Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems

Evan Speight and Martin Burtscher
School of Electrical and Computer Engineering
Computer Systems Laboratory, Cornell University, Ithaca, NY 14853
{espeight,burtscher}@csl.cornell.edu

Abstract

Software distributed shared memory (SDSM) systems traditionally exhibit poor performance on applications with significant fine-grain or false sharing. Techniques such as relaxed-consistency models and multiple-writers protocols improve the performance of SDSM systems significantly, but their performance still lags that of hardware shared memory implementations. This paper describes Delphi, a system that borrows techniques from microarchitectural research on value prediction and applies them to software distributed shared memory. We run a small software predictor on each node in the Delphi system to predict which virtual pages will be needed in the future. We use the predictions to prefetch pages in order to reduce the number of accesses to invalid data and thereby reduce expensive network accesses. Experimental results show that Delphi is able to reduce the number of read misses to virtual pages by up to 62% on a set of well-known scientific benchmarks with minimal runtime overhead in extra processing and memory requirements. This translates into a 14% reduction in execution time over a comparable base system that does not employ prediction techniques.

Keywords: prediction, prefetching, software DSM

1. Introduction

Recent improvements in commodity general-purpose networks and processors have made networks of multiprocessor PC workstations an inexpensive alternative to large monolithic multiprocessor systems. However, applications for such distributed systems are difficult to develop due to the need to explicitly send and receive data between machines. By providing an abstraction of globally shared memory on top of the physically distributed memories present on networked workstations, it is possible to combine the programming advantages of shared memory and the cost advantages of distributed memory. These distributed shared memory (DSM), or shared virtual memory, runtime systems transparently intercept user accesses to remote memory and translate them into messages appropriate to the underlying communication media. The programmer is thus given the illusion of a large global address space encompassing all available memory, which eliminates the task of explicitly moving data between processes located on separate machines.

Both hardware DSM systems (e.g., Alewife [2], DASH [13], FLASH [12]) and software DSM systems (e.g., Ivy [14], Munin [5], and Brazos [16]) have been implemented. Recent increases in PC performance, the exceptionally low cost of PCs relative to that of workstations, and the introduction of advanced PC operating systems combine to make networks of PCs an attractive alternative for large scientific computations.

Software DSM systems use the page-based memory protection hardware and the low-level message passing facilities of the host operating system to implement the necessary shared memory abstractions. The large size of the unit of sharing (a virtual page) and the high latency associated with accessing remote memory challenge the performance potential of software DSM systems. A variety of techniques have been developed over the last decade to address these issues [7, 10, 11]. This paper examines the use of prediction techniques borrowed from the value-prediction domain, a promising new research area in computer architecture. Value predictors predict the result of instructions before the CPU can compute them, which speeds up program execution [15]. We have implemented one of these predictors in software and adapted it to predict page faults to increase the accuracy of prefetching multiple shared pages together. We show that a predictor requiring a minimal amount of runtime overhead, both in terms of memory usage and execution time, can reduce the number of network accesses by up to 56%, and the overall runtime by 14% on a set of well-known, shared memory parallel benchmarks.

The rest of this paper is organized as follows. Section 2 discusses the prediction scheme we have implemented in the Delphi system to prefetch virtual pages. Section 3 describes our experimental setup and the applications used in our evaluations. Section 4 presents performance results for several scientific applications. We conclude and discuss future work in Section 5.

2. Delphi Implementation

This section provides a brief overview of the baseline coherence protocol used in the Delphi system, as well as the prediction techniques employed to reduce the number of page faults and network operations.

2.1. Overview

Delphi uses a home-based protocol for maintaining consistency between shared memory processes in the cluster. Initially, groups of ten virtual pages are assigned the same home node in a round-robin fashion. During the execution of an application, if a single process is the only writer for a given page, the home designation is migrated to that writing process in order to reduce the number of network updates that must be propagated to the home node. Similar to other software DSM systems, modifications to virtual pages are tracked through the process of *twinning* and *diffing* [5]. When a page is first modified by a process, a copy of the page (known as a *twin*) is created. When changes to the page must be propagated to other nodes in the system in order to maintain coherence, a runlength encoding of the changes is created by conducting a word-by-word comparison of the twin with the current page. This encoding is referred to as a *diff*, and is simply a compact representation of the changes to shared data.

In home-based systems such as Delphi, a global synchronization event such as a barrier causes all modified pages to be invalidated and diffs to be made to track the page changes prior to the invalidation. In Delphi, each page's diff is eagerly flushed to the correct home node at a synchronization point. Subsequently, when a process faults on an invalid page, the home node is contacted by the faulting process, and the request is satisfied with a single response consisting of the entire page from the home node. In the current Delphi system, the page will always be up-to-date at the home node, resulting in a tradeoff of longer synchronization times for faster individual page-fault response time.

2.2. Prediction in Delphi

Several studies have examined the performance impact on software DSM systems of dynamically altering the virtual page size in order to improve performance. By aggregating pages that are sequential in virtual memory into *page groups* [3], data is implicitly prefetched, which may result in fewer page faults and less network overhead. The downside of this approach is that it may lead to an increase in false sharing, as a larger unit of coherence increases the chance that multiple writers will attempt to modify different portions of the same, larger page. The adaptive protocol presented in [3] attempts to reduce this effect by dynamically changing the page size to compensate for the prefetching vs. false-sharing contention.

The Adaptive++ system [6] relies on two modes of operating to predict which pages to prefetch, referred to as repeated-page mode and repeated-stride mode. Separate lists are maintained of pages likely to be referenced in the future based on observed access patterns. In addition to using a less-general prediction algorithm than that used in Delphi, the Adaptive++ system does not update pages with prefetched data until a process faults on the page. Finally, the Adaptive++ system has been integrated into the TreadMarks software DSM system [11], a distributed-page based system, while Delphi has been added to a home-based software DSM system. This makes direct comparisons between the performance of the two approaches difficult.

The approach taken in Delphi differs in several aspects from previous efforts. First, standard, validated prediction techniques are used to predict which pages will likely be accessed next based on the history of previous accesses. Second, due to the extremely low runtime overhead associated with our prediction technique, Delphi can predict and aggregate pages based on several independent streams of references, improving prediction accuracy. Finally, Delphi's home-based protocol allows prefetched updates to be applied at requesting nodes in an overlapped manner while application computation continues.

We investigated several different value predictors and configurations and found the third-order differential finite context method predictor [9] to yield the highest accuracy over a large range of predictor sizes. This predictor continuously monitors all misses to virtual, shared pages. For any three consecutive page misses, it records the page number of the next miss in a hash table. During a prediction, a table lookup determines which page miss followed the last time the predictor encountered the same three most recent misses.

When a process faults on a virtual page, the predictor responsible for predicting the next page access will return the values of the next *N* predicted pages the process will access that share the same home node as the page incurring the fault. We maintain a separate predictor for each home node to improve the prediction accuracy. The faulting node requests these (up-to) N pages from the home node, which responds with not only the necessary page, but also all other predicted pages. In this way, Delphi seeks to avoid faulting and the resulting network accesses for the pages that will likely be accessed in the near future. Additionally, computation is allowed to proceed as soon as the request page has been delivered, overlapping computation with the processing of the prefetched pages.

2.3. Predictor Operation

The third-order differential finite context method (DFCM) predictor consists of two levels. The first level retains the page numbers of the three most recent misses. However, only the most recent page number is stored as an absolute value. The remaining values are the differences (strides) between consecutive page numbers. The number of strides determines the *order* of the predictor. We use a third-order DFCM since it yields good results. However, other orders work equally well.

The strides stored in the first level are combined to form an index into the second level, which is essentially a hash table. Whenever a page miss occurs, the difference (stride) between the previous and the current miss is stored in this table so it can later be retrieved when the same sequence of three preceding misses is encountered. Since the retrieved value is a stride, it needs to be added to the stored most recent page miss number to yield the actual prediction.

Our DFCM predictor uses the select-fold-shift-xor function [9] to compute the index into the hash table. The following example shows how to compute this function for the three stride values st1, st2, and st3 for a 4096-entry hash table. The symbol "⊕" represents XOR and the subscripts refer to bit positions.

$$\begin{gathered} \mathit{hash}(\mathrm{st}) = \mathrm{st}_{63..60} \oplus \mathrm{st}_{59..50} \oplus \mathrm{st}_{49..40} \oplus \mathrm{st}_{39..30} \oplus \\ \mathrm{st}_{29..20} \oplus \mathrm{st}_{19..10} \oplus \mathrm{st}_{9..0} \end{gathered}$$

$$\begin{gathered} \mathit{index}(\mathrm{st1}, \mathrm{st2}, \mathrm{st3}) = \mathit{hash}(\mathrm{st1}) \oplus \mathit{hash}(\mathrm{st2}) \ll 1 \\ \oplus\, \mathit{hash}(\mathrm{st3}) \ll 2 \end{gathered}$$

Each of the three values is broken down into n-bit chunks. If the last chunk is shorter than n bits, it is zero padded to n bits. The chunks are then XORed to yield an n-bit hash value. The hashes of the three values are shifted by zero, one, or two bits, respectively, and then XORed again to form an $(n+2)$-bit index. Hence, for a hash table with *size* entries, $n = 1+\log_2(\mathit{size}) - \mathit{order}$. In the above example, $n = 10$.

During a prediction, an index is computed using the three strides from the DFCM's first level to access the hash table, which provides the predicted stride. During updates, the same index calculation is performed, the corresponding hash-table entry is overwritten with the new stride, and the new stride is shifted into the predictor's first level.

3. Experimental Setup

The experiments presented in this section were carried out on a cluster of 8 Dell PowerEdge 1500 rack-mounted servers running Windows 2000, Advanced Server Edition. Each machine contains dual 866 MHz Pentium III processors, 1 GByte of RAM, and is interconnected with the cLAN system area network. The cLAN GNN1000 NIC provides a user-level network that complies with the Virtual Interface Architecture specification [1] for system area networks. The applications used for the initial prediction results include four shared memory parallel benchmarks chosen to represent the observed behavior of our prediction scheme across applications with different access patterns. Two applications are taken from the NAS parallel benchmark suite [4]: 3DFFT, a 3-D fast-Fourier transform partial differential equation benchmark; and MG, a 3-D multigrid solver. We also examine results for Ilink [8], a genetic-linkage package used to trace genes through family histories; and Gaussian Elimination with back-substitution.

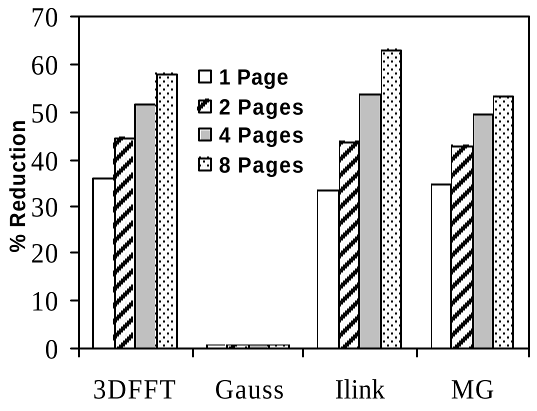

Figure 1. Reduction in Miss Rate for Varying Predictor Window Sizes. Predictor Size = 16 Kbytes.

4. Results

We first present results comparing the performance of Delphi with a fixed-size predictor to the baseline SDSM system. For the following results, we use a predictor with 2^{12} entries and vary the number of pages that we prefetch from the home node (referred to as the *prediction window*) on a faulting page access. For these experiments, we use prediction windows of 1, 2, 4, and 8 pages.

4.1. Results with a Fixed Predictor Size

Figures 1 through 5 show the results for different performance parameters for the four applications studied. Figure 1 depicts the percentage reduction in the number of misses experienced by each application for the 4 prediction window sizes listed above. 3DFFT, Ilink, and MG all experience a large reduction in the number of misses when the Delphi predictor is used. When only 1 page is predicted, the percentage of reduction in these applications is 36%, 34%, and 35%, respectively. As we increase the prediction window, the number of read misses is further reduced as more and more shared pages are prefetched and used before being invalidated. Figure 2 shows the percentage of these predicted pages that are actually used by the application before they are next invalidated, indicating the effectiveness of our prediction scheme. 3DFFT, Ilink, and MG all use a very high percentage of the predicted pages. Rates for a single-page prediction window are near or above 80% for these three applications. As the predictor is asked to "look further into the future" and more pages are predicted on each fault, the percentage of useful prefetched pages goes down for several reasons. First, the predictor's accuracy goes down with each additional page predicted. Second, no information regarding page invalidations that may occur between the time a page is prefetched and when that page is invalidated is taken into account, meaning that many pages are prefetched but not used until after a global synchronization event that invalidates the page, resulting in a wasted prefetch.

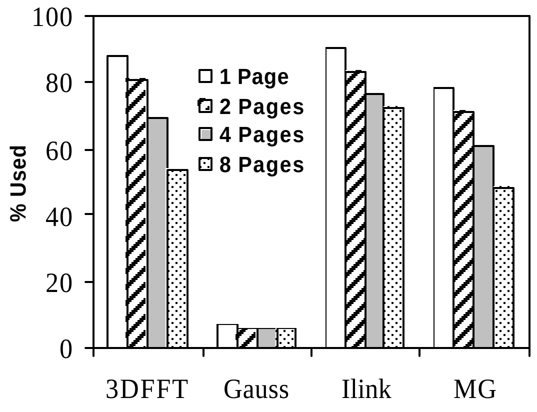

Figure 2. Percentage of Prefetched Pages Used. Predictor Size = 16 Kbytes.

Gauss is an example of an application for which the prediction scheme employed in Delphi basically does not reduce the number of misses, as shown in Figure 1. Regardless of the prediction window size used, the number of misses experienced by Gauss remains relatively constant. Figure 2 shows that less than 10% of the pages predicted and prefetched are used by Gauss.

Figure 3 shows the reduction in the number of network send operations for each application for the four prediction window sizes examined. Again, 3DFFT, Ilink, and MG show a large reduction in the number of network accesses required for program completion, as piggybacking predicted pages on "normal" page fault messages eliminates subsequent network requests that would have occurred for the prefetched pages. The network access count goes down with an increasing prediction window size, showing that even though Figure 2 indicates that the percentage of predicted pages goes down with an increased window size, enough of the predicted pages are being used to reduce the overall number of network send operations.

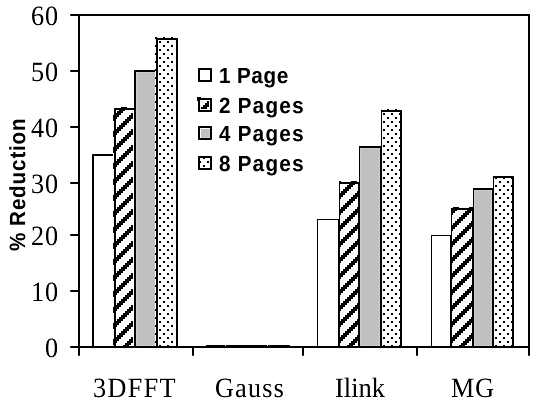

Figure 3. Reduction in the Number of Network Send Operations. Predictor Size = 16 Kbytes.

Figure 4 shows the total number of bytes sent across the network. Unsurprisingly, when the prediction window grows, the number of bytes sent also increases as more and more pages are prefetched on each program page fault. Figures 3 and 4, when taken together, show that if accessing the network (e.g., the fixed costs such as protocol overhead) is dominant over the variable costs (e.g., the wire time of a network send operation), the prediction scheme in Delphi should benefit performance substantially. Again, due to the ineffectiveness of the Delphi predictor on Gauss, Figure 3 shows almost no reduction in the number of network accesses over the baseline case for this application, and Figure 4 shows a modest increase in network traffic.

Finally, Figure 5 shows the percentage reduction in execution time for the four applications studied. 3DFFT, Ilink, and MG show reductions in execution time for 1, 2, and 4 page prediction window sizes, with 3DFFT showing the largest performance improvement of 13.5% when a 1-page prediction window is used.

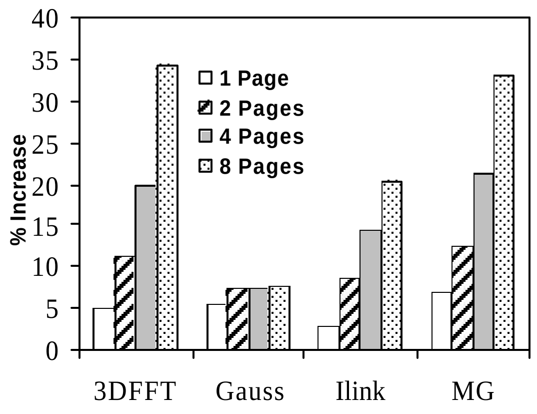

Figure 4. Increase in Bytes Sent Across the Network. Predictor Size = 16 Kbytes.

As the prediction window increases, the performance is determined by two competing factors: the increase in network utilization as shown in Figure 4, and the reduction in the number of misses as shown in Figure 1. In the case of MG, a prediction windows size of 8 pages causes the increase in network bytes sent to overcome the gains achieved by prefetching 8 pages on each page fault. 3DFFT and Ilink show similar trends with increasing prediction window sizes, but to lesser degrees. Gauss, as expected from the data presented in Figures 1 through 4, shows only a small performance change between the predicted and the baseline case.

The performance of Gauss is poor because the predictor often cannot find pages that are likely to miss in the near future and that have the same home node as the currently missing page. Moreover, the majority of the pages the predictor manages to prefetch end up being invalidated before they are used.

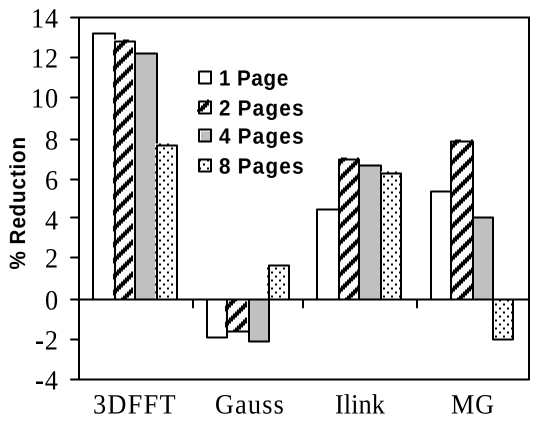

Figure 5. Reduction in Execution Time. Predictor Size = 16 Kbytes.

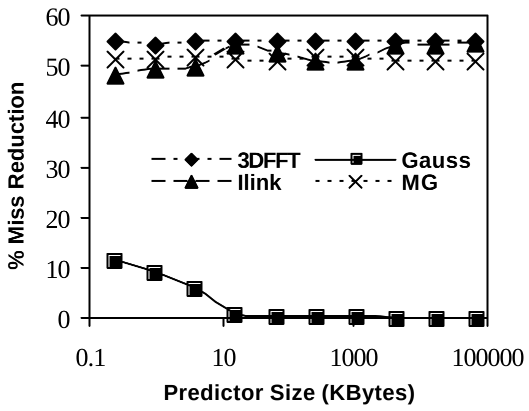

Figure 6. Reduction in Misses with Varying Predictor Sizes. Prediction Window = 2 pages.

4.2. Results with Varying Predictor Size

The results in the preceding section maintained a constant predictor size of 2^{12} entries, where each entry is 4 bytes in size. We have 1 predictor per remote node running in each instance of the SDSM system, bringing the total predictor size to 2^{12}*4*(num_nodes-1) = 112 Kbytes for 8 nodes. Each baseline SDSM runtime system consumes on the order of 88 Mbytes, meaning that the predictor structure overhead is a negligible 0.1% overhead with the predictor size used in Section 4.1. In this section, we examine the effect of keeping the prediction window constant at a value of 2, and varying the predictor size from 256 bytes up to 64 Mbytes. Figure 6 shows that for 3DFFT, Ilink, and MG, the reduction in the miss rate for these applications is mostly insensitive to the size of the predictor. Gauss, on the other hand, shows a marked reduction in the effectiveness of the predictor as the size grows. This counterintuitive result is due to constructive aliasing in the small predictor tables that is eliminated as the table size increases. As a result, fewer useful pages are prefetched with the larger predictors, explaining Gauss' lower performance.

5. Conclusions

We have presented the Delphi software DSM system that incorporates prediction techniques borrowed from microarchitecture research in value prediction to intelligently guide prefetching decisions.

By incorporating into Delphi a group of predictors that consume negligible processing and memory resources relative to the overall DSM runtime system, we have shown that miss rate reductions of up to 62% can be achieved with a concomitant improvement in overall execution time. We are currently investigating several ways to further improve the performance of the Delphi system. First, we plan to make use of the use of remote DMA operations available in the VI Architecture for prefetched pages to reduce the processing overhead associated with prefetched pages. Second, we are extending the work presented here to adaptively adjust the predictor size and prediction window dynamically during runtime to better adjust to a range of applications. Finally, we are working on aggregating predicted requests from multiple home processes to address problems such as those shown in Gauss, in which pages referenced in the near-future either do not reside on the same home node as the page currently needed, or are invalidated before being used by the computation threads.

References

[1] Virtual Interface Architecture Specification 1.0. 1997.

[2] A. Agarwal, R. Bianchini, D. Chaiken, and K. L. Johnson. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the International Symposium on Computer Architecture*, pp. 2-13, May 1995.

[3] C. Amza, A. Cox, K. Rajamany, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 1997.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. NASA Ames RNR-91-002, August 1991.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pp. pages 168-176, March 1990.

[6] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proceedings of the International Conference on Supercomputing*, pp. 385-392, July 1998.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *Transactions on Computer Systems*, vol. 13, pp. 205-243, 1995.

[8] S. Dwarkadas, R. W. C. Jr., P. Keleher, A. A. Schaffer, A. L. Cox, and W. Zwaenepoel. Parallelization of General Linkage Analysis Problems. *Human Heridity*, vol. 44, pp. 127-141, 1994.

[9] B. Goeman, H. Vandierendonck, and K. Bosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[10] L. Iftode, C. Dubnicki, E. W. Felton, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pp. 14-25, February 1996.

[11] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pp. pages 115-131, January 1994.

[12] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302-313, April 1994.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, vol. 25, pp. 63-79, 1992.

[14] K. Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. Ph.D. Thesis, Yale University, 1986.

[15] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, October 1996.

[16] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the First USENIX Windows NT Workshop*, pp. pages 95-106, August 1997.