# Data Prefetching for Software DSMs *

Ricardo Bianchini, Raquel Pinto, and Claudio L. Amorim

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

{ricardo,raquel,amorim}@cos.ufrj.br

## Abstract

In this paper we propose and evaluate the *Adaptive++* technique, a novel runtime-only data prefetching strategy for software-based distributed shared-memory systems (software DSMs). Adaptive++ improves the performance of regular parallel applications running on software DSMs by using the past history of memory access faults to adapt between repeated-phase and repeated-stride prefetching modes. Adaptive++ does not issue prefetches during periods when the application is not exhibiting one of these two types of behavior and is thus behaving irregularly. Through detailed execution-driven simulations of several applications, we show that our prefetching technique is very successful at reducing the data access overheads of regular applications running on the TreadMarks software DSM. Adaptive++ also reduces the overhead of applications that are not strictly regular but that exhibit periods of regularity. In terms of overall performance, our results show that Adaptive++ can provide speedup improvements as significant as 34% on 16 processors. A direct comparison against two runtime-only prefetching techniques proposed thus far shows that Adaptive++ is consistently competitive in terms of performance, while being able to optimize a larger set of applications. Our main conclusion is that Adaptive++ should definitely be considered by software DSM designers as an effective way of tolerating the overhead of remote data accesses.

## 1 Introduction

Software-based distributed shared-memory systems (software DSMs) combine the ease of shared-memory programming with the low cost of message-passing architectures. However, these systems often exhibit high remote data access latencies when running real parallel applications. Prefetching strategies can conceivably be used to reduce these latencies by performing the operations involved in accesses to remote data in advance of the actual data accesses. However, prefetching effectively in software DSMs can be quite complex for two main reasons: (1) it is often difficult to predict future data accesses and (2) prefetches generate significant overhead when issued unnecessarily. In fact, prefetching effectively in software DSMs is somewhat harder than in their hardware counterparts; the relatively simple techniques proposed for hardware DSMs [13] are usually not effective for software DSMs.

In order to reduce the remote data access overhead in software DSMs while avoiding these two problems, in this paper we propose and evaluate the *Adaptive++* technique, a novel prefetching technique that issues prefetches only when access faults can be predicted with reasonable confidence. More specifically, Adaptive++ is a runtime-only data prefetching strategy that improves the performance of regular applications without requiring the intervention of sophisticated users or compilers. The technique uses the past history of memory access faults to adapt between repeated-phase and repeated-stride prefetching modes. Adaptive++ does not issue prefetches when the application is not exhibiting one of these two types of behavior and is thus behaving irregularly.

Through detailed execution-driven simulations of several applications, we show that our technique is very successful at reducing the data access overheads of regular applications running on the TreadMarks software DSM [2]. Our results show that these applications can achieve data access overhead reductions of as much as 58%. Adaptive++ also reduces the overhead of applications that are not strictly regular but that exhibit periods of regularity. For these applications data access overhead reductions are more modest but still substantial. In terms of overall performance, our results demonstrate that Adaptive++ can provide speedup improvements as significant as 34% on 16 processors.

Adaptive++ is not the first runtime-only prefetching technique proposed thus far in the literature; the techniques of Bianchini *et al.* (B+) [3], Karlsson and Stenström (KS) [7], and Amza et al. (Dynamic Aggregation) [1] are the best-known previous proposals. The KS technique is similar in flavor but not directly comparable to Adaptive++. A direct comparison of Adaptive++ against the other two strategies shows that our technique performs as well or better than B+ for all applications in our suite. In addition, Adaptive++ performs better than Dynamic Aggregation for regular applications, while achieving performance comparable to that of Dynamic Aggregation for non-regular applications. In essence, our comparison shows that Adaptive++ is consistently competitive in terms of per-

*This research was supported by FINEP/Brazil grant no. 56/94/0399/00 and CNPq/Brazil.

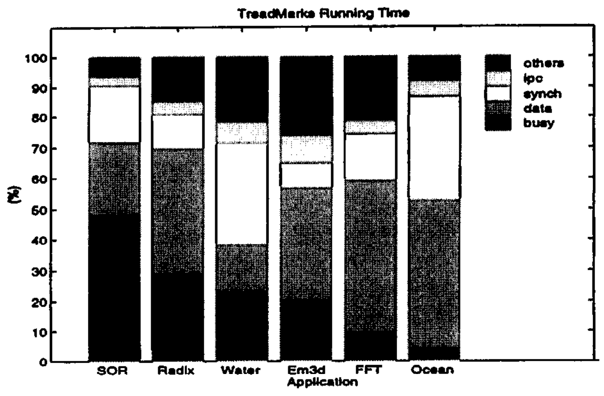

Figure 1: Run Times under TreadMarks on 16 processors.

formance, while being able to optimize a larger set of applications than the other techniques.

Our main conclusion is that Adaptive++ should definitely be considered by software DSM designers as an effective way of tolerating the overhead of remote data accesses. Furthermore, we believe that it is unlikely that any runtime-only technique can deal with irregular behavior satisfactorily. Tackling this type of behavior automatically requires the use of sophisticated compilers that can insert prefetches in the source code of the applications. However, prefetching compilers for software DSMs have received little attention to date and are beyond the scope of this paper.

The remainder of this paper is organized as follows. The next section motivates the paper by describing the main characteristics of TreadMarks and showing that remote data fetch overheads significantly degrade the performance of applications running on top of it. Section 3 describes our Adaptive++ prefetching technique. Section 4 describes our simulation methodology and the prefetching techniques we compare performance against. Section 5 presents our experimental results. In section 6 we describe related work. Finally, section 7 draws our conclusions.

## 2 Overheads in Software DSMs

Several software DSMs use virtual memory protection bits to enforce coherence at the page level. In order to minimize the impact of false sharing, these DSMs only guarantee memory consistency at synchronization points, and allow multiple processors to write the same page concurrently.

TreadMarks is an example of a system that enforces consistency lazily. In TreadMarks, page invalidation happens at lock acquire points, while the modifications (diffs) to an invalidated page are collected from previous writers at the time of the first access (fault) to the page. The modifications that the faulting processor must collect are determined by dividing the execution in *intervals* associated with synchronization operations and computing a *vector timestamp* for each of the intervals. A synchronization operation initiates a new interval. The vector timestamp describes a partial order between the intervals of different processors. Before the acquiring processor can continue execution, the diffs of intervals with smaller vector timestamps than the acquiring processor's current vector timestamp must be collected. The previous lock holder is responsible for comparing the acquiring processor's current vector timestamp with its own vector timestamp and sending back write notices, which indicate that a page has been modified in a particular interval. When a page fault occurs, the faulting processor consults its list of write notices to find out the diffs it needs to bring the page up-to-date. It then requests the corresponding diffs and waits for them to be (generated and) sent back. After receiving all the diffs requested, the faulting processor can then apply them in turn to its outdated copy of the page. A more detailed description of TreadMarks can be found in [8].

The main overheads in TreadMarks (and most other software DSMs) are related to communication latencies and coherence actions. As a result of these overheads, the speedups achieved by most of our applications running on top of TreadMarks range between 4 and 8 on 16 processors, which is somewhat low but not at all uncommon for applications running on software DSMs. Two applications exhibit speedups that are not within this range: Em3d and Ocean. Em3d achieves superlinear speedup as a result of its terrible cache behavior on 1 processor. Ocean does not perform well at all, achieving a speedup of about 2 on 16 processors.

Figure 1 presents a detailed view of the execution time performance of our applications running on top of TreadMarks on 16 processors. The bars in the figure show normalized execution time broken down into busy time, data fetch latency, synchronization time, IPC overhead, and other overheads. The latter category is comprised by TLB miss latency, write buffer stall time, interrupt time, and the most significant of these overheads, cache miss latency. The busy time represents the amount of useful work performed by the computation processor. Data fetch latency is a combination of coherence processing time and network latencies involved in fetching pages and diffs as a result of page access violations. Synchronization time represents the delays involved in waiting at barriers and lock acquires/releases, including the overhead of interval and write notice processing. IPC overhead accounts for the time the processor spends servicing requests coming from remote processors.

Figure 1 shows that most applications running on top of TreadMarks suffer severe remote data fetch and synchronization overheads. Prefetching techniques can be used to alleviate the data fetch overhead. Note however that prefetching cannot completely eliminate this overhead, since prefetching does not overlap the time to apply diffs and generate twins with useful computation. We find that the actual potential performance improvements prefetching can provide range from 9% to 32% for our applications, assuming that the synchronization overheads are not affected by prefetching. Given that prefetching may reduce the overhead of barriers by reducing load imbalance and may reduce the overhead of locks by speeding up critical sections, performance gains achievable via prefetching may be even greater. In order to approach these improvements, we propose the prefetching technique described next.

## 3 The *Adaptive++* Technique

### 3.1 Description

Adaptive++ predicts near-future remote data accesses and issues prefetches for these data prior to the actual accesses. It relies solely on runtime information on data accesses (the history of remote accesses of each node) and is intended to optimize the performance of regular applications. These applications exhibit two main types of behavior: (1) the set of remote data accesses performed during a phase of execution (delimited by consecutive barrier events) is repeated during a subsequent phase, or (2) the stride between the different remote accesses during a phase is repeated in other phases. Adaptive++ does not issue prefetches when the application is not exhibiting one of these behaviors, i.e. when "unexpected" remote accesses must be performed. Thus, in order to tackle regular applications, our technique adapts between two modes of operation: repeated-phase and repeated-stride modes.

The exact implementation of each of these modes is obviously dependent on the underlying software DSM. To make the description of the technique more concrete, in the following we detail the implementation of Adaptive++ for TreadMarks, while describing the different execution modes and the concept of "expected" remote accesses in the context of this implementation.

**Repeated-Phase Mode.** To determine which pages to prefetch for in this mode of operation, our technique maintains two lists: `previous-list` and `before-previous-list`. `previous-list` contains the ids of the pages that experienced faults outside of critical sections in the previous phase of execution, while `before-previous-list` records the faults that occurred outside of critical sections during the phase before the previous one. One of these lists is to be chosen as the `expected` set of faults, i.e. a prediction of the page faults that will likely be experienced by the processor in the next phase.

On the third barrier synchronization episode, the similarity between the `previous-list` and `before-previous-list` lists determines which list is to be chosen on barrier events. Similar lists determine that the list corresponding to the previous phase is to be chosen on every subsequent barrier episode. When the lists are not similar, the one corresponding to the phase before the last is to be chosen every time. Two lists are deemed similar if more than 50% of their page ids belong to both lists. Note that this strategy only tries to determine the similarity between two consecutive phases, and assume that similar phases alternate if the phases are dissimilar. The reason why we do this is that it is rare for applications to exhibit similar phases that are not consecutive or alternating.

**Repeated-Stride Mode.** Pages to prefetch for are chosen differently under the repeated-stride mode of operation, although this mode also involves the two lists of page faults and chooses the one to use based on their similarity. This mode uses the most frequent page fault stride of the chosen list to determine the pages to prefetch for in the next phase. The ordered list of `expected` faults is comprised by all the pages that are a multiple of this stride away from the page faulted on when the stride is first detected during the phase.

Note that in repeated-stride mode Adaptive++ may issue prefetches for pages that were not accessed before and so prefetches may be issued for whole pages as well as their diffs. In repeated-phase mode Adaptive++ only issues prefetches for pages that have been touched before, so it only issues prefetches for diffs.

**Picking a Mode.** The decision of which mode to apply to a phase is taken during the barrier episode that starts the phase. The decision is based on the technique that is most likely to be adequate for the phase. If no technique seems appropriate, prefetching is avoided at least until the next barrier episode.

The metric that determines a technique's potential to succeed is different for each mode of operation. For the repeated-phase mode, the metric is the percentage of useful prefetches (the ones that prevented remote operations on faults) this mode issued (if it was the chosen mode in the previous phase) or would have issued (if it was not the chosen mode in the previous phase). For the repeated-stride mode, the metric is the frequency of the most common page access stride observed in the chosen list of faults. On every barrier event, the mode to be used in the next phase is the one that leads to the largest value of its metric. In case of a tie, the repeated-phase mode is the one of choice. Picking a mode to use obviously involves some overhead, which our detailed simulations show is almost always completely overlapped with the barrier overhead.

**Issuing Prefetches.** In the repeated-phase mode, a user-defined number of pages (24 pages in our experiments) whose ids belong in the `expected` list is prefetched for right after a barrier crossing point. In addition, the repeated-phase mode tries to issue prefetches for another user-defined number of pages (4 pages in our experiments) following the faulting page in `expected`, provided that the fault occurred outside of critical sections. Prefetches are only issued for: a) pages that are not already valid in the local memory; b) pages for which the same prefetches have not been completed (i.e. the required data is not in the local memory); and c) pages for which the same prefetches have been issued but are still outstanding. No action is taken on faults occurring inside of critical sections or on pages that are not in `expected`.

In contrast with the repeated-phase mode, no prefetches are issued at the barrier point in the repeated-stride mode, since at that point it is still not possible to determine the `expected` set of faults. On faults outside a critical section, the repeated-stride mode tries to issue prefetches for yet another user-defined number of pages (4 pages in our experiments) following the faulting page in `expected`. Again, prefetches are only issued on faults to pages found in `expected`, provided that conditions a, b, and c are met.

**Receiving Prefetch Replies.** The prefetch replies are saved until an actual access to the page is made by the processor, at which point any diffs that had not been prefetched (or that are yet to be received) are collected, all the diffs (prefetched and otherwise) are applied to the outdated version of the page, and the page is made valid. Note that an access fault is experienced by the node, even if all the necessary diffs had been received. Prefetches are issued on this type of fault as well as on faults for which no diffs were prefetched, if the faulting page belongs in the `expected` list of faults. Adaptive++ waits for all prefetch replies to have been received before crossing a subsequent synchronization point.

Complete and detailed justifications for each of the design decisions behind the Adaptive++ technique can be found in [4].

# 4 Methodology

## 4.1 Simulation and Workload

Our simulator consists of two parts: (1) a front end, MINT [12], that simulates the execution of the computation processors; and (2) a back end that simulates the software DSM protocol (TreadMarks and various prefetching extensions to it) and the memory system (finite-sized TLBs, write buffers, and caches, network transfer costs including contention effects, and memory access costs including contention effects) in great detail. We simulate a network of workstations with 16 nodes. Each node consists of a computation pro-

| System Constant Name | Default Value |
|---|---|
| Number of processors | 16 |
| TLB size | 128 entries |
| TLB fill service time | 100 cycles |
| All interrupts | 4000 cycles |
| Page size | 4K bytes |
| Total cache per processor | 256K bytes |
| Write buffer size | 4 entries |
| Cache line size | 32 bytes |
| Memory setup time | 18 cycles |
| Memory access time | 1.25 cycles/word |
| PCI setup time | 18 cycles |
| PCI access time | 3 cycles/word |
| Network path width | 16 bits (bidir) |
| Messaging overhead | 400 cycles |
| Switch latency | 4 cycles |
| Wire latency | 2 cycles |
| List processing | 6 cycles/element |
| Page twinning | 5 cycles/word + mem accesses |
| Diff creation and application | 7 cycles/word + mem accesses |

Table 1: Default Values for Parameters. 1 cycle = 5 ns.

cessor, a write buffer, a first-level direct-mapped data cache (all instructions are assumed to take 1 cycle), local memory, and a mesh network router (using wormhole routing). Table 1 summarizes the default parameters used in our simulations.

We report results for six representative parallel programs: Em3d, Water-nsquared, FFT, Radix, Ocean, and SOR. Em3d is from UC Berkeley. Water-nsquared, FFT, Radix, and Ocean are from the Splash-2 suite [14]. SOR is from the University of Rochester.

Em3d simulates electromagnetic wave propagation through 3D objects. We simulate 40064 electric and magnetic objects, each connected randomly to 24 other objects with a 0.1% probability that neighboring objects reside in different nodes. The interactions between objects are simulated for 40 iterations. Water-nsquared is a molecular dynamics simulation computing inter- and intra-molecule forces for a set of water molecules. Interactions are computed using an $O(n^2)$ algorithm. The algorithm is run for 10 steps and the input size used is 512 molecules. FFT performs a complex 1-D FFT that is optimized to reduce interprocessor communication. The data set consists of 1M data points to be transformed, and another group of 1M points called roots of units. Each of these groups of points is organized as a $256 \times 256$ matrix. Ocean studies large-scale ocean movements based on eddy and boundary currents. We simulate a $258 \times 258$ ocean grid. Radix is an integer radix sort kernel. The algorithm is iterative, performing one iteration per digit of the 4M keys. SOR performs successive over-relaxation of a $256 \times 640$ matrix of doubles for 100 iterations.

The applications in our suite exhibit widely varying page fault behaviors that should provide for interesting comparisons; from the highly-regular Em3d, SOR, and FFT, to the highly-irregular Radix, and the mixed Ocean and Water-nsquared. For SOR and Em3d, Adaptive++ runs in repeated-phase mode all the time, while for FFT it runs in repeated-stride mode just about all the time. For the other applications Adaptive++ applies both modes of operation.

## 4.2 Comparable Prefetching Techniques

**The B+ Technique.** B+ is a straightforward prefetching technique that we proposed in [3]. B+ uses page invalidations to guide prefetching. The technique assumes that a page that has been recently accessed by a processor and is later invalidated by another one will likely be referenced again in the near future. Thus, the B+ technique prefetches diffs for each of the pages invalidated at synchronization points, provided that the page is currently valid at the local node. Prefetches are issued right after lock acquire and barrier events. Just like in Adaptive++, in B+ all prefetch replies must have been received before crossing a subsequent synchronization point.

B+ differs from Adaptive++ in several important ways: a) B+ issues prefetches after lock acquire and barrier points, while Adaptive++ does not prefetch after lock acquires and does prefetch on access faults; b) B+ is driven by page invalidations, while both Adaptive++ modes of operation are driven by the actual access faults; c) B+ does not stop prefetching for irregular applications, while Adaptive++ does; and d) B+ cannot deal with cold start faults, while Adaptive++ can tackle regular cold start accesses.

**The Dynamic Aggregation Technique.** This technique groups pages based on the access faults experienced by processors, so that the system's fetch unit can be enlarged without incurring the harmful effects of false sharing. Prefetching takes place because a group of pages is fetched together on an access fault to any of the pages in the group. Only the faulting page is turned valid when the requested data arrive however, so that changes in fault behavior can be tracked. After having fetched a group of pages, the group is destroyed. Again, all prefetch replies in Dynamic Aggregation must be received before crossing a subsequent synchronization point.

Page groups are computed at each synchronization point based on the access faults experienced by the processor prior to the synchronization. A fault occurs on every first access to an invalid page. All access faults form a fault sequence that is divided into groups in such a way that a group is completely filled before the next is created. The maximum size of page groups is defined by the user. After a large number of experiments with different maximum group sizes, we determined that the best value of this parameter for our applications is 5, i.e. we issue prefetches for 4 pages on an access fault that requires remote communication.

Besides prefetching, Dynamic Aggregation also improves performance by combining multiple diff requests to the same processors, thus reducing the number of messages involved in fetching diffs to validate the pages of a group. B+ and Adaptive++ also combine as many prefetch requests as possible.

Dynamic Aggregation differs from Adaptive++ in two important ways: a) it only issues prefetches on access faults that require remote communication, while Adaptive++ also prefetches on faults that do not require additional remote communication, and after barrier events in repeated-phase mode; and b) it does not deal with cold start faults, while Adaptive++ does.

# 5 Experimental Results

## 5.1 Prefetching Effectiveness

**Coverage.** We start to evaluate the effectiveness of the prefetching techniques we study by assessing the *prefetching coverage* they

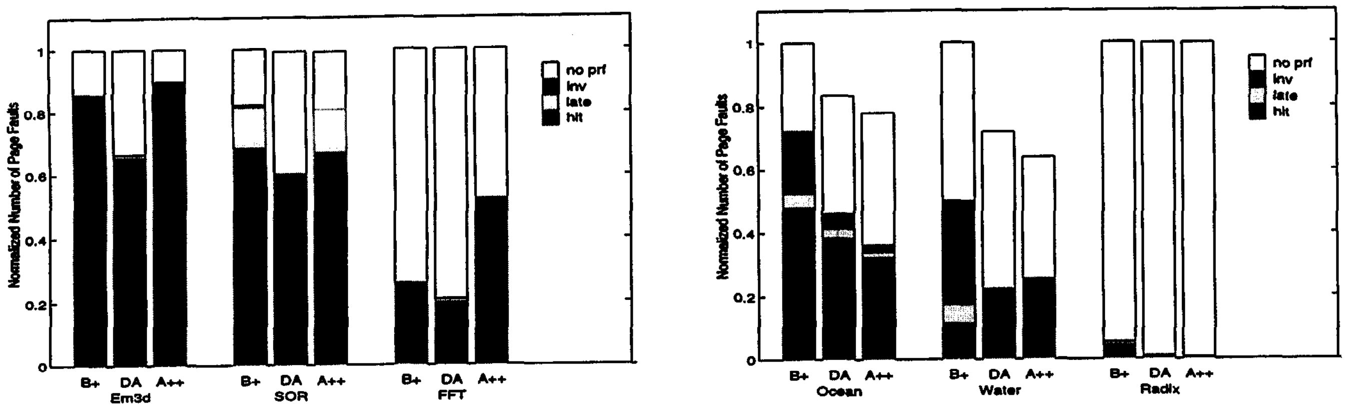

Figure 2: Prefetch Coverage.

achieve for our applications. The coverage is defined as the percentage of the access faults for which prefetches are actually issued. Figure 2 presents the number of access faults experienced by our applications under each of the prefetching techniques we study: B+, Dynamic Aggregation ("DA"), and Adaptive++ ("A++"). The numbers of faults are normalized according to the B+ results. Note that some applications experience different numbers of faults under the different prefetching techniques since the timing of events affects their data access and sharing behavior.

Each bar in the figure is broken down into access faults for which, by the time of the faults, the required data (pages and/or diffs) had already been collected ("hit"), the required data had been prefetched but did not return in time ("late"), prefetches had been issued but were invalidated since then ("inv"), and no prefetch had been issued ("no"). The access faults in the two latter classes require communication that is exposed to the application, i.e. overhead that is in the critical path of the execution. The coverage factor is defined as the sum of the three first categories.

Figure 2 shows that B+ achieves a high coverage factor for SOR, Em3d, and Ocean. These are exactly the applications for which invalidations better represent the set of future access faults. For FFT, Radix, and Water-nsquared less than 50% of the invalidations actually lead to access faults. The B+ results for Ocean and Water-nsquared are interesting in that a large percentage of access faults corresponds to pages for which data had been previously prefetched, but that nevertheless required communication. This effect suggests that issuing prefetches as soon as they seem required does not always reduce the number of faults that must involve communication.

Dynamic Aggregation is more conservative in terms of issuing prefetches, since (a) prefetches are not issued for pages that cannot be found in any group; and (b) at least one access fault requiring remote communication must be experienced for a group of pages to be prefetched. As a result, Dynamic Aggregation achieves a lower coverage factor than B+ for all applications in our suite. For SOR and Em3d, the coverage factor of Dynamic Aggregation is lower than that of B+ because of (b). For the other applications, both (a) and (b) lower the Dynamic Aggregation coverage factors.

Adaptive++ is also a conservative prefetching technique in that (c) it does not issue prefetches for pages faulted on inside of critical sections; and (d) it does not issue prefetches when the faulting page was not expected to experience a fault. Adaptive++ covers many fewer access faults than B+ for three of our applications, as a result of (c) and (d). For SOR and Em3d, the two techniques cover about the same number of faults, since the applications contain no locks and phases repeat consistently. Adaptive++ covers more faults than Dynamic Aggregation for four of our applications (Em3d, SOR, FFT, and Water-nsquared), again mostly as a consequence of (b). For the other applications, Dynamic Aggregation covers more faults than Adaptive++ mostly because of (c).

The coverage numbers for FFT and Radix deserve further discussion. For FFT, Adaptive++ is the only application to achieve a relatively high coverage factor (55%), as this application is dominated by cold start faults with a most common stride of 4 between access faults. The other techniques cannot deal effectively with cold start faults, achieving a little more than 20% coverage.

The Radix results are interesting in that no prefetching technique achieved more than a 10% coverage factor. The problem with Radix is that its access behavior is so irregular that past invalidations, access faults, and fault strides cannot be used to predict future faults effectively.

**Utilization.** Considering coverage information alone is not enough to evaluate a prefetching technique however. Techniques that prefetch aggressively naturally achieve high coverage factors, but sometimes at the cost of issuing a large number of *useless prefetches*, i.e. prefetches for data that will not be subsequently used. This is exactly the problem with the B+ technique.

Figure 3 presents the number and usefulness of the prefetches issued by the prefetching techniques for each of our applications. Again, the numbers of prefetches are normalized according to the B+ results. Each bar in the figure is broken down into useful and useless prefetches. Note that the useful prefetches correspond to the prefetches issued to cover the three categories of faults that comprise the coverage factor.

The results in the figure show that the B+ technique generates a significant percentage of useless prefetches for FFT and Radix. The technique is particularly ineffective for these applications since their invalidations are a terrible predictor of future page faults. In addition, note that for Ocean and Water-nsquared B+ issues many more prefetches than the other techniques. This is due to two factors. The first is that the coverage achieved by B+ is somewhat greater than that of the other techniques. The second factor is more subtle: several useful prefetches are issued for most "inv" access faults (figure 2) exhibited by B+.

Dynamic Aggregation and Adaptive++ are more conservative

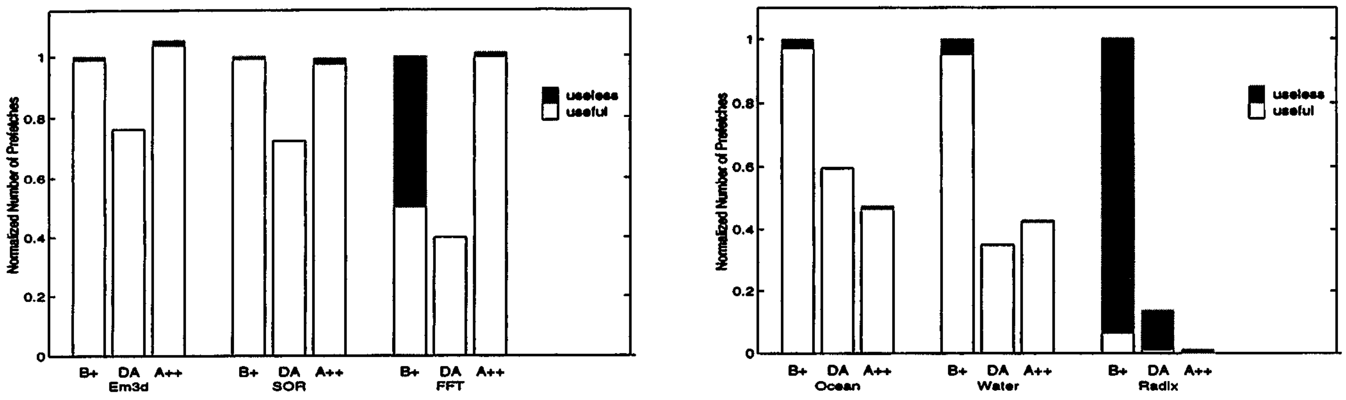

Figure 3: Prefetch Utilization.

| Appl | B+ | Dynamic Aggregation | Adaptive++ |
|---|---|---|---|
| SOR | 40% | 23% | 40% |
| Em3d | 56% | 40% | 58% |
| FFT | 17% | 12% | 49% |
| Ocean | 19% | 23% | 23% |
| Radix | -58% | -6% | 1% |
| Water-nsq | -2% | 9% | 22% |

Table 2: Data Access Overhead Reduction.

prefetching techniques and only issue prefetches when they have a high confidence that the prefetches will be useful. As a result, both techniques exhibit very low useless prefetch percentages in all cases, except Radix. Unfortunately, in attempting to avoid useless prefetches, these techniques may end up issuing a significantly smaller number of useful prefetches than B+. This problem is most serious for Dynamic Aggregation, which always issues many fewer prefetches than B+.

**Effectiveness.** In order to assess the effectiveness of a prefetching technique both its coverage factor and its percentage of useless prefetches must be considered. An effective technique is one that covers most access faults with a relatively small number of prefetches and a relatively small percentage of useless prefetches. Such a technique can significantly reduce the data access overhead of the application.

The data just presented demonstrates that B+ is not an effective technique, as it either issues too many prefetches (Ocean and Water-nsquared) or issues an excessive percentage of useless prefetches (FFT and Radix). Dynamic Aggregation is more effective than B+, but is not as effective as Adaptive++. The difference in effectiveness between Adaptive++ and Dynamic Aggregation is most glaring for FFT, but is also significant for Em3d and SOR.

**Data Access Overheads.** The effectiveness of each prefetching technique reflects itself directly on the data access overhead reductions the technique provides. Table 2 presents the percentage reductions in data access overhead provided by each prefetching technique to the applications in our suite. The table shows that only in two cases (SOR and Em3d) are the reductions provided by B+ greater than 20%. In fact, for Radix and Water-nsquared the data access overhead entailed by B+ is actually increased, as a result of its terrible effectiveness for these applications. Dynamic Aggregation usually does not reduce data access overheads as much as B+ for the applications where B+ works well. However, Dynamic Aggregation never severely worsens the data access overhead.

Adaptive++ is able to reduce data access overheads more significantly and consistently than B+ and Dynamic Aggregation. Differences with respect to B+ are most significant for FFT, Radix, and Water-nsquared. Differences with respect to Dynamic Aggregation are most significant for Em3d, SOR, and FFT. The reductions provided by Adaptive++ average 49% for the regular applications, while averaging 32% when both the regular and the mixed applications are considered. The same averages for B+ are only 38% and 12%, respectively. For Dynamic Aggregation these averages are only 25% and 18%, respectively.

## 5.2 Network Traffic

Another important aspect in the evaluation of Adaptive++ and in its comparison against B+ and Dynamic Aggregation is the communication traffic generated by TreadMarks ("Tmk"), B+, Dynamic Aggregation ("DA"), and Adaptive++ ("A++"). Table 3 presents the total amount of data and the total number of messages transferred on 16 processors in the presence of each prefetching technique.

As one would expect, the table shows that TreadMarks transfers the least amount of data among the different systems in most cases. Prefetching techniques increase the amount of data exchanged mostly as a result of useless prefetches and prefetches for pages that end up invalidated simply because they were fetched too early. Among the different prefetching techniques, B+ is the one that usually consumes the most communication bandwidth, as it involves the largest number of these two types of prefetches. The other prefetching techniques also increase the amount of data traffic somewhat, but increases are usually less significant.

Note however that the most important metric when assessing the communication behavior of software DSMs is the number of messages transferred, since the overhead of starting and receiving messages in such systems is usually much more significant than variations in the amount of data transferred per message. Two factors greatly influence the number of messages transferred by each prefetching technique: a) the technique's ability to combine messages to the same node; and b) the number of useless prefetches the technique generates. All three techniques can combine messages, especially B+ and Adaptive++, which at certain synchronization

| Appl | Data (in MBytes) | | | | Messages (in thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| | Tmk | B+ | DA | A++ | Tmk | B+ | DA | A++ |
| SOR | 31.5 | 31.8 | 31.5 | 32.2 | 17.1 | 11.4 | 14.4 | 11.6 |
| Em3d | 56.0 | 59.1 | 60.4 | 57.4 | 75.7 | 18.0 | 55.3 | 21.7 |
| FFT | 356.9 | 380.6 | 356.8 | 355.2 | 143.2 | 113.5 | 125.1 | 141.8 |
| Ocean | 994.3 | 1192.2 | 1062.9 | 974.3 | 655.0 | 511.3 | 560.2 | 515.6 |
| Radix | 237.5 | 422.8 | 259.2 | 238.6 | 134.7 | 123.6 | 146.2 | 135.1 |
| Water-nsq | 323.7 | 350.0 | 325.0 | 334.4 | 243.6 | 249.1 | 235.3 | 247.7 |

Table 3: Network Traffic.

| Appl | B+ | Dynamic Aggregation | Adaptive++ |
|---|---|---|---|
| SOR | 14% | 10% | 14% |
| Em3d | 27% | 14% | 26% |
| FFT | 8% | 8% | 34% |
| Ocean | -12% | 10% | 7% |
| Radix | -26% | -4% | 0% |
| Water-nsq | -11% | -1% | -2% |

Table 4: Speedup Improvements wrt TreadMarks.

points prefetch for a potentially large numbers of pages. Dynamic Aggregation has fewer opportunities to combine messages, since its prefetches are always issued only for the pages of a relatively small group. Nevertheless, Dynamic Aggregation (and Adaptive++) does not issue a large number of useless prefetches. Thus, Adaptive++ is the only technique that couples a serious potential for combining with a small number of useless prefetches.

Table 3 confirms these observations. The table shows that B+ greatly reduces the number of messages involved in standard TreadMarks as a result of its ability to combine messages. Even when the number of useless prefetches B+ issues is large, as for Radix, the technique can reduce the number of messages transferred by the system. Dynamic Aggregation is indeed the prefetching technique that usually involves the largest number of messages due to its limited combining potential. Adaptive++ sits in between the two other techniques in terms of the number of messages transferred.

## 5.3 Overall Performance

Table 4 presents the speedup improvements (with respect to standard TreadMarks on 16 processors) achieved by B+, Dynamic Aggregation, and Adaptive++. To complete the performance information on the techniques, figure 4 presents the running time breakdown of each of our applications under the different prefetching techniques.

The table shows that B+ improves the speedup of TreadMarks for three of our applications, SOR, Em3d, and FFT, by as much as 27%. For SOR and Em3d, the set of invalidations received during synchronization events is a very good representation of future accesses. For FFT, only about half of the invalidations received by each processor correctly predict future faults, which is enough for B+ to improve speedup for this application.

B+ is not without performance problems however. B+ degrades the running time of the other three applications (Ocean, Radix, and Water-nsquared) significantly. This effect is most visible for Radix, an application for which only a negligible number of the invalidations actually lead to future access faults; prefetching only contributes to creating traffic that ends up increasing the data access, synchronization, and IPC overheads as seen in figure 4. For Ocean and Water-nsquared, the poor B+ behavior is caused by waiting at relatively frequent synchronization points for a usually large number of previous prefetches to complete.

Dynamic Aggregation improves the speedup of most applications as a result of its ability to reduce data access overheads, as can be seen in figure 4. The running time improvements provided by the technique are not very significant however, at most 9%. Exceptions are Radix and Water-nsquared, applications for which Dynamic Aggregation does not affect performance in any relevant way, as suggested by the prefetching effectiveness data presented in the previous subsection. In contrast with B+, Dynamic Aggregation never degrades performance significantly.

Adaptive++ does not degrade performance either. Moreover, the speedup improvements provided by this technique are at least as substantial as any of the other techniques for the regular applications. Speedup improvements range from 14% to 34% for these applications. In fact, the speedup improvement obtained for FFT is by far superior to those provided by B+ and Dynamic Aggregation. For the non-regular applications, Adaptive++ behaves about the same as Dynamic Aggregation. The running time data for these applications show that even though the data access overheads of Adaptive++ are usually the lowest among the different prefetching techniques, it sometimes leads to higher synchronization and/or IPC overheads than Dynamic Aggregation. The main reason for this effect is that Adaptive++ clusters prefetch requests in time by prefetching on all access faults. Even though this strategy reduces the data access overhead significantly, it may stress the communication and memory systems as well as create greater interference on remote processors. Nevertheless, our results show that Adaptive++ is indeed beneficial.

# 6 Related Work

Most of the previously published work on reducing the overhead of data accesses in software DSMs has been concentrated on update-based coherence strategies, e.g. [6, 11, 9]. Prefetching and multithreading techniques for software DSMs, on the other hand, have received little attention so far [5, 3, 7, 1, 10]. Due to space limitations, we only focus on runtime-based prefetching techniques here. A more complete discussion of the related work can be found in [4]. Dynamic aggregation [1] has been studied in this paper.

Karlsson and Stenström [7] proposed a prefetching technique (called *KS* here) that is similar in flavor to Adaptive++ in that it is

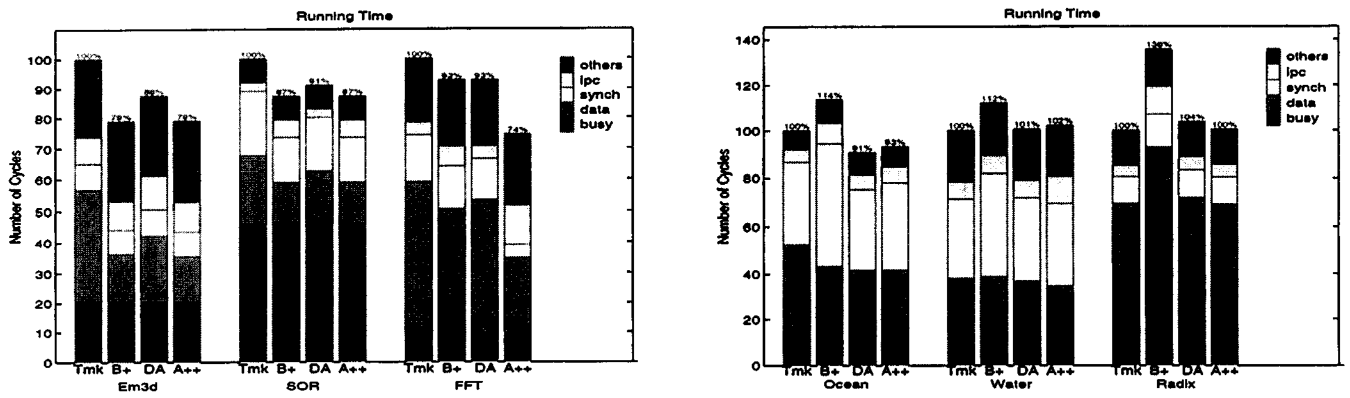

Figure 4: Running Times on 16 Processors.

dynamic, adaptive, and was implemented for TreadMarks. However, Adaptive++ is more general than KS, as the latter technique restricts prefetching to producer-consumer data and resorts to straight (stride-one) sequential prefetching when this sharing pattern is not detected. In addition, the specific details of the two techniques differ quite a bit, since KS was designed for a network of SMP workstations rather than a network of uniprocessor workstations like Adaptive++.

Our previous work [3] proposed the use of simple hardware support for aggressively tolerating overheads in software DSMs. We evaluated the B+ prefetching technique both under standard TreadMarks and under a modified version of the system that takes advantage of the extra hardware. Our experiments detected the performance problems of B+, but showed that it can profit substantially from our hardware support. We believe that Adaptive++ should benefit from this support even more significantly than B+.

## 7 Conclusions

In this paper we proposed and evaluated a dynamic and adaptive prefetching technique for the TreadMarks DSM that is intented to optimize regular applications. Simulation results of this system running on a network of workstations showed that our prefetching technique can deliver speedup performance improvements over standard TreadMarks of up to 34% for regular applications running on 16 processors. A comparison against other well-known runtime prefetching techniques showed that our strategy achieves the largest reductions in data access overhead. As a result, our technique is consistently competitive in terms of performance, while being able to optimize a larger set of applications than the other strategies. The main conclusion of this paper is that our technique should definitely be considered by software DSM designers as an effective way of tolerating the overhead of remote data accesses.

## Acknowledgements

We would like to thank Leonidas Kontothanassis for contributing to our simulation infrastructure. We would also like to thank Cristiana Amza and Karthick Rajamani for answering several questions about the Dynamic Aggregation technique. Finally, we are grateful to Cristiana Seidel for comments that helped improve this paper.

## References

[1] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th PPoPP*, June 1997.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2), Feb 1996.

[3] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th ASPLOS*, Oct 1996.

[4] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. Technical Report ES-463/98, COPPE Systems Engineering, Federal University of Rio de Janeiro, March 1998.

[5] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th ASPLOS*, Oct 1996.

[6] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th ISCA*, May 1993.

[7] M. Karlsson and Per Stenström. Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems. *JPDC*, 43(7), July 1997.

[8] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, Jan 1994.

[9] L. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th HPCA*, Feb 1998.

[10] T. Mowry, C. Chan, and A. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th HPCA*, Feb 1998.

[11] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 ICPP*, Aug 1997.

[12] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd MASCOTS*, Jan 1994.

[13] S. Wiel and D. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. *Computer*, 30(7), July 1997.

[14] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd ISCA*, May 1995.