Using reference counters in Update Based Coherent Memory

Evangelos P. Markatos and Catherine E. Chronaki

Institute of Computer Science, FO.R.T.H., P.O. Box. 1385, Heraklion, Crete, Hellas GR-711 10, markatos@ics.forth.gr, chronaki@ics.forth.gr

To appear in PARLE 94 (Parallel ARchitectures and Languages Europe), July 1994, Athens, Greece.

Abstract. As the disparity between processor and memory speed continues to widen, the exploitation of locality of reference in shared-memory multiprocessors becomes an increasingly important problem in parallel processing. In this paper, we explore the problem of managing locality at the operating system level. In specific, we study the use of reference counters in making informed decisions about page placement and movement. We use trace-driven simulation of real applications to evaluate the effectiveness of reference counters in providing useful hints to the memory manager of the operating system. Our main conclusion is that reference counters provide a simple and inexpensive mechanism for detecting the reference patterns of pages and making robust page placement decisions that result in significant performance improvement.

1 Introduction

The fundamental mechanism for exploiting locality at the operating system level is replicating pages close to processors that frequently use them. Unfortunately, the existence of multiple copies of the same page, introduces the problem of memory coherence. That is, when a processor modifies its local copy of a page, all processors that have a copy of the same page need to be informed of the change. This update can be accomplished using invalidate or update mechanisms. In invalidate-based mechanisms, all copies the page where the datum resides are *invalidated.* Thus, the next time another processor reads or writes a datum on that page, it will page-fault and request a new copy. In update-based mechanisms, a message is sent to all processors that have a copy of the page. The message contains the updated location and its new value, so that all processors can *update* their copies. Traditionally, invalidate-based protocols have been used in multiprocessor operating systems, mainly for their simplicity. However, their performance has not been satisfactory mainly due to the high replication and invalidation costs they incur. Update-based protocols, on the other hand, usually have better performance over a wider range of parameters [3].

The two most important decisions the operating system needs to make is *when* and *if* to replicate a page to a processor that references it, and *when* to invalidate (unreplicate) a page from a processor that does not use it anymore. When a page is replicated close to a processor p that accesses it, all future accesses to that page by p will be local, a fact that may result in performance improvement. Unfortunately, replicating a page involves the cost of *data transfer*, the *operating system* overhead, and the *cost of updates.* Update-based protocols need to update all copies of a page for each update that is made to that page by *any* processor. In this way, if we have 64 copies of a page, at least 64 update packets have to be sent for each update to keep all copies up-to-date. Thus, a page should be replicated only when the benefits of replication offset its cost. To approximate the read frequency of a page, Bolosky et. al. [1] have proposed the use of $DELAY$ counters. The first time a processor accesses a page, the counter is initialized to some value, and the page is mapped remotely. Each time the processor accesses the page (remotely), the counter is decremented. When the counter reaches zero, it sends an interrupt to the operating system, which replicates the page locally. Assuming that the recent past reference behavior approximates the near future reference behavior, the above counter helps in replicating pages that are accessed several times, so that their replication cost will probably be covered by the benefits of local access. If a processor has a local copy of a page it does not access very frequently, or if other processors modify the same page very frequently, the operating system should consider the option of invalidating this copy due to the high cost of keeping it. For this reason, we propose the use of an $UPDATE$ reference counter that counts the number of updates to a local page that arrive from the network. The operating system sets this counter to an initial value. Each time an update arrives the counter is decremented. When it reaches zero, an interrupt is generated, and the operating system examines whether the page should be invalidated or not. If there have been more updates from the network than local accesses on the given page, the page is invalidated. Otherwise, the operating system resets the counter to its initial value.

Section 2 describes our experimental environment, presents our experiments with the DELAY counter, and describes our experiments of using both counters. Section 3 presents our conclusions.

2 Experimentation

Experimental Environment. To evaluate the tradeoffs in the design of a memory coherency policy, we use trace-driven simulation. The traces we use are 64-processor traces gathered from four programs: FFT, SIMPLE, WEATHER and SPEECH [2]. Table 1(a) describes the applications further. 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 that page local. The parameters of the architecture are shown in table 1(b). 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 *NAMC* is equal to 2, then the average memory access is twice the time it would have taken if all data were in local memory. Our results report *NAMC* for different page sizes ranging from 1 to 16 Kbytes.

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

(a)

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

(b)

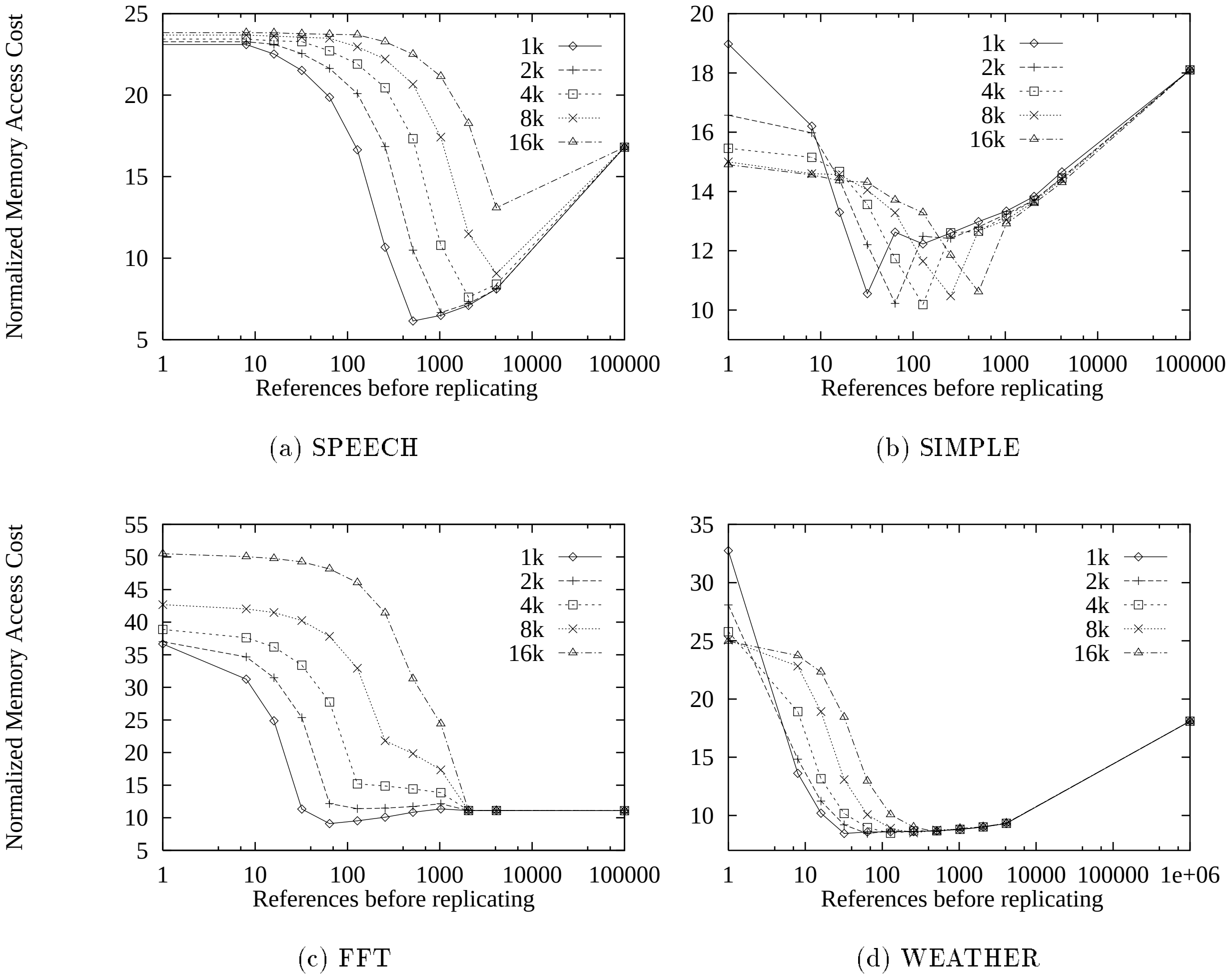
Table 1. (a) Applications used in trace-driven simulation. (b) Architecture Overheads.

The effect of the $DELAY$ **counter.** We started our experiments by exploring the effects of using only the $DELAY$ counter on the performance of memory coherence protocols. We fed the traces to the simulator and varied the initial value of the $DELAY$ counter. The update protocol (UPD) we use is:

- **UPD Read:** If a processor p wants to read a memory location M and it has a local copy of M's page, it just reads M. Otherwise, the $DELAY$ counter is decremented. If the $DELAY$ counter reaches 0, q_i is replicated locally, never to be unreplicated. Otherwise, p reads M remotely. The initial value of the $DELAY$ counter depends on the application and is set by the operating system.
- **UPD Write:** If a processor p 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, p writes M remotely.

The initial value of the $DELAY$ counter, is the number of references a processor should do on a page before the page is replicated locally. When the initial value of the counter is zero, each page is replicated on the first access. This may seem too eager a protocol, but it widely used in cache coherence systems. When the initial value of the $DELAY$ counter is very large, the page is almost never replicated, and the protocol is equivalent to one that accesses all pages remotely.

Figure 1 presents the performance of the four applications update policy, UPD for various page sizes and various initial values of the $DELAY$ counter. Figure 1(a) presents the average memory access cost of the SPEECH application for various page sizes, and various initial values of the $DELAY$ counter. The x axis runs in units of the initial value of the counter. We notice that independent of the page size, when we replicate a page on the first access, i.e. the initial value of the counter is one, the performance of the protocol is consistently close to 23, which is 4 times higher than the cost of accessing all data locally. Increasing the initial value of the $DELAY$ counter generally improves performance by reducing memory access cost. We see that the lowest normalized memory access cost is achieved when the page size is 1K and the initial value of the $DELAY$ counter is 512 references. It is interesting to note that the normalized memory access cost decreases rapidly (performance improves) with the initial value of the counter, then it reaches a minimum and then it increases slowly. SPEECH has the best performance for the smallest page size we simulated. This is reasonable, since small pages mean less false sharing. It is surprising though, that even for larger pages, appropriately chosen initial values of the $DELAY$ counter result in very good performance, even better than the performance of small pages that have poorly chosen initial values of the $DELAY$ counter. SIMPLE 1(b) behaves in a similar fashion. Performance rapidly increases with the initial value of $DELAY$, reaches a maximum and then slowly decreases again. Both SIMPLE and SPEECH share a common observation: The initial value of the $DELAY$ counter where the best performance is observed, doubles with page size.



(a) SPEECH (b) SIMPLE

(c) FFT (d) WEATHER

Fig. 1. The effect of the $DELAY$ counter. Different lines depict different page sizes.

WEATHER behaves accordingly, its only difference being that for a large range of initial values of the $DELAY$ counter (200-2028) the performance is close to the best achievable. Thus, carefully choosing the initial value for SIMPLE and SPEECH is more important than choosing an appropriate initial value for WEATHER. FFT behaves much like WEATHER. Although the performance of the applications above differ from each other, the following comments apply to all of them:

1. Immediate replication is generally a bad idea. When the initial value of the counter is 1, the memory access cost is quite high, higher than having all pages remote ($DELAY = \infty$) in most cases.
2. When the value of the counter is low (256 to 2048) the performance is within acceptable bounds, usually close to the optimal.
3. The performance for the optimal value of $DELAY$ is much better than having all pages mapped remotely. This observation suggests that locality management does not pay off if pages are replicated immediately, while it is worthwhile when a mechanism like $DELAY$ counters is used.

Using both Counters. The previous section suggested that the use of $UPDATE$ counter provides marginal or no improvement when the $DELAY$ counter is initialized to its optimal value. In this section, we present experiments that quantify the performance benefits when both counters are used. To keep the experimental space small, we focused on the SPEECH application for 1Kbyte pages only. We run the update protocol for various values of both counters in the range of 8 to 4096. We measured the performance of the policy using both reference counters and compared it to the performance of the application running the simple UPD protocol without any counters, by calculating the ratio of the performances. This ratio is plotted in figure 2. The x axis is the value of the $DELAY$ reference counter. The y axis is the value of the $UPDATE$ reference counter. The z axis plots the improvement ratio. For instance, a value of 10 on the z axis means that using both counters results in average access costs 10 times smaller than not using any counters at all.

The first thing we notice is that for small values of the $DELAY$ counter, using both counters results in significant performance improvements. Using both counters does not only improves the performance, but it also improves the overall best performance observed in our experiments. Actually, when the initial values of the $DELAY$ and $UPDATE$ counters are 64 and 32 respectively, the average memory access cost is 5.55, the minimum observed in our experiments (the best performance we have observed using just the DELAY reference counter was 6.14.) We see that using both counters for a large range of their values we get a very good performance.

In essence, our results suggest that if we replicate pages too soon, i.e. the initial value of the $DELAY$ counter is small, then unreplicating, i.e. using the $UPDATE$ counter, significantly improves performance. If however, we replicate pages on time, or *too* late then unreplication *does* not improve performance, on the contrary, an untimely unreplication may hurt performance.

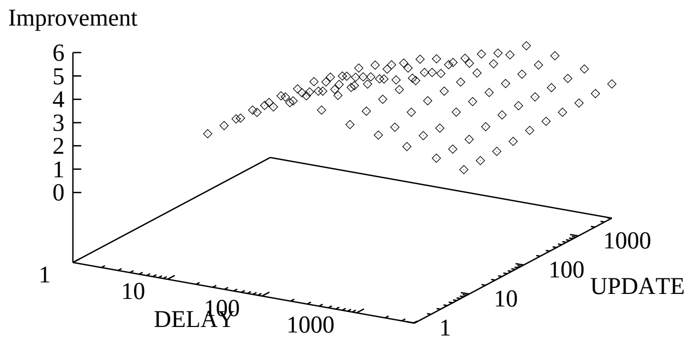


Fig. 2. Delaying Replication and unreplication in SPEECH.

3 Conclusions

In this paper, we used trace driven simulation and analysis to evaluate the benefits of using reference counters in update-based memory coherence protocols. Reference counters approximate the page access patterns of various processors. Based on our experiments we conclude:

- The use of the $DELAY$ counter significantly improves performance. We have seen a performance improvement of a factor of 5 (see figure 1(a) and figure 1(c)).
- The use of both $UPDATE$ and $DELAY$ counters results in more efficient and more *robust* policies. In other words, the use of the $UPDATE$ counter reduces the need for careful fine tuning of the initial value of the $DELAY$ counter. We may initialize the $DELAY$ counter to a relatively low value, and the use of the second counter will eliminate the effects of any excessive replication.

Based on our experiments we conclude that reference counters, coupled with memory management policies similar to the ones presented here, are a necessary hardware mechanism for the efficient performance of memory management in parallel operating systems. Reference counters prevent several pathological cases and improve performance. The use of both $DELAY$ and $UPDATE$ counters results in robust memory management policies, reducing the need for fine-tuning their initial values.

Acknowledgments. Manolis G.H. Katevenis provided useful feedback during the early stages of this research. Financial support for part of 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 provided us with the multiprocessor traces.

References

1. 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.
2. 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.
3. E. P. Markatos and C. E. Chronaki. Trace-Driven Simulation of Data-Alignment and Other Factors Affecting Update and Invalidate Based Coherent Memory. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '94)*, pages 44–52, January 1994. Also appeared as ICS-FORTH Technical Report 93, July 1993.