

Comparing Latency-Tolerance Techniques for Software DSM Systems

Raquel Pinto, Ricardo Bianchini, *Member, IEEE Computer Society*, and
Claudio L. Amorim, *Member, IEEE*

Abstract—This paper studies the isolated and combined effects of several latency-tolerance techniques for software-based distributed shared-memory systems (software DSMs). More specifically, we focus on data prefetching, update-based coherence, and single-writer optimizations for page-based software DSMs. Our experimental results with six parallel applications show that, when these techniques are carefully combined, they can provide running time and speedup improvements of up to 54 percent and 110 percent, respectively, on a cluster of eight PCs.

Index Terms—Distributed systems, performance.

1 INTRODUCTION

OUT of the two main parallel programming models in use today, namely, shared memory and message passing, the former is widely recognized as providing a simpler abstraction (i.e., a single address space) than the latter. The shared-memory model can be implemented directly in hardware, in software, or with some combination of the two. Hardware-intensive shared-memory systems are typically efficient, but are often expensive. Software-intensive systems are less expensive, but are not always as efficient as their hardware counterparts, especially in large systems [9].

Our goal is then to explore optimizations that can bring the performance of software-based shared-memory systems closer to that of hardware multiprocessors. In particular, we focus on software distributed shared-memory systems (software DSMs). These systems implement the shared-memory abstraction completely in software on top of message-passing hardware. In terms of performance, software DSMs often exhibit high remote data access latencies when running real parallel applications. Relaxed consistency and multiple-writer coherence are common approaches to mitigating these latencies. Unfortunately, these techniques have not been enough to significantly broaden the class of applications for which software DSMs are efficient. As a result, several other techniques have been proposed to address these high latencies. The best-known of these techniques are prefetching and coherence protocol adaptation to sharing patterns.

Prefetching strategies can 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 used in hardware DSMs [22] are often not effective for software DSMs [6]. For example, sequential prefetching, the most common prefetching strategy in hardware DSMs, works well for these systems because of the relatively small size of the prefetching units (cache blocks). In contrast, sequential prefetching is not as useful for systems that work at page-level granularity.

Remote data access latencies can also be addressed by adapting between invalidate and update-based coherence. Update-based protocols usually transfer more data than necessary, but can improve performance with respect to invalidate-based protocols when the sharing patterns are such that most updates are indeed useful.

Another way in which the protocol can be adapted involves selecting between single and multiple-writer coherence according to sharing behavior. Multiple-writer protocols alleviate the effect of false sharing in systems with large coherence units, such as virtual memory pages. However, the overhead of detecting, recording, and merging changes to shared data is always incurred, independently of whether false sharing is indeed a problem. This overhead can be eliminated for pages that are not subject to false sharing by allowing only a single writer at a time for them.

In this paper, we study the isolated and combined effects of these different latency-tolerance techniques for page-based software DSMs. In particular, we focus on the Adaptive++ prefetching technique [5] and the adaptation strategies of the ADSM system [13], called SMA (for Single/Multiple-writer Adaptation) and IUA (for Invalidate/Update Adaptation). The combination of all techniques takes the sharing behavior of each page into consideration,

- R. Pinto is with the Universidade Estácio de Sá, Av. Presidente Vargas 2560-Centro, Rio de Janeiro, RJ, Brazil 20213-900.
E-mail: raquel.gomes@estacio.br.
- R. Bianchini is with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019.
E-mail: ricardob@cs.rutgers.edu.
- C.L. Amorim is with the Programa de Engenharia de Sistemas e Computação, COPPE, Centro de Tecnologia-Sala H318, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil 21941-972.
E-mail: amorim@cos.ufrj.br.

Manuscript received 11 Dec. 2002; revised 25 June 2003; accepted 28 June 2003.

For information on obtaining reprints of this article, please send e-mail to: tpd@computer.org, and reference IEEECS Log Number 117935.

making sure that all important behaviors are covered and no conflicts exist between techniques. All techniques and combinations were implemented for the TreadMarks system [3].

To evaluate these techniques, we performed experiments with six parallel applications on a cluster of eight Pentium-based PCs. We isolated the performance benefits of each technique and their combination by comparing them to TreadMarks. Our results show that most applications benefit from only one of the techniques, although some applications do benefit from multiple techniques. The speedup improvements of the different techniques in isolation can reach 75 percent, whereas these improvements can reach 110 percent when all techniques are applied in combination.

The vast majority of the previous work in latency tolerance for software DSMs has considered data prefetching, update-based coherence, and single-write optimizations independently (e.g., [5], [13], [16], [11]). By directly comparing these techniques under the same hardware and software infrastructure, we can more completely and accurately contrast them.

Two other studies have compared prefetching and updates directly [2], [8]. Castro and de Amorim [8] showed that update-based coherence frequently outperforms prefetching. However, they studied a very simple prefetching technique (originally proposed in [4]) that relies on page invalidations to guide prefetching. We have shown that this technique is not as efficient as Adaptive++ in [5]. Both the simple technique and Adaptive++ were studied in simulation. Castro and Amorim attempted to combine prefetching and updates for one application and achieved good results. Unfortunately, their work was also based solely on simulations. Our paper differs from their study in that we focus on real executions of several parallel applications and a sophisticated prefetching strategy. In fact, this paper is the first to study a real implementation of Adaptive++ and shows that prefetching can frequently outperform update-based coherence.

Amza et al. [2] studied another sophisticated prefetching technique called Dynamic Aggregation. In [5], we also showed Adaptive++ to be more efficient than Dynamic Aggregation. Amza et al. also attempted to combine prefetching and updates, but did so without considering sharing patterns. Consequently, the two techniques wound up tackling the same access faults, leading to poor performance. A similar problem was described in [14], when multithreading was combined with compiler-based prefetching. Our paper differs from these works by studying an efficient, runtime-based prefetching technique and an efficient combination of techniques that uses a categorization of the sharing patterns experienced by pages to avoid conflicts.

Finally, a few other studies proposed the use of other runtime-based latency-tolerance techniques for software DSMs. We proposed hardware support for latency tolerance in [4]. Mowry et al. [14] and Thitikamol and Keleher [19] studied multithreading. A few groups [10], [15], [12], [17] studied reducing coherence overheads with home-based software DSMs and remote memory writes. Several groups

have proposed home migration strategies for home-based software DSMs (e.g., [20], [21], [18]). Bilas et al. [7] proposed network interface and protocol extensions to completely eliminate the need for interrupts and polling in home-based software DSMs. Our work complements these previous studies.

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 access overheads significantly degrade the performance of applications running on top of it. Sections 3, 4, and 5 describe the Adaptive++ prefetching technique, the SMA and IUA techniques, and their combination, respectively. Section 6 describes our experimental environment, whereas Section 7 presents our experimental results. Finally, Section 8 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 and allows multiple concurrent writers per page. In TreadMarks, page invalidation happens at lock acquire points, whereas the modifications to an invalidated page are collected from previous writers at the time of the first access (fault) to the page. On a lock acquire operation, the releaser can determine the set of write notices (descriptions of the modifications made to shared data) that the acquiring processor needs to receive. Upon receiving the notices, the acquirer can then change the state of its memory accordingly. No action is taken on lock release operations. A barrier episode can be seen as a release followed by an acquire performed by each processor.

The propagation of the actual modifications made to a shared page is deferred until the acquirer suffers an access fault on the page. These modifications are determined using the twinning and diffing mechanism as follows: A page is initially write-protected so that on the first write to it a violation occurs. On such a violation, an exact copy of the page (a twin) is made and the original copy of the page is made writable. When the actual modifications are required, the twin and the current version of the page are compared to create a runlength encoding of the modifications (a diff). When an access 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 from remote processors 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.

The main overheads in TreadMarks (and most other software DSMs) are related to communication latencies and coherence actions. Fig. 1 presents a detailed view of the execution time performance of our applications running on top of TreadMarks on eight processors. The bars in the figure show normalized execution time broken down into busy time, data fetch latency, synchronization time, and

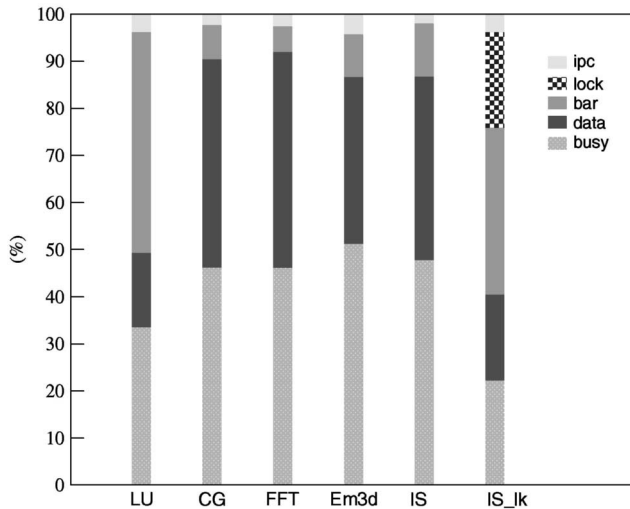


Fig. 1. Running time on eight nodes.

IPC overhead. The busy time category represents the amount of useful work performed by the computation processor. It also includes TLB miss latency, write buffer stall time, interrupt time, and the most significant of these overheads, cache miss latency. 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.

Fig. 1 shows that most applications running on top of TreadMarks indeed 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. These diff-related overheads can be reduced or even eliminated by using an update strategy combined with single/multiple-writer adaptation.

3 PREFETCHING: ADAPTIVE++

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 consecutive 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.

3.1 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 numbers 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. (A page number is appended to the appropriate list as the corresponding fault happens.) 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 percent of their page numbers belong to both lists.

Note that this strategy only checks the similarity of each pair of consecutive phases. When the two lists are dissimilar, the strategy assumes (without verification) that similar phases alternate. In this case, a third list of page numbers could be used to make sure that alternate phases are indeed similar, at the cost of extra memory and processing overhead. We do not perform this extra verification because it is rare for applications to exhibit similar phases that are not consecutive or alternating [6]. Furthermore, predicting an irregular application as having alternate phases that are similar causes no harm. As will soon be clear, Adaptive++ does not degrade performance if applications end up behaving differently than it expects.

3.2 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 in 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.

3.3 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.

3.4 Issuing Prefetches

In the repeated-phase mode, a user-defined number of pages (24 pages in our experiments) from the front of the expected list is prefetched for right after a barrier crossing point. In addition, on a fault outside of any critical section, the repeated-phase mode tries to issue prefetches for another user-defined number of pages (four pages in our experiments). These pages follow the faulting page on the expected list. Prefetches are *not* issued for: 1) pages that are already valid in the local memory and 2) pages for which the same prefetches have been issued. Note that an issued prefetch may never complete since there is no guarantee that the prefetch request or the corresponding reply will be received. No action is taken on faults occurring inside of critical sections or on pages that are not on 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 a fault outside any critical section, the repeated-stride mode tries to issue prefetches for yet another user-defined number of pages (four pages in our experiments) following the faulting page on expected. Again, prefetches are only issued on faults to pages that appear on expected, provided that the corresponding data are not already valid in memory and the same prefetches have not been issued.

3.5 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 to the expected list of faults. Adaptive++ waits for all prefetch replies to have been received before crossing a subsequent synchronization point.

3.6 Justification for Main Design Decisions

Adaptive++ targets regular applications because their performance can be improved without resorting to sophisticated users or compilers that can insert explicit prefetch calls in the source code. The fault behavior of these applications can be predicted based on their past history of faults.

Adaptive++ focuses solely on repeated-phase and repeated-stride types of regular applications. The reason is that our previous experience [6] has shown that these are the two most important forms of regularity found in a large set of parallel applications. The few previously proposed dynamic prefetching techniques have not been shown effective for both types of regular applications.

Adaptive++ always tests whether a faulting page is among the ones it expected to experience a fault, before issuing prefetches. This test helps Adaptive++ have a greater confidence that the application is behaving regularly and that it will be able to improve performance. In irregular applications, the faults experienced by each processor are very frequently "unexpected."

Adaptive++ avoids issuing prefetches within critical sections. The reason for this is that our previous experience [4] has shown that several parallel applications exhibit short critical sections that may be substantially lengthened by issuing prefetches. Longer critical sections increase synchronization overheads sometimes significantly in the presence of lock contention. Adaptive++ also avoids prefetching for pages faulted on inside of critical sections. The reason is that these pages usually receive write notices at the lock acquire point, forcing the system to collect diffs on the first accesses to the pages anyway.

Finally, our experiments with Adaptive++ issue prefetches for a relatively large number of pages (24) right after barrier events, while being much less aggressive when issuing (four) prefetches on access faults. This strategy uses the fact that processors frequently experience faults right in the beginning of each phase of execution, permitting an overlap of interprocessor request and data fetch overheads. Later in the execution of the phase, this type of overlap becomes less probable and prefetches end up interfering with computation on the affected remote processors more frequently. The particular values we selected for these user-defined parameters are the ones that led to the highest performance for our applications.

4 ADAPTIVE COHERENCE: SMA AND IUA

We study two techniques for adapting the coherence protocol to the sharing patterns of each page. Specifically, we adapt between invalidate and update-based coherence (IUA) and between single and multiple-writer coherence (SMA). Both techniques rely on a dynamic categorization (called Sharing Pattern Categorization or SPC) of the type of sharing experienced by each page [13].

SPC creates an association between each lock variable and the pages that experience faults inside of the critical

sections the variable delimits. This association and the write notice information allow SPC to classify pages as multiple-writer (i.e., falsely-shared) and single-writer (i.e., migratory under locks, migratory under barriers, or producer/consumer). A single-writer page can only be modified by its current owner, after it retrieves the latest version of the page. Multiple-writer pages have no owner. A producer/consumer page has a fixed owner (the producer), whereas the ownership of migratory pages is transferred as appropriate. Transfers of ownership do not involve extra protocol messages. On a lock acquire operation, the processor will automatically get the ownership of the pages associated with the lock. When releasing the lock, the processor also releases the ownership of its pages. The ownership of a migratory page that is not associated with a lock is transferred along with a copy of the page, if the processor servicing the page is its owner.

When we exploit IUA in isolation, pages that are classified as migratory under locks or producer/consumer are updated. More specifically, the transfer of updates of lock-protected data occurs on lock acquire operations in the following way: The releaser sends a lock grant message to the acquirer, including the write notices belonging to intervals not seen by the acquirer and the numbers of the pages to be updated. The actual updates follow the lock grant message in separate messages. Updates of producer/consumer data are sent to all the consumers of the data at a barrier. The transfer of these updates is overlapped with the synchronization overhead of barriers. In both situations, processors do not have to wait for the updates to be received at synchronization points. Migratory pages under barriers and multiple-writer pages are made coherent with invalidations. Even though entire pages are sent as updates, all pages are treated under the standard twinning and diffing mechanism.

When we exploit SMA in isolation, multiple-writer pages are treated under the twinning and diffing mechanism, whereas the coherence of single-writer pages is maintained by transferring whole pages. No updates are sent.

When SMA and IUA are combined (as originally proposed for the ADSM system [13]), multiple-writer pages are treated under the twinning and diffing mechanism, single-writer pages are transferred as a whole, and pages that are classified as migratory under locks or producer/consumer are updated.

4.1 Justification for Main Design Decisions

IUA and SMA target applications with single-writer pages. For these applications, avoiding the overhead of twinning and diffing can improve performance significantly, as originally demonstrated in [1].

IUA and SMA depend on SPC to categorize pages. The set of SPC categories was chosen to be comprehensive, but be simple enough that categorizing pages would not involve performance degradation.

As a result of SPC, IUA and SMA can be more aggressive in performing their optimizations than Adaptive++. In fact, IUA does attempt to eliminate certain types of page faults inside of critical sections. The reason is two-fold: 1) SPC is accurate in its categorization and 2) the latest version of a

migratory page under a lock is held by the lock releaser, which needs to communicate with the acquirer anyway.

5 COMBINING TECHNIQUES

We are interested in finding effective ways to combine prefetching and adaptation techniques such as Adaptive++, SMA, and IUA. As we mentioned in the introduction, some previous studies [2], [14] of combined techniques failed to produce more efficient systems. The main reason is that the combined techniques tried to tackle the same page faults.

This problem can be avoided in systems that detect sharing patterns by realizing that each strategy is most effective for different types of sharing. The Adaptive++ technique is not efficient for page faults that occur inside critical sections. IUA and SMA work well for those faults. IUA is efficient for single-writer pages that are lock-protected in migratory state, or are barrier-protected in producer/consumer state. IUA coherence does not work so well for multiple-writer pages. Adaptive++ can tackle some of these faults very effectively.

Thus, to combine these strategies, we rely on the SPC categorization. As we mentioned before, the ADSM system relies on SPC to combine SMA and IUA. When we combine all techniques, the access to pages classified as falsely shared (or multiple-writer) is optimized by Adaptive++, SMA optimizes the single-writer pages, whereas IUA is applied to lock-protected migratory pages and barrier-protected producer/consumer pages.

6 METHODOLOGY AND WORKLOAD

Our experimental environment consists of a cluster with eight PCs. Each PC has 512 Mbytes of memory and a 650 MHz Pentium III with split data and instruction level1 caches (16 Kbytes each) and one level2 cache (256 Kbytes). As the implementation of Adaptive++, SMA, and IUA is based on the TreadMarks system, we evaluate these strategies and their combination by comparing our results against standard TreadMarks. In all systems, interprocessor communication is implemented with UDP over a 1 Gbit/sec Myrinet switch.

We report results for six representative parallel programs: LU, CG, FFT, Em3d, and two versions of IS, lock-based (IS_lk) and barrier-based (IS). LU is from the Splash-2 suite [23], Em3d is from UC Berkeley, and the other applications are from the TreadMarks distribution. The applications from this distribution are optimized for software DSMs (more precisely for TreadMarks), whereas the others are not. Table 1 presents the inputs we used.

LU factors a matrix into two matrices: one lower triangular and the other upper triangular. In the CG benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. FFT solves a partial differential equation using forward and inverse FFTs. Em3d simulates electromagnetic wave propagation through 3D objects. IS (Integer Sort) ranks an array of N integers using keys in the range [0, Bmax].

TABLE 1
Application Inputs

| Appls | Inputs |
|-------|--|
| LU | 2800 × 2800, block of 128 bytes |
| CG | 14000 × 14000 elements with 2030000 non-zeros |
| FFT | 128 × 128 × 128, 10 iterations |
| Em3d | 80000 nodes, 0.1% remote, 200 iterations |
| IS | $N = 2^{23}$, $B_{max} = 2^{15}$, 100 iterations |
| IS_lk | $N = 2^{23}$, $B_{max} = 2^{15}$, 100 iterations |

7 EXPERIMENTAL RESULTS

7.1 Effectiveness

7.1.1 Coverage

We start to evaluate the effectiveness of the techniques we study by assessing the *coverage* they achieve for our applications. The coverage is defined as the percentage of the access faults for which the strategies took some action.

Fig. 2 presents the number of access faults experienced by our applications under Adaptive++ (adp), SMA+IUA (adsm), and the combination of these techniques (cmb). The numbers of faults are normalized to the Adaptive++ results. 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 by prefetches (“hit”), the required data had been prefetched, but did not return in time (“late”), the prefetches had been issued, but were invalidated before use (“inv”), the required data had already been received as updates (“upd”), the required data will be collected with the corresponding page since the page has been classified as single-writer (“sma”), and no operation has been or will be issued (“no”). The coverage factor of Adaptive++ is defined as the sum of the three first categories. In the ADSM system, the coverage factor is defined by the sum of upd and sma.

We start the discussion by focusing on Adaptive++. Fig. 2 shows that Adaptive++ achieves a high coverage factor for CG, FFT, Em3d, and IS, as they are regular applications. These applications take enormous advantage of the repeated-phase mode. CG and IS behave almost the same: their similar phases are consecutive and they take advantage of the repeated-stride mode at the initial phase of execution. In the repeated-stride mode, the prefetches are issued on access faults, so the data collected by these prefetches are not always received early enough to avoid stalling the processor, which explains the late prefetches. FFT also has repeated phases, but these phases alternate. Em3d exhibits a different behavior. It has two main barriers. The similar phases associated with the first barrier alternate, whereas the similar phases associated with the second barrier are consecutive. In Em3d, the access fault prefetches also cause late prefetches. The access faults for which no prefetches are issued occur in the initial phase of execution, when the Adaptive++ is yet to identify the application behavior.

LU does not exhibit as high a coverage as these highly regular applications. Most access faults in LU are to consecutive pages, so Adaptive++ uses the repeated-stride mode ($stride = 1$). However, due to the initial phase of execution (when the application behavior has not yet been determined) and the fact that not all strides are 1, Adaptive++ only covers 41 percent of the faults. The Adaptive++ coverage for IS_lk is not very high either. Even though IS_lk is a lock-based application, it does access a shared data structure outside critical sections. This data structure is read with stride -1.

Now, we concentrate on the coverage achieved by the ADSM system (SMA+IUA). ADSM achieves a high coverage for CG, FFT, Em3d, IS, and IS_lk. In the case of CG, the coverage is dominated by the updates, as most of the access faults refer to pages classified as producer/consumer. FFT has a low update percentage (17 percent) due to the same kind of access faults, but its high coverage is due to sma, which demonstrates that almost all pages are modified by one processor at a time. The access faults that are not covered are cold start faults. Em3d and IS_lk have their coverage equally divided between the upd and sma categories. The updates of Em3d are due to pages classified

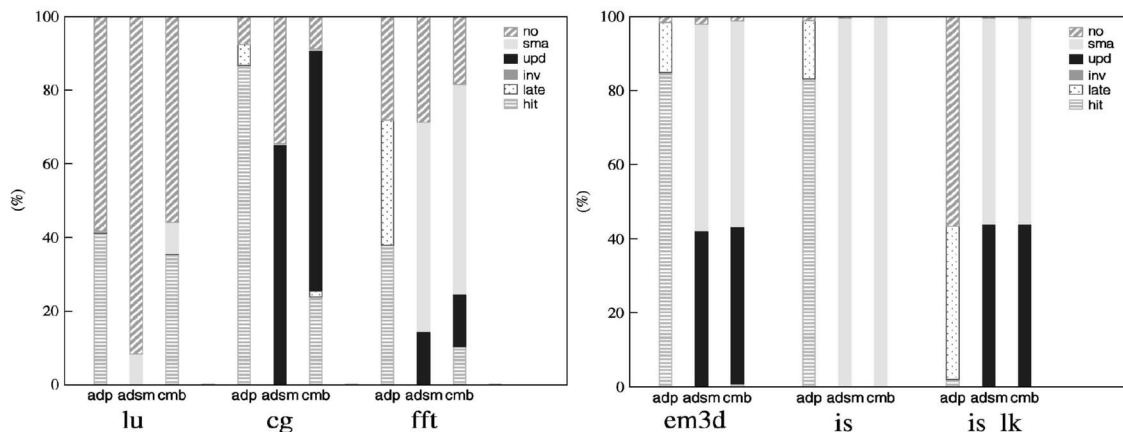


Fig. 2. Access fault coverage.

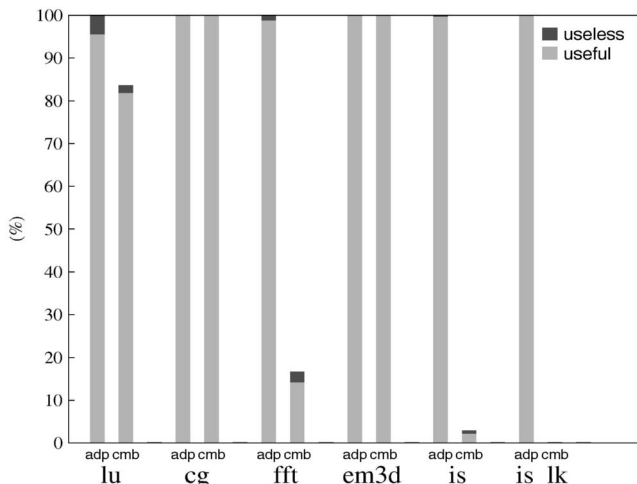


Fig. 3. Prefetch utilization.

as producer/consumer, whereas the updates of IS_lk are issued to pages categorized as migratory by locks.

The low coverage of LU is due its large number of cold start faults. Adaptive++ achieves a higher coverage for LU because it covers cold start faults using its repeated-stride mode.

Analyzing the coverage achieved by combining all techniques, we observe that LU, CG, FFT, and Em3d are the applications that take the most advantage of this combination. In LU, FFT, and Em3d, SMA and/or IUA cover the single-writer pages, whereas Adaptive++ issues prefetches for the cold start faults. In CG, which has multiple as well as single-writer (producer/consumer) pages, IUA deals with the producer/consumer pages by updating them, whereas Adaptive++ prefetches for the multiple-writer pages. For the other applications, no prefetches are issued (all shared pages are single-writer pages), so the coverage is the same as that of the ADSM system.

7.1.2 Utilization

Considering coverage information alone is not enough to evaluate the different techniques, however. Techniques that prefetch aggressively, for instance, naturally achieve high coverages, but sometimes at the cost of issuing a large number of *useless* prefetches, i.e., prefetches for data that will not be subsequently used.

Fig. 3 presents the number and usefulness of the prefetches issued by our systems. The figure shows two bars for each application. The bar on the left represents the Adaptive++ technique when applied in isolation, whereas the bar on the right represents the Adaptive++ technique as applied in combination with the other techniques. The numbers of prefetches are normalized according to the Adaptive++ results. Each bar 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.

Adaptive++ only issues prefetches when it has a high confidence that the prefetches will be useful. As a result, it exhibits very low useless prefetch percentages in all cases. Again, we observe that CG is the only application that

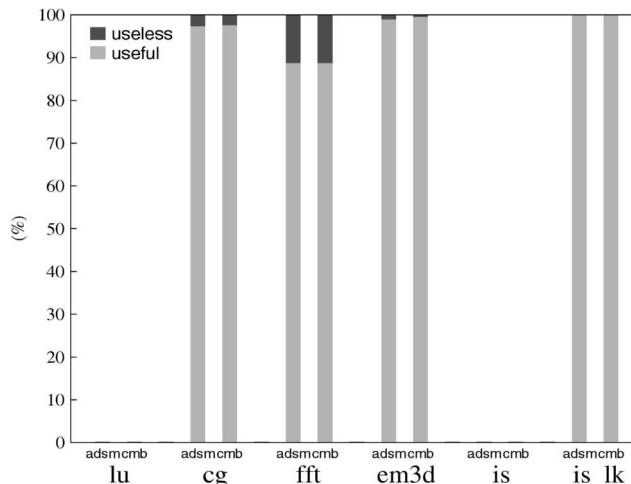


Fig. 4. Update utilization.

issues a considerable number of prefetches under the combination of strategies.

Fig. 4 presents the number and usefulness of the updates sent by our systems. Again, we present two bars for each application. The left bar represents the utilization of updates in the ADSM system, whereas the right bar shows the utilization of updates when all techniques are combined. This figure clearly shows that the strategy of restricting the updates only to the pages classified as producer/consumer or migratory by locks generates low percentages of useless updates. Note that no updates are sent by the LU and IS applications.

7.1.3 Data Access Overheads

An effective latency-tolerance technique is one that covers most access faults with a relatively small number of useless operations. Such a technique can significantly reduce the data access overhead of the application. Table 2 presents the percentage reductions in data access overhead provided by each strategy for the applications in our suite. The table shows that the Adaptive++ technique provides considerable reductions for the barrier-based regular applications (LU, CG, FFT, Em3d, and IS), ranging from 19 to 62 percent. Adaptive++ has no effect for IS_lk. Overall, Adaptive++ reduces the data access overheads by an average of

TABLE 2
Data Access Overhead Reduction

| Appls | Adp | IUA | SMA | ADSM | Comb |
|-------|-----|------|------|------|------|
| LU | 30% | 5% | 43% | 43% | 54% |
| CG | 62% | 62% | 8% | 62% | 76% |
| FFT | 19% | 14% | 29% | 39% | 38% |
| Em3d | 51% | 25% | -72% | -7% | -6% |
| IS | 57% | -2% | 62% | 62% | 61% |
| IS_lk | 0% | -16% | 62% | 73% | 73% |

TABLE 3
Speedups

| Appls | Tmk | Adp | IUA | SMA | ADSM | Comb |
|-------|-----|-----|-----|-----|------|------|
| LU | 3.3 | 3.4 | 3.3 | 4.7 | 4.7 | 4.7 |
| CG | 3.6 | 4.7 | 4.7 | 3.7 | 4.8 | 5.1 |
| FFT | 3.1 | 3.4 | 3.3 | 4.0 | 4.2 | 4.0 |
| Em3d | 3.2 | 4.3 | 3.4 | 3.0 | 3.5 | 3.5 |
| IS | 4.2 | 4.6 | 4.1 | 5.7 | 5.7 | 5.7 |
| IS_lk | 2.0 | 2.0 | 2.8 | 3.5 | 4.2 | 4.2 |

37 percent. Also, we can observe that this technique is the only one that never worsens the data access overhead.

IUA performs significantly worse than Adaptive++. For the two versions of IS, IUA even increases the data access overhead by 15 percent when all applications are considered. SMA achieves mixed results. It performs well for all applications, except for Em3d and CG. In Em3d, SMA's use of page transfers significantly increases the amount of communication traffic since only a small fraction of the pages are actually modified. SMA does not benefit CG because the data access overhead in this application is dominated by network latency. On average, SMA reduces data access overheads by 22 percent.

When we combine IUA and SMA, data access overheads are reduced by at least as much as each technique in isolation, which nicely shows the synergy that can be achieved by combining techniques. The only case in which this effect is not observed is Em3d. Again, this is caused by SMA's transferring of whole pages as opposed to much shorter diffs. On average, ADSM reduces the access overhead by 45 percent.

When we combine all techniques, we again see that the overhead reductions are at least as high as those of the isolated techniques, except in the case of Em3d. LU and CG are particularly interesting applications. LU takes advantage of the combination of Adaptive++ (cold start prefetches) and SMA, whereas CG takes advantage of the combination of Adaptive++ and IUA. For Em3d, the key to good performance is to prefetch diffs of single-writer pages, which our combined system does not do. On average, the combination of all techniques reduces the access overhead by 49 percent.

7.2 Overall Performance

Table 3 lists the speedups achieved by our applications under TreadMarks (Tmk), TreadMarks using Adaptive++ (Adp), TreadMarks using IUA (IUA), TreadMarks using SMA (SMA), TreadMarks using IUA and SMA (ADSM), and the system combining all techniques (Comb). These results show that speedups are relatively low under TreadMarks, but increase substantially when we apply latency-tolerance techniques. The table also shows that Adaptive++, IUA, and SMA exhibit inconsistent benefits. Adaptive++ improves

the speedup of the barrier-based applications by 3 to 34 percent, but does not affect IS_lk. When IUA is beneficial, it improves speedups between 6 and 40 percent. However, it does not improve the speedup of LU and IS. SMA improves speedups more significantly, between 3 and 75 percent, but hurts Em3d. ADSM provides substantial speedup improvements, ranging from 33 to 110 percent, for LU, CG, FFT, IS, and IS_lk. For Em3d, the speedup improvement achieved by ADSM is smaller (9 percent). CG is the only application for which the speedup achieved by combining all techniques is higher than that of ADSM. For FFT, ADSM scales better than the combination of all techniques. For the other applications, this combination scales as well as ADSM.

To explain our results in more detail, Fig. 5 presents the running time of each of our applications under the different systems. The bars show normalized execution time broken down into busy time, data access overhead, synchronization time (barrier and lock time), and IPC overhead (i.e., the time a processor spends servicing requests coming from remote processors).

Let us start by discussing the performance of Adaptive++. The graphs show that the regular, barrier-based applications achieve performance improvements up to 26 percent (Em3d). Note, however, that the significant data access overhead reductions provided by Adaptive++ are not always directly translated into better overall performance, due to increases in IPC overhead. These increases are caused by the overhead of receiving prefetch replies. To see this, note that on a remote data access caused by an actual page fault, the page/diffs is/are received without interrupting the processor which is stalled waiting for them. In case of a prefetch, the processor is hopefully busy with useful computation when the reply arrives, so it has to be interrupted to handle the reply. Furthermore, under Adaptive++, any diffs prefetched have to be copied to a pool of diffs, whereas diff replies to an actual page fault are received in the pool directly. Finally, any useless prefetches represent unnecessary IPC overhead.

Considering the busy time of the applications, we note that Adaptive++ decreases this time in LU, FFT, and Em3d, which at first might sound counter-intuitive. Note, however, that FFT and Em3d prefetch pages in repeated-phase mode on barrier events. Thus, the diffs created during a phase are now requested and created at the beginning of the phase, concentrating the effect of cache pollution in this initial period and reducing the negative impact of pollution on the computation time. As for LU, it uses the repeated-stride mode to issue two prefetches on each access fault, concentrating the diff creation requests and again reducing the cache pollution effect. SMA, ADSM, and Combined also decrease the busy time of LU, FFT, and Em3d considerably. These reductions are also related to reduced cache pollution because these systems eliminate the need for diffs and twins for single-writer pages.

IUA does not reduce data access overheads as significantly as Adaptive++, so its performance results are not as good in most cases. The most interesting result obtained by IUA is that it was able to reduce the synchronization overheads of IS_lk significantly to achieve an overall performance improvement with respect to TreadMarks of 29 percent. The reason for this effect is that IUA does reduce the overhead of data accesses made inside critical sections.

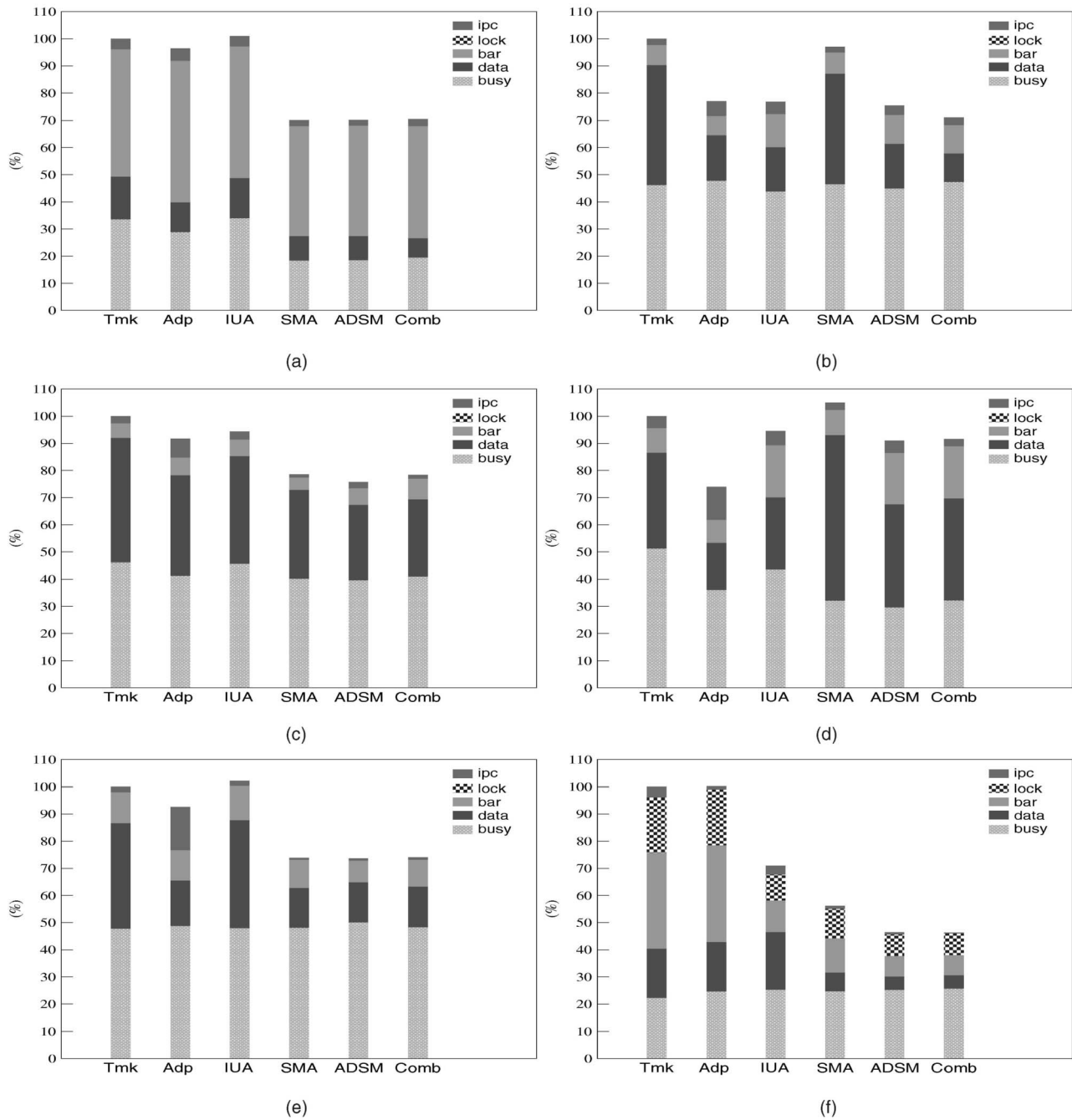


Fig. 5. Running times on eight nodes for (a) LU, (b) CG, (c) FFT, (d) Em3d, (e) IS, and (f) IS_lk.

The shorter critical sections decrease lock contention and, as a result, reduce the serialization that causes high barrier wait times. SMA, ADSM, and Combined also exhibit these gains. Adaptive++ does not prefetch for accesses that occur inside critical sections.

CG and Em3d are the only applications that do not benefit from SMA. SMA does reduce the diffing overhead in CG as 40 percent of the diffs are eliminated, but the overhead of handling page requests increases in a way that the data access latency remains roughly the same. In the case of Em3d, SMA does not provide improvements because it exchanges large pages for short diffs, increasing the network traffic, as we mentioned before.

ADSM and Combined achieve the best overall performance, except for Em3d, as these systems take advantage of their primary techniques without negative side-effects. Nevertheless, the Em3d results demonstrate that we should consider not only the data sharing pattern, but also the size of diffs when deciding what technique to apply.

Finally, we believe that the performance trends suggested by these results should extrapolate to systems that are larger than eight PCs only. However, scaling the systems further would only make sense if problem sizes are increased as well, as indicated by the speedups in Table 3. A similar increase in both system size and problem size would suggest a significant increase of the fraction of busy time for these applications, due to their higher than

linear computational complexities. On the other hand, such a scaled system would also incur more cold start faults, more sharing, and a higher level of resource contention for some of these applications. The exact overall effect of our latency-tolerance techniques on the scaled system would depend on how these different effects balance out.

7.3 Summary

Our experimental results show that, in terms of isolated techniques, SMA achieves the best performance for most of our applications, even though Adaptive++ produces the most consistent reductions in data access overhead. Our results also show that combining SMA with IUA (ADSM) produces performance that is at least as good as that achieved by either of these two techniques independently. Finally, we find that the Combined technique (ADSM plus Adaptive++) is able to reduce data access overheads by almost 50 percent, on average. However, compared to the ADSM results, the gains achieved by Combined are only marginal, since most of our applications are dominated by single-writer pages. ADSM and Combined achieve the best overall performance.

8 CONCLUSIONS

In this paper, we evaluated the isolated and combined effects of three different latency-tolerance techniques for page-based software DSMs: prefetching, selective update-based coherence, and selective single-writer coherence. Our experimental results for six applications on a cluster of eight PCs show that carefully combining techniques obtains at least as high an improvement as each strategy in isolation, except for one application. When we combine all the techniques we study, the barrier-based applications achieve up to 29 percent running time (42 percent speedup) improvement, whereas the lock-based application achieves 54 percent running time (110 percent speedup) improvement.

Based on our experience and results, we draw the following conclusions:

- Latency-tolerance techniques can reduce data access overheads significantly. However, these techniques have to be designed carefully in order to avoid increasing other overheads. For example, we find that (even sophisticated) prefetching can increase IPC overheads, whereas (even selective) update-based coherence can increase synchronization overheads.
- In isolation, prefetching is better than selective update-based coherence. However, selective single-writer coherence achieves the best results, as it can reduce data access overheads significantly without noticeable side-effects.
- Despite previous negative results, combining latency-tolerance techniques can achieve better performance than any technique in isolation. The key is that the combination has to be designed carefully to avoid interference between the primitive techniques.

Finally, despite the performance improvements due to our latency-tolerance techniques, our results suggest that there is room for further improvement. In particular, the data access and synchronization overheads of several

applications are still significant, even when multiple latency-tolerance techniques are combined.

ACKNOWLEDGMENTS

The research described in this paper was partly supported by FINEP/Brazil grant # 56/94/0399/00 and CNPq/Brazil.

REFERENCES

- [1] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel, "Software DSM Protocols that Adapt between Single Writer and Multiple Writer," *Proc. Third Int'l Symp. High-Performance Computer Architecture*, Feb. 1997.
- [2] C. Amza, A.L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel, "Adaptive Protocols for Software Distributed Shared Memory," *Proc. IEEE, Special Issue on Distributed Shared Memory Systems*, vol. 87, no. 3, pp. 397-532, Mar. 1999.
- [3] 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*, vol. 29, no. 2, pp. 18-28, Feb. 1996.
- [4] 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," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
- [5] R. Bianchini, R. Pinto, and C.L. Amorim, "Data Prefetching for Software DSMs," *Proc. Int'l Conf. Supercomputing*, July 1998.
- [6] R. Bianchini, R. Pinto, and C.L. Amorim, "Page Fault Behavior and Prefetching in Software DSMs," Technical Report ES-401/96, COPPE Systems Eng., Federal Univ. of Rio de Janeiro, July 1996.
- [7] A. Bilas, D. Jiang, and J.P. Singh, "Accelerating Shared Virtual Memory via General-Purpose Network Interface Support," *ACM Trans. Computer Systems*, vol. 19, pp. 1-35, Feb. 2001.
- [8] M. Castro and C. de Amorim, "Efficient Categorization of Memory Sharing Patterns in Software DSM Systems," *Proc. 15th Int'l Parallel and Distributed Processing Symp.*, Apr. 2001.
- [9] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "Software versus Hardware Shared-Memory Implementation: A Case Study," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, Apr. 1994.
- [10] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li, "Improving Release-Consistent Shared Virtual Memory Using Automatic Update," *Proc. Second IEEE Symp. High-Performance Computer Architecture*, Feb. 1996.
- [11] P. Keleher, "Update Protocols and Cluster-Based Shared Memory," *Computer Comm.*, vol. 22, pp. 1045-1055, July 1999.
- [12] L.I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott, "VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, May 1997.
- [13] L. Monnerat and R. Bianchini, "Efficiently Adapting to Sharing Patterns in Software DSMs," *Proc. Fourth IEEE Symp. High-Performance Computer Architecture*, Feb. 1998.
- [14] T. Mowry, C. Chan, and A. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory," *Proc. Fourth IEEE Symp. High-Performance Computer Architecture*, Feb. 1998.
- [15] M. Rangarajan and L. Iftode, "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance," *Proc. Third Extreme Linux Workshop*, Oct. 2000.
- [16] C.B. Seidel, R. Bianchini, and C.L. Amorim, "The Affinity Entry Consistency Protocol," *Proc. 1997 Int'l Conf. Parallel Processing*, Aug. 1997.
- [17] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," *Proc. 16th ACM Symp. Operating Systems Principles*, Oct. 1997.
- [18] R. Stets, S. Dwarkadas, L. Kontothanassis, U. Rencuzogullari, and M.L. Scott, "The Effect of Network Total Order and Remote-Write Capability on Network-based Shared Memory Computing," *Proc. Sixth IEEE Symp. High-Performance Computer Architecture*, Jan. 2000.

- [19] K. Thitikamol and P. Keleher, "Per-Node Multi-Threading and Remote Latency," *IEEE Trans. Computers*, vol. 47, no. 4, pp. 414-426, Apr. 1998.
- [20] L. Whately, R. Pinto, M. Rangarajan, L. Iftode, R. Bianchini, and C.L. Amorim, "Adaptive Techniques for Home-Based Software DSMs," *Proc. 13th Symp. Computer Architecture and High Performance Computing*, pp. 164-171, Sept. 2001.
- [21] W. Hu, W. Shi, and Z. Tang, "Home Migration in Home-Based Software DSMs," *Proc. First Workshop Software Distributed Shared Memory*, June 1999.
- [22] S. Wiel and D. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *Computer*, vol. 30, no. 7, pp. 23-30, July 1997.
- [23] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, May 1995.

Raquel Pinto received the DSc degree in systems engineering and computer science from the Systems and Computer Engineering Program at COPPE, Federal University of Rio de Janeiro, Brazil, in 2001. Since 2002, she has been an assistant professor with Estácio de Sá University, Brazil. Her current research interests include parallel and distributed computing.



Ricardo Bianchini received the PhD degree in computer science from the University of Rochester in 1995. From 1995 until 1999, he was an assistant professor at the Federal University of Rio de Janeiro, Brazil. Since 2000, he has been an assistant professor with Rutgers University. His current research interests include the performance, availability, and energy consumption of cluster-based network servers. He has received several awards, including the US National Science Foundation CAREER award. He is a member of the IEEE Computer Society and the ACM.



Claudio L. Amorim received the MSc and PhD degrees in computer science at Imperial College, London, United Kingdom, in 1980 and 1984, respectively. He is an associate professor in the Systems and Computer Engineering Program at COPPE, Federal University of Rio de Janeiro, Brazil. He is a cofounder of the Parallel Computing Laboratory where he led several R&D projects including the MACACO project in 1988, which resulted in the first parallel computer prototype in Brazil. He is an advisor on high-performance computing for Brazilian R&D Agencies and was a cofounder of the Symposium on Computer Architecture and High-Performance Computing in 1987. During 1995-1996, he was a visiting professor at the Department of Computer Science, University of Rochester, New York. He served on the editorial board of the *Scientific Programming Journal* (1992-1999) and the *EuroMicro* (1994-1995). He has published more than 90 technical papers in conferences and journals, and two books (in Portuguese) on advanced computer architecture. His current research interests include scalable cluster-based network servers for multimedia and e-commerce applications. He is a member of the IEEE, ACM, and SBC-Brazil. More information can be found at <http://lcp.coppe.ufrj.br>.

▷ For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.