Characterizing Parallel Workloads to Reduce Multiple Writer Overhead in Shared Virtual Memory Systems

Salvador Petit, Julio Sahuquillo, and Ana Pont

Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
Cno. de Vera s/n, 46071, Valencia (Spain)
{spetit, jsahuqui, apont}@disca.upv.es

Abstract. Shared Virtual Memory (SVM) systems, because of their software implementation, enable shared memory programming at a low design and maintenance cost. Nevertheless, as hardware implementations become faster, their performance is still far from that achieved by distributed shared memory (DSM) systems. Nowadays, SVM use relaxed memory consistency models and multiple writer protocols as techniques to reduce latencies and false sharing, respectively; however, these techniques induce additional overhead that decreases system performance.

In this paper, we perform a statistical study of workload behavior aimed at improving the design of SVM protocols. The work focuses on the identification of the type of shared data patterns that can appear in the accesses to protected sections using semaphores. Most coherence actions in SVM systems are performed as a consequence of the write operations executed in critical sections, so this study pays special attention to the write operations performed when multiple writers are allowed. As these write operations may present spatial locality, we also study the write patterns on shared pages with similar behavior.

Different software filters are applied in the instrumented parallel workloads selected to capture and classify the most common sharing patterns. This enables the recognition of those patterns in which coherence overhead can be reduced by modifying the coherence actions performed by the protocol. Despite the fact that performance evaluation of new coherence solutions is not the main goal of this paper, the ideas presented to improve the behavior of SVM systems can be implemented at a reasonable hardware/software cost.

Keywords: shared virtual memory systems, memory consistency protocols, workload characterization.

1 Introduction

Shared Virtual Memory (SVM) [1] is an economic method to implement shared memory systems. The underlying operating software environment detects writes to shared addresses by using the virtual memory that takes care of coherence maintenance. These characteristics make SVM systems cheap and portable compared with other interconnection architectures; however, they have two main performance drawbacks. First, the probability that false sharing will appear is high because pages are used as the consistency unit. Second, coherence maintenance is performed via software with long communication latencies.

To alleviate these drawbacks, research has focused on relaxed memory consistency models and multiple writer protocols. Memory consistency models define when coherence actions must be performed to maintain the semantics of the parallel programs. Current relaxed memory consistency models delay the coherence actions, saving communication. Coherence actions are actually page invalidations (also called write notices) that take effect under those conditions defined by the memory consistency model and usually not immediately. Multiple writer protocols allow several writers on a page at the same time, hiding most of the false sharing effects. In general, these kinds of protocols store and send page differences (also called diffs) instead of whole pages to detect which parts of a given page have been written by each node. Nevertheless, write notices and diffs introduce additional overhead due to their software management.

The main goal of this paper is to identify the sharing patterns of parallel workloads that can improve the design of SVM protocols by reducing the protocol overhead derived from current multiple writer capabilities. If those sharing patterns (i.e., serialized writers or page partition) are relatively significant, current SVM
protocols can be improved by taking into account this behavior in the design of coherence actions. Though performance evaluation and detailed implementation are not in the scope of this paper, we discuss the hardware/software complexity that this new proposal would assume.

The remainder of this paper is organized as follows. Section two presents some background about shared virtual memory systems. In section three, we introduce the main purpose of this paper. In section four, we describe the framework used. Section five presents the sharing patterns recognized in this study. In section six, we present some ideas to take into account during the design process of consistency protocols to reduce the overhead. Finally, in section seven, the main conclusions are drawn.

2 Shared Virtual Memory

SVM systems introduced by K. Li et al. [1] are Software Distributed Shared Memory (SDSM) organizations that detect writes to shared data by using the states of virtual memory pages. Initially, shared pages are in read-only state; then, when a process in a node (usually, a symmetric multiprocessor system) writes to a page, the correspondent processor issues an exception. As a consequence, the operating software environment performs coherence maintenance and communicates with those nodes sharing that page. Past research [2, 3, 4, 5, 6, 7, 8] has mainly focused in reducing latencies inherent in software management and false sharing effects. Below we summarize research in relaxed consistency models and multiple writer protocols.

2.1 Relaxed Memory Consistency Models

Most shared memory programs are data-race-free [2] so they are properly-labeled [5] (usually with semaphores and barriers) to avoid data races. This assumption implies that coherence actions can be carried out at any time between the write of a process in the source node and the lock performed by a process in the target node; because both events are partially ordered by a happened-before [2] relationship.

All Lazy Release Consistency model (LRC) [4, 5, 6, 7, 8] protocols basically postpone the coherence actions until a process of a given node executes a lock operation. Then, the node invalidates those pages corresponding to the write notices previously issued according to the happens-before partial ordering. Figure 1 shows an example of how write notices are applied. The horizontal (execution time) and diagonal (lock access ordering) arrows indicate the happens-before partial ordering. Writes of nodes A and B happen-before the lock call of node C, so their write notices are sent when C acquires the lock.

![Coherence actions in the LRC model. Legend: wn(i,a) represents a write notice from node i to address a.](image)

Initial LRC implementations postponed the sending of the write notices until a target process acquires the semaphore. Thus, the node of the acquiring process receives only those write notices that the process needs to

---

Note that others protocols can choose to update instead of invalidate; however, this is unusual in SVM systems due to the large size of the coherence unit.
be aware of, and thus reducing network traffic. However, this technique increases the protocol overhead because it needs a garbage collection mechanism to free those resources taken by the write notices received by every node. Recent SVM implementations [7, 8] broadcast the write notices earlier, reducing contention in lock acquires and avoiding garbage collection, but increasing network traffic.

2.2 Multiple Writer Protocols

Several published techniques focus on how to reduce false sharing by allowing several processors to write at the same time to the same page.

Most Release Consistency (RC) [3, 4, 5, 7, 8] protocols use diffs to achieve multiple writer capabilities. A diff is a compressed table of differences between the initial read-only page and the written page. The nodes update data by sending diffs instead of whole pages, which enable several nodes to write to a page at the same time. Diffs may update the target page in several ways, depending on the multiple writer implementation.

Figure 2 shows three classic implementations of multiple writer RC protocols using diffs. Under Eager Release Consistency (ERC) [3] diffs are broadcast earlier to all nodes sharing the page. In Standard Lazy Release Consistency (SLRC) [4] diffs are postponed until an invalidated target node asks for them. Home Lazy Release Consistency (HLRC) [5] generates diffs earlier, and applies them to the written page in the home processor.

ERC improves network traffic due to diff broadcasts, SLRC saves this traffic but increases protocol overhead due to diffs garbage collection, and HLRC may exhibit contention in the home nodes. In these three protocols, both creating and applying diffs adds its own overhead that grows linearly with the page size. Automatic Update Release Consistency (AURC) [6] avoids diffs because it uses specific hardware [9] to perform writes at word granularity. By using hardware snooping, the SMP bus in the nodes detects writes and sends them to the interconnection network. Though the approach enhances performance, it is a flexibility constraint because the additional hardware is highly coupled to the node and the interconnection network.

Fig. 2. Diff updates in RC protocols.
3 Motivation

Performances of any kind of computer systems are based on the characteristics of the workload running on them; i.e., some recent cache schemes [10] use two independent cache organizations to exploit the kind of locality that data exhibit (spatial or temporal). As we are interested in the design improvement of SVM protocols, an initial step must be made to study those characteristics of the parallel workload that can help designers reduce that overhead.

Research in SVM systems introduces the techniques discussed in section 2 to reduce network traffic and false sharing; although, their applications present new overhead. One of the sources of the overhead comes from the multiple writers capability. Each time a node has a page fault, it needs an up-to-date copy of the page, which is an aggregate of diffs created by previous writers. In a pure software SVM cluster, each writer creates one or more diffs for the page to be applied later in one or more nodes. The cost to create and apply diffs grows linearly with the page size. The worst case appears in the LRC protocol, where the faulting node asynchronously asks every writer involved for the diffs, and so interrupting their potentially useful workload computation. The HLRC protocol tries to palliate that overhead by concentrating the asynchronous communication in the home node of the page; but as only one home node exists per page, it may become a contention point. This problem gets worse if that node is also the home of more frequently accessed pages.

Semaphores synchronize parallel workload in SVM systems becomes a potential source of serialization; thus, they may limit the number of multiple writers in the parallel workload. The overhead introduced when multiple writers are considered in pure software SVM systems is due to the use of multiple writer capability. This paper studies to which extent they become necessary in typical parallel workloads. In cases where multiple writers (and diffs) are needed, the study checks the spatial locality for write operations in shared pages. The spatial locality that may help us speed up the diff creation and application occurs when a given page is written in several neighboring words.

To do that, we instrument the parallel workload to trap the synchronization and write operations issued. Below, the instrumentation technique and target parallel workload are detailed.

4 Experimental Framework

The tool used to instrument the workload is a part of LIMES [11], an i386 SMP execution driven simulator similar to TangoLite [12]. In our experiments we only use the compiler and instrumentation tool provided by the SMP simulator. LIMES uses a modified version of GCC v2.6.3 which compiles applications with the O2 flag. The instrumentation tool traps memory accesses by adding augmentation code that calls the memory simulator after each memory reference. The synchronization operations can also be trapped by redefining the ANL macros to memory simulator calls.

To carry out our experiments we use eight benchmarks (Barnes, Cholesky, FFT, FMM, LU, Ocean, Radix, and Water) from the SPLASH-2 benchmark suite. As in [13], the measurements are taken just after the parallel processes are created. Table 1 shows the problem size used for each benchmark, as well as the number of semaphores and acquires obtained under such problem size. Every benchmark was executed considering 32 processes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem Size</th>
<th>Total semaphores</th>
<th>Total acquires</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>2K particles</td>
<td>78</td>
<td>4579</td>
</tr>
<tr>
<td>Cholesky</td>
<td>1k 14.0</td>
<td>64</td>
<td>21559</td>
</tr>
<tr>
<td>FFT</td>
<td>32K points</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FMM</td>
<td>2K particles</td>
<td>22</td>
<td>4449</td>
</tr>
<tr>
<td>LU</td>
<td>512x512 points</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ocean</td>
<td>66x66 ocean</td>
<td>2</td>
<td>3648</td>
</tr>
<tr>
<td>Radix</td>
<td>128K integer</td>
<td>32</td>
<td>2048</td>
</tr>
<tr>
<td>Water</td>
<td>512 molecules</td>
<td>516</td>
<td>17728</td>
</tr>
</tbody>
</table>

Table 1. Benchmarks characteristics
The characterization study results are independent of the system architecture because we trap the memory accesses and synchronization operations directly from the workload, before they arrive at the memory system. Thus, to reduce the memory requirements of the simulator, each computing process runs in a dedicated node with a one single issue, one instruction per cycle, processor. Processors share memory through a perfect RAM (PRAM) memory model.

The gathered traces of the trapped accesses contain, for each memory reference, the following information:

- The processor identifier.
- The memory operation (read or write).
- The virtual address of the referenced data.
- The identifier of the current semaphore (if the memory operation occurs in a section protected by a semaphore).

5 Sharing Patterns

Most coherence actions in SVM systems are performed as consequence of the write operations carried out in a protected section. Therefore, we will focus on the identification of the type of shared data patterns that can appear in the accesses to protected sections using semaphores.

We also will pay special attention to identifying the write patterns associated with a shared page in order to recognize the locality of those writes in parallel workloads.

Both patterns will be helpful to characterize the workload in SVM systems in order to propose new ideas to avoid a large amount of the overhead produced in these architectures due to consistency maintenance.

5.1 Serial and Concurrent Data Sharing

Two assumptions may imply that parallel workloads can limit the use of multiple writer protocols by synchronizing using semaphores. First, processors accessing the same semaphore have a high probability of sharing the same data. This assumption seems reasonable because a parallel program tends to associate certain data to certain semaphores. Program locality also gives data a high probability of being accessed in the same code areas, like caches base their effectiveness on data localities. Other more relaxed consistency models like Entry [14] and Scope [15] are based on this characteristic of the workload code, though they force the programmer to define this relation in the source code. Second, writes to shared data have also a high probability of happening in protected sections. This assumption is reasonable too, although exceptions (like some implementations of distributed linked lists) can occur. From these two assumptions, we can conclude that writers to the same shared data may be serialized at the same protected sections. Furthermore, common practices in concurrent programming show that readers of shared data will access sections protected by the same semaphores as writers, so every access to the same shared data may be serialized using the same semaphore.

5.1.1 Filtering Traces

To test the above assumptions, a software filter applied to traces gathered from the benchmark execution is implemented. When a process accesses a memory address, we check if any other process has written the same address. If so, we check if such process wrote to the section protected by the same semaphore. In such a case, because semaphores serialize writers there is no need for a multiple writer coherence mechanism. We refer to these writes as serial shares. On the other hand, if any other process wrote outside the semaphore, such writes could have been performed concurrently and so multiple writers capabilities become necessary. We refer to those writes as concurrent shares. If there is not a previous writer, we call the access a cold share. Figure 3 shows the pseudocode of the filter algorithm and plots the results for an 8KB page size. The filter results are independent of the page size because the filter classifies data shares at granularity of word. LU and FFT do not appear in the figure because they have no semaphore (as shown in Table 1).

Figure 3.b shows that, in general, the percentages of serial shares between processes accessing a given semaphore is meaningful among the benchmarks, confirming our previous assumptions. The only exception can be found in Barnes with a value of just 7%. The remaining cases reach a value higher than 18%, Radix
even surpasses 60%. On average, serial shares are nearly three times more frequent than the concurrent shares. That is shown clearly in Table 2, which summarizes these percentages. As cold shares represent accesses to unwritten data words just written during the cold start, they have been removed because they are not useful for our proposals.

```plaintext
Algorithm shares
Begin
  For Each Access Do
    If (the access is inside a semaphore) Then
      If (there is no previous writer to the address) Then
        COLD_SHARES++ /* Data was written during the cold start */
      Else
        If (any other process has written to the same data) Then
          If (the data was written in the current semaphore) Then
            SERIAL_SHARES++
          Else
            CONCURRENT_SHARES++
          End If
        End If
      End If
    End For
End
```

a) Software filter.

![Chart](chart.png)

b) Percentages of shares.

Fig. 3. Software filter to classify shared data accesses in protected sections and results.

Table 2. Concurrent versus serial shares.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Concurrent</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>Cholesky</td>
<td>20%</td>
<td>80%</td>
</tr>
<tr>
<td>FMM</td>
<td>45%</td>
<td>55%</td>
</tr>
<tr>
<td>Ocean</td>
<td>2%</td>
<td>98%</td>
</tr>
<tr>
<td>Radix</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Water</td>
<td>5%</td>
<td>95%</td>
</tr>
<tr>
<td>Average</td>
<td>27%</td>
<td>73%</td>
</tr>
</tbody>
</table>
5.2 Writing Localities

Write operations will likely perform over a chunk of continuous words (ranging from only one to the full page) due to data localities. Those write locality patterns could be used in several ways to reduce diff overhead. As in the previous section, we implement a software filter to count the occurrence of those profitable write locality patterns.

5.2.1 Filtering Traces

We classify the possible situations in four categories depending on the locality of writes performed by a computing process in a page. The classification approach is as follows:

- The process writes the full page.
- The process only writes in continuous addresses.
- The process writes just a single word.
- The process only writes in discontinuous addresses.

Figure 4 presents the software filter that takes account of write localities, as well as results. When a process references an address previously written by another process, it checks the type of write locality that the previous writer exhibited in the page. Then, it clears the statistics of the previous writer for that page (so as not to cause a jam in latter accesses).

```
Algorithm localities
Begin
  For Each Access Do
    If (the address was written by other process) Then
      Switch (locality of writes of the other process)
        The process wrote the full page:
          FULL++
        The process wrote just a single word:
          SINGLE++
        The process only wrote in continuous addresses:
          CONTINUOUS++
        The process wrote in discontinuous addresses:
          DISCONTINUOUS++
      End Switch
      Reset statistics of writes of the other process
    End If
  End For
End
```

a) Software filter.
b) Percentages of write patterns.

Figure 4. Software filter to classify page writes and results.

Figure 4.b plots the percentages of discontinuous, continuous, single word, and full page write operations obtained when varying the page size. We use a very small page size (256B) to check how the percentage of full page writes depends on the page size. As can be seen, for larger sizes (1KB, 4KB and 8KB) the percentage is negligible, with the only exception of FFT with a page size of 1 KB.

As expected, due to false sharing, the percentage of full pages is smaller when the page size is larger. However, for all the page sizes used, there is a large percentage of continuous and single patterns. To check the commonest sizes of continuous patterns, Table 3 represents the distribution of the sizes of each continuous pattern generated by all the benchmarks using an 8KB page size. Most write chunk areas less than 256 bytes and just 7.1% of chunks have a larger size.

<table>
<thead>
<tr>
<th>Chunk Size Area</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 256B]</td>
<td>92.90</td>
</tr>
<tr>
<td>[256B, 1KB]</td>
<td>5.22</td>
</tr>
<tr>
<td>[1KB, 4KB]</td>
<td>1.20</td>
</tr>
<tr>
<td>[4KB, 8KB]</td>
<td>0.69</td>
</tr>
</tbody>
</table>

6 Implementation Ideas to Improve SVM Protocols

In this section we discuss some ideas that could be implemented to reduce the multiple writer overhead and diff overhead when frequent serial shares occur and spatial locality is detected.

6.1 Reducing Multiple Writer Overhead

To carry out all the suggestions explained in section 5.1, the first design step is to associate each page with the semaphore where the write operation was performed. Then, the invalidated page is marked as written by that semaphore. In an ideal case, where only serial shares would occur, each node stores the same semaphore descriptor each time it has to invalidate a page. It is possible that several nodes store different semaphore descriptors for the same page because each node does not change semaphore when it writes into a page. This situation happens when different parts of the page are written on different semaphores by several nodes. Figure 5 shows a possible scenario where two sets \( I, J \) of nodes are serialized by two semaphores \( r, s \) for writing in a page \( p \). Nodes \( i_1, i_2, \ldots, i_n \) and \( j_1, j_2, \ldots, j_n \) have just passed their critical sections and generated write notices.
Fig. 5. Semaphores serializing the writers to a page. Legend: wn(i, a, r) represents a write notice from node i to address a in the section protected by the semaphore r.

We can take advantage of this situation by allowing invalidated nodes to ask for the whole page from the last node that wrote to that page in the protected section by the associated semaphore. This is possible because the invalidated node knows (by means of the write notices) that this page was only written in that section, and the writers had to access the requested page in serial order.

This technique could improve LRC-based protocols by reducing the number of computed diffs, which are related to computing time and memory consumption. It also allows dedicated hardware to make copies of the whole page, instead of asynchronously interrupting a potential computing node to compute a diff. Some examples of using available hardware for this purposes can be found in [7] and [8]. HLRC-based protocols could also benefit from this situation by spreading the contention among the home nodes.

Ideally, home nodes do not receive update requests because invalidated nodes can request the up-to-dated pages from the last writers in the semaphores. That is close to a multiple home protocol. In this protocol, we consider a main home that collects diffs like those found in HLRC protocol. The remaining homes are migrating across serial writers, and there are as many homes for each page as the number of semaphores whose protected sections write to that page. To have an overall perspective of how an increase in the number of homes would benefit the system performance, we summarize in Table 4 the mean number of writers per page varying the page sizes in 1, 4 and 8KB. As can be seen, there are several benchmarks with a high density of writers per page, even for a small page size; that means that they could benefit from a multiple home protocol like the one discussed above.

Table 4. Mean number of writers per page varying the page size.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1KB writers/page</th>
<th>4KB writers/page</th>
<th>8KB writers/page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>4.8</td>
<td>5.6</td>
<td>5.3</td>
</tr>
<tr>
<td>Cholesky</td>
<td>2.0</td>
<td>2.7</td>
<td>3.5</td>
</tr>
<tr>
<td>FFT</td>
<td>1.0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>FMM</td>
<td>3.5</td>
<td>5.1</td>
<td>5.5</td>
</tr>
<tr>
<td>LU</td>
<td>8.0</td>
<td>8.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Ocean</td>
<td>1.1</td>
<td>1.3</td>
<td>1.6</td>
</tr>
<tr>
<td>Radix</td>
<td>18.9</td>
<td>23.6</td>
<td>21.8</td>
</tr>
<tr>
<td>Water non</td>
<td>3.8</td>
<td>3.8</td>
<td>3.5</td>
</tr>
</tbody>
</table>

We think that the discussed technique could offer a potentially higher advantage than home migrations as proposed by Stet et al. [7], though they can be applied together, because in [7] only one home migrates (our main home).

The LRC consistency model enables concurrent shares so the ideal case may not occur every time the workload is running; thus, we must consider a mechanism to contemplate them. If a node receives a write notice from an unexpected semaphore, the write could have been concurrent. Thus, to have an up-to-date copy of the page, a multiple writer mechanism must be available. In the case of a HLRC-based protocol, the simplest solution is to ask the main home for an up-to-date page. LRC-based protocols can request the writer for the diff related to the write notice. If the invalidated node receives more than one write notice, it needs to request each previous writer for its diff. If these writers are serialized, a possible improvement could be to combine the twin page of the first writer with the page copy of the last writer in order to compute an accumulative diff. This represents a tradeoff with the network traffic because the technique involves three nodes (requester, twin owner, and up-to-date copy owner) per diff request.

6.2 Reducing Diff Overhead

The ideas commented in section 5.2 could significantly reduce the number of asynchronous diff requests in pure software SVM protocols, but in some cases they must still be used. Diff calculation as implemented today is a summary of the writes of a certain writer in a page; and it is general enough to allow writers to intercalate data in the same page and so enabling full multiple writer capabilities.
The detection of patterns of continuous writes can be performed via software by comparing the twin page with the written page, or via simple hardware by snooping the write addresses.

When that situation is detected, it can be notified by indicating the address and size of the page chunk that was written along with the write notices. This action is likely to increase the performance of LRC based protocols, because invalidated nodes can ask for a copy of the written chunk instead of an asynchronously calculated diff. Invalidated nodes can also receive the whole page, provided they are notified through write notices, which page chunks have been written. In the case of discontinuous writes, it is possible to send a bitmask (as wide as the words in the page) with the write notice indicating the written words in the page. By using these bitmasks, as in the continuous written chunk case, the nodes can ask for the whole page instead of asynchronously initiate a diff calculation. As a lateral effect, avoiding diff calculation in LRC saves memory because diffs are stored until garbage collection time.

The protocol designer can select a range of continuous pattern sizes (i.e., from 1 word to 64 words) in which all the writings are updated by the write notices. This option slightly increases network traffic when sending write notices because they are larger now; however, it will reduce a high percentage of asynchronous communication both in LRC and HLRC protocols as small-size single and continuous writing patterns are frequent enough. In addition, HLRC contention in homes will be reduced if all the updates for their pages are sent along with write notices.

Detection of only single patterns can be performed without intrusive hardware by means of double page faults. The first fault indicates that there was a write, then the page is write protected to detect any other write. If no write occurs, the writing pattern is just a single word. As results show, single writing patterns are so frequent that sending them as write notices could save a high percentage of asynchronous communication. The induced overhead is very cheap in terms of network traffic because write notices would be just two words larger (address and value), but the double page faults represent a tradeoff to be taken into account.

7 Conclusions

Coherence actions carried by SVM memory consistency protocols are strongly dependent on the data sharing patterns of the running workloads; thus, it is worthwhile addressing consistency protocol design at this point. This paper focuses on how the workload sharing patterns behave and it is aimed at helping protocol design.

Multiple writer capabilities introduce overhead in SVM protocols by using asynchronous communication to calculate diffs, or to request pages to home nodes. Diff calculation has an overhead that grows linearly with the page size. Homes would introduce contention when they become overloaded. To set bounds to this overhead, this paper concentrates the sharing patterns of parallel workloads from experimental traces.

First, we see if parallel processes make an extensive use of those multiple writer capabilities. Experiments show that, on the average, sharing between processes is mainly serialized by semaphores. Accesses to potentially concurrent written data are three times less frequent than those serialized by semaphores. That workload behavior can be taken into account in protocol design to reduce diff calculation time, diff memory consumption, and to spread home contention; i.e., this can be achieved by allowing assistant homes to store those pages whose writers are serialized by a semaphore.

Second, when the overhead of multiple writer capabilities cannot be avoided, it is still possible to optimize protocols by profiting from the writing locality of processes. Results show that a significant percentage of writers write in continuous areas before other processes access their written data. Furthermore, around 93% of the continuously written areas is smaller than 256 bytes. One can directly update those small areas thus reducing diff memory consumption as well as asynchronous communication. Furthermore, early updates could also reduce home contention.

Some software and hardware implementations are also discussed in order to adapt the protocols to workload behavior. Most of the implementations would not only improve protocol performance but also they add small complexity.

As for future work, we are planning to implement and check the performance of the ideas discussed in this paper by modifying some of the current protocols.
References