# Reactive Proxies: a Flexible Protocol Extension to Reduce ccNUMA Node Controller Contention

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

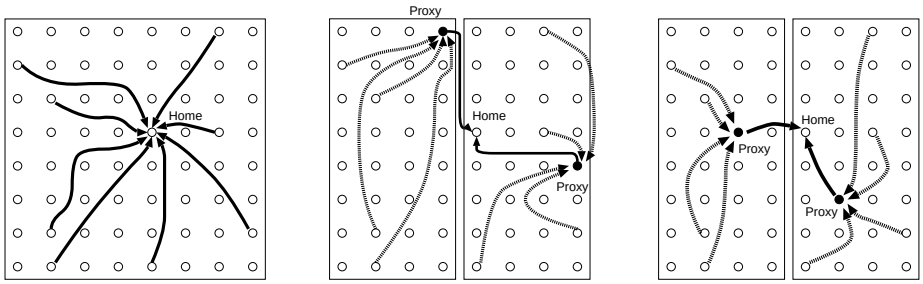
**Abstract.** Serialisation can occur when many simultaneous accesses are made to a single node in a distributed shared-memory multiprocessor. In this paper we investigate routing read requests via an intermediate proxy node (where combining is used to reduce contention) in the presence of finite message buffers. We present a *reactive* approach, which invokes proxying only when contention occurs, and does not require the programmer or compiler to mark widely-shared data. Simulation results show that the hot-spot contention which occurs in pathological examples can be dramatically reduced, while performance on well-behaved applications is unaffected.

## 1 Introduction

Unpredictable performance anomalies have hampered the acceptance of cache-coherent non-uniform memory access (ccNUMA) architectures. Our aim is to improve performance in certain pathological cases, without reducing performance on well-behaved applications, by reducing the bottlenecks associated with widely-shared data. This paper moves on from our initial work on proxy protocols [1], eliminating the need for application programmers to identify widely-shared data.

Each processor's memory and cache is managed by a node controller. In addition to local memory references, the controller must handle requests arriving via the network from other nodes. These requests concern cache lines currently owned by this node, cache line copies, and lines whose home is this node (*i.e.* the page holding the line was allocated to this node, by the operating system, when it was first accessed). In large configurations, unfortunate ownership migration or home allocations can lead to concentrations of requests at particular nodes. This leads to performance being limited by the service rate (occupancy) of an individual node controller, as demonstrated by Holt *et al.* [6].

Our proxy protocol, a technique for alleviating read contention, associates one or more proxies with each data block, *i.e.* nodes which act as intermediaries for reads [1]. In the basic scheme, when a processor suffers a read miss, instead of directing its read request directly to the location's home node, it sends it




(a) Without proxies (b) With two proxy clusters (read Line $l$) (c) Read next data block (*i.e.* read Line $l+1$)

**Fig. 1.** Contention is reduced by routing reads via a proxy

to one of the location's proxies. If the proxy has the value, it replies. If not, it forwards the request to the home: when the reply arrives it can be forwarded to all the pending proxy readers and can be retained in the proxy's cache. The main contribution of this paper is to present a *reactive* version, which uses proxies only when contention occurs, and does not require the application programmer (or compiler) to identify widely-shared data.

The rest of the paper is structured as follows: reactive proxies are introduced in Section 2. Our simulated architecture and experimental design are outlined in Section 3. In Section 4, we present the results of simulations of a set of standard benchmark programs. Related work is discussed in Section 5, and in Section 6 we summarise our conclusions and give pointers to further work.

## 2 Reactive Proxies

The severity of node controller contention is both application and architecture dependent [6]. Controllers can be designed so that there is multi-threading of requests (*e.g.* the Sun S3.mp is able to handle two simultaneous transactions [12]) which slightly alleviates the occupancy problem but does not eliminate it. Some contention is inevitable, and will increase the latency of transactions. The key problem is that queue lengths at controllers, and hence contention, are non-uniformly distributed around the machine.

One way of reducing the queues is to distribute the workload to other node controllers, using them as *proxies* for read requests, as illustrated in Fig. 1. When a processor makes a read request, instead of going directly to the cache line's home, it is routed first to another node. If the proxy node has the line, it replies directly. If not, it requests the value from the home itself, allocates it in its own cache, and replies. 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 Fig. 2. It should be noted that write requests are

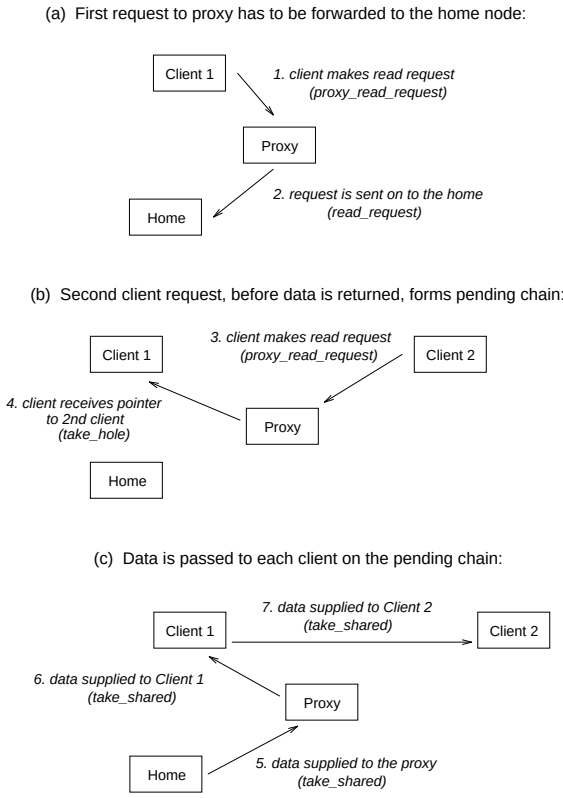
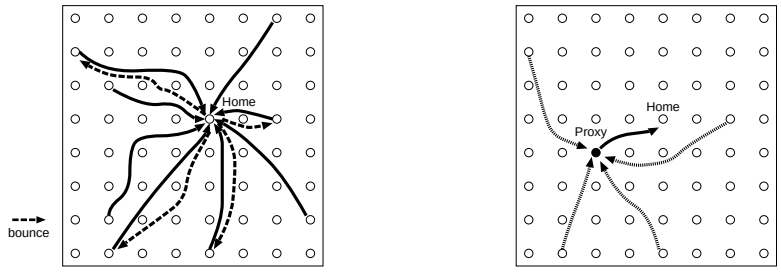



**Fig. 2.** Combining of proxy requests

not affected by the use of proxies, except for the additional invalidations that may be needed to remove proxy copies (which will be handled as a matter of course by the underlying protocol).

The choice of proxy node can be at random, or (as shown in Fig. 1) on the basis of locality. To describe how a client node decides which node to use as a proxy for a read request, we begin with some definitions:

- $\mathcal{P}$: the number of processing nodes.
- $\mathcal{H}(l)$: the home node of location $l$. This is determined by the operating system's memory management policy.
- $\mathcal{NPC}$: the number of proxy clusters, *i.e.* the number of clusters into which the nodes are partitioned for proxying (*e.g.* in Fig. 1, $\mathcal{NPC}$=2). The choice of $\mathcal{NPC}$ depends on the balance between 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 block will be directed to the same 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.




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

**Fig. 3.** Bounced read requests are retried via proxies

- $\mathcal{PCS}(C)$: the set of nodes which are in the cluster containing client node $C$. In this paper, $\mathcal{PCS}(C)$ is one of $\mathcal{NPC}$ disjoint clusters each containing $\mathcal{P}/\mathcal{NPC}$ nodes, with the grouping based on node number.
- $\mathcal{PN}(l, C)$ the proxy node chosen for a given client node ($C$) when reading location $l$. We use a simple hash function to choose the actual proxy from the proxy cluster $\mathcal{PCS}(C)$. If $\mathcal{PN}(l, C) = C$, or $\mathcal{PN}(l, C) = \mathcal{H}(l)$, then client $C$ will send a read request directly to $\mathcal{H}(l)$

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 hashing 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 blocks go to different proxies (as illustrated in Fig. 1(c)). This will reduce network contention [15] and balance the load more evenly across all the node controllers.

In the basic form of proxies, the application programmer uses program directives to mark data structures: all other shared data will be exempt from proxying [1]. If the application programmer makes a poor choice, 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 fail to mark data structures that would benefit from proxying.

Reactive proxies overcome these problems by taking advantage of the finite buffering of real machines. When a remote read request reaches a full buffer, it will immediately be sent back across the network. With the reactive proxies protocol, the arrival of a buffer-bounced read request will trigger a proxy read (see Fig. 3). This is quite different to 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, proxies are only used when congestion occurs. As soon as the queue length at the destination node has reduced to below the limit, read requests will no longer be bounced and proxy reads will not be used.

The repeated bouncing of read requests which can occur with finite buffers leads to the possibility of deadlock: the underlying protocol has to detect the continuous re-sending of a remote read request, and eventually send a higher priority read request which is guaranteed service. Read requests from proxy nodes to home nodes will still be subject to buffer bouncing, but the combining and re-routing achieved by proxying reduce the chances of a full input buffer at the home node.

The reactive proxy scheme has the twin virtues of simplicity and low overheads. No information needs to be held about past events, and no decision is involved in using a proxy: the protocol state machine is just set up to trigger a proxy read request in response to the receipt of a buffer-bounced read request.

## 3 Simulated Architecture and Experimental Design

In our execution-driven simulations, 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 Fig. 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. Table 1 summarises the architecture.

We simulate a simplified interconnection network, which follows the the *LogP* model [3]. 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. This represents a fast network, comparable to the point-to-point latency used in [11].
- $o$: the occupancy of the node controller. Like Holt *et al.* [6], 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. The processes which cause occupancy are simulated in more detail (see Table 2).
- $g$: the gap between successive sends or receives by a processor, 5 cycles.
- $P$: the number of processor nodes, 64 processing nodes.

We limit our message buffers to eight for 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 $\sqrt{P}$ is an arbitrary but reasonable limit.

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 (*i.e.* the owner); has guaranteed space in DRAM for the line; and holds directory information for the line (head and state of the sharing

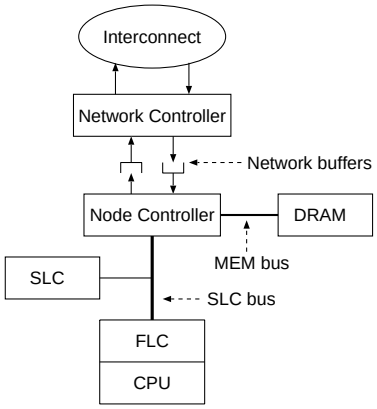



**Fig. 4.** The architecture of a node

**Table 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 2 |
| | Cycle time | 10ns |
| Interconnection network | Topology | full crossbar |
| | Incoming message queues | 8 read requests |
| Cache coherence protocol | Invalidation-based, sequentially-consistent ccNUMA, home nodes assigned to first node 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 [14] | |

**Table 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 3.** Benchmark applications

| application | problem size | shared data marked for basic proxying |
|---|---|---|
| Barnes | 16K particles | all |
| CFD | 64 x 64 grid | 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 |

list). The distributed directory holds the identities of nodes which have cached a particular line in a sharing chain, currently implemented as a singly-linked list.

The directory entry for each data block provides the basis for maintaining the consistency of the shared data. Only one node at a time can remove entries from the sharing chain (achieved by locking the head of the sharing chain at the home node), and messages which prompt changes to the sharing chain are ordered by their arrival at the home node. This mechanism is not affected by the protocol additions needed to support proxies.

Proxy nodes require a small amount of extra store to be added to the node controller. Specifically we need to be able to identify which data lines have outstanding transactions (and the tags they refer to), and be able to record the identity of the head of the pending proxy chain. In addition, the node controller has to handle the new proxy messages and state changes. We envisage implementing these in software on a programmable node controller, *e.g.* the MAGIC node controller in Stanford's FLASH [9], or the SCLIC in the Sequent NUMA-Q [10].

The benchmarks and their parameters are summarised in Table 3. GE is a simple Gaussian elimination program, similar to that used by Bianchini and LeBlanc in their study of eager combining [2]. We chose this benchmark because it is an example of widely-shared data. CFD is a computational fluid dynamics application, modelling laminar flow in a square cavity with a lid causing friction [13]. We selected six applications from the SPLASH-2 suite, to give a cross-section of scientific shared memory applications [16]. We used both Ocean benchmark applications, in order to study the effect of proxies on the "tuned for data locality" and "easy to understand" variants. Other work which refers to Ocean can be assumed to be using Ocean-Contig.

## 4 Experimental Results

In this work, we concentrate on reactive proxies, but compare the results with basic proxies (which have already been examined in [1]). The performance results for each application are presented in Table 4 in terms of relative speedup with no proxying (i.e. the ratio of the execution time for 64 processing nodes to the execution time running on 1 processor), and percentage changes in execution time when proxies are used. The problem size is kept constant.

The relative changes results in Fig. 5 show three different metrics:

**Table 4.** Benchmark performance for 64 processing nodes

| applications | 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 | 43.2 | basic | 0.0 | -0.1 | 0.0 | 0.0 | -0.2 | +0.3 | -0.2 | -0.1 |
| | | reactive | +0.3 | +0.2 | +0.3 | +0.2 | +0.1 | -0.1 | 0.0 | +0.3 |
| CFD | 30.6 | basic | +6.6 | +7.7 | +6.4 | +8.3 | +6.7 | +5.8 | +6.0 | +10.2 |
| | | reactive | +5.6 | +5.4 | +4.7 | +4.5 | +5.6 | +4.1 | +4.1 | +4.8 |
| FFT | 47.4 | basic | +9.3 | +9.0 | +9.8 | +9.4 | +9.2 | +8.8 | +8.9 | +8.9 |
| | | reactive | +11.5 | +11 | +10.8 | +10.8 | +11.1 | +11.6 | +11.0 | +10.6 |
| FMM | 36.1 | basic | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 |
| | | reactive | +0.4 | +0.3 | +0.3 | +0.4 | +0.3 | +0.4 | +0.3 | +0.3 |
| GE | 22.0 | basic | +28.7 | +28.7 | +28.7 | +28.7 | +28.7 | +28.8 | +28.8 | +28.7 |
| | | reactive | +23.3 | +22.9 | +22.3 | +21.4 | +21.5 | +21.4 | +21.7 | +21.5 |
| Ocean-Contig | 48.9 | basic | -3.2 | +2.6 | +0.1 | -0.2 | -0.8 | -2.0 | -2.1 | -1.8 |
| | | reactive | -0.2 | 0.0 | -0.1 | -0.3 | +0.2 | +2 | +1.3 | +2.1 |
| Ocean-Non-Contig | 50.5 | basic | -0.3 | +1.6 | -1.7 | +4.1 | +1.0 | -0.4 | +0.2 | +4.2 |
| | | reactive | +3.1 | +1.4 | +1.4 | +0.8 | +5.1 | +1.8 | +1.2 | +5 |
| Water-Nsq | 55.5 | basic | -0.7 | -0.6 | -0.6 | -0.5 | -0.5 | -0.5 | -0.7 | -0.5 |
| | | reactive | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 | +0.2 | +0.1 | +0.2 |

- *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

The message ratios shown in Fig. 6 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),
- *remote read delay*: the ratio of the delay between issuing a read request and receiving the data, to the same delay when proxies are not used.
- *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.
- *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.

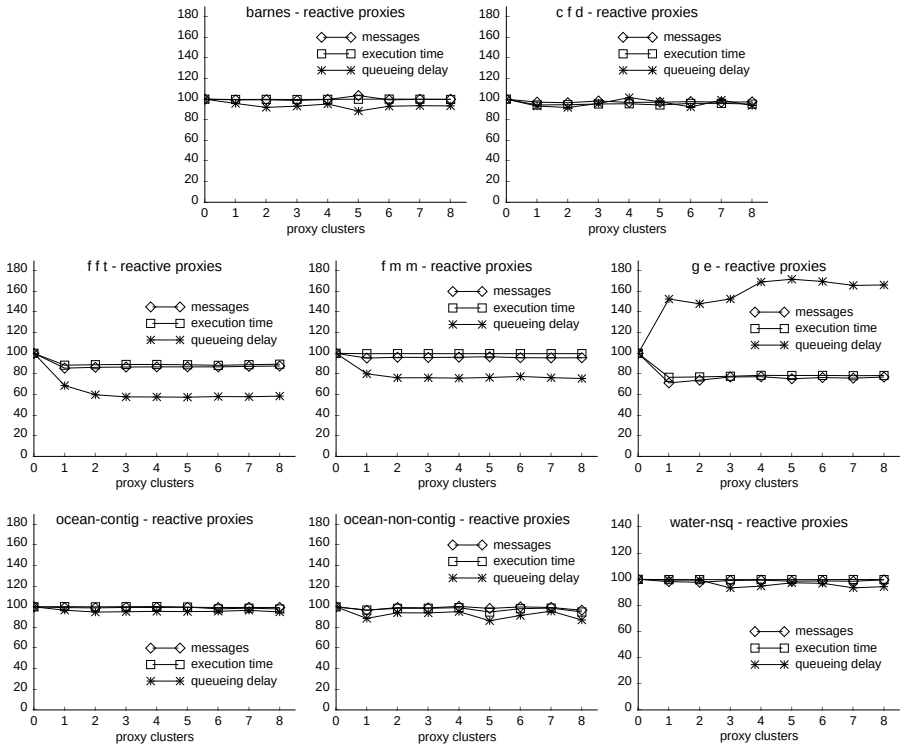



**Fig. 5.** Relative changes for 64 processing nodes with reactive proxies

The first point to note from Table 4 is that there is no overall "winner" between basic and reactive proxies, in that neither policy improves the performance of all the applications for all values of proxy clusters. 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 two proxy cluster values that improve the performance of all the benchmarks, *i.e.* $\mathcal{NPC}$=5 and 8 achieve a balance between combining, queue distribution, and length of the proxy pending chains. Reactive proxies may not always deliver the best performance improvement, but by providing stable points for $\mathcal{NPC}$ they are of more use to system designers. It should also be noted that, in general, using reactive proxies reduces the number of messages, because they break the cycle of re-sending read messages in response to a finite buffer bounce (see Fig. 5).

Looking at the individual benchmarks:

**Barnes.** In general, this application benefits from the use of reactive proxies. However, 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. Two examples illustrate this: when $\mathcal{NPC}$=6 for basic proxies, load miss delay is the same as with no proxies, store miss delay has increased slightly

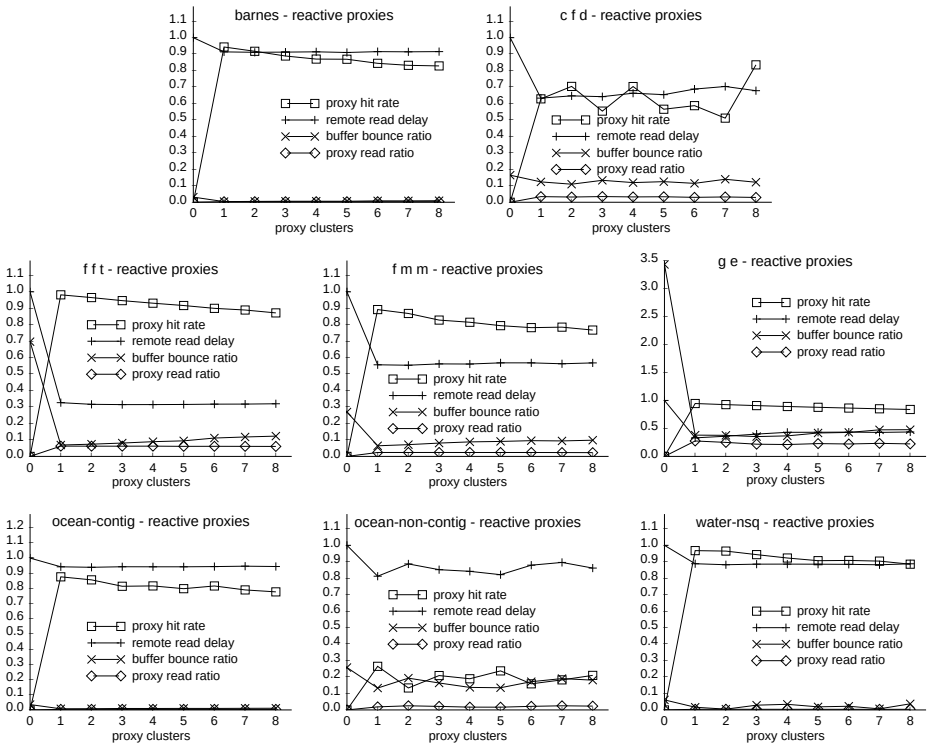



**Fig. 6.** Message ratios for 64 processing nodes with reactive proxies

from the no proxy case, yet a reduction of 0.4% in lock and barrier delays results in an overall performance improvement of 0.3%. Conversely, for reactive proxies when $\mathcal{NPC}$=6, the load and store miss delays are the same as when proxies are not used, but a slight 0.1% increase in lock delay gives an overall performance degradation of -0.1%.

**CFD.** This application benefits from the use of reactive proxies, with performance improvements in the range 4.1% to 5.6%. However, the improvements are not as great as those obtained with basic proxies. The difference is attributable to the delay in triggering each reactive proxy read: for this application it is better to use proxy reads straight away, rather than waiting for read requests to be bounced. It should also be noted that the proxy hit rate oscillates, with peaks at $\mathcal{NPC}$=2,4,8 (see Fig. 6). This is due to a correspondence between the chosen proxy node and ownership of the cache line.

**FFT.** This shows a marked speedup when reactive proxies are used, of between 10.6% and 11.6%. The number of messages decreases with proxies because the buffer bounce ratio is cut from a severe 0.7 with no proxies. The mean queueing delay drops down as the number of proxy clusters increases, reflecting the benefit of spreading the proxy read requests. However, this is balanced by a slow increase in the buffer bounce ratio, because as more nodes act as proxy there

will be more read requests to the home node, and these read requests will start to bounce as the number of messages in the home node's input queue rises to $\sqrt{\mathcal{P}}$ and above.

**FMM.** There is a marginal speedup compared with no proxies (between 0.3% and 0.4%). This is as expected given only the *f_array* (part of *G_Memory*) is known to be widely-shared, which was why it was marked for basic proxies. However, the performance improvement is slightly better than that achieved using basic proxies, so the reactive method dynamically detects opportunities for read combining which were not found by code inspection and profiling tools.

**GE.** This application, which is known to exhibit a high level of sharing of the current pivot row, shows a large speedup in the range 21.4% to 23.2%. However, the improvement is not as good as that obtained using basic proxies. This was to be expected, because proxying is no longer targeted by marking widely-shared data structures. Instead proxying is triggered when a read is rejected because a buffer is full, and so there will be two messages (the read and buffer bounce) before a proxy read request is sent by the client. It should also be noted that the execution time increases as the number of proxy clusters increases. As the number of nodes acting as proxy goes up, there will be more read requests (from proxies) being sent to the home node, and the read requests are more likely to be bounced, as shown by the buffer bounce ratio for GE in Fig. 6. Finally, the queueing delay is much higher when proxies are in use. This is because without proxies there is a very high level of read messages being bounced (and thus not making it into the input queues). With proxies, the proxy read requests are allowed into the input queues, which increases the mean queue length.

**Ocean-Contig.** Reactive proxies can degrade the performance of this application (by up to -0.3% at $\mathcal{NPC}$=4), but they achieve performance improvements for more values of $\mathcal{NPC}$ than the basic proxy scheme. Unlike basic proxies, reactive proxies reduce the remote read delay by targeting remote read requests that are bounced because of home node congestion. The performance degradation when $\mathcal{NPC}$=1,3,4 is attributable to increased barrier delays caused by the redistribution of messages.

**Ocean-Non-Contig.** This has a high level of remote read requests. These remote read requests result in a high level of buffer bounces, which in turn invoke the reactive proxies protocol. Unfortunately the data is seldom widely-shared, so there is little combining at the proxy nodes, as is illustrated by the low proxy hit rates. With $\mathcal{NPC}$=4, this results in a concentration of messages at a few nodes, overall latency increases, and the execution time suffers. For $\mathcal{NPC}$=5, the queueing delay is reduced in comparison to the no proxy case, and this has the best execution time. Given these results, we are carrying out further investigations into the hashing schemes suitable for the $\mathcal{PN}(l, C)$ function, and the partitioning strategy used to determine $\mathcal{PCS}(C)$, to obtain more reliable performance for applications such as Ocean-Non-Contig.

**Water-Nsq.** Using reactive proxies gives a small speedup compared to no proxies (around 0.2%). However, this is better than with basic proxies, where performance is always worse (in the range -0.5% to -0.7%, see Table 4). The

extremely low proxy read ratios shows that there is very little proxying, but the high proxy hit rates indicate that when proxy reads are invoked there is a high level of combining. It is encouraging to see that the proxy read ratio is kept low: this shows that the overheads of proxying (extra messages, cache pollution) are only incurred when they are needed by an application.

To summarise, the results show that for reactive proxies, when the number of proxy clusters ($\mathcal{NPC}$) is set to five or eight, the performance of all the benchmarks improves, *i.e.* they achieve the best balance between combining, queue length distribution, and the length of the proxy pending chains in our simulated system. This is a very encouraging result, because without marking widely-shared data we have obtained sizeable performance improvements for three benchmarks (GE, FFT, and CFD), and had no detrimental effect on the other well-behaved applications. By selecting a suitable $\mathcal{NPC}$ for an architecture, the system designers can provide a ccNUMA system with more stable performance. 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.

## 5 Related Work

A number of measures are available to alleviate the effects of contention for a node, such as improving the node controller service rate [11], and combining in the interconnection network for fetch-and-update operations [4]. Architectures based on clusters of bus-based multiprocessor nodes provide an element of read combining since caches in the same cluster snoop their shared bus. Caching extra copies of data to speed-up retrieval time for remote reads has been explored for hierarchical architectures, including [5]. 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.

Attempts have been made to identify widely-shared data for combining, including the GLOW extensions to the SCI protocol [8, 7]. GLOW intercepts requests for widely-shared data by providing agents at selected network switch nodes. In their dynamic detection schemes, which avoid the need for programmers to identify widely-shared data, agent detection achieves better results than the combining of [4] 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.

In Bianchini and LeBlanc's "eager combining", the programmer identifies specific memory regions for which a small set of server caches are pre-emptively updated [2]. Eager combining uses intermediate nodes which act like proxies for marked pages, *i.e.* their choice of server node is based on the page address rather than data block address, so their scheme does not spread the load of messages around the system in the fine-grained way of proxies. 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. Our less aggressive scheme reduces cache pollution at the proxies.

## 6 Conclusions

This paper has presented the reactive proxy technique, discussed the design and implementation of proxying cache coherence protocols, and examined the results of simulating eight benchmark applications. We have shown that proxies benefit some applications immensely, as expected, while other benchmarks with no obvious read contention still showed performance gains under the reactive proxies protocol. There is a tradeoff between the flexibility of reactive proxies and the precision (when used correctly) of basic proxies. However, reactive proxies have the further advantage that a stable value of $\mathcal{NPC}$ (number of proxy clusters) can be established for a given system configuration. This gives us the desired result of improving the performance of some applications, without affecting the performance of well-behaved applications. In addition, with reactive proxies, the application programmer does not have to worry about the architectural implementation of the shared-memory programming model. This is in the spirit of the shared-memory programming paradigm, as opposed to forcing the programmer to restructure algorithms to cater for performance bottlenecks, or marking data structures that are believed to be widely-shared.

We are currently doing work based on the Ocean-Non-Contig application to refine our proxy node selection function ($\mathcal{PN}(l, C)$). In addition, we are continuing our simulation work with different network latency ($L$) and finite buffer size values. We are also evaluating further variants of the proxy scheme: adaptive proxies, non-caching proxies, and using a separate proxy cache.

## Acknowledgements

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 Andrew Bennett and Ashley Saulsbury for their work on the ALITE simulator and for porting some of the benchmark programs.

## References

1. 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, August 1996.
2. 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, Dallas*, pages 204–213, October 1994.

3. 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.
4. Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer – designing a MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.
5. 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.
6. 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.
7. David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.
8. Stefanos Kaxiras, Stein Gjessing, and James R. Goodman. A study of three dynamic approaches to handle widely shared data in shared-memory multiprocessors. In *(to appear) 12th ACM International Conference on Supercomputing, Melbourne*, July 1998.
9. Jeffrey Kuskin. *The FLASH Multiprocessor: designing a flexible and scalable system.* PhD thesis, Computer Systems Laboratory, Stanford University, November 1997. Also available as a technical report, CSL-TR-97-744.
10. Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. *23rd Annual International Symposium on Computer Architecture, Philadelphia, in Computer Architecture News*, 24(2):308–317, May 1996.
11. Maged M. Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. *24th Annual International Symposium on Computer Architecture, Denver, in Computer Architecture News*, 25(2):219–228, June 1997.
12. Andreas Nowatzyk, Gunes Aybay, Michael Browne, Edmund Kelly, Michael Parkin, Bill Radke, and Sanjay Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the International Conference on Parallel Processing Vol. 1*, pages 1–10, August 1995.
13. 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.
14. Manu Thapar and Bruce Delagi. Stanford distributed-directory protocol. *IEEE Computer*, 23(6):78–80, June 1990.
15. 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.
16. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Proceedings of the 22nd Annual International Symposium on Computer Architecture, in Computer Architecture News*, 23(2):24–36, June 1995.

This article was processed using the LaTeX macro package with LLNCS style