FORTH/ICS/TR/030 July 1993

TRACE-DRIVEN SIMULATION OF DATA-ALIGNMENT AND OTHER FACTORS AFFECTING UPDATE AND INVALIDATE BASED COHERENT MEMORY

Evangelos P. Markatos[*] *and Catherine E. Chronaki*[*]

Abstract

The exploitation of locality of reference in shared memory multiprocessors is one of the most important problems in parallel processing today. Locality can be managed in several levels: hardware, operating system, runtime environment of the compiler, user level.
In this paper we investigate the problem of exploiting locality at the operating system level and its interactions with the compiler and the architecture. Our main conclusion, based on trace-driven simulations of real applications, is that exploitation of locality is effective only if all three levels cooperate. The compiler should do sophisticated data alignment, the operating system should perform on-line caching and page replication, while the architecture should provide simple but effective hardware mechanisms that assist the operating system in avoiding unnecessary movement of data.

[*]Institute of Computer Science, F.O.R.T.H.
E-mail {markatos,chronaki}@csi.forth.gr or {markatos,chronaki}@ariadne.bitnet

1 Introduction

The exploitation of locality of reference in shared memory multiprocessors is one of the most important problems in parallel processing today. Locality can be managed in several levels: hardware, operating system, runtime environment of the compiler, user level.

One way to exploit locality at any level is by bringing data close to the processor that references them. Hardware does this transparently by bringing data to local caches and keeping all caches coherent by using a cache coherence protocol [1]. The operating system may perform this same function by replicating pages close to processors. The user may do the same thing by explicitly copying data in the parallel program. Although bus-based multiprocessors [18, 19], and several large-scale multiprocessors [8] make use of hardware coherent caches, software coherence implemented in the operating system/run time system is starting to become more attractive for several reasons:

- Software protocols can afford to be more sophisticated than hardware protocols. They can be debugged easier and changed easily.
- Hardware protocols start to loose the traditional advantage of speed that they had over software protocols, because the overhead of locality management is dominated by data transfers, and not by code execution. As processors are getting faster and faster than memories [17], the cost of operating system code execution compared to the cost of a data transfers is getting lower. Given the fact that both hardware and software protocols have to pay the cost of data transfer, the portion of the protocol execution that hardware can speed up is decreasing with time.
- Protocols implemented at the operating system level manage locality transparently, just like hardware implemented protocols. Thus, the user does not need to explicitly program locality, nevertheless (s)he may supply hints to the compiler or the operating system.
- Hardware protocols have a high implementation cost, especially when large directory structures are needed. The cost of keeping *large* directories in hardware, and reading/updating one or more directory words in each processor cycle (which is typically a few nanoseconds) is prohibitive for most parallel machines.

In this paper we address locality of reference for distributed shared memory systems mostly at the operating system and run time system levels; the use of limited hardware support is also studied in some cases. The fundamental mechanism for exploiting locality in the operating system level is replicating pages close to processors that frequently use them. Page replication may result to up to an order of magnitude performance improvement, since local memory accesses are often about an order of magnitude faster than remote memory access. Unfortunately, the existence of multiple copies of the same page, introduces the problem of memory coherence. That is, when one processor updates its local copy of a page, the other processors that have a copy of the same page need to be informed of the change.

The developer of a memory coherence protocol in software has to evaluate several trade-offs and make a number of important decisions. In this paper we examine and evaluate four of these issues as described below.

1.1 Update-based or Invalidate-based protocols?

Suppose that shared data have been replicated to several processors, and a processor wants to update a shared datum in its local memory. Then, all other copies of the datum need to be updated, so that all processors have a consistent view of memory. This update can be done using one of two mechanisms:

> (a) Invalidate of all copies of the page where the datum resides, except for the writer's. The next time a processor, besides the writer, wants to read or write a datum on that page, it will page-fault and request a new copy.
> (b) Update all existing copies of the datum by sending a message to all processors that have a copy of the page where the datum resides. The message contains the address of the data that changed and the new value.

Invalidate-based mechanisms are generally simple and can be implemented completely in software. That's why most operating system protocols developed so far, use it. Their main drawback is that the unit of invalidation is the page. If pages are large, invalidations may result in frequent page faults and expensive page replications. *Page bouncing* [14], the phenomenon where a page is constantly invalidated and replicated, is not unusual in memory coherence systems.

Update-based mechanisms are more complicated and need some hardware support to perform non-local updates. Nevertheless only the necessary information is transferred over the network. Update-based protocols are best suited for the producer/consumer data reference pattern: the producer updates the common data buffer, and the hardware sends the updates to the consumer transparently. When the consumer will later need to access the produced data, the data will be in its local memory. However, this mechanism is not well suited when there are multiple execution phases in which a datum is first used by one subset of the processors and then by another. A processor that does not need updates for some datum anymore will still receive them, at the cost of higher network traffic.

The results of our experiments indicate that protocols based on updates are almost always better than those based on invalidates (see section 3.2).

1.2 Small or Large Page size?

In a memory coherent system, the unit of sharing and coherence is usually the page. Large page sizes may increase the amount of false sharing[*] significantly. It is much more probable to have false sharing in a 64K page, than in a 256 byte page (or cache line). Small page sizes, on the other hand, may increase the operating system overhead. For example, if a range of memory will eventually be replicated to a processor, replicating it in small pages, and taking one fault for each page is rather expensive. Small page sizes, also increase the number of TLB misses, which may also sum up to a considerable cost. Our experiments indicate that larger pages generally result in worse performance. The effect of large page size on invalidate-based protocols is *very* pronounced. The performance penalty in update-based protocols increases very slowly with the page size, and sometimes, it even decreases with the page size.

1.3 With a little help from the hardware? The role of hardware counters.

Hardware counters may be used to assist the system software in making informed decisions about replicating pages. For example, several protocols (esp. the hardware ones) replicate a page the first time it is accessed by the processor. What we would like instead is to replicate a page only after we make sure that the cost of the replication will be offset by the benefits of replication. Thus, we would like to replicate a page, only if we know that the processor will make frequent use of it. Bolosky [4] proposes the use of a hardware counter, that counts the number of non-local accesses to the page, as

[*]A page is falsely shared if it is accessed by at least two processors, at least one of which updates it, but the processors do not access the same memory location in the page.

an indication of the frequency of use of a page by a processor. Only after the processor has made a specified number of non-local accesses, is the page replicated locally.

Hardware counters at a small additional cost improve the performance of coherence protocols, and provide a useful mechanism for eliminating page bouncing (see section 3.3).

1.4 Can sophisticated data alignment reduce False Sharing?

The applications we use have been compiled with a traditional compiler that does not attempt to do any sophisticated data alignment. Thus, it is possible that data with different access patterns will all end up in the same page. Such cases introduce significant false sharing. If a page is falsely shared, the memory coherence policy has no good solution. Say for example that processor P_1 frequently reads and updates one memory location of some page, while processor P_2 frequently reads and updates another memory location of the same page. If the page is replicated to both processors, a large number of updates (or invalidates) will travel over the communication network. If the page is replicated to one processor only, the other will make all its operations remotely. In both cases, the memory coherence policy has no good solution. The right solution for this case is to allocate the data that correspond to two different locations on different pages, so that each page can be replicated to the processor that accesses it, and no information will have to travel over the interconnection network.

Our experiments suggest that sophisticated data alignment is the single most important source of performance improvement. This fact calls for a cooperation between the compiler and the operating system to solve the false sharing problem together (see section 3.4).

The next section provides and overview of the previous work and places our contributions in context. Section 3 presents our experiments and answers to the above stated questions. Finally, section 4 summarizes our results and presents our conclusions.

2 Previous Work

Memory coherence is an active area of research. Several implementations of memory coherence protocols exist in shared-memory multiprocessors [15, 7, 5, 12, 13]. A lot of work has also been done in implementing coherent memory in message-passing multiprocessors [16, 10].

Bolosky *et. al* [4] have compared several memory coherence policies. In a companion paper [3] he has shown that the dominant overhead in memory coherence policies is false sharing, which can be significantly reduced by decreasing the page size.

Eggers and Katz [9] have evaluated the performance trade offs of invalidate and update-based cache-coherence protocols, for small scale bus-based multiprocessors that use snooping cache coherence protocols. Her results were not conclusive towards any kind of protocols, and concluded that each protocol outperforms the others under the appropriate conditions.

Veenstra and Fowler [21, 20] have compared invalidate and update-based cache-coherence protocols. They conclude that for large cache lines update-based protocols are better. They also conclude that a hybrid protocol that combines both updates and invalidates is always the best.

Our work differs from previous work in memory coherency protocols in several aspects:

- We take into account the effect of sophisticated *data alignment* in the performance of the memory coherence protocol. Thus, we quantify not only the performance of memory coherence protocols on given applications, but also how much overhead is *inherent* to the application, and how much is attributed to naive data alignment decisions made by the compiler.

- We quantify the use of *hardware counters* for memory coherence protocols. Although some of the effects of hardware counters had been quantified before ([4]), their interaction with sophisticated data re-alignment, and update-based policies has not been quantified. Such a quantification is important to make sure that hardware counters are a generally useful hardware mechanism.

- We quantify the relative trade offs of update-based and invalidation-based protocols for large-scale multiprocessors. Although, some of relevant trade offs have been studied [9], that study was done in the domain of small scale hardware cache-coherent bus-based multiprocessors. We are interested in studying the alternatives in *large scale* multiprocessors, that support software coherence.
- Our study applies in virtual shared memory systems where caching and coherency is under the operating system control (with a little help from the hardware), while most previous studies have focused on hardware cache-coherent systems.

3 Experiments

3.1 Experimental Environment

3.1.1 The traces

To evaluate the tradeoffs in the design of a memory coherency policy, we use trace-driven simulation. Trace-drive simulation consists of feeding the traces (memory access patterns) of an application into a simulation that accurately simulates a memory coherence protocol. For each trace record, the simulator:

- Examines the action it needs to perform (read or write) and the address it needs to access.
- Simulates the necessary replication, invalidation, update and other actions it needs to take.
- Updates the cost metrics depending on the number and the kind of actions it took in the previous step.

The traces we use are 64-processor traces gathered from four programs: FFT, SIMPLE, WEATHER and SPEECH [6]. FFT is a fast Fourier transform. SPEECH is the lexical decoding stage of a phonetically based spoken language understanding system. WEATHER is an application for weather simulation based on a grid method to solve a set of partial differential equations. SIMPLE models the behavior of fluids and solves equations describing hydrodynamic behavior. The following table describes the applications further:

Application	Length (in million refs)	working set size (in MB)	Language
FFT	7.44	0.25	FORTRAN
SIMPLE	27.03	2.5	FORTRAN
WEATHER	31.76	5.4	FORTRAN
SPEECH	11.77	2.1	Mul-T (Multilisp)

3.1.2 The multiprocessor

We simulated a shared-memory multiprocessor. Each processor has a portion of the shared-memory local to it, but it can also reference the shared-memory of the other processors. Such references are called non-local or remote memory accesses, and are considerably more expensive than local memory accesses. To avoid non-local memory accesses, processors may replicate a page, and map it in the page table as local. Thus, all future accesses to this page will be local. When a processor updates a page, the protocol can either invalidate all other existing copies, or send the update to all other existing copies of the page. We simulate both protocols.

The parameters of the architecture we used are shown in table 1. `Network latency` is the time its takes for a message to do a round-trip travel in the interconnection network of the multiprocessor system. `Local memory access` is the time it takes to access local memory. `Remote memory read` is the time it takes to read a non-local memory location and is the sum of `network latency`, the

parameter	cost (in cycles)
`network latency`	90
`local memory access`	5
`remote memory read`	100
`remote memory write`	10
`cycles to transfer a word in replication mode`	4
`page fault overhead`	500

Table 1: Architecture Overheads

`local memory access`, and some small additional hardware overhead. `Remote memory write` is the time to do a non-local write and is set to a cost slightly higher than the `local memory access`, but significantly lower than the `remote memory read`, because write operations do not have to stall the processor, until they are actually performed [11]. In our simulated system system we assume the existence of a "block transfer" mode like the one that exists in the BBN Butterfly family of large scale multiprocessors [2], where pages and long messages in general are not transferred a word at a time, so the amortized cost for each word transfer is considerably lower than the cost of a remote-memory access. Finally, the `page fault overhead` is the cost of operating system code execution and it does not include to cost of data transfer.

3.1.3 Performance Measurements

The performance criterion we used is normalized average memory access latency. This is the time it takes to do an average memory reference, to the time it takes to do a local memory reference. So, if the normalized average memory access cost is two, then the average memory access took twice as much as it would have taken if all data were in the local memory. Our results report average memory access latency for different page sizes ranging from 64 bytes, to 64 Kbytes. Page size in the horizontal axis of all figures runs in units of log base 2 of the page size. In all our experiments we measure the memory latency for the shared-memory operations only.[†] For non-shared memory operations, memory coherency protocols are not needed as no two processors will ever change a non-shared memory location. Thus, accesses to non-shared memory locations are not entered into averaging. To put the performance of memory coherence protocols in perspective, our performance graphs also include an "all local" and an "all remote" line. The later represents the performance of the protocol that fixes all shared-memory words to a remote memory and *never* replicates. The former represents the ideal case where all shared-memory words are local, and no overhead is needed to make all copies consistent. This, of course, is not a realizable protocol, but serves as a lower bound for any policy.

Good policies should be between the two lines. Although, no policy will ever cross the lower bound, it is very easy for a policy to cross the upper bound. This happens especially when pages are frequently replicated, but all the words in the page are not accessed frequently enough to offset the cost of replication.

3.2 Updates versus Invalidates

We simulated update-based and invalidation-based protocols. The simplest update-based protocol (called UPT) works as follows:

- Suppose processor P wants to read memory location M: If P has a local copy of the page that contains M, it just reads M. Otherwise, it replicates the page that contains M locally. Once a replication is made, P keeps the page that contains M locally forever.

[†]Shared-memory is a region of memory known to the compiler and the operating system.

- When processor P wants to write memory location M: If P has a local copy of the page that contains M, it performs a local write and sends the updated value to all processors that have a copy of the page. The write operation returns as soon as the local write operation completes, and does not wait for all copies to be updated first. If P does not have a local copy of the page, it maps the page remotely, and sends the update to the remote page.

The simplest invalidate-based policy (called INV) works as follows:

- Suppose processor P wants to read memory location M: If P has a local copy of the page that contains M, it just reads M. Otherwise, it replicates the page that contains M locally.

- When processor P wants to write memory location M: P *first* invalidates all replicas of the page that contains M that may exist, and *then* performs the write on M.

One major performance problem with INV, is that is may lead to page bouncing. Because each update operation invalidates all other copies, the next read operation (by a different processor than the writer) will fault, and a new copy of the page will be replicated. If different processors frequently read and write the same page (but not necessarily the same memory location), then each read access that follows an update by another processors will result in a page fault and a page replication. For a page size of 1Kbytes, this overhead may sum up to something more than 1500 cycles. Paying a cost of 1500 cycles (256 words $\times$ 4 cycles/word + 500 cycles OS overhead) for just a read operation seems intolerable, especially given the fact that remote read operations cost 100 cycles, which is an order of magnitude lower. Thus, we augmented the INV policy with a freezing (called INV.FREEZ) mechanism:

> When a processor faults on a read access, it checks to see if the page was recently invalidated. In this case, the page is declared frozen, and will not be replicated again.

If the page was recently invalidated, then (according to the principle of locality) it will probably be invalidated in the near future. Thus, a page replication will probably not offset its cost, as it will probably be followed by a page invalidation.

To adapt to changing reference patterns frozen pages are defrost periodically.

Figures 1–4 present the results of executing the above memory coherence protocols on the described traces.

Fig. 1 presents the performance of the protocols for FFT, and the "all local" and "all remote" lines which demonstrate the performance of the FFT application when all shared data are local and all shared data are remote respectively. We see that nearly all protocols perform very bad. They all perform close to, or much worse than the "all remote" case. This may sound surprising, as the worst behavior that one would expect from a caching policy, is the one where *no* data are local. Unfortunately, this is not so. Memory coherence policies incur a high cost at each replication/migration/invalidation operation. All policies stride to reduce this cost, and maximize the benefits of a page replication, or similar operation. If, for example, a 64K page is replicated, but the processor is going to access only a few elements of the page, then that was a wrong decision, whose cost has already been paid, but whose benefit will never be realized.

The rest of the applications are described in figures 2-4. We see that almost all behave in a similar manner. The INV policy is consistently the worst (at least for large page sizes). The cost of accessing shared memory for the INV policy increases linearly (or superlinearly) with page size. This can be attributed to the following effects:

1. Large page sizes may increase the number of falsely shared locations, which may increase the number of read accesses that fault on a recently invalidated page, which increases the number of times a page is replicated (unless we use freezing).

2. Even when the number of read accesses that follow a write access does not increase with the page size, the cost of replicating a page for each of these accesses increases linearly with the page size.

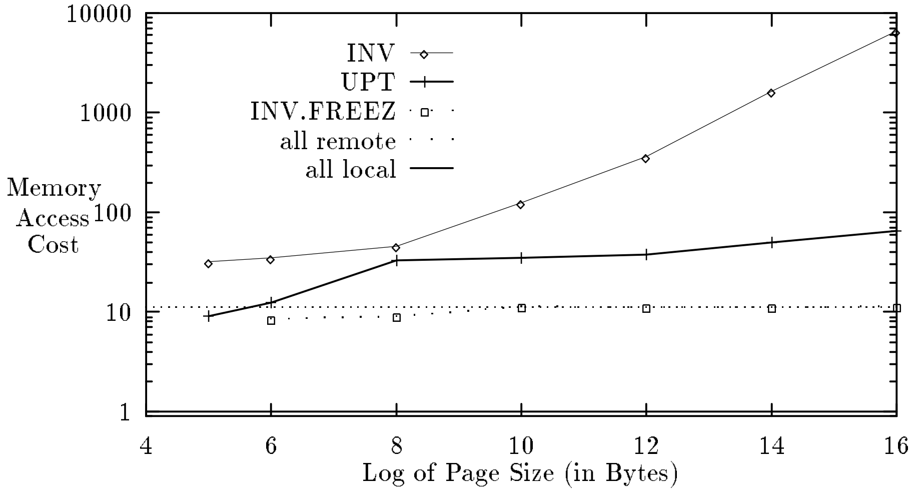

Figure 1: FFT

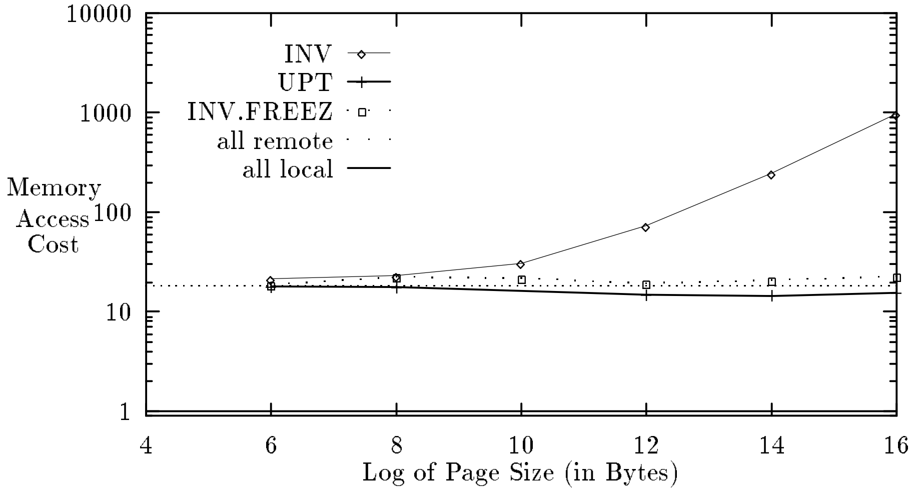

Figure 2: SIMPLEX

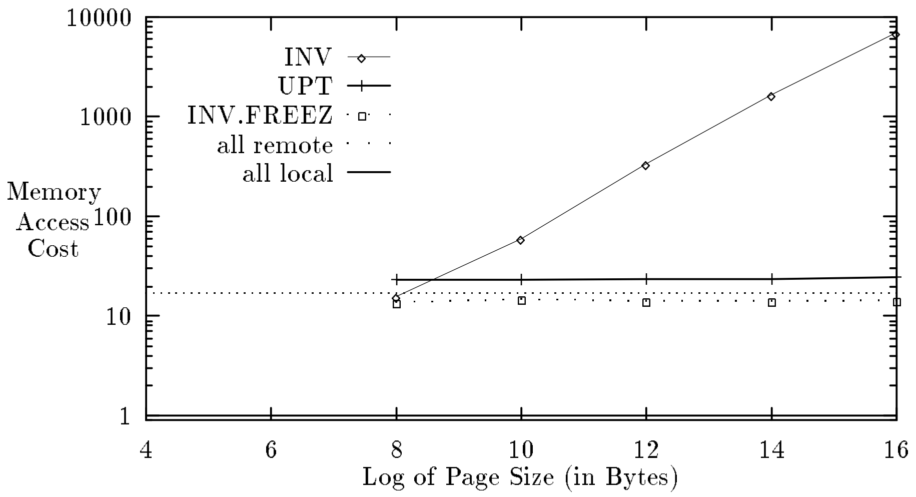

Figure 3: SPEECH

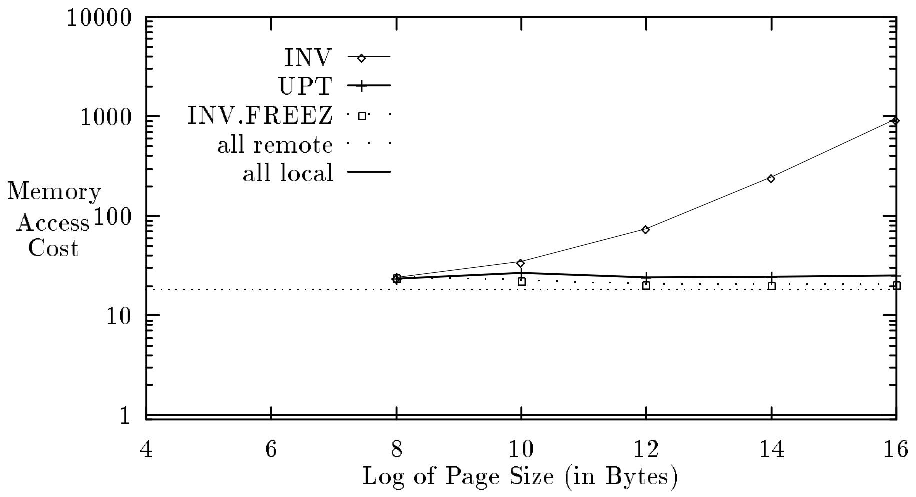

Figure 4: WEATHER

If just the second of the above factors occurred, then the increase in the cost of memory access would be due to the increase in the cost of replication, which is linear to the page size. If both of the above factors occur, system software has to replicate larger pages, more often, which results in a superlinear increase of the memory access cost.

The performance of the INV.FREEZ policy verifies that it is the excessive page replication which is the main source of overhead for the INV policy. Indeed, when pages are frozen in remote memory, the cost of INV.FREEZ does not increase with page size, but, unfortunately, it is close to the "all remote" case. The UPT policy is almost always better than the INV policy, because it avoids page bouncing. It is, however, close to INV.FREEZ, because its does not have a mechanism to avoid sending updates to processors that do not need them any more. INV and INV.FREEZ avoid traffic to processors that do not use pages any more, by invalidating the unused pages on the first write.

3.3 The effect of hardware counters

All the previously studied policies were eager to replicate a page on the first access. There are at least two reasons behind this choice:

- It is easy to make a decision the first time the page is encountered, rather than delegate the decision for later.
- A page can be replicated only when the processor runs operating system code. This code runs as a result of a page fault. If the page is not replicated during the service of the fault, the system has two choices:
  - Map the page remotely: the page table indicates that the page is remote, and all future accesses to the page are remote. This solution may lead to having a page mapped remotely for too long.
  - Do not map the page: the next access to the page will create a new page fault and the system will again have the opportunity, to replicate the page or not. This choice may result in a large number of page-faults before each page is replicated, which may result in a significant operating system overhead.

Bolosky *et. al* [4] have proposed the use of hardware counters, that count accesses to remotely mapped pages. The counter starts from a given value, and when it reaches zero, it sends an interrupt to the operating system. The operating system as a response to the interrupt, replicates the page and maps it locally. In this way, the operating system uses past reference history as an indication of the future reference history, assuming that an application which has made several accesses to a page, will probably make more accesses to it in the near future, so it is worthwhile replicating the page locally. We have adapted all previously described policies to include this optimization. When the application faults for the first time on a page, the operating system sets a counter to one eighth of the size of the page (in bytes). Each time the application access the page, the counter is decremented. When the counter reaches zero, it sends an interrupt, which gives control to the operating system, which replicates the page locally.

Figures 5–8 show the performance of memory coherence protocols both with and without the hardware counter. Protocols that have a .DEL suffix are the same as their counterparts that do not have the suffix, except for their use of the hardware counter to effectively DELay page replication. We generally see that the hardware counter is a good idea. The policies that are augmented with the hardware counter perform much better than their counterparts that do not make use of it. The most spectacular improvement is to the INV policy. For example, in figure 6, for 64K page size, the memory access cost for the SPEECH application, running under the INV protocol is about 7000. The same protocol augmented with the hardware counter results in memory access cost of about 17. These experiments show that hardware counters are especially helpful in bounding the worst performance of the INV policy. Thus, hardware counters can be used as a mechanism to control page bouncing.

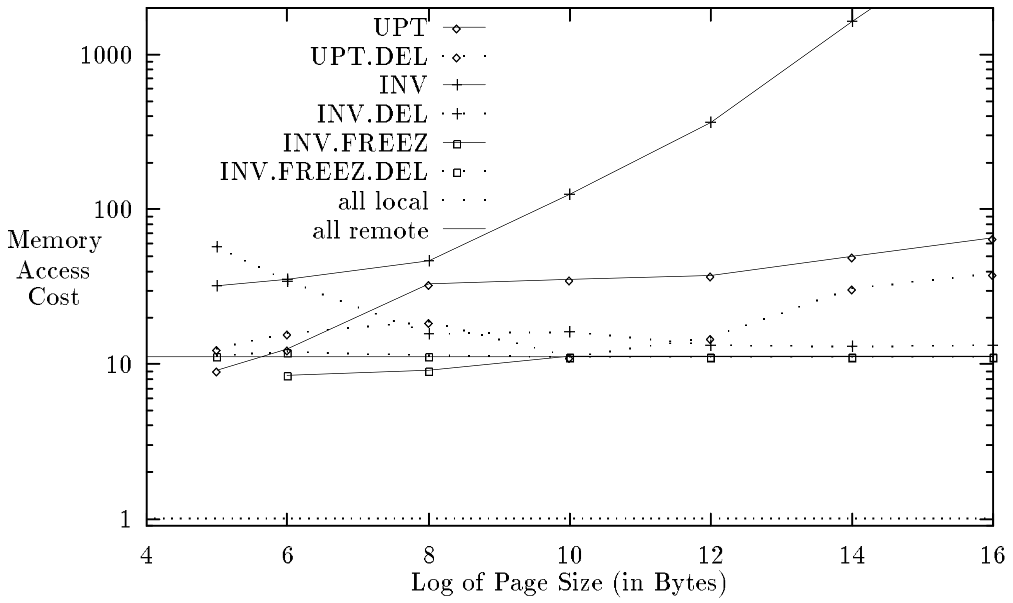

Figure 5: FFT

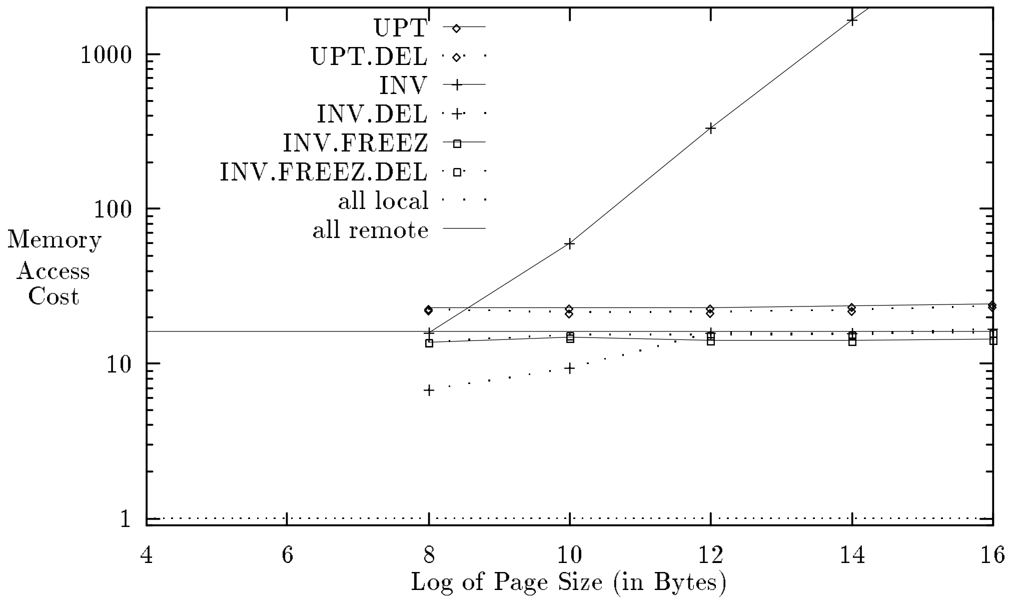

Figure 6: SPEECH

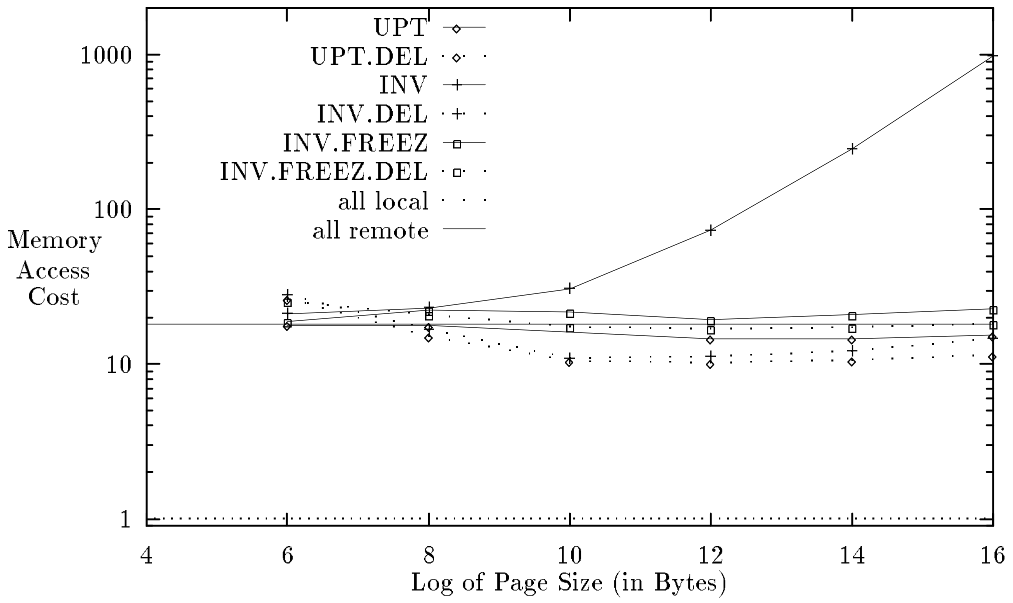

Figure 7: SIMPLEX

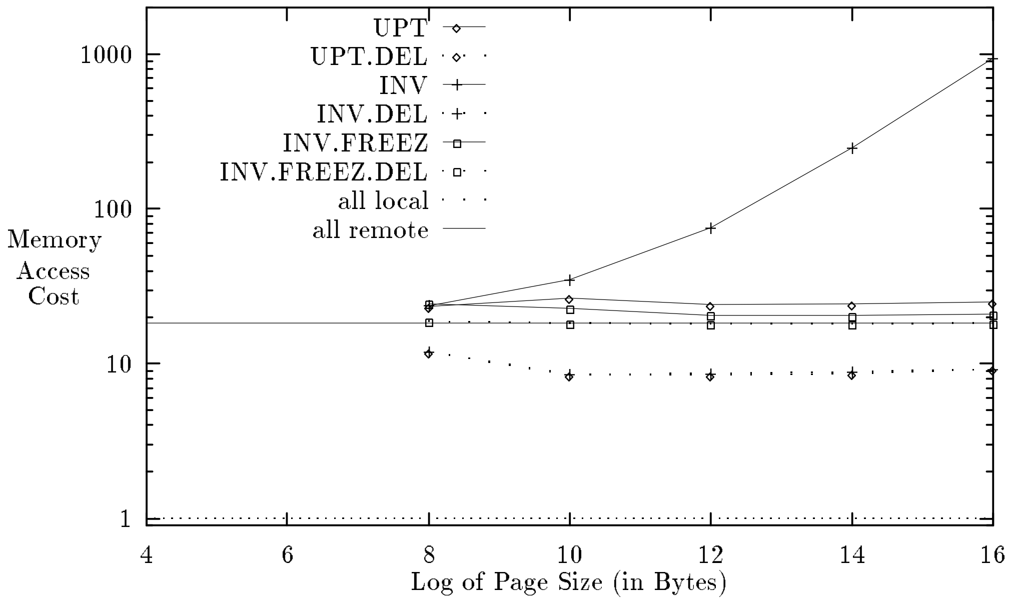

Figure 8: WEATHER

The benefits of hardware counters are even more pronounced in SIMPLEX (figure 7) and WEATHER (figure 8). For those applications, hardware counters, not only bound the worst case behavior of INV, but make INV and UPT attractive as memory coherence protocols. Indeed, figures 7 and 8 show that the performance of INV.DEL and UPT.DEL are somewhere between "all local" and "all remote". Thus, INV.DEL and UPT.DEL are protocols that can be used for effective locality management in shared-memory multiprocessors. They replicate pages transparently, resulting in a low memory access cost (half of the "all remote" protocol).

3.4 The effect of optimal data alignment

Our experiments so far have focused on the performance of a memory coherence protocol on a fixed stream of virtual addresses. Thus, we have quantified what the operating system *alone* can do to manage locality by replicating pages. We have shown that a significant obstacle in managing locality is false sharing, which results in a large number of page invalidations, and/or a large number of updates travelling through the interconnection network. Memory coherence policies cannot eliminate false sharing by themselves. The only way to eliminate false sharing is to change the assignment of data to pages, so that data referenced by different processors end up in different pages. The assignment of data to pages can be changed by the user and the compiler.

In this set of experiments, we evaluate the effect of sophisticated data alignment to the performance of the memory coherence protocols for various page sizes and examine if it is worthwhile to dedicate compile and programmer time to improve data alignment and through it, the performance of the memory coherence protocol.

To answer these questions, we have implemented a data re-alignment policy, that allocates data to pages, so as to reduce false sharing. Our policy is off-line and given a trace, it reorganizes the mapping of data to pages. The data re-alignment policy works as follows:

1. It reads one million memory accesses from the input trace.

2. For the most recently read set of references, it marks all pages referenced.

3. For each page referenced it determines if it may be falsely shared, that is, accessed by more than one processor, and written by at least one processor.

4. For each page that could be falsely shared, it examines each memory location of the page. If the location is accessed by one processor (say A) only, the location is reassigned to a page that has not been used before, and contains reassigned locations accessed by A only. If the location is accessed by more than one processor, then it is truly shared and we do not reassign it.

5. If there are are more references in the input, go to the first step.

Our algorithm is not the best possible for data re-alignment. It is simple enough however, to demonstrate if data re-alignment is an effective way to improve the performance of memory coherence policies. We repeated our trace-driven simulations, after passing the traces through the data re-alignment module.

Figure 9 shows the memory access cost of the FFT application running on top of several memory coherence policies, using traces with re-aligned data. The first thing we notice is that the performance of all policies is remarkably better than that in our previous experiments. For example, the memory access cost for UPT.DEL and INV.DEL for page sizes of 4K is close to 2.2 when data re-alignment is used. Figure 5 suggests that the cost for the same policies without data re-alignment is close to 14.

Figure 9 suggests that almost all policies, for most page sizes are in the acceptable range between "all local" and "all remote". The policies that perform best, are the UPT.DEL and INV.DEL, which are the "vanilla" policies augmented with the hardware counter.

Both freezing policies (INV.FREEZ, INV.FREEZ.DEL) perform close to the "all remote" line. Even though page freezing is generally a good idea, (as figures 1–4 suggest) that it is very weak

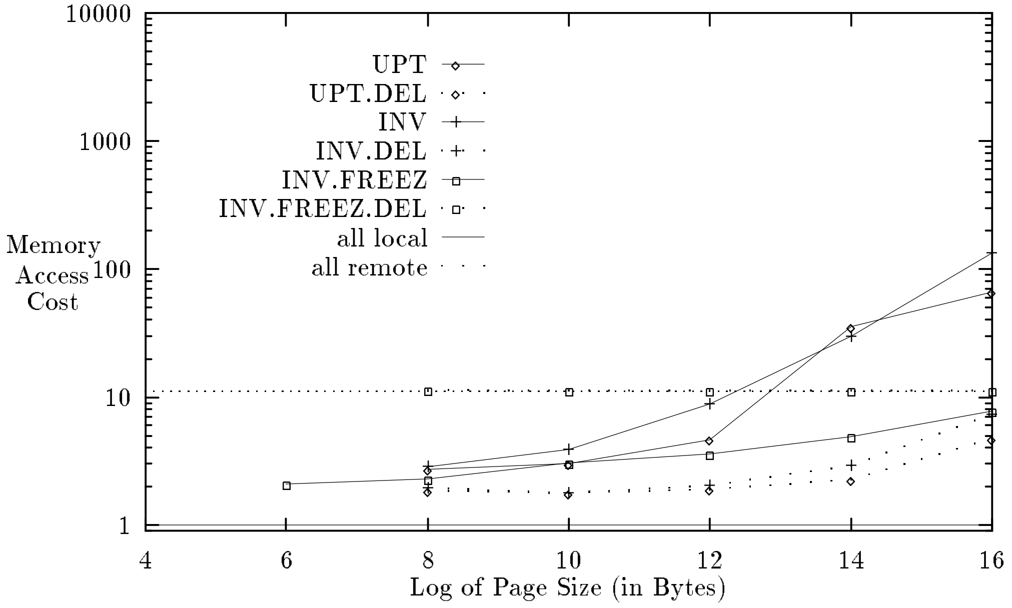

Figure 9: FFT with data re-alignment

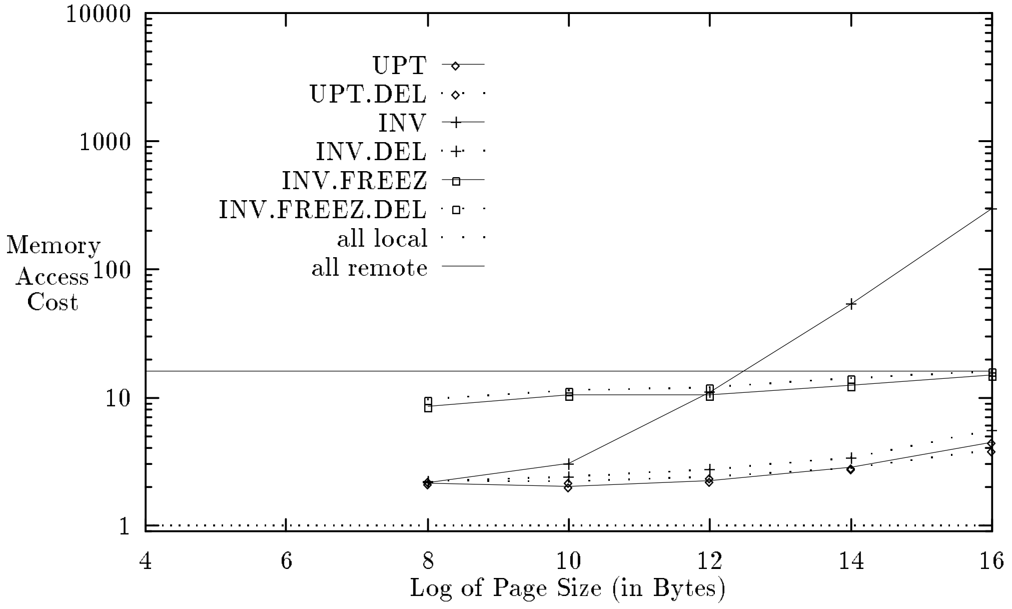

Figure 10: SPEECH with data re-alignment

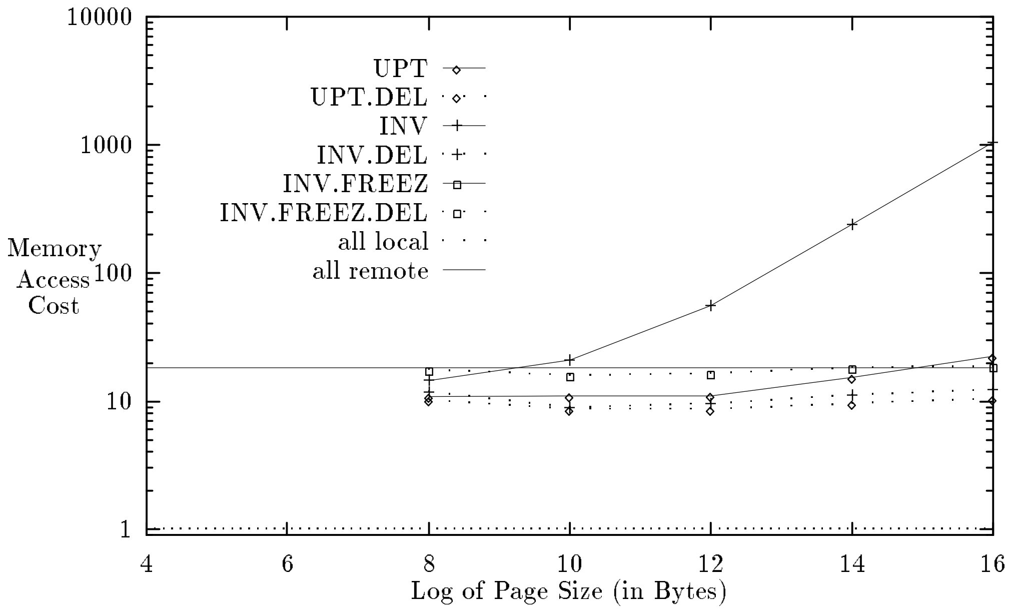

Figure 11: SIMPLE with data re-alignment

when coupled with data re-alignment. This is because the main advantage of freezing policies, namely reduction in the effects of false sharing does not exist anymore, since there is almost no false sharing.

SPEECH in figure 10 has a similar behavior to FFT. When the page size is small all policies, with the exception of the freezing policies, perform well. When the page size gets larger than 4K, UPT, UPT.DEL and INV.DEL are the ones that perform well.

SIMPLE in figure 11 behaves similarly. UPT.DEL and INV.DEL are the best policies. In this figure we can easily notice that the behavior of these policies is not very good for small pages sizes (around 256 bytes), gets better at medium pages sizes (1-4K) and then gets worse again for large page sizes. The reason that they do not perform very well for small page sizes, is that they suffer from high operating system overhead. Large page sizes imply fewer page faults to transfer a page, and therefore less system overhead. WEATHER (figure 12) is no exception; as in the previous cases, INV.DEL and UPT.DEL perform the best.

Summarizing, our experimentation with data-alignment suggests that data re-alignment results in significant performance improvement for all memory coherence protocols.

4 Conclusions

In this paper we considered several factors in the design of memory coherence protocols. We studied the effectiveness of update-based protocols, of invalidate-based protocols, of freezing mechanisms to avoid page-bouncing, of hardware counters to support delayed replication, and of data re-allocation to reduce false sharing.

Based on our simulation results we conclude:

- *Data re-alignment, is the largest single factor of performance improvement.* This result calls for a cooperation between the operating-system and the compiler to exploit locality together.

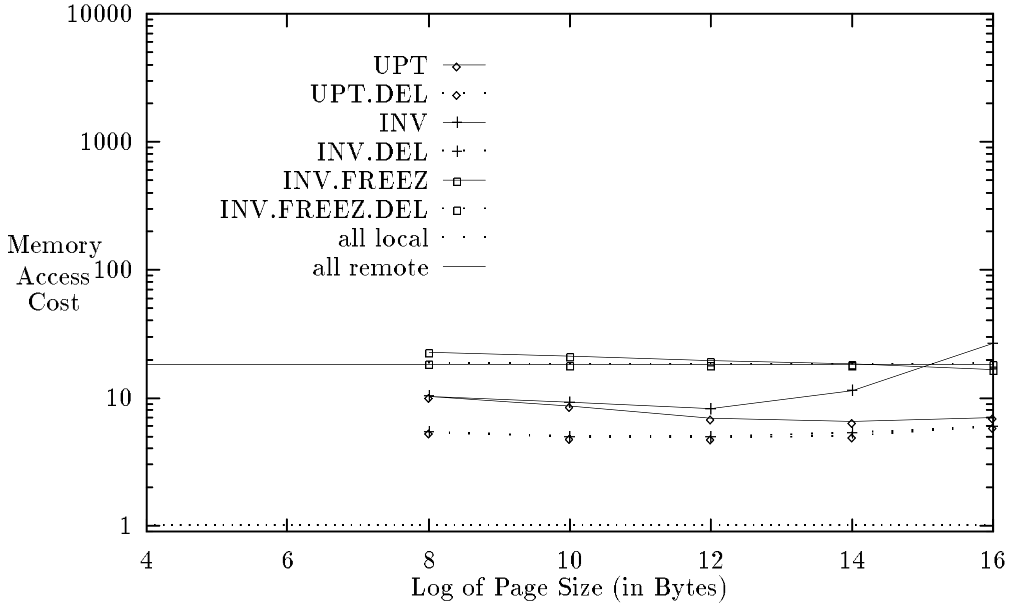

Figure 12: WEATHER with data re-alignment

The operating system alone, even when assisted by hardware mechanisms is not able to exploit locality effectively on an application that has a naive data alignment. On the other hand, the compiler by itself does not know when to replicate data, as this decision depends on the dynamic execution of the application. However, the compiler may have lots of information about the data access patterns of the applications, and place data with different reference patterns on different pages.

Our simulations indicate that the performance improvement that data re-alignment can result in two orders of magnitude of performance improvement for simple invalidate-based policies, and up to an order of magnitude of performance improvement for update-based policies (contrast figures 1 and 9).

- *Hardware counters that support delayed replication are always useful.* Hardware counters are very effective in reducing page bouncing. When coupled with data alignment, hardware counters result in the best performance we observed.

- *Update-based protocols are almost always better than invalidate-based protocols, but the difference between the two is small for the cases of good data re-alignment.* The reason is that for the large page sizes we simulated, it is always better to send the a new value, rather than to invalidate a page, and replicate it on the next access. The very few cases where invalidate-based protocols were better, happened at small page sizes (less than 64 words), which is in agreement with previous results [9].

- *Although freezing mechanisms seem to help, they provide no advantage when combined with (or when compared to) data re-alignment, or hardware counters.* In the absence of hardware support, freezing can be used to reduce excessive replication and updates. Freezing, however, does not cooperate well with other mechanisms like the hardware counters, and the data re-alignment.

The reason is that freezing is too eager to freeze a page, to avoid false sharing. Although this eagerness is good when there is lots of false sharing going on, it is definitely a bad idea when there is almost no false sharing.

In conclusion, our results indicate that locality management is a global system issue and that all system levels should cooperate to exploit. The compiler should perform sophisticated data alignment, the operating system should perform memory caching and coherence, while the hardware should assist the operating system in avoiding unecessary data movement.

Acknowledgements

Manolis G.H. Katevenis contributed several of the ideas described in the paper, and improved their presentation via several helpful discussions. Financial support for this work was provided by the Commission of the European Communities (CEC), through ESPRIT contract P6253 "Supercomputer Highly Parallel System" (SHIPS). We would like to thank D. Chaiken from MIT for providing us the multiprocessor traces.

References

[1] J. Archibald and J.-L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[2] BBN Laboratories. "Butterfly Parallel Processor Overview". Technical Report 6148, BBN Laboratories, Cambridge, MA, March 1986.

[3] W. J. Bolosky and M. L. Scott. "A Trace-Based Comparison of Shared Memory Multiprocessor Architectures". Technical Report 432, University of Rochester, Computer Science Department, July 1992.

[4] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. "NUMA Policies and Their Relation to Memory Architecture". In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[5] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. "Simple But Effective Techniques for NUMA Memory Management". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[6] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. "Directory-Based Cache Coherence in Large-Scale Multiprocessors". *IEEE Computer*, 23(6):49–58, June 1990.

[7] A.L. Cox and R.J. Fowler. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[8] T. H. Dunigan. "Kendall Square Multiprocessor: Early Experiences and Performance". Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.

[9] S. J. Eggers and R. H. Katz. "Evaluation of the Performance of Four Snooping Cache Coherency Pro tocols". In *Proceedings of the 16th International Conference on Computer Architecture*, page 2 15, 1989.

[10] A. Forin, J. Barrera, and R. Sanzi. "The Shared Memory Server". In *Proceedings of the USENIX Winter '89 Technical Conference*, pages 229–244, January 1989.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[12] M. A. Holliday. "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures". In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989.

[13] M. A. Holliday. "On the Effectiveness of Dynamic Page Placement". Technical report, DCS, Duke University, Durham, NC, September 1989.

[14] R.P. LaRowe and C.S. Ellis. "Virtual Page Placement Policies for NUMA Multiprocessors". Technical Report CS-1990-10, Department of Computer Science, Duke University, Dec 1988.

[15] R. P. LaRowe, Jr. and C. S. Ellis. "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors". *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[16] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[17] E.P. Markatos and T.J. LeBlanc. "Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance". Technical Report 420, University of Rochester, Computer Science Department, March 1992.

[18] Sequent Computer Systems Inc. *Balance 8000 System*, 1985.

[19] Sequent Computer Systems Inc. *Symmetry Multiprocessor Architecture Overview*, 1991.

[20] J.E. Veenstra. *Hybrid Cache Coherency Protocols*. PhD thesis, University of Rochester, Computer Science Department. in preparation.

[21] J.E. Veenstra and R.J. Fowler. "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols". In *Proceedings of the fifth ACM Symposium on Architectural Support for Operating Systems and Programming Languages*, pages 149–160, 1992.

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	6	4.832	0.000	5.092	34.955	16.533	0.529	0.000	3.568	12.945
UPT	8	4.842	0.000	10.854	113.002	15.685	2.008	0.000	13.324	36.537
UPT	10	4.959	0.000	8.785	137.454	3.943	2.019	0.000	15.802	32.135
UPT	12	4.989	0.000	8.445	163.629	1.009	2.066	0.000	17.438	35.459
UPT	14	4.996	0.000	9.064	256.414	0.275	2.254	0.000	23.307	47.392
UPT	16	4.999	0.000	12.036	256.414	0.092	3.005	0.000	31.036	62.970
INV	6	2.555	2.277	40.133	0.000	130.646	4.181	0.228	0.000	0
INV	8	2.459	2.400	92.416	0.000	134.056	17.159	0.240	0.000	0
INV	10	0.978	4.203	429.364	0.000	192.066	98.338	0.420	0.000	0
INV	12	0.707	4.716	1622.379	0.000	193.468	396.222	0.472	0.000	0
INV	14	0.142	4.880	7937.827	0.000	241.850	1981.238	0.488	0.000	0
INV	16	0.062	4.917	32499.516	0.000	247.927	8124.070	0.492	0.000	0

Table 2: FFT

A Analytical Figures

In this appendix we provide detailed measurements of our simulations of the INV and UPT policies and their performance when helped by hardware counters and data re-alignment. Each table has 4 major columns:

- *policy:* this is the protocol simulated.
- *page:* this is the log (base two) of the page size in bytes. The page sizes simulated are 256 bytes to 64Kbytes.
- *latency overhead:* This is the average overhead (in cycles) for each memory operation. This overhead is further divided into:
  - *local*: cost of local accesses
  - *remote*: cost of remote accesses
  - *repl.*: cost of replication
  - *multicasts:* cost of sending update packets
  - *os ovhead*: operating system overhead (excluding data transfers)
- *network traffic:* This is the number of network packets (address - datum) generated by each protocol. This traffic is further divided into:
  - *repl.*: traffic due to page replication
  - *remote*: traffic due to remote memory accesses
  - *multicasts:* cost of sending update packets
- *mls:* This is the replication factor of each page that *is* replicated to more than one processor. We count it only for update-based policies, in order to get a feeling of how many update packets each store operation needs to generate.

A.1 Plain protocols

Tables 2-5 show the various statistics for the experiments described in figures 1-4.

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	6	3.997	0.000	18.179	7.462	59.539	1.905	0.034	0.746	13.517
UPT	8	4.204	0.000	24.963	19.629	36.214	4.635	0.029	2.272	38.250
UPT	10	4.415	0.000	28.254	35.482	12.687	6.496	0.023	3.567	57.144
UPT	12	4.502	0.000	27.139	39.147	3.242	6.640	0.021	3.799	59.181
UPT	14	4.525	0.000	27.934	41.037	0.848	6.945	0.019	3.918	60.271
UPT	16	4.541	0.000	30.499	43.945	0.232	7.614	0.017	4.102	61.919
INV	6	3.750	0.593	23.747	0.000	77.855	2.491	0.059	0.000	0
INV	8	3.838	0.612	45.861	0.000	65.899	8.435	0.061	0.000	0
INV	10	3.987	0.634	103.190	0.000	46.214	23.661	0.063	0.000	0
INV	12	3.994	0.772	322.551	0.000	38.452	78.750	0.077	0.000	0
INV	14	4.002	0.801	1180.912	0.000	35.964	294.615	0.080	0.000	0
INV	16	3.991	0.807	4858.252	0.000	37.061	1214.409	0.081	0.000	0

Table 3: SIMPLE

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.257	0.000	1.829	22.820	2.644	0.338	0.002	10.518	62.170
UPT	10	4.269	0.000	2.664	22.820	1.196	0.612	0.002	10.699	62.048
UPT	12	4.276	0.000	3.097	22.820	0.370	0.758	0.001	10.771	62.113
UPT	14	4.280	0.000	4.100	22.820	0.124	1.020	0.001	10.864	62.270
UPT	16	4.283	0.000	7.208	22.820	0.055	1.800	0.000	10.990	60.817
INV	8	3.926	0.019	30.709	0.000	43.973	5.629	0.002	0.000	0
INV	10	3.545	0.019	204.319	0.000	91.626	46.913	0.002	0.000	0
INV	12	2.723	0.343	1464.099	0.000	178.174	364.901	0.034	0.000	0
INV	14	2.006	0.765	8082.520	0.000	246.624	2020.343	0.076	0.000	0
INV	16	1.826	0.878	34546.178	0.000	263.564	8636.472	0.088	0.000	0

Table 4: SPEECH

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.020	0.000	41.780	10.394	60.175	7.702	0.042	1.090	17.404
UPT	10	4.315	0.000	65.820	30.239	29.464	15.086	0.029	3.366	47.803
UPT	12	4.519	0.000	64.226	46.042	7.666	15.701	0.022	4.361	57.613
UPT	14	4.589	0.000	65.168	53.513	1.978	16.200	0.016	4.988	61.628
UPT	16	4.608	0.000	67.823	53.513	0.517	16.932	0.014	5.168	62.661
INV	8	3.806	0.799	47.238	0.000	67.927	8.695	0.080	0.000	0
INV	10	3.907	0.857	116.958	0.000	52.424	26.841	0.086	0.000	0
INV	12	3.982	0.926	331.997	0.000	39.596	81.093	0.093	0.000	0
INV	14	4.002	0.940	1195.950	0.000	36.415	298.316	0.094	0.000	0
INV	16	4.007	0.943	4666.988	0.000	35.601	1166.578	0.094	0.000	0

Table 5: WEATHER

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT.DEL	6	4.502	3.307	5.092	32.855	33.065	0.529	0.066	3.272	11.890
UPT.DEL	8	2.666	21.957	9.547	31.814	27.593	1.766	0.439	2.962	8.628
UPT.DEL	10	2.562	24.203	5.924	17.423	5.318	1.361	0.482	1.502	3.112
UPT.DEL	12	2.558	24.745	1.435	42.162	0.343	0.351	0.488	4.228	9.131
UPT.DEL	14	2.532	24.944	4.642	103.849	0.282	1.154	0.493	11.936	29.926
UPT.DEL	16	2.495	25.131	8.024	134.353	0.122	2.003	0.501	15.359	62.814
INV.DEL	6	2.443	16.053	20.355	0.000	132.177	2.115	0.379	0.000	0
INV.DEL	8	2.036	29.771	11.928	0.000	34.473	2.206	0.558	0.000	0
INV.DEL	10	0.087	52.736	14.291	0.000	12.829	3.284	0.970	0.000	0
INV.DEL	12	0.033	54.498	8.813	0.000	2.105	2.156	0.991	0.000	0
INV.DEL	14	0.007	55.049	9.096	0.000	0.552	2.261	0.998	0.000	0
INV.DEL	16	0.019	54.788	10.719	0.000	0.163	2.676	0.996	0.000	0

Table 6: FFT with hardware counters

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT.DEL	6	3.577	11.908	14.499	6.952	94.866	1.518	0.163	0.672	13.296
UPT.DEL	8	2.915	32.113	8.503	6.338	24.617	1.575	0.392	0.577	29.815
UPT.DEL	10	2.700	37.278	5.269	1.641	4.730	1.211	0.455	0.092	8.562
UPT.DEL	12	2.629	38.716	5.293	1.842	1.264	1.295	0.473	0.110	14.091
UPT.DEL	14	2.502	41.202	5.257	2.513	0.319	1.307	0.499	0.176	23.534
UPT.DEL	16	2.369	43.462	6.825	2.990	0.104	1.704	0.526	0.221	39.241
INV.DEL	6	3.351	14.544	16.162	0.000	105.809	1.693	0.209	0.000	0
INV.DEL	8	2.570	38.230	10.736	0.000	31.106	1.991	0.455	0.000	0
INV.DEL	10	2.596	40.091	5.915	0.000	5.310	1.359	0.475	0.000	0
INV.DEL	12	2.344	44.654	6.802	0.000	1.625	1.664	0.530	0.000	0
INV.DEL	14	2.100	49.440	7.237	0.000	0.439	1.799	0.580	0.000	0
INV.DEL	16	1.582	59.056	10.763	0.000	0.164	2.687	0.683	0.000	0

Table 7: SIMPLE with hardware counters

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT.DEL	8	4.161	2.616	0.730	22.820	2.111	0.135	0.029	10.199	61.573
UPT.DEL	10	4.098	4.321	0.806	22.820	0.724	0.185	0.046	9.694	59.032
UPT.DEL	12	4.051	5.496	1.079	22.820	0.258	0.264	0.058	9.688	56.284
UPT.DEL	14	3.952	7.715	1.567	22.820	0.095	0.390	0.083	9.691	56.654
UPT.DEL	16	3.729	12.342	2.349	22.820	0.036	0.586	0.139	9.865	57.725
INV.DEL	8	3.778	10.810	4.743	0.000	13.740	0.879	0.113	0.000	0
INV.DEL	10	3.158	26.036	8.621	0.000	7.748	1.984	0.274	0.000	0
INV.DEL	12	1.413	56.448	16.903	0.000	4.037	4.134	0.713	0.000	0
INV.DEL	14	1.241	60.119	17.023	0.000	1.033	4.232	0.751	0.000	0
INV.DEL	16	0.952	66.170	17.937	0.000	0.273	4.478	0.809	0.000	0

Table 8: SPEECH with hardware counters

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT.DEL	8	2.864	30.939	5.901	2.245	17.055	1.092	0.410	0.131	14.589
UPT.DEL	10	2.646	36.401	0.663	1.030	0.595	0.152	0.470	0.006	57.063
UPT.DEL	12	2.631	36.738	0.809	1.022	0.193	0.198	0.474	0.006	60.604
UPT.DEL	14	2.593	37.476	0.801	1.017	0.049	0.199	0.481	0.005	63.000
UPT.DEL	16	2.449	40.234	1.099	0.983	0.017	0.274	0.510	0.002	25.215
INV.DEL	8	2.824	32.613	5.982	0.000	17.288	1.106	0.418	0.000	0
INV.DEL	10	2.632	37.930	0.745	0.000	0.669	0.171	0.473	0.000	0
INV.DEL	12	2.603	38.542	0.950	0.000	0.227	0.232	0.479	0.000	0
INV.DEL	14	2.533	39.929	1.077	0.000	0.065	0.268	0.493	0.000	0
INV.DEL	16	2.440	41.733	1.099	0.000	0.017	0.274	0.512	0.000	0

Table 9: WEATHER with hardware counters

A.2 Protocols with delayed replication

Tables 6-9 show the various statistics for the experiments described in figures 5-8.

A.3 Protocols data re-alignment

Tables 10-13 show the various statistics for the experiments described in figures 9-12.

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.950	0.000	3.162	0.924	4.569	0.585	0.001	0.092	13.798
UPT	10	4.972	0.000	5.683	1.814	2.551	1.306	0.001	0.181	28.472
UPT	12	4.981	0.000	14.191	2.385	1.695	3.471	0.000	0.238	37.477
UPT	14	4.983	0.000	52.824	101.019	1.603	13.134	0.000	11.826	62.110
UPT	16	4.983	0.000	209.505	101.019	1.596	52.303	0.000	11.826	62.110
INV	8	4.915	0.059	3.849	0.000	5.562	0.712	0.006	0.000	0.000
INV	10	4.922	0.064	10.129	0.000	4.546	2.328	0.006	0.000	0.000
INV	12	4.925	0.067	35.098	0.000	4.192	8.586	0.007	0.000	0.000
INV	14	4.924	0.067	140.124	0.000	4.253	34.841	0.007	0.000	0.000
INV	16	4.924	0.067	557.302	0.000	4.246	139.131	0.007	0.000	0.000
UPT.DEL	8	4.709	2.784	1.077	0.355	3.113	0.199	0.055	0.008	1.262
UPT.DEL	10	4.619	3.933	1.180	0.576	1.059	0.271	0.075	0.022	3.474
UPT.DEL	12	4.559	4.864	1.340	0.606	0.320	0.328	0.088	0.021	3.410
UPT.DEL	14	4.525	5.409	1.039	0.469	0.063	0.258	0.095	0.006	1.000
UPT.DEL	16	3.613	14.524	4.012	1.381	0.061	1.002	0.277	0.006	1.000
INV.DEL	8	4.698	3.249	1.170	0.000	3.381	0.216	0.057	0.000	0.000
INV.DEL	10	4.608	4.487	1.213	0.000	1.089	0.279	0.077	0.000	0.000
INV.DEL	12	4.553	5.361	1.340	0.000	0.320	0.328	0.089	0.000	0.000
INV.DEL	14	4.281	8.502	1.825	0.000	0.111	0.454	0.144	0.000	0.000
INV.DEL	16	2.635	26.599	7.146	0.000	0.109	1.784	0.473	0.000	0.000

Table 10: FFT with data re-alignment

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.271	0.000	18.635	5.917	27.034	3.460	0.031	0.561	18.240
UPT	10	4.411	0.000	30.228	11.108	13.568	6.947	0.023	1.213	34.238
UPT	12	4.528	0.000	36.190	13.351	4.322	8.851	0.012	1.504	32.982
UPT	14	4.575	0.000	51.578	18.301	1.565	12.822	0.006	2.130	30.757
UPT	16	4.593	0.000	72.791	31.367	0.555	18.172	0.003	3.330	42.258
INV	8	4.121	0.392	28.711	0.000	41.537	5.317	0.039	0.000	0.000
INV	10	4.217	0.351	70.486	0.000	31.541	16.149	0.035	0.000	0.000
INV	12	4.256	0.333	230.839	0.000	27.596	56.516	0.033	0.000	0.000
INV	14	4.214	0.294	1143.586	0.000	34.825	285.284	0.029	0.000	0.000
INV	16	4.186	0.269	5193.959	0.000	39.622	1298.336	0.027	0.000	0.000
UPT.DEL	8	3.390	23.501	6.419	2.269	18.557	1.188	0.286	0.180	10.690
UPT.DEL	10	3.144	30.569	4.536	1.519	4.072	1.042	0.362	0.103	6.271
UPT.DEL	12	3.047	33.021	5.121	1.622	1.223	1.253	0.389	0.112	6.770
UPT.DEL	14	2.806	38.699	6.236	1.679	0.379	1.551	0.438	0.127	4.400
UPT.DEL	16	2.572	42.332	4.570	1.230	0.070	1.141	0.485	0.073	7.851
INV.DEL	8	3.221	27.378	7.853	0.000	22.725	1.454	0.325	0.000	0.000
INV.DEL	10	3.071	32.783	5.089	0.000	4.569	1.170	0.380	0.000	0.000
INV.DEL	12	2.856	37.081	6.153	0.000	1.470	1.505	0.427	0.000	0.000
INV.DEL	14	2.287	48.033	8.375	0.000	0.508	2.082	0.542	0.000	0.000
INV.DEL	16	2.124	51.380	6.563	0.000	0.100	1.638	0.575	0.000	0.000

Table 11: SIMPLE with data re-alignment

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.257	0.000	1.845	1.157	2.666	0.341	0.002	0.114	22.628
UPT	10	4.269	0.000	2.707	1.242	1.215	0.622	0.002	0.123	22.354
UPT	12	4.277	0.000	3.252	2.486	0.389	0.796	0.001	0.247	25.406
UPT	14	4.281	0.000	4.549	4.172	0.138	1.131	0.001	0.449	36.101
UPT	16	4.283	0.000	10.081	6.423	0.077	2.517	0.000	0.731	13.021
INV	8	4.250	0.019	2.409	0.000	3.482	0.446	0.002	0.000	0.000
INV	10	4.253	0.018	7.096	0.000	3.188	1.632	0.002	0.000	0.000
INV	12	4.236	0.017	45.017	0.000	5.372	11.002	0.002	0.000	0.000
INV	14	4.216	0.016	258.415	0.000	7.851	64.317	0.002	0.000	0.000
INV	16	4.189	0.017	1468.570	0.000	11.202	367.071	0.002	0.000	0.000
UPT.DEL	8	4.160	2.638	0.738	0.820	2.134	0.137	0.029	0.079	30.495
UPT.DEL	10	4.096	4.361	0.811	0.328	0.728	0.186	0.047	0.030	11.452
UPT.DEL	12	4.049	5.554	1.071	0.426	0.256	0.262	0.059	0.039	17.095
UPT.DEL	14	3.955	7.622	1.367	0.350	0.083	0.340	0.083	0.029	9.328
UPT.DEL	16	3.735	12.056	2.477	0.343	0.038	0.618	0.138	0.020	18.310
INV.DEL	8	4.147	2.918	0.851	0.000	2.458	0.157	0.032	0.000	0.000
INV.DEL	10	4.060	5.201	1.064	0.000	0.955	0.245	0.055	0.000	0.000
INV.DEL	12	3.979	7.165	1.443	0.000	0.345	0.353	0.076	0.000	0.000
INV.DEL	14	3.846	10.109	2.053	0.000	0.125	0.510	0.110	0.000	0.000
INV.DEL	16	3.450	18.791	4.519	0.000	0.069	1.128	0.209	0.000	0.000

Table 12: SPEECH with data re-alignment

policy	page	latency overhead					network traffic			mls
		local	remote	repl.	multicasts	os ovhead	repl.	remote	multicasts	
UPT	8	4.412	0.000	19.272	0.167	27.956	3.578	0.008	0.008	62.907
UPT	10	4.570	0.000	27.022	0.089	12.144	6.218	0.001	0.008	63.000
UPT	12	4.643	0.000	26.519	0.085	3.169	6.489	0.000	0.008	63.000
UPT	14	4.662	0.000	27.190	0.084	0.825	6.761	0.000	0.008	63.000
UPT	16	4.667	0.000	30.142	0.084	0.230	7.525	0.000	0.008	63.000
INV	8	4.407	0.084	19.630	0.000	28.476	3.645	0.008	0.000	0.000
INV	10	4.563	0.007	28.665	0.000	12.877	6.593	0.001	0.000	0.000
INV	12	4.637	0.002	32.666	0.000	3.902	7.991	0.000	0.000	0.000
INV	14	4.656	0.002	51.360	0.000	1.559	12.768	0.000	0.000	0.000
INV	16	4.661	0.001	126.390	0.000	0.963	31.554	0.000	0.000	0.000
UPT.DEL	8	3.860	17.249	1.925	0.289	5.562	0.356	0.197	0.006	56.562
UPT.DEL	10	3.823	18.963	1.387	0.241	1.245	0.319	0.210	0.006	56.400
UPT.DEL	12	3.811	19.404	1.510	0.227	0.361	0.369	0.214	0.005	63.000
UPT.DEL	14	3.786	20.205	1.511	0.208	0.092	0.376	0.221	0.005	63.000
UPT.DEL	16	3.615	23.359	1.936	0.275	0.030	0.483	0.264	0.002	25.215
INV.DEL	8	3.853	17.637	1.988	0.000	5.746	0.368	0.198	0.000	0.000
INV.DEL	10	3.812	19.426	1.469	0.000	1.319	0.338	0.213	0.000	0.000
INV.DEL	12	3.789	20.117	1.651	0.000	0.394	0.404	0.220	0.000	0.000
INV.DEL	14	3.738	21.525	1.787	0.000	0.108	0.444	0.233	0.000	0.000
INV.DEL	16	3.607	23.777	1.936	0.000	0.030	0.483	0.265	0.000	0.000

Table 13: WEATHER with data re-alignment