Trace-Driven Simulation of Data-Alignment and other Factors affecting Update and Invalidate Based Coherent Memory

Evangelos P. Markatos
markatos@ics.forth.gr

Catherine E. Chronaki
chronaki@ics.forth.gr

Institute of Computer Science, FO.R.T.H.,
P.O. Box. 1385, Heraclion, Crete, Hellas GR-711 10

To appear in Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'94), January 31 – February 2, 1994 Durham, North Carolina, USA

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.

KEYWORDS: trace-driven simulation, locality management, memory coherence, data alignment.

1 Introduction

One way to exploit locality of reference in shared memory multiprocessors is by bringing data close to processors that reference them. This may be accomplished by bringing data to local caches and keeping all caches coherent via a cache coherence protocol at the hardware level, replicating pages close to processors at the operating system level, and explicitly copying data in the parallel program at the user level.

Although bus-based multiprocessors and several large-scale multiprocessors use hardware coherent caches, software coherence implemented in the operating system/run time system has started to become more attractive lately due to a number of reasons: First, hardware protocols start to lose their advantage of *speed*: As processors are getting faster than memories the cost of operating system code execution is getting lower when compared to the cost of data transfer. Second, software protocols can afford to be more sophisticated than hardware protocols, since they can be easier to *debug* and *change*. Third, although operating system level protocols manage locality transparently the user may supply *hints* to the compiler or the operating system. Finally, hardware protocols have a *high implementation cost*, especially when large directory structures are needed.

Page replication, the fundamental mechanism for exploiting locality in the operating system, may result to up to an order of magnitude performance improvement, since local memory access is 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. 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 some of these issues:

Update-based vs Invalidate-based protocols. When shared data have been replicated to several processors, and a processor changes its local copy, all other copies must be updated to preserve memory consistency. This update can be done using one of two basic mechanisms:

(a) *Invalidate* all copies of the page where the datum resides, except for the writer's. When another processor attempts 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.

Invalidate-based mechanisms are simple and can be implemented completely in software. Their main drawback is that the unit of invalidation is the page. If pages are large, invalidations may result in frequent page faults and page replication. Update-based mechanisms are more complicated and need some hardware support to perform non-local updates.

The results of our experiments indicate that protocols based on updates are almost always better than those based on invalidates (see section 4.1).

The role of hardware counters. Hardware counters may be used to assist the system software in making informed decisions about replicating pages.

Several hardware protocols replicate a page the first time it is accessed. What we would prefer though, is to replicate a page, only if we know that the processor will use it frequently in the future. Bolosky [2] proposes the use of a hardware counter that counts the number of non-local accesses to the page, as 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.

We show that hardware counters improve the performance of coherence protocols and provide a useful mechanism for eliminating page bouncing (see section 4.2).

Reducing false sharing via data re-alignment Since traditional compilers do not perform any sophisticated data alignment, data with different access patterns might all end up in the same page, introducing significant false sharing.

If a page is falsely shared, the memory coherence protocol has no good solution. If, for example, two processors read and write two different memory locations which happen to be on the same page and the page is replicated to both processors, then a large number of updates (or invalidates) will travel over the communication network. If the page is not replicated to both processors, one processor will make all its operations remotely. The *only* low overhead solution in this case is to allocate the data that correspond to the two different locations on different pages.

Our experiments suggest that sophisticated data alignment is the single most important source of performance improvement (see section 4.3).

The next section provides and overview of previous work and places our contributions in context. Section 3 describes our experimental environment and section 4 presents our experiments. Finally, section 5 summarizes our results and presents our conclusions.

2 Previous Work

There exist several implementations of memory coherence protocols in shared-memory multiprocessors [3, 2, 7]. A lot of work has been done in implementing coherent memory in message-passing multiprocessors [8]. Bolosky et. al [2] have compared several memory coherence policies. They have shown that the dominant overhead in memory coherence policies is false sharing, which can be reduced by decreasing the page size. Eggers and Katz [6] have evaluated the performance trade-offs of invalidate and update-based cache-coherence protocols, for small scale bus-based multiprocessors with snooping caches. Their results suggested that each protocol outperforms the others under the appropriate conditions. Veenstra and Fowler [10] have compared invalidate and update-based cache-coherence protocols. They concluded that for large cache lines update-based protocols are better and suggested that a hybrid protocol that combines both updates and invalidates is always the best.

Our work differs from previous work in memory coherence protocols in several aspects. We measure how much of the memory coherence overhead is *inherent* to the application and how much can be attributed to naive data alignment. We extend previous work [2] on the effects of hardware counters by quantifying the interaction of hardware counters with sophisticated data alignment and update-based policies. The relative trade-offs of update-based and invalidation-based protocols have been studied [6] in the domain of small scale hardware cache-coherent bus-based multiprocessors. In this paper we study the alternatives for *large-scale* multiprocessors that support software coherence. Finally, our study applies to virtual shared memory systems where caching and coherence 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 Experimental Environment

The traces (Memory Access Patterns): We use trace-driven simulation to evaluate the tradeoffs in the

Application	Length (10^6 refs)	Working set (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	Multilisp

Table 1: Applications used in trace-driven simulation.

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

Table 2: Architecture Overheads.

design of a memory coherency protocol. Trace-driven simulation consists of feeding the application traces into a memory coherence protocol simulator which computes cost metrics of the overheads. For each trace record:

1. Examines the type of access (read or write) and the address to be accessed.
2. Simulates the appropriate actions of the protocol.
3. Updates the cost metrics.

The traces we use are 64-processor traces gathered from four programs (see table 1): FFT, SIMPLE, WEATHER, and SPEECH [4]. FFT is a fast Fourier transform. SIMPLE models the behavior of fluids and solves equations describing hydrodynamic behavior. SPEECH is the lexical decoding stage of a phonetically based spoken language understanding system. WEATHER is an application for weather simulation.

Shared-memory Multiprocessor: Each processor has a portion of the shared-memory local to it (local access), but it can also reference the shared-memory local to other processors (remote memory access). To avoid remote memory accesses, processors may replicate a page and map it in the page table as local, making all future accesses to this page local.

The parameters of the architecture are shown in table 2. `Network latency` is the time it takes for a message to do a round-trip in the interconnection network of the multiprocessor. `Remote memory read` is the sum of `network latency`, `local memory access`, and some small hardware overhead. `Remote memory write` is slightly higher than `local memory access`, but significantly lower than `remote memory read`, because write operations do not have to stall the processor We assume the existence of a "block transfer" mode like the one in the BBN Butterfly family of large scale multiprocessors [1], where pages and long messages are transferred in blocks rather than a word at a time. In this way the amortized cost for word transfer is considerably lower than that of a remote memory access. Local and remote writes may result in sending updates to all copies of a page. The `update` cost is set to 10 cycles. `Page fault overhead` is the cost of operating system code execution and it does not include to cost of data transfer.

Performance Measurement: The performance metric we use is Normalized Average Memory access Cost, $NAMC$; the time it takes to do an average memory reference, to the time it takes to do a local memory reference. If the $NAMC$ cost is 2, then the average memory access is twice the time it would have taken if all data were in the local memory. Our results report $NAMC$ 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_2 of the page size.

To put the performance of memory coherence protocols into perspective, our performance graphs also include an "all remote" line. The "all remote" case represents the performance of the protocol that fixes all shared-memory words to a remote memory and *never* replicates. Good policies should lie below the "all remote" line. Nevertheless, it is common for a policy to cross the "all remote" line, as in the case where although pages are frequently replicated, they are not accessed frequently enough to offset the cost of their replication.

4 Experiments

4.1 Updates versus Invalidates

UPT is the basic update-based protocol we simulate:

- **UPT Read:** If a processor wants to read a memory location M and it has a local copy of M's page it just reads M. Otherwise, it replicates M's page locally, never to be unreplicated.

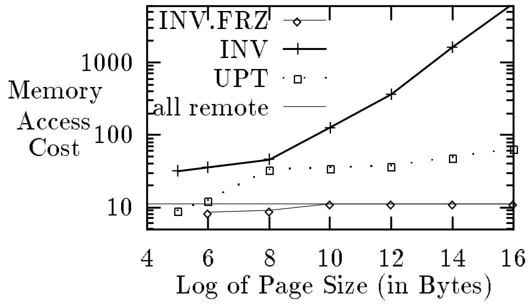

Figure 1: FFT for the basic protocols.

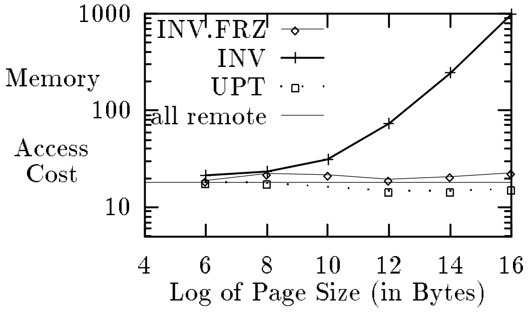

Figure 2: SIMPLE for the basic protocols.

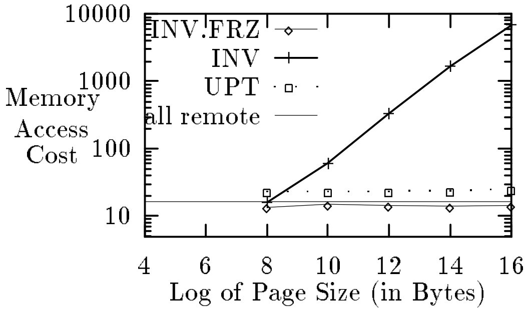

Figure 3: SPEECH for the basic protocols.

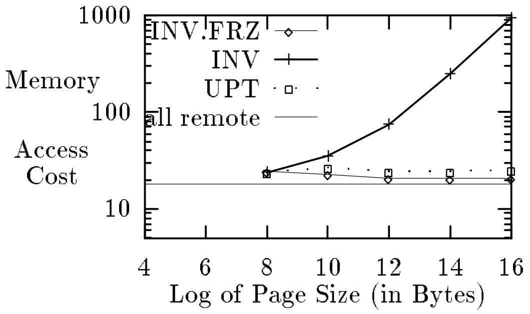

Figure 4: WEATHER for the basic protocols.

• **UPT Write:** If a processor wants to write a memory location M and it has a local copy of M's page, it writes M locally and sends the new value of M, to all processors with a copy of M's page. Otherwise, it does a remote memory access. All updates by all processors are appropriately serialized so that all copies of a page are consistent.

INV is the basic invalidate-based protocol we simulate:

• **INV Read:** If processor wants to read a memory location M and it has a local copy of M's page, it just reads M. Otherwise, it replicates the page locally.

• **INV Write:** If a processor wants to write a memory location M, it *first* invalidates all copies of M's page, and *then* writes M locally.

One major performance shortcoming of INV is that it may lead to page bouncing in the presence of false sharing. To alleviate this shortcoming we augmented the INV protocol with a freezing/defrost mechanism first proposed in [5]. The revised protocol is called INV.FRZ:

• **INV.FRZ Read:** If the processor wants to read a memory location M and it has a local copy of its page, it just reads M. Otherwise, if M's page has been recently invalidated, then M's page is frozen[1]. If M's page is not frozen, it is replicated locally. If the processor has a local copy of M's page it reads M locally, otherwise it does a remote access of M.

• **INV.FRZ Write:** Same as INV Write.

Figures 1–4 present the results of executing the UPT, INV, and INV.FRZ memory coherence protocols. All protocols perform close to, or worse than the "all remote" case. The INV protocol is consis-

[1] A frozen page cannot be replicated. Frozen pages are defrost periodically.

tently the worst. The cost of accessing shared memory for the INV protocol increases superlinearly with page size. This can be attributed to two effects: First, larger page size may lead to more false sharing, which will in turn increase page replication. Second, the cost of page replication increases linearly with the page size. In the presence of both effects, the system software has to replicate larger pages more often which results in superlinear increase of the memory access cost.

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

4.2 The effect of hardware counters

The protocols in the previous section were eager to replicate a page on the first access. Bolosky et. al [2] have proposed the use of hardware counters to count accesses to remotely mapped pages. The counter counts down from a given value and when it reaches zero, it sends an interrupt to the operating system to replicate the page and map it locally. In this way, the operating system uses past reference history as an indication of the future, assuming that an application which has made several accesses to a page in the past, 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 protocols to include this optimization. When a processor faults on a page for the first time, the operating system sets a counter to one eighth of the page size. Every time the processor 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. The revised protocols have a .DEL suffix to indicate the use of a hardware counter to effectively DELay page replication.

Figures 5–8 show the performance of Delay memory coherence protocols. The most spectacular improvement occurs to the INV protocol, indicating that hardware counters can be used as a mechanism to control page bouncing. For example, in figure 3, for 64K page size, the memory access cost for the SPEECH application, running under the INV protocol is about 7000. The same application in figure 7 running under the INV.DEL protocol for 64K page size, incurs memory access cost of about 17! For the SIMPLE (figure 7) and WEATHER (figure 8) applications, hardware counters not only bound the worst case behavior of INV, but make INV.DEL and UPT.DEL attractive as memory coherence protocols. They replicate pages transparently, resulting in a low memory access cost (half of the "all remote" protocol cost).

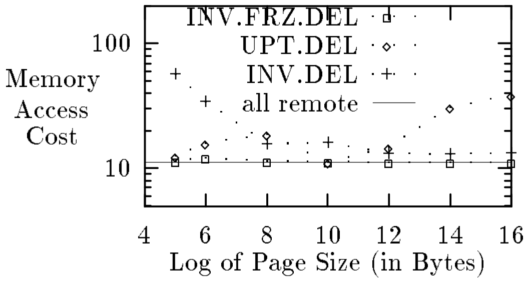

Figure 5: FFT for the delay protocols.

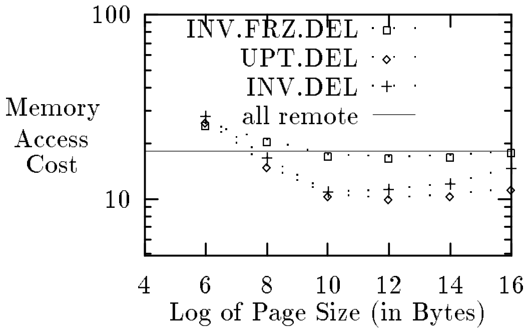

Figure 6: SIMPLE for the delay protocols.

4.3 Effects of Optimal Data Alignment

Our experiments so far have quantified what the operating system *alone* can do to manage locality by

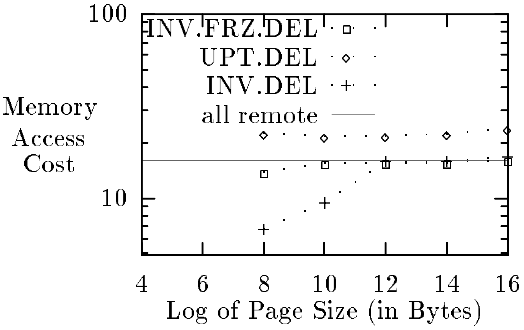

Figure 7: SPEECH for the delay protocols.

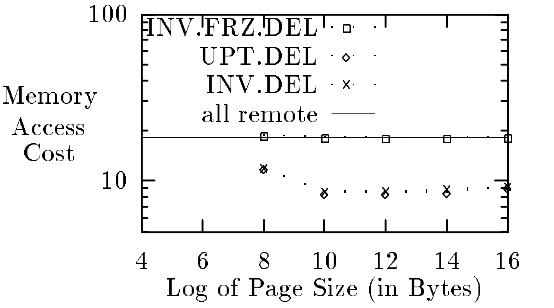

Figure 8: WEATHER for the delay protocols.

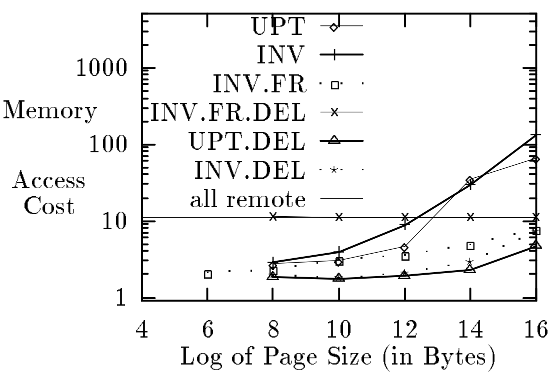

Figure 9: FFT after data re-alignment.

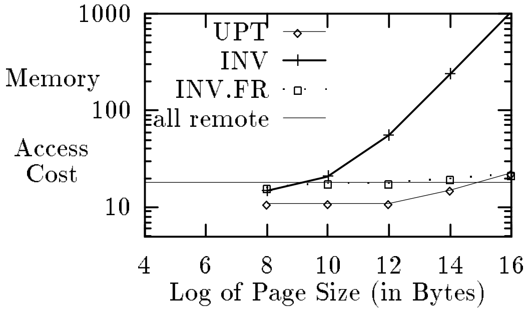

Figure 10: SIMPLE after data re-alignment.

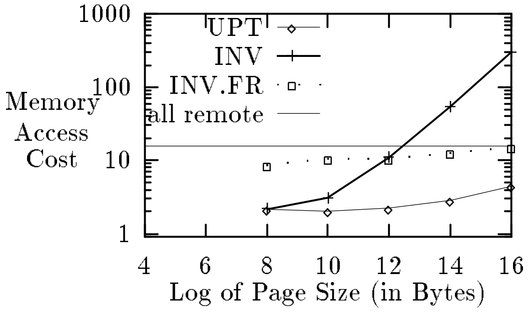

Figure 11: SPEECH after data re-alignment.

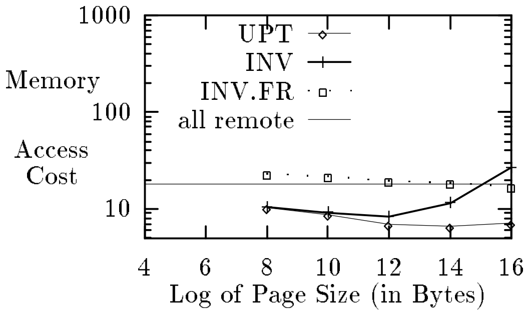

Figure 12: WEATHER after data re-alignment.

Off-line Data Re-alignment Protocol
1: Read 10^6 memory accesses from the input.
2: For each page p in the most recently read set of references, accessed by more than one processor, and written by at least one processor do:
2.1: For each memory location M on page p that is accessed only by processor P: reassign M to page p' that contains reassigned locations accessed only by P.
3: If there are memory accesses in the input, go to 1.

Table 3: The off-line data re-alignment protocol.

page replication. A significant obstacle in managing locality is false sharing. Only the user and/or the compiler can eliminate false sharing by changing the assignment of data to pages, so that data referenced by different processors are in different pages.

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. We implemented a off-line data re-alignment protocol which reorganizes the mapping of data to pages to reduce false sharing. The data re-alignment protocol appears on table 3. Note that if a memory location is accessed by more than one processor then it is truly shared and we do not need to reassign it. Clearly, 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. 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. It is clear 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 after data re-alignment, while the cost for the same policies before data re-alignment is close to 14 (contrast figures 9 and 5). We observe similar behavior on the rest of the applications in figures 10- 12. When the page size is smaller than 4K, all polices, with the exception of the freezing policies, INV.FRZ and INV.FRZ.DEL which perform close to the "all remote" line, perform well. Even though page freezing is generally a good idea, as figures 1– 4 suggest, it is very weak when coupled with data re-alignment. The main advantage of freezing policies, namely reducing the effects of false sharing does not apply anymore.

In SIMPLE (figure 10) we observe that the behavior of the INV.DEL and UPT.DEL policies is not so good for small pages sizes (around 256 bytes), improves for medium pages sizes (1-4K) and deteriorates for larger page sizes. When the page size is small, INV.DEL and UPT.DEL suffer from high operating system overhead. Larger page size imply fewer page faults to transfer a page, and lower system overhead. More details can be found in [9].

Summarizing, our experiments suggest that data re-alignment results in significant performance improvement for all memory coherence protocols. The policies that perform the best, are the UPT.DEL and INV.DEL: the "vanilla" policies augmented with the hardware counter.

5 Conclusions

We examined several factors in the design of memory coherence protocols: the effectiveness of update and invalidate protocols, of freezing mechanisms to avoid page-bouncing, of hardware counters to support delayed replication, and of data re-alignment to reduce false sharing. Our main conclusions are:

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

2: Hardware counters for delayed replication are effective in reducing page bouncing. When coupled with data alignment, hardware counters result in the best performance we observed.

3: Update-based protocols perform better than invalidate-based protocols on large page size especially when no data re-alignment is performed: It is better to send the new data value to the remote page, rather than invalidate a large page now which will be replicated in the near future. In aggreement with [6], invalidate protocols were better for very small page sizes. **4: Freezing mechanisms are not useful when combined with data re-alignment, or hardware counters.** In the absence of hardware support, freezing can be used to reduce excessive replication and updates. Its eagerness to freeze a page pays off when there is lots of false sharing, but is definitely a bad idea when there isn't any.

In conclusion, our results indicate that locality management is a global systems issue that all system levels should cooperate to exploit: The compiler should perform sophisticated data alignment and the operating 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). D. Chaiken from MIT for providing us the multiprocessor traces.

References

[1] BBN Laboratories. Butterfly parallel processor overview. Technical Report 6148, BBN Laboratories, Cambridge, MA, March 1986.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[9] Evangelos Markatos and Catherine Chronaki. Trace-driven simulation of data-alignment and other factors affecting update and invalidate based coherent memory. Technical Report 93, ICS-FORTH, July 1993. Online version at pythis.ics.forth.gr at /tech-reports/1993.TR93.DATA_ALIGNMENT_IN-_VIRTUAL_SHARED_MEMORY.ps.Z.

[10] 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.