J Supercomput (2006) 38:49–72
DOI 10.1007/s11227-006-7866-4

Addressing a workload characterization study to the design of consistency protocols

Salvador Petit · Julio Sahuquillo · Ana Pont · David Kaeli

Abstract Shared Virtual Memory (SVM) provides a low-cost and effective way to implement the shared-memory programming paradigm. SVMs utilize a number of concepts that include consistency models/protocols, sharing patterns, false sharing, and fragmentation issues. The range of issues encountered in an SVM introduces a level of complexity and presents a challenge to many SVM researchers.

This paper presents a careful study of SVM systems focusing on how the workload characteristics can affect the performace of consistency protocols. This knowledge is used to propose a novel consistency protocol that improves the system performance.

This paper pursues two main goals: (i) to illustrate how different SVM workload characteristics are interrelated, and (ii) to motivate the design of a new multiple-writer memory consistency protocol. To achieve the first goal, we provide a detailed workload characterization analysis and discussion on how consistency models and protocols work. To achieve the second goal, we describe a software-based SVM protocol that achieves better performance than a hardware protocol proposed in the literature. In some workloads, the speedup obtained over the baseline protocol is more than 20%.

Keywords Shared virtual memory · Memory consistency protocols · Asynchronous communication · Workload characterization · Performance evaluation

1 Introduction

Li and Hudak first introduced the SVM system concept in [1] and presented implementation details in [2]. Four main features define an SVM system: (i) nodes share a common virtual

S. Petit (✉) · J. Sahuquillo · A. Pont
Department of Computer Engineering, Polytechnic University of Valencia, Spain
e-mail: spetit@disca.upv.es
e-mail: jsahuqui@disca.upv.es
e-mail: apont@disca.upv.es

D. Kaeli
Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts
e-mail: kaeli@ece.neu.edu

memory address space using the virtual memory system provided by the supporting operating system; (ii) a page is the sharing unit; (iii) the supporting software (operating system, libraries, etc.) takes charge of maintaining coherence of shared pages; and (iv) the workload is not dependent upon the underlying interconnection network or node hardware. These features make SVM systems especially attractive because they allow the use of shared memory code without involving modifications to the hardware platform.

SVM systems are usually composed of several processing nodes (single or dual-processor nodes, or SMP systems), connected by a commodity network. Since nodes are physically distributed and independent (and thus potentially heterogeneous), this approach offers great flexibility when maintaining and upgrading the processing nodes of the systems.

While SVM provide a number of advantages, they suffer from performance limitations due to their inherent software implementation, as well as from issues present in SVM workloads such as *critical section dilation* [3] and *sharing pattern conversion* [4]. Most current parallel workloads have been optimized to run efficiently on distributed hardware systems (e.g., SMPs or clusters). SVM systems lack hardware support for a lot of tasks supported by these hardware systems. Therefore, SVM systems experience performance degradation because they implement these tasks using software-based asynchronous communication protocols [5].

We begin this paper by analysing key workloads characteristics (both theoretically and empirically) that are responsible for performance loss. Then, we review the present set of consistency protocols/models, which are the rules that the system must follow in order to enforce data consistency, paying close attention to multiple-writer protocols that have been shown to achieve good performance.

Our work offers a detailed vision of the whole process involved in the design of a consistency protocol. In other words, how more efficient consistency protocols can be designed using the knowledge obtained from workload characterizations. In this sense, this paper join and extends the work performed in [7, 8, 24].

The remainder of this paper is organized as follows. Sections 2 and 3 discuss workloads characteristics which are used to identify the main sources of performance loss, as well as present an empirical study which quantifies these characteristics in a set of benchmarks. Section 4 reviews consistency model fundamentals and the protocols used for implementing those models, focusing primarily on multiple-writer protocols that have been shown to achieve better performance. Section 5 details the step-by-step design process we followed to arrive at a new protocol that achieves improved performance. Finally, Section 6 presents some concluding remarks.

2 Axes of characterization

Some of the key characteristics of shared memory behavior include critical section dilation and sharing pattern conversions. To better understand shared-memory workloads in an attempt to characterize the intrinsic behavior associated with critical section dilation and sharing pattern conversion, we selected three axes: the *frequency of sharing*, the *granularity of sharing*, and the dynamic changes in the *sharing pattern*.

Frequency of sharing: Coherence actions in SVM systems are carried out at synchronization points; therefore, the frequency of the synchronization operations matches the frequency of sharing. The frequency of sharing metric is calculated as the average computation time between synchronization events [6]. We assume there is *fine-grained* synchronization (FGS) if the average computation time is close to the average synchronization time. Otherwise, the synchronization is *coarse-grained* (CGS).

Granularity of sharing: This characteristic quantifies the average amount of data transferred when an update occurs. This value is computed with respect to the granularity of the system (i.e., the page size, which is typically 4 or 8 Kbytes). The granularity of sharing is classified as *fine-grained* (FG) when only a few words (less than 30%) of the page is shared, *medium-grained* (MG) when at least 30% of the page is shared, and *coarse-grained* (CG) if more than 60% of the page is shared. The granularity of sharing can be further broken down, depending on the type of memory operation performed on the shared data (i.e., *granularity of reading* and *granularity of writing*). Both granularities are commonly present in shared memory workloads and different sizes are commonly observed.

Sharing pattern: During workload execution, data sharing can follow a number of discernable patterns. Sharing patterns can be stable throughout the workload execution or can change dynamically. Based on the number of producers and consumers of data, the sharing pattern for a given instance of data can be classified in one of four categories: (i) *1P-1C,* there is only one producer (P) and one consumer (C). This category includes the case known as migratory sharing, where the consumer becomes the producer of the same data in the future; (ii) *1P-MC*, there is just one producer and multiple (M) consumers; (iii) *MP-1C*, there are multiple producers and only one consumer; and (iv) *MP-MC*, there are both multiple producers and consumers. In addition, we consider the patterns *0P-1C* and *0P-MC*, which refer to one and multiple consumers of the first-loaded data, respectively.

2.1 Background

Previous research in the open literature has explored the behavior of parallel workloads as run on SVM systems [3, 4, 6]. Iftode et al. [4] established a workload taxonomy based on sharing patterns and the granularity of sharing. Jiang et al. [3] modified source code based on the same set of described axes to improve the performance of SVM systems. Zhou et al. [6] studied the behavior of workloads running on several protocols and systems, establishing a number of rules about the optimal granularity of sharing. In their workload taxonomy, they also studied the frequency of synchronization.

These studies have provided us with a sound foundation for this characterization study. Our characterization study presented herein differs in that we gather information about the behavior of the workload, independent of the underlying system. This allows us to focus on the workload behavior objectively, without considering the impact of the underlying hardware system. Table 1 summarizes the axes of characterization that exhibit the workload used in this paper. More details can be found in [7].

2.2 Sources of performance loss

In this section we discuss performance loss issues related to *critical section dilation* and *sharing pattern conversion*. We analyze these issues using our axes of characterization.

2.2.1 Critical section dilation.

Hardware systems can provide for efficient fine-grained synchronization by natively supporting atomic read-modify-write instructions. In contrast, SVM systems depend upon high-latency message-based synchronization constructs. In both types of systems, synchronization

Table 1 Workload characteristics according to the axes of characterization

	Granularity of sharing			
Benchmark	Writing	Reading	Frequency of sharing	Sharing pattern
Radix	FG	MG or CG	CGS	1P-1C
Ocean	CG		CGS	1P-1C
FFT	CG	FG	CGS	1P-1C
LU (continuous)	CG		CGS	1P-MC
LU (discontinuous)	FG		CGS	1P-MC
Barnes	FG		FGS	1P-MC
Water (nsquared)	CG		FGS	Migratory
Water (spatial)	CG		CGS	1P-1C

primitives are used to manage entering critical sections. Critical sections are used inside the workloads to protect shared data or to implement shared task queues. For SVM systems, the time spent in critical sections is greater than in a hardware-based system for two main reasons:

- The cost to support FGS is smaller in a hardware-based system versus a SVM system because SVM systems do not support FGS synchronization directly; synchronization primitives such as locks and barriers are mapped to a distributed set of software queues. In addition, hardware systems do not need to perform metadata maintenance associated with relaxed memory consistency models, as explained in Section 4.2.
- Some SVM systems perform invalidations at synchronization points, increasing the probability that a page fault occurs while executing in the critical section.

Both situations introduce latency due to software message passing and asynchronous communication between nodes. In addition, the software management of the SVM protocol adds more latency.

The sum of these latencies implies that the total time that workloads spend in the critical section is much higher in SVM systems than in hardware systems. Since critical sections are frequently executed, contention increases, which also results in lower performance. This effect is so important that some sections of code, which represent a very small percentage of the total execution time in hardware systems, may become performance bottlenecks in SVM systems.

2.2.2 Sharing pattern conversion.

The execution of each parallel workload follows a given pattern that we will refer to as the *inherent sharing pattern*. This pattern can be static or can change dynamically throughout workload execution. On the other hand, data instances are shared by the workload at a given granularity. We will refer to this granularity as the *workload granularity*, while we will refer to the sharing unit granularity supported by the system as the *system granularity.* In general, the workload granule size is small (less than 64 bytes, i.e., FG) but it can change with the problem size, while the granularity of the SVM system is usually CG. When the workload and system possess different granule sizes, it is probable that a new sharing pattern occurs. The chance that these new patterns arise depends upon the characteristics of the workload and is a function of the difference between the two granule sizes.

Table 2 Benchmark problem sizes

Benchmark	Problem size	Execution cycles
Barnes	8 K particles	47441193
FFT	1 M points	54372159
LU	512 × 512 points	48591413
LU-CONT	512 × 512 points	48589005
Ocean	258 × 258 ocean	29082695
Radix	4 M integers	13389554
Water-NSQ	512 molecules	34211024
Water SP	512 molecules	26965078

When the workload granule is smaller than the system granule size, there are two phenomena that can produce sharing pattern conversion: *false sharing* and *fragmentation*. Both can appear simultaneously.

False sharing appears when the *write* granularity of the workload is smaller than the system granularity. In this case, the producer only writes a fraction of the whole sharing unit, so several producers could write data to the same sharing unit. Thus, the inherent sharing pattern observed with one producer (1P) becomes an induced sharing pattern with multiple producers (MP).

Fragmentation appears when the *read* granularity of the workload is smaller than the system granularity. In this case, the consumer only reads a fraction of the whole sharing unit, so several consumers could read data from the same sharing unit. Thus, the inherent sharing pattern observed with one consumer (1C) becomes the induced sharing pattern with multiple consumers (MC).

3 Workload characterization analysis

In this section we perform an empirical characterization of our target workloads, concentrating on the characterization axes.

We used the Limes execution-driven simulator to instrument our shared memory workloads to capture memory references. Limes, which simulates an SMP system using a similar methodology as TangoLite [9] does, runs on Intel IA32 architectures. The simulator uses a modified version of the GCC v2.6.3 compiler. The instrumentation tool traps memory accesses by adding instrumentation code. The instrumentation code calls the memory simulator after each memory reference. We trap synchronization operations by mapping the Argonne National Laboratory (ANL) macros to memory simulator calls. Measurements are taken just after the parallel processes are created, as described in [10]. Table 2 lists the problem size used for each benchmark. These sizes are chosen to be close to the sizes used in [11]. Every benchmark executes 16 processes.

Experimental results are independent of the system architecture. This is accomplished by trapping both memory accesses and synchronization operations directly from the workload, before they arrive at the memory system. To reduce the memory requirements of the simulator, each computing process runs in a dedicated node on a single issue processor. Processors share memory through a perfect RAM memory model [10].

Memory access information is collected in traces and analyzed offline. For each access we capture the processor ID, the memory operation type, the virtual address of the referenced data, and the simulated time (in processor cycles) when the reference is issued.

3.1 Critical section dilation

During this step in the characterization, we determine the synchronization period, measured as the number of cycles between synchronization operations. We measure the period for one randomly selected process; the characteristics for other processes exhibit similar periodic behavior. Figure 1 shows the average time between synchronization events found in each workload. From the results we can identify two distinct behaviors: (i) applications that have a relatively very small synchronization period (Barnes, Ocean, Radix, and Water-NSQ) and (ii) applications possessing a much longer synchronization period (FFT, LU, and Water-SP).

A synchronization operation can take several microseconds to be processed in a typical SVM system [6], which translates to thousands of clock cycles in current microprocessors. Taking this into account, we classify the workloads in the first group as having FGS synchronization and workloads in the second group as having CGS synchronization.

These values help us identify which workloads may suffer performance degradation due critical section dilation. We further analyze these results by checking if the synchronization period is evenly distributed across the entire workload execution, or instead synchronizations are concentrated in short intervals or "hot spots". To adequately evaluate the dynamic behavior of synchronizations, we calculated both the average and the standard deviation of the period between synchronizations. Results are shown in Table 3. All the benchmarks possess a very large standard deviation, which indicates that the synchronization period is not uniform and there may be hot spots.

To confirm this, we divided the execution time of each workload into *intervals* of equal length (1 million cycles) and measured the synchronization period occurring in each interval. We chose this interval length for all the applications because we found it filtered most of the cyclic behavior. Results are shown in Figure 2. The results with high ordinate values (those points are not shown as they are well off the ordinate scale and are less interesting) indicate that the synchronization period is very high (i.e., there are few or no synchronization points occur in those intervals). The kernel FFT is not plotted because it performs just 5 synchronizations. LU and LU-CONT possess the same shape because they only differ in their data distribution characteristics, and not in their synchronization patterns.

To characterize the synchronization period, we classify a workload as belonging to one of three groups: (i) *pure CGS,* (ii) *medium FGS*, and (iii) *high FGS*.

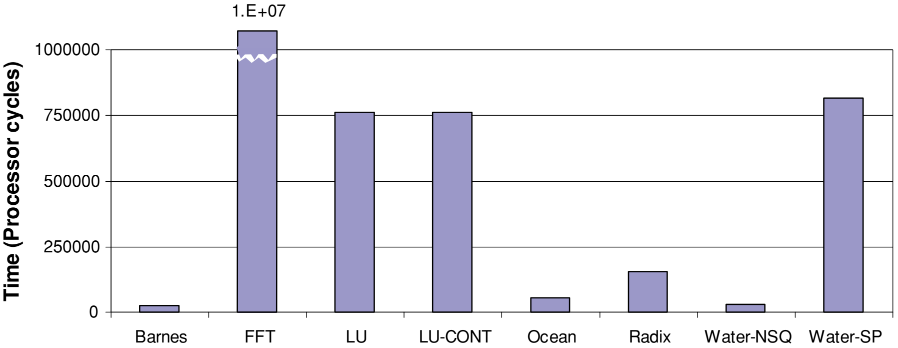

Fig. 1 Cycles between synchronizations

Table 3 Average and standard deviation of the synchronization period

Benchmark	Average	Standard deviation
Barnes	22516	709681
FFT	13593000	12891100
LU	759240	961535
LU-CONT	759202	958547
Ocean	54519	163341
Radix	155345	738216
Water-NSQ	28967	494870
Water-SP	817122	2518130

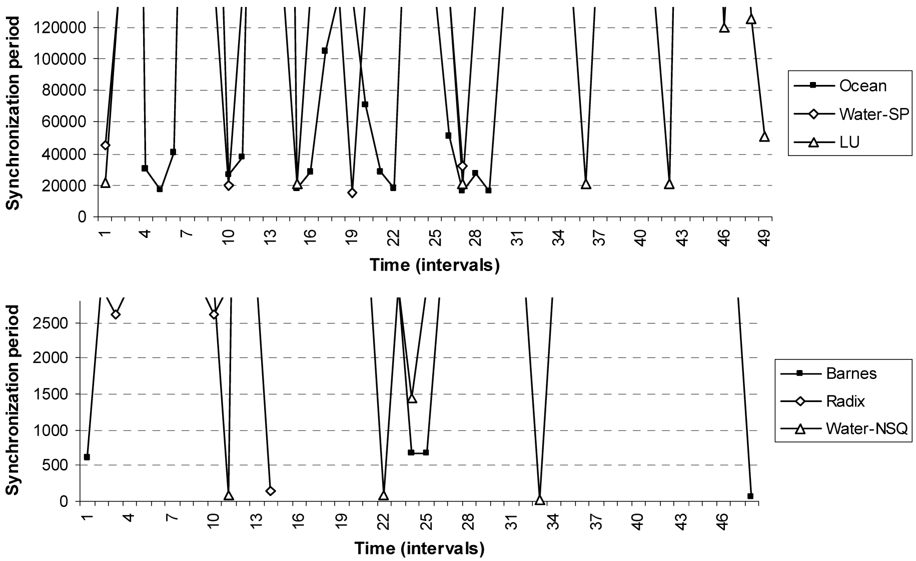

Fig. 2 Synchronization period measured by interval

The pure CGS category includes those workloads exhibiting a large average synchronization period (e.g., FFT, LU, LU-CONT, Water-SP); thus, critical section dilation will not appear. Although these applications (with the exception of FFT) have some intervals possessing a relatively small synchronization period (less than 50 K cycles), the number of synchronizations periods they represent is negligible.

The medium FGS category includes Ocean, where most synchronization periods were smaller than 50 K cycles. Although this period is much shorter than that those found in pure CGS, it is much longer (several orders of magnitude) than the lengths found in the other FGS workloads. Zhou et al. [6] state that Ocean possesses a CGS pattern; however, our results indicate that in some intervals, Ocean is FGS. For example, the synchronization period in the intervals 5, 15, 16, 22, 27 and 29 appear in bursts (with a synchronization period less than 100 cycles). This FGS behavior can produce critical section dilation.

In the high FGS category we find Barnes, Radix and Water-NSQ, which show the shortest synchronization periods, usually less than 1 K cycles. Higher values (e.g.,

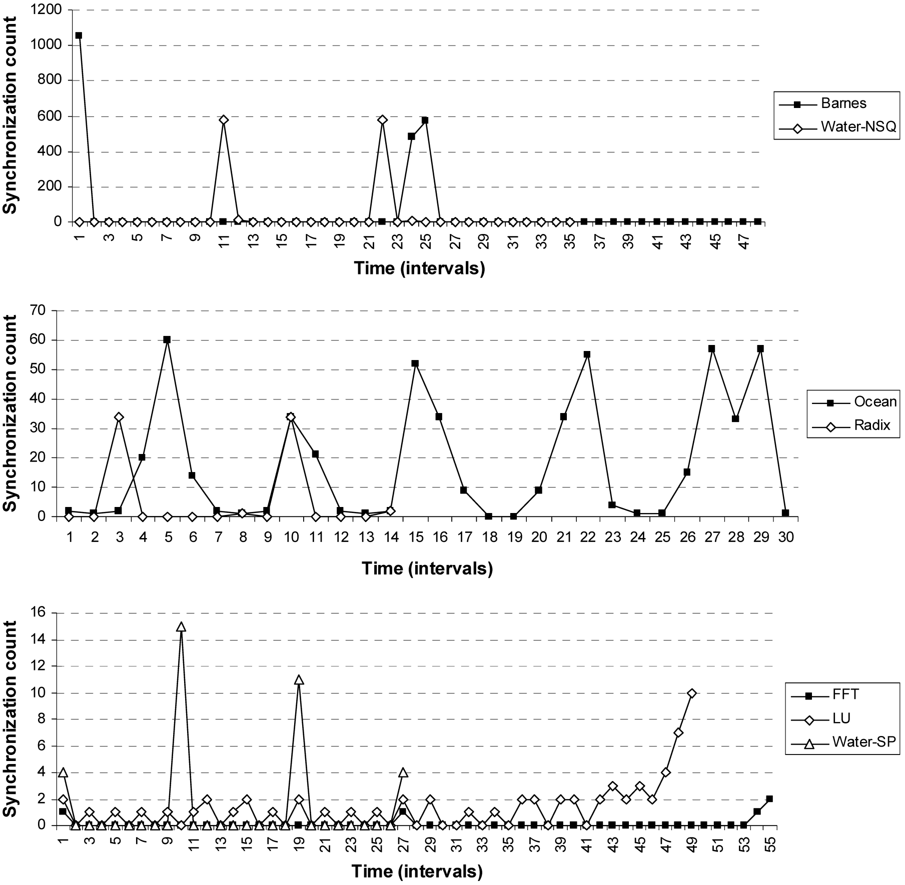

Fig. 3 Synchronization count measured by interval

those found in Radix and Water-NSQ) are due to the impact of outlier values on the average.

The smaller the average period exhibited by an application, the higher the chances that critical section dilation will occur. Iftode [4] and Zhou [6] also come to similar conclusions by identifying those portions of an application that consume the greatest execution time when run on SVM systems.

The probability that critical section dilation is present increases with the number of fine-grained synchronizations. Given this relationship, to determine how critical section dilation can impact on performance, we measured the number of synchronizations issued during each interval. Figure 3 shows the results.

Inspecting the results, one can observe that Barnes and Water-NSQ are the two applications most affected by critical section dilation. Both of these applications suffer from FGS and possess a large number of synchronizations (more than 500). Ocean and Radix show the same behavior, but to a lesser extent. In the case of Radix, synchronizations are distributed in the application, similar to Barnes and Water-NSQ. For Ocean, the synchronizations are spread evenly across the application. Finally, FFT, LU and Water-SP have much fewer synchronizations,

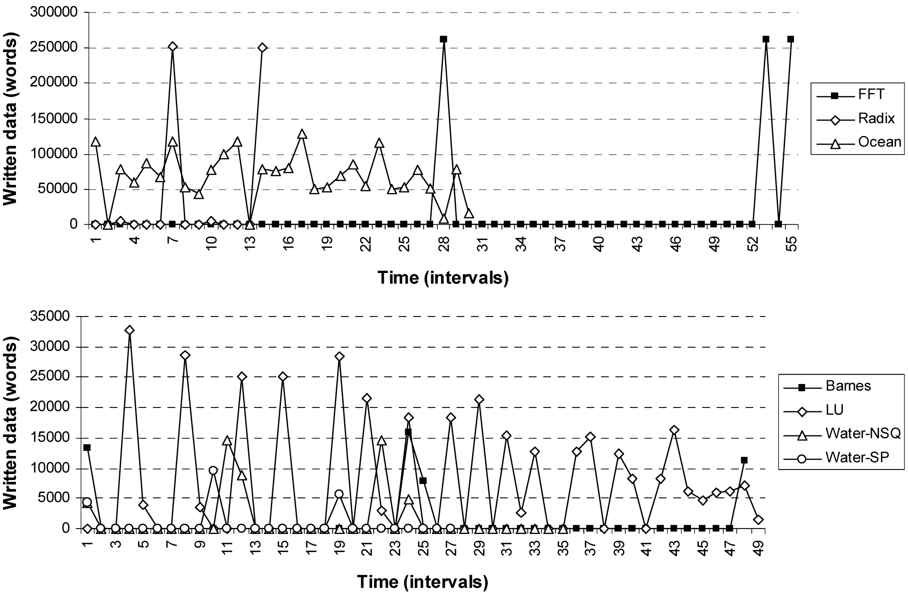

Fig. 4 Total written data between synchronizations for each interval

which together with their CGS behavior, confirm their insensitivity to critical section dilation effects.

The impact of the critical section dilation effect depends also on the amount of data that is shared between synchronizations. Figure 4 plots the amount of data shared between synchronizations for each interval. As LU and LU-CONT share the same amount of data, they are represented by only one plot.

Synchronizations in FFT and Radix occur in only a very few intervals (3 and 2, respectively), but the amount of data they share is very large. In contrast, Ocean and LU share less data per synchronization, though the total amount LU shares matches that of FFT and Radix, while Ocean doubles this quantity. In contrast, both Water and Barnes share only a maximum of 50 K words across their respective workloads.

3.2 Sharing pattern conversion

When sharing patterns change in an application, the management policy of shared data may also need to change. In order to determine the impact of sharing pattern conversion effects in each workload, we gather the workload granularity of sharing, by measuring the size of each instance of shared data written between synchronization points (locks, unlocks and barriers). As in the synchronization period study above, intervals are set to 1 M cycles long. For each synchronization points, we measure the size in words of the writes performed between synchronization points.

Each continuous data write area is called a *chunk*. Figure 5 shows the average chunk size per interval, namely the average granularity. Those points with granularity 0 represent intervals where no writes were performed. Based on the results, workloads are classified in one of three groups: (i) *FG,* (ii) *MG* and (iii) *CG.*

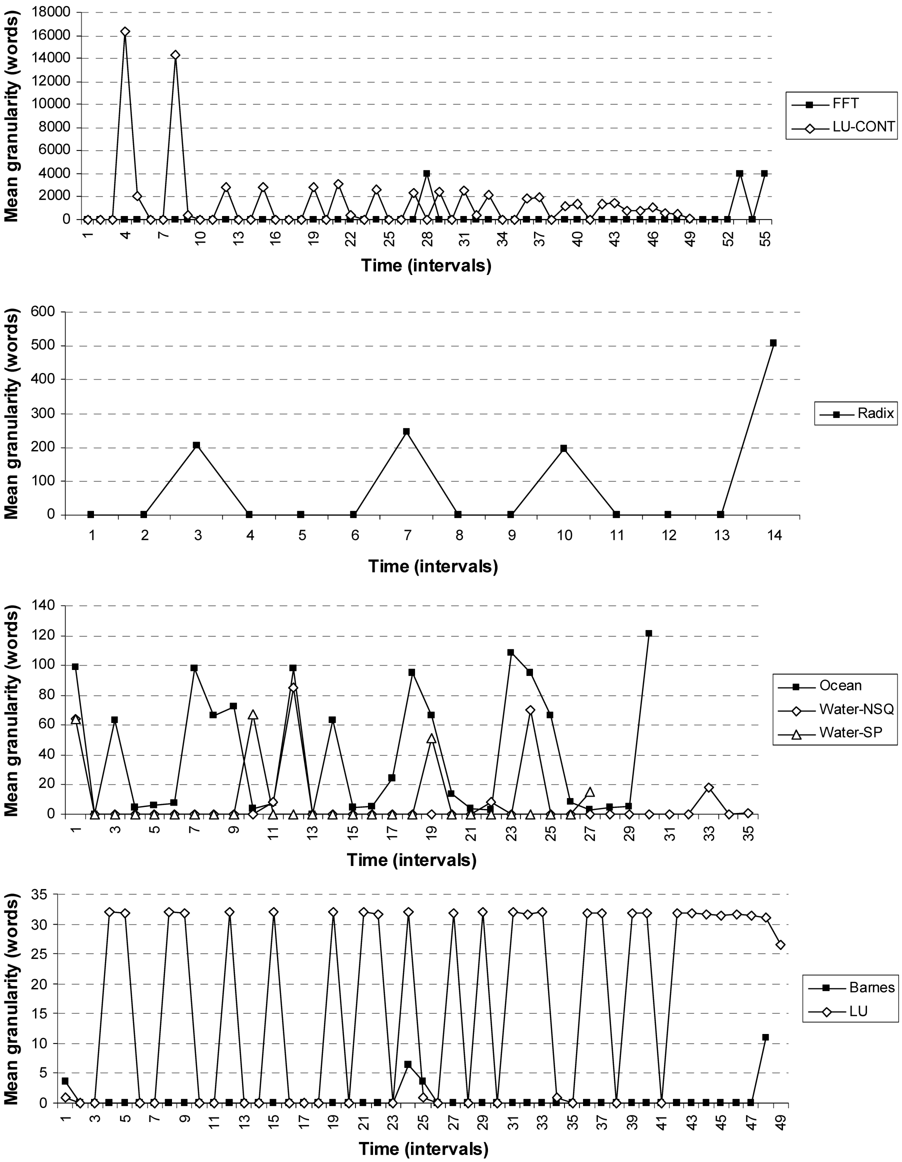

Fig. 5 Mean granularity measured by interval

In the FG category, the average granularity size is by about 10 words. We found that no interval has an average size greater than 128 words. This category includes Barnes, LU, Ocean, Water-NSQ and Water-SP. LU and Barnes possess the smallest average granularity value because they only share chunks smaller than 32 and 12 words, respectively. The MG category only includes the Radix kernel, which performs synchronizations in only a few intervals (just 4 of 14), but with an average granularity of around 326 words. The CG category includes the FFT and LU-CONT kernels. Both workloads exhibit coarse granularity in all the intervals where

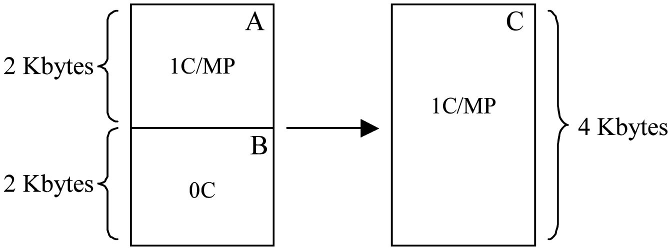

Fig. 6 Fragmentation effect

synchronizations occurred. FFT performs very few synchronization operations (5 barriers in total).

Results show that most workloads (six out of eight workloads) have FG and MG sharing granularity. This means that they are likely to encounter sharing pattern conversion, (i.e., the sharing pattern would change as the system granularity exceeds the workload granularity).

To quantify how large the conversion is at different system granularities and find the cause of the conversion (false sharing or fragmentation), we measured the sharing pattern frequency by increasing the sharing unit size from one word (4 bytes) to one page (4096 bytes). Note that a sharing unit size of only one word (4 bytes) cannot suffer sharing pattern conversion, which appears as the sharing unit size increases.

Figure 7 shows the results. The lines represent the number of bytes exhibiting a given pattern. In addition to the patterns discussed in Section 2.2.2, the figure includes a line for the *0C* patterns, which represents those sharing units having no consumers. We compute the number of bytes based on the size of the entire sharing unit, instead of counting individual bytes accessed. For example, if the sharing unit size is 1024 bytes, and a given sharing unit has a 1P-1C sharing pattern, we add 1024 to the 1P-1C count. Proceeding in this way, the count value represents the data that would suffer from false sharing and fragmentation.

The *0C* pattern represents non-shared data. If the sharing unit size becomes large enough, the non-shared data will join other shared data in a larger unit. This will now be treated as one unit (both the non-shared and the shared) and classified under the same sharing pattern, thus increasing the shared data count. Figure 6 shows an example. If the sharing unit size is 2 KB (left side), the data in the *B* sharing unit is not shared because there are no consumers. When the sharing unit size is 4 KB (right side), the data in the *B* shared unit becomes a subset of the *C* shared unit. Thus, fragmentation reduces the non-shared data count.

For our analysis, both the individual points in Figure 7 and the slope of the lines are meaningful. Individual points can be used to determine how important a given pattern is for a given sharing unit size. The slope of each line helps us identify what is the cause of the sharing pattern conversion as the sharing unit size increases.

The frequency of the 0C pattern observed, which is several orders of magnitude higher than those observed for the other patterns, does not help to explain why sharing conversion evolves but the slope is helpful to understand how the other patterns change. Thus, Figure 7 plots its shape, only shifted down to the X-axis.

In general, the applications can be classified in four groups: those unaffected by increasing system granularity, those that only present false sharing, those that only present fragmentation and those presenting both effects.

Unaffected applications. This group includes applications that encounter neither false sharing nor fragmentation. This is the case for FFT and LU-CONT, which both exhibit a

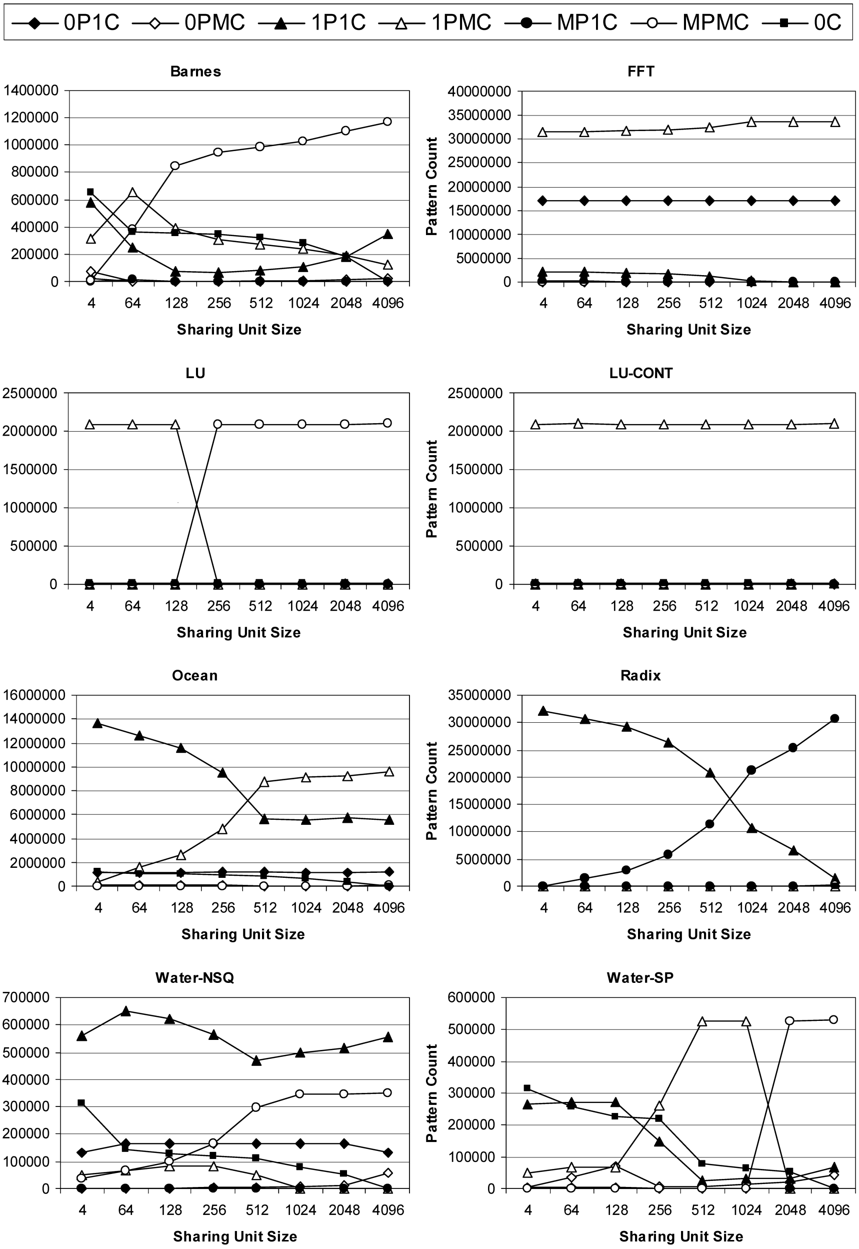

Fig. 7 Sharing pattern count

predominant 1P-1C pattern. In the case of FFT, the other important sharing pattern is 0P-1C, which indicates that a high percentage of the shared data is not written after the initialization phase.

Applications affected only by false sharing. This group includes those applications where patterns with one producer become patterns with multiple producers (as explained in

Section 2.2.2). This is the case of LU and Radix. In LU the inherent 1P-MC sharing pattern becomes MP-MC. In Radix, the inherent sharing pattern 1P-1C becomes MP-1C.

Applications affected only by fragmentation. This group includes those applications where patterns with one consumer become patterns with multiple consumers (as explained in Section 2.2.2). This is the case for Ocean, where its inherent 1P-1C pattern becomes a 1P-MC pattern.

Applications affected both by false sharing and fragmentation. Applications in this group exhibit both false sharing and fragmentation. This is the case of Barnes, Water-NSQ, and Water-SP. Barnes shows inherent 1P-1C and 1P-MC sharing patterns. As the sharing unit increases from 0 to 64 bytes, the frequency of 1P-MC increases due to fragmentation. For larger sizes both false sharing and fragmentation occur because the amount of MP-MC grows. In Water-NSQ, for small sizes (4 and 64 bytes), the frequency of the 0P-1C, 1P-1C, 1P-MC and MP-MC patterns increases due to fragmentation at the expense of decreased 0C. For intermediate sizes (128 and 256 bytes), the MP-MC frequency increases due to both false sharing and fragmentation. In Water-SP, from 256B to 1024B, fragmentation becomes dominant, raising the frequency of the 1P-MC pattern. For larger page sizes, the impact of false sharing is so accentuated that the MP-MC patterns entirely replace the 1P-MC patterns.

4 Memory consistency models and protocols

To understand how the workload characteristics discussed above impact system performance, we need to fully understand how a shared memory system handles the different workload characteristics. One key element is the underlying memory consistency model used. We will discuss these models next, while focusing on multiple writer protocols.

4.1 Memory consistency models

Programmer writing parallel programs would like to be presented with a shared memory system model that is formally defined. This formally defined model is called a *memory consistency model*. A memory consistency model specifies when a memory operation becomes visible to the memory system (i.e., when can the rest of network nodes see the operation result.) The most intuitive model is the *sequential memory consistency model*, which states that at a given time a shared page can be written by only one processor. In other words, write operations performed by different processors must follow a single, well-defined, ordering. This ordering is often referred to as *perform ordering*. This ordering can be easily implemented by using an element to serialize write operations, e.g., a single bus. The perform ordering establishes the sequence of modifications of the shared memory known by all the processors in the system.

Protocols implementing a sequential consistency model are called *single writer protocols* and they are the most intuitive way to guarantee the strict serialization of operations that the sequential model requires.

The performance issues associated with false sharing were the main reason why SVM system designers introduced *multiple writer protocols*. These protocols allow several processes to write to a page in parallel. The potential performance gains obtained through multiple writer protocols are higher for SVM systems than in hardware-based DSM systems because SVM systems manage coherence on a coarser sharing unit (a virtual memory page), which increases the probability of false sharing. When false sharing occurs in a single writer protocol, sharing units will *ping-pong* between nodes. This effect will tend to be magnified in SVM systems because of their inherent software (versus hardware) nature and because of the high latencies of the commodity network.

For these reasons, sequential consistency is not used in current implementations. Current SVM systems use *relaxed memory consistency models* [12], which allow different processors to view different memory access perform orderings; therefore, they support multiple writer protocols. The different perform orderings are due to write operation reordering, which occurs due to network delays, caches, etc.

4.1.1 Release memory consistency model

The *release memory consistency model* [13] is a relaxed memory consistency model that allows memory operations to be reordered between two synchronization operations: *acquire* and *release*. More precisely, this model states that: (i) all memory operations following an acquire in the program ordering must appear after the acquire in the perform ordering, and (ii) all memory operations occurring prior to a release in program ordering must appear before the release in the perform ordering.

Under release consistency, acquire operations are conveniently mapped to lock synchronization operations, because a lock prevents any future operations from being performed until the lock operation is performed. Release operations are associated with an unlock synchronization operation, which means that when the process leaves the critical section all the previous writes have been performed. Acquire and release operations can also be mapped to other synchronization operations. In particular, barrier operations can be associated with an acquire and a release operation because writes occurring prior to the barrier in program order are expected to be performed before the barrier, and writes occurring after the barrier in program order cannot be performed before the barrier.

4.1.2 Lazy release memory consistency model

When using release consistency, parallel programs must rely on synchronization primitives to properly manage shared data. Consider the code in Figure 8. Release consistency ensures that the write to variable *a* issued by process *A* appears in the performed ordering, as seen by the process *B*, before the *unlock*(*x*). This condition is sufficient to make consistent the read of variable *a* by process *B* but it is not necessary. The necessary condition is that the write issued by *A* appears in the performed ordering seen by *B* before the *lock*(*x*). Thus, the write issued by *A* only needs to be seen by a process *B* if and only if *B* acquires the lock. Thus, although the protocol had non-coherent data copies in different nodes of the system, consistency of that data was still maintained. The memory consistency model that permits this kind of reordering is called the *lazy release memory consistency model* [14].

`a=1;`
`unlock(x);`
	`lock(x)`
	`b=a;`

A code	B code

Fig. 8 Code example

4.2 Multiple writer protocols

The main challenge to tackle when designing multiple writer protocols is how to enable concurrent remote processes to write to a shared page being written by the local processor, without losing local modifications. To prevent this problem, multiple writer protocols work as follows: (i) the OS labels all the shared pages as read-only in order to detect the write operations; (ii) when the underlying virtual memory system detects the first write to a page, it creates a copy of the page before performing the write (this copy is called a *twin*), and marks the page as read-write; (iii) then, if another remote node requires the page, the twin and the page are compared to determine the differences between them. The results of the comparison are stored in a table called *diff*; and, (iv) only the diff is updated to the remote node instead of the whole page. Proceeding in this way, the remote node only receives the updated portion of the page specified by the diff, and its local modifications are preserved.

Diffs enable multiple nodes to write in parallel to the same page and this can reduce write latency. In addition, coherence actions are not applied immediately (as they are in SMP systems), so memory operations can be postponed (as they are in relaxed memory consistency models). This suggests that several coherence actions can be packed in just one message; thus, reducing the total number of coherence action messages.

4.2.1 Eager release consistency protocol

The multiple writer protocol implementation of the release memory consistency model is known as the Eager Release Consistency (ERC) protocol. Typical implementations of this protocol are described in [15] and [16].

In a release consistency model, memory operations perform releases globally. This can be implemented by updating diffs in those nodes sharing the page during the release. This means that the ERC protocol allows for multiple writers by using multicast operations. Thus, the bandwidth need to implement an ERC protocol is heavily dependent on the granularity of sharing. False sharing increases the number of multicasts and fragmentation increases the number of destinations of those multicasts. Additionally, the frequency of sharing also affects the number of multicasts, and this frequency is directly related to the number of release operations occurring. Even given these drawbacks, the ERC protocol has been used in recent pure software SVM implementations [16] because ERC eliminates invalidations and thus reduces the amount of asynchronous communication.

4.2.2 Lazy release consistency protocol

The ERC protocol performs poorly when traffic due to multicasts saturates the available network bandwidth. In this case, it is better to relax the memory consistency model to allow point-to-point messages, instead of multicasting to main coherence. The LRC protocol [17] (which implements the lazy release memory consistency model) requires coherence actions just at nodes that access a semaphore or barrier, instead of requiring multicast traffic on coherence actions, as is done with ERC.

Since these coherence actions are sent to selected nodes, invalidation protocols can be considered since many of the nodes in the network will not be affected by the action. In this context, the invalidation information is called a *write notice*.

When a node invalidates a page due to a write notice, the node will obtain updates from the writers upon a page miss. Because it not possible to know the moment when diffs will be needed, they must be stored until they can be applied to all nodes. Generally, diffs are not stored indefinitely, but are broadcast at the same time that barriers are issued, and are later discarded.

The performance drops in the LRC protocol when there are many page faults, because the faulting node starts asynchronous communication sessions to fetch the corresponding diffs. Asynchronous communication has a high operational cost in SVM systems, because it adds latency to the update fetch. In this context, the granularity of sharing is important because false sharing increases the number of nodes that receive asynchronous requests.

4.2.3 Home lazy release consistency protocol

The HLRC protocol [18] also implements the lazy release memory consistency model. Using this protocol, each page has an associated *home* node that manages all the diffs for that page. Whenever a node writes in a page, the node only has to supply the diffs to the home node. Once supplied, diffs are removed from the writing node. Similar to the LRC protocol, the remaining nodes invalidate the page by applying write notices. When a node encounters a page fault, the OS asks the home node for an updated page. Due to network delays, the needed diffs may not have arrived yet at the home node. In this case, the request is queued until the diffs arrive.

Figure 9 shows how modifications to a given page from the writing node (node *A*) arrive at the home node of the page (node *B*) before the invalidated node (node *C*) asks for an updated copy from the home node (node *B*).

Of course, for a given page, processes running in their home node never perform asynchronous requests related to that page. Therefore, if the home is carefully chosen (e.g., by profiling), asynchronous communication should be reduced. In addition, bandwidth consumption is also reduced because writers only update their home. For this reason, HLRC multiple-writer protocols are the most commonly used protocol in SVM system implementations.

Reducing communication overhead is a key factor to improving SVM performance and one of the most problematic points in SVM design. A important amount of research focusing on this topic [5, 11, 16, 19–22] have addressed reducing the impact of asynchronous communication in SVM systems. The next section discusses asynchronous communication, summarizing key results from recent research in this area.

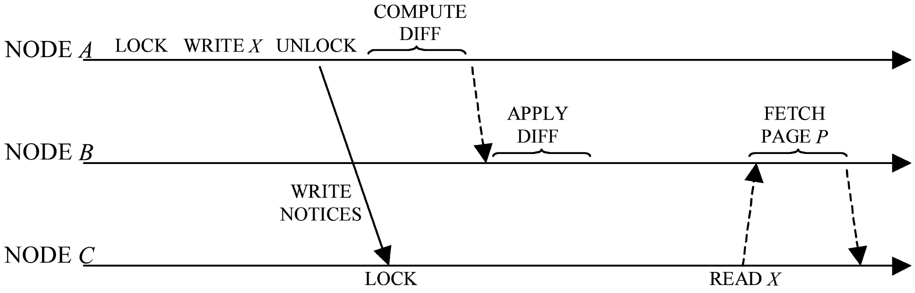

Fig. 9 HLRC protocol example

4.3 Asynchronous communication

In asynchronous communication, a *client* node initiates a request and a *server* node services the request. For example, the client node asynchronously communicates with the server node to read a given page or to lock a given semaphore. This communication involves a context switch in the server node, which implies high service latencies and wastes precious computing time in the server [19].

Several methods have been proposed to reduce overhead associated with asynchronous communication. Some of these mechanisms include hardware support that partially, or totally, avoids this kind of communication [5, 19–21]. Others try to reduce this communication or hide its latency using software techniques [11, 16, 22].

In HLRC protocols, asynchronous communication associated with data requests are a major cause of performance degradation. When a client node tries to access an invalid page, it starts an asynchronous communication with the server to fetch the data. In HLRC-based protocols, the server replies by sending the entire page, while on the LRC-based protocols the server only sends the corresponding diffs.

Software techniques can be used to avoid asynchronous data requests by updating the corresponding data. The Quarks system [16] implements a pure ERC protocol, while other systems implement hybrid protocols that update when particular conditions are present. The Brazos system [22] uses a multicast protocol to update other nodes in the copyset of the page, if a node has a data request for the same page, as well as to update anticipated clients (the set of future clients is predicted ahead of time) before they exit a barrier. Stets et al. [21] measured the performance of a multicast protocol based on a history record.

Hardware techniques can update data utilizing the same techniques as have been implemented in software, as well as perform data requests automatically without processor intervention [5, 19]. In [19], hardware support for an LRC-based protocol that services data requests was proposed. The hardware changes needed to support this protocol in hardware is small since the processor is already interrupted to perform metadata maintenance tasks of related data structures. In [5], the Myrinet NI processor is used to serve pages automatically under the HLRC protocol.

5 Proposed protocols

Section 2.2 analyzed the factors why asynchronous communication arises in SVM systems. We concluded that due to the large granularity used in SVM systems, both false sharing and fragmentation (sharing pattern transformations) are produced. These effects significantly increase the number of asynchronous page requests (a metric directly related to performance).

To address these problems, we will select the Home Lazy Release Consistency (HLRC) protocol as the baseline protocol. One possible way to reduce asynchronous communication is to selectively update data, instead of invalidating the page [7]. The key to achieve good performance is to update data without increasing network traffic. In this section, we describe how this protocol works and study its performance with respect to a baseline protocol and a pure hardware protocol.

5.1 Simulation environment

To evaluate the impact of write updates on performance, we use the LIDE execution-driven simulator [23]. We utilize the same compiler optimization flags as those described in Section 3.

The modeled system consists of a single cluster composed of 32 nodes, connected through an overclocked 1 Gb/s Ethernet network. Network contention is faithfully modeled. Each node contains a single 1 GHz processor.

The load in each node includes both the parallel application plus the operating system overhead introduced by the memory consistency model. Each node has two-cache levels: a direct-mapped 64 KB L1 cache and a direct-mapped 1 MB L2 cache. The hit latencies of the L1 and L2 caches are modeled to be 1 and 8 cycles, respectively, while an access to main memory is 20 cycles. When a page fault or a remote request occurs, the operating system takes 100 μsec to change the context. Before returning to the parallel application, the system checks if there are requests pending from a remote processor. In this case, the system services those requests, each one taking 10 μsec.

Diff creation and application time grows linearly with the page size (4 cycles per word). Since the page size is assumed to be 4 KB and the word size 4 bytes, each protocol takes the same amount of time (4096 nanoseconds) to either create or apply a diff. This overhead is not present when copying a single page, because the model assumes that a DMA device performs this task.

5.1.1 The baseline HLRC protocol

The baseline protocol implementation is based on the description provided in Section 4.2.3. The page homes are selected by means of a modulo function of the most significant bits of the page addresses. Each write notice contains the identification of the writing process, the timestamp of the write, and the page address. There is no garbage collection performed on globally known write notices. Write notices are only sent to a given process, even if the process acquires a semaphore or to all processes when they reach a barrier. Once a process releases a semaphore or a barrier, it sends write notices to the acquirer and diffs to the homes produced by previous writes, in order to keep the home versions updated.

Just as each page has a home, each semaphore and barrier has a home node selected. Home assignment is determined using a modulo function. The semaphore home node queues acquire requests and remembers which node was the last to release the semaphore. This allows a node to forward requests to the last releaser, as needed. Then, the releaser will directly issue write notices to the acquirer node without involving the home node.

Nodes that reach a barrier send write notices to the barrier home, and then are blocked. When the home has received all the barrier requests, the home sends to all pending processes the write notices it has received. Finally, nodes invalidate their copy of the corresponding pages and release the barrier. Barriers are implemented without using the broadcast capabilities of ethernet.

5.2 The HLRC-DU protocol

Unlike the baseline HLRC, in the HLRC-DU protocol some pages are updated instead of being invalidated. Selected pages are updated by using diffs, which are attached to the write notices. We refer to this collective information as *write update*. Those diffs not sent with a write update, are sent later to the page home following the baseline protocol. If during the acquisition of a semaphore, a process receives both a write update and a write notice for the same page, the page is invalidated. Then, if the node has a page fault, it proceeds requesting an up-to-date copy of the page, as in the baseline protocol. Therefore, there is a need to update the home node.

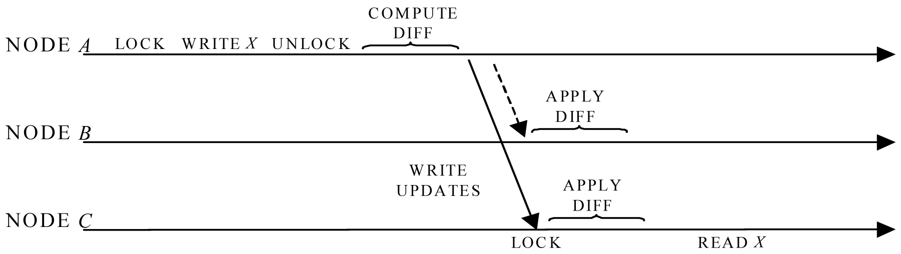

Fig. 10 HLRC-DU protocol example.

Figure 10 shows a working example under the same scenario that the discussed in Section 4.2.3. The example shows how modifications to the writer (node *A*) on a given page are sent both to the node acquiring the semaphore (node *C*) and to the home node (node *B*). Future accesses of node *C* (*READ X*) to this page will not result in a page fault; therefore, this node will not asynchronously request the page home for an up-to-date copy. Thus, if the application requires a large number of write updates, this protocol can significantly reduce the number of both page faults and asynchronous page requests.

The main decision of the protocol is to choose which diffs should be updated. This is a critical decision since different update algorithms tradeoff asynchronous communication versus network utilization. A threshold value acts as a mechanism that can throttle the amount of traffic injected into the network due to write updates. The HLRC-DU protocol detects diffs smaller than this threshold and updates them via write updates. Note that, as the threshold value approaches zero, the protocol performs similarly to the baseline HLRC; on the other hand, as the threshold value rises, the protocol approaches to a pure update protocol.

5.3 Experimental results

To obtain the greatest performance benefits, the HLRC-DU protocol should select to update those diffs based on two key criteria: (1) the overall *update volume* (defined as the size of the diff per unit frequency) should be small enough so that the available network bandwidth does not become saturated, and (2) the update should reduce asynchronous communication. This section analyzes the characteristics of different workloads in an attempt to guide the selection of these two criteria.

5.3.1 Protocol sensitiveness

To consider the first criterion, we obtain the size distribution per processor of the diffs produced by the write notices. Inspecting Table 4, we can see that most diffs are relatively small since, on the average, by about 58% of the diffs contained fewer than 128 words. Thus, in principle, one could think that this value would be a good value; however, as we discuss later, using this value would saturate the network in many cases. Since the algorithm uses write updates to avoid page requests to the home node, the data in Table 4 represents the maximum number of requests that can be avoided.

The threshold value imposes a trade-off between bus utilization and the reduction of asynchronous communication. This means that both issues are strongly dependent on the threshold value. To explore both criteria, we use six different threshold values ranging from 16 to 512 words, besides of no threshold restrictions. Table 5 shows the results, that is, the

Table 4 Distribution of diff sizes

Diff size	Benchmark								Cumulative
range (words)	Barnes	FFT	LU	LU-CONT	Ocean	Radix	Water-NSQ	Water-SP	percentage
]0,1]	3366	1	0	0	12	7	0	3	2%
]1,2]	93	0	0	0	665	7	161	164	3%
]2,4]	14525	0	0	0	21	21	74	3	14%
]4,8]	839	0	32	32	1608	38	95	16	16%
]8,16]	746	0	0	0	1128	99	148	256	18%
]16,32]	1322	0	9215	0	1016	139	9065	2	33%
]32,64]	261	0	0	0	4570	10869	77	560	45%
]64,128]	647	0	5600	0	3345	7886	28	17	58%
]128,256]	303	0	23488	0	3689	2037	70	78	79%
]256,512]	40	0	3002	1585	2166	1373	36	980	86%
]512,1024]	0	3120	0	4912	8760	113	704	0	99%
]1024,2048]	0	0	0	0	1560	0	0	0	100%

Table 5 Percentage of saved home page requests varying the threshold size

		Threshold size (words)						
Benchmark	Requests	16	32	64	128	256	512	∞
Barnes	11539	70%	76%	76%	78%	85%	87%	87%
FFT	26616	0%	0%	0%	0%	0%	0%	30%
LU	39507	0%	0%	2%	14%	69%	82%	82%
LU-CONT	2880	0%	0%	0%	0%	0%	0%	24%
Ocean	35231	7%	7%	9%	16%	32%	45%	90%
Radix	22944	0%	0%	0%	6%	6%	6%	7%
Water-NSQ	4083	5%	40%	41%	41%	42%	44%	74%
Water-SP	7266	6%	6%	13%	13%	13%	72%	72%

percentage of home page requests eliminated by write updates by varying the threshold value. Results in the last column are less than 100% due to including in our numbers the initial home page requests.

In general, larger write updates (see Table 4) reduce the number of page requests among the benchmarks (see Table 5). This is because larger updates are reducing the amount of false sharing, since they leave less space in the page that can be updated by another process. Although larger diffs can greatly reduce the number of page requests, network traffic will increase too. Consequently, we must trade off this reduction for increased network traffic (and increased bus utilization) to improve overall system performance.

5.3.2 Performance results

Figure 11 shows the network utilization in the baseline HLRC model for each benchmark used while varying the threshold size. While in most cases (FFT, LU, LU-CONT, Ocean

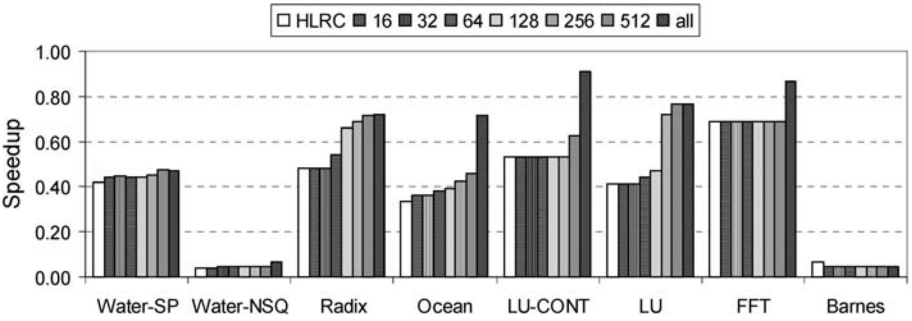

Fig. 11 Network utilization varying the threshold size

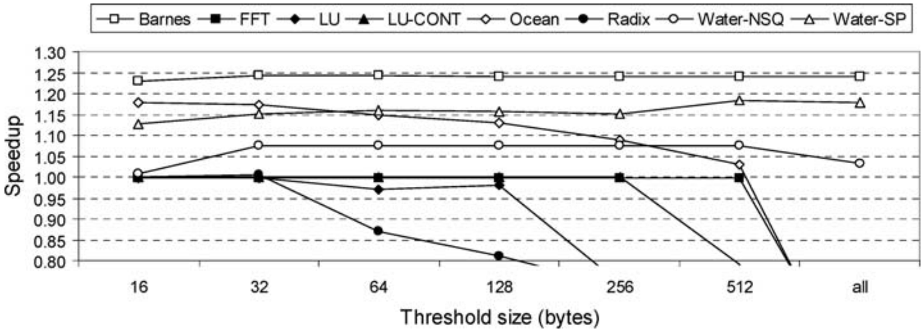

Fig. 12 Speedup relative to the baseline protocol varying the threshold size

and Radix), utilizing a larger threshold value injects write updates that considerably increase network traffic, it is remarkable that no benchmark experiences an increase in network traffic with respect to the baseline protocol when using a small threshold value.

It is unclear from Figure 11 when the network becomes a performance bottleneck. To check the impact of network utilization on performance, we measure the speedup of the proposed protocol over the baseline protocol, varying the threshold size. Figure 12 shows the results. As we can see, the network becomes a bottleneck when utilization reaches 50%. For example, the network becomes a bottleneck in Ocean (with no threshold restrictions) and for Radix (for threshold values higher than 64B). Note that in the case of FFT, network utilization was already an issue for the baseline protocol. For lower utilization values, the speedup primarily depends on the percentage of asynchronous communication saved by the HLRC-DU protocol.

Experiments show that threshold values of 16 to 64 words achieved the best overall speedup, but that the best choice was workload dependent.

5.4 Performance versus hardware techniques

In this section we compare the HLRC-DU performance versus a relevant hardware approach found in the literature [5]. In general, hardware techniques for improving performances of HLRC protocols use specific hardware, or dedicated processors, for avoiding asynchronous communication at the node serving the page. Adopting this approach, pages are served automatically and the home node is uninterrupted. We modeled this feature in our simulator by assuming that the page is served in zero time. Figure 13 presents the speedup of HLRC-DU

Table 6 Speedup relative to the baseline protocol

Benchmark	HLRC-DU 32	Hardware	HLRC-DU 32 + Hardware
Barnes	1.24	1.02	1.25
FFT	1.00	1.03	1.02
LU	1.00	1.08	1.08
LU-CONT	1.00	1.06	1.04
Ocean	1.17	1.03	1.21
Radix	1.01	1.04	1.04
Water-NSQ	1.08	1.01	1.08
Water-SP	1.15	1.05	1.19

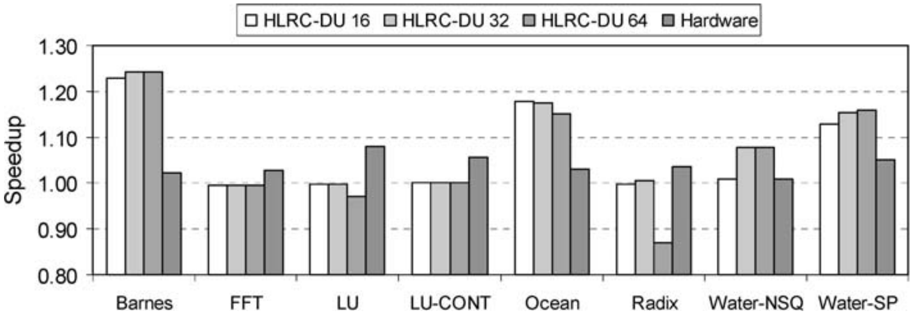

Fig. 13 Speedup of the HLRC-DU protocol using a threshold size of 32 words relative to the baseline protocol

using different threshold sizes (16B, 32B, and 64B) and the baseline protocol with the hardware that automatically serves pages without asynchronously interrupting the processor.

Results show that specific hardware performs better than HLRC-DU in only those cases where HLRC-DU does not obtain benefits relative to the baseline protocol. In all other cases, HLRC-DU outperforms hardware. This occurs because write updates save two interrupts, one at the node accessing the page and the other at the home node. In contrast, the hardware only saves one interrupt at the home node. Although pages are served in zero time, in four of the eight workloads considered (Barnes, Ocean, Water-NSQ and Water-SP), the additional interrupt reduces this performance benefit.

Note that the hardware protocol and the HLRC-DU protocol could be used together effectively. Table 6 shows the results of combining both approaches. In some workloads such as Barnes, Ocean and Water-SP, the HLRC-DU protocol and hardware working together produce a strictly additive performance benefit. In others programs, the advantages of the combined hardware/software approach are negligible. There are only two workloads (FFT and LU-CONT) where HLRC-DU conflicts with the hardware protocol, but in those two cases the degradation was less than 2%.

6 Conclusions

Software-based Shared Virtual Memory systems are quite attractive since they can provide shared memory without the need for special hardware. The overhead associated with the software management of SVM systems introduces extra latency that can adversely impact system performance. One way to mitigate this overhead is to design more efficient SVM consistency protocols.

To address communication overhead, we must obtain a good understanding of how workload exercises elements of the SVM system. In the first part of the paper, we studied a set of parallel workloads, identifying and quantifying those characteristics (sources of performance loss) that impact the performance of applications running on SVM systems. From this characterization, we found that the synchronization rate and the sharing granule are key elements to consider. The main contributions from this part of our study include the identification and quantification of some useful cause-effect relationships, including: (i) how the sharing rate is related to critical section dilation, and (ii) how the sharing granule size affects the sharing pattern, causing sharing pattern transformations. In our analysis, we quantified the severity of critical section dilation and sharing pattern transformation that occurs dynamically.

The workload characterization results motivated our design of the HLRC-DU protocol, which is based on the HLRC protocol. In the proposed protocol, the writer node sends write updates instead of issuing write notices when it detects diffs smaller than an experimental threshold. We studied the impact of using different threshold values, which affect network bottlenecks as well the amount of asynchronous communication.

The HLRC-DU protocol can significantly reduce the number of home page requests, but need to carefully select an appropriate threshold value to achieve the best performance. Results show that in some cases (Barnes and Water-NSQ), performance can improve by more than 40% by selecting the appropriate threshold value. Reducing the number of home page requests is why HLRC-DU can obtain speedups higher than 20% over the HLRC baseline protocol.

We also compared the HLRC-DU protocol to a hardware approach found in the literature [5]. Results showed that, for half of the benchmarks, our protocol performs better than the hardware approach. We also modeled a hybrid protocol that combines HLRC-DU with a hardware mechanism. We found that, on the average, this scheme improves performance by 11% over the baseline protocol.

References

1. Li K, Hudak P (1986) Memory coherence in shared virtual memory systems. In Proceedings of the 5th Annual Symposium on Principles of Distributed Computing
2. Li K (1988) IVY: A shared virtual memory system for parallel computing. In Proceedings of the 1988 International Conference on Parallel Processing
3. Jiang D, Shan H, Singh JP (1997) Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In Proceedings of the 6^{th} Symposium on Principles and Practice of Parallel Programming
4. Iftode L, Singh JP, Li K (1996) Understanding application performance on shared virtual memory. In Proceedings of the 23rd Annual Symposium on Computer Architecture
5. Bilas A, Liao C, Singh JP (1999) Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In Proceedings of the 26th Annual International Symposium on Computer Architecture
6. Zhou Y, Iftode L, Singh JP, Li K, Toonen BR, Schoinas L, Hill MD, Wood DA (1997) Relaxed consistency and coherence granurality in DSM systems: A Performance evaluation. In Proceedings of 6th Symposium on Principles and Practice of Parallel Programming
7. Petit S, Sahuquillo J, Pont A (2001) About the sensitivity of the HLRC_DU protocol to the written area size and page size. In Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software

8. Petit S, Sahuquillo J, Pont A, Kaeli D (2004) Characterizing the dynamic behavior of workload execution in SVM systems. In Proceedings of the 16-th Symposium on Computer Architecture and High Performance Computing
9. Davis H, Goldschmidt SR, Henessy J (1991) Multiprocessor simulation and tracing using tango. In Proceedings of the 1991 Conference on Parallel Processing
10. Woo S, Ohara M, Torrie E, Pal Singh J, Gupta A (1995) The SPLASH-2 programs: Characterization and methodological considerations. In Proceedings of the 21st Annual International Symposium on Computer Architecture
11. Bilas A, Singh JP (1997) The effects of communication parameters on end performance of shared virtual memory clusters. In Proceedings of the Supercomputing '97 Conference
12. Iftode L, Singh JP (1999) Shared virtual memory: Progress and challenges. In Proceedings of the IEEE, 87(3)
13. Gharachorloo K, Lenoski D, Laudon J, Gibbons P, Gupta A, Henessy J (1990) Memory consistency and event ordering in scalable shared-memory multiprocessors. In Proceedings of the 17th Annual Symposium on Computer Architecture
14. Keleher P (1995) Lazy release consistency for distributed shared memory, Ph.D. Thesis, Rice University
15. Carter JB, Bennet JK, Zwaenepoel W (1991) Implementation and performance of Munin. In Proceedings of the 13th Symposium on Operating Systems Principles
16. Swanson M, Stoller L, Carter JB (1998) Making distributed shared memory simple, yet efficient. In Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments
17. Keleher P, Cox AL, Dwarkadas S, Zwaenepoel W (1994) TreadMarks: Distributed shared memory on standard workstations and operating systems. In Proceedings of the Winter 1994 USENIX Conference
18. Zhou Y, Iftode L, Li K (1996) Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation
19. Bianchini R, Kontothanassis LI, Pinto R, De Maria M, Abud M, Amorim CL (1996) Hiding communication latency and coherence overhead in software DSMs. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems
20. Blumrich MA, Alpert RD, Bilas A, Chen Y, Clark DW, Damianakis S, Dubnicki C, Felten EW, Iftode L, Li K, Martonosi M, Shillner RA (1998) Design choices in the SHRIMP system: An empirical study. In Proceedings of the 25th Annual Symposium on Computer Architecture
21. Stets R, Dwarkadas S, Komothanassis L, Rencuzogullari U, Scott ML (2000) The effect of network total order, broadcast and remote-write capability on network-based shared memory computing. In Proceedings of the 6th Symposium on High-Performance Computer Architecture
22. Speight E, Bennett J (1998) Using multicast and multithreading to reduce communication in software DSM systems. In Proceedings of the 4th Symposium on High-Performance Computer Architecture
23. Petit S, Gil JA, Sahuquillo J, Pont A (2000) LIDE: A simulation environment for shared virtual memory systems. ACM Computer News 28(4)
24. Petit S, Sahuquillo J, Pont A (2005) A comparison study of the HLRC-DU protocol versus a HLRC hardware assisted protocol. In P roceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing