each node controller. We need to identify the data blocks for which a node is currently obtaining data as a proxy, and hold the head of each pending proxy chain. The node controller also has to handle the new proxy messages and state changes: we envisage implementing this in software on a programmable node controller.

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, then the overheads incurred by proxies may outweigh any benefits and degrade performance. 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 not mark data structures that would benefit from proxying.

Reactive proxies, which take advantage of the finite buffering of real machines, overcome these problems and do not need application program directives. When a remote read request reaches a full buffer, it will immediately be sent back across the network. When the originator receives the bounced message, and the reactive proxies protocol is in effect, the arrival of the buffer-bounced read request will trigger a proxy read (see Figure 1.3). 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.



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

Figure 1.3 Bounced read requests are retried via proxies

1.4 SIMULATED ARCHITECTURE AND EXPERIMENTAL DESIGN

This paper uses results obtained from our execution-driven simulations of a cc-NUMA system. Each node contains a processor with an integral first-level cache (FLC), a large second-level cache (SLC), memory (DRAM), and a node controller (see Figure 1.4). The node controller receives messages from, and sends messages to, both the network and the processor. The SLC, DRAM, and the node controller are connected using two decoupled buses. This decoupled bus arrangement allows the processor to access the SLC at the same time as the node controller accesses the DRAM. We simulate in detail the contention between the local CPU and the node controller for the buses, between the CPU and incoming messages for the node controller, and the use of the SLC to hold proxy data. Table 1.1 summarises the specifications of the architecture. We simulate a direct-mapped cache, but note that its large size (4 Mb) will have a miss rate roughly equivalent to a 2-way associative cache of half that size (Hennessy and Patterson, 1996).

We simulate a simplified interconnection network, which follows the *LogP* model (Culler et al., 1993). We have parameterised the network and node controller as follows:

- L: the latency experienced in each communication event: 10 cycles for long messages (which include 64 bytes of data, *i.e.* one cache line), and 5 cycles for all other messages.
- o: the occupancy of the node controller. We have adapted the LogP model to recognise the importance of the *occupancy* of a node controller, rather than just the overhead of sending and receiving messages (Holt et al., 1995). Simulated in more detail (see Table 1.2).
- g: the gap between successive sends or receives by a processor: 5 cycles.
- P: the number of nodes: set to 64.

We limit our finite length input message buffers to eight read requests. There can be more messages in an input buffer, but once the queue length has risen above eight, all read requests will be bounced back to the sender until the queue length has fallen below the limit. This is done because we are interested in the effect of finite buffering on read requests rather than all messages, and we wished to be certain that all transactions would complete in our protocol. The queue length of $\sqrt{P}$, where P is the number of processing nodes, is an arbitrary but reasonable limit, and was chosen to reflect the limitations in queue length that one would expect in large cc-NUMA configurations.

Each cache line has a home node (at page level) which: either holds a valid copy of the line (in SLC and/or DRAM), or knows the identity of a node which does have a valid copy (the owner); has guaranteed space in DRAM for the line; and holds directory information for the line (head and state of the sharing list).

The benchmarks and their parameters are summarised in Table 1.3. GE is a simple Gaussian elimination program, similar to that used to study eager combining (Bianchini and LeBlanc, 1994). We chose this benchmark because it is an example of widely-shared data, and should benefit from using proxies, but

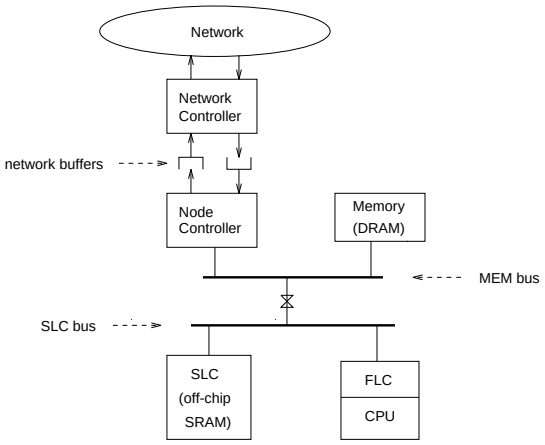


Figure 1.4 The architecture of a node

Table 1.1 Details of the simulated architecture

CPU	CPI	1.0
	Instruction set	based on DEC Alpha
Instruction cache	All instruction accesses assumed primary cache hits	
First level data cache	Capacity	8 Kbytes
	Line size	64 bytes
	Direct mapped, write-through	
Second-level cache	Capacity	4 Mbytes
	Line size	64 bytes
	Direct mapped, write-back	
DRAM	Capacity	Infinite
	Page size	8 Kbytes
Node controller	Non-pipelined	
	Service time and occupancy	See Table 1.2
	Cycle time	10ns
Interconnection network	Topology	full crossbar
	Incoming message queues	8 read requests
Cache coherence protocol	Invalidation-based, sequentially-consistent cc-NUMA. Home nodes allocated on "first-touch-after-initialisation" or "round-robin" basis. Distributed directory, based on the Stanford Distributed-Directory Protocol (Thapar and Delagi, 1990), using singly-linked sharing list.	

Table 1.2 Latencies of the most important node actions

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 1.3 Benchmark applications

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

we also wanted to observe how its performance would be affected by the two simple page placement policies. GE is interesting because it is a relatively long-running iterative code, where first-touch page placement becomes increasingly inappropriate over the execution time.

We selected six applications from the SPLASH-2 suite (Woo et al., 1995), to give a cross-section of scientific shared memory applications. We used both Ocean benchmark applications, in order to study the effects of page placement, and proxies, on the "tuned for data locality" and "easy to understand" variants. The Ocean-Contig implementation allows the grid partitions to be allocated contiguously and entirely in the local memory of the processors that "own" them, improving data locality but increasing algorithm complexity. In contrast, Ocean-Non-Contig implements the grids as 2-D arrays which prevents the partitions being allocated contiguously, but it is easier to understand and program. Other work which only refers to Ocean can be assumed to be using Ocean-Contig.

1.5 EXPERIMENTAL RESULTS

In this section, we present the results obtained from our simulations, and discuss the benefits and potential drawbacks of using proxies in conjunction with a default page placement policy. Because first-touch is often the default page placement policy, we have normalised the results with respect to first-touch. The results for each benchmark are expressed as percentages, *e.g.* it will be 100% for the relative execution time of each benchmark running with first-touch page placement and without proxies. The "number of messages" reflects the total number of messages sent. The "remote read response" measures the delay from sending a read request message to the receipt of the data.

1.5.1 Infinite Buffers

Without proxies, GE performs better with round-robin page placement, showing a 7.3% speedup over first-touch (see Figure 1.5). This was expected because the responsibility for updating rows of the matrix shifts during execution, so the first-touch placement slowly becomes inappropriate. Also, the remote access bottlenecks get worse towards the end of execution, as the accesses concentrate on fewer and fewer home nodes. In contrast, the round-robin strategy will have worse locality at the start of parallel execution, but it does not have the later

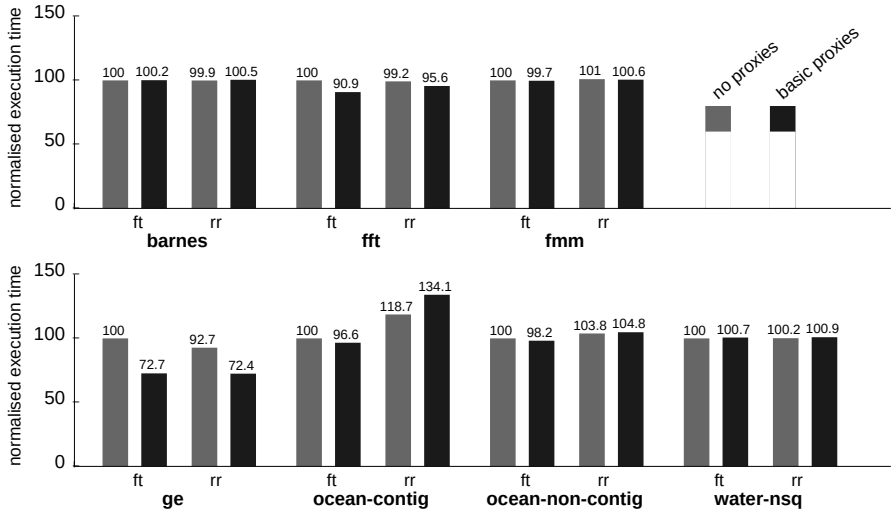


Figure 1.5 Relative performance, 64 nodes, infinite buffers

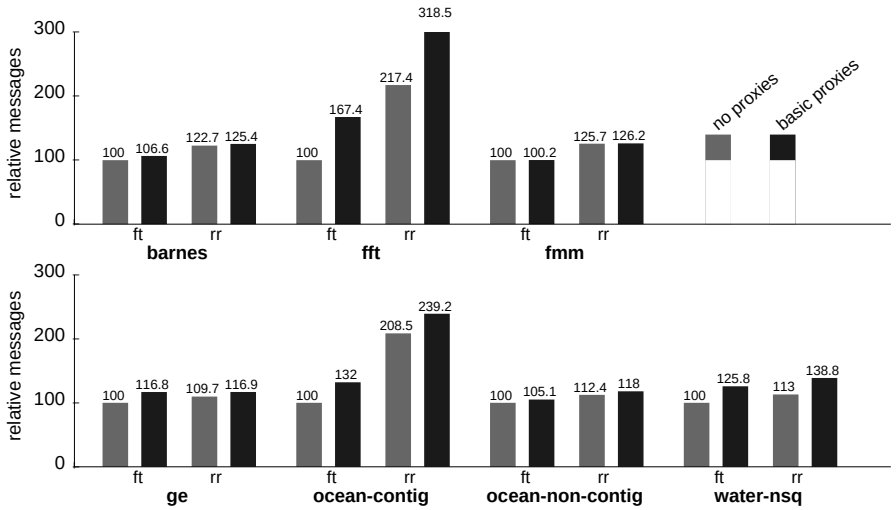


Figure 1.6 Relative number of messages, 64 nodes, infinite buffers

problem of access concentration. Figure 1.6 shows that, for GE, the overall number of messages increases by nearly 10% when round-robin is used, which reflects the increase in remote access requests owing to the loss of locality. However, because it avoids first-touch's problem of an increasing concentration of requests as the algorithm progresses, the service time for remote reads improves by 19% (as shown in Figure 1.7).

The performance of GE improves even more by using proxies. Both placement policies achieve more than 27% speedup over first-touch without proxies. Using

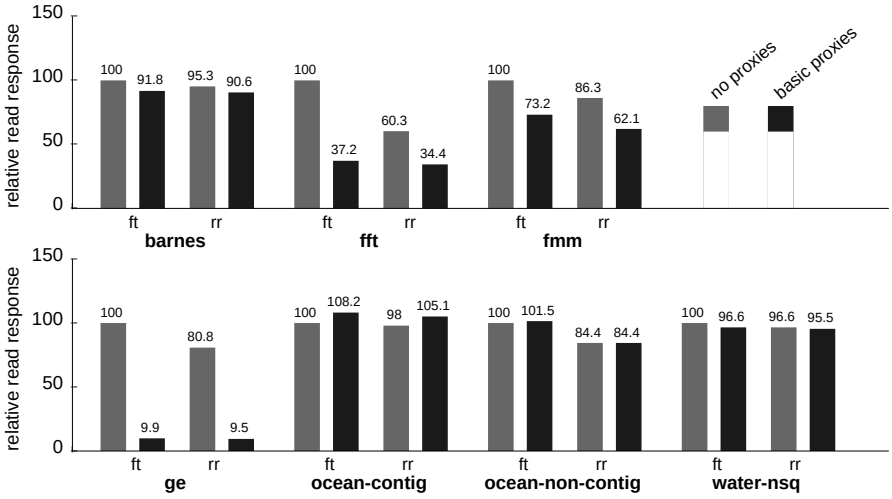


Figure 1.7 Relative remote read response times, 64 nodes, infinite buffers

proxies increases the overall number of messages by 17% in comparison to just first-touch, due to the additional proxy read requests and acknowledgements, but there is a dramatic reduction in the remote read response time of around 90%, because the combining of requests at proxies reduces the read messages queueing at the home node(s).

In contrast, the performance results for Ocean-Contig show first-touch as the best page placement policy, because this application has been written to exploit data locality. Round-robin leads to more remote reads, degrading performance to be nearly 19% worse than with first-touch. Using proxies makes the performance even worse for round-robin, increasing both mean remote read response time and the total number of messages, and these extra messages cause the network to overload. The best performance for the application is obtained using proxies with first-touch page placement.

FMM, Ocean-Non-Contig, and Water-Nsq perform marginally better with first-touch page placement, whereas Barnes and FFT perform marginally better with round-robin. In addition, FFT, FMM, and Ocean-Non-Contig get their best performance using first-touch page placement with proxies, with speedups of 9%, 0.3%, and 1.8% respectively. However, the performance of both Barnes and Water-Nsq suffers when basic proxies are used, showing slowdowns of 0.2% and 0.7%. These slight drops in performance illustrate the main pitfall of basic proxies, *i.e.* a poor choice of data marked for proxying.

1.5.2 Finite Incoming Message Buffers

The introduction of finite buffers favours the first-touch page placement policy (see Figure 1.8). The round-robin policy suffers because its lack of locality results in more remote access requests, which increases the chance that an input buffer already has eight or more messages, and so it is more likely that

Table 1.4 Mean input buffer queueing cycles

	infinite buffers				finite input buffers					
	no proxies		basic		no proxies		basic		reactive	
	ft	**rr**	**ft**	**rr**	**ft**	**rr**	**ft**	**rr**	**ft**	**rr**
Barnes	3.49	2.55	2.57	2.45	1.70	1.55	1.88	1.71	1.55	1.48
FFT	70.98	28.83	7.74	7.01	11.83	7.82	6.96	6.13	7.00	5.92
FMM	19.87	11.82	11.47	5.58	5.15	4.52	4.30	3.98	3.93	3.78
GE	301.60	213.62	15.95	13.13	38.59	37.23	10.01	8.11	60.77	59.06
OceanC	5.80	5.21	6.01	5.28	4.61	4.80	5.17	5.19	4.40	4.89
OceanN	15.21	11.03	15.93	11.14	9.83	9.45	10.34	9.09	8.72	8.49
Water	6.37	4.82	4.89	4.07	4.20	3.71	3.67	3.48	3.95	3.55

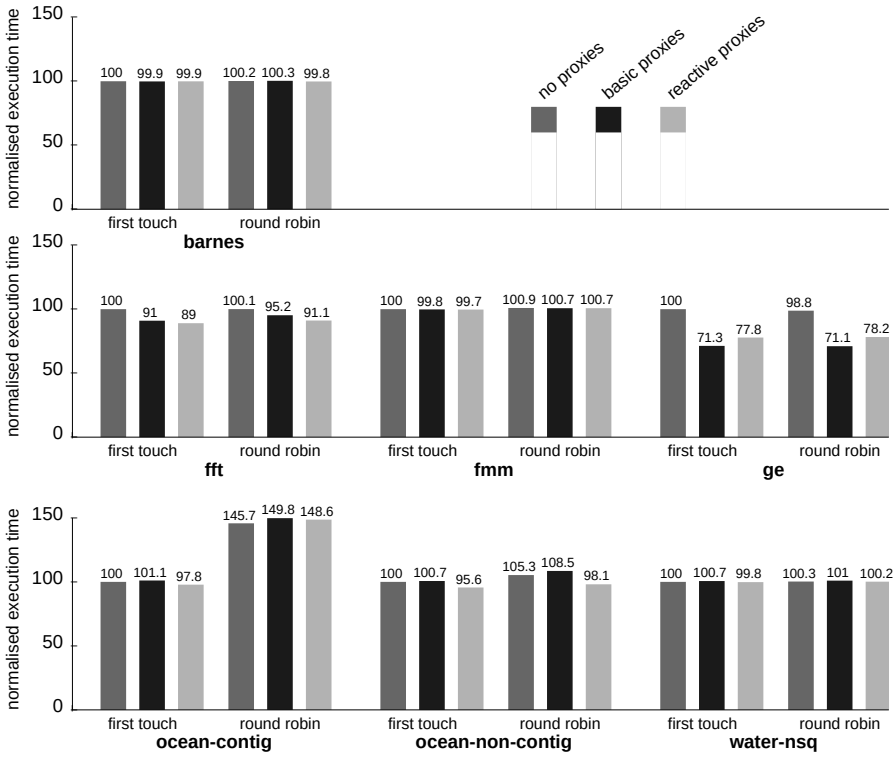


Figure 1.8 Relative performance, 64 nodes, finite buffers

a read request will be bounced. For six of the seven benchmark applications, round-robin page placement performs worse than first-touch, and for GE, the performance benefit has been cut to 1.2%.

Looking at the results with proxies, reactive proxies have the advantage that they handle all cases where contention occurs, but this is at the cost of a delay while the original read request is bounced by the home node. GE illustrates this change in behaviour (see Figure 1.9), where the use of proxies results in fewer messages because they stop the repeated sending and bouncing of read requests. However, the relative read response time does not improve as much

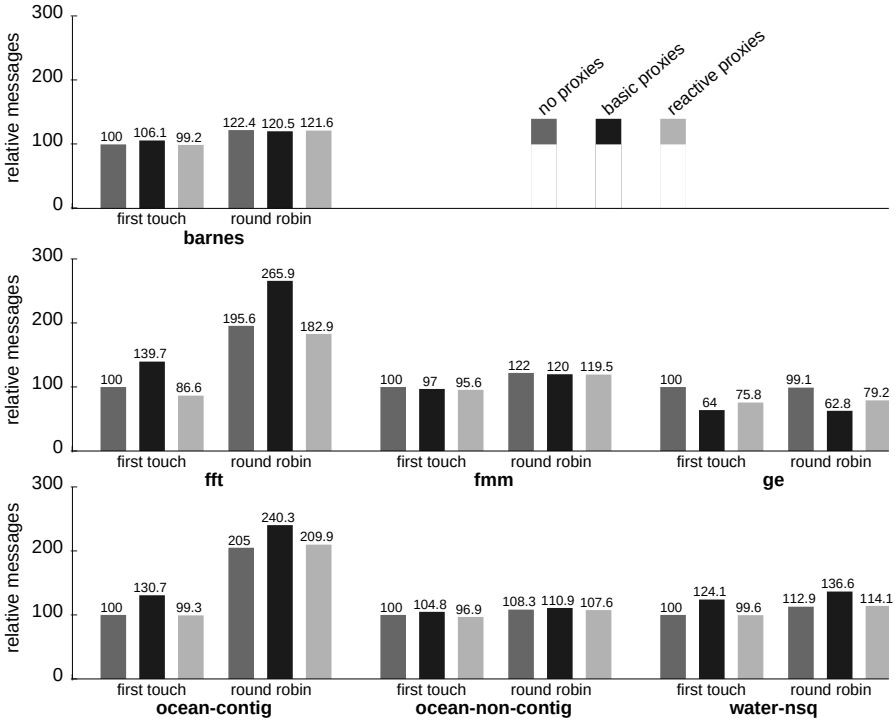


Figure 1.9 Relative number of messages, 64 nodes, finite buffers

for reactive as it does for basic proxies, as is shown in Figure 1.10. This is because of the initial delay, of read request and bounce, before a proxy read request is made. In addition, for GE, the mean input buffer queueing cycles increase with the introduction of proxies: this is because there are now more messages being accepted into the buffers rather than bouncing (see Table 1.4).

Using reactive proxies in conjunction with first-touch page placement results in the best performance for six of the seven benchmarks. The exception is GE: as we have already noted, it is particularly well-suited to the targeted approach of basic proxies. However, it still shows a marked performance improvement of 22.2% with reactive proxies. This suggests that a default policy of first-touch page placement with reactive proxies will give stable performance, and it avoids the more spectacular performance pitfalls that can occur using round-robin, such as occur for Ocean-Contig(Figure 1.8).

1.5.3 Summary

The choice of the best simple page placement policy, in the absence of any additional mechanism such as proxies or dynamic page migration/replication, depends on the individual applications. Some applications, such as our GE benchmark, suit the even distribution of shared data given by round-robin. Other applications, such as Ocean-Contig, have been specifically written to

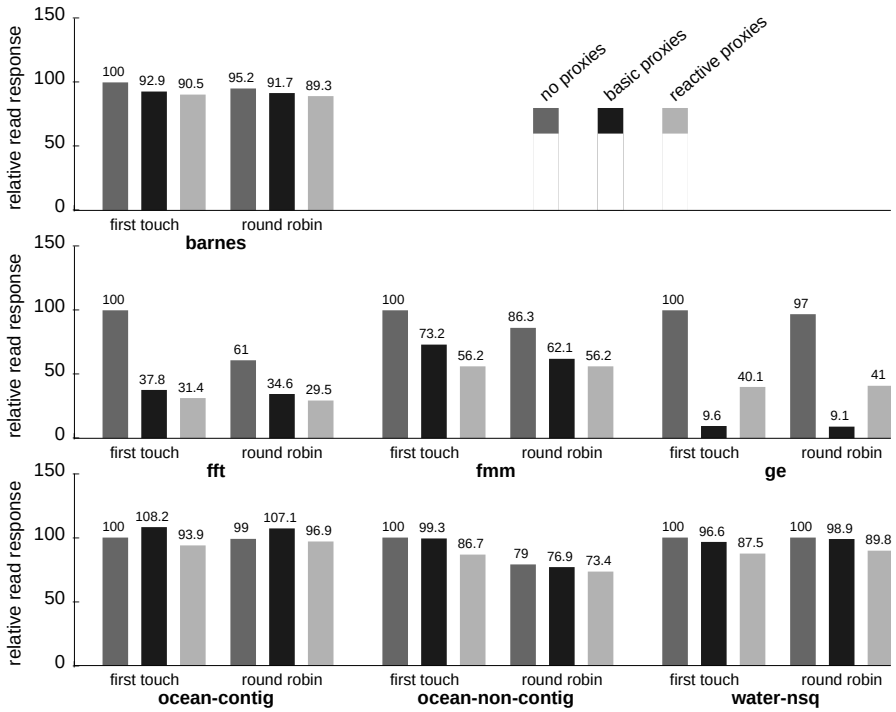


Figure 1.10 Relative remote read response times, 64 nodes, finite buffers

exploit data locality, and so suit the first-touch policy. This leaves the problem that the best performance may only be obtained by experienced programmers who know which page placement policy to choose, or by engaging in time-consuming performance tuning.

The use of proxies alters this situation. Proxies introduce a finer-grained sharing, at the level of data blocks rather than pages, and reduce queueing (as shown in Table 1.4). For basic proxies, this can be detrimental to performance where inappropriate data structures are marked as "hot", because every load (for addresses subject to proxying) goes via a proxy, whereas without proxies no indirection would be involved. Using reactive proxies has the benefit that proxies are only used when contention occurs at run-time: for six of our benchmarks this, in conjunction with first-touch page placement, resulted in their best performance. Water-Nsq illustrated this, where basic proxies degraded performance, whereas reactive proxies improved performance. Most importantly, first-touch page placement in conjunction with reactive proxies always resulted in performance that was better than either page placement policy without proxies. There are some applications, such as GE, where using reactive, rather than basic, proxies results in a smaller performance improvement because of the delay in invoking proxies; however, the application still showed a noticeable speedup over not using proxies.

1.6 RELATED WORK

The effects of page placement policy have been investigated for shared memory architectures where the coherence protocol is implemented in hardware (for cc-NUMA) or in software (for DVSM) (Marchetti et al., 1995). Using a base policy of round-robin, they also considered a first-touch after initialisation scheme, a dynamic migration scheme, and three replication schemes. Their first-touch scheme improved the performance of all their applications compared to naïve first-touch, regardless of whether coherence was maintained in hardware or software. For their five applications, there was no performance benefit from the dynamic migration or replication schemes. Their choice of benchmarks was limited, in that they did not include an example of algorithms such as Gaussian Elimination which are better suited to round-robin page placement.

A study of dynamic page migration and replication on Stanford FLASH, and distributed FLASH, considered three dynamic page placement policies, migration and/or replication, and three simple policies of round-robin, first-touch, and post facto (Verghese et al., 1996). They found that first-touch always gave better performance than round-robin for their workloads, and that post facto was the best simple policy. Their dynamic policies generally obtained better performance than first-touch, and were never worse, even given the overheads associated with implementing the dynamic policies. However, three of their five workloads were multiprogrammed, and this put their first-touch policy at a distinct disadvantage. For example, in their SPLASH workload, the jobs were redistributed across the processors as applications entered and left the system, but they did not re-invoke first-touch page placement when a job migrated.

Proxies allow read requests for data to be combined in controllers away from the home node: this is a restricted instance of the combining of atomic read-modify-write operations, *e.g.* as proposed for the NYU Ultracomputer (Gottlieb et al., 1983), although proxies retain the data in cache, which allows for more combining. Caching extra copies of data to speed-up retrieval time for remote reads has been explored for hierarchical architectures, *e.g.* in the Swedish Institute of Computer Science DDM (Haridi and Hagersten, 1989). The proxies approach is different because it does not use a fixed hierarchy; instead it allows requests for copies of successive data lines to be serviced by different proxies.

Eager combining uses intermediate nodes which act like proxies for "hot" pages, i.e. the programmer is expected to mark data structures (Bianchini and LeBlanc, 1994). Unlike proxies, their choice of server node is based on the page address rather than data block address. In addition, their scheme eagerly updates all proxies whenever a newly-updated value is read, unlike our protocol, where data is allocated in proxies on demand, which reduces cache pollution.

The GLOW extensions for widely-shared data are, like proxies, designed to be added to existing cache coherence protocols (Kaxiras and Goodman, 1996). GLOW uses agents to intercept requests for widely-shared data at selected network switch nodes. At present, GLOW requires application program directives to identify widely-shared data.

1.7 CONCLUSIONS

We have used execution driven simulations to study the benefits of using proxies and simple page placement. Our results confirm that there is no ideal default policy for page placement. However, by using reactive proxies with first-touch page placement, we obtained better performance than using either page placement policy without proxies. This suggests that, with a default of first-touch page placement with reactive proxies, application programmers can be confident that they will obtain stable performance. The programmer will not have to worry about the cc-NUMA implementation, and will rarely have to do time-consuming performance tuning.

There are some overheads associated with proxies. As we noted in Section 1.3, there are the costs of implementing proxies: in hardware to hold the head of each pending proxy chain, and in software to handle the additional message types and state changes. In addition, there will be cache pollution, because allocating a proxy copy in the cache may displace another line, with invalidation overhead for the displaced line, and possibly a later cache miss. However, these costs may be balanced by the considerable benefits of performance stability, the promise of architecture-independent application programs, and the saving of performance tuning effort.

Study of further benchmarks will provide deeper insight into the trade-offs, and in particular we are looking for applications which have not been carefully optimised for existing architectures. We are currently investigating the cache pollution effect, by examining both a "no-allocate" proxy scheme (where nodes do not cache the proxy lines), and the use of a separate proxy cache. We plan to continue our simulation work to evaluate how changing the architectural balance (*e.g.* slower interconnection networks) affects our conclusions.

Acknowledgments

This work was funded by the U.K. Engineering and Physical Sciences Research Council: through the CRAMP project (GR/J 99117), and a Research Studentship. We would also like to thank Ashley Saulsbury for the ALITE simulator, and Andrew Bennett and the anonymous referees for their comments on this work.

References

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 et al, editor, *Euro-Par 96 European Conference On Parallel Architectures, Lyon*, volume 1124 of *Lecture Notes in Computer Science*, pages 445–452. Springer-Verlag, 1996.

Ricardo Bianchini and Thomas J. LeBlanc. Eager combining: a coherency protocol for increasing effective network and memory bandwidth in shared-memory multiprocessors. In *6th IEEE Symposium on Parallel and Distributed Processing (SPDP), Dallas*, pages 204–213, October 1994.

John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. *4th Symposium on Principles and Practice of Parallel Programming, in Sigplan Notices*, 28(7):1–12, July 1993.

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.

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, June 1989*, vol. 365 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag.

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2nd edition, 1996.

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-660, Computer Systems Laboratory, Stanford University, January 1995.

Stefanos Kaxiras and James R. Goodman. The GLOW cache coherence protocol extensions for widely shared data. In *10th ACM International Conference on Supercomputing, May 25-28, Philadelphia, PA*, pages 35–43, May 1996.

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.

James Laudon and Daniel Lenoski. The SGI Origin 2000: A CC-NUMA highly scalable server. *24th Annual International Symposium on Computer Architecture, Denver, May 1997, in Computer Architecture News*, 25(2):241–251.

Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael L. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *9th International Parallel Processing Symposium (IPPS), Santa Barbara, CA*, pages 480–485, April 1995.

Manu Thapar and Bruce Delagi. Stanford distributed-directory protocol. *IEEE Computer*, 23(6):78–80, June 1990.

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *ASPLOS-VII, Cambridge, Mass, in ACM SIGPLAN Notices*, 31(9):279–289, September 1996.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture, June 1995, in Computer Architecture News*, 23(2):24–36.