Update Protocols and Iterative Scientific Applications
Pete Keleher
University of Maryland

Abstract
Software DSMs have been a research topic for over a decade. While good performance has been achieved in some cases, consistent performance has continued to elude researchers. This paper investigates the performance of DSM protocols running highly regular scientific applications. Such applications should be ideal targets for DSM research because past behavior gives complete, or nearly complete, information about future behavior. We show that a modified home-based protocol can significantly outperform more general protocols in this application domain because of reduced protocol complexity.

Nonetheless, such protocols still do not perform as well as expected. We show that the one of the major factors limiting performance is interaction with the operating system on page faults and page protection changes. We further optimize our protocol by completely eliminating such memory manipulation calls from the steady-state execution. Our resulting protocol improves average application performance by a further 34%, on top of the 19% improvement gained by our initial modification of the home-based protocol.

1. Introduction
Iterative scientific applications would seem to be ideal applications from the perspective of a software distributed shared memory (DSM) system. While aspects of such programs differ, many have highly regular sharing behaviors. The set of shared data accessed by individual threads is often invariant from one iteration to the next. This regular behavior can be used by DSMs to predict future accesses, and to move data in advance of subsequent accesses [1-3]. Such update protocols allow much of the latency of remote data fetches to be hidden. Given reasonably efficient communication, DSMs should be able to achieve good speedups on such applications.

The output of parallelizing compilers, such as SUIF [4], is a good source for this type of application. While parallelizing compilers can analyze many sequential programs well enough to generate message-passing versions, this process is by no means easy or common. The process of generating parallel applications that run in shared memory environments requires far less analysis. Further, the set of applications that can currently be analyzed well enough to turn into a shared memory application is much larger than for message-passing applications.

By combining parallelizing technology with sophisticated runtime systems [1, 3, 5], we can create a programming environment that is flexible and easy to use. Scientists are not required to write message-passing programs or use data-parallel languages such as HPF. Instead, they can write sequential programs, rewriting a few computation-intensive procedures, and adding parallelism directives where necessary. This combination has the advantage of producing programs that can run on large-scale parallel machines as well as the more pervasive multiprocessor workstations. This portability is important for scientists and engineers who want to develop applications that run well on their multiprocessor workstations, but who desire the ability to scale their applications up for larger parallel machines as needed. The combination of ease of use and scalability of the resulting software is a key appeal of shared-memory compilers.

1.1 Contributions
This paper presents the design and performance of several new protocols to handle this and similar types of applications. We first look at a relatively straightforward update protocol based on multi-writer lazy release consistency (LRC) [6].

We then show that modified home-based [7] protocols can perform even better than LRC protocols for applications of this type. Whereas "home-less" LRC protocols can perform poorly for applications that modify (and communicate) large amounts of data, home-based protocols maintain relatively little state, and such states lives have short lifetimes. The main drawbacks of home-based protocols are related to problems adapting to dynamic sharing patterns, precisely the sort of pattern that the applications we are investigating here do not have.

The rest of the paper is organized as follows. Section 2 discusses the background of LRC and home-based protocols and describes the specific protocols that will be investigated in this work. Section 3 presents the performance of these protocols and attempts to relate their differences to how the protocols interact with specific sharing patterns. Sections 4 and 5 describe and analyze the performance of extensions to our home-based protocols that impose less of a load on the underlying operating system. Finally, Section 6 concludes.

2. Background and Protocol Descriptions
Our home-based protocols are based on those discussed by Zhou [7]. Home-based protocols are, in turn, based on the multi-writer lazy release consistent (LRC) protocols used by DSMs such as TreadMarks [8] and CVM [9].

2.1 Multi-Writer LRC Protocols

2.1.1 Lmw-i
LRC protocols allow the shared memory system to delay performing shared updates until specific synchronization events occur. For the applications in this study, this means the next barrier. The advantage of LRC in this context is that sharing (either true or false) does not cause invalidations between barriers.

This is useful in two cases. First, consider an example with true sharing. Shared data item x is initially valid on both nodes i and j. If process pi writes to data x during the same barrier epoch (between the same two barriers) in which pj reads x, the value returned by the read does not depend on the relative ordering of the read and the write. The value that is returned by the read is always the last value written prior to the previous barrier. By contrast, the canonical sequentially consistent [10] memory model would require node j's copy of x to be invalidated before pi.
is allowed to modify it. The result is that if the write occurs before the read in "wall-clock" time, the read will return the new value. However, if the read occurs before the write, the previous value is returned.

Consider where this is useful. If the programmer intended \( p_i \)'s read to return the previous value of \( x \), then an anti-dependence exists between the two accesses. The compiler would need to add an extra round of synchronization to enforce the anti-dependence for a sequentially consistent system. For an LRC system, on the other hand, the notice of the write is only propagated to \( p_j \) at the next barrier. The anti-dependence is therefore protected without recourse to extra synchronization.

The second case where LRC is useful for our applications is in hiding the effects of false sharing. Assume two data items \( x \) and \( y \) reside on the same shared page. Sequentially consistent systems require processes to gain exclusive access to shared pages before modifying any items that reside on the pages. Therefore, modification of \( x \) and \( y \) by distinct processes during the same barrier epoch requires the processes to communicate in order to arbitrate access to the page. By contrast, multi-writer LRC protocols allow the modifications to proceed in parallel, without communication. The separate modifications are merged at the next synchronization.

\textbf{Lmw-i} implements an invalidate-based multi-writer LRC protocol. Multi-writer protocols allow multiple processes to simultaneously modify the same page without network communication. Our programs are presumed to be free of data races, so such concurrent modifications are constrained to be to disjoint sections of the pages. At the end of each barrier epoch, the modifications to each shared page are captured in the form of diffs. A diff is a run-length encoding of the changes made to a single virtual memory page. Diffs are created by performing a page-length comparison between the current contents of the page and copy of the page that was created at the first write access. If each concurrent writer summarizes its modifications as a diff, the system can create a copy that reflects all modifications by applying the concurrent diffs to the same copy.

Since Lmw-i is an invalidate protocol, these diffs are not immediately sent to other processes. Instead, structures called write notices are distributed to other processes via existing synchronization (barrier) messages. Each write notice informs the recipient that a shared page has been modified, and the recipient invalidates any local copy of the page. The write notice also names the diff that needs to be applied in order to bring the local copy of the page up to date. Accesses to invalid pages cause page faults, during which the diffs named by write notices are retrieved and applied to the faulting page. Once the diffs have been applied, the page protections are re-validated and the application continues.

\subsection*{2.1.2 Lmw-u}

\textbf{Lmw-u} is an extension of \textit{lmw-i} that uses updates to communicate data whenever possible. A more complete description can be found in the literature [1], but we provide a short description here.

Accesses to shared pages are tracked by using per-page copysets, which are bitmaps that specify which processors cache a given page. This information can be used to selectively employ a hybrid invalidate/update coherence protocol. Coherence for pages that are consistently communicated between the same set of processors can be updated, rather than invalidated, after writes. Such updates eliminate page faults. Coherence for the remaining pages is maintained by using the above invalidate protocol.

On the first iteration of each time-step loop, copysets for all pages are empty, and page faults can occur. By the second iteration, however, copyset information accurately reflects stable sharing patterns by indicating the processors that need each page. Page faults can be then be eliminated by sending any local updates to all processors on the copyset for each page. Since updates are sent before the data is needed, subsequent remote page faults are avoided.

\textit{SUIF} was modified to automatically insert calls to DSM routines to mark pages to be updated at barriers. For a given page, local modifications are then flushed to all other processors in the page's local copyset at each barrier. A processor \( p \) is inserted into processor \( q \)'s copyset for a page if \( p \) requests a diff for the page, or if \( q \) sees a write notice for the page that was created by \( q \).

Compiler analysis needed to use such a protocol is much simpler than communication analysis needed in HPF compilers. The identities of the sending/receiving processors do not need to be computed at compile time. Instead, the compiler only needs to locate data that will likely be communicated in a stable pattern, then insert calls to DSM routines to apply the update protocol for those pages at the appropriate time. More precise compiler analysis can be used to explicitly clear or set the copysets of data to be communicated. The compiler's annotations do not need to be guaranteed correct, since the page annotations only affect efficiency, not program correctness.

As previously discussed, barrier flushes of updates (essentially a restricted update model) have both advantages and disadvantages. On the plus side, flushes ideally move data before it is needed, allowing computation and communication to be wholly overlapped. The result can be fewer page invalidations and page faults. A second advantage is that lost flush messages do not affect correctness, only performance. Flush messages can be unreliable, and therefore do not need to be acknowledged. A `flush' therefore consists of only a single message, whereas a miss to shared data incurs at least one request and response message pair.

All consistency information in lazy-release-consistency systems is piggybacked on synchronization messages (barrier messages in the case of compiler-parallelized applications). By contrast, diff requests are inherently two-way, and so cost two messages. On the minus side, if sharing patterns are not stable, out-of-date copysets will cause data to be sent to processors that do not need it. Correctness is not affected, but the unneeded flushes cause unnecessary overhead.

\subsection*{2.2 Home-Based Protocols}

Home-based protocols differ from "homeless" protocols, such as lmw-i and lmw-u, in that each page has a statically-assigned \textit{home}. Each update to a page is flushed to the home at the next synchronization. The result is that the "home" copy of each page is always up to date, at least with respect to the synchronization operations that have been performed at it.

The home-based protocol has two potential advantages: the "home" effect, and the short lifetimes of many data structures. As discussed above, homeless protocols summarize modifications in the form of diffs. Aside from the expense of creating and applying diffs, they can cause homeless protocols to have voracious appetites for memory because diffs have long lifetimes. The "home" effect refers to the fact that home-based protocols allow the owner of a page (i.e. the home) to dispense with creating diffs describing its own modifications. Diffs are created only to describe modifications made to a page by processes other than the
page’s home node. Modifications made by the home node are merely noted locally. No network communication is required.

Second, home-based diffs have short lifetimes. The most inelegant aspect of homeless protocols is that the data structures that describe shared modifications cannot be discarded until explicitly garbage-collected. For example, consider Figure 1. This example shows three processes, \( P_1 \) through \( P_3 \), that access migratory data \( x \). First, \( P_1 \) modifies \( x \) and releases a synchronization variable. \( P_2 \) then acquires the synchronization variable, causing any local copy of the page that contains \( x \) to be invalidated.

Lastly, \( P_3 \) accesses \( x \).

With a homeless protocol, touching \( x \) causes a page fault that is satisfied by requesting the diff of \( w(x) \) from \( P_1 \). However, the diff can not be discarded by \( P_2 \) even after it has been supplied to \( P_2 \), because \( P_1 \) can not know if or when some other process (\( P_4 \), for example) might subsequently request the diff as well. More generally, the diff can not be discarded until the system can guarantee that no process will request it in the future. The situation is complicated even more by the fact that if and when \( P_1 \) requests \( \text{diff}_2 \), the diff may be requested from \( P_2 \) rather than \( P_1 \). For performance reasons, then, \( P_3 \) can not discard the diff either. The result is that no diff, nor any of the write notices that name diffs, can be discarded until garbage-collection occurs.

By contrast, diffs have very short lifetimes under home-based protocols. Diffs are created at synchronization points, flushed to the home nodes, and immediately discarded. This is correct under home-based protocols because all page faults are serviced via complete page copies from the home node, rather than by applying diffs to pre-existing page replicas.

Consider Figure 1 again. Assuming that the manager of the affected page is \( P_2 \), both \( P_1 \) and \( P_2 \) will create and send diffs to \( P_2 \) prior to their releases. \( P_2 \) will also need to request a new copy of the page from \( P_1 \) before it can perform the local modification. The advantage is that both diffs can be immediately discarded. The disadvantage of this approach is that more messages are sent than with homeless protocols. The situation can be even worse. Consider the case where a fourth process, \( P_4 \), is the home node for the page. In this case, both \( P_1 \) and \( P_2 \) will send diffs to \( P_2 \). Both \( P_2 \) and \( P_4 \) will then request copies of the page from \( P_1 \), a node that isn’t even involved in the communication. This behavior means that home assignment must take expected sharing behavior into account in order to get efficient communication. Furthermore, dynamic sharing patterns would require additional mechanisms in order modify home assignments.

To summarize, home-based protocols have the following advantages:

1. Modifications made by the home node do not require diffs to be created. They do need to be tracked, however, and so still require local interrupts the first time each page is modified during each barrier epoch.
2. Validating a page always requires exactly one request-reply pair since validations are always accomplished by retrieving a complete new copy of the page.
3. There is very little persistent state. No garbage collection is needed.

There are also several obvious disadvantages:

1. The home node must be chosen wisely, and the application must not change sharing patterns drastically.
2. Communication of data between two non-home processes requires the data to be sent through the home node. Consider a piece of migratory data \( x \). Each time \( x \) moves from one process to the next, it must be sent back to the home node in the form of a diff, and then paged in from the second node. By contrast, the data travels directly from one process to the next in a homeless protocol.
3. Potentially more diffs are created, as diffs are created promptly at the end of each interval rather than lazily [11], as with homeless protocols.

### 2.2.1 Bar-i and Bar-u

In order for the “home” effect to be useful, modifications of shared data should be mainly done by the data’s home. This is somewhat analogous to the “owner computes” rule often used in parallelizing compilers. Unsurprisingly, therefore, the output of parallelizing compilers is a good candidate for a home-based protocol. Our bar-i protocol is a simplified home-based protocol that has several extensions to support iterative scientific codes.

First, by limiting the protocol to codes that only use barrier synchronization, we can prevent any diff or consistency state from living past the next barrier. Page coherence is maintained by using a per-page scalar version index, which is maintained by the page’s home node. The index is incremented with any local modification of the page (but only once per barrier epoch), and for each applied diff from another node. Indexes for modified pages are distributed via the barrier process. The new page versions are used by other processes to decide which pages need to be invalidated.

Second, bar-i has been augmented to provide explicit support for reductions. Many of our applications were automatically parallelized by SUIF [4], and reductions are used in most of these codes. Accesses to shared pages are tracked by using per-page copysets, which are bitmaps that specify which processors cache a given page. This information can be used to selectively employ a hybrid invalidate/update coherence protocol. Using updates rather than invalidates allows page faults to be eliminated for pages that are consistently communicated between the same processes. Coherence for the remaining pages is maintained using an invalidate protocol in order to avoid excessive communication.

On the first iteration of the time-step loop, the copysets of each page are empty and page faults occur. By the second iteration, however, copyset information indicates the processors that need each page, accurately reflecting stable sharing patterns.

Third, our barrier protocols assign page homes at runtime, rather than requiring the compiler or user to do so statically. The most obvious drawback of a scheme with statically assigned homes is that the initial assignment must be done well. Secondly, the scheme will not adapt if an application undergoes a phase change; a point in the code where the set of pages written by a given process change.

Zhou [7] addressed the problem of assignments by requiring user annotations on each section of data, and observed that making such assignments is easy for the majority of cases. This is especially true for scientific applications, which tend to distribute

![Figure 1: Diff Exchanges](image-url)
computation among processes as large array slices. Even when this is the case, however, such annotations are an additional burden on the programmer.

Our protocols collect access behavior information during the first iteration of a program, and migrate pages before the second iteration begins. We migrate any pages that have not been written by their initial owner, but have been written by at least one other process. The migration decisions are distributed on release messages at the next barrier.

Similarly to lmw-u, then, page faults can be eliminated for bar-u by updating processors on the copyset for each page, sending the data before it is accessed. Additional extensions are discussed in Section 4.

3. Experimental Results

3.1 Applications

The applications used in this study are summarized in Error! Reference source not found. The shared segment size is the size of the shared portion of the address space, while “Sync. Gran.” is the average period between barrier synchronizations. Barnes is a version of the n-body simulation from SPLASH-2 [12] that has been modified to use less synchronization. Expl is a dense stencil kernel typical of those found in iterative PDE solvers. FFT is a three-dimensional implementation of the Fast Fourier Transform that uses matrix transposition to reduce communication. Jacobi is a stencil kernel combined with a convergence test that checks the residual value using a max reduction.SOR is a simple nearest-neighbor stencil. Shal and Swm are different versions of the shallow water simulation, differing primarily in synchronization granularity. Both swm and tomcat are programs from the SPEC benchmark suite and contain a mixture of stencils and reductions. We used the APR version of tomcat, in which the arrays have been transposed to improve data locality.

In all cases, speedups are calculated with reference to a single-process version of the same program with all page protection changes using an 8k granularity. We used 8k pages in CVM by the simple expedient of ensuring that all page protection changes use an 8k granularity.

3.2 Experimental Environment

Our experimental environment consists of an 8-node IBM SP-2. The SP-2 has a high-performance switch (HPS) in which each bidirectional link is capable of a sustained bandwidth of approximately forty megabytes per second. Each processor is a 66MHz RS/6000 POWER2 and has 128 megabytes of memory. The applications were run on a version of CVM that uses UDP/IP over the high-performance switch.

Simple RPC’s in our environment require 160 μsecs. Remove page faults require 939 μsecs. In the best case, AIX requires 128 μsecs to call user-level handlers for page faults, and improt system calls require 12 μsecs. However, the costs of virtual memory primitives in the current system are location-dependent, occasionally increasing the cost of page protection changes by an order of magnitude.

Although AIX’s default virtual memory page size is 4k bytes, we used 8k pages in CVM by the simple expedient of ensuring that all page protection changes use an 8k granularity.

3.3 Base Results

Table 1 shows the number of diff creations, misses (remote faults) that cause network traffic, the number of data and synchronization requests sent (there are an equal number of replies), and the total amount of data communicated, in kilobytes. Protocols lmw-i, lmw-u, bar-i, and bar-u are abbreviated li, lu, bi, and bu, respectively.

Since all of our applications are repetitive scientific computations, the update versions of lmw and bar are almost uniformly faster than the invalidation versions. Both update protocols eliminate the majority of remote misses in most of the applications. The exceptions are Barnes and swm, both of which perform much worse for the update version of lmw than for the invalidate version. In both cases, the poor performance is an artifact of the data structures used to store out-of-order updates.

Second, the home-based protocols outperform the homeless protocols for all but jacobi, which performs similarly for both update-based protocols. A number of factors contribute to this difference. First, the home effect allows the invalidate version of bar to create an average of 36% fewer diffs than the corresponding lmw protocol. This translates into 31% fewer remote misses, and a total of 49% fewer messages. This is partially offset by the fact that bar-i sends 74% more data. The reason is that lmw moves most data in the form of diffs, whereas bar-i satisfies all remote misses with complete new copies of pages. diffs are usually much smaller than page size.

The complexion of these statistics changes when we look at the update protocols. bar-u uses diffs to push data before it is needed, just as with lmw-u. Since page faults no longer occur (with the sole exception of a small number for shallow running on lmw-u), the total amount of data moved is almost identical between the two update protocols, approximately equal to the amount moved by lmw-i. bar-u creates about 14% more diffs than lmw-i and sends about 12% fewer messages. Overall, bar-u aver-

<table>
<thead>
<tr>
<th></th>
<th>Diffs</th>
<th>Remote Misses</th>
<th>Messages</th>
<th>Data (kbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>li</td>
<td>lu</td>
<td>bi</td>
<td>bu</td>
</tr>
<tr>
<td>Barnes expl</td>
<td>3261</td>
<td>3261</td>
<td>2688</td>
<td>3274</td>
</tr>
<tr>
<td>Expl</td>
<td>632</td>
<td>642</td>
<td>270</td>
<td>648</td>
</tr>
<tr>
<td>fft</td>
<td>2720</td>
<td>2464</td>
<td>140</td>
<td>2464</td>
</tr>
<tr>
<td>jacobi</td>
<td>179</td>
<td>198</td>
<td>77</td>
<td>220</td>
</tr>
<tr>
<td>shallow sor</td>
<td>5501</td>
<td>5929</td>
<td>2882</td>
<td>5929</td>
</tr>
<tr>
<td>sor</td>
<td>126</td>
<td>126</td>
<td>0</td>
<td>126</td>
</tr>
<tr>
<td>swm</td>
<td>4408</td>
<td>4858</td>
<td>4873</td>
<td>7462</td>
</tr>
<tr>
<td>tomatc</td>
<td>898</td>
<td>899</td>
<td>413</td>
<td>911</td>
</tr>
</tbody>
</table>
ages approximately 19% more speedup than the better of the two
lmw protocols.

Although the message count differential can certainly account
for some of the performance difference between the update proto-
cols, there are at least two other factors at work. The first is the
sheer complexity of the homeless protocols. Deciding what con-
sistency actions need to be performed consists of filtering locally
known consistency actions by what is known of the node that is
being synchronized with. Since lmw supports locks, flags, and
other non-global synchronization types, as well as programs with
dynamic sharing behavior, consistency information has long life-
times, and can not be discarded without explicit garbage col-
lection.

Furthermore, nodes running lmw often receive diffs from
nodes other than the diffs’ creators (see [8] for more details). The
result is that in the general case, the producer of a diff often has
limited information about consumers of the diff. This has the
effect of making copysets (and hence updates) less precise. As a
consequence, lmw-u does not immediately validate pages when
diffs that make it possible arrive by update. Instead, lmw merely
stores updates to locally invalid pages and checks to see if all
required diffs are present when the next access to that page oc-
curs. This next access is signaled by a segmentation fault.

By contrast bar-u is designed for static, iterative, barrier-based
programs, and consumer information is distributed globally at
barrier synchronizations. This allows producers to have exact
knowledge of consumers. Hence, consumers of data wait for up-
dates before leaving barrier operations, allowing the segmentation
fauls and additional page protection changes to be avoided.

4. Eliminating OS Memory overhead

Figure 3 shows a breakdown of the execution time for each of the
applications running the bar-u protocol. Runtime is broken into
sigio handling, wait time, operating system overhead, and appli-
cation computation. Sigio handling refers to time spent handling
incoming requests. Wait time is the time spent waiting for remote
requests to succeed. Since nearly all remote misses have been
eliminated, this refers primarily to slaves waiting for barrier re-
lease messages. Operating system overhead consists of time spent
in operating system traps, such as ‘send’, ‘recv’, and ‘mprotect’,
which is used to change page protections. Application computa-
tion is time spent doing useful work.

Note two things. First, several of the applications (fft, shallow,
and swm) have substantial OS components. Although this graph
doesn’t break OS overhead down into contributions from individ-
ual traps, the majority of this time is spent in mprotect calls. This
is an order of magnitude more time than implied by the mprotect
time given in Section 3.2.

Second, the efficiency implied by the “app” component of
several of the applications does not gibe well with the corre-
sponding speedups. For instance, swm spends 41.7% of the time
doing useful work, implying that speedup should be near 8 * 
41.7% = 3.3, assuming that the parallel version does no more
work than the sequential version. However, the actual speedup is
closer to 1.8.

We theorized that these discrepancies are caused by a degra-
dation of the operating system performance when under stress. In
this case, the stress is an application (CVM) that uses memory in
unorthodox ways, i.e. modifying page protections in large address
spaces in an unpredictable order. In order to test our theory, we
decided to modify bar-u in order to minimize operating system
traps that manipulate the application’s address space. We could
presumably make CVM’s behavior appear more orthodox if we
ceased manipulating page protections and ceased using segment-
tation violations (segvs) to detect invalid accesses. Once CVM’s
behavior fit the standard envelope, we would expect CVM’s per-
formance to scale much better.

4.1 Bar-s

Eliminating these two mechanisms one at a time, we first address
the use of segvs. Segvs are used for write trapping, the trapping of
inappropriate accesses to pages. These can be either write ac-
tesses to pages that are readable but not writable, or any type of
access to invalid pages. If segvs are not used, some other method
of write trapping must be used. Given the repetitive nature of our
applications, the obvious choice is to use historical behavior to
infer future behavior. We call the protocol that uses this technique
bar-s.

After gathering information for some period of time, bar-s
goes into overdrive mode, and uses another method of write trap-
ing. This is analogous to what the update protocol already does
via copysets. However, copysets are indiscriminate in that they
provide no information on how shared accesses relate to synchro-
nization.

For example, Figure 5 shows two iterations of a single process in
a parallel CVM run. Each iteration is composed of two barrier
epochs: that of barrier 1, and that of barrier 2. After barrier 1, p1
modifies data x. After barrier 2, p2 modifies y. The application
goes into overdrive mode after the first iteration. Hence, given the
behavior during the first iteration, we expect x to be modified after
the next occurrence of barrier 1, and y to be modified after
the next occurrence of barrier 2. Bar-u would trap this write by
making a twin when the first write to the page during the barrier
epoch occurs. However, without recourse to segvs, we must as-
Figure 5: Iterations and Overdrive

Figure 4 shows the speedups of bar-s and bar-m for seven of our eight applications. The best speedup from the two lmw protocols and bar-u’s speedup are shown for comparison. Barnes is not shown because its sharing pattern, although iterative, is highly dynamic. Work is allocated via non-deterministic traversals of a shared tree structure, resulting in slightly different sharing patterns each iteration. The other seven applications ran correctly under bar-s and bar-m without modification.

Figure 4 shows that bar-s improves upon bar-u by only an average of 2%, indicating that segv handling is not a major source of overhead. However, bar-m achieves a 34% gain on top of the 19% gain from bar-u to the lmw protocols. Since the shared access behavior of the applications is invariant across iterations, bar-u, bar-s and bar-m send exactly the same number of messages and communicate the same amount of data. Hence the difference is entirely due to the lack of memory system interaction via mprotect calls.

5. Conclusions

While protocols like bar-m are not likely to be the first option for many users of software DSMs, they could be useful in specific circumstances. Their primary application domain would likely be large, iterative scientific codes, like most of those discussed in this paper. Such applications tend to stretch operating systems in ways that are unexpected for the operating system developers. However, eliminating interrupts and kernel traps will always improve performance even if operating system support is tuned for DSM-like consistency actions.

Nonetheless, compilers are a more trustworthy source of information, whether they be parallelizing compilers or compilers for explicitly parallel languages. The information needed to determine that sharing behavior is invariant is clearly a subset of the information that is needed in order to determine precisely what that sharing behavior is. We therefore expect that this information could be obtained more easily and therefore could be more commonly implemented. Moreover, there is likely to be a large class of applications for which determining invariance of access patterns is possible, but determining the nature of the patterns is not.
We then described bar-u, a modified home-based [7] protocol that performs even better than LRC protocols for this type of application. The reasons include “home” effects, and the fact that home-based protocols are much less complex. Whereas “home-less” LRC protocols can perform poorly for applications that modify (and communicate) large amounts of data, home-based protocols incur less system overhead because they maintain relatively little state, and such state has very short lifetimes. The main drawbacks of home-based protocols are related to problems adapting to dynamic sharing patterns, precisely the sort of pattern that the applications we are investigating do not have.

Finally, we presented and analyzed bar-s and bar-m, two protocols that successively strip away all reliance on kernel protection manipulation. The result is that the application’s behavior conforms more closely to the type of behavior expected (and optimized for) by the underlying operating system, and therefore performs much better. Overall, our update home-based protocols average 51% better than the original lmw invalidate protocols for our environment.

7. Bibliography


