Searching an Optimal History Size for History-Based Page Prefetching on Software DSM Systems

Cristian Ruz[1] and José M. Piquer[2]

[1] Escuela de Ingeniería Informática, Universidad Diego Portales
cristian.ruz@udp.cl
[2] Depto. Ciencias de la Computación, Universidad de Chile
jpiquer@dcc.uchile.cl

Abstract. This work presents a study regarding the search for an optimal value of the history size for the prediction/prefetching technique *history-based prefetching*, which collects the recent history of accesses to individual shared memory pages and uses that information to predict the next access to a page. On correct predictions, this technique allows to hide the latency generated by page faults on the remote node when the access is effectively done. Some parameters as the size of the *page history* structure that is stored and transmitted among nodes can be fine-tuned to improve the prediction efficency.
Our experiments show that small values of history size provide a better performance in the tested applications, while bigger values tend to generate more latency when the *page history* is transmitted, without improving the prediction efficiency.

Keywords: Distributed shared memory, data prefetching, distributed systems.

1 Introduction

Software distributed shared-memory (DSM) systems provide programmers with a virtual shared memory space on top of low cost message-passing hardware, and the ease of programming of a shared memory environment, running on a network of standard workstations [1]. However, in terms of performance, DSM systems suffer from high latencies when accessing remote data due to the overhead of the underlying message-passing layer and network access [2,3]. To address these issues several latency-tolerance techniques have been introduced. One of these techniques is called *prefetching*, it reduces latency by sending data to remote nodes in advance of the actual data access time.

Many *prefetching* techniques have been proposed. In this work we will focus on *history-based prefetching*, a prediction/prefetching strategy that has proved useful to reduce latency issues for a DSM system on certain applications that show a regular memory access pattern [4]. In order to reduce latency, this technique collects the recent history of accesses to individual shared memory pages,

L.T. Yang et al. (Eds.): HPCC 2005, LNCS 3726, pp. 133–142, 2005.

and uses that information to predict the next access to a page through the identification of memory access patterns.

Throughout the execution of an application, the number of accesses made to a certain page can be very large. Hence it could be highly inneficient to store the whole history of accesses made to each page. The solution to this problem is to store only the M most recent accesses. The parameter M therefore determines the size of the *page history*.

This work presents a study regarding the search for an optimal value of the parameter M. There are adventages and disadvantages of giving M a big value. On one hand, a big history is more likely to contain the information required to make a correct prediction, therefore reducing latency. But, on the other hand, since the *page history* must travel along with the ownership of the page from one node to another, a bigger history involves the size of the messages that must be sent to grow, causing latency to increase due to the excess of data that has to be transmitted.

A series of experiments were done with three applications running over a page-based DSM system, on a 14-nodes linux cluster. Results show that small values for M provide a better performance in the tested applications. Bigger values tend to generate more latency when *page history* is transmitted, and does not provide a major benefit in the prediction efficiency.

The rest of this paper is organized as follows. In section 2, some related work regarding other prefetching techniques is discussed. Section 3 describes the prediction technique, and the issue of the size of the *page history*. In section 4, experimental results and analysis are presented. Finally, section 5 gives conclusions and perspectives of future work.

2 Related Work

Bianchini et al. have done important work in prefetching techniques for software DSM systems. They developed the technique *B+* [5] which issues prefetches for all the invalidated pages at synchronization points. The result is a high decrease of page faults, but at the cost of sending too many pages that will not be used and increasing bytes transfer. They also presented the *Adaptive++* technique [6] that predicts data access and issues prefetches for those data prior to actual access. Their work uses a local per-node history of page accesses that records only the last two barrier-phases and issues prefetches in two modes: repeated-phase and repeated-stride. Lee et al. [7] improved their work, using an access history per synchronization variable. *History-based prefetching* uses a distributed per-page history to guide prefetching actions, in which multiple barrier-phases can be collected leading to a more complete information about the page behavior. Prefetches are only issued at barrier synchronization events.

Karlsson et al. [8], propose a history prefetching technique that uses a per-page history, and exploits the producer-consumer access pattern, and, if the access pattern is not detected, uses a sequential prefetching. *History-based prefetching* differs in that it supports more access patterns, and the *page history*

mechanism provides more flexibility to find repeated patterns that are not categorized. Also, if no pattern is detected, no prefetching action is generated, avoiding useless prefetches.

3 History-Based Prediction

History-based prediction is a technique that allows processors to prefetch shared memory pages and make them available before they are actually accessed. Using a correct prefetching strategy, page faults are avoided and the latency due to interruptions and message waiting from other nodes is hidden, improving the overall application performance.

Historical information about page behavior is collected between two consecutive barrier events. Predictions are generated inside a barrier event to make sure that no other node may generate regular consistency messages, hence avoiding overlapping of those messages and prefetching actions. When every node has reached a barrier, a *prediction phase* is executed, in which every node makes predictions using the information collected, and speculatively sends pages to other nodes. After that, the barrier is released.

3.1 Page History

History-based prediction uses a structure that stores the access history for each shared memory page. This structure is called *page history*. A *page history* is a list H, composed of M *history elements*: $H = \langle h_1, h_2, \ldots, h_M \rangle$. Each *history element* h_i represents one access to one shared memory page and contains the following information regarding the node that accessed the page: the access mode used; the number of the execution phase when the access was actually made; and a list of predictions that were made in the past when this history element was the last in the *page history*. Only the last M *history elements* are kept, reflecting the last M accesses to the page.

A *page history* is updated using a sequential-consistent model. A *page history* migrates between nodes along with the ownership of the page where it belongs, every time a node gets permission to write on that page. At any time, only the owner of the page can update its *page history*.

The *page history* of page p is updated by the owner of p when a local read fault or write fault on p is detected, and also when remote faults over p are received. After the *page history* of p has been updated, the owner may reply to the remote request according to the rules of the consistency protocol.

3.2 Prediction Strategy

Predictions are made at the end of each execution phase, when all nodes have reached a barrier, to avoid overlapping with regular consistency actions. Every node executes a predictive routine for each page that it owns.

The predictive routine attempts to find a pattern on the *page history*, by looking for the previous repetition of the last D accesses seen, and predicting

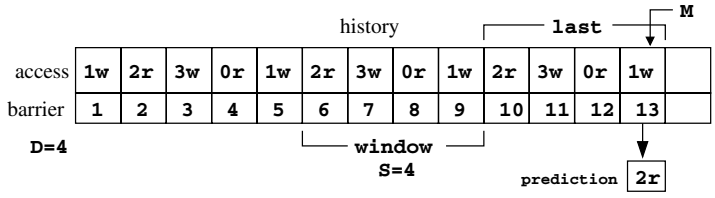

Fig. 1. Prediction of access `2r` with $D = 4$. The match was found when $S = 4$.

the same behavior that was seen, in the past, after that sequence. In applications that show a regular memory access pattern, the repetition of a certain sequence of accesses can be expected, so that this prediction should be correct in most situations.

If the routine can not deduce a possible next access, then no prediction is made ; otherwise, another routine is called to actually execute the predictions.

A *page history* is analyzed using a fixed-size window W of length $D < M$, $W = \langle w_1, w_2, \ldots, w_D \rangle$ and comparing them to the last D accesses on *page history*, stored in the list $Last = \langle h_{M-(D-1)}, \ldots, h_{M-1}, h_M \rangle$. *Last* is compared to $W_S = \langle h_{M-(D-1)-S}, h_{M-(D-2)-S}, \ldots, h_{M-1-S}, h_{M-S} \rangle$ for $S \in \{1, 2, ..., M-D\}$ in increasing order. If both lists happen to be equal, then the access h_{M-S+1} is predicted as the next access for the page, as shown in Fig. 1.

```
p.Prediction():
   if (p.status == PREFETCHED)
      return 0;
   for(i=0;i<D;i++)
      last[i] = p.history[M-(D-i-1)];
   for(s=1;s<M-D;s++) {
      for(i=0;i<D;i++)
         window[i] = p.history[M-(D-i-1)-s)];
      if(compare(last,window)) {
         p.history[M].addPrediction(p.history[M-s+1)]);
         return 1;
      }
   }
   return 0;
```

If the status of the page is `PREFETCHED`, it means that the page is owned by the node because of a previous prefetching action and was not accessed during the previous *execution phase*. In this situation the page was predicted but it was not used by the destination node, so it should not be predicted again.

3.3 Pattern Analysis

The page behavior may also be predicted by the identification of three predefined page access patterns and taking appropriate actions for each one. If no pattern can be detected, then the prediction routine described in section 3.2 can be used.

Pattern detection is based on the categorization described by Monnerat and Bianchini [9], where page behavior can be classified as 1PMC, MIG or MW.

1PMC is a *one producer - multiple consumers* pattern, where there is only one node that writes on the page, and many readers. This behavior has a low rate of page ownership transfers. MIG stands for *migratory* pages. It is a pattern where the page ownership is transferred to a different node in each execution phase. This behavior allows a good hit ratio to be reached using the basic prediction strategy of *history-based prefetching*. MW is a *multiple writer* pattern, in which one page is owned by different nodes inside the same execution phase. In this case it is hard to make a good prediction. Since the prediction strategy only works at the end of an execution phase, at most 1 page fault will be avoided and the $N-1$ remaining writes will still generate page faults. The case may be even worse if the nodes must compete for the access to the page, which will make the access almost random at every phase.

3.4 Page History Size

Previous work [4] showed the usefulness of *history-based prefetching* to be applied on some applications that show a regular memory access pattern. Experiments considered a fixed size M for *page history* assuming this choice was good enough for each application.

The size M of the *page history* data structure determines the amount of history that is kept to make prediction decisions, and that is sent to remote nodes along with the ownership of the related page. Theoretically, a large value of M could improve the prediction accuracy, because a greater amount of information is transmitted every time a *page history* is transferred between two nodes. On the other hand, a smaller history size is faster to transmit, but may not have enough information to deal with some situations and could lead to wrong predictions.

4 Experiments

The experiments were executed on a platform of 14 Pentium IV processors, 256MB RAM, running linux Fedora Core 1. All computers execute the applications over DSM-PEPE [10], a page-based software DSM system designed to execute parallel applications over a shared-memory environment on multicomputers and different consistency protocols. Tested applications follows.

- LIFE is an implementation of Conway's *Game of Life* [11] on a 2048×2048 circular matrix. The parallelization is done through stripes. On each iteration, each node computes a different stripe of the matrix. Processors wait in a barrier before computing the next iteration.
- SHEAR is an implementation of the *Shearsort* algorithm [12] to sort integers inside a 1024×1024 matrix. The execution goes through a fixed number of alternate row-phases and column-phases. The parallelization is done through stripes. At every iteration, each node works over a fixed set of consecutive rows, or consecutive columns.

– GRAPH is a distributed *single-source shortest-path* search on a graph of N vertexes on shared memory [13]. Each node is assigned a fixed set of vertexes to evaluate. On the kth iteration, nodes compute the shortest path from each vertex to a distance of k vertexes away, using the information from their neighbors, if necessary. After N iterations, the weight of the shortest-path to every node has been calculated.

4.1 Methodology

Each experiment was executed under a sequential-consistent protocol, using *history-based prefetching* and different values for M. Also, a normal execution without any kind of prefetching was done.

Statistics collected in each case include execution time, locally-generated read-faults and write-faults, prefetches done for *read-only* and *read-write* modes, and correct prefetches detected.

The metrics used to evaluate the effectiveness of the technique are *coverage* and *hit ratio*, as it has been used in previous related works [4,3]. *Coverage* refers to the percentage of page faults which are eliminated by prefetching pages in advance. *Hit Ratio*, or *utilization*, is defined as the percentage of valid prefetches among total prefetches. A *valid prefetch* is a prefetch that successfully avoids the generation of a *page fault*. *Coverage* and *hit ratio* are calculated as follows:

$$\text{Coverage} = \frac{\text{Valid Prefetches}}{\text{Total Page Faults}}, \quad \text{Hit Ratio} = \frac{\text{Valid Prefetches}}{\text{Total Prefetches}} \tag{1}$$

A technique that shows a high *coverage* avoids a high percentage of page faults. As an example, prefetching all shared memory pages would achieve a high coverage, but at the cost of a low hit ratio. On the other side, a technique could provide a high *hit ratio* making only correct predictions, but covering only a low percentage of all page faults. A good prefetching strategy should aim to get both a high coverage, and a high hit ratio.

4.2 Results

Results obtained are shown for each application. The parameter $D = 14$ was used as the fixed size of the prediction window. This value is based on the number of nodes where the applications were ran, based on the assumption that the window size should be at least as big as the number of active processors in order to be able to reflect at least one action of each one of them. A further study is required to validate this assumption.

Results are presented in terms of the time taken by the system to complete the execution, including the time required to execute the prediction routine and the prefetching actions; and the coverage and hit ratio obtained for each value of the *page history* structure size, M.

LIFE. LIFE is a case of an extremely regular application. The shared memory access pattern induced by the matrix division alternates between reads and

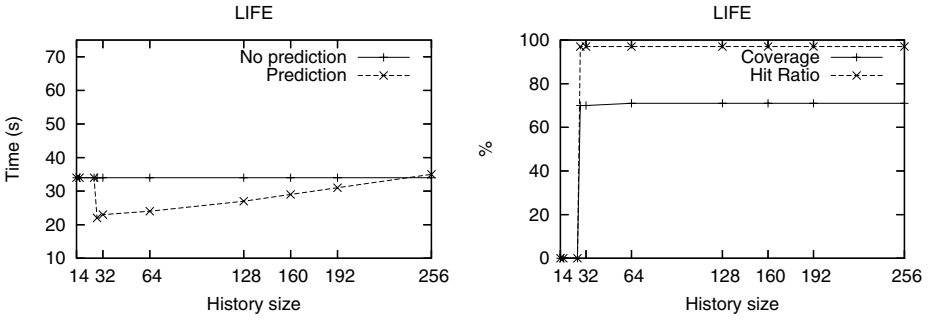

Fig. 2. Execution time, coverage and hit ratio for LIFE application

writes of different nodes in a repetitive sequence along iterations. Pages present a migratory pattern, in which by every two iterations, page ownership changes from one node to another, and then returns to the first one. This pattern is quite suitable for *history-based prefetching*, since the sequence of page accesses always follows the same cycle.

This case shows the benefit of a small size for the *page history* structure. Execution time linearly increases as the history size grows. On the other hand, the efficiency of the predictions is not affected as *coverage* and *hit ratio* remains the same. For history size values lower than 14, the techniques makes no predictions. For values greater than 150, the execution time increases over the no-prediction execution, but giving no improvement in the quality of the prediction. This is due to the greater amount of history that has to be transmitted every time the ownership of a page is transferred.

SHEAR. SHEAR is a case of a difficult application for this strategy: row-phases produce a uniform page access per node due to the row-assignment, but column-phases produce *false sharing*, since when sorting a column a node must access every page, and this page must be written by every node in the same phase. This produces a multiple-writer access pattern and an almost random order in the access sequence to a page, since nodes must compete for the access to a page every time a column is sorted.

The SHEAR application is a case where few predictions can be done, so the cost of searching for patterns through the page history in order to make predictions, and the additional page faults generated because of wrong predictions, begins to take importance. For small history size values, the execution time is almost the same as in the no-prediction execution, while for bigger values the execution time increases as a consequence of the page faults generated by wrong *read-write* predictions, a higher number of entries in the *page history* structure, and a bigger time taken to find patterns.

Coverage remains with a low value because only a little number of page faults are successfully avoided. Inside that little number, however, the *Hit Ratio* is high enough to show the accuracy of the predicting strategy and tends to stabilize as the *page history* size increases.

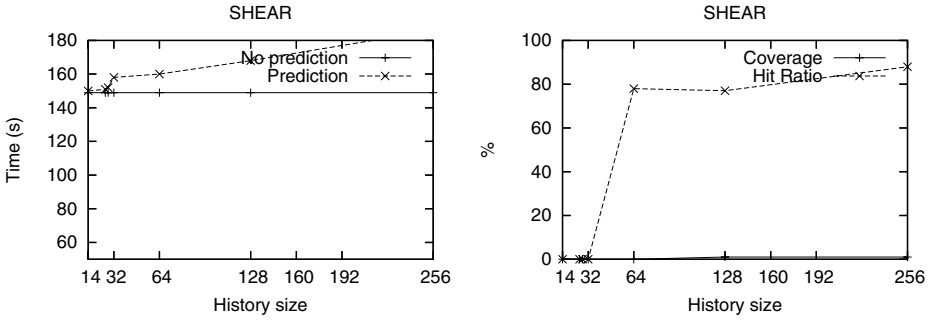

Fig. 3. Execution time, coverage and hit ratio for SHEAR application

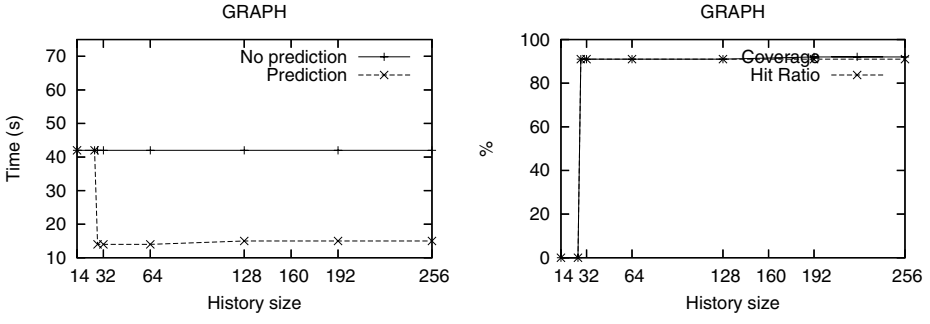

Fig. 4. Execution time, coverage and hit ratio for GRAPH application

GRAPH. GRAPH presents a case for a 1PMC pattern due to the *node-to-page* allocation. At each iteration, all nodes update the information of their vertexes regarding shortest distances to other vertexes, writing on their pages and, in the next iteration, that information is read by the other nodes to update their information on the next phase. This way, for every page there is only one node that writes on it, and every other node only reads from it, producing the *one-producer multiple-consumers* access pattern.

In this application, almost no page ownership is transferred among nodes because of the 1PMC pattern. Each node makes read access to pages owned by remote nodes, and writes only on locally owned pages. When *page history* size is increased the cost associated to *page history* transfers is not relevant, as *page history* is seldom transferred, so the execution time only varies depending on the additional time required to generate predictions.

Coverage is not affected by the *page history* size, and *hit ratio* barely increases, showing that the quality of the prediction is not affected by a bigger amount of information available.

4.3 Analysis

The results show that the size of the *page history* does not significatively increase the quality of the predictions made. *Coverage* and *hit ratio* are generally not

harmed by the storage of a small list of past access, providing that it is big enough to allow a pattern to be found. Once again the fixed parameter D of the window size, reflecting the number of nodes involved in the execution, has proved to be useful. In most applications, predictions can be made with a *page history* size being at least two times the window size used. A further study should prove this conjecture.

Applications can achieve a better performance by looking only at a small list of past accesses, rather than having bigger amounts or information. In all situations, maintaining a complete history of events will lead to a poor performance, as the time taken to transmit and search through the history overcomes the prediction improvement.

5 Conclusions and Future Work

This work presented an experimental study on finding an optimal size for the M parameter used in *history-based prefetching*. This parameter represents the size of the *page history* that is stored, and that is used to make predictions about the future shared memory access behavior of the nodes on a software DSM system.

Results show that the tested applications achieve a better performance when the *page history* structure stores a small portion of the most recent shared memory accesses made to each page. Large amounts of information lead to a poor performance when the page ownership, and therefore, the *page history* is repeatedly transferred among nodes, and has to be searched.

The quality of the prediction, measured in terms of *coverage* and *hit ratio*, is, in most cases, barely improved by a bigger *page history* size. In the applications tested, a small size of M provides a prediction efficiency almost as good as that obtained with a large size.

As a future work, a similar study has to be done to measure the influence of the D parameter, that represents the size of the prediction window used to find patterns. Our conjecture, based on the fixed size used in this work and the results obtained, is that the history size should be at least two times bigger than the window size, and probably not bigger.

References

1. Li K. and Hudak P.: Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems **7** (1989) 321–359
2. Cox A. L. and Dwarkadas S. and Keleher P. and Lu H. and Rajamony R. and Zwaenepoel W.: Software Versus Hardware Shared-Memory Implementation: A Case Study. In: Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94). (1994) 106–117
3. Pinto R. and Bianchini R. and De Amorim C. R.: Comparing Latency-Tolerance Techniques for Software DSM Systems. IEEE Transactions on Parallel and Distributed Systems **14** (2003) 1180–1190
4. Ruz C.: History-based Prefetching for Software DSM Systems. Master's thesis, Pontificia Universidad Católica de Chile (2005)

5. Bianchini R. and Kontothanassis L. I. and Pinto R. and De Maria M. and Abud M. and De Amorim C. L.: Hiding Communication Latency and Coherence Overhead in Software DSMs. In: Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII). (1996) 198–209
6. Bianchini R. and Pinto R. and De Amorim C.L.: Data Prefetching for Software DSMs. In: ICS '98: Proceedings of the 12th International Conference on Supercomputing, ACM Press (1998) 385–392
7. Lee S. and Yun H. and Lee J. and Maeng S.: Adaptive Prefetching Technique for Shared Virtual Memory. In: First IEEE International Symposium on Cluster Computing and the Grid, CCGRID, IEEE Computer Society (2001) 521–526
8. Karlsson M. and Stenström P.: Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared-Memory Systems. Journal of Parallel and Distributed Computing **43** (1997) 79–93
9. Monnerat L. R. and Bianchini R.: Efficiently Adapting to Sharing Patterns in Software DSMs. In: Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4). (1998) 289–299
10. Meza F. and Campos A. E. and Ruz C.: On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In: International Conference on Computational Science and Its Applications, ICCSA 2003. Number 2667 in Lecture Notes in Computer Science, Montreal, Canada, Springer-Verlag (2003) 967–976
11. Gardner M.: Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. Scientific American **223** (1970) 120–123 The original description of Conway's game of LIFE.
12. Sen S. and Scherson I. D. and Shamir A.: Shear Sort: A True Two-Dimensional Sorting Technique for VLSI Networks. In: ICPP. (1986) 903–908
13. Wilkinson B. and Allen M.: Parallel programming: techniques and applications using networked workstations and parallel computers. Prentice-Hall, Inc. (1999)