PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE

ESCUELA DE INGENIERIA

HISTORY-BASED PREFETCHING FOR SOFTWARE DISTRIBUTED SHARED MEMORY SYSTEMS

CRISTIAN DANIEL RUZ RUZ

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering

Advisor:
YADRAN ETEROVIC S. (ALVARO CAMPOS U.)

Santiago de Chile, April, 2005

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA
Departamento de Ciencia de la Computación

HISTORY-BASED PREFETCHING FOR SOFTWARE DISTRIBUTED SHARED MEMORY SYSTEMS

CRISTIAN DANIEL RUZ RUZ

Members of the Committee:

YADRAN ETEROVIC S.

DOMINGO MERY Q.

JOSÉ M. PIQUER G.

CRISTIAN VIAL E.

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering

Santiago de Chile, April 2005

To ...

ACKNOWLEDGMENTS

Thanks ...

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

RESUMEN

Este artculo es excelente.

ABSTRACT

This work presents *history-based prefetching*, a prediction-prefetching technique that collects the recent history of accesses to individual shared memory pages and uses that information to predict the next access to a page. When a prediction is made, consistency actions are taken and the page is *prefetched* from the remote node as if the access request had been done. The analysis is extended by searching an optimal value for the history size used by the prediction.

On correct predictions, this technique allows to hide the latency generated by page faults on the remote node when the access is effectively done. The prediction strategy is improved by adding support to recognize some previously defined memory access patterns.

Our experiments show that *history-based prefetching* can dramatically decrease latency and improve the performance of parallel applications that show a regular access pattern (between 30 and 60% execution time). Furthermore, applications with an irregular access pattern can also be benefitiated (around 2% execution time).

I. 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 (Li and Hudak, 1989).

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 (Cox, Dwarkadas, Keleher, Lu, Rajamony and Zwaenepoel, 1994; Pinto, Bianchini and de Amorim, 2003). To address these issues several latency-tolerance techniques have been introduced, including relaxed consistency protocols designed to reduce the amount of communication and coherence overhead, and multiple-writer protocols that reduce the effects of false sharing (Amza, Cox, Dwarkadas, Keleher, Lu, Rajamony, Yu and Zwaenepoel, 1996; Keleher, Cox and Zwaenepoel, 1992). Other approaches are runtime adaptation between update and invalidate coherence, adaptation between single-writer and multiple-writer protocols (Amza, Cox, Dwarkadas, Jin, Rajamani and Zwaenepoel, 1999), and detection of memory sharing patterns (Monnerat and Bianchini, 1998).

A technique closely related to adaptation is *prefetching*, which reduces latency by sending data to remote nodes in advance of the actual data access time. However, effective prefetching in software DSM systems can be quite complex for two main reasons: 1) it is offen difficult to predict future data accesses, and 2) prefetches generate significant overhead when issued unnecessarily (Pinto et al., 2003). Hence, a good prefetching technique must achieve a high hit ratio, and provide a way to

detect and avoid incorrect prefetches. Also, the overhead of making a prediction must be considered.

This work presents *history-based prefetching*, a prediction-prefetching strategy that collects the recent history of accesses to individual shared memory pages, and uses that information to predict the next access to a page through the identification of memory access patterns. When an access to a page is predicted, consistency actions are taken and the page is *prefetched* from the remote node as if the access had been done. This technique hides the latency generated by the page fault when the access is effectively done. The remote node does not have to wait for the page to arrive; it can use the prefetched copy immediately and continue working. A history update scheme avoids repeating wrong predictions when they are detected.

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*. Experiments were conducted to estimate an optimal value for this parameter.

The predictive strategy is presented in a general way. Although the prefetching implementation is shown over a sequential-consistent protocol, the concept can be applied to other consistency protocols as well.

A series of experiments were done with three applications running over a page-based DSM system, on a 8-nodes linux cluster. Results show that *history-based prefetching* is an effective technique to reduce latency in applications that show a regular memory access pattern, and thus improve their performance, even though sending more data

than a normal execution.

Regarding the optimal value for M, 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.

II. RELATED WORK

A large number of papers have been published on adaptation and prefetching in software DSM systems (Pinto et al., 2003; Amza et al., 1999; Monnerat and Bianchini, 1998; Lee, Yun, Lee and Maeng, 2001; Bianchini, Pinto and Amorim, 1998; Seidel, Bianchini and de Amorim, 1997; Amza, Cox, Rajamani and Zwaenepoel, 1997; Bianchini, Kontothanassis, Pinto, Maria, Abud and de Amorim, 1996; Karlsson and Stenström, 1997; Mueller, 1999). Amza *et al.* (Amza et al., 1999) presented an adaptive version of Treadmarks (Amza et al., 1996) that dynamically adapts between single-writer and multiple-writer modes, and also between invalidate and update coherence protocols. They also show *dynamic page aggregation*, where pages are dynamically grouped, and sent when a page fault happens inside the group (Amza et al., 1997). This page grouping has a similar effect to prefetching, though they do not consider sharing patterns.

Bianchini *et al.* have done important work in prefetching techniques for software DSM systems. They developed the technique *B+* (Bianchini et al., 1996) 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.

Seidel *et al.* proposed the *affinity entry consistency* protocol (Seidel et al., 1997), based on *entry consistency* (Bershad and Zekauskas, 1991), and introduced *Lock Acquirer Prediction* (LAP), a technique to predict the next acquirer of a lock using centralized managers to gather global information. *History-based prefetching* avoids the necessity of centralized information.

Bianchini *et al.* also presented the *Adaptive++* technique (Bianchini et al., 1998) 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.* (Lee et al., 2001) 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.

Monnerat and Bianchini (Monnerat and Bianchini, 1998) present a categorization of pages called *sharing pattern categorization* (SPC), based on the sharing behavior of each page. This approach uses association between lock variables and the data they protect. Their main ideas have been applied to the sharing patterns used by *history-based prefetching*.

Karlsson *et al.* (Karlsson and Stenström, 1997), 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.

III. 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, 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, which are called *execution phases*. Predictions are generated inside a barrier event to make sure that no other node may generate regular consistency messages, and avoid 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, as it is showed in Figure III.1. After that, the barrier is released.

History-based prediction introduces extra messages on wrong predictions. The processing time required to generate the predictions and to issue prefetches also has to be considered, but it will be shown that the benefits from useful predictions overweight the additional processing needed to generate them.

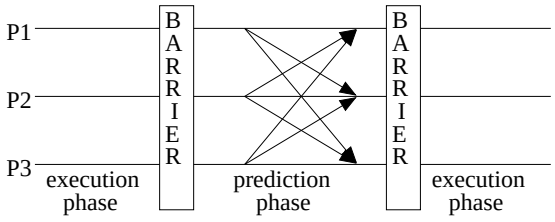

Figure III.1 Execution and prediction phases

The prefetching scheme considers three steps:

1. Collect access history information in a per-page basis.

2. Make predictions according to a prediction strategy using the access history or the identification of some defined page access patterns.

3. Execute prefetching actions and, if needed, consistency actions when a write access is predicted, to send the page to the destination nodes.

III.1 Page History

History-based prediction is based on 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 includes the following data:

- A pair `(host,access)` indicating the node that accessed the page and the access mode used. Access mode can take the values `r`, if the access is *read-only*, or `w` if it is *read-write*. This is represented as labels consisting of `host` and `access`. For example, label `2r` means that node 2 accessed the page in *read-only* mode.

- `barrier`. Number of the execution phase when the access was actually made.

- A list of pairs `(predDest,predMode)`. Each pair indicates a prediction that was made in the past, when this history element was the last in the *page history*.

Because the number of accesses to a page could be very large, only the last M *history elements* are kept, reflecting the last M accesses to the page. Our assumption is that only the most recent accesses are relevant in order to predict the immediate page behavior. The parameter M determines the amount of history that is kept to take prediction decisions, and that is sent to other nodes along with the ownership of the related 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*. This scheme can be used with any consistency protocol that preserves a unique owner.

The *page history* of page p is updated as follows:

1. When owner `j` receives a read-fault notice from remote node `k` on page p, then a new *history element* labeled `kr` is added to the history of page p. Only the first read access by the remote node is recorded, as subsecuent reads will not generate read-faults and will not be noticed by the owner.

2. When owner `j` generates a write-fault on page p, then the previous element, if it exists, is examined. If the previous element is labeled `jr` and it belongs to the same barrier phase as the actual access, then the previous element is replaced by the new one, labeled `jw`. If it does not belong to the same barrier phase, a new element labeled `jw` is added to the history of page p.

3. When owner `j` receives a write-fault notice from remote node `k` on page p, then the previous element, if it exists, is examined. If the previous element is

labeled `kr` and it belongs to the same barrier phase as the actual access, then the previous element is replaced by the new one, labeled `kw`. If it does not belong to the same barrier phase, a new element labeled `kw` is added to the history of page p. After that, the *page history* is transferred to the new owner `k`.

The last two cases, where element `xr` is replaced by `xw`, represent the fact that applications often read from a variable and then immediately write on the same position during the same execution phase. This is called *history compression* and is done to get a more precise record of the page behavior.

When remote reads are detected, only the first read access of each remote node is recorded, as subsecuent reads will not generate read-faults, so they will not be noticed by the owner. This is not important for the *page history* because the sharing pattern is based on information about which nodes read the page, and not how many read accesses were done. Such complete information would lead to an excessive processing overhead, because every access would have to be detected.

III.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, and makes a list of predicted destinations. The pseudocode is shown in Figure III.2.

The `Prediction` routine attempts to find a pattern on the *page history*, by looking for the most recent repetition of the last D accesses seen, and predicting the same

```
Node k:
   for each page p owned by k
      if( p.Prediction() )
         p.ExecutePrediction()
```

Figure III.2 Pseudocode executed by each node to generate predictions

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 is hoped to be correct in most situations.

In some cases, more than one future read access can be predicted, and a list of read accesses is generated instead of only one. In that case, one copy of the page is sent to every predicted destination.

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 $W_S = \langle h_{M-(D-1)-S}, h_{M-(D-2)-S}, \ldots$ 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 Figure III.3. The Pseudocode is presented in Figure III.4.

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.

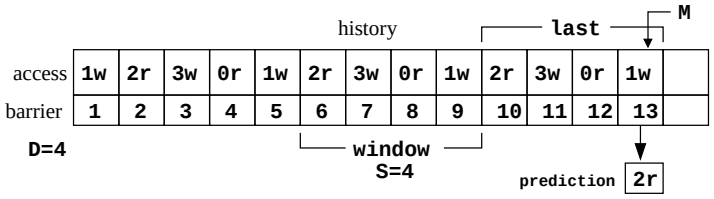

Figure III.3 Prediction of access `2r` with $D = 4$. The match was found when $S = 4$.

```
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;
```

Figure III.4 Pseudocode to generate a prediction for one page

III.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 III.2 can be used.

Pattern detection is based on the categorization described by Monnerat and Bianchini (Monnerat and Bianchini, 1998), where page behavior can be classified as 1PMC, MIG and MW.

- 1PMC is a *one producer - multiple consumers* pattern. It may be recognized when the last access is a local *read-write*, and the previous are two or more consecutive *read-only* access. In this case the generated predictions are *read-only* accesses to each node that reads the page. An example is shown in Figure III.5. Node 1 is the only one that writes on the page, and nodes 0, 2 and 3 are consumers.

  Another case of 1PMC pattern is when the last accesses are two or more consecutive *read-only* accesses. The generated prediction is a *read-write* to the last node that wrote the page. In the example shown in Figure III.5, the prediction for the *history element* `0r` in barrier 6, would have been `1w`.

- MIG is the pattern where one page is owned by different nodes in different execution phases. In this case no special action is taken, since the basic prediction strategy is capable of dealing with that case. An example is shown in Figure III.6.

- MW is the pattern where one page is owned by different nodes in the same

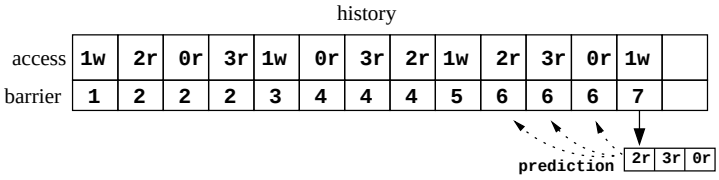

Figure III.5 1PMC behavior. *Read-only* copies are sent to readers 0,2,3 by the writer 1.

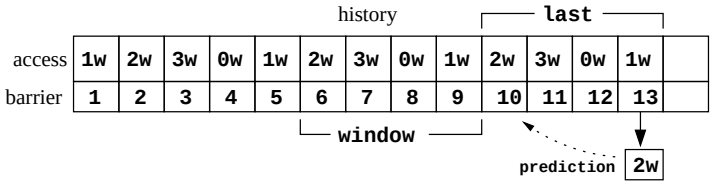

Figure III.6 MIG behavior. The page is owned by a different node at each execution phase.

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 may be avoided and the $N-1$ remaining writes will still generate page faults. An example is shown in Figure III.7.

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, as it is shown in Figure III.8. In this case, the prediction strategy should avoid making any prediction. A discussion is presented in section IV.5.

As it happens in the `Prediction` routine, if the page is in `PREFETCHED` state, then no prediction is made.

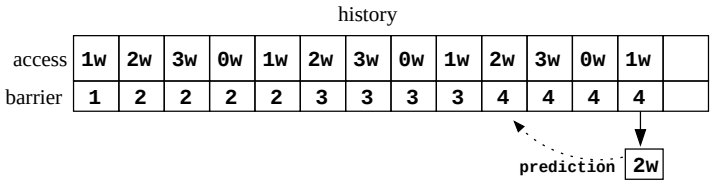

Figure III.7 MW behavior. Only `2w` can be predicted without generating page faults.

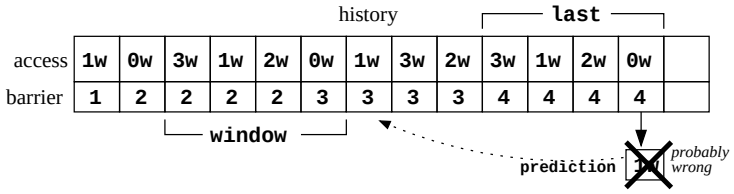

Figure III.8 MW behavior. `2w` is predicted but the next node to access the page may be anyone.

III.3.1 Page History Size

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.

IV. PREFETCHING EXECUTION

When a prediction has been made for a page, prefetching actions are executed in order to send the page to the next node that is supposed to make an access to it. The specific prefetching actions to be taken depend on the underlying memory consistency protocol used, as the objective is to send a copy of the page as if the remote node had made a request for it.

The implementation described below considers a sequential-consistent protocol that uses invalidation coherence, and sends three kind of messages: `req` messages (to make page requests due to page faults), `inv` messages (to synchronuosly invalidate remote copies) and large `page` messages (to send copies of one shared memory page). At any given moment a page has a unique *owner*, and may be in one of three different status: `INVALID`, `RO` and `RW`. The prefetching scheme considers two additional states: `PREFETCHED_RO` and `PREFETCHED_RW`.

Different actions are taken if the prediction is for a *read-only* access or for a *read-write* access.

IV.1 *Read-only* predictions

The algorithm for executing a *read-only* prediction sets the owner copy on `READONLY` mode. The destination node is added to the `copySet` of the owner, and the copy is sent.

Only one message is required and no local or remote invalidations are made, so no additional page faults are incurred when executing a possibly unnecessary *read-only*

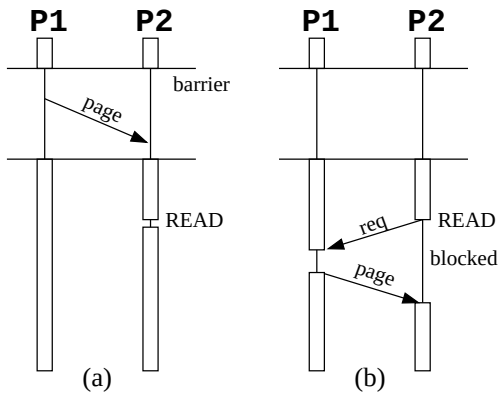

Figure IV.1 (a) RO prediction; (b) Normal execution

```
Prefetch_RO(dest):
   setMode(READONLY);
   addToCopySet(dest);
   sendPage(dest);
```

Figure IV.2 Pseudocode executed by the sender on a prefetch

prediction. If the destination is correct, but the access is *read-write*, then the only consequence is an additional `inv` message. If the access mode is correct, but the copy is sent to an incorrect node, then the additional action is the unnecesary `page` sent. If both the destination and the access mode are incorrect, the additional actions are one `inv` message and one `page` message sent.

On the other hand, correct predictions hide the delay produced by blocking the requesting node while the owner is found and the requested copy is sent as shown in Figure IV.1.

```
Prefetch_Recv(sender):
   storeCopy();
   setMode(PREFETCHED_RO);
   setProbOwner(sender);
```

Figure IV.3 Pseudocode executed by the receiver on a prefetch

```
PageFault_RO(p):
   if(p.status == PREFETCHED_RO || p.status == PREFETCHED_RW)
      setMode(RO);
      return;
   else
      sendRequest(RO);
```

Figure IV.4 Pseudocode executed by the receiver on a *read-only* page fault

IV.2 *Read-write* predictions

Read-write predictions are more intrusive in the state of the page. Every copy on the `copySet`, as well as the local copy are invalidated, and the only valid copy is sent to the destination node, as well as the *page history* and the ownership of the page. The number of messages depends on the size of the `copySet`, but at least involves one `page` message to send the valid copy to the new owner. The time needed to invalidate the remote copies also increases when synchronous invalidations are used.

When a *read-write* prediction is sent to an incorrect node, at least one more `page` message is necessary to send the valid copy to the correct node. If the access mode was incorrect, then the `inv` messages sent are also unnecessary. If both access mode and destination node are incorrect, then one additional `page` message is incurred.

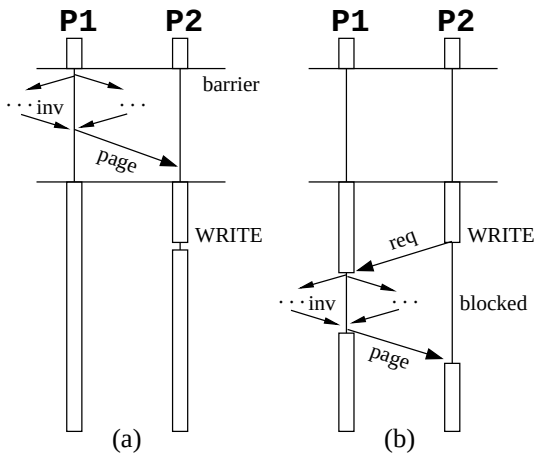

Figure IV.5 (a) RO prediction; (b) Normal execution

```
Prefetch_RW(dest):
  setMode(INVALID);
  invalidate(copySet);
  sendPageHistory(dest);
  sendPage(dest);
  setProbOwner(dest);
```

Figure IV.6 Pseudocode executed by the sender on a prefetch

The worst case is when the correct node was the one that originally had the page ownership, since the correct action would have been keeping the page and do not send any message. The cost in this case is at least two additional `page` messages, and all the `inv` messages if the access mode was also incorrect.

IV.3 Prefetching

Prefetching is done concurrently by the nodes when they have arrived to a barrier and before leaving it. This ensures that no regular consistency message are being

```
Prefetch_Recv(sender)
  storeCopy();
  setMode(PREFETCHED_RW);
  storeHistory();
  setProbOwner(localNode);
```

```
PageFault_RW(p):
  if(p.status == PREFETCHED_RW)
     setMode(RW);
     return;
  else
     sendRequest(p,RW);
```

Figure IV.7 Pseudocode executed by the receiver on a prefetch, and on a *read-write* page fault

sent while predictions are made. The execution of no other action than prediction and prefetching in this phase also decreases the time that nodes would block due to interruptions if page faults could be generated.

Prefetching actions over one individual page should be mutually exclusive. This is enforced by the property that the owner is the only one that can make predictions about a page, and at any time there is only one owner for each page.

Upon reception of a prefetch message, the received copy is stored and the page turns to `PREFETCHED` mode. In that state, the page copy is valid, but it is protected against read and write operations, so it can be detected if an access to the page really happens, and then validate the prediction. The page `probOwner` entry is updated depending on the message received. If the prediction is *read-only*, then the `probOwner` is the node that sent the copy. If the prediction is *read-write*, the `probOwner` is the local node.

IV.4 Correctness

A good prediction technique should detect when wrong predictions were made and avoid repeating them. In *history-based prefetching* this is accomplished by the *page history* update mechanism. The *page history* data structure records only actual accesses to a page. So, when a wrong prediction is made, the corresponding prefetching action will not show up in the *page history*. At the next execution of the predictive routine, the effect of the wrong prediction will not be found among the more recent elements of the *page history* and the wrong prediction will not be repeated. This also allows adaptation to different patterns of sharing memory.

This mechanism also permits to take account of wrong predictions, helping in the evaluation of the runtime effectiveness of *history-based prefetching*. When the effectiveness goes down below a defined threshold, prefetching can be stopped to avoid degrading the performance of the application.

IV.5 Cost of predictions

The cost analysis for *history-based prefetching* can be done focussing in two aspects: number of messages sent, and processor blocked time.

On successful predictions, each prefetching action effectively avoids one page fault on the remote node, and the subsequent request message to the probable owner, so at least one requesting message is avoided. However, in most executions wrong predictions can be made, and each one of them generates more messages than if the prediction had not been made, as was discussed in the implementation details

for *read-only* and *read-write* prefetching. This makes that a even a small percent of wrong predictions will generate more messages than an execution with no predictions at all.

On the other hand, the advantage of *history-based prefetching* is that the time that processors are blocked during *execution phases* is decreased even though more messages could be sent.

When one page access is correctly predicted not only the request message is avoided, but the delay that is produced by the page fault and the time that the processor has to wait for the remote request to be completed is also avoided, as shown in Figures IV.1 and IV.5. This blocked time also depends on the workload that the page owner may have before the request can be processed, and the time required to find the real owner of the page, both of which are not relevant in *prediction phases* as copies are directly sent to their destinations, and processors are not interrupted from doing their computation.

V. EXPERIMENTS

The experiments were executed on a platform of 8 Pentium IV processors, 256MB RAM, running linux Fedora Core 1. All computers execute the applications over DSM-PEPE (Meza, Campos and Ruz, 2003), a page-based software DSM system designed to execute parallel applications over a shared-memory environment on multicomputers and different consistency protocols.

Tested applications are: LIFE, SHEAR, and GRAPH.

- LIFE is an implementation of Conway's *Game of Life* (Gardner, 1970) on a 2048×2048 circular matrix, a cellular automaton that represents the evolution of a population inside a grid. The paralellization 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. This is an example of a regular application, where the order of read access is the same through all the execution.

- SHEAR is an implementation of the *shearsort* algorithm (Sen, Scherson and Shamir, 1986) to sort integers inside a 1024×1024 matrix. The execution goes through a fixed number of alternate row-phases and column-phases. In row-phases, odd rows are sorted increasingly and even rows are sorted decreasingly. In column-phases, all columns are sorted increasingly. At the end, the sorted array can be read downwards, from left to right in odd rows, and right to left in even rows. The paralellization is done through stripes. At every iteration, each node works over a fixed set of consecutive rows, or consecutive columns.

This application has a partially regular memory access behavior, as two sharing patterns alternate in execution phases.

– GRAPH is a distributed *single-source shortest-path* search on a graph of N vertexes on shared memory (Wilkinson and Allen, 1999). Each node is assigned a fixed set of vertexes to evaluate. On the kth iteration, nodes computes 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. This application was selected to show an irregular access pattern, where the information to read depends on the particular path found on the graph.

V.1 Methodology

Each experiment was executed under a sequential-consistent protocol, with *history-based prefetching*, and differente values for M, and compared to a normal execution without any kind of prefetching. Statistics collected in each case include execution time, locally-generated read-faults and write-faults, remote read-faults, remote write-faults, barriers executed, prefetches done for *read-only* and *read-write* modes, and correct prefetches detected. Also, the number and size of messages sent were taken into account.

The metrics used to evaluate the efectiveness of the proposed technique are *coverage* and *hit ratio*, as it has been used in previous related works (Pinto et al., 2003). *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:

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

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.

V.2 Results

Results obtained are shown in two stages for each application. In the first stage, different values for M were tested, to find optimal values of coverage and hit ratio. In the second stage, a fixed value $M = 32$ was used to find a detailed description of the cost of the prediction strategy, in terms of page faults incurred and avoided, and predictions made.

In both cases, the value $D = 8$ was used for 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 the number of active processors in order to be able to reflect at least one action of each one of them. Certainly, a further study is required to validate this assumption.

Results are presented in terms of the time taken by the system to complete the

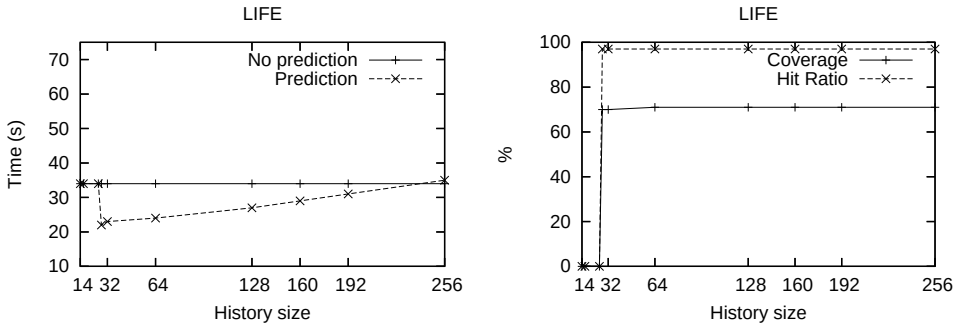

Figure V.1 Execution time, coverage and hit ratio for LIFE application

execution, including the time required to execute the prediction routine and the prefetching actions; the average number of read and write-faults generated by each node, and those received as remote requests; the average number of *read-only* and *read-write* predictions issued; and, finally, the average number of messages sent by each node and the amount of bytes transferred. Each parameter is compared to a normal execution with neither prediction nor prefetching actions, and the difference is shown below.

V.2.1 LIFE

LIFE is a case of an extremely regular application. The shared memory access pattern induced by the matrix division alternates between reads and writes of different nodes in a repetitive sequence along iterations. Pages present a migratory pattern, in which 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 one of the best benefits that *history-based prefetching* can achieve in regular applications, as can be seen in Table V.1 obtaining a 23% of improvement

Table V.1 Detailed results for LIFE application

LIFE 2048x2048	normal	prediction	DIFF	%DIFF
Time (s)	45.215	34.712	10.503	23.23 %
Local PF-Read	12,981	2,259	10,722	82.59 %
Local PF-Write	12,784	2,045	10,739	84.01 %
Remote PF-Read	14,982	4,261	10,721	71.56 %
Remote PF-Write	37,120	5,077	32,043	86.32 %
Predictions RO		11,122		
Predictions RW		10,867		
Msg. SENT	116,501	85,213	31,288	26.86 %
Msg. SENT size	116,433,918	126,181,569	-9,747,536	-8.37 %

Coverage	83,30 %
Hit Ratio	97,60 %

in execution time. A regular access pattern allows a very high *hit ratio*, due to repeateness of the pattern, making almost every prediction to be correct, and 83% less page faults generated. The presence of a unique pattern also gives a high *coverage* because prefetching can begin immediately when there is enough information to make predictions. Hence, effective prefetching begins after D iterations.

This case also shows the benefit of a small size for the *page history* structure. Execution time linearly increases as the history size grows, as seen in Figure V.1 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.

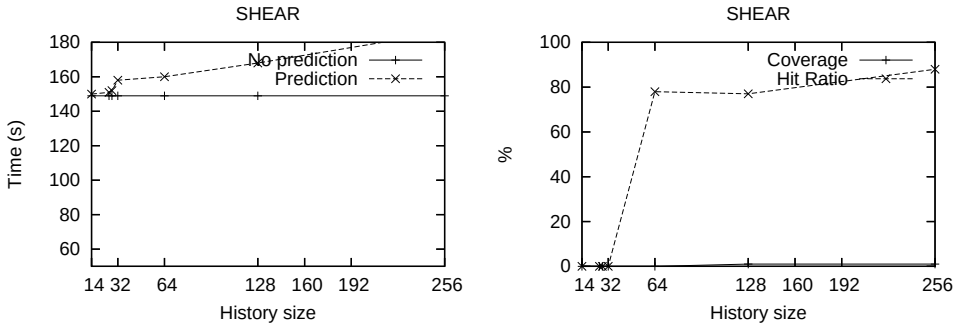

Figure V.2 Execution time, coverage and hit ratio for SHEAR application

V.2.2 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 execution gives little improvement in favour of *history-based prefetching*, as seen on Table V.2. The number of *write-faults* increase due mostly to wrong *read-write* predictions and the consequent additional page faults generated by the wrong invalidations. In this case, the technique can not make much predictions since every possible prediction made at the beggining of a column-phase has a great probability of being wrong, and in the case that predictions could be right, at most one of the `Nnodes`-1 *write-faults* can be avoided, producing a low *coverage*. Even in those little cases, a 79% *hit ratio* is achieved, and a small benefit in page faults, and messages sent is obtained. Performance is not dramatically harmed by the prefetching scheme, even though more information is sent.

Table V.2 Results for SHEAR application

SHEAR 1024x1024	normal	prediction	DIFF	%DIFF
Time (s)	64.88	64.94	-0.06	-0.09 %
Local PF-Read	3,108	2,924	184	5.92 %
Local PF-Write	3,108	3,120	-12	-0.39 %
Remote PF-Read	14,680	14,036	644	4.39 %
Remote PF-Write	3,108	2,924	184	5.92 %
Predictions RO		0		
Predictions RW		216		
Msg. SENT	33,465	32,149	1,316	3.93 %
Msg. SENT size	28,253,004	29,628,908	-1,375,904	-4.87 %

Coverage	2,77 %
Hit Ratio	79,63 %

Figure V.2 shows that 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.

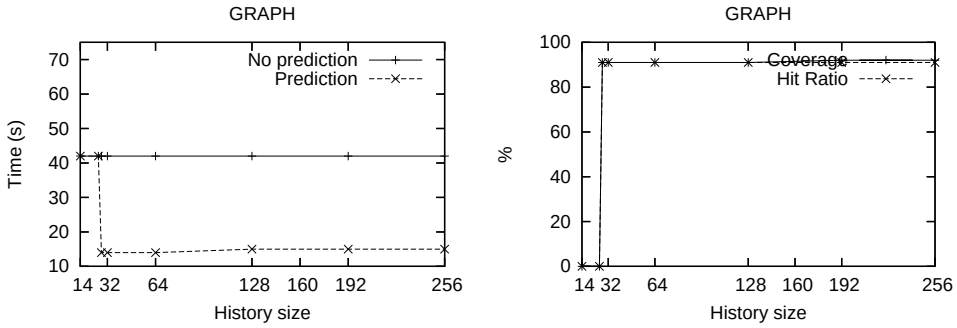

Figure V.3 Execution time, coverage and hit ratio for GRAPH application

V.2.3 GRAPH

GRAPH presents a case for a 1PMC pattern due to the *node-to-page* allocation. In each iteration, 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.

The benefits from the specialized detection of this pattern are presented in the results obtained, which can be seen in Table V.3 A 61% less execution time is the consequence of a clear access pattern and a high *hit ratio* due to a regular behavior and the prefetching actions defined. Also, a high *coverage* is obtained, elimating 87% the page faults.

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 trans-

Table V.3 Results for GRAPH application

GRAPH 128 vertex	normal	prediction	DIFF	%DIFF
Time (s)	39.16	15.03	24.13	61.61 %
Local PF-Read	17,791	275	17,516	98.45 %
Local PF-Write	2,558	2,228	330	12.90 %
Remote PF-Read	17,896	380	17,516	97.88 %
Remote PF-Write	18	18	0	0 %
Predictions RO		18,044		
Predictions RW		330		
Msg. SENT	72,242	72,769	-527	-0,73 %
Msg. SENT size	74,690,154	76,872,079	2,181,925	-2,92 %

Coverage	87,7 %
Hit Ratio	97,1 %

ferred, so the execution time only varies depending on the additional time required to generate predictions. The results are shown in Figure V.3.

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

V.3 Analysis

General analysis is done in terms of application performance and page faults generated, as well as network traffic, and history size.

Execution Time and Page Faults Two of the studied applications showed a general decrease in execution time. This is closely related to the number of *page faults*

produced. Applications that decrease their *page faults* due to prediction obtain a benefit in execution time. This happens because processors that receive prefetched copies will not generate *page faults* on that page, so no requesting message is sent to other nodes, and they will not block to wait for answers. Latency generated by sending messages, blocking and waiting is hidden by the prefetching actions previously taken.

Experiments showed that not only the reduction of *page faults* leads to less execution time, but also the moment when pages are sent is also important. *History-based prefetching* is executed in special prefetching phases, when nodes are not doing application work. This is a gain in the sense that processors are not interrumped from doing useful job, helping to hide latency by doing separate prefetching phases.

Messages sent In LIFE and SHEAR, the number of messages is smaller due to the request messages avoided when a page fault occurs. The amount of bytes transmitted increases as a consequence of the incorrect predictions made and the useless pages sent in prefetching messages. In the case of GRAPH, the number of messages increases slightly. The reason is that most of the messages sent are related to confirmation of correct *read-only* predictions, and in this application all the messages sent are `PREDICTED_RO` messages. There are no `PREDICTED_RW` messages because all *read-write* predictions made are for the actual page owner. Every correct *read-only* prediction made involves two messages: a prefetching message and a confirmation message. The confirmation messages are small in comparison to the prefetching message, so the number of messages increases, but the number of bytes sent remains small.

History size 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.

VI. FUTURE WORK

The prediction scheme used by *history-based prefetching* can be extended to be used with other consistency protocols, and to improve other areas of distribute shared memory. In this work, the technique was presented and implemented over a sequential-consistent protocol, but the principles can be applied to other weaker consistency protocols. An interesting topic of research the application of *history-based prefetching* to a causal consistent protocol (Campos and Navarro, 2004). In lock-driven protocols like *entry consistency* (Bershad and Zekauskas, 1991), this technique could be applied to predict the behavior of *locks* instead of pages and, thus, update the pages needed by the remote node before it makes the lock request. Another lock-driven protocols like *scope consistency* (Iftode, Singh and Li, 1996), may be improved by predicting the lock in advance.

It is also necessary to study the impact of the parameters used for the size of the the prediction window used, to determine optimal values and improve predictions and execution time. 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.

Other improvements that can be made involve the pattern matching when searching in the *page history*. Pattern matching could be improved by adding support to detect similar patterns instead of exact matching. Support for other shared memory patterns can also be added to improve the hit ratio of the predictions made.

A broader study of the flexibility of this technique to be used in some non-regular application is still necessary to clearly define the family of applications that can be

improved by *history-based prefetching*.

VII. CONCLUSIONS

This work presented *history-based prefetching*, a prediction/prefetching technique that uses the recent history of individual shared memory pages to predict the next access to a shared memory page, sending that page to the future requesting node before it actually needs it. This way, page faults are avoided on the requesting node and the latency of waiting for the updated copy of the page is reduced. Support to detect some known page access patterns, like migratory, multiple-writer, and one producer-multiple consumers is added in the predictive scheme. Experiments showed that in applications where a regular access pattern is detected, it is possible to achieve a high hit ratio for predictions.

Using a good prediction strategy, latency can effectively be hidden and performance of the DSM applications can be greatly improved. Results for *history-based prefetching* show that sending pages in dedicated prefetching phases is better in terms of elapsed ime and induced latency, than stopping the computation on page faults and waiting for the requested pages to arrive, as happens in a non-prediction execution.

We have shown that *history-based prefetching* is an effective technique to improve the performance of some applications that show a regular access pattern. Still, the whole family of applications that can be benefitiated has to be determined, and it is object of a further study.

Optimal parameters for the window size also have to be found, and the prediction strategy must be fine-tuned. However the current results are encouraging, showing that this work can be extended to a wider range of applications, due to the high benefit in execution time obtained for two of the applications tested. Results for the

third application indicates that the time used to calculate the predictions, even in the case when little correct predictions are done, does not lead to a worse performance than the non-prediction execution, showing that the prediction strategy has a low overhead.

Results for the size of the *page history* structure 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.

In summary, we believe that *history-based prefetching* can be applied directly to existent DSM systems to improve their performance with applications showing regular access patterns.

BIBLIOGRAPHY

AMZA, C., COX, A. L., DWARKADAS, S., JIN, L.-J., RAJAMANI, K. and ZWAENEPOEL, W. (1999). Adaptive Protocols for Software Distributed Shared Memory, *Proc. of the IEEE, Special Issue on Distributed Shared Memory* **87**(3): 467–475.

URL: *citeseer.ist.psu.edu/amza99adaptive.html*

AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W. and ZWAENEPOEL, W. (1996). TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer* **29**(2): 18–28.

URL: *citeseer.ist.psu.edu/amza96treadmarks.html*

AMZA, C., COX, A. L., RAJAMANI, K. and ZWAENEPOEL, W. (1997). Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory., *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pp. 90–99.

URL: *citeseer.ist.psu.edu/amza97tradeoffs.html*

BERSHAD, B. N. and ZEKAUSKAS, M. J. (1991). Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, *Technical Report CMU-CS-91-170*, Carnegie Mellon University, Pittsburgh, PA (USA).

BIANCHINI, R., KONTOTHANASSIS, L. I., PINTO, R., MARIA, M. D., ABUD, M. and DE AMORIM, C. L. (1996). Hiding Communication Latency and Coherence Overhead in Software DSMs., *Proc. of the 7th Symp. on Architectural Support for*

Programming Languages and Operating Systems (ASPLOSVII), pp. 198–209.
URL: *citeseer.ist.psu.edu/bianchini96hiding.html*

BIANCHINI, R., PINTO, R. and AMORIM, C. L. (1998). Data Prefetching for Software DSMs, *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, ACM Press, pp. 385–392.

CAMPOS, A. E. and NAVARRO, J. E. (2004). A page-coherent, causally consistent protocol for distributed shared memory., *Journal of Systems and Software* **72**(3): 305–319.

COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R. and ZWAENEPOEL, W. (1994). Software Versus Hardware Shared-Memory Implementation: A Case Study, *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pp. 106–117.
URL: *citeseer.ist.psu.edu/cox94software.html*

GARDNER, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life', *Scientific American* **223**(4): 120–123. The original description of Conway's game of LIFE.

IFTODE, L., SINGH, J. P. and LI, K. (1996). Scope Consistency: A Bridge between Release Consistency and Entry Consistency, *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pp. 277–287.
URL: *citeseer.ist.psu.edu/iftode96scope.html*

KARLSSON, M. and STENSTRÖM, P. (1997). Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared-Memory Systems, *Journal of Par-*

allel and Distributed Computing **43**(2): 79–93.
URL: *citeseer.ist.psu.edu/karlsson97effectiveness.html*

KELEHER, P., COX, A. L. and ZWAENEPOEL, W. (1992). Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Annual Int'l Symposium on Computer Architecture (ISCA'92)*, pp. 13–21.
URL: *citeseer.ist.psu.edu/keleher92lazy.html*

LEE, S.-K., YUN, H.-C., LEE, J. and MAENG, S. (2001). Adaptive Prefetching Technique for Shared Virtual Memory, *First IEEE International Symposium on Cluster Computing and the Grid, CCGRID*, IEEE Computer Society, pp. 521–526.

LI, K. and HUDAK, P. (1989). Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems* **7**(4): 321–359.

MEZA, F., CAMPOS, A. E. and RUZ, C. (2003). On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers., *International Conference on Computational Science and Its Applications, ICCSA 2003*, number 2667 in *Lecture Notes in Computer Science*, Springer-Verlag, Montreal, Canada, pp. 967–976.

MONNERAT, L. R. and BIANCHINI, R. (1998). Efficiently Adapting to Sharing Patterns in Software DSMs, *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pp. 289–299.
URL: *citeseer.ist.psu.edu/monnerat98efficiently.html*

MUELLER, F. (1999). Adaptive DSM—Runtime Behavior via Speculative Data Distribution, *Parallel and Distributed Processing - Workshop on Run-Time Systems for Parallel Programming*, Vol. 1586 of *Lecture Notes in Computer Science*, Springer,

pp. 553–567.

URL: *citeseer.ist.psu.edu/mueller99adaptive.html*

PINTO, R., BIANCHINI, R. and DE AMORIM, C. L. (2003). Comparing Latency-Tolerance Techniques for Software DSM Systems, *IEEE Transactions on Parallel and Distributed Systems* **14**(11): 1180–1190.

SEIDEL, C. B., BIANCHINI, R. and DE AMORIM, C. L. (1997). The Affinity Entry Consistency Protocol., *Proc. of the 1997 Int'l Conf. on Parallel Processing (ICPP'97)*, pp. 208–217.

SEN, S., SCHERSON, I. D. and SHAMIR, A. (1986). Shear Sort: A True Two-Dimensional Sorting Technique for VLSI Networks., *ICPP*, pp. 903–908.

WILKINSON, B. and ALLEN, M. (1999). *Parallel programming: techniques and applications using networked workstations and parallel computers*, Prentice-Hall, Inc.