DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

UNIVERSIDAD POLITECNICA DE VALENCIA

Ph.D. Thesis

Efficient Home-Based Protocols for Reducing Asynchronous Communication in Shared Virtual Memory Systems

Salvador Petit Martí

Advisors:

Dr. Julio Sahuquillo Borrás

Dr. Ana Pont Sanjuán

Valencia, Spain. February 2003.

Approved, Thesis Committee:

A Montse

Agradecimientos

Sin el apoyo y la ayuda de muchas otras personas, éste trabajo no hubiera sido posible. Esta página va dedicada a todas ellas.

En primer lugar, a mis directores de tesis, Ana Pont y Julio Sahuquillo, por su dedicación inestimable en todos estos años. Sin la motivación que Ana supo insuflarme, y el optimismo y seguridad expresados por Julio, seguramente yo no habría sido capaz de terminar este trabajo.

A los profesores Veljko Milutinovic y David Kaeli, por su ayuda en diversas etapas de esta investigación. Veljko me ayudó durante mi estancia en Belgrado en los comienzos de este trabajo, ofreciéndome su hospitalidad, y aportando excelentes ideas a lo largo de estos años. David ha sido una magnifica ayuda técnica y un importante aporte como persona a este trabajo en el último año.

Agradezco al Área de Coordinación de Lenguas Extranjeras de la Universidad Politécnica de Valencia por su ayuda en la traducción de algunas partes de este trabajo.

A mis compañeros y amigos en el trabajo: Silvia, Vicent, Patricia, Marian, Raúl, José Luis, Josep, Pau, Juan Carlos, Juan Luis, Ismael, Sergio y Alberto. Siempre han estado ahí cuando he necesitado su ayuda o su compañía.

Mas allá de la vida laboral, he tenido la suerte de contar con magníficas personas como mis amigos, cada una de las cuales con cualidades personales e irrepetibles, con las que he disfrutado, sufrido y sobre todo, crecido durante un montón de años. Perdonadme, no tengo espacio aquí para nombraros a todos, pero sabéis que os tengo siempre presentes.

Y finalmente, a mis padres, Vicenta y Salvador, y a mi hermano Vicente, sin olvidar a todo el resto de mi familia, por su paciencia, aguante e interés por mi trabajo durante todo este tiempo.

Resumen

En la presente tesis se realiza una evaluación exhaustiva de los Sistemas de Memoria Distribuida conocidos como Sistemas de Memoria Virtual Compartida. Este tipo de sistemas posee características que los hacen especialmente atractivos, como son su relativo bajo costo, alta portabilidad y paradigma de programación de memoria compartida.

La evaluación consta de dos partes. En la primera se detallan las bases de diseño y el estado del arte de la investigación sobre este tipo de sistemas. En la segunda, se estudia el comportamiento de un conjunto representativo de cargas paralelas respecto a tres ejes de caracterización estrechamente relacionados con las prestaciones en estos sistemas. Mientras que la primera parte apunta la hipótesis de que la comunicación asíncrona es una de las principales causas de pérdida de prestaciones en los Sistemas de Memoria Virtual Compartida, la segunda no sólo la confirma, sino que ofrece un detallado análisis de las cargas del que se obtiene información sobre la potencial comunicación asíncrona atendiendo a diferentes parámetros del sistema.

El resultado de la evaluación se utiliza para proponer dos nuevos protocolos para el funcionamiento de estos sistemas que utiliza un mínimo de recursos hardware, alcanzando prestaciones similares e incluso superiores en algunos casos a sistemas que utilizan circuitos hardware de propósito específico para reducir la comunicación asíncrona. En particular, uno de los protocolos propuestos es comparado con una reconocida técnica hardware para reducir la comunicación asíncrona, obteniendo resultados satisfactorios y complementarios a la técnica comparada. Todos los modelos y técnicas usados en este trabajo han sido implementados y evaluados utilizando una nuevo entorno de simulación desarrollado en el contexto de este trabajo.

Resum

En la present tesi, es realitza una avaluació dels Sistemes de Memòria Distribuïda coneguts com Sistemes de Memòria Virtual Compartida. Este tipus de sistemes posseeix característiques que els fan especialment atractius, com són el seu relatiu baix cost, alta portabilitat i paradigma de programació de memòria compartida.

L'avaluació consta de dues parts. En la primera es detallen les bases de disseny i l'estat de l'art de la investigació sobre este tipus de sistemes. En la segona, s'estudia el comportament d'un conjunt representatiu de càrregues paral·leles respecte a tres eixos de caracterització estretament relacionats amb les prestacions en estos sistemes. Mentre que la primera part apunta la hipòtesi que la comunicació asíncrona és una de les principals causes de perduda de prestacions en els Sistemes de Memòria Virtual Compartida, la segona no sols la confirma, sinó que ofereix una detallada anàlisi de les càrregues de què s'obté informació sobre la potencial comunicació asíncrona atenent a diferents paràmetres del sistema.

El resultat de l'avaluació s'utilitza per a proposar dos nous protocols per al funcionament d'estos sistemes que utilitzen un mínim de recursos hardware, aconseguint prestacions semblants i superiors en alguns casos a sistemes que fan us de hardware de propòsit específic per a reduir la comunicació asíncrona. En particular un dels protocols proposts és comparat amb una reconeguda tècnica hardware per a reduir la comunicació asíncrona, obtenint resultats satisfactoris i complementaris a la tècnica comparada. Tots els models i tècniques usats en este treball han sigut avaluats utilitzant un nou entorn de simulació desenvolupat en el context d'este treball.

Abstract

In this thesis, an exhaustive evaluation of Distributed Shared Memory Systems known as Shared Virtual Memory Systems is performed. This kind of systems has characteristics that made them specially attractive, like their relatively low cost, high portability and shared memory programming paradigm.

The evaluation is performed in two parts. In the first part, the design principles and the state of the art of the research related with this kind of systems is performed. In the second part, it is studied the behavior of a representative set of parallel workloads regarding to three axes of characterization intimately related with the performance of this kind of systems. While the first part points to the hypothesis that asynchronous communication is one of the main causes of performance loss, the second does not only confirm it, but also offers a detailed analysis of the workloads that shows useful information about the potential asynchronous communication attending to different system parameters.

The evaluation results are used to propose two new protocols for this kind of systems that uses minimal hardware resources, reaching similar and in some cases superior performance to that obtained by systems that make use of specific hardware for reducing asynchronous communication. In particular, one of the proposed protocols is compared with a well-known hardware technique for reducing asynchronous communication, obtaining satisfactory and complementary results to the compared technique. All the modeled systems and techniques used in this work have been implemented and evaluated using a new simulation environment developed in the context of this work.

Contents

Figures

Tables

Chapter 1

Introduction

Parallel workloads executed on distributed systems, multiprocessors or multicomputers, are usually based on two distributed programming paradigms: message passing and shared memory. In the first paradigm, parallel processes have separate memory address spaces and they communicate with the other processes explicitly by means of message passing. In the second paradigm, the processes share (partially or totally) the memory address space and, in a dynamic and transparent way for the programmer, write actions on the shared memory from a process are perceived later by other parallel processes.

Traditionally, message passing distributed programming has been used to parallelize and execute workloads in low cost distributed computer environments, such as those made up by Networks of Workstations (NOW) [AND95]. The main reasons for this phenomenon are:

- There are standard libraries such as PVM [SUN90] and MPI [HAN98] that have been developed for different operating systems and hardware, assuring platform independence for the workload execution.

- The message-passing paradigm can be adapted to almost any environment because the interface it offers to the parallel workload is independent of the supporting hardware, which can be even a heterogeneous NOW.

- As the interface is independent of any specific hardware, system maintenance is facilitated because it becomes open and independent of the manufacturer (as long as it follows a standard, such as Ethernet). This reduces costs since the nodes and the interconnection network can be easily replaced and upgraded.

- Communication between processes is mainly carried out by software, with practically no hardware restrictions (any network is valid, even those which are non-specifically designed for parallel computing).

On the other hand, the main disadvantages of the message-passing paradigm are:

- It is much less intuitive for the programmer than the shared memory paradigm, so to parallelize code is more difficult than in Shared Memory Multiprocessor (SMP) systems [CUL99] or Distributed Shared Memory (DSM) systems [PRO98].

- A great amount of the code that is currently executed in monoprocessor systems may easily be parallelized by means of the shared programming paradigm; for example, the code in multimedia software, databases, and the Internet. This is because several threads (multithread), which share the same memory address space, carry out the execution.

The parallelization of the execution of shared memory workloads has been carried out by means of a hardware specifically designed to allow this type of programming. Usually, SMP or DSM systems are used. SMP systems are inexpensive but they lack scalability (only less than 16 nodes are feasible) as a consequence of their use of a shared medium for communication, which is not easily expandable because it saturates quickly, so becoming a bottleneck. DSM systems allow higher scalability and are more easily expandable, but they require a higher cost in hardware and design.

1.1. Software Distributed Shared Memory

Software Distributed Shared Memory (SDSM) systems [LI_86] are an inexpensive alternative, scalable above SMP systems, easier to maintain, and open. This kind of systems can be implemented by means of two methods:

- Supported by the programming language or by extensions of the language in which the workload is programmed. The compiler or the preprocessor generates the necessary code to establish the communication. This assumes that the programmer should somehow mark the memory zones or objects to be shared.

- Supported by the operating system: The Operating System (OS) detects write operations in memory zones and carries out the communication. Typically, virtual memory mechanisms, which are present in all modern operating systems, are used for this case. This scheme is transparent for the programmer.

This work is aimed at the study of the latter type of SDSM systems, the Shared Virtual Memory (SVM) systems.

Li and Hudak suggested first the SVM system concept in [LI_86], and their implementation details were published in [LI_88]. Four main features define an SVM system:

a) Nodes share a common virtual memory address space, by using the virtual memory system provided by the supporting OS.

b) The page is the sharing unit.

c) The supporting software (OS, libraries, etc.) takes charge of guaranteeing coherence maintenance of the shared pages (when necessary).

d) The parallel workload is independent of the interconnection network and the hardware supporting it.

These features make SVM systems especially attractive because they allow the use of shared memory code without modifications, allowing its execution in heterogeneous and decoupled networks.

In general, SVM systems are usually composed of several inexpensive nodes (single processors or SMP systems) connected by a commodity network. In addition, SVM systems are cheaper than other alternatives. As in the case of NOWs that use the message passing paradigm, this approach enables fault tolerance and offers good flexibility when maintaining and upgrading the processing nodes of the system, since these are physically independent.

On the other hand, these features become the source of the main problems in these systems:

- False sharing: Since the page size is usually large (4 or 8 Kbytes), the probability of false sharing rises, so increasing the communication traffic. The effect of false sharing is that coherence actions are performed between nodes that are not sharing data. False sharing may have adverse effects for the system such as the ping-pong effect of pages among writers [TOR94].

- High latency: Usually, coherence messages are triggered by page faults detected by the OS, which uses the appropriate software to inject them in the interconnection network. Therefore, their latency penalties are usually very high.

Both problems are related, since false sharing produces additional messages to maintain the coherence, which introduce their own latency affecting the whole system performance.

1.2. Relaxed Memory Consistency Models

To lessen these problems, much research has focused on relaxed memory consistency models [IFT99]. Memory consistency models specify when a memory reference can be carried out and become visible to the memory system, so that the rest of network nodes can see it. Depending on whether the model is more or less restrictive, better or worse performances will be achieved. The most restrictive model is called sequential and it is simpler to implement, but it offers the worst performance. To improve performance, research has focused on reducing the restrictions in order to increase performance. The answer has been the release memory consistency models, which allow reordering of memory references according to certain rules specified by the model. In SVM systems, the release memory consistency models most frequently used are Release Consistency (RC) model [GHA90] and Lazy Release Consistency (LRC) model [KEL95].

Most parallel workloads use synchronization methods when several processes access shared data. In this chapter we will assume that the primitives used are semaphores, as semaphores can implement any synchronization primitive.

The main idea behind release memory consistency models is that if a parallel workload is correctly programmed, it must be exempt from race conditions [ADV93]. A race condition occurs when a possible execution of the program may allow a write access and another access at the same time (read or write). If race conditions are not allowed, writing accesses to shared data will be performed serially using any synchronization primitive (for instance, semaphores). This implies that it is only necessary to send coherence messages when a given process leaves a section protected by a semaphore.

When these models were introduced, new techniques were added to allow multiple writers on a page. As race conditions are not allowed, writes carried out at the same time by different writers refer to different variables. Therefore, multiple writers carry out modifications at the same time on different addresses in the page. These techniques face the problems derived from false sharing, as there is no conflict between writers accessing the same page.

The implementation of a given memory consistency model (with single or multiple writers) is called memory consistency protocol. The first memory consistency protocol implementing the RC model with multiple writers was the Eager Release Consistency (ERC) protocol [GHA90] and it was implemented on the Munin system [CAR91]. The model carries out the coherence actions when the semaphore is released. Its advantage is that it reduces the number of coherence messages by delaying the coherence actions (several coherence actions are compressed in one message). In addition, the software overhead is reduced (only one message is sent) and the probability of coherence actions due to false sharing is also reduced. A more recent implementation can be found in the Quarks system [SWA98].

Many of the messages that the ERC protocol sends are unnecessary. As we will see in Chapter 2, only the next process accessing the semaphore needs to apply coherence actions before entering the semaphore. In fact, it is not necessary to carry out the coherence actions until the next process accessing the semaphore is known. The Lazy Release Consistency (LRC) [KEL95] model exploits this idea. The first protocol implementing it was proposed in the TreadMarks system [KEL94]. Other implementation can be found in [BIA96].

The most frequently used memory consistency protocol that implements the LRC model is called Home Lazy Release Consistency (HLRC) [ZHO96] protocol. The main difference between the Treadmarks LRC protocol and the HLRC protocol is that in the latter there is a home node for each page, which concentrates the modifications. In this way, when a process needs a copy of the updated page it only interrupts the home node (in the Treadmarks LRC protocol, multiple nodes can be interrupted). In addition, if the home node is chosen carefully it is possible to reduce the number of page faults, as it is continuously updated. Because of these advantages, some recent systems have implemented the HLRC protocol [STE00][BIL98], and for this reason it is used as the baseline protocol in this dissertation.

However, SVM systems are far from obtaining performances close to those reached by hardware based shared memory systems. The main reason is that the software characteristics of SVM systems interact adversely with the parallel workloads, thus reducing the performance [IFT96][JIA97][ZHO97]. This is because parallel workloads are usually optimized for hardware systems. In general, there is a performance loss if the workload has frequent synchronizations and the granularity of the shared data is small. Release memory consistency protocols generate coherence actions as a consequence of synchronizations and, as it has been already said, the coherence messages have high latencies in SVM systems. On the other hand, if the granularity of the shared data is small, false sharing and fragmentation occur and the number of required coherence actions increases. The HLRC protocol mitigates these problems, but introduces new ones:

- Readers interrupt writers asynchronously to update their data. The asynchronous communication is a critical factor in the performance of current SVM systems due to its high cost.

- The complexity of the release memory consistency protocols adds computing time to each message, thus increasing its latency.

- Release memory consistency protocols tend to arrange the coherence actions at synchronization points, causing contention points during the execution.

1.3. Thesis Overview

This thesis focuses mainly on the problems of reducing both asynchronous communication traffic and latency. To mitigate these problems so improve the system performance, new and efficient protocols are suggested. The proposed protocols are based on the understanding of the main characteristics of the workload at runtime. The main contributions of the thesis are:

- Characterization of several parallel workloads from the SPLASH-2 suite [WOO95] in those aspects that can negatively impact the performance of SVM systems. The characterization quantifies the sources of performance loss by measuring three axes that are related to latency in asynchronous communication: frequency of sharing, granularity of sharing, and entropy in sharing patterns. The results illustrate the impact of the sharing granule size, quantifying the relationship between page size and fragmentation/false sharing. The effects of sampling across fixed intervals are also studied, showing how many applications exhibit distinct phases during execution. Some of the results found in those studies have been published in [PET02]. Others have been submitted and are pending revision [PET03].

- Implementation of a new simulation environment for SVM systems. The developed tool is an execution-driven simulator [PET00] aimed at studying the behavior of memory consistency models. This tool can take as input any of the SPLASH-2 benchmark suites or can use the real workloads. It simulates the detailed behavior of these systems, varying both memory consistency models and the local area network configuration.

- Design of two new SVM memory consistency protocols (HLRC-CU and HLRC-DU protocols) that use the results of the SPLASH-2 characterization to improve the baseline HLRC protocol. The HLRC-DU protocol is a pure software protocol while the HLRC-CU protocol uses a specific hardware table. Both protocols use a specific message (referred to as write update) to update written data, reducing asynchronous communication. The HLRC-CU protocol also reduces latency caused by multiple writer protocol overhead by focusing on those write operations that perform over small continuous areas. The behavior of both protocols is also characterized in function of the maximum write update size. The characteristics of both protocols and studies of their performance metrics have been published in [PET01][PET01b].

This thesis has been structured in chapters as follows: Chapter 2 introduces SVM systems and describes an important subset of the memory consistency models and protocols found in the open literature. It also describes the latest contributions from recent research by explaining asynchronous communication design and implementation in SVM systems. Chapter 3 describes the LIDE simulation environment for SVM systems. Chapter 4 characterizes SPLASH-2 parallel workloads from the SVM point of view and discusses the main sources of performance loss in current SVM systems. Chapter 5 proposes two new consistency protocols designed from the results obtained in Chapter 4. The performance of the proposed protocols is compared with other classical solutions. Finally, Chapter 6 presents the most relevant conclusions of the thesis and the open research lines for future works.

Chapter 2

Shared Virtual Memory Systems

As we introduced in Chapter 1, SVM systems allow shared memory programming at a low design and maintenance cost due to their software implementation; nevertheless, as hardware implementations work faster, their performance are still far from that achieved by hardware based distributed shared memory (DSM) systems. Nowadays, SVM systems use relaxed memory consistency models and multiple writer protocols as techniques to reduce latencies and false sharing respectively; however, these techniques induce additional overheads that reduce performance. The four main characteristics that define the SVM systems are:

- Shared virtual address space: The processes access the same memory areas through logic memory addressing. This virtual address space is split up into pages. This relies on the same mechanism that allows several processes to share pages in single processor systems. However, in an SVM system each node has its own local physical memory. In other words, different nodes do not map their virtual addresses to the same physical memory. Thus, the OS needs to maintain the memory coherence among the local memories of the nodes.

- Page as sharing unit: The sharing unit of the system is the virtual memory page. Usually, the most commonly page size is used is 4 KB.

- Software maintained coherence: The OS updates or invalidates the non-coherent pages. Thus, coherent actions are usually performed by software.

- Heterogeneous interconnection network: Coherence actions can be performed using message passing because the OS performs the coherence actions. This allows a high degree of independence from the supporting hardware, as happens in message passing parallel systems. In other words, the shared memory programming paradigm is accomplished by a message passing architecture.

In this chapter a general overview of the SVM systems is presented. The remainder of the chapter is organized as follows. Section 2.1 describes a simple SVM system as an example to illustrate the characteristics mentioned and how they affect the design and implementation of this kind of systems, section 2.2 explains memory consistency models, in particular relaxed memory consistency models, section 2.3 discusses the memory consistency protocols implementing relaxed memory consistency models, section 2.4 details asynchronous communication in SVM systems. Finally, section 2.5 concludes with some remarks linking the different concepts introduced in this chapter.

2.1. A Simple SVM System Example

We use a simple example to show the main characteristics and the software nature of SVM systems. For the sake of simplicity, we suppose that the interconnection network is a bus for two main reasons: i) it is the simplest network topology and, ii) each node connected to the bus can snoop all the traffic on the bus, so monitoring all the transmitted messages.

Let us assume that the pages in memory can be in three different states:

- *Invalid*: Any access to the page sends an interrupt to the local OS of the node. It is used for pages whose content has been invalidated by a previous writer.

- *Read-only*: Any write access to the page sends an interrupt to the local OS of the node. It is used in order to detect writings to the page.

- *Read-write*: Reads or writes to the page can be performed without interrupts. This happens when a process of the node has performed at least one write to the page.

We assume that each node connected to the interconnection network can host one or several processes. As processes in a node share the same physical memory (for example, an SMP system), page states are set per node, instead of per process. This reduces memory overhead for storing page tables and allows local modifications in a node to be seen by all the local processes of the node without need of SVM coherence actions. An implementation of this technique can be found in [SAM98].

Whenever an interrupt is produced, the local OS of the interrupted node takes control and performs the corresponding coherence actions:

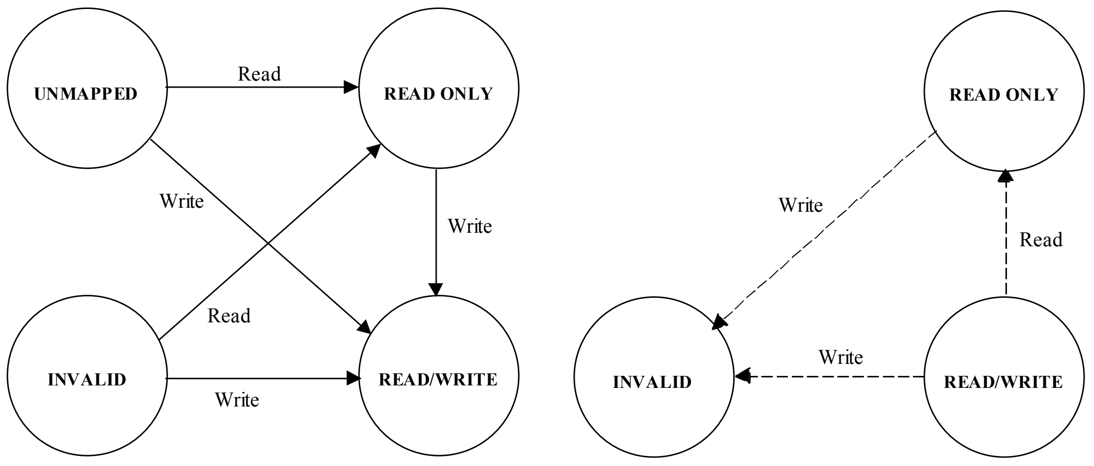
- If the page is invalid, it performs the necessary correspondent actions to make it coherent.
- If the page is read-only, the writing is detected, and an invalidation is generated for other remote processes sharing the page.

Initially, only the process 0 owns a copy of all the shared pages in read-write access mode. Let us assume that this process is running in node 0. All the other nodes do not have a mapped copy of the shared pages.

In the initial scenario, if a process in a node other than 0 (we can call it remote node) tries to read a page, the access will result in an unmapped page and it will generate an interrupt due to a page fault. Then, the local OS requires a copy of the page to the node that has the page in read-write access mode (in this case, node 0). When the update is accomplished, both node 0 and the node of the reader process will have a read-only copy of the page. This state is maintained in all the copies of the page while there are only reader processes.

When a remote process tries to write a shared page, it will have a page miss and will produce an interrupt. Then, the local OS requires a copy of the page to the node that has the page in the read-write access mode (in this case, node 0), then invalidates all the other node copies, and sets its copy to a read-write state. This node will serve future remote accesses to that page. Note that both nodes that have an invalid copy of the page and the node that has the page in the read-write state are known by all the nodes, because we assume that all the nodes monitor the messages transmitted through the bus. In systems that do not broadcast the coherence actions, a directory of nodes sharing the page is needed. This directory can be maintained centralized or distributed.

Figure 1 shows the state transition diagram. This graphic summarizes how local and remote pages change their state due to read and write accesses to the page. The continuous arrows on the left represent state transitions due to local accesses; the dotted arrows on the right represent state transitions due to remote accesses. For example, a read-write to read-only transition is triggered by a remote read, and a local write triggers a read-only to a read-write transition, invalidating remote copies of the page.



a) Transitions due to local accesses **b) Transitions due to remote accesses**

Figure 1 – State transition diagram of a page

Figure 2 shows the handler code of both page faults and remote message requests. Each page has an associated lock that assures atomicity for the code executed by the handlers between the processes running in a given node.

If a given process in a node *N* has a read page fault on page *P*, its OS will execute the *read fault handler*. This handler sends a message to the node that has a read-write copy of the page *P* (*rw_node*). On receiving this message, the *rw_node* executes the *read message handler*, which returns an up-to-date copy of the page *P* to node *N* and sets its copy to the read-only state. The node *N* receives the up-to-date copy of page *P* and sets it to the read-only state.

```
Read fault handler:
        lock(page);
        send(page.rw_node, READ, page.address);
        recv(page.address);
        page.state = READ_ONLY;
        unlock(page);

Read message handler:
        lock(page);
        reply(page.address);
        page.state = READ_ONLY;
        unlock(page);

Write fault handler:
        lock(page);
        send(ALL, WRITE, page.address);
        recv(page.address);
        page.state = READ_WRITE;
        page.rw_node = this_node;
        unlock(page);

Write message handler:
        lock(page);
        if (page.rw_node == this_node) then reply(page.address);
        page.state = INVALID;
        page.rw_node = sender_node;
        unlock(page);
```

Figure 2 – OS interrupt handlers

If a given process in a node *N* has a write page fault on page *P*, its OS will execute the *write fault handler*. This handler broadcasts a message to all the nodes in the system, including the node that has a read-write copy of the page *P* (*rw_node*). On receiving this message, all nodes execute the *write message handler*, which sets their copies of the page *P* to the invalid state. Only the *rw_node* returns an up-to-date copy of the page *P* to node *N*, which receives the copy and sets it to read-write state. Finally, all the nodes set the *rw_node* of the page *P* to the node *N*.

From this example it is noticeable that the software nature of the protocol handlers (executed by the OS) and the large size of the coherence (a page) are the root of the main performance drawbacks that a simple system like the above described will suffer:

The example behaves similarly to an SMP snoopy cache invalidation protocol. This means that it will send a message each time a process in a node writes to a read-only page. As message latency is much higher in SVM systems than in SMP systems due to their software nature and the commodity network, the performance will decrease enormously. In addition, as a consequence of the big size of pages, the number of messages sent due to false sharing increases.

To solve these problems, the design and implementation focused on new relaxed memory consistency models, which are detailed in the next section.

2.2. Memory Consistency Models

Parallel programmers wish a shared memory system behavior that is formally defined by the memory consistency models (regardless of whether the system architecture is centralized or distributed). Intuitively, programmers assume the sequential memory consistency model (see section 2.2.2), which disables some optimizations that improve SVM performance. Although the first SVM proposal followed the sequential consistency model [LI_86], it is not used in current implementations. The SVM systems focus on relaxed memory consistency models [IFT99] because these models enable to delay coherence actions, thus reducing the number of coherence messages, and so saving latency. In addition, these models allow the design of multiple writer protocols, which enormously reduce the number of messages sent due to false sharing.

There are two main concepts related with consistency models: the performing order and the fence. Both are described in the next following sections.

2.2.1. Performing Order

Memory consistency models define the order in which memory operations from one process can perform with regard to other processes.

Formally:

- A memory operation issued by a process *i performs* with regard to a process *j* when the result of the memory operation is visible by the process *j*.

- The *performing order* of a process *i* with regard to a process *j* is the order in which memory operations of process *i* perform with regard to process *j*.

Depending on the memory system, the performing order can be unique or dependent on *j*. For instance, in an SMP system all the processes see the same performing order because they all read the same information from the bus, which serializes shared memory operations. However, in a DSM system, different orders can be seen at different points of the interconnection network.

In a monoprocessor system, the performing order of memory operations is quite flexible and it can be easily reordered when it affects different addresses without violating the program semantics. This enables improvements in performance, for example, by adding write buffers and/or caches. However, reordering is less flexible when a parallel program is running in several processors, because it can violate the program semantics. Figure 3 shows an example of a code running in two different processors (*A* and *B*). The two instructions of processor *A* write different variables, so processor *A* does not need to monitor which instruction is performed first if the code is only running in one processor (there are not data dependencies). However, if this code is a fragment of a parallel program with the code of processor *B*, the order is important because the value assigned to the variable *b* in processor *B* depends on the *performing order* of processor *A* with regard to processor *B*.

`a=1;`
`x=1;`
	`while (x==0) {};`
	`b=a;`

A code	B code

Figure 3 – Code example running in two different processors

This example shows that to guarantee parallel program semantics, memory systems need to restrict the order in which memory operations from a process can perform with regard to other processes. We define a *fence* as a new operation in the program flow that delimits the begin and/or the end of a possible area of reordering. In particular, we can distinguish two types, as shown in Figure 4:

- *Begin Fence* (F_b): Ensures that all the operations issued by a process *i* that are located after the fence in program order, perform after the fence with regard to any process *j*.
- *End Fence* (F_e): Ensures that all the operations issued by a process *i* that are located before the fence in program order, perform before the fence with regard to any process *j*.

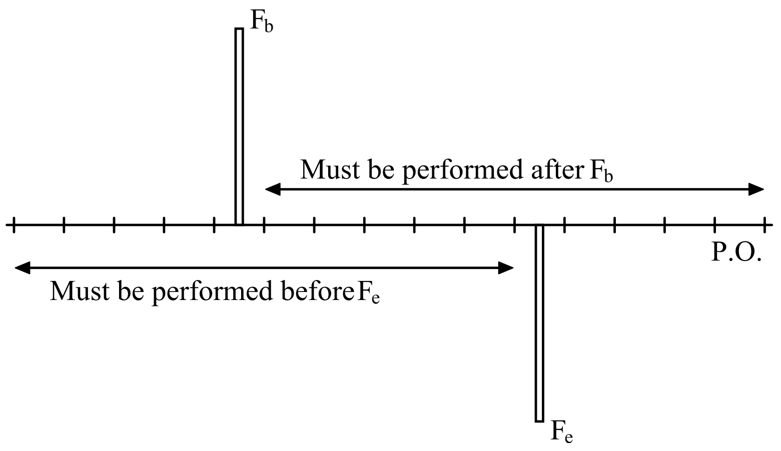


Figure 4 – Begin fence (F_b) and end fence (F_e)

Both types are not exclusive so the same point may act both as begin and end fence. We refer to such points as *total fence* or simply a *fence*.

Memory consistency models differ depending on what instructions set a fence in the parallel program execution and what is the type of that fence. Below, we describe the memory consistency models, from the most to the least restrictive.

2.2.2. Sequential Memory Consistency Model

Prohibiting changes in the performing order is the most straightforward manner to prevent semantic problems caused by reordering. This is accomplished by considering each memory operation as a total fence. This defines the *sequential memory consistency model* [LI_86] where memory operations issued from a process perform with regard to any other processes in program order.

Formally, the condition for sequential consistency is:

- A read, write, or synchronization operation can perform with regard to any other process if all previous reads, writes and synchronization operations have already been performed.

Figure 5(a) describes a possible sequence of memory operations issued by a parallel process. Each memory operation is issued to a different, unrelated memory address (*a, b, c, d, e, f, x,* and *y*). Under the sequential memory consistency model, reordering is not allowed and so the performing order matches the program order.

The main drawback of the sequential model is that it does not allow reordering. This implies that each memory operation has to wait that the previous instructions in program order have performed. In SVM systems, this restriction would force the transmission of a coherence message through the network each time a shared memory operation is issued, dramatically reducing the performance. Thus, the sequential memory consistency model is not implemented and some degree of reordering is allowed.

By allowing reordering the system cannot assure that the performing order matches the program order. Consequently, the semantics of a parallel program cannot depend on the program order of the memory operations. To solve this problem, the programmer uses synchronization operations (i.e., locks, unlocks and barriers) to impose order restrictions.

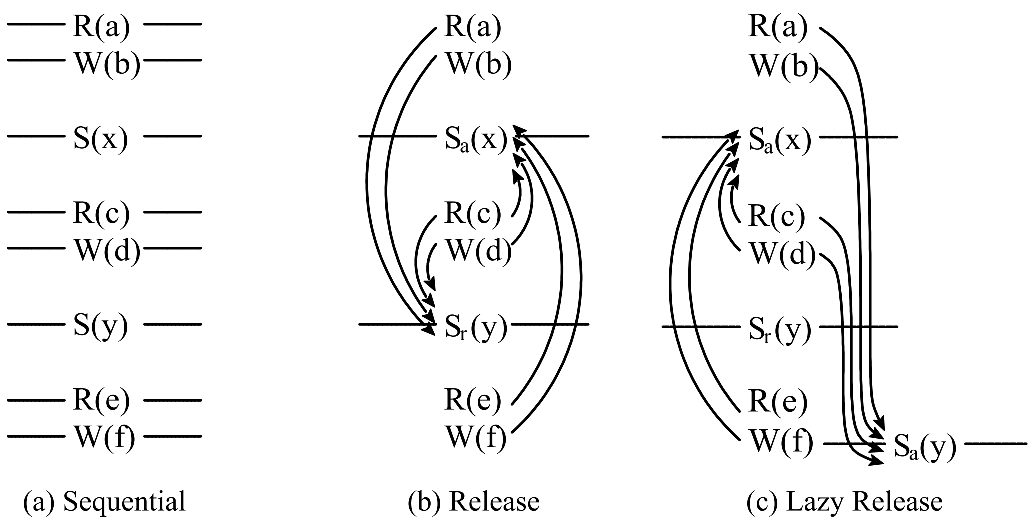


Figure 5 – Fences in memory consistency models

It can be shown that if the parallel workload is assumed to be *data-race-free* [ADV93] and *correctly labeled* [GHA90] by synchronization operations, the performing of the memory operations can be postponed to the next synchronization point in program order.

For example, we can specify that the synchronization operations execute in program-order, although memory operations located between such synchronization operations can be reordered. The systems working in this way follow relaxed memory consistency models [IFT99]. We can consider several models according to the level of relaxation. Below, the release and lazy release memory consistency models, which are the most commonly used in SVM systems, are described.

2.2.3. Release Memory Consistency Model

The *release memory consistency model* [GHA90] allows more reordering than the sequential model by using two types of synchronization operations: *acquire* and *release*. The former behave as begin fences, and the latter as end fences. In this manner, all the memory operations following an acquire operation in program order must perform after the acquire operation, and all the memory operations prior to a release operation in program order must perform before the release operation.

Under release consistency, acquire operations can be conveniently mapped to the lock synchronization operation because a lock precludes the following operations being performed before the lock. Release operations are associated with the 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 writings prior to the barrier in program order are expected to perform before the barrier, and writings following the barrier in program order cannot perform before the barrier. In this manner, barriers still behave like total fences.

Formally, the conditions for release consistency are:

- A read or write can perform with regard to any other process if all previous acquires have already been performed.
- A release can perform with regard to any other process if all previous reads and writes have already been performed.
- Synchronization operations cannot be reordered.

Figure 5(b) shows how release consistency allows reordering. *R(a)* and *W(b)* can be reordered but they cannot perform after $S_r(y)$ because it is a release operation (end fence). On the other hand, *R(e)* and *W(f)* cannot perform before $S_a(x)$ because it is an acquire operation (begin fence). Finally, *R(c)* and *W(d)* have both reordering limits.

2.2.4. Lazy Release Memory Consistency Model

Applying release consistency, the parallel program shown in the previous example of Figure 3, must rely on synchronization primitives to share data. Figure 6 shows the new code.

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

A code	B code

Figure 6 – Code example

Release consistency ensures that the write issued by *A* to variable *a* performs with regard to the process *B* before *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* performs with regard to *B* before the *lock(x)* performs. Thus, the write issued by *A* only needs to perform with regard to process *B*, even if more processes exist. Thus, although the protocol had non-coherent data copies in different nodes of the system, it maintains the consistency. The model that permits this kind of reordering is called the *lazy release memory consistency model* [KEL95].

Formally, the conditions for lazy release consistency are:

- A read or a write can perform with regard to another process if all previous acquires have already been performed with regard to that process.
- A release can perform with regard to another process if all previous reads and writes have already been performed with regard to that process.
- Synchronization points cannot be reordered.

Figure 5(c) showed the possible reorderings under lazy release consistency. As release consistency, acquires behave as begin fences, so *R(c)*, *W(d)*, *R(e)*, *W(f)* cannot perform before $S_a(x)$. The difference is that the performing of memory operations can be postponed until other process acquires the same synchronization variable than the following release in program order. In other words, the variable (*y*) used by the following release ($S_r(y)$) can be acquired ($S_a(y)$) by other process. In such a case, that acquire operation states the latest time a memory operation can be postponed until. When the acquire operation performs, previous writes must only perform with regard to the acquirer process.

The memory operations that must perform with regard to a given acquirer are those occurring before following a specified order, called *happened-before* [KEL95]. This order marks as previous all the memory operations issued by the process that released the same semaphore or barrier before the current acquire. It also means (in a recursive way) the memory references performed by the processes releasing the semaphores or barriers accessed by the process that issued the last release. Figure 7 shows a hypothetical sequence of acquires and releases between different processes. According to the happened-before order all the writes in the figure occurred before process *E* executed the acquire.

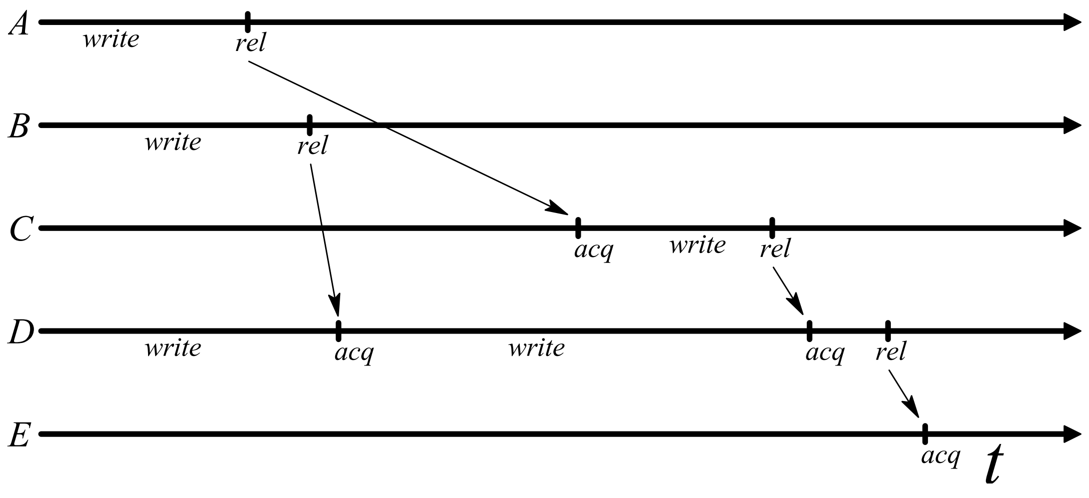


Figure 7 – Happened-before order example

2.3. Multiple Writer Protocols

To enable the reordering of memory operations under the lazy release memory consistency model an important piece of research has focused on multiple writer protocols, which allow several writers on a page at the same time, hiding most of the false sharing effects. In general, multiple-writer protocols store and send page differences (also called *diffs*) instead of the whole page to detect which parts of a given page each node has written to. Below, we discuss some details of the implementation.

In the sequential memory consistency model shared pages only can be written at a given time by a single processor. In other words, write operations must be serialized between processors. Protocols working in this manner are called single writer protocols and they are the most intuitive way to guarantee the single and strict ordering of operations that the sequential model requires. Relaxed memory consistency models allow reordering of non-synchronization operations, which supposes different processors may see different orderings. Moreover, in the lazy release consistency model, write operations are not globally seen, so writers on the same page do not need to carry out coherence actions (for example, page invalidations) among them, allowing several writers to write simultaneously on different parts of the page.

False sharing was the main reason because SVM system designers introduced multiple writer protocols. Its impact on performance is higher in SVM systems than in hardware DSM systems, because in the former it is more probable that two or more processes write to the same coherence unit, because of the larger page size. When false sharing occurs in a single writer protocol the well-known *ping-pong* effect is produced, which is magnified in SVM systems because of their software nature and the high latencies of the commodity network. Multiple writer protocols reduce this problem.

The main problem to solve when designing multiple writer protocols is how to prevent local modifications of a given page from being overwritten by those page modifications of a concurrent remote writer. In other words, if process *A* modifies a piece of its local page copy, an update to the same page from other process *B* will overwrite the whole page, and so local modifications of process *B* are lost. 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 (due to a page fault), it creates a copy of the page before the write is done – referred to as a *twin*-, and marks the page as read-write; iii) then, if another node requires the page, the twin and the page are compared to obtain the differences between them. The comparison results are stored in a table, called *diff*; and, iv) only the diff is updated, instead of the whole page, so avoiding that remote updates overwrite local modifications.

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

2.3.1. Invalidating versus Updating

Two kinds of multiple writer protocols are distinguishable depending on how they handle coherence actions: invalidation and update protocols. The relative performance of invalidation versus update protocols strongly depends on two important workload characteristics: the *granularity of sharing* and the *frequency of sharing.* These are explained below:

Granularity of Sharing

This characteristic quantifies the mean amount of data transferred when an update occurs. It is computed with regard 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 are 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 different sizes.

Frequency of Sharing

In SVM systems, coherence actions 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 two consecutive synchronization events [ZHO97]. We assume we are performing *fine-grained* synchronization (FGS) if the average computation time is close to the average synchronization time. Otherwise, we assume synchronization is *coarse-grained* (CGS).

The larger the granularity of sharing and frequency of sharing are, the larger the network utilization. When both are relatively large, the network saturates and becomes a congestion point. This point establishes the performance border between invalidate and update protocols.

Update protocols usually achieve a better performance than invalidate protocols before the network saturates. When this occurs, their performance dramatically drops, offering poorer performance than invalidate protocols. That is because invalidate protocols generate less traffic and so they saturate the network later.

On the other hand, the performances of invalidate protocols in SVM systems are limited by the frequency of sharing. This happens because page requests must be performed using asynchronous communication in SVM systems, which has a high latency, and the number of page requests grows as the frequency of sharing increases. For this reason, in general, invalidate protocols offer worse performance than update protocols when the frequency of sharing is high.

The multiple writer protocols commonly used in SVM (with the exception of those needing additional hardware or compiler support) are: Eager Release Consistency (ERC) protocol, the Lazy Release Consistency (LRC) protocol and the Home Lazy Release Consistency (HLRC) protocol. The ERC protocol is implemented as an update protocol, while both LRC and HLRC protocols are usually implemented as invalidate protocols, although they can be implemented in either way.

2.3.2. Eager Release Consistency Protocol

The multiple writer protocol implementation of the release memory consistency model is known as ERC protocol. It is implemented by updating diffs in those nodes sharing the page when the release is executed. Typical implementations of this kind of protocol can be found in [CAR91] and [SWA98].

Figure 8 shows a working example of an ERC protocol. When a process in node *A* executes the release that frees the semaphore *sem1*, the OS of node *A* must calculate and send the *diff(X,A)* containing the writes performed in page *X* by the processes in node *A*. The diff must be sent before the release operation ends in order to be applied in those nodes sharing the page. In this way, when a process in a remote node acquires the semaphore *sem1* it will have the page updated.

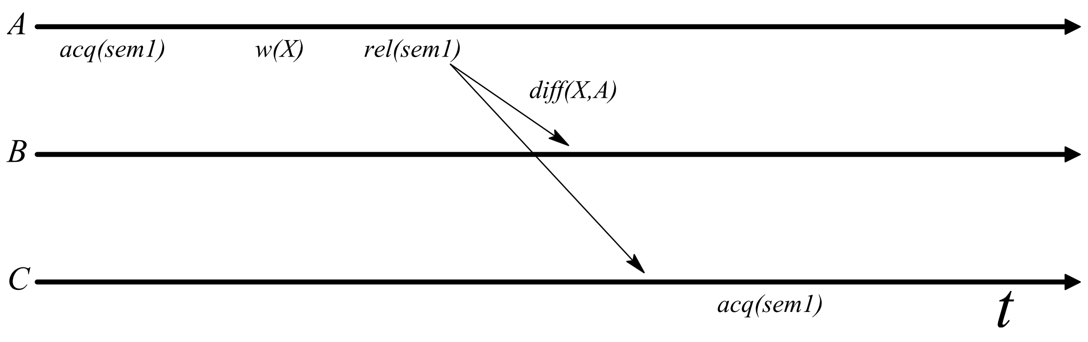


Figure 8 – ERC protocol example

In the release memory consistency model, memory operations perform releases globally. Therefore, implementations of the ERC protocol allow multiple writers by using update operations. This means that the ERC protocol is very sensitive to the granularity of sharing, because it broadcasts all the previous writes at the release. The frequency of sharing also affects the number of broadcasts (one per release operation). Nevertheless, the ERC protocol has been used in recent pure software SVM implementations [SWA98] because the absence of invalidations reduces most asynchronous communication.

2.3.3. Lazy Release Consistency Protocol

The ERC protocol performs poorly when the broadcasts saturate the available network bandwidth. In this case, it is a better to relax the memory consistency model to allow point-to-point messages instead of broadcasting to carry out the coherence actions through the network. The LRC protocol [KEL94] (which implements the lazy release memory consistency model) applies the coherence actions just to those nodes accessing a semaphore or barrier, instead of broadcasting the coherence actions to the other nodes as the protocol ERC does.

As these coherence actions are only applied to a given node, invalidation protocols can be implemented allowing multiple writers because the rest of nodes in the network will not be affected by the action. In this context, the invalidation information is called *write notice*.

Figure 9 shows an example of an invalidate LRC protocol. The writes produced by processes in nodes *A* and *B* to page *X* produce write notices (*wn(X,A)* and *wn(X,B)*). When a process in node *B* has a page miss, it receives the diff of page *X* from node *A* (*diff(X,A)*). So, when a process in node *C* has a page miss, it only needs to contact with node *B* to obtain the two diffs (*diff(X,A)* and *diff(X, B)*.

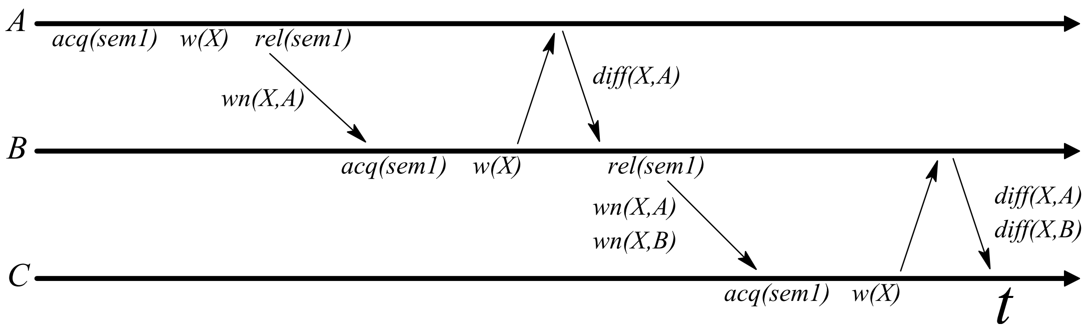


Figure 9 – LRC protocol example

It is not necessary to communicate all the previous write notices to the process acquiring the semaphore or barrier, but only those that have not been applied yet. The mechanism to know at the moment of the acquire operation which coherence actions have been applied and which must be applied is based on *intervals* and *timestamps vector*. Each time a process in a node executes an acquire operation or a release operation; the node increases its interval number. The timestamp vector contains the intervals of each node known to the node of the process that performs the acquire operation (through coherence actions). By comparing the timestamps with that of the node that performed the release, it is possible to know what coherence actions must be applied: those corresponding with the intervals unknown to the node that performs the acquire operation and known to the node that performed the previous release operation.

Table 1 shows an example of a node calculating the new vector timestamp when some of its processes execute an acquire operation. To handle the calculation, the node uses its current vector timestamp and the vector timestamp of the node of the last process that executed the corresponding release operation. The node that performs the acquire operation must apply all the coherence actions associated with the intervals of other nodes that are not in its vector timestamp, but are in the vector timestamp of the node that performed the release. For example, if we consider *A* as the node acquiring a semaphore, it will apply the actions from the intervals 2 to 10 of *B* and the interval 12 of *D*. The intervals of *C* are not applied because *A* has applied more intervals than the node releasing the semaphore.

Node	Release	Acquire	
		After	Before
A	0	5	6
B	10	1	10
C	7	9	9
D	12	11	12

Table 1 – Vector timestamps calculation example

After applying the intervals, node *A* updates its own vector timestamp with the intervals applied and increases its own interval (from 5 to 6) because it has just finished the acquire operation.

When a node invalidates a page due to a write notice, it will obtain updates from the writers at the moment of a page miss. Because it is impossible to know the moment when the diffs will be needed, they must be stored until the moment they are applied in all the nodes. Generally, they are not stored indefinitely but they are broadcasted at the same time than barriers and 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. Thus, given sufficient bandwidth, the frequency of sharing is a key performance factor even more important than the granularity of sharing in invalidation based LRC protocols.

2.3.4. Home Lazy Release Consistency Protocol

The HLRC protocol [ZHO96], like previous protocols, implements the lazy release memory consistency model. In this protocol each page has associated a *home* node that concentrates all the diffs. When a node writes in a page, it supplies the diffs only to the home node. Then, diffs can be removed from the writing node. The remaining nodes invalidate the page by applying write notices following the same happened-before order as the LRC protocols.

In the case of a page miss, due to write notices, the OS of the faulting node asks the home node for an updated page. Because of network delays, the needed diffs may have not yet arrived at the home node. In this case, the request is queued until the diffs arrive. Vector timestamps are used to discover if the page of the home node is sufficiently updated. The vector timestamp associated with a page encodes the number of intervals updated by each node.

Figure 10 shows a write sequence equivalent to that of Figure 9 but following an HLRC protocol. The diffs generated by nodes *A* and *B* are gathered by node *D* because it is the home node of page *X*. Write notices are distributed in the same way than in the LRC protocol. As nodes *B* and *C* have their page invalidated by the write notices (*wn(X,A)* and *wn(X,B)* respectively) they ask the home node for an updated page version. The request message of node *B* arrives (t_1) after the diff of the node *A* (t_0) therefore the OS of node *D* can immediately satisfy the request. But, the request from node *C* (t_2) must wait to be satisfied until the diff of *B* arrives (t_3).

Processes running in the home node of a page never have a fault for that page, because they always have the page updated by diffs produced by remote writers. Thus, if the home is properly chosen (for example, by profiling), asynchronous communication is reduced. This fact mitigates the importance of the frequency of sharing. The granularity of sharing is not as important as under the ERC protocol because writers only update the home. For this reason, HLRC multiple writer protocols are the most used in SVM system implementations.

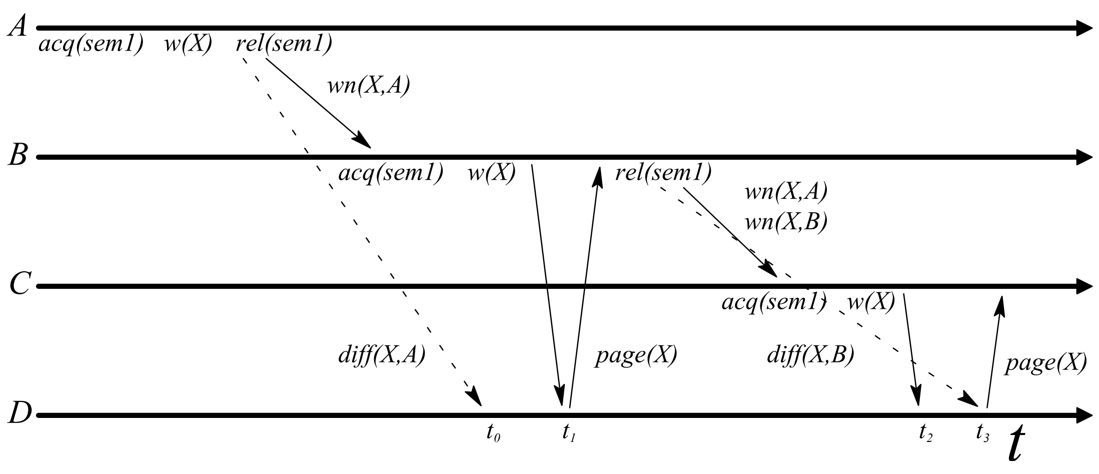


Figure 10 – HLRC protocol example

Recent research in SVM systems has attempted to further reduce the impact of asynchronous communication in SVM systems, which is a key factor in improving their performance and one of the most problematic points in their design. The next section explains asynchronous communication, detailing the most recent research in this area.

2.4. Asynchronous Communication

Most current parallel workloads have been optimized to run on distributed hardware systems (e.g., symmetric multiprocessors) or supercomputers. SVM systems lack hardware support for a lot of tasks supported by these hardware systems. This lack is the reason why SVM systems can experience performance losses and why they must be implemented using asynchronous communication [BIL98].

Basically, in all asynchronous communication, a *client* node initiates a request and a *server* node services the request. For example, the client node can require the server node to read a given page or to lock a given semaphore. Then, the server node is interrupted to service the request. This asynchronous communication involves a context switch in the server node, which introduces both high service latencies and overhead in the server. This implies a high operational cost that produces high service latencies and wastes precious computing time in the server [BIA96].

An important piece of recent research in SVM systems is aimed at reducing asynchronous communication by several methods. Some mechanisms include hardware support that partially, or totally, avoids this kind of communication [BIA96][BIL98][BLU98][STE00]. Others try to reduce this communication, or hide its latency using software techniques [BIL97][SPE98][SWA98].

The section below discusses asynchronous communication implementation in SVM systems, as well as the kinds of asynchronous communication that can be found in the design of a memory consistency protocol. We also analyze and discuss some techniques used to reduce the impact on performance for each particular kind.

2.4.1. Asynchronous Communication Implementation

Asynchronous communication can be implemented by using polling or interrupt techniques. Polling periodically wastes some processor cycles (usually, close to 10) when checking if there are new messages to serve. However, the service time of polling techniques is much lower (several orders of magnitudes [ZHO97]) than that offered by interrupts. Therefore, in general, polling is preferable. On the other hand, processor cycles are spent in polling whatever the communication is. Consequently, if there is little communication, interrupt techniques are preferable.

The use of interrupts or polling depends on the OS the nodes are executing. The Brazos system [SPE98] uses the Windows NT operating system, which incurs too much overhead dealing with interrupts. To solve this problem, Brazos (designed using multithreading) dedicates one thread to poll the requests. On the other hand, systems like Quarks [SWA98], implemented under UNIX, are based in interrupts. These systems often enter in busy waiting mode when the node is blocked waiting an answer from other node. If the nodes are multiprocessors, this technique is more effective, because it is more likely to find a blocked processor than it is in single processor nodes [KAR96].

In general, in multiprocessor nodes, only one processor serves asynchronous communication. When using interrupts, the balancing of this overhead, for example by using a round-robin scheme, can incur longer interrupt service times [BIL97]. If the system uses polling it is sufficient that one process in the node deals with this overhead

[SAM98][STE97].

2.4.2. Types of Asynchronous Requests

In the SVM protocols, there are several kinds of client requests that produce asynchronous communication. A possible classification is: data request, data receive, semaphore request, and barrier request.

Data Request

Data requests need to be served with a high priority. They appear in protocols that use invalidations as coherence actions. 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 answers submitting the whole page while the action the server performs on LRC based protocols as a response is to submit the correspondent diffs.

Software techniques update the data avoiding the consequent asynchronous request. The Quarks system [SWA98] uses an ERC update protocol, but most cases use a hybrid protocol that behaves like an invalidation protocol, switching to an update policy when certain conditions occur. The Brazos system [SPE98] uses multicast to update other nodes in the copyset of the page if they have a data request for the same page, as well as to update predicted clients before they leave the barriers. Stets *et al*. [STE00] measure the performance of a multicast protocol based on a history record, but with hardware support.

The hardware techniques can update the data like the software techniques or can serve data requests automatically without processor intervention [BIA96][BIL98]. In [BIA96], a hardware support for an LRC based protocol that serves data requests is proposed, but the processor is still interrupted to perform metadata maintenance tasks of the data structures related to intervals. In [BIL98] the NI processor of the Myrinet is used to serve pages automatically in a HLRC based protocol.

Data Receive

As mentioned in section 2.3.1, some protocols update data in order to avoid data requests. In a data receive request, a client produces the data and pushes it into a repository server (for example, the home of the page), producing asynchronous communication.

This asynchronous communication can be easily removed by software. Swanson *et al.* [SWA98] proposed a remote deposit mechanism based on low latency software messages. This mechanism can be also implemented by hardware [BLU98][BIL98][STE00]. Software implementation leaves the data in a low priority queue, that is later checked by the server but hardware mechanisms go a step further, allowing data updates directly in memory.

Semaphore Request

SVM semaphores are implemented by mapping lock and unlock calls to message requests. Because they have high priority, a semaphore request must be served as soon as it arrives at the server. In SVM systems, each semaphore has a *home* node, which maintains a queue of semaphore requests. The *home* also forwards lock requests between nodes to maintain a distributed list of semaphore requesters. An unlock request is performed without home intervention using this distributed list.

SVM semaphores work as shown in Figure 11. Semaphores are managed in a FIFO distributed queue. Each node in the queue points to, by means of a *next* field, the following node that requests the semaphore. To do this, a node sends a *lock* message (arc solid line) to the *home* of the semaphore (node *H*), which redirects it to the last queued node (node *P*). This node will update its *next* field to point to the new requester (node *Q*). When a node releases the semaphore, it leaves the queue and grants the semaphore to the node pointed to by its *next* field by sending an *unlock* message (straight solid line). Following requesters (nodes *Q*, *R*, and *S*) will follow the same rules (dotted lines).

Relaxed memory consistency models together with multiple writer protocols increase the service latency of semaphore requests. The overhead includes the maintenance of interval lists and write notices, as well as diff calculation if the protocol starts some automatic update at synchronization points (as occurs in ERC and HLRC based protocols).

There are software techniques that try to reduce the impact of this overhead. Some models such as [STE00] and [BIL98] broadcast the write notices as soon as they are generated, uncoupling their transmission from the synchronization. This technique is effective due to the low bandwidth cost the write notices have. Although in both cases hardware is used to broadcast (remote deposit) the write notices, it is possible to do the same using software as is performed in [SWA98]. If the protocol updates the data at synchronization points (updating the home as in HLRC, or broadcasting the modifications as in ERC), the overhead is avoided by updating the data after the semaphore release [BIL97]. In [SWA98], data updates are also performed in semaphore acquire operations, overlapping the updates with semaphore waiting time.

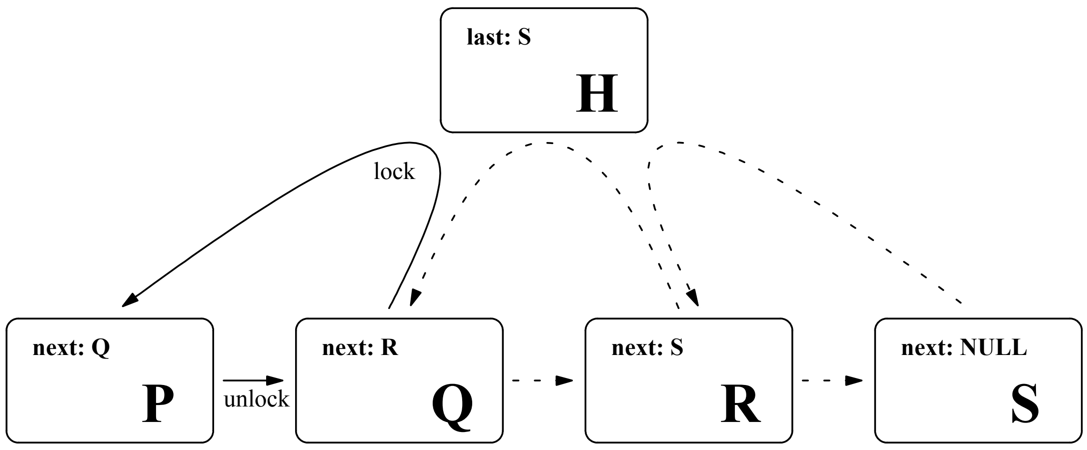


Figure 11 – SVM semaphore management example

Hardware support can automatically serve the semaphore. This can be implemented in different ways. For example, in [BIL98] the Myrinet NI serves the semaphore following the mapping of semaphores over the message passing scheme shown in Figure 11. Stets *et al.* [STE00] use the total ordering capabilities of the network to implement spinlocks.

Barrier Request

Barrier requests do not have priorities as high as semaphore requests because they require all the nodes of the system to be involved in the communication. Thus, there is no need to start any asynchronous communication each time a barrier request is received. This can be done without additional hardware [BIL97]. When the server enters the barrier, it checks the incoming message queue. If it has a request from each node in the system, the server releases it. Otherwise, it polls for the barrier requests of the missing nodes.

2.5. Conclusions

SVM systems are parallel systems that use the OS virtual memory subsystem to share memory. This makes SVM systems a both highly portable and inexpensive alternative to other hardware based parallel systems. In section 2.1, we introduced a simple example of an SVM system to illustrate the basic aspects related with their design.

In the past, SVM systems enforced memory consistency by sending explicit messages to other nodes when a write operation was detected to a shared address. This kind of memory consistency is known as sequential memory consistency. Sequential memory consistency has severe performance drawbacks because it needs to send a lot of coherence messages which have high latency. To mitigate these drawbacks, past research focused on relaxing memory consistency models and multiple writer protocols. In section 2.2 and section 2.3 we explained the most widely accepted memory consistency models and protocols from the open literature.

Relaxed memory consistency models highly reduce the number of coherence messages. However, there is still an important performance gap between hardware based systems and SVM systems. This performance gap is due to asynchronous communication. Asynchronous communication is used in SVM systems for data sharing and synchronization purposes. Both types of asynchronous communication introduce long latencies because they are usually implemented with software interrupts that are sent via a commodity network (typically a low speed network). In section 2.4, we categorize the different types of messages used for asynchronous communication and summarize the most important research aimed at reducing the performance gap.

To reduce the performance gap, a common technique is to use polling instead of interrupts to implement the asynchronous communication [KAR96][SAM98][SPE98][STE97][ZHO97]. In addition, recent research tries to reduce the number of asynchronous messages sent. On one hand, pure software methods propose: a) update protocols that avoid asynchronous communication due to page faults [SPE98][STE00][SWA98], b) uncoupling some overhead from asynchronous communication [BIL97][SWA98], and c) lowering priorities of some asynchronous communication messages [BIL97]. On the other hand, some proposals include hardware accelerators to: a) serving shared data automatically when requested [BIA96][BIL98], b) updating shared data before it is requested [BLU98][BIL98][STE00], and c) serving semaphores [BIL98][STE00].

Chapter 3

The Simulation Environment: LIDE

Current results presented in the open literature regarding SVM systems are usually obtained from real systems. This is an inflexible way to design the whole system. The use of simulators is a cheaper and more flexible way to handle the design. This chapter introduces the simulation environment we developed for DSM systems, specifically aimed at SVM systems in networks of workstations. The proposed tool simulates the detailed behavior of these systems, taking as inputs the memory consistency model and the local area network. It is an execution-driven simulator [DAV91] that can make use of real typical benchmarks such as the SPLASH-2 benchmark suite.

The proposed tool is an execution-driven simulator aimed at studying the behavior of memory consistency models, with the exception of those needing compiler modifications. It presents a cheap and flexible way to undertake performance studies and design efficient consistency models for SVM systems.

The proposed simulation environment is called LIDE (LImes & siDE). It was developed from two simulators, LIMES [MAG97] and SIDE (initially called SMURPH) [DOB93]. Figure 12 shows the block diagram of the proposed environment, and how the simulators connect to each other. LIMES enables parallel program execution and it is used to collect the memory references generated by the workloads. SIDE allows memory consistency model design, and injects the transactions needed for coherence maintenance.

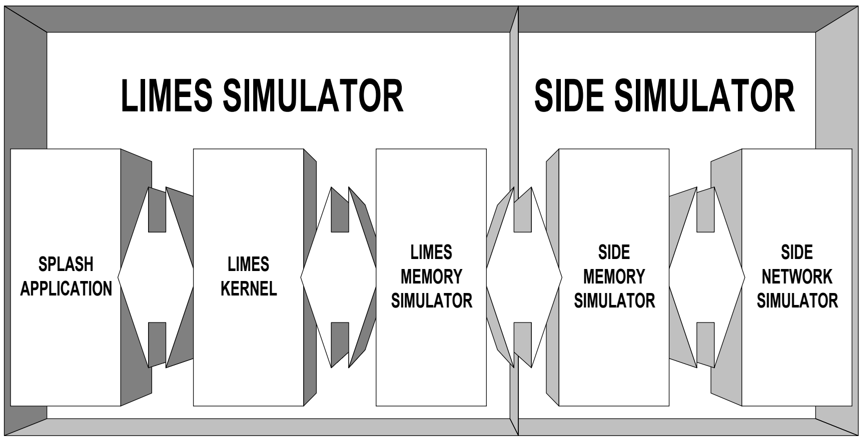


Figure 12 – Block diagram of the LIDE simulation environment

This chapter is organized as follows. Sections 3.1 and 3.2 describe respectively the LIMES and SIDE simulators. Section 0 details how both simulators communicate in order to implement the LIDE simulation environment. Section 3.4 discusses the block structure of LIDE. Finally, section 3.5 presents some concluding remarks of this chapter.

3.1. LIMES

LIMES is a cache memory simulator of SMP systems running on i486 compatible processors similar to TangoLite [DAV91]. Although LIMES was designed to simulate i486 SMP systems, it can be concluded [MAG97b] that the number and composition of the memory references generated by these systems is similar to those obtained from a RISC multiprocessor machine. Therefore, the memory references obtained from simulations of i486 SMP systems can be extended for simulating other RISC SMP systems.

An SMP memory model is very different from an SVM model. However, although the differences are only expressed from a physical point of view (architectural design and temporary behavior), the references generated by distributed programs executed on these systems are equivalent. In other words, the logical behavior of a DSM system does not really differ from an SMP system. In consequence, to simulate an SVM system, it is only necessary to take the references generated by an SMP simulator such as LIMES and inject them in a simulator of the SVM memory and network system (SIDE). By using UNIX pipes, LIMES sends to SIDE the memory references to be simulated for each processor, and SIDE replies to LIMES when the reference is satisfied.

Figure 13 shows the temporal behavior of the components of the LIMES simulator. LIMES does not call the memory system each simulation cycle because that would suppose multiple context-changes; and consequently would increase the simulation time. Instead, LIMES only calls the memory system when there is a pending memory reference. One of the parameters of this call is the time elapsed since the last memory reference was issued. The memory simulator executes a loop for each elapsed cycle from the last reference to the current time. If within the loop, the simulator is in a stable state, which does not change until a new reference is generated, then it simply increases the simulation time to the current reference without further spinning. This optimization speeds up the simulation time.

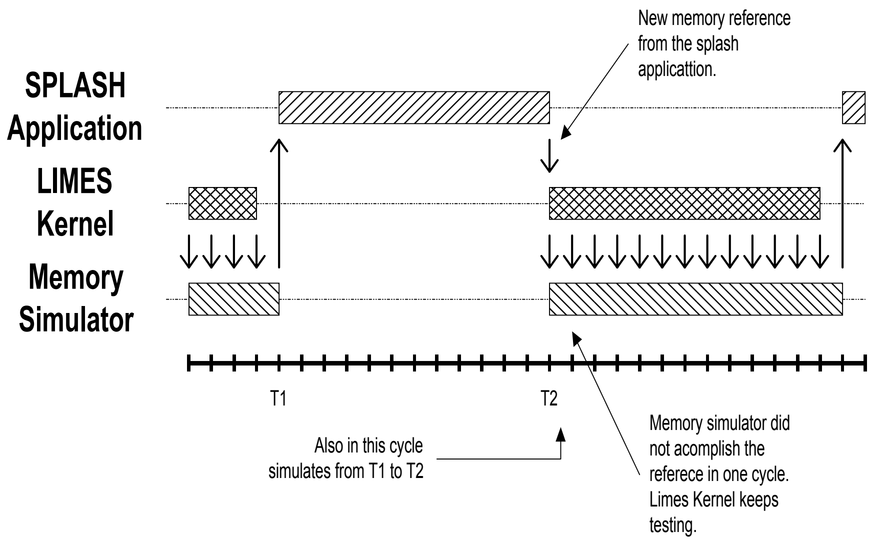

New memory reference from the splash applicattion.
SPLASH Application
LIMES Kernel
Memory Simulator
T1
T2
Also in this cycle simulates from T1 to T2
Memory simulator did not acomplish the referece in one cycle. Limes Kernel keeps testing.

Figure 13 – Temporal relation example between the SPLASH application, the simulator kernel, the memory simulator and the simulation time in LIMES

If the current memory reference is not satisfied in the cycle that the memory simulator was notified, LIMES calls the memory simulator during each processing cycle until the reference is satisfied. Therefore, the memory simulator is called each cycle only if a reference is pending. Generalizing for more processors, the LIMES kernel keeps testing the memory simulator if some processors are still waiting for a pending reference.

The current version of LIMES does not have threads support. Therefore, a stalled processor only can wait for a pending reference.

3.2. SIDE

SIDE is a heterogeneous environment for the simulation of networks and processes (state machines). It has two main advantages. Firstly, it offers an object-oriented programming model with defined classes for the management of several types of synchronization between processes (tails, signals, shared memory, messages, etc.). Secondly, it provides several examples of high performance networks.

We have introduced in SIDE the code to simulate the SVM systems handled by the operating system. The code includes both the consistency model and the simulator of the physical network interconnecting the nodes. Both models are duly isolated. Thus, it is quite easy to change the physical network module without any, or just minimal, modification of the memory system code.

SIDE controls the simulation of a scheme of asynchronous events and processes. The execution of different parts of the code for each process depends on the arrival of the expected events, as shown in Figure 14. Events are message arrivals, signals from other processes, arrival of a process to a particular state, timeouts generated by processes, etc.

Processes in SIDE are executed in nodes. A node can manage several processes, such as processes for simulating the OS, others for network devices, etc. The processes can only synchronize through events. Therefore, it is impossible to know if the state of the system is stable at a given time as in LIMES, because it depends on the events asynchronously produced by other processes (for instance, incoming network messages).

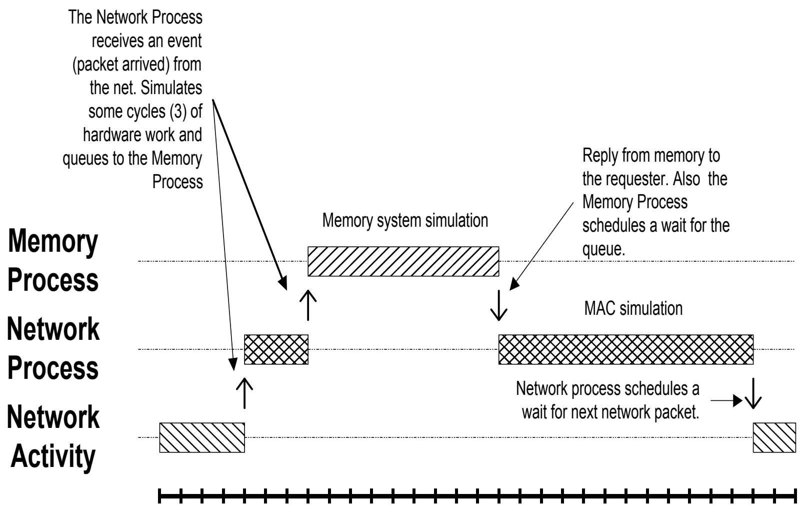

The Network Process receives an event (packet arrived) from the net. Simulates some cycles (3) of hardware work and queues to the Memory Process
Reply from memory to the requester. Also the Memory Process schedules a wait for the queue.
Memory system simulation
Memory Process
MAC simulation
Network Process
Network process schedules a wait for next network packet.
Network Activity

Figure 14 – Temporal simulation example in the SIDE simulator

3.3. Connecting and Executing LIMES with SIDE

As introduced above, LIDE uses LIMES to obtain the processor references, and it injects them in SIDE, which simulates the behavior of the memory and network systems. SIDE processes several references at the same time, and replies to LIMES when any reference is satisfied. Communication between both simulators is made by means of UNIX pipes.

Because both simulators use different simulation techniques (LIMES simulation is synchronized cycle by cycle, SIDE simulation is asynchronously event driven), the main problem in the design and implementation of LIDE was to establish at run-time the convenient flow of events between LIMES and SIDE to assure the correct execution of the parallel benchmarks.

If SIDE is blocked while waiting a new memory reference from LIMES, its own event driven simulation is blocked. On the other hand, SIDE cannot simply check for a new memory reference from LIMES on the UNIX pipe and continue with its simulation, because some memory references from LIDE can be missed if they arrive before the next pipe check.

The obvious solution to this problem is that SIDE performs the pipe synchronization cycle-by-cycle. However, because of the asynchronous nature of SIDE simulation, it is impossible to know the global system state at any given moment. Therefore, it is impossible to know when the SIDE simulator has reached a stable state and carry out the optimization explained in Section 3.1 to reduce the synchronization overhead. Without this optimization, synchronizing each cycle introduces a huge overhead, which grows because the synchronization is performed by UNIX pipes. This can be tackled in a fast DSM system simulation because the memory service takes just a few cycles; but this takes much longer in an SVM system (thousands of cycles), and consequently, this method is not feasible.

To solve this problem, we developed a new synchronization scheme that does not need cycle-by-cycle synchronization. In the proposed scheme, SIDE counts the real simulation time and LIMES is only aware of the correct order in which the references are satisfied. The working scheme is organized in the five steps discussed below:

1) The memory simulator (LIMES) waits for all the processors to issue a memory reference.

2) The memory simulator (LIMES) sends to SIDE the memory reference and the number of processing cycles of each process to be simulated. These cycles are independently counted in each processor as the amount of processing time elapsed since the previous memory reference.

3) With the information from 2) SIDE then simulates in parallel for each processor:

    a) The corresponding processing cycles.

    b) The service time for the current memory reference.

4) SIDE replies to LIMES when one of the memory references is satisfied.

5) Return to the step 1.

Proceeding in this way, we ensure that SIDE will only check for new memory references from LIMES when a pending memory reference is satisfied. Therefore, it does not use the pipe every cycle.

Two different simulators are used by LIDE, so two different time-scales are available. By using the above scheme, the correct time information comes from SIDE because LIMES is used only as a helper process that generates references.

Figure 15 shows how the proposed scheme works. Only when the LIMES simulator arrives at T1 it is possible to know all the references from the processors involved (step 1). Then, LIMES sends to SIDE the corresponding information (step 2) by means of the pipe. SIDE then simulates the number of cycles corresponding to the number of instructions from the previous reference to the actual reference in each processor (step 3). The processor 0 reference is satisfied when the timeline arrives at T2, and then LIMES is restarted (step 4). Nine cycles later, processor 0 issues another reference and SIDE continues simulating these 9 cycles from processor 0 and retake the work left in the memory system 1 (step 5).

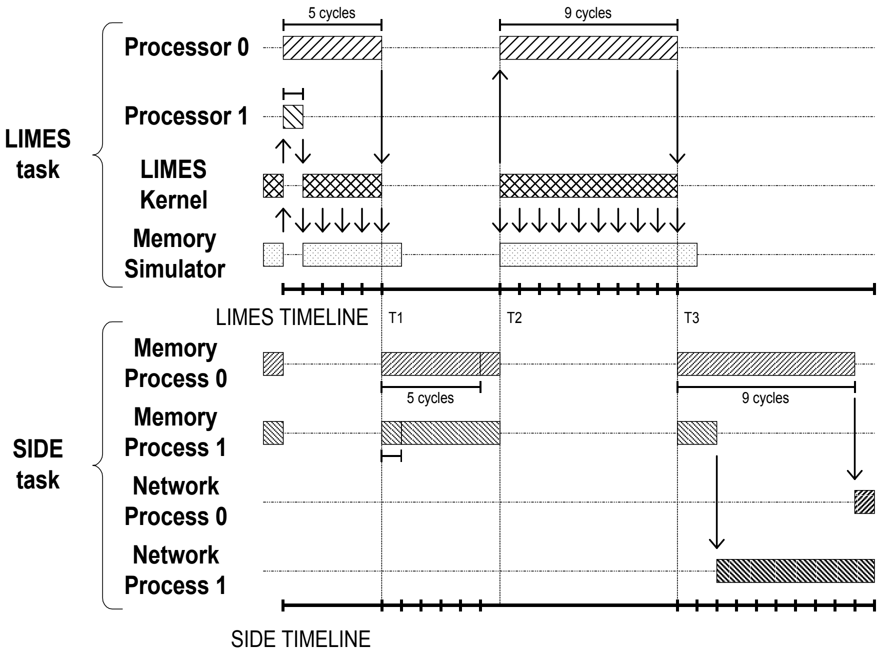


Figure 15 – Example of LIMES and SIDE working together

In this way, we achieve the correct execution because LIMES knows the order in which the events are generated, and we obtain the correct time because SIDE knows it.

3.4. LIDE Block Structure

This section details the block diagram of the proposed simulator. We have developed a new memsim (from LIMES memory simulator) that is connected to SIDE by means of two named pipes. The feedback that LIMES receives from SIDE, gives the order in which memory requests are accomplished, and this ensures the correct operation of the workloads code that it is dependent on synchronization.

In SIDE we have several processes synchronized by events. These are represented in Figure 16. The root process implements the communication with the LIMES memory simulator by UNIX pipes, and multiplexes the references among the memory processes. Memory processes simulate the memory system behavior for each processor. Finally, SIDE mailboxes communicate the memory and root processes.

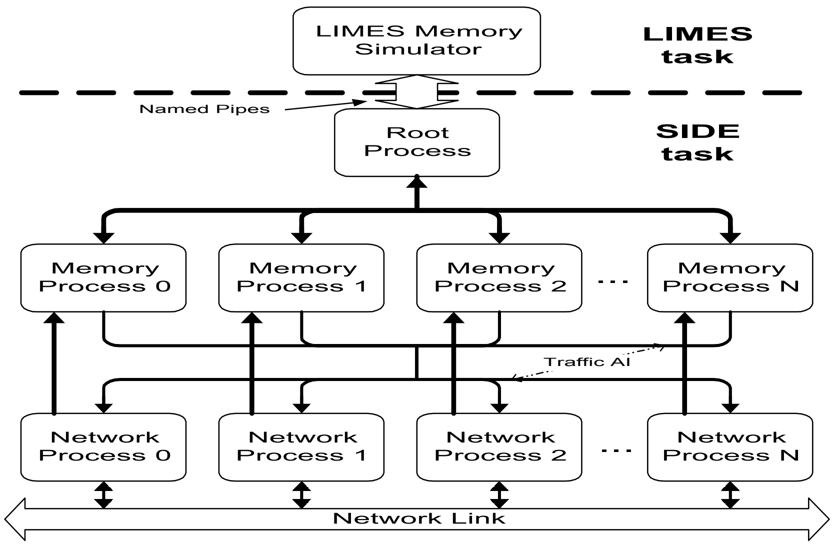


Figure 16 – LIDE synchronization paths and processes

The simulator design ensures independence between the network topology and the memory processing modules. Therefore, we can check different network protocols and configurations without modifying the memory simulator code. For example, Traffic AI is the statistical method used to generate packets in SIDE. This method is not used in LIDE to generate packets; but packets coming from the memory processes are sent to the network processes using the Traffic AI as an interface.

Class heritage and macros encapsulate other issues such as system setup and packet reception from the network to the memory processes. As Figure 17 shows, the network code module is only dependent on the lideRoot module. All interfacing for network programming is coded in the modules lideRoot and lideStation.

The lideProcess module has the memory process code. This code uses macros to access top classes, which implement transaction requests and memory management. In this module, different consistency models can be implemented.

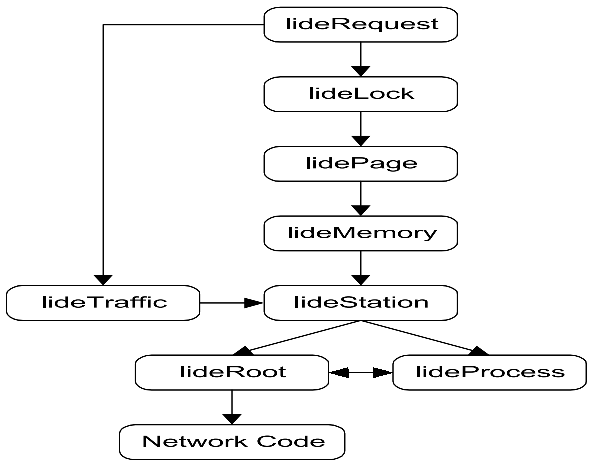


Figure 17 – LIDE module dependencies

3.5. Conclusions

This chapter presents LIDE, a simulation environment for distributed shared memory systems. In this environment, network simulation and memory architecture simulation are placed in two independent and de-coupled structures to make component changes easier. LIDE is based on two well-known simulators: LIMES and SIDE. LIMES is an execution driven simulator that collects and simulates the references generated by the execution of typical parallel benchmarks. SIDE simulates the memory architecture and the interconnection network.

There are two possible purposes of LIDE used in SVM systems:

1) Consistency models evaluation studies: The tool allows the development and

evaluation of current SVM consistency models as well as the implementation and evaluation of new proposals. This gives an opportunity to make performance comparisons between different solutions, in a much more flexible way than when using real systems.

2) Network of workstations evaluation studies: To study which standard types of NOW are most suitable for the implementation of SVM systems, by using the different examples included with SIDE (ATM, Ethernet, etc.); or by modeling new ones (100VG AnyLan, FDDI, etc.).

LIDE is free software currently available at the URL:
http://godzilla.disca.upv.es:8181/~spetit/lide/.

Chapter 4

Workload Characterization

Chapter 2 discussed the nature and basic working of SVM systems. We focused on the reasons causing performance limitations, and presented the current research directions aimed at improving performance. The main solutions presented in the open literature often improve some specific aspects of the SVM system design and implementation, for example, most of those solutions improve the performance of the protocols used, and others propose hardware accelerators.

In addition to the performance problems discussed in Chapter 2, there are other problems related with the synergies between the parallel workloads and the SVM system running them. For example, a common tradeoff in parallel systems that involves update versus invalidation protocols is dependent on two important characteristics of the workload such as the granularity and the frequency of sharing [IFT96].

To be able to improve the performance of an SVM system, we need to previously develop a good understanding of how typical workloads interact with the underlying system. In this chapter we characterize a set of parallel workloads suite from the standpoint of the SVM protocol designer, identifying the critical workload features that must be exploited to improve performance.

This chapter is organized as follows. Section 4.1 discusses the axes of characterization we selected. Section 4.2 describes the parallel workloads used. Section 4.3 concentrates on the sources of performance loss. Section 4.4 characterizes the workloads with the selected axes, analyzing the results. Finally, section 4.5 presents some concluding remarks.

4.1. Axes of the Characterization

To characterize the workload, we have selected three axes to capture the intrinsic behavior associated with the asynchronous communication presents in real SVM systems. The axes we will use in this study are: the *frequency of sharing*, the *granularity of sharing*, and the dynamic changes in the *sharing pattern*.

The software nature of SVM systems interacts with the parallel workload, often dropping performance with regard to hardware systems. The main reason for performance loss is that parallel workloads are usually optimized for hardware systems. This fact defines their granularity of sharing and their frequency of sharing. Both metrics will be defined in section 2.3.1.

In addition to the frequency and granularity of sharing characteristics, we also use a third axis of characterization that it is useful to express the potential synergies between parallel workloads and SVM systems: the *sharing pattern*.

During workload execution, data sharing follows a given pattern. This sharing pattern can be stable throughout the workload execution or can dynamically change. According to 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:

- *1P-1C*: There is only one (1) 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.
- *1P-MC*: There is just one producer and multiple (M) consumers.
- *MP-1C*: There are multiple producers and only one consumer.
- *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.

4.1.1. Performance Synergies

With regard to the frequency of sharing, hardware systems support fine-grained synchronization (for example, via spinlocks) and the workloads are optimized to this point in order to minimize the communication to computation ratio. In these systems, the cost of the synchronization events is not high, and this allows small critical sections to be frequently executed. This technique is often used to protect access to small sections of shared data or to implement shared task queues. When running in SVM systems, the synchronization is implemented via asynchronous communication as commodity hardware has no support for fine-grained synchronization between nodes. This software communication and its frequency are the main causes of performance loss.

In relation to the granularity of sharing, Zhou *et al.* [ZHO97] showed that the software overhead in SVM systems reduces the performance in workloads showing fine-grained granularity of sharing. In that case, the writers only produce a small percentage of the page. In SVM systems, data is shared at virtual page granularity (typically 4 KB), as a consequence, the number of page faults due unrelated processes writing and reading the same page increases. As page faults involve asynchronous communication in current protocols like LRC and HLRC, the performance drops dramatically in those workloads showing fine-grained granularity of sharing. In other words, invalidation protocols like LRC and HLRC save an important percentage of network bandwidth, but the latency costs of the added asynchronous communication due to workloads showing fine-grained granularity of sharing, defeats this advantage. Using a pure broadcast protocol like ERC does not easily solve this problem, because some workloads still need a lot of bandwidth to obtain a good performance.

The fine-grained granularity of sharing, which most parallel workloads present, also changes the sharing patterns that can be observed during execution in an SVM system. The whole effect is detailed in section 3.3.2.

4.2. Workload Description

The workloads that we use in this study are a subset of the SPLASH-2 benchmark suite [WOO95].

Below, we describe them while focusing on the axes of characterization. For description purposes, workloads are divided into two groups: *regular* and *irregular,* according to the distribution of the shared data on the nodes.

4.2.1. Regular Applications

In these applications the shared data is organized in arrays, and uniformly distributed across the nodes. Data distribution patterns are often predictable, as well as the granularity and frequency of sharing, which often depends on the problem size.

- **Radix**: This kernel implements a distributed integer radix sort. It follows the 1P-1C sharing pattern [IFT96]. Its granularity of writing is always FG, while its granularity of reading is MG or CG (depending on the problem size). Our results will show that Radix has CGS synchronization.

- **Ocean**: This application simulates large-scale ocean movements based on eddy and boundary currents. Ocean follows a 1P-1C sharing pattern [IFT96], with CG granularity smaller than typical problem sizes. Ocean has CGS synchronization [ZHO97].

- **FFT**: This kernel calculates the Fast Fourier Transform in six parallel steps, and uses barriers for global synchronization. FFT follows a 1P-1C sharing pattern. The granularity of writing is CG, while the granularity of reading is FG or MG, depending on the problem size [IFT96]. This kernel represents the workload possessing the smallest amount of synchronization (it contains five barriers during its entire execution).

- **LU**: This kernel factors a dense matrix into the product of a lower triangular and upper triangular matrix. Its associated frequency of synchronization is CGS (through barriers). Its sharing pattern is 1P-MC [IFT96]. There are two versions of LU (continuous and non-continuous), each requiring different partitions of the input matrix. The continuous version factors the matrix as an array of blocks. This data structure maximizes the data locality in each partition. In the continuous version, the granularity of sharing is CG. The non-continuous version implements the matrix to be factored as a bi-dimensional array. The resultant algorithm is conceptually simpler than the continuous version, but it exhibits lower data locality. In the non-continuous version, the granularity of sharing is FG [JIA97]. Both versions have CGS synchronization [ZHO97].

4.2.2. Irregular Applications

In these applications the data distribution is less predictable than the distribution found in regular applications, which complicates the load-balancing task. To this end, the workloads usually use dynamic mechanisms that use task queues. The granularity of sharing and the frequency of sharing are fine-grained, which allows for significant overlap of communications with computations.

- **Barnes**: This application implements the Barnes-Hut method to simulate the interaction of a system of bodies (N-body problem). Barnes shows a 1P-MC sharing pattern through its whole execution. Two different stages can be identified in this application [IFT96]. The first stage includes the particle update and force calculation phases, which have regular behavior. In the second stage, a shared tree of particles is built among the parallel processes in a migratory sharing pattern, and the tree is then partitioned into spatial zones protected by semaphores. The tree processing follows a FG sharing pattern, and FGS synchronization, regardless of the problem size [ZHO97]. Thus, Barnes can be considered a hybrid application, because there are two distinct stages exhibiting different behaviors (regular and irregular).

- **Water**: This application evaluates both forces and potentials that occur in systems of water molecules. There are two versions of the water application, 1) Water-nsquared and 2) Water-spatial. Water-nsquared uses an $O(n^2)$ algorithm to calculate the motion of the water molecules over time. Water-nsquared has CG granularity, and possesses a migratory sharing pattern [IFT96]. Water-nsquared has FGS synchronization [ZHO97]. Water-spatial uses a different algorithm that imposes an uniform 3D grid of cells on the problem domain, using an O(n) algorithm to calculate the motion. During the motion, molecules move through cells owned by different processes. In general, water-spatial follows a 1P-1C pattern for large problem sizes. For smaller problem sizes, this application follows a 1P-MC sharing pattern, possessing CG granularity [IFT96] and has CGS synchronization [ZHO97].

Table 2 summarizes the behavior of the studied workloads along the three axes of characterization.

	Benchmark	**Granularity of Sharing**		**Frequency of Sharing**	**Sharing Pattern**
		Writing	**Reading**		
Regular	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	
Irregular	Barnes	FG		FGS	1P-MC
	Water (nsquared)	CG		FGS	Migratory
	Water (spatial)	CG		CGS	1P-1C

Table 2 – Workload characteristics according to the three axes of characterization

4.3. Sources of Performance Loss

In this section we describe the performance loss associated with the proposed characterization axes. In general, performance loss arises when the frequency of sharing is high and/or the granularity of sharing is small. If the frequency of sharing is high, performance drops due to *critical section dilation* [JIA97]. If the granularity of sharing is small, the performance drops due to *sharing pattern conversion* [IFT96]. Below we describe both phenomena.

4.3.1. Critical Section Dilation

Hardware systems usually support fine-grained synchronization and workloads are optimized to minimize the communication to computation ratio. In systems supporting FGS, the cost of synchronization events is small (in relation to an SVM system), which allows short critical sections to be frequently executed. Critical sections are used to protect shared data or to implement shared task queues.

For SVM systems, the time that parallel workloads spend in critical sections increases because of two main reasons:

- SVM systems do not support FGS synchronization. Thus, the synchronization primitives, such as locks or barriers, are mapped to a distributed set of queues that are managed by particular nodes in the distributed system.
- Some SVM systems carry out invalidations at synchronization points; thus, increasing the probability that a page fault occur while executing the critical section. A page fault usually requires the invalidated node to retrieve a copy of the page from another node.

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

The sum of the mentioned latencies implies that the total time that workloads spend in the critical sections is much higher in SVM systems than in hardware systems. Since critical sections are frequently executed, the contention increases, which also results in lower performance. This effect is so important than 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.

4.3.2. Sharing Pattern Conversion

Parallel workloads follow a pattern we will refer to as the *inherent sharing pattern*. This pattern can be static or can change dynamically throughout workload execution. Since data instances are shared at a given granularity, this granule size can be carefully optimized to map efficiently onto specific hardware systems. We will refer to this granularity as *workload granularity*, while we will refer to the sharing unit granularity supported by the system as *system granularity.* In general, the workload granule size is small (less than 64 bytes, FG), but it can change with the problem size, while the granularity of the SVM system is usually CG.

When both granularities have 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 disparity between granule sizes.

When the workload granule is smaller than the system granule size there are two main effects that can produce sharing pattern conversion: *false sharing* and *fragmentation*.

False Sharing

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 with one producer (1P) becomes an induced sharing pattern with multiple producers (MP).

The problems produced by false sharing (such as the ping-pong effect) are partially reduced in SVM systems by using relaxed memory consistency models and multiple writer protocols. Anyway, it remains a performance drawback because the consumers are compelled to require the page produced by multiple producers instead of just one producer.

False sharing mainly affects LRC based protocols, where a page fault in a reader can produce asynchronous communication with several writers. The HLRC protocol mitigates this problem because the writers always update the home node. Thus, asynchronous communication performed by readers only needs to affect the home node. This advantage can become an inconvenience when the home suffers contention due multiple readers requiring updates.

Fragmentation

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 with one consumer (1C) becomes an induced sharing pattern with multiple consumers (MC).

The fragmentation occurs because the readings have finer granularity than writings. In systems with FG granularity (such as SMP systems) this is not a problem because readers only access small blocks of data. In SVM systems, due their CG granularity, reader page faults imply fetching the whole page, even if the data needed represents a small percentage of the page. As a consequence, there is higher bandwidth utilization, which increases the service latency when data is required.

False sharing and fragmentation can appear simultaneously. For example, if both read and write granularities are FG or MG (very common in irregular applications) and the system granularity is CG, it is likely that we obtain a MP-MC pattern from a 1P-1C pattern, thus, showing false sharing and fragmentation [IFT96].

Table 3 shows the inherent patterns and the induced patterns that can arise. Information is shown for write and read operations individually (though not together), because the induced sharing pattern is always MP-MC, and is independent of the inherent pattern. The induced effect was also shown in [IFT96]. Sharing patterns which refer to the pattern on initial loading (0P-1C and 0P-MC) can move to new induced sharing patterns, though induced patterns due to false sharing depend on the sharing unit size. If the size is small, the most likely induced sharing pattern will have just one producer (1P), while if the size is relatively large the probability of moving to a multiple producer (MP) sharing pattern increases.

Sharing pattern conversion is strongly correlated with critical section dilation; i.e., when the induced sharing pattern becomes MP and/or MC, the number of page faults increases (one for each producer or consumer of a given page).

Inherent pattern	Access	Effect	Induced pattern
0P-1C	Read	Fragmentation	0P–MC
	Write	False Sharing	1P–1C or MP-1C
0P-MC	Read	Fragmentation	0P–MC
	Write	False Sharing	1P-MC or MP–MC
1P–1C	Read	Fragmentation	1P–MC
	Write	False Sharing	MP–1C
1P–MC	Read	Fragmentation	1P–MC
	Write	False Sharing	MP–MC
MP–1C	Read	Fragmentation	MP–MC
	Write	False Sharing	MP–1C
MP–MC	Read	Fragmentation	MP–MC
	Write	False Sharing	MP–MC

Table 3 - Transition from the inherent pattern to the induced pattern

4.4. Workload Characterization Analysis

In this section we perform a detailed characterization of our target workloads and also describe the simulation environment. Our characterization is presented around the characterization axes discussed in section 4.1.

4.4.1. Simulation Environment

The execution driven simulator Limes (introduced in section 3.1) has been used to instrument the workload in order to capture memory references. In our experiments we use both the compiler and instrumentation tool provided by the SMP simulator. The Limes simulator uses a modified version of the GCC v2.6.3 compiler using the -O2 optimization flag. The instrumentation tool traps memory accesses by adding augmentation code that calls the memory simulator after each memory reference. We trap synchronization operations by mapping the ANL macros to memory simulator calls. Measurements are taken just after the parallel processes are created, as described in [WOO95]. Table 4 lists the problem size used for each benchmark. These sizes are close to those used in [BIL97]. Every benchmark was executed considering 16 processes.

Benchmark	Problem Size	Execution Cycles
Barnes	8K particles	47441193
FFT	1M points	54372159
LU	512x512 points	48591413
LU-CONT	512x512 points	48589005
Ocean	258x258 ocean	29082695
Radix	4M integers	13389554
Water-NSQ	512 molecules	34211024
Water SP	512 molecules	26965078

Table 4 – Benchmark problem sizes

Experimental results are independent of the system architecture. Independence 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 with a single issue, one instruction per cycle, processor. Processors share memory through a perfect RAM (PRAM) memory model.

Memory access information is collected in traces and analyzed offline. For each access we capture the following information:

- The processor ID
- The memory operation type
- The virtual address of the referenced data
- The simulated time (in processor cycles) when the referenced was issued.

4.4.2. Frequency of Sharing

In this experiment we measure the synchronization period (number of cycles between synchronization operations) present in each workload. To reduce simulation overhead, we measure the results for one randomly selected process. Results for other processes were also checked with negligible differences. Figure 18 shows the average time between synchronization events found in each workload. From the results we can distinguish between two distinct groups of applications: 1) Barnes, Ocean, Radix, and Water-NSQ have a much smaller average synchronization period, and 2) FFT, LU, and Water-SP have a much larger synchronization period.

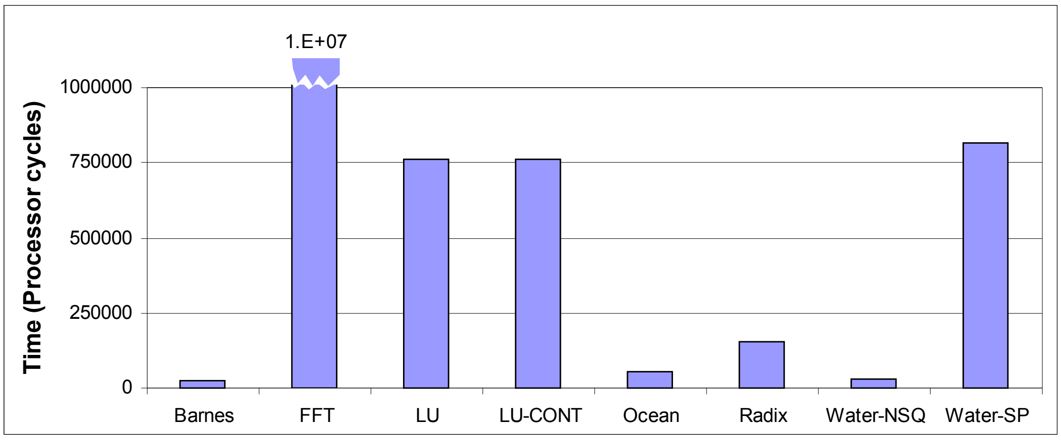


Figure 18 – Cycles between synchronizations

A synchronization operation can take several μsec to be processed in a typical SVM system [ZHO97], which is equivalent to thousands of clock cycles in current microprocessors. Taking this into account, we can characterize the workloads in the first group as having FGS synchronization and workloads in the second group as having CGS synchronization.

These values help to identify critical workload parameters that will affect performance. However, we also want to know if the synchronization period is homogeneous across the whole workload, or instead is concentrated in some interval or "hot spots". To identify how well distributed synchronization events appear in each workload, we calculated both the average and the standard deviation for the elapsed time between synchronizations. Results are shown in Table 5.

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

Table 5 – Average and standard deviation in frequency of sharing

Table 5 shows that the benchmarks have a huge standard deviation. This characteristic indicates that the synchronization period is not uniform and there may be hot spots. To confirm our reasoning, we divide the execution time of each workload into *intervals* of equal length (1 million cycles), and we then measured the synchronization period occurring in each. We chose to use this interval for all the applications because we found it provided us with a meaningful sampling interval.

Results are shown in Figure 19. The points 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., few or no synchronization points (or a negligible number) occur in those intervals. The kernel FFT is not included in Figure 19 because it contains 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.

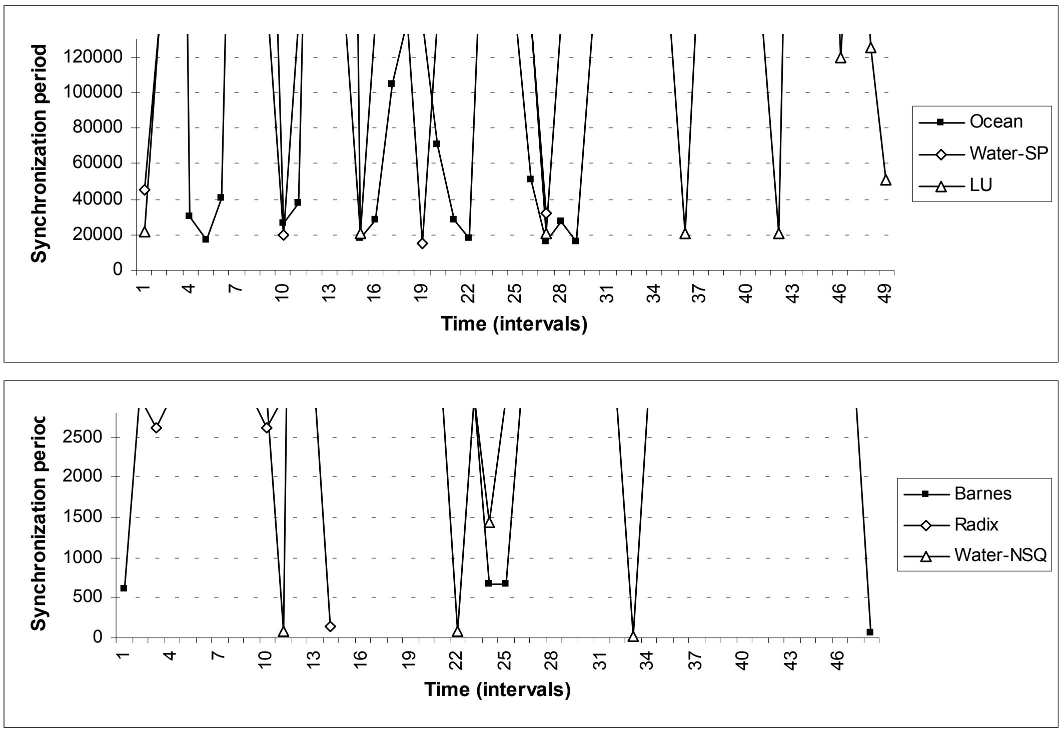


Figure 19 – Synchronization period measured by interval

Based on the synchronization period, we can classify a workload as belonging to one of three groups:

- *Pure CGS*: This category includes those workloads exhibiting a high degree of CGS synchronization during execution. Since these applications possess a large synchronization period, critical section dilation will not appear in the workloads. Some examples of pure CGS are FFT, LU, LU-CONT, and Water-SP. Some applications (e.g., Water-SP and LU) possess a short average synchronization period (less than 50,000 cycles). However, the number of synchronizations involved is negligible. Only the last interval of LU contains a non-negligible number of synchronizations.

- *Medium FGS*: This category includes Ocean, and possesses a minimum synchronization period that is much smaller than that found in pure CGS, but it is still higher (several orders of magnitude higher) than other FGS workloads. Zhou *et al*. [ZHO97] state that Ocean possesses a CGS pattern; however, our results indicate that in some intervals of its execution, Ocean is FGS. For example, the synchronization period in the intervals 5, 15, 16, 22, 27 and 29 has high variance, which means that synchronizations appear in bursts (with less than 100 cycles between synchronizations). This FGS behavior can produce critical section dilation in those intervals.

- *High FGS:* This category includes Barnes, Radix and Water-NSQ, which possess a minimum synchronization period that is much smaller (several orders of magnitude smaller) than medium FGS. Although Radix seems to have the higher synchronization period in two intervals (3 and 10), this is due to just two outlier values that greatly impact on the average. If these values were ignored, the results would be one order of magnitude smaller (368 average cycles between synchronizations instead of 2627 cycles). This situation also occurs in Water-NSQ in the interval 24, where the synchronization period is 1445 cycles, but if we leave out the one anomalous case, the average synchronization period falls to 240 cycles.

The smaller the average period exhibited by an application, the higher the chances that FGS will occur. This non-homogeneity of synchronization intervals indicates critical section dilation problems. Iftode [IFT96] and Zhou [ZHO97] also come to similar conclusions by identifying those parts of the code that consume the greatest execution times when run on SVM systems.

The probability that critical section dilation appears increases with the number of fine-grained synchronizations. For this reason, to see how much critical section dilation can impact performance, we measured the number of synchronizations issued during each interval. Figure 20 shows the results.

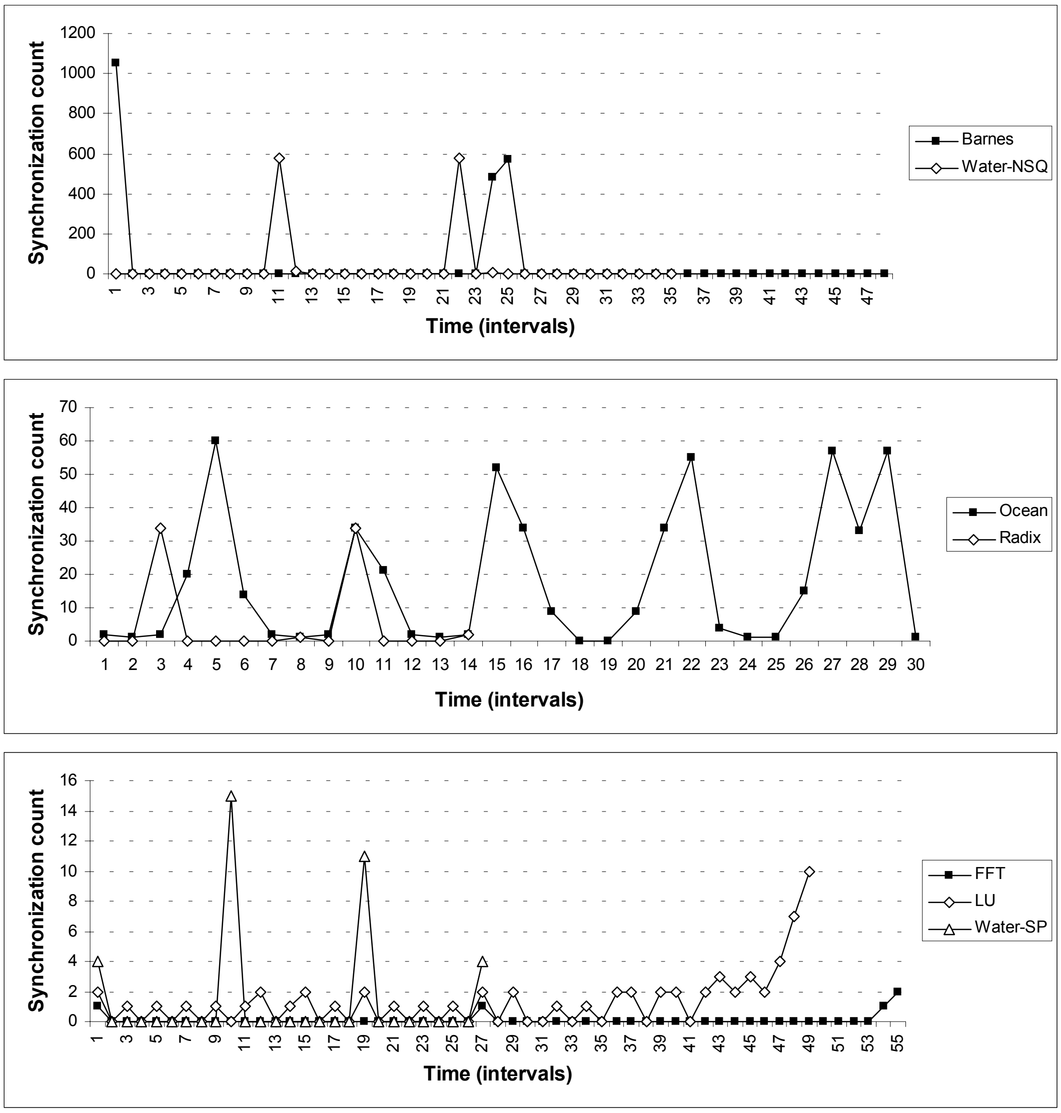


Figure 20 – Synchronization count measured by interval

Inspecting the results, one can observe that Barnes and Water-NSQ will be the applications most affected by critical section dilation because both are FGS and possess a large number of synchronizations in the affected intervals (more than 500). Ocean and Radix have the same problem (but to a lesser extent*)* because their synchronization counts are smaller. In the case of Radix, synchronizations are distributed in the application similarly to Barnes and Water-NSQ, while in Ocean the synchronizations are spread evenly across the trace.

Finally, FFT, LU and Water-SP have few synchronizations, which together with their CGS behavior, confirm their robustness against critical section dilation effects.

4.4.3. Granularity of Sharing

To determine the granularity of sharing, we measure the size of each instance of shared data written between synchronization points (locks, unlocks and barriers). Similar to the characterization techniques presented in the previous section, we divide the execution time of each workload into intervals of 1000000 cycles. For each synchronization point, we measure both the size in bytes and number of the writes performed since the previous synchronization point.

Each continuous area of data written is called a *chunk*. Figure 21 shows the average size of a chunk in bytes, classified by intervals, namely the average granularity. The points in Figure 21 that possess a value of 0 represent intervals where no writes were detected. Based on the granularity results obtained, workloads can be classified in three groups:

- FG: This group includes Barnes, LU, Ocean, Water-NSQ and Water-SP. LU non-continuous and Barnes possess a smaller average granularity value. LU and Barnes do not share chunks larger than 32 and 16 words, respectively. Ocean has a slightly larger than average sharing granularity size, and issues synchronization events in all the intervals. The granularity size is less than 16 words in half of the intervals, and in all the intervals the granularity size is less than 128 words, with an average size of about 10 words. We also found that the sizes of the chunks shared by Ocean show a high variance.

- MG: In this group we only classify the Radix kernel as having MG granularity. This kernel performs synchronizations in only a few intervals (just 4 of 14), but with an average granularity of around 326 words for those intervals.

- CG: This group includes the FFT and LU-CONT kernels. Both workloads exhibit CG granularity in all the intervals where synchronizations were issued. FFT performs very few synchronization operations (5 barriers: 1 during interval 1 without sharing data, 1 during interval 11 and 3 during the last interval).

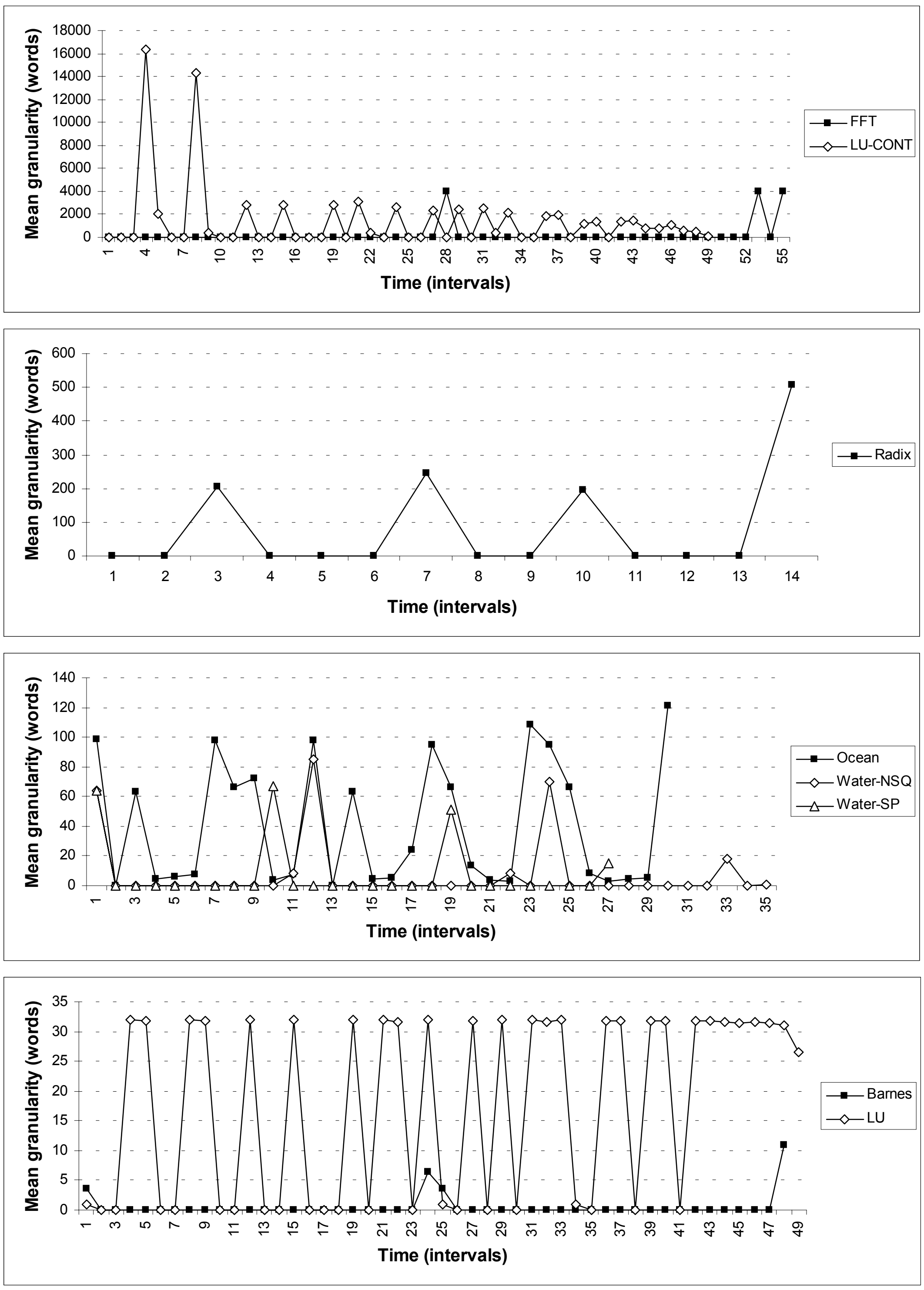


Figure 21 – Mean granularity measured by interval

Results show that most workloads (with the exception of FFT and LU-CONT) have FG and MG sharing granularity. This means that they are likely to have sharing pattern conversion due to both false sharing and fragmentation.

The entropy in the sharing pattern conversion is affected by the amount of written data at a given granularity. Figure 22 plots the amount of written data between synchronizations for each interval. This information complements the information shown in Figure 21 because, in general, as the number of chunks that are written at small granularity increases, the probability that sharing pattern conversion occurs also increases. This means that sharing pattern conversion will have a large impact on performance in Radix and Ocean, because they are MG and FG, respectively, and they share a large amount of data.

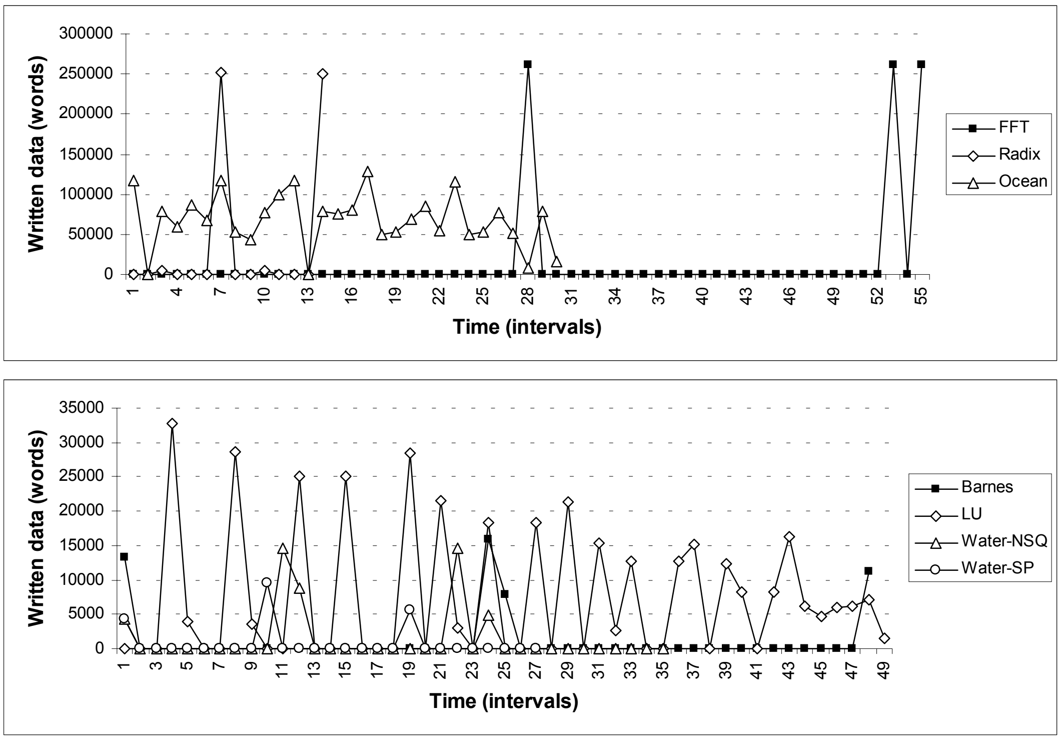


Figure 22 – Total written data between synchronizations for each interval

An update protocol could solve the asynchronous communication derived from sharing pattern conversion because it avoids a lot of page faults; however, the performance of this protocol is limited by the amount of data that the network can process without becoming a contention point. This is a typical trade-off between invalidate versus update protocols.

Looking at the results, one can deduce that an SVM update protocol will achieve poor performance in FFT, Radix and Ocean due to the large amount of written data. Synchronizations in FFT and Radix are concentrated in very few intervals, but the amount of data to be updated is very large. On the other hand, the number of synchronizations in Ocean is distributed among more intervals, but with less data to be updated. This means that using a fast enough network could guarantee no performance loss in Ocean, though it would result in writing more shared data than Radix and FFT together.

LU and LU-CONT share the same amount of data (so they are represented by the same plot, LU), which means that both will have the same performance when using an update protocol. Sharing pattern conversion differences between LU and LU-CONT are due only to granularity differences.

As in Ocean, the number of synchronizations in LU is distributed among many intervals. These numbers are relatively low, but the total amount of data written by each LU application is close to Radix. In contrast, both Water and Barnes share only a maximum of 50,000 words across the workload, which makes them suitable for an update protocol.

4.4.4. Sharing Pattern

This section studies the sharing pattern conversion characteristics for each workload. For this experiment, we explore the sharing patterns of each benchmark varying the sharing unit size. The results show that the sharing patterns in each workload are very sensitive to this parameter. Only two workloads (FFT and LU-CONT) are not affected by changes in granularity size.

Figure 23 presents the sharing pattern frequency obtained by varying the sharing unit size from one word (4 bytes) to one page (4096 bytes). The plotted lines represent the number of bytes exhibiting a given pattern. Bytes are not individually accounted; sharing patterns are considered on a sharing unit basis, so we compute the number of bytes based on the size of the entire sharing unit. 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. On the other hand, if we just counted the number of sharing units, our results would be less accurate since this number is a function the sharing unit size.

For our analysis, both individual points in Figure 23 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 one, or several lines, can help us identify sharing pattern conversion when the sharing unit size changes.

In addition to the pattern classification shown in Table 3, Figure 23 shows the *0C* pattern, which represents those sharing units having no consumers (i.e., non-shared data). If the sharing unit size is 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 24 shows an example. When the sharing unit size is 2 Kbytes (left side), the data in the *B* sharing unit is not shared because there are no consumers. If the sharing unit size is 4 Kbytes (right side), the data is a subset of the shared unit. Thus, fragmentation reduces the non-shared data count. This kind of fragmentation is representative of the behavior in half of the studied workloads (Barnes, Ocean, Water-NSQ, and Water-SP).

The frequency of the 0C pattern is several orders of magnitude higher than that observed in other patterns. This high frequency appears because most of the address space does not have consumers. Thus, it is difficult to observe if all the patterns are plotted on the same figure; however, the slope of the 0C pattern is helpful to understand how the others patterns evolve. Thus, we represent its shape shifted to the X-axis in order to discover its trends.

Studying sharing at the granularity of 4 bytes (one word) unit gives us a better view of the inherent sharing pattern because larger granules may be subject to false sharing and fragmentation. Using larger units can blur the sharing picture. Below, each workload is analyzed.

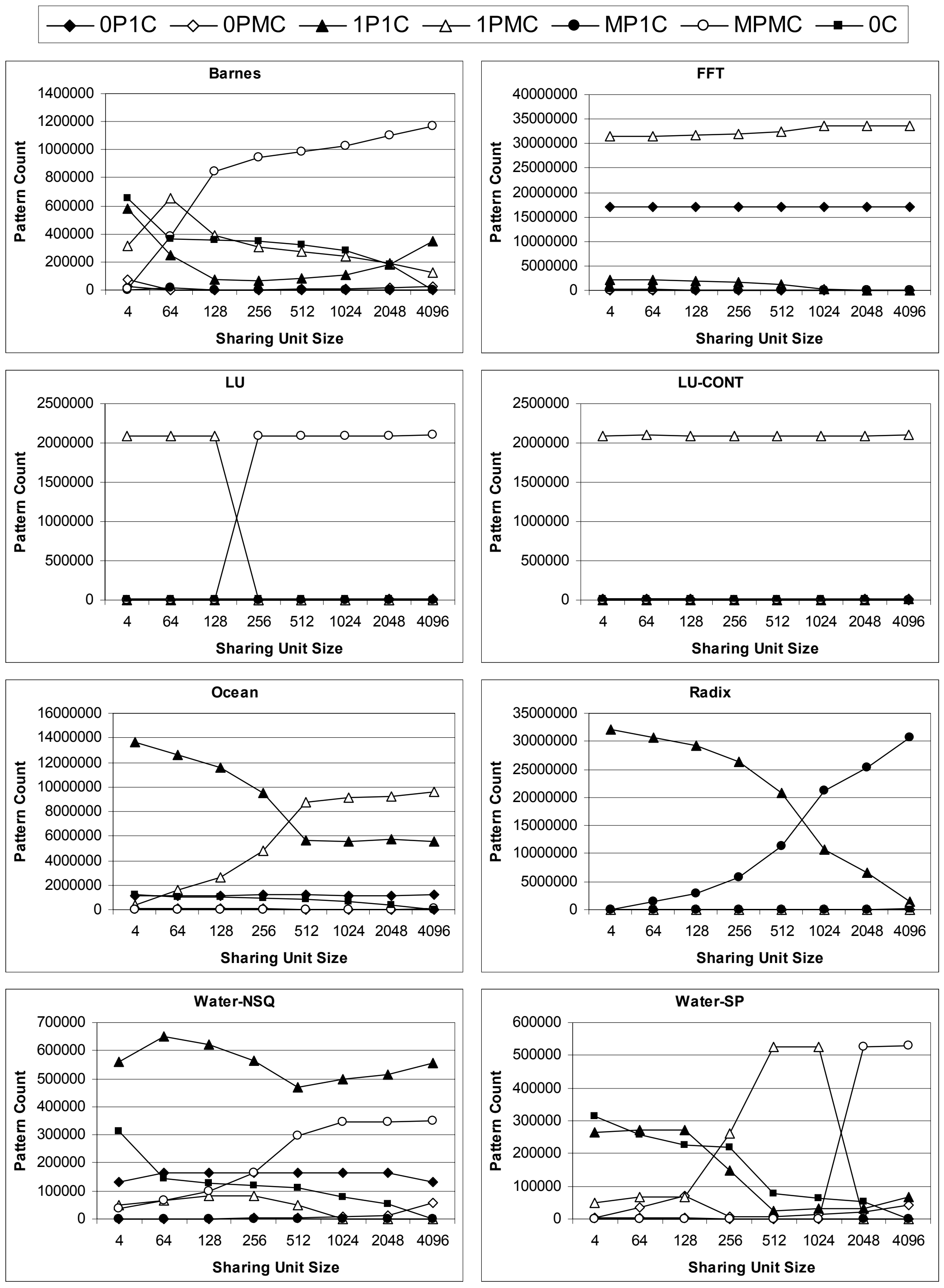


Figure 23 – Sharing pattern count

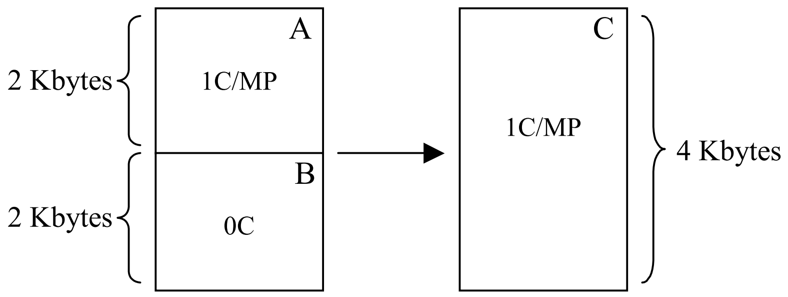


Figure 24 – Fragmentation effect

Barnes

This application shows inherent 1P-1C and 1P-MC sharing patterns, as well as 0C fragmentation. When we increase the size of the sharing unit to 64 bytes, we increase the frequency of 1P-1C (due to fragmentation effects), and in turn we decrease the 0C and the 1P-1C patterns. False sharing also occurs because the amount of MP-MC grows. For intermediate sharing unit sizes (128-512 bytes), false sharing becomes the dominant effect, increasing the MP-MC pattern at the expense of reducing both the frequency of 1P-1C and 1P-MC patterns. For larger sharing units (1024-4096 bytes) there is an accelerated transition. False sharing continues to increase (MP-MC grows), but 1P-1C also increases. It seems that 1P-1C increases at the expense of 1P-MC, but that is not the case. A closer look reveals that the growth of 1P-1C is caused by increased fragmentation (0C).

FFT

This kernel exhibits a fairly constant behavior and so no induced sharing patterns appear. Strictly, a slight growth in the inherent 1P-MC pattern occurs at the expense of reducing the frequency of the 1P-1C pattern. This change can be attributed to the increase in fragmentation, though differences are negligible. One other important sharing pattern present in FFT is 0P-1C, which indicates than a high percentage of the shared data is not written after the initialization phase.

LU

For small sharing unit sizes (4, 64, and 128 bytes), there is just one predominant sharing pattern present (1P-MC). However, this pattern practically disappears for larger unit sizes. This is because false sharing then becomes the dominant effect, increasing the MP-MC pattern at the expense of the 1P-MC pattern.

LU-CONT

This kernel is an optimized version of the studied LU kernel that considerably reduces the amount of false sharing in LU. This explains why we see large changes in LU when sharing units larger than 256KB and this does not occur in LU-CONT. The 1P-1C pattern is the predominant pattern across different sharing unit sizes. Both LU-CONT and FFT are CGS and CG, so they maintain the same sharing pattern for different page sizes.

Ocean

This application shows one dominant 1P-1C pattern for sharing unit sizes less than 512 bytes. The frequency of this pattern progressively decreases, while we see increases in the 1P-MC pattern. This dynamic is due to fragmentation. Sharing patterns stabilize for sharing unit sizes larger than 512KB.

Radix

This kernel shows just one dominating inherent 1P-1C pattern. The frequency of this pattern dramatically drops to zero while the frequency of the MP-1C pattern increases. False sharing causes this change.

Water-NSQ

As the sharing unit size increases both false sharing and fragmentation effects are more accentuated in Water-NSQ. The particular sharing patterns change depending on the sharing unit size. For smaller sizes (8 and 64 bytes), the frequency of the 0P-1C, 1P-1C, 1P-MC and MP-MC patterns increase due to fragmentation (note that 0C decreases). For intermediate unit sizes (128 and 256 bytes), the 1P-1C frequency decreases while the MP-MC frequency increases due to both false sharing and fragmentation. For larger sharing unit sizes, MP-MC stabilizes, while 1P-1C increases (0C decreases). This is due to increased fragmentation.

Water-SP

As in Water-NSQ, the Water-SP application suffers from false sharing for small page sizes and fragmentation for larger sizes. In the first 128 bytes, the original 1P-1C sharing pattern remains constant because very little fragmentation occurs (0C decreases). From 128 bytes to 1024 bytes, fragmentation becomes dominant, increasing the frequency of the 1P-MC pattern, which increases at the expense of the 0C and 1P-1C patterns. For larger page sizes, the impact of false sharing is accentuated, and as a consequence, 1P-MC patterns are replaced by MP-MC patterns.

4.5. Conclusions

The overhead associated with the software management of SVM systems introduces extra latency that can negatively impact system performance. One way to limit this overhead is to design more efficient SVM consistency protocols that reduce overall communication overhead.

To begin to address communication overhead, it is vital to better understand how workload interacts with the system. In this chapter we focused on a number of parallel workload characteristics that can negatively impact the performance of SVM systems. More precisely, we have both identified and quantified the sources of performance loss in each workload.

We first characterized the workload based on three axes related to asynchronous communication latency: the frequency of sharing, the granularity of sharing, and the sharing pattern.

From this characterization, we find that typically the synchronization rate is high in some intervals of execution, and the sharing granule is much smaller than the page size. We also explore dynamic characteristics that are directly related to performance degradation in SVM systems. We have both identified and quantified some useful cause-effect relationships, including: i) a high frequency of sharing rate causes critical section dilation, and ii) a small sharing granule interacts with the sharing pattern, causing sharing pattern transformations. In our analysis, we quantified the severity of critical section dilation and sharing pattern transformation that each workload can incur throughout its execution.

We presented results, varying the page size from 4B to 4KB. We also studied the effects of sampling across fixed intervals. This form of sampling can provide the designer with a more precise view of workload dynamics, which can translate into improved protocol efficiency.

Previous research in the open literature has explored the behavior of parallel workloads as run on SVM systems [IFT96][JIA97][ZHO97]. Iftode *et al.* [IFT96] established a workload taxonomy based on sharing patterns and the granularity of sharing. Jiang *et al.* [JIA97] modified source code based on the described axes to improve the performance of SVM systems. Zhou *et al.* [ZHO97] studied the behavior of workloads running on several protocols and systems, developing a number of rules about the optimal granularity of sharing. In addition, they introduced the concept of the frequency of synchronization in their workload taxonomy.

Our characterization differs in that we gather information about the behavior of the workload independently of the underlying system. This allows us to focus on the workload behavior without any system interference. We concentrate on workload characterization and how the workload characteristics impact the performance metrics.

Chapter 5

The HLRC-DU and the HLRC-CU Protocols

As it has been widely discussed in previous chapters, asynchronous communication is one of the main performance drawbacks in SVM systems. In Chapter 4 we detailed the factors why asynchronous communication rises in these systems through investigating the different workloads. We concluded that, as a consequence of the large granularity supported by SVM systems (the page) the false sharing and fragmentation phenomena (sharing pattern transformation) appears, considerably increasing asynchronous page requests and contributing to the dilation of critical sections.

From the explained above, one could deduce that protocol performance can be improved if their design is addressed either to reduce false sharing and fragmentation effects; i.e., to reduce the amount of asynchronous page requests. This can be accomplished by reducing the amount of invalidations.

We explore this idea by using the Home Lazy Release Consistency (HLRC) protocol as the baseline protocol. We propose an improved version of this protocol: the HLRC Diff Update (HLRC-DU) protocol. This protocol updates data through write notices when diffs are smaller than a given threshold. In this way, the write-notices update the data instead of invalidating the page, so we refer them as write updates. The purpose of this proposal is to reduce the amount of invalidations in order to avoid asynchronous requests to page homes.

Diffs can perform over one or more continuous written areas in a page. We refer to each continuous area as a written chunk – or simply, a chunk. We detect (see section 5.1) that a huge percentage of diffs perform over a single chunk, so showing spatial locality. One advantage of these kinds of diffs is that they can be easily detected and calculated by a simple hardware, as we will discuss later. For this reason, we propose an alternative protocol, the HLRC Chunk Update (HLRC-CU) that uses specific hardware to automatically detect and calculate those diffs.

This chapter is organized as follows. Section 5.1 performs a preliminary study to find the potential benefits of write updates. Section 5.2 explains the proposed protocols HLRC-DU and HLRC-CU, studies the hardware complexity of the specific hardware used in the HLRC-CU protocol, and compares both proposals. Section 5.3 studies the sensitivity of the HLRC-DU protocol to the threshold size used for performance tuning. Section 5.4 compares a hardware technique used in the open literature [BIL98] against the HLRC-DU protocol. Finally, section 5.5 presents some concluding remarks.

5.1. A Preliminary Study

One straightforward solution for the protocol designer in order to reduce asynchronous communication is to update, instead of invalidating data, as explained in section 2.3.1. However, update techniques have a tradeoff between the benefits obtained for reducing asynchronous communication and the consequent increase in network utilization. To maximize the benefits in this tradeoff, updates have to be a) small enough to fit in the network bandwidth without causing congestion, b) useful enough to reduce the asynchronous communication in a high percentage.

An idea derived from this tradeoff is to update only small diffs, given that the granularity of sharing is in general small in parallel workloads, as shown in Chapter 4. The protocol designer could 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, however, it can reduce a high percentage of asynchronous communication both in LRC and HLRC protocols as small granularity of sharing is frequent enough. In addition, HLRC contention in home nodes is reduced if all the updates for their pages are sent along with write notices. We will refer to a write notice plus its associated diff as a write update.

On the other hand, diffs will be likely to perform over continuous words (ranging from only one word to the complete page) due to spatial localities (we will refer to a range of continuous written words as a *page chunk* or simply a *chunk*). This locality could be used to reduce diff calculation overhead by detecting patterns of continuous writes via small hardware. When that situation is detected, it can be notified to the OS by indicating the address and size of the page chunk that was written. This optimization will reduce the elapsed time spent on diff calculation at the nodes.

In this section we examine whether the discussed ideas could be useful for improving performance by reducing asynchronous communication and diff calculation costs. We use a baseline implementation of the HLRC protocol described in the next section to check the new proposals.

5.1.1. The Baseline HLRC Protocol

The baseline protocol is based on the implementation of the HLRC protocol (see section 2.3.4) proposed by Zhou *et al.* in [ZHO96]. In our implementation, each page has a home node to accumulate the diffs calculated by the writer nodes. When a node receives a write notice for a page, it marks its local copy as invalid and asks the home for an up-to-date version of the page, whenever any local process tries to access it. The page homes are selected by means of a module function of the most significant bits of their page addresses.

Each write notice contains the identification of the writer process, the timestamp of the write, and the page address. There is no garbage collection of globally known write notices. Write notices are only sent to a given process if it acquires a semaphore, or to all processes when they reach a barrier.

Once a process releasing a semaphore or a barrier sends the write notices to the acquirer, it immediately sends to the homes those diffs produced by its previous writes in order to keep the homes updated. The acquirer will invalidate the pages corresponding to the addresses in the received write notices. Then, if the acquirer accesses the invalidated page, a page fault will occur. Consequently, the protocol asks the home for an up-to-date page. If the home is not updated, the diff will arrive immediately because it was sent just after the write notice. Figure 25 shows how modifications of the writer node (node *A*) on page *P* arrive at the home node of the page (node *B*) before the invalidated node (node *C*) can ask for an up-to-date copy from the home node (node *B*).

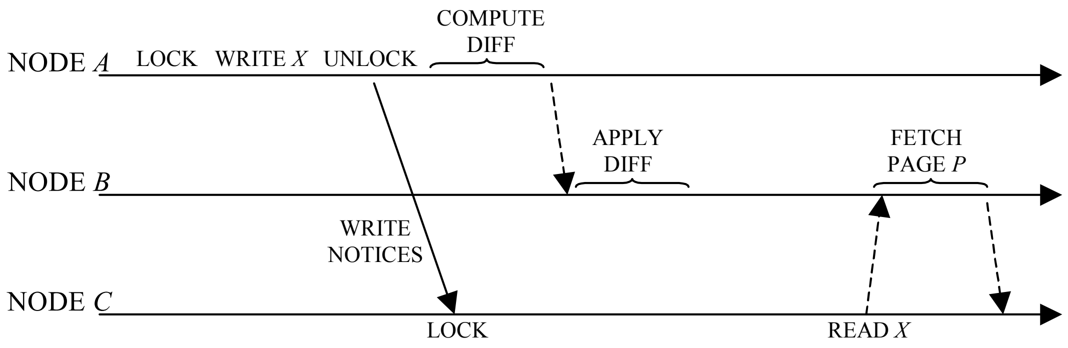


Figure 25 – Baseline protocol example

Each semaphore and barrier has a home node selected by using a module function. The semaphore home node queues the acquire requests and remembers which was the last node to release the semaphore. This allows it to forward those requests to the last releaser when needed. Then, the releaser will directly send the write notices to the acquirer node without crossing the home.

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

5.1.2. Simulation Environment

We use the LIDE execution driven simulator described in Chapter 3 to evaluate if write updates could benefit the performance and all the proposed protocols in this chapter. We compile the running benchmarks with a modified version of GCC v2.6.3, applying the O2 optimization flag.

The modeled architecture consists of a single cluster composed of 32 monoprocessor nodes connected through an overclocked (1 gigabit per second) Ethernet network. The contention of the network is also modeled. The internal clock rate of each processor works at 1 Ghz (1 cycle = 1 nanosecond).

The load in each node consists of the parallel application plus the operating system overhead introduced by the memory consistency model. A two level cache hierarchy is simulated for the memory accesses. In the first level, we assume a 64KB cache. In the second level, we assume a 1 MB cache. Both caches are direct mapped. Hits at the first level cache take one cycle. In the case of a miss occurring in the first level, the second level cache solves it, taking 8 cycles. If both caches have a miss, the block is loaded from memory, taking 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, a check is made to see if there is any request pending from a remote processor. If so, those requests have higher priority than the local requests, and each takes 10 μsec before issuing a response.

In addition to the previous times, diff creation in writer nodes and its application in home nodes takes time that grows linearly with the page size (4 cycles per word). As page size is assumed to be 4KB and word size 4bytes, all protocols take 4096 nanoseconds in either creating or applying the diff. This overhead is not present when copying a single page because the model assumes that a DMA device performs this task.

To carry out our experiments we use the same benchmarks as in Chapter 4. Table 6 shows the problem size used for each benchmark. The problem sizes match those used in the open literature to perform similar studies for workload characterization.

Benchmark	Problem Size
Barnes	2K particles
FFT	256K points
LU	512x512 points
LU-CONT	512x512 points
Ocean	130x130 ocean
Radix	1M integers
Water-NSQ	512 molecules
Water-SP	512 molecules

Table 6 – Benchmark problem sizes

5.1.3. Experimental Results

Write notices are received when a process acquires a semaphore or a barrier. We instrument the baseline protocol to obtain the size of the diff produced by the write notice when they are sent. Then, we measure the number of sent diffs of each size. This helps us to check the potential benefits that we could achieve by avoiding the write notices that produce the smallest diffs.

Table 7 represents the distribution per processor of those sizes. As can be seen, most diffs are relatively small (about the 58% contain less than 128 words), except in FFT. To update them seems to be a good compromise between the expected reduction in asynchronous communication and the induced traffic. Thus, we choose 128 words as an experimental threshold value for this preliminary study.

Note that in order to update most diffs in FFT we should choose a much higher threshold. In addition, the distribution of diff sizes in LU and Ocean also seems to suggest a higher threshold. We explicitly avoided this, in order to keep network traffic bounded. Section 5.3 contains a detailed analysis for determining the optimal threshold value.

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 7 – Distribution of diff sizes

As explained at the beginning of this section, writers often update only a continuous chunk of data in a page, due to data localities. The diffs produced by this pattern will be composed of only one chunk of data and could be calculated by simple hardware snooping of the data bus. This will improve performance by reducing diff calculation costs. To check if this continuous pattern is frequent enough, we split the diffs into four categories:

1. Those that can become write updates because they are smaller than the threshold value, and are composed of just a single chunk (Update-single).
2. Those that can become write updates because they are smaller than the threshold value, but are composed of multiple chunks (Update).
3. Those greater than the threshold so they cannot become write updates (Invalidate).

Figure 26 shows these results. There is a huge percentage of diffs belonging to the two first categories. On average, around the 56% of the diffs will become write updates in a protocol that updates all diffs smaller than 128 words using write updates. If the protocol only updates continuous chunks, the percentage falls to 36%. Thus, if we choose only to update continuous chunks the benefits of reducing diff calculation time is at the expense of increasing the percentage of asynchronous communication. This is a tradeoff that we will explore in section 5.2.3.

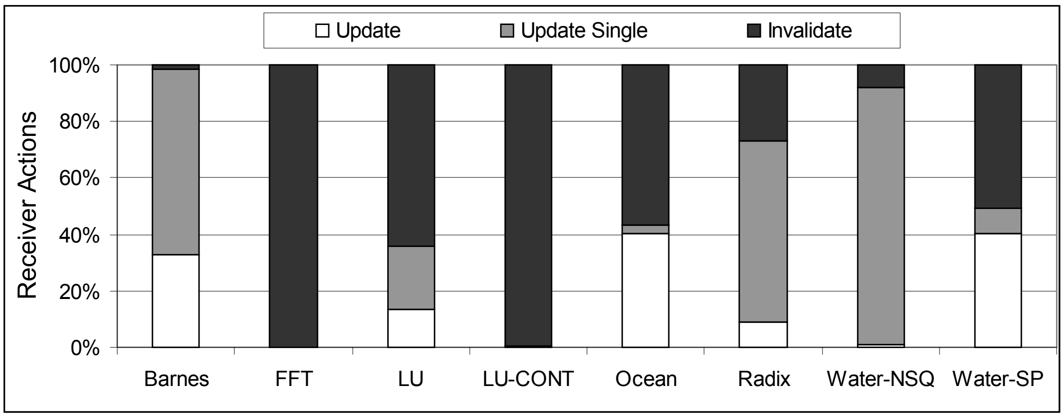


Figure 26 – Breaking down received write notices. Legend: Update) Diffs smaller than 128 words; Update-Single) Continuous diffs smaller than 128 words; Invalidate) Diffs greater than 128 words.

Note that in both FFT and LU-CONT there are not write notices to be updated (Update or Update-single); thus, the behavior of both approaches will be similar to the baseline protocol. On the other hand, Barnes and Water-NSQ update a great amount of diffs below 128 words. Other workloads lay between the two extremes.

The benchmark Water-NSQ also has more spatial locality than the other workloads under this threshold. Therefore, it is expected that both approaches will perform equally well in this workload. This occurs also in Radix, but to a lesser extent. The other side is represented by the benchmarks Ocean and Water-SP, which do not show spatial locality on small diffs, and Barnes and LU, which show both high percentages of continuous and non-continuous diffs. It is difficult to predict how these last workloads will behave when running under the proposed protocols due the trade-off mentioned above.

5.2. Proposed Protocols

In the previous section we find that many diffs are relatively small, and in addition, due spatial localities they are composed by one chunk of continuous words. The proposed HLRC-CU protocol just updates this kind of diffs, while the HLRC-DU protocol updates all diffs. In both cases, diffs are only updated if they are smaller than the threshold.

As the threshold approaches zero, both protocols perform more invalidation actions and their behavior is close to the baseline HLRC behavior. On the other hand, they behave like pure update protocols when there are no threshold restrictions.

5.2.1. The HLRC-DU Protocol

The HLRC-DU protocol detects diffs smaller than a threshold size and injects them as updates via write notices, which become write updates. Larger diffs are only sent to the home nodes and the associated write notice invalidates the page as in the baseline HLRC protocol. No threshold will induce network congestion in some applications; consequently, we use the threshold to control the load injected in the network by write updates.

Figure 27 shows how modifications of the writer node (node *A*) on page *P* arrive directly to node *C* when entering the semaphore. In this way, the home node is not interrupted from workload computation. The page fault (on *READ X*) in node *C* is also saved.

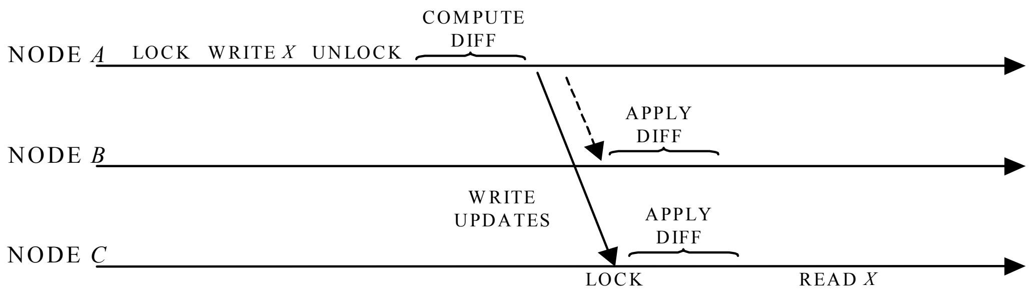


Figure 27 – HLRC-DU protocol example

If during the acquisition of a semaphore, a process receives both an update and invalidation for the same page, the page becomes invalid. In such a case, the protocol invalidates the node; as a consequence, the node will need an up-to-date copy from the home, which will contain all the modifications. Thus, there is a need to update the home (node *B*) as occurs in the base HLRC protocol.

5.2.2. The HLRC-CU Protocol

HLRC-DU protocol emphasizes every diff smaller than the threshold; thus it can contain just a single chunk or multiple chunks. On the other hand, the HLRC-CU protocol emphasizes only those diffs containing a single chunk. This is because, as mentioned in section 5.1.3, a large number of page modifications perform over a single chunk and these diffs are easy to detect and calculate with simple hardware. In this sense, the HLRC-CU protocol tries to benefit from the workload behavior.

Diffs are composed of one or more chunks. For each chunk, the information sent is composed of i) the initial address, ii) the total data size, and iii) the data to be updated. The first two components of that information are an unavoidable overhead when updating data.

The minimum overhead occurs when the diff is composed of just one chunk. In addition, that situation is easily detectable both by software and simple hardware. If continuous writing patterns are frequent enough, a simple table hardware will take advantage of the situation, because most diffs will be calculated by hardware; thus, saving software overhead.

To accelerate the HLRC-CU protocol we introduce a *Page Information Table* (PIT) that is a hardware table for detecting chunk areas at run-time. In this sense, the hardware alleviates the operating system from this task. Figure 28 shows the information structure of this table. A written chunck is defined by its initial and final address and one array of continuous page data. Table complexity is negligible when compared with the generic diff calculators used in [BIA96][BIL98]. The PIT is a fully associative table working as a small cache indexed by the page address.

The PIT works as follows. Initially the node fetches the page; then when the processor issues a write operation a new entry for that page is created, both the *first access* and the *continuous bit* are set, and the address is placed both in the *initial access address* and in the *last access address* fields. Then, each time a new local write is performed, a substractor calculates if the current address is continuous with the last stored address (less or equal to a word width). If not, the continuous bit is reset, and it remains cleared until an unlock or a barrier operation is performed. Then, the OS retrieves and resets the full PIT information.

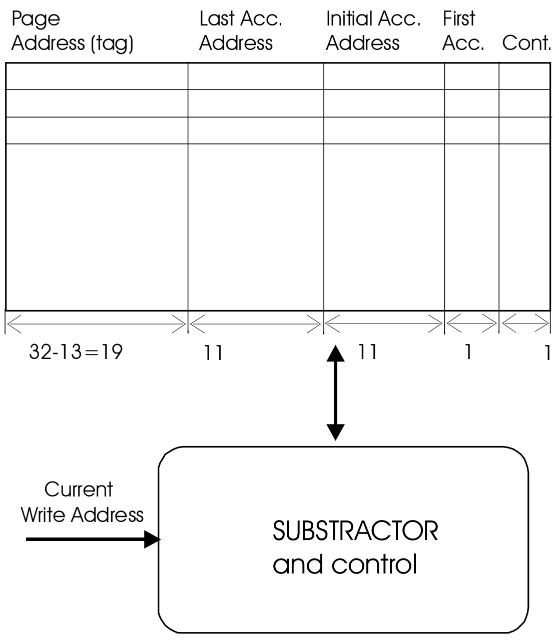


Figure 28 – Page information table structure

The OS can use the information provided by the PIT (initial access address and last access address) to embed the continuous chunk of data in write updates. Note that this task can be done with little OS intervention, through DMA and so further accelerating the creation of protocol messages.

PIT Hardware Overhead

The PIT behaves as a fully associative cache indexed by page address. Ideally, the PIT must be large enough to store all the page entries of those pages written by each process before it releases a semaphore or a barrier. At that moment, the entire table contents will be flushed to memory by the protocol. To choose the dimensions of the table, we measure the maximum number of diffs sent when a release occurs, that is the number of pages written after the previous release.

Results are shown in Table 8. We found that the maximum size is similar between processes when running the same workload. Thus, only maximum values between processes are shown. When the PIT size is not large enough to fit all the page addresses (of the input data set), those entries could be clearly detected by software or treated as in the baseline HLRC protocol. In accordance with the obtained results for the size of the workloads used, the PIT only needs 1024 lines, which means little hardware is needed.

Benchmark	Diffs updated
Barnes	44
FFT	65
LU	128
LU-CONT	32
Ocean	37
Radix	979
Water-NSQ	16
Water-SP	18

Table 8 – Maximum number of required entries in the PIT

5.2.3. HLRC-DU versus HLRC-CU

In this section we compare the performances of both proposed protocols. This comparison also checks if the benefits of reducing diff calculation costs (HLRC-CU) are higher than extending the range of write updates to non-continuous diffs (HLRC-DU).

As explained in section 5.1.3, an experimental threshold value limiting the maximum size of diffs sent with the write updates is needed to avoid network congestion. From the results in section 5.1.3, we chose an experimental threshold of 128 words (512 bytes). To check if this threshold value also works for the HLRC-CU protocol, we gathered the distribution of diff sizes. Figure 29 shows the values from diffs obtained under the HLRC-CU algorithm without threshold restrictions. As one can see, most are smaller than 128 words. Thus, we also chose 128 as the threshold value for the HLRC-CU.

Table 9 shows the total number of diffs being updated by the HLRC-CU protocol as well as the increment that the HLRC-DU protocol produces. We show the total increment as well as a break down into smaller intervals. The total increment ranges from no increment in LU-CONT to more than one order of magnitude in Ocean, LU and Water-SP.

Diff Size Range	**Benchmark**							
	Barnes	**FFT**	**LU**	**LU-CONT**	**Ocean**	**Radix**	**Water-NSQ**	**Water-SP**
Total HLRC-CU	14600	1681	9247	6529	860	20222	9525	186
]0,1]	0	0	0	0	0	0	0	0
]1,2]	0	0	0	0	0	0	0	0
]2,4]	3489	0	0	0	0	0	0	0
]4,8]	795	0	0	0	1578	0	0	0
]8,16]	736	0	0	0	1102	1	4	256
]16,32]	1303	0	0	0	998	6	14	2
]32,64]	261	0	0	0	4526	322	77	560
]64,128]	631	0	5600	0	3301	1982	28	17
]128,256]	287	0	23488	0	3689	38	70	78
]256,512]	40	0	3002	0	2166	17	36	980
]512,1024]	0	1440	0	0	8760	1	704	0
]1024,2048]	0	0	0	0	1560	0	0	0
Total HLRC-DU	22142	3121	41337	6529	28540	22589	10458	2079

Table 9 – Increment of diffs updated by HLRC-DU versus HLRC-CU

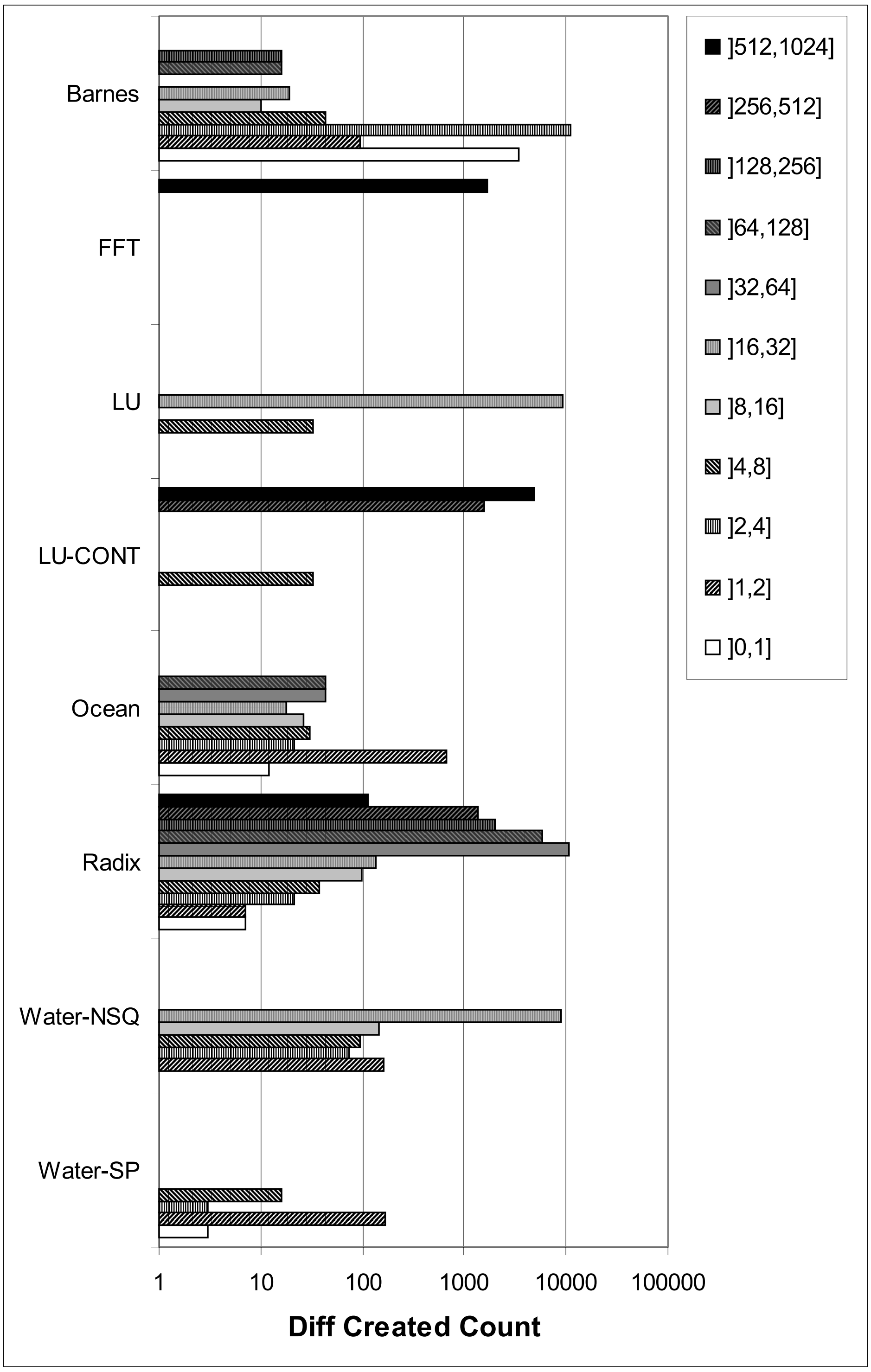


Figure 29 – Distribution of diff sizes

Figure 30 summarizes the speedup results for both proposed protocols over the baseline protocol. The results show the large gains that both the HLRC-DU and HLRC-CU protocols achieve. As explained in Figure 26, both protocols obtain the same results in FFT and very similar results in LU-CONT. The best speedups are achieved by those workloads that update more diffs under the HLRC-DU protocol (Barnes, Ocean and Water-NSQ).

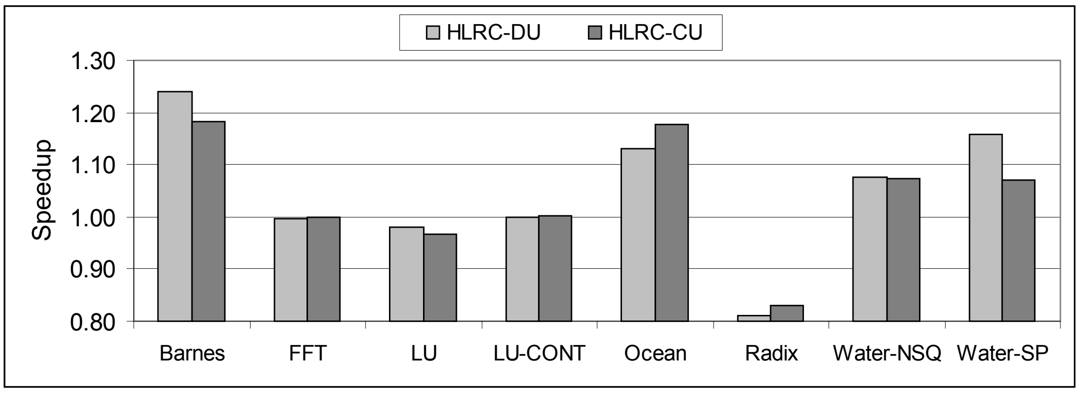


Figure 30 – Speedup over the baseline protocol of the proposed protocols

In Radix, however, the updates have an adverse effect, because they increase the utilization of the network, producing a bottleneck. Thus, once again, we emphasize the compromise between the number of write updates and network congestion. We will explore this tradeoff in section 5.3. In general, the results show that most writes perform on small *diffs* that can be attached to write notices and updated by the receiver node. As they are small, they do not suppose a high impact on bus utilization.

Both protocols achieve, on average, on the average, a speedup of nearly 5% over the baseline HLRC protocol. This means that the proposed PIT, although useful for reducing diff calculation overhead, also considerably reduces the range of potential write updates. Consequently, it would be interesting to have a hybrid protocol that updates in the same way as the HLRC-DU all those diffs smaller than the given threshold, as well as using the PIT for those diffs composed of only one chunk.

The 5% average can seem small but it is important to notice that the benefits obtained across the benchmarks have a high variance. As expected, FFT and LU-CONT behave like the baseline, but 4 of the 8 benchmarks perform 8% more quickly. In some cases (Barnes, Ocean, Water-SP), the speedup surpasses the 15%, reaching the 25% in Ocean. The negative results found in LU and especially in Radix are explored in section 5.3.

5.3. Sensitivity to the Threshold Size

In section 5.2 we proposed the HLRC-DU memory consistency protocol and found that it performs better than the baseline HLRC protocol. The HLRC-DU protocol updates data through write notices when diffs are smaller than a given threshold. In this way, the write-notices update the data instead of invalidating the page, so we refer them as write updates. The goal is to reduce the number of invalidations in order to avoid asynchronous requests to page homes.

The threshold size imposes a trade-off between bus utilization and the reduction of asynchronous communication. In this section we study the sensitivity of the HLRC-DU to this parameter. For this study, six different threshold values ranging from 16 to 512 words have been explored, as well as a seventh options with infinite threshold or no threshold restrictions.

5.3.1. Experimental Results

The advantage of a write update over a write notice is that it can avoid some page requests to the home. So, the distributions shown in Table 7 were an upper bound on the number of requests that can be avoided. These results indicate that the benchmarks are able to avoid more requests in Barnes, Radix and Ocean than in LU-CONT and FFT, for example.

Table 10 shows the percentage of home page requests saved by write updates considering different values for the threshold in HLRC-DU. Values from the last column of Table 10 are lower than 100% due to the initial home page requests. Results for our benchmarks fall into three categories:

1. Benchmarks highly sensitive to the threshold value (Barnes and Water-NSQ): For example, Barnes shows savings from 70% (with threshold 16) to 87% (with threshold 512).

2. Benchmarks slightly less sensitive (Ocean, Water-SP and LU): Their results are consistent with the results shown in Table 7.

3. Benchmarks insensitive to the threshold value (saving less than a 4% of the home page requests, FFT and LU-CONT): Due to the low values shown in Table 7, the performance of these benchmarks in the HLRC-DU protocol will not differ much from those found for the baseline protocol when using small thresholds, for example, 256 words.

Radix is an exception because its results are non-coherent with those presented in Table 7. Radix performs in this way because many updates that could save page requests are cancelled by invalidations. In addition, many page requests are produced by the initial loads and therefore cannot be saved.

		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%

Table 10 – Percentage of saved home page requests varying the threshold size

Comparing Table 7 with Table 10, it can be seen that larger write updates tend to save more home page requests across the benchmarks. This is because larger diffs present less false sharing, i.e. they leave fewer places in the page that could be invalidated. Although larger diffs hugely reduce the number of requests, we must trade off this reduction in network traffic to improve the system performance.

Figure 32 shows the network utilization in the baseline HLRC model for each benchmark used while varying the threshold size. While in most cases, under a large threshold value, the injection of write updates considerably increases the network traffic (e.g. FFT, LU, LU-CONT, Ocean and Radix), it is remarkable that no benchmark increases the network traffic when using a small threshold value.

While it is unclear from Figure 32 when the network saturates, this can be clearly observed in Figure 31. This graph summarizes the speedup results for HLRC-DU over the baseline protocol, while varying the threshold size value. When the speedup increases, it is obvious that there is no contention point (e.g. Ocean, Barnes, Water-NSQ and Water-SP). In general, the network becomes saturated when using a threshold of 32 words. Below this threshold, our protocol behaves close to the baseline protocol for FFT, LU, LU-CONT and Radix.

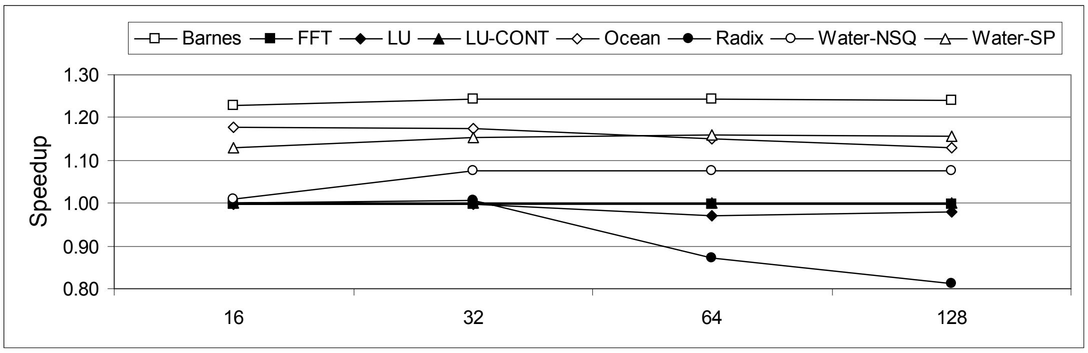


Figure 31 – Speedup relative to the baseline protocol varying the threshold size

In general, our experiments show that the threshold values that achieve the best speedup are those in the range of 32 to 128 words (depending on the workload used), and the performance of any benchmark does not drop when using a threshold value lower than 32 words.

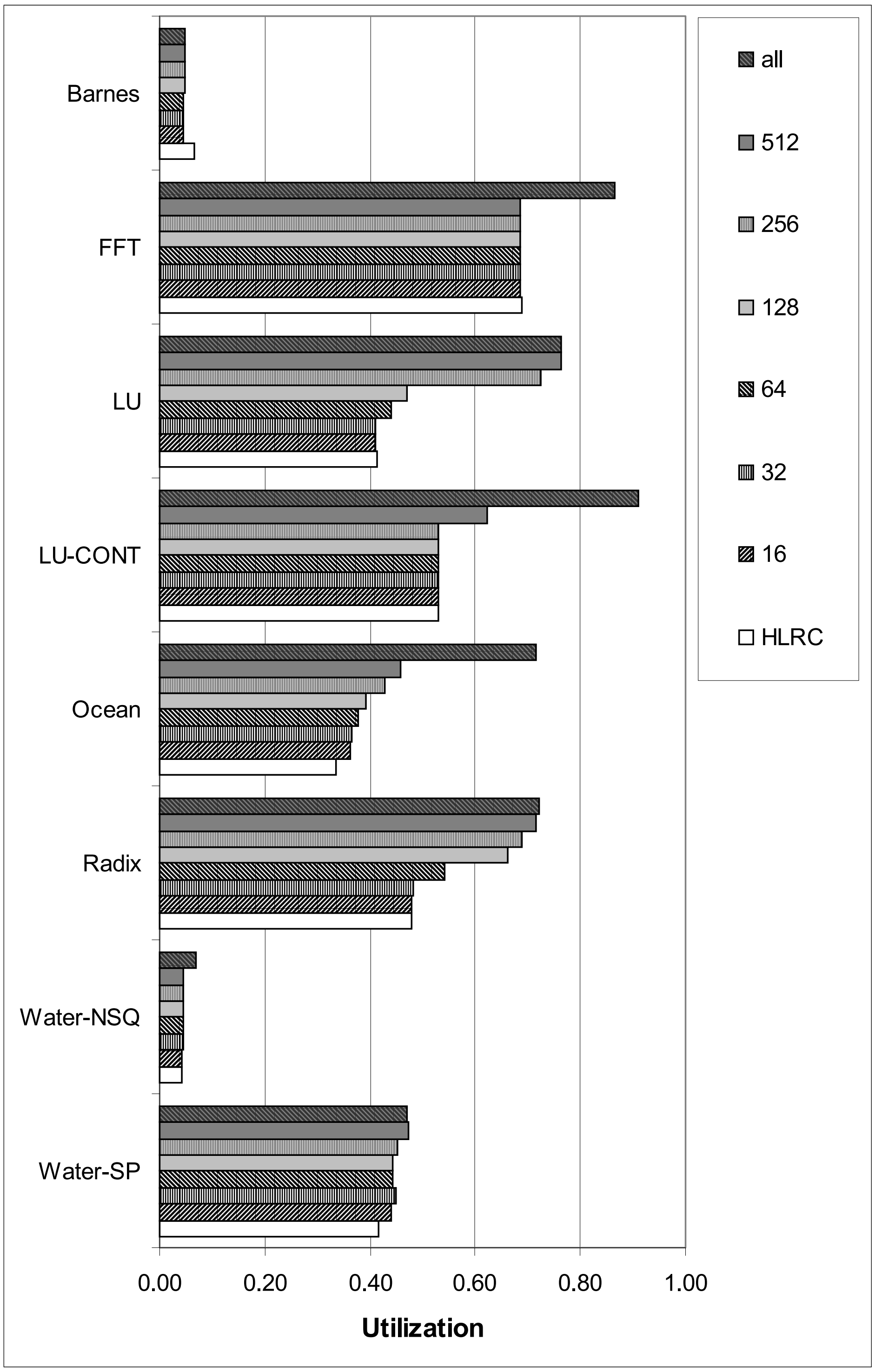


Figure 32 – Network utilization varying the threshold size

5.4. Performance versus Hardware Techniques

In previous sections we saw how the HLRC-DU performs regarding to the baseline HLRC protocol and its sensitivity to the threshold. In this section we compare the HLRC-DU with the best threshold to a hardware approach found in the literature [BIL98].

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. In this way, pages are served automatically and the home node is uninterrupted. We modeled this feature in the simulator by assuming that in these kinds of systems the page is served in zero time. Figure 33 presents the speedup of HLRC-DU with a 32 threshold and the baseline protocol with the hardware that automatically server pages without asynchronously interrupting the processor.

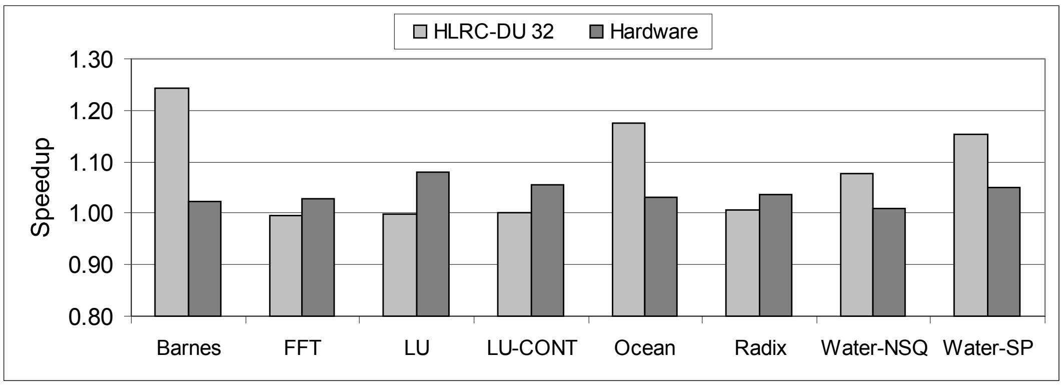


Figure 33 – Speedup of the HLRC-DU protocol using a threshold size of 32 words relative to the baseline protocol

Results show that specific hardware performs better than HLRC-DU only in those cases where HLRC-DU does not obtain benefits relative to the baseline protocol. In all other cases, HLRC-DU performs better than 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 saved interrupt improves the performance.

Note that the compared hardware and the HLRC-DU protocol are compatible. In fact, they are complementary approaches. Table 11 shows the results of combining both approaches. In some workloads like Barnes, Ocean and Water-SP, the HLRC-DU protocol and hardware sum up performances. In others, the advantage of just one approach remains. There are only two workloads where HLRC-DU impacts negatively together with the hardware (FFT and LU-CONT), but in those two cases the negative impact is less than 2%.

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

Table 11 – Speedup relative to the baseline protocol

5.5. Conclusions

Open SVM systems research tries to avoid or reduce asynchronous communication whenever possible because it is one of the main sources of performance loss. In previous chapters we studied why the workloads produce a large amount of asynchronous communications in SVM systems and we measured these amounts. In this chapter we also explored the diff sizes and we showed that most of them are really small. From this observation, we propose two new protocols to take advantage of this empirical remark. To reduce asynchronous communication and contention at home nodes, the proposed protocols attach the small diffs to write notices in order to be updated by the receiver node. We refer to this metadata (write notices plus associated *diff*) as write updates.

The goal of the protocols is to reduce the asynchronous requests to the page homes caused by write notices. This is accomplished by receiving a write update instead of a write notice. In this manner, the page copy remains valid and the asynchronous request is saved.

This approach introduces higher network bandwidth, because write updates are larger than write notices. To avoid the network becoming a bottleneck, we chose an empirical threshold. *Diffs* whose size is greater than the threshold are not sent and the protocol proceeds as the baseline protocol.

The protocols proposed and detailed in this chapter have been:

1. The HLRC-DU protocol, as a version of the baseline HLRC protocol, where the writer node sends updates instead of invalidations when diffs smaller than an experimental threshold value (128 words) are detected. This protocol can convert about 21% of the invalidations in the baseline protocol to updates, saving significant asynchronous communications.

2. The HLRC-CU protocol, which is proposed due to the large number of writes performed over continuous areas detected in the HLRC-DU. The HLRC-CU only sends those diffs through write updates. This protocol saves, on average, about 13% of the asynchronous communication. Its main advantage is that diff calculation can be easily hardwired by a simple hardware table, discussed earlier. This specific hardware has less complexity than some alternative proposals that can be found in the open literature [BIA96][BIL98].

There is a trade-off between asynchronous communication savings and the acceleration of diff calculation. To investigate this trade-off, we compare both protocols to check which one achieves the best performance. Both protocols achieve better performance than the baseline HLRC. The experiments in section 5.2 show that in four of out of eight workloads the speedup is over the 8% and in two workloads exceeds 15%. On average the speedup achieved is a 5% faster than the baseline protocol. In general, the performances of both protocols are similar. This means that asynchronous communication savings are more important for performance than diff calculation accelerators.

Because asynchronous communication and network utilization are strongly dependent on the threshold value used, we also check the optimal threshold that maximizes system performance. Our results show that the HLRC-DU protocol can save about the 50% of request petitions to the page homes in some of the benchmarks, when working under their optimal threshold. For small threshold values, the network traffic only increases marginally while the speedup increases, and in some cases it reaches the 20% versus the HLRC baseline protocol. Although the optimal threshold value is workload dependent, it does not surpass 32 words, a small value that avoids the network becoming a bottleneck.

Chapter 6

Conclusions

SVM systems are an economic and flexible way to run parallel workloads although their performances are still far from those achieved by hardware systems. This thesis has focused on how this performance gap can be reduced; therefore the study has concentrated on performance loss. The main performance drawbacks in SVM systems are related with asynchronous communication. Recent research often proposes solutions involving specific hardware to avoid this kind of communication.

This dissertation analyzes parallel workloads characteristics in order to identify the sources of high latencies appeared in asynchronous messages. The main goal of this analysis is to help protocol designers reduce these latencies and investigate when asynchronous messages can be avoided.

From this study new SVM protocols based on the HLRC protocol have been designed as proposed. These protocols are the HLRC-CU and HLRC-DU. Both reduce asynchronous communication by using pure software mechanisms that take profit of the workload characteristics observed in the characterization study. The HLRC-CU protocol also introduces specific hardware for accelerating remaining asynchronous communication.

In order to make performance evaluation studies in these kinds of systems a simulation environment, called LIDE, has been specifically developed. It has been the test bed where the protocols discussed in this dissertation have been developed and checked.

Next sections summarize some of the most relevant conclusions extracted from the research made.

6.1. Workload Characterization in SVM Systems

The performance drawbacks that parallel workloads like SPLASH-2 suffer on SVM systems are mainly caused by the assumptions made by programmers. Because generic languages and compilers are unable to support all the various kinds of parallel platforms where a parallel program can be executed, it is the programmer's job to optimize the code aimed to the capabilities offered by the distributed systems. Often software based parallel architectures like SVM systems are not considered when developing these workloads. In addition, the optimizations aimed at hardware systems produce performance losses in SVM systems because they reduce the granularity of sharing of the workload and increase the frequency of sharing.

Several research papers in the open literature propose workload modifications in hardware based parallel systems to improve performance in SVM systems without performance losses [IFT96][JIA97][ZHO97]. Some of the work presented in this dissertation has its sources in these studies but it mainly differs from those works in the selection of the characterization indexes, which have been carefully chosen to be useful for tuning the SVM performance protocols.

From the parallel workload study, we can concisely state both the grain of sharing unit and the capability of synchronization assumed by the workload programmer, concluding similar remarks to previous studies although with deeper details. In general, we found that most of the workloads are programmed aiming to parallel systems that support both fine-grained sharing and synchronization. This is the source of a high percentage of asynchronous communication that occurs when the same workloads are executed in SVM systems. We also are able to parameterize the effects produced by the grain of sharing and the synchronization, such as the critical section dilation and the sharing pattern transformation. In this sense, the studies done have been very valuable for the design of the protocols proposed in this dissertation.

6.2. Developed Protocols

The reasons that induce us to propose the protocols explained in Chapter 5 spring from the previous workload characterization study described in Chapter 4 and in Appendix A. In particular, from the fact that sharing is mostly performed at small sizes and that a high percentage of the sharing shows spatial localities.

The protocols HLRC-DU and HLRC-CU exploit these facts by sending small updates instead of invalidations. This benefits the performance by reducing asynchronous communication and contention at home nodes because the page remains valid. However, there is a tradeoff with the traffic they induce because the network can become a bottleneck. Therefore, we introduced a threshold value to limit the size of updates in order to find a good compromise between the number of write updates sent and network traffic.

We found that these updates improve performance except when the network becomes a bottleneck. Small updates have a negligible impact on the network bandwidth. Because there are many synchronizations performed after small writings (as shown in Chapter 4), the number of asynchronous page requests is substantially reduced, and consequently the speedup can be improved. This is not the case when using larger updates because they appear less frequently, thus there are fewer asynchronous page request candidate to eliminate. In addition, larger updates consume more network bandwidth.

The workloads that benefit from this strategy are known in the literature as irregular. This kind of applications often use task queues synchronized by active waiting semaphores, which are natively supported by the hardware systems. When synchronizing, only small amounts of data are invalidated or updated in hardware systems while in SVM systems one or more pages of data in several nodes can be affected. Performing small updates reverses this effect. On the other hand, the regular workloads distribute parallel work by defining static partitions for the data. In general, SVM systems achieve good performance with this kind of applications if the problem size is large enough. If the application offers low performance, it is often possible to tune the data distribution or the program code to gain performance (for example, the continuous version of LU). In these workloads the proposed protocols neither increase nor reduce performance except when using high thresholds due to the large utilizations they produce.

6.3. Simulation Environment

The LIDE simulation environment made in the initial phase of this work is a helpful tool to simulate and explore the behavior of existing protocols, or new protocols proposals, and for making performance evaluation studies. More precisely, the environment allows in a flexible way to:

- Check the impact on performance of different configurations while varying the physical conditions of the system and the network.
- Easily verify protocols and debug errors.
- Better understand the drawbacks and gains that the protocols achieve.

All these features can be reached because of the tool allows us total control of the execution of the parallel workload and the simulation of the memory accesses.

6.4. Future Lines of Research

Regarding to the practical aspects, we plan to develop extensions in current operating systems in order to support the designed protocols. Because of their software nature, once they have been designed and tested, it should be easy to implement them in real architectures.

An open area of research with regard to the workload characterization is the study of the Inter Reference Gap (IRG) [PHA95] sequences (in the context of SVM systems) that provide a framework for studying not only the frequencies and sizes of the shares but also for studying the correlation between the accesses made by different processes to different words.

We hope that the results of the characterization help us to design new versions of the protocols presented in this dissertation that could adaptively invalidate or update in function of the predicted accesses of other processes. In this way we could also look for solutions that help us to reduce the number of updates in some workloads and so reducing network utilization.

Finally, the contention in the nodes that serve the pages and their synchronization are also an important source of performance losses in SVM systems. In the context of home-based protocols used in this dissertation the main problem is that the HLRC protocol tries to reduce asynchronous communication by concentrating the asynchronous communication in the home node of the page; but due to only one home node exists per page, it may become a contention point. We are planning to explore new protocols that migrate or replicate the home nodes in an intelligent way, so spreading the traffic and avoiding that contention points will arise.

6.5. Publications Related with This Dissertation

Preliminary versions of fragments of this work have been published in proceedings of several national and international conferences:

- S. Petit, "LIDE: Un Entorno de Simulación para Sistemas de Memoria Virtual Compartida," *Actas de las X Jornadas de Paralelismo*, Murcia, Spain, September 1999.

- S. Petit, J. A. Gil, J. Sahuquillo, and A. Pont, LIDE: A Simulation Environment for Shared Virtual Memory Systems, September 2000 issue of the ACM Computer News, ISSN 0163-5964, Vol. 28, No. 4.

- S. Petit, J. Sahuquillo, and A. Pont, "Performance Evaluation of Consistency Models using a New Simulation Environment for SVM systems" *Proceedings of the 2nd ACM International Workshop on Software Distributed Shared Memory, (in conjunction with the International Conference on Supercomputing)*, Santa Fe, New Mexico, USA, May 2000.

- S. Petit, "Evaluación de Modelos de Consistencia mediante un Nuevo Entorno de Simulación," *Actas de las XI Jornadas de Paralelismo*, Granada, Spain, September 2000.

- S. Petit, J. Sahuquillo, J.A. Donet, and A. Pont, “Detecting Spatial Locality to Improve SVM Consistency Protocols,” *Proceedings of the 2nd International Conference on Advances in Infrastructure for Electronic Business, Science and Education in Internet*, L´Aquila, Italy, August 2001.

- S. Petit, J. Sahuquillo, and A. Pont, “About the Sensitivity of the HLRC-DU Protocol to the Written Area Size and Page Size,” *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, Arizona, USA, November 2001.

- S. Petit, J. Sahuquillo, y A. Pont, “Reducing Multiple Writer Overhead in Memory Consistency Protocols for SVM Systems,” *Actas de las XII Jornadas de Paralelismo*, Valencia, Spain, September 2001

- S. Petit, J. Sahuquillo, and A. Pont, “Characterizing Parallel Workloads to Reduce Multiple Writer Overhead in Shared Virtual Memory Systems*,” Proceedings of the 10th IEEE Euromicro Workshop on Parallel, Distributed and Network-based Processing*, Gran Canaria, Spain, January 2002.

- S. Petit, J. Sahuquillo, y A. Pont, “Accelerating Consistency Protocols through Write Updates,” *Actas de las XIII Jornadas de Paralelismo*, Lleida, Spain, September 2002.

Another important piece of work related with this dissertation is already pending to be published, and has been recently submitted to an important conference in the area.

- S. Petit, J. Sahuquillo, A. Pont, and D. Kaeli, “Temporal Characterization of Parallel Workloads targeting SVM Systems”.

Appendix A

Preliminary Workload Studies

Performances of any kind of computer systems are based on the characteristics of the workload running on them; i.e., some recent cache schemes [SAH00] use two independent cache organizations to exploit the kind of locality that data exhibits (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 to reduce that overhead.

Research in SVM systems introduces the techniques discussed in Chapter 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 could asynchronously ask 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.

Semaphore synchronization of 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 chapter 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 this, we instrument the parallel workload to trap the synchronization and write operations issued. Below, the instrumentation technique and target parallel workload are detailed.

A.1. Experimental Framework

The tool used to instrument the workload is a part of LIMES [MAG97]. 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 [WOO95], the measurements are taken just after the parallel processes are created. Table 12 shows the problem size used for each benchmark, as well as the number of semaphores and semaphore acquisitions obtained under such problem size. Every benchmark was executed while taking into account 32 processes.

Benchmark	Problem Size	Total Semaphores	Total Semaphore Acquires
Barnes	2K particles	78	4579
Cholesky	Tk 14.0	64	21559
FFT	32K points	0	0
FMM	2K particles	22	4449
LU	512x512 points	0	0
Ocean	66x66 ocean	2	3648
Radix	128K integer	32	2048
Water	512 molecules	516	17728

Table 12 – Benchmark 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 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).

A.2. 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 in characterizing the workload in SVM systems. This will help us to propose new ideas for avoiding the large amount of the overhead produced in these architectures due to consistency maintenance.

A.2.1. Serial and Concurrent Data Sharing

Two assumptions may imply that parallel workloads can limit the use of multiple writer protocols by synchronizing using semaphores. Firstly*, 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, in the same way as caches base their effectiveness on data localities. Other more relaxed consistency models such as Entry [BER93] and Scope [IFT96b] are based on this characteristic of the workload code, although they force the programmer to define this relation in the source code. Secondly, *writes to shared data have also a high probability of happening in protected sections*. This assumption is reasonable too, although exceptions (such as 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.

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 a process wrote to the section protected by the same semaphore. If so, as semaphores serialize writers there is no need for a multiple writer coherence mechanism. We refer to those 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 34 shows the pseudocode of the filter algorithm and Figure 35 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 35 because they have no semaphore (as shown in Table 1).

Figure 35 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 concurrent shares. That is shown clearly in Table 13, which summarizes these percentages. As cold shares represent accesses to unwritten data words written during the cold start, they have been removed because they are not useful for our proposals.

```
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 processor 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 If
       End For
End
```

Figure 34 – Software filter to classify shared data accesses inside the semaphores

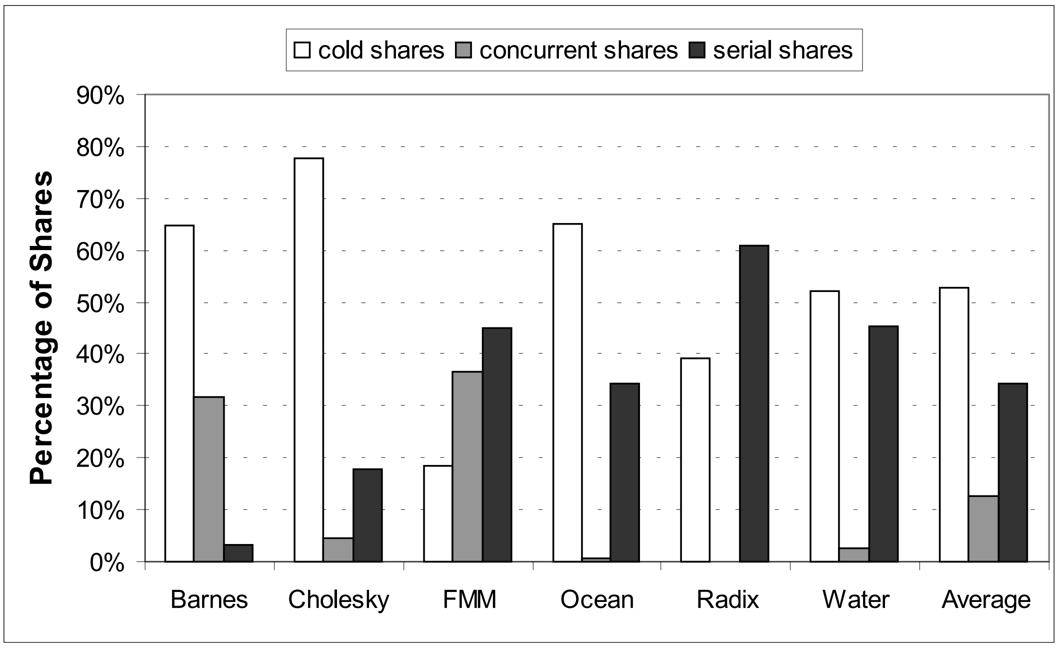


Figure 35 – Percentage of shares

Benchmark	Concurrent	Serial
Barnes	91%	9%
Cholesky	20%	80%
FMM	45%	55%
Ocean	2%	98%
Radix	0%	100%
Water	5%	95%
Average	*27%*	*73%*

Table 13 – Concurrent versus serial shares

A.2.2. Writing Localities

Write operations will probably be performed 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.

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 36 presents the software filter that takes account of write localities. 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 processor) Then
                     Switch (locality of writes of the other processor)
                            The processor wrote the full page: FULL++
                            The processor wrote just a single word: SINGLE++
                            The processor only wrote in continuous addresses: CONTINUOUS++
                            The processor wrote in discontinuous addresses: DISCONTINUOUS++
                     End Switch
                     Reset statistics of writes of the other processor in the page
              End If
       End For
End
```

Figure 36 – Software filter to classify page writes

Figure 37 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 the page size grows the percentage of discontinuous page writes also grows stabilizing at 4KB. The results between 4KB and 8KB differ slightly because the SPLASH-2 benchmark suite uses a page size parameter to distribute data among nodes [WOO95]. So, this affects the data distribution algorithm, producing similar results when varying the page size across that range.

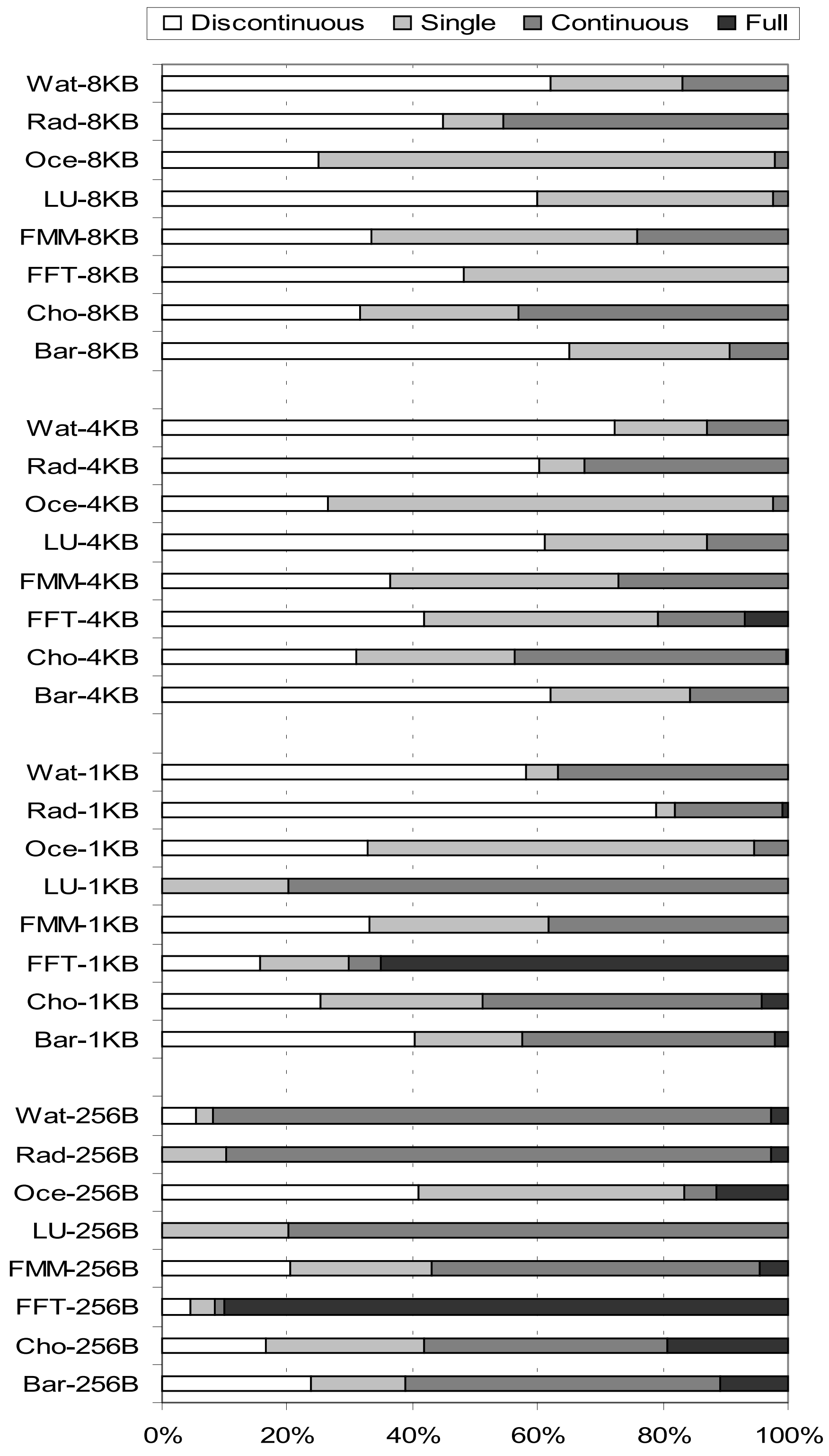


Figure 37 – Percentage of Write Patterns

As expected, due to false sharing, the percentage of full pages is smaller when the page size is larger. However, independently of the page size used there is a large percentage of continuous and single patterns. For typical page sizes (i.e. 4KB) it ranges from about 30% (Water) to 75% (Ocean). To check the commonest sizes of continuous patterns, Table 14 represents the distribution of the sizes of each continuous pattern generated by all the benchmarks using an 8KB page size. Most write chunk areas smaller than 256 bytes and just 7.1% of chunks are larger.

Chunk Size Area	Percentage (%)
[0, 256B [	92.90
[256B, 1KB [	5.22
[1KB, 4KB [	1.20
[4KB, 8KB]	0.69

Table 14 – Chunk size distribution

A.3. 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.

A.3.1. Reducing Multiple Writer Overhead

To carry out all the suggestions detailed 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 to a page. This situation happens when different parts of the page are written on different semaphores by several nodes. Figure 38 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 and j_1 have just passed their critical sections and generated write notices.

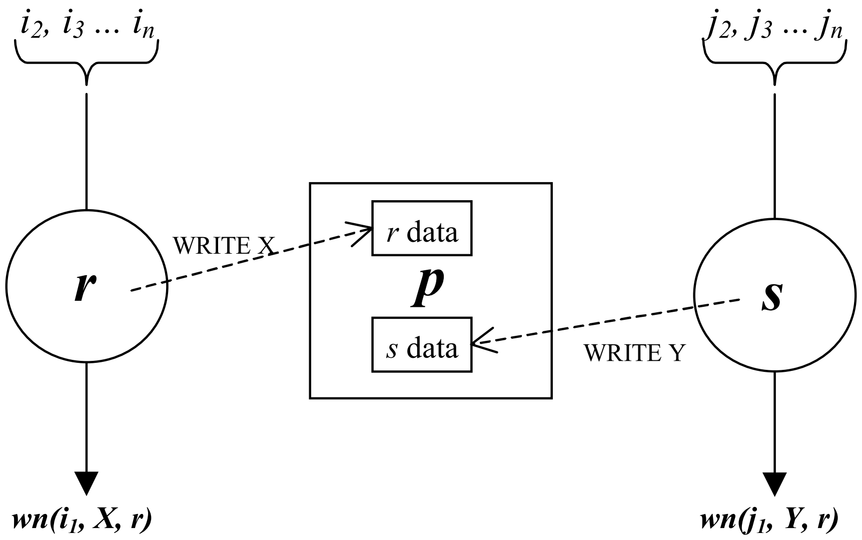


Figure 38 – Semaphores serializing the writers to a page. Legend: *wn(i, a, r)* represents a write notice from node *i* to address *a* in the section protected by the semaphore *r*

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 using 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 have to access the requested page in serial order.

This technique improves LRC-based protocols [AMZ99] 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 these purposes can be found in [STE00] and [BIL98].

We think HLRC-based protocols could also benefit from this situation by spreading contention among 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.

The LRC consistency model enables concurrent shares so the ideal case explained above may not occur every time the workload is running; thus, we must consider a mechanism to consider 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 an 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.

To have an overall perspective of how an increase in the number of homes would benefit the system performance, we summarize in Table 15 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; and that means that they could benefit from a multiple home protocol as discussed above.

We think that the discussed technique could offer a potentially higher advantage than home migrations as proposed by Stet et al. [STE00], although they can be applied together, because in [STE00] just one home migrates (our *main home*).

Benchmark	Page Size		
	1KB	4KB	8KB
	writers/page	writers/page	writers/page
Barnes	4.8	5.6	5.3
Cholesky	2.0	2.7	3.5
FFT	1.0	1.1	1.1
FMM	3.5	5.1	5.5
LU	8.0	8.0	9.0
Ocean	1.1	1.3	1.6
Radix	18.9	23.6	21.8
Water	3.8	3.8	3.5

Table 15 – Mean number of writers per page varying the page size

A.3.2. 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 to 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). However, the double page faults represent a tradeoff to be taken into account.

A.4. 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 chapter focuses on how the workload sharing patterns behave and it is intended to help 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 chapter concentrates the sharing patterns of parallel workloads from experimental traces.

Firstly, we examine 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.

Secondly, when the overhead of multiple writer capabilities cannot be avoided, it is still possible to optimize protocols taking advantage of 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. Those small areas can be directly updated, thus reducing diff memory consumption as well as asynchronous communication. Furthermore, early updates could also reduce home contention.

To adapt the protocols to workload behavior, some software and hardware implementations are also discussed. Most of the implementations would not only improve protocol performance but they would add little complexity.

References

[ADV93] S. V. Adve, Designing Memory Consistency Models for Shared Memory Multiprocessors, Ph.D. Thesis, University of Wisconsin, December 1993.

[AMZ99] C. Amza, A. L. Cox, S. Dwarkadas, L. Jin, K. Rajamani, and W. Zwaenepoel, Adaptative Protocols for Software Distributed Shared Memory, Proceedings of the IEEE, vol. 87(3), pp. 467-475, March 1999.

[AND95] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team, A Case for Networks of Workstations: NOW, IEEE Micro, pp. 54-64, February 1995.

[BER93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," *Proceedings of the 38th International Computer Conference*, February 1993.

[BIA96] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim, "Hiding Communication Latency and Coherence Overhead in Software DSMs," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[BIL97] A. Bilas and J. P. Singh, "The Effects of Communication Parameters on End Performance of Shared Virtual Memory Clusters," in *Proceedings of the Supercomputing '97 Conference*, November 1997.

[BIL98] A. Bilas, C. Liao, and J. P. Singh, "Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems," *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[BLU98] M. A. Blumrich, R. D. Alpert, A. Bilas, Y. Chen, D. W. Clark, S. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner, "Design Choices in the SHRIMP System: An Empirical Study,"' in *Proceedings of the 25th Annual Symposium on Computer Architecture*, June 1998.

[CAR91] J. B. Carter, J. K. Bennet, and W. Zwaenepoel, "Implementation and Performance of Munin," *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[CUL99] D. E. Culler, J. Pal Singh, and A. Gupta, Parallel Computer Architecture: A Hardware-Software Approach, Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[DAV91] H. Davis, S. R. Goldschmidt, and J. Henessy, "Multiprocessor Simulation and Tracing using Tango," *Proceedings of the 1991 Conference on Parallel Processing*, August 1991.

[DOB93] W. Dobosiewicz and P. Gburzynski, "SMURPH: An Object-Oriented Simulator for Communication Networks and Protocols," *Proceedings of the 93 International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, January 1993.

[GHA90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Henessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual Symposium on Computer Architecture*, May 1990.

[HAN98] P. B. Hansen, An Evaluation of the Message-Passing Interface, ACM Sigplan Notices, vol. 33-3, pp. 65-72, March 1998.

[IFT96] L. Iftode, J. P. Singh, and K. Li, "Understanding Application Performance on Shared Virtual Memory," *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[IFT96b] L. Iftode, J. P. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, June 1996.

[IFT99] L. Iftode and J. P. Singh, Shared Virtual Memory: Progress and Challenges, Proceedings of the IEEE, vol. 87(3), March 1999.

[JIA97] D. Jiang, H. Shan, and J. P. Singh, "Application Restructuring and Performance Portability across Shared Virtual Memory and Hardware-Coherent Multiprocessors," *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming*, June 1997.

[KAR96] M. Karlsson and P. Stenstrom, "Performance Evaluation of Cluster-Based Multiprocessor built from ATM Switches and Bus-Based Multiprocessor Servers," *Proceedings of the 2nd Symposium on High-Performance Computer Architecture*, February 1996.

[KEL94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proceedings of the Winter 1994 USENIX Conference*, January 1994.

[KEL95] P. Keleher, Lazy Release Consistency for Distributed Shared Memory, Ph.D. Thesis, Rice University, January 1995.

[LI_86] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the 5th Annual Symposium on Principles of Distributed Computing*, August 1986.

[LI_88] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[MAG97] D. Magdic, LIMES: A Multiprocessor Simulation Environment, TCCA Newsletter, pp. 68-71, March 1997.

[MAG97b] D. Magdic, LIMES: An Execution-driven Multiprocessor Simulation Tool for the i486+-based PCs User's Guide, School of Electrical Engineering, Department of Computer Engineering, University of Belgrade, Yugoslavia.

[PET00] S. Petit, J. A. Gil, J. Sahuquillo, and A. Pont, LIDE: A Simulation Environment for Shared Virtual Memory Systems, September 2000 issue of the ACM Computer News, Vol. 28, No. 4.

[PET01] S. Petit, J. Sahuquillo, J.A. Donet, and A. Pont, "Detecting Spatial Locality to Improve SVM Consistency Protocols," Proceedings of the Second International Conference on Advances in Infrastructure for Electronic Business, Science and Education in Internet, August 2001.

[PET01b] S. Petit, J. Sahuquillo, and A. Pont, "About the Sensitivity of the HLRC-DU Protocol to the Written Area Size and Page Size," *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, November 2001.

[PET02] S. Petit, J. Sahuquillo, and A. Pont, "Characterizing Parallel Workloads to Reduce Multiple Writer Overhead in Shared Virtual Memory Systems," *Proceedings of the 10th IEEE Euromicro Workshop on Parallel, Distributed and Network-based Processing*, January 2002.

[PET03] S. Petit, J. Sahuquillo, A. Pont, and D. Kaeli, "Temporal Characterization of Parallel Workloads Targeting SVM Systems", submitted.

[PHA95] V. Phalke, B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proceedings of SIGMETRICS '95*, 1995.

[PRO98] J. Protic and V. Milutinovic, Distributed Shared Memory Concepts and Systems, IEEE Computer Society Press, Los Alamitos, California, 1998.

[SAH00] J. Sahuquillo and A. Pont, Splitting the Data Cache: a Survey, July-September Special Issue of the IEEE Concurrency, pp. 30-35, 2000.

[SAM98] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, "Home-based SVM protocols for SMP clusters: Design and performance," *Proceedings of the* 4th *Symposium on High-Performance Computer Architecture*, February 1998.

[SPE98] E. Speight and J. Bennett, "Using Multicast and Multithreading to Reduce Communication in Software DSM Systems," *Proceedings of the 4th Symposium on High-Performance Computer Architecture*", February 1998.

[STE97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.

[STE00] R. Stets, S. Dwarkadas, L. Komothanassis, U. Rencuzogullari, and M. L. Scott, "The Effect of Network Total Order, Broadcast and Remote-Write Capability on Network-Based Shared Memory Computing*," Proceedings of the 6th Symposium on High-Performance Computer Architecture*, January 2000.

[SUN90] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, vol .2, num. 4, pp 315-339, December, 1990.

[SWA98] A. M. Swanson, L. Stoller, and J.B. Carter, "Making Distributed Shared Memory Simple, Yet Efficient," *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 1998.

[TOR94] J. Torrellas, M. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," IEEE Transactions on Computers, vol. 43, n. 6, pp. 651-663, June 1994.

[WOO95] S. Woo, M. Ohara, E. Torrie, J. Pal Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, June 1995.

[ZHO96] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[ZHO97] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, L. Schoinas, M. D. Hill, and D. A. Wood, "Relaxed Consistency and Coherence Granurality in DSM Systems: A Performance Evaluation," *Proceedings of 6th Symposium on Principles and Practice of Parallel Programming*, June 1997.